Topics in combinatorial pattern matching

Vildhøj, Hjalte Wedel

Publication date:
2015

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
TOPICS IN
COMBINATORIAL
PATTERN MATCHING

Hjalte Wedel Vildhøj
This doctoral dissertation was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in partial fulfilment of the requirements for acquiring a doctoral degree. The dissertation presents selected results within the general topic of theoretical computer science and more specifically combinatorial pattern matching. All included results were obtained and published in peer-reviewed conference proceedings or journals during my enrollment as a PhD student from September 2011 to September 2014.

Acknowledgements  I am grateful to my advisors Inge Li Gørtz and Philip Bille who have provided excellent guidance and support during my studies and have introduced me to countless of interesting problems, papers and people. I also thank my colleagues, office mates and friends Frederik Rye Skjoldjensen, Patrick Hagge Cording, and Søren Vind for making the past three years anything but lonely. I am further grateful to our section head Paul Fischer and the rest of our section for creating a fantastic atmosphere for research. I also owe my thanks to Gadi Landau and Oren Weimann for hosting me during my external stay at Haifa University in Israel, and for broadening my mind about their beautiful country. I also thank Raphaël Clifford and Benjamin Sach for hosting me during a memorable research visit at Bristol University. Moreover, I wish to thank all my co-authors Philip Bille, Søren Bøg, Patrick Hagge Cording, Johannes Fischer, Inge Li Gørtz, Tomasz Kociumaka, Tsvi Kopelowitz, Benjamin Sach, Tatiana Starikovskaya, Morten Stöckel, Søren Vind and David Kofoed Wind. Last but not least, I thank Maja for her love and support.

Hjalte Wedel Vildhøj
Copenhagen, September 2014
This dissertation studies problems in the general theme of combinatorial pattern matching. More specifically, we study the following topics:

**Longest Common Extensions.** We revisit the longest common extension (LCE) problem, that is, preprocess a string $T$ into a compact data structure that supports fast LCE queries. An LCE query takes a pair $(i, j)$ of indices in $T$ and returns the length of the longest common prefix of the suffixes of $T$ starting at positions $i$ and $j$. Such queries are also commonly known as longest common prefix (LCP) queries. We study the time-space trade-offs for the problem, that is, the space used for the data structure vs. the worst-case time for answering an LCE query. Let $n$ be the length of $T$. Given a parameter $\tau$, $1 \leq \tau \leq n$, we show how to achieve either $O(n/\sqrt{\tau})$ space and $O(\tau)$ query time, or $O(n/\tau)$ space and $O(\tau \log(|\text{LCE}(i, j)|/\tau))$ query time, where $|\text{LCE}(i, j)|$ denotes the length of the LCE returned by the query. These bounds provide the first smooth trade-offs for the LCE problem and almost match the previously known bounds at the extremes when $\tau = 1$ or $\tau = n$. We apply the result to obtain improved bounds for several applications where the LCE problem is the computational bottleneck, including approximate string matching and computing palindromes. We also present an efficient technique to reduce LCE queries on two strings to one string. Finally, we give a lower bound on the time-space product for LCE data structures in the non-uniform cell probe model showing that our second trade-off is nearly optimal.

**Fingerprints in Compressed Strings.** The Karp-Rabin fingerprint of a string is a type of hash value that due to its strong properties has been used in many string algorithms. We show how to construct a data structure for a string $S$ of size $N$ compressed by a context-free grammar of size $n$ that supports fingerprint queries. That is, given indices $i$ and $j$, the answer to a query is the
fingerprint of the substring $S[i, j]$. We present the first $O(n)$ space data structures that answer fingerprint queries without decompressing any characters. For Straight Line Programs (SLP) we get $O(\log N)$ query time, and for Linear SLPs (an SLP derivative that captures LZ78 compression and its variations) we get $O(\log \log N)$ query time. Hence, our data structures has the same time and space complexity as for random access in SLPs. We utilize the fingerprint data structures to solve the longest common extension problem in query time $O(\log N \log \ell)$ and $O(\log \ell \log \log \ell + \log \log N)$ for SLPs and Linear SLPs, respectively. Here, $\ell = |\text{LCE}(i, j)|$ denotes the length of the LCE.

**Sparse Text Indexing.** We present efficient algorithms for constructing sparse suffix trees, sparse suffix arrays and sparse positions heaps for arbitrary positions of a text $T$ of length $n$ while using only $O(b)$ words of space during the construction. Our main contribution is to show that the sparse suffix tree (and array) can be constructed in $O(n \log^2 b)$ time. To achieve this we develop a technique, that allows to efficiently answer $b$ longest common prefix queries on suffixes of $T$, using only $O(b)$ space. Our first solution is Monte-Carlo and outputs the correct tree with high probability. We then give a Las-Vegas algorithm which also uses $O(b)$ space and runs in the same time bounds with high probability when $b = O(\sqrt{n})$. Furthermore, additional tradeoffs between the space usage and the construction time for the Monte-Carlo algorithm are given. Finally, we show that at the expense of slower pattern queries, it is possible to construct sparse position heaps in $O(n + b \log b)$ time and $O(b)$ space.

**The Longest Common Substring Problem.** Given $m$ documents of total length $n$, we consider the problem of finding a longest string common to at least $d \geq 2$ of the documents. This problem is known as the longest common substring (LCS) problem and has a classic $O(n)$ space and $O(n)$ time solution (Weiner [FOCS’73], Hui [CPM’92]). However, the use of linear space is impractical in many applications. We show several time-space trade-offs for this problem. Our main result is that for any trade-off parameter $1 \leq \tau \leq n$, the LCS problem can be solved in $O(\tau)$ space and $O(n^2/\tau)$ time, thus providing the first smooth deterministic time-space trade-off from constant to linear space. The result uses a new and very simple algorithm, which computes a $\tau$-additive approximation to the LCS in $O(n^2/\tau)$ time and $O(1)$ space. We also show a time-space trade-off lower bound for deterministic branching programs, which implies that any deterministic RAM algorithm solving the LCS problem on documents from a sufficiently large alphabet in $O(\tau)$ space must use $\Omega(n \sqrt{\log(n/(\tau \log n))/ \log \log(n/(\tau \log n))})$ time.
Structural Properties of Suffix Trees. We study structural and combinatorial properties of suffix trees. Given an unlabeled tree $T$ on $n$ nodes and suffix links of its internal nodes, we ask the question “Is $T$ a suffix tree?”, i.e., is there a string $S$ whose suffix tree has the same topological structure as $T$? We place no restrictions on $S$, in particular we do not require that $S$ ends with a unique symbol. This corresponds to considering the more general definition of implicit or extended suffix trees. Such general suffix trees have many applications and are for example needed to allow efficient updates when suffix trees are built online. We prove that $T$ is a suffix tree if and only if it is realized by a string $S$ of length $n - 1$, and we give a linear-time algorithm for inferring $S$ when the first letter on each edge is known.
Denne afhandling studerer problemer inden for det generelle område kombinatorisk mønstergenkendelse. Vi studerer følgende emner:

Længste fælles præfiks. Vi vender tilbage til længste-fælles-præfiks-problemet, det vil sige præprocesser en streng $T$ til en kompakt datastruktur, der understøtter hurtige LCE-forespørgsler. En LCE-forespørgsel tager et par $(i, j)$ af positioner i $T$ og returnerer det længste fælles præfiks af de to suffikser, der starter på position $i$ og $j$ i $T$. Sådanne forespørgsler er også kendt som LCPforespørgsler. Vi studerer mulige afvejninger af tid og plads for problemet – det vil sige den plads, som datastrukturen anvender versus den tid, den skal bruge til at svare på en LCE-forespørgsel. Lad $n$ betegne længden af $T$. Vi viser at givet en parameter $\tau$, $1 \leq \tau \leq n$, så kan problemet løses i enten $O(n/\sqrt{\tau})$ plads og $O(\tau)$ forespørgselstid eller $O(n/\tau)$ plads og $O(\tau \log(||LCE(i, j)||/\tau))$ forespørgselstid, hvor $||LCE(i, j)||$ betegner længden af det længste fælles præfiks, som forespørgslen returnerer. Disse grænser giver de første jævne afvejninger for LCE-problemet og svarer næsten til de kendte grænser ved de to ekstremiteter $\tau = 1$ eller $\tau = n$. Vi bruger dette resultat til at forbedre grænserne for adskillige anvendelser, hvor LCE-forespørgsler er den beregningsmæssige flaskehals, inklusiv approksimativ mønstergenkendelse og beregning af palindromer. Vi viser også en effektiv måde at reducere LCE-forespørgsler på to strenger til en streng. Endelig giver vi en nedre grænse for tidspladsproduktet af LCE-datastrukturer i den ikke-uniforme cell-probe model, der viser, at vores sidste algoritme næsten er optimal.

Fingeraftryk i komprimerede streng. Karp-Rabin-fingeraftrykket af en streng er en slags hashværdi, der på grund af sine stærke egenskaber har været anvendt i mange strengalgoritmer. Vi viser, hvordan man konstruerer en datastruktur
for en streng $S$ af længde $N$, der er komprimeret af en kontekstfri grammatik $G$ af størrelse $N$, der kan svare på fingeraftryksforespørgsler. For positionerne $i$ og $j$ er svaret på denne forespørgsel fingeraftrykket af delstrengen $S[i,j]$. Vi giver den første datastruktur, der bruger $O(n)$ plads og svarer på fingeraftryksforespørgsler uden at dekomprimere nogen symboler. For Straight Line Programs (SLP) opnår vi $O(\log N)$ forespørgselstid, og for lineære SLP’er (en SLP-afledning, der omfatter LZ78 kompression og dens varianter), opnår vi $O(\log \log N)$ forespørgselstid. Således har vores datastrukturer samme tids- og pladskompleksitet som for tilfældig adressering i SLP’er. Vi anvender fingeraftryksdatastrukturen til at løse længste-fælles-præfiks-problemet med en forespørgselstid på henholdsvis $O(\log N \log \ell)$ og $O(\log \ell \log \log \ell + \log \log N)$ for SLP’er og lineære SLP’er. Her betegner $\ell$ længden af det længste fælles præfiks.

Tynd tekstindeksering. Vi præsenterer de første effektive algoritmer, der konstruerer tynde suffikstræer, tynde suffikstabeller og tynde positionsdynger for arbitrære positioner i en tekst $T$ af længde $n$, alt imens der under hele konstruktionen kun anvendes $O(b)$ plads. Vores største bidrag er at vise, at det tynde suffikstræ (samt tabel) kan konstrueres i $O(n \log^2 b)$ tid. For at opnå dette udvikler vi en ny teknik, der effektivt kan svare på $b$ LCE-forespørgsler på $T$, mens der kun bruges $O(b)$ plads. Vores første løsning er Monte-Carlo og returnerer med stor sandsynlighed det korrekte træ. Vi giver derefter en Las-Vegas-algoritme, der også bruger $O(b)$ plads og med stor sandsynlighed har samme tidsgrænse, så længde $b = O(\sqrt{n})$. Endvidere viser vi nogle yderligere tidspladsafvejninger for Monte-Carlo-algoritmen, og til sidst viser vi, hvordan det med langsommere mønsterforespørgsler er muligt at konstruere tynde positionsdynger i $O(n + b \log b)$ tid og $O(b)$ plads.

Længste fælles delstreng. Vi studerer følgende problem: Givet $m$ dokumenter af samlet længde $n$, find den længste fælles delstreng, der optræder i mindst $d \geq 2$ af dokumenterne. Dette problem bliver kaldt længste-fælles-delstrengsproblemet (LCS-problemet) og har en klassisk $O(n)$ plads og $O(n)$ tids løsning (Weiner [FOCS’73], Hui [CPM’92]). Imidlertid kan forbruget af lineær plads være upraktisk i mange anvendelser. Vi viser flere tidspladsafvejninger for dette problem. Vores hovedbidrag er, at for en vilkårlig afvejningsparameter $1 \leq \tau \leq n$, så kan LCS-problemet blive løst i $O(\tau)$ plads og $O(n^2/\tau)$ tid, hvilket giver den første jævne deterministiske tidspolasfvejning helt fra konstant til lineær plads. Resultatet gør brug af en ny og meget simpel algoritme, der beregner en $\tau$-additiv approksimation til den længste fælles delstreng i $O(n^2/\tau)$
tid og $O(1)$ plads. Vi viser også en nedre grænse for tidspladsafvejninger, der medfører, at alle deterministiske RAM-algoritmer, der løser LCS-problemet på dokumenter fra et tilstrækkeligt stor alfabet med $O(\tau)$ plads, nødvendigvis må anvende $\Omega(n\sqrt{\log(n/(\tau \log n))}/\log \log(n/(\tau \log n))$ tid.

**Strukturelle egenskaber af suffikstræer.** Vi studerer strukturelle og kombinatoriske egenskaber af suffikstræer. For et umærket træ $T$ på $n$ knuder med suffikspegere på dets interne knuder stiller vi spørgsmålet: “Er $T$ et suffikstræ?”, det vil sige, findes der en streng, hvis suffikstræ har den samme topologiske struktur som $T$? Vi stiller ingen krav til strengen $S$, og specifikt antager vi ikke, at $S$ ender med et unikt symbol. Dette svarer til at betragte den mere generelle definition af implicitte eller udvidede suffikstræer. Disse generelle suffikstræer har mange anvendelser og er for eksempel nødvendige for at opnå hurtige opdateringer, når suffikstræer bygges online. Vi beviser, at $T$ er et suffikstræ, hvis og kun hvis det kan realiseres af en streng af længde $n – 1$, og vi giver en algoritme, der i lineær tid kan udlede $S$, når det første symbol på hver kant kendes.
Contents

Preface i
Abstract iii
Danish Abstract vii
Contents xi

1 Introduction 1
  1.1 Overview and Outline 2
    1.1.1 Additional Publications 3
  1.2 Model of Computation 4
  1.3 Fundamental Techniques 4
    1.3.1 Karp-Rabin Fingerprints 4
    1.3.2 Suffix Trees 7
  1.4 On Chapter 2: Time-Space Trade-Offs for Longest Common Extensions 10
    1.4.1 Our Contributions 10
    1.4.2 Future Directions 11
  1.5 On Chapter 3: Fingerprints in Compressed Strings 11
    1.5.1 Our Contributions 12
    1.5.2 Future Directions 13
  1.6 On Chapter 4: Sparse Text Indexing in Small Space 14
    1.6.1 Our Contributions 15
    1.6.2 Future Directions 15
  1.7 On Chapters 5 & 6: The Longest Common Substring Problem 16
    1.7.1 Our Contributions 18
1.7.2 Future Directions ............................................. 19
1.8 On Chapter 7: A Suffix Tree or Not A Suffix Tree? ........ 20
  1.8.1 Our Contributions ...................................... 23
  1.8.2 Future Directions ...................................... 23

2 Time-Space Trade-Offs for Longest Common Extensions ...... 25
  2.1 Introduction ................................................. 26
    2.1.1 Our Results ........................................... 26
    2.1.2 Techniques ........................................... 27
    2.1.3 Applications ......................................... 28
  2.2 The Deterministic Data Structure .......................... 30
    2.2.1 Difference Covers .................................... 30
    2.2.2 The Data Structure .................................. 31
  2.3 The Monte-Carlo Data Structure ............................ 32
    2.3.1 Rabin-Karp fingerprints .............................. 32
    2.3.2 The Data Structure .................................. 32
  2.4 The Las-Vegas Data Structure ............................... 34
    2.4.1 The Algorithm ....................................... 34
  2.5 Longest Common Extensions on Two Strings ............... 37
    2.5.1 The Data Structure .................................. 37
  2.6 Lower Bound ................................................ 39
  2.7 Conclusions and Open Problems ............................. 40

3 Fingerprints in Compressed Strings ............................... 41
  3.1 Introduction ................................................. 42
    3.1.1 Longest common extension in compressed strings .... 44
  3.2 Preliminaries ................................................. 45
    3.2.1 Fingerprinting ....................................... 46
  3.3 Basic fingerprint queries in SLPs ........................... 47
  3.4 Faster fingerprints in SLPs ................................ 47
  3.5 Faster fingerprints in Linear SLPs .......................... 49
  3.6 Finger fingerprints in Linear SLPs .......................... 51
    3.6.1 Finger Predecessor ................................... 51
    3.6.2 Finger Fingerprints ................................. 53
  3.7 Longest Common Extensions in Compressed Strings ........ 53
    3.7.1 Computing Longest Common Extensions with Fingerprints 54
    3.7.2 Verifying the Fingerprint Function .................. 55

4 Sparse Text Indexing in Small Space ............................. 59
## CONTENTS

4.1 Introduction .............................................. 60
  4.1.1 Our Results .................................. 61
4.2 Preliminaries ........................................... 62
4.3 Batched LCP Queries ................................. 63
  4.3.1 The Algorithm ................................. 63
  4.3.2 Runtime and Correctness ..................... 64
4.4 Constructing the Sparse Suffix Tree ............... 66
4.5 Verifying the Sparse Suffix and LCP Arrays ....... 67
  4.5.1 Proof of Lemma 4.4 ......................... 68
4.6 Time-Space Tradeoffs for Batched LCP Queries ... 78
4.7 Sparse Position Heaps .................................. 79
  4.7.1 Position Heaps ................................ 79
  4.7.2 A Monte-Carlo Construction Algorithm ....... 81
4.8 Conclusions ............................................. 83

5 Time-Space Trade-Offs for the Longest Common Substring Problem 85
5.1 Introduction ............................................ 85
  5.1.1 Known Solutions ................................ 86
  5.1.2 Our Results .................................. 87
5.2 Preliminaries ........................................... 88
  5.2.1 Suffix Trees ................................... 88
  5.2.2 Difference Cover Sparse Suffix Arrays .......... 89
5.3 Longest Common Substring of Two Strings .......... 91
  5.3.1 A Solution for Long LCS ........................ 91
  5.3.2 A Solution for Short LCS .................... 93
5.4 Longest Common Substring of Multiple Strings .... 93
  5.4.1 A General Solution for Long LCS .............. 94
  5.4.2 A General Solution for Short LCS .......... 97
5.5 Open Problems .......................................... 98

6 Sublinear Space Algorithms for the Longest Common Substring Problem 101
6.1 Introduction ............................................ 102
  6.1.1 Our Results .................................. 102
6.2 Upper Bounds .......................................... 104
  6.2.1 Approximating LCS in Constant Space .......... 104
  6.2.2 An \(O(\tau)\)-Space and \(O(n^2/\tau)\)-Time Solution 105
  6.2.3 Large alphabets ................................ 111
6.3 A Time-Space Trade-Off Lower Bound ............... 111
6.4 Conclusions ............................................. 114
Over the years the expression *combinatorial pattern matching* has become a synonym for the field of theoretical computer science concerned with the study of combinatorial algorithms on strings and related structures. The term *combinatorial* emphasizes that these are algorithms based on mathematical properties and a deep understanding of the individual problems, in contrast to e.g., statistical or machine learning approaches, where general frameworks are often applied to model and solve the problems.

Work in this field began in the 1960s with the study of how to efficiently find all occurrences of a pattern string $P$ in a text $T$. The seminal work by Knuth, Morris, Pratt, Boyer, Moore, Weiner and many others through the 1970s, showed that this problem was solvable in linear time in the length of $P$ and $T$, and started several decades of research on algorithms and data structures for strings. Today the field has matured and we have come far in our understanding of its fundamental problems, but with the ever-increasing amount of digitized textual information, the study of efficient and theoretically well-founded algorithms for strings remains more relevant than ever.

In this dissertation we study several different, but fundamental problems on strings. A common theme in our work is the design of *time-space trade-offs*. In practical situations, space can be a more precious resource than time. Prominent examples include embedded devices with small amounts of writable memory, and data structures with a space requirement that exceeds the capacity of the fast memory. Under such circumstances we are interested in algorithms that allow their space complexity to be reduced at the cost of increasing their...
running time. From a purely theoretically perspective it is also intriguing why some problems allow time-space trade-offs and others do not.

To highlight a specific example from our work, consider the problem of finding the longest common substring of two documents consisting of a total of $n$ characters. Solving this problem efficiently is relevant in, for instance, plagiarism detection, and algorithms using $O(n)$ time and $O(n)$ space have been known since 1973. In our work we provide the first time-space trade-off, which implies that the problem can be solved in $O(n^{1+\varepsilon})$ time and $O(n^{1-\varepsilon})$ space for any choice of $\varepsilon \in [0, 1]$. For $\varepsilon = 0$ this captures the known linear time and space solution, and at the other extreme it provides an algorithm that solves the problem in constant space and quadratic time.

1.1 Overview and Outline

In addition to this general introduction, the dissertation consists of the following papers, which have all been written and published (or accepted for publication) during my PhD studies from 2011-2014.

Chapter 2 Time-Space Trade-Offs for Longest Common Extensions.

Chapter 3 Fingerprints in Compressed Strings.

Chapter 4 Sparse Suffix Tree Construction in Small Space.
Philip Bille, Johannes Fischer, Inge Li Gørtz, Tsvi Kopelowitz and Benjamin Sach and Hjalte Wedel Vildhøj. In proceedings of the 40th International Colloquium on Automata, Languages and Programming (ICALP 2013).

Chapter 5 Time-Space Trade-Offs for the Longest Common Substring Problem.

Chapter 6 Sublinear Space Algorithms for the Longest Common Substring Problem.
Tomasz Kociumaka, Tatiana Starikovskaya and Hjalte Wedel Vildhøj. In proceedings of the 22nd European Symposium on Algorithms (ESA 2014).

Chapter 7  A Suffix Tree or Not A Suffix Tree? Tatiana Starikovskaya and Hjalte Wedel Vildhøj. In proceedings of the 25th International Workshop on Combinatorial Algorithms (IWOCA 2014).

With minor exceptions, the papers appear in their original published form. As a consequence, notation, terminology and language are not always consistent across chapters. Some of the conference papers have been revised or extended and are currently in submission for a journal. The updated versions in this dissertation can therefore differ slightly from the published versions. The titles of the papers have not been changed in the revised versions, with the exception of Sparse Suffix Tree Construction in Small Space, which in this dissertation has the title Sparse Text Indexing in Small Space.

The remaining part of this chapter describes some important concepts common to many of the above papers, and in turn introduces the problems and contributions of each paper. The introduction to each paper establishes a broader context of our work and summarizes the most important results, techniques and ideas. We conclude the introduction of each paper by discussing problems left open by our work, very recent progress, and future directions of research.

1.1.1 Additional Publications

In addition to the above papers I have published the following papers during my PhD, which are not part of this dissertation.


The first two papers contain results partially obtained prior to my PhD studies and are thus omitted for formal reasons. The third paper was written during my PhD, but falls outside the theme of combinatorial pattern matching.
1.2 Model of Computation

Unless otherwise noted, our algorithms are designed for and analyzed in the word-RAM model [76]. This theoretical model of computation is an abstraction of any real world computing unit based on a processor and a random access memory. We briefly summarize the most important concepts of this model.

In the word-RAM model computation is performed on a random access machine with access to an unlimited number of memory registers, or cells, each capable of storing a \( w \)-bit integer, which we refer to as a word. The parameter \( w \) is called the word size, and we adopt the standard assumption that \( w \geq \log n \), where \( n \) is the number of cells required to store the input to our algorithm. Under this assumption, a word can hold a pointer (or address) to any input cell. Moreover, since all our algorithms and data structures use at most \( n^c \) cells for some constant \( c \), accessing any relevant cell can be done in \( O(1) \) time. The machine can perform basic arithmetic operations on words including addition, subtraction, multiplication, division, comparisons and standard bitwise operations in unit time, and these operations are allowed to compute addresses of other cells. The time used by an algorithm is the total number of unit operations it performs. The space used is the number of distinct cells the algorithm writes to during its operation. We assume that the input cells are available in read-only memory, and we emphasize that the input cells are not counted in the space used by the algorithm.

The input to many of our algorithms are strings, i.e., sequences of characters from some alphabet. We will generally assume that the size of the alphabet is at most polynomial in the length of the input, so any character can be stored in \( O(1) \) words.

1.3 Fundamental Techniques

In the following two sections we introduce the important concepts of Karp-Rabin fingerprints and suffix trees, which appear as core techniques in much of our work.

1.3.1 Karp-Rabin Fingerprints

The task of comparing substrings of some string \( T \) for equality is often a bottleneck in algorithms on strings. When a large number of substrings is to be tested for equality, comparing them character by character is very expensive. To speed up such algorithms we can use randomization and compare the hash
values $\phi(S_1)$ and $\phi(S_2)$ instead of comparing the substrings $S_1$ and $S_2$ directly. For this to work, we need a hash function $\phi$ that with high probability guarantees that $\phi(S_1) = \phi(S_2)$ if and only if $S_1 = S_2$. If $S_1 = S_2$ then we also have that $\phi(S_1) = \phi(S_2)$, but it can happen that $\phi(S_1) = \phi(S_2)$ even though $S_1 \neq S_2$. In this case we say that $\phi$ has a collision on $S_1$ and $S_2$.

That $\phi$ is collision free guarantees correctness of the computation. To actually gain a speedup when comparing many pairs of strings, we need to be able to compute $\phi(S_1)$ and $\phi(S_2)$ without examining the individual symbols in $S_1$ and $S_2$ one by one.

The Karp-Rabin fingerprinting function [97] provides both of these properties. It maps arbitrary strings to integer hash values, which we call fingerprints. More specifically, if we need to compare substrings of a string of length $n$ and we want $\phi$ to be collision free with probability at least $1 - 1/n^c$, for an arbitrary constant $c$, then the Karp-Rabin fingerprinting function $\phi$ can be defined as follows,

$$\phi(S) = \left( \sum_{i=1}^{|S|} S[i] \cdot b^{|S|} \right) \mod p ,$$

where $p$ is an arbitrary prime in the range $[2n^{c+4}, 4n^{c+4}]$, and $b$ is chosen uniformly at random in $\mathbb{Z}_p$. Note that the upper bound on $p$ ensures that a fingerprint fits in a constant number of machine words. The lower bound ensures the field $\mathbb{Z}_p(\mod p)$ is large enough that the probability of a collision for any fixed substring pair is upper bounded by $n/p$. Consequently, a union bound over all $\Theta(n^3)$ substring pairs shows that $\phi$ is collision free with probability at least $1 - 1/n^c$.

The fingerprint $\phi(S_1)$ can be computed in $O(|S_1|)$ time by standard modular exponentiation. However, the crucial property of the Karp-Rabin fingerprinting function is that fingerprints can be composed efficiently from other fingerprints in constant time, thereby eliminating the need to explicitly compute some fingerprints. As an example, suppose we have computed $\phi(S_1)$ and $\phi(S_2)$, then we can compute the fingerprint of the concatenation of $S_1$ and $S_2$, i.e., $\phi(S_1 S_2)$, in constant time as follows:

$$\phi(S_1 S_2) = \left( \phi(S_1) + \phi(S_2)b^{|S_1|} \right) \mod p .$$

To perform this computation in constant time, we need the number $b^{|S_1|} \mod p$, which we will assume is always stored together with the fingerprint $\phi(S_1)$.

\footnote{This probability bound follows easily from well-known properties of abstract algebra.}
Note that this assumption is without loss of generality, since in particular, we can obtain the exponent \( b^{|S_1|+|S_2|} \mod p \) in constant time from \( b^{|S_1|} \mod p \) and \( b^{|S_2|} \mod p \).

This important composition property of the Karp-Rabin fingerprint function is what allows us to speed up algorithms over the naive approach of comparing substrings character by character. As an example, consider the exact pattern matching problem, in which we want to report the occurrences of a pattern string \( P \) of length \( m \) in a text \( T \) of length \( n \). Karp and Rabin introduced fingerprints [97] as a mean to efficiently solve this problem. The idea is to compare the fingerprint \( \phi(P) \) to the fingerprints of all substrings of \( T \) of length \( m \). Evaluating the fingerprints of these \( n - m + 1 \) substrings of \( T \) by directly applying the definition of \( \phi \) would lead to an \( O(nm) \) time algorithm, similar to the naive approach. But by exploiting the composition property, we can obtain an \( O(n + m) \) time and constant space algorithm. The trick is to use \( \phi(T[i...i+m-1]) \) to compute \( \phi(T[i+1...i+m]) \) in constant time, which implies that in \( O(n) \) time, we can compute all the relevant fingerprints by sliding a window of length \( m \) over \( T \). This technique is commonly known as a sliding window, and hash functions allowing this technique are also known as rolling hash functions.

Algorithms that use Karp-Rabin fingerprints are Monte Carlo, meaning that there is a small probability that they encounter a collision and consequently output an incorrect answer. Even though this error probability can be made arbitrarily small, we sometimes wish to obtain algorithms that output the correct answer with certainty. To do so, we typically design a deterministic verification algorithm, which can check the correctness of the result. If the output is incorrect, we pick a new random number \( b \in \mathbb{Z}_p \) for use in \( \phi \), and run the algorithm again, and so on. The resulting algorithm is called a Las Vegas algorithm and always outputs the correct answer. Let \( t_a(n) \) and \( s_a(n) \) denote the time and space used by the Monte Carlo algorithm, and similarly, let \( t_b(n) \) and \( s_b(n) \) be the time and space used by the verifier. The Las Vegas algorithm then runs in \( O(t_a(n) + t_b(n)) \) time with high probability, and uses space \( O(t_a(n) + t_b(n)) \). Obtaining Las Vegas algorithms typically comes at the cost of increasing the time or space complexity, since typically \( t_b(n) = \omega(t_a(n)) \) or \( s_b(n) = \omega(s_a(n)) \). For example, we can design a generic verifier with \( t_b(n) = O(n^2) \) and \( s_b(n) = O(n) \) by checking all \( \Theta(n^3) \) substring pairs in \( T \) for collisions using a hash table. However, in most applications, this is too slow. Instead we typically exploit problem specific properties, or the fact that not all \( \Theta(n^3) \) substring pairs can be compared, to design better verifiers.
1.3.2 Suffix Trees

A trie is a data structure that stores a set of strings $S$ from an alphabet $\Sigma$ in an ordered, rooted tree $T$ where each edge is labeled with a character from $\Sigma$. Sibling edges must be labeled with distinct characters, and sorted according to the lexicographic ordering of their labels. Each string $x \in S$ is stored in $T$ as a path starting from the root and ending in a node $v$, i.e., $x = \text{str}(v)$, where $\text{str}(v)$ denotes the string obtained by concatenating the labels on the path from the root to $v$. The leaves of $T$ must all correspond to strings in $S$. A compacted trie is a trie in which all nodes with a single child have been removed by joining their parent edge with their child edge. The resulting edge is labeled by the concatenation of the parent and child edge labels. It is easy to verify that for an arbitrary set of strings $S$, both the trie and the compacted trie on $S$ are uniquely defined.

Given a string $S$, the suffix tree of $S$ is the compacted trie on the set of all suffixes of $S$, i.e., $S = \{S[1..n], S[2..n], \ldots, S[n]\}$. Figure 1.1(a) shows the suffix tree for the string $S = \text{acacbacbacc}$. In most applications, we append a unique character $\$ to $S$ before constructing the suffix tree. This ensures a one to one correspondence between the leaves in the suffix tree and suffixes of $S\$ and is also required by some construction algorithms. See Figure 1.1(b).

We refer to nodes in the suffix trees as explicit nodes, and we use implicit nodes to refer to locations on edges corresponding to nodes only appearing in the associated uncompacted suffix trie. Nodes that are labeled by suffixes of $S$ are called suffix nodes, and can be either implicit or explicit. If the suffix tree is built for a string ending with $\$ \notin \Sigma$, the suffix nodes are precisely the leaves. In Figure 1.1 the suffix nodes have been numbered according to the suffix they represent.

The internal explicit nodes in a suffix tree are often annotated with suffix links. The suffix link of a node $v$ labeled by the string $x = \text{str}(v)$ is a pointer from $v$ to the node labeled by the string $x[2..|x|]$. The suffix links are shown as dotted lines in Figure 1.1. It is a well-known property that the suffix link of an internal explicit node always points to another internal explicit node [147]. The suffix tree has $O(n)$ explicit nodes, and can be stored in $O(n)$ space if the edge labels are represented as pointers to substrings of $S$.

History

We briefly summarize some important historical developments. For a more detailed account, we refer to [14] and references therein.
(a) The suffix tree of acacbacbacc.
(b) The suffix tree of acacbacbacc\$. 

Figure 1.1: Examples of suffix trees and suffix links.

The suffix tree was introduced by Weiner in 1973 [147], who showed how to construct it in $O(n)$ time for a string $S$ of length $n$ from a constant size alphabet. Weiner’s algorithm constructed the suffix tree by inserting the suffixes of $S$ from right to left. In 1976 McCreight [123] gave an algorithm that inserted suffixes from left to right. Weiner and McCreight’s algorithms were both offline algorithms in the sense that they required the complete input string $S$ before they could start. In 1995 Ukkonen [144] gave an online algorithm that maintained the suffix tree of increasing prefixes of $S$, thereby constructing the suffix tree for $S$ in $O(n)$ time. However, Weiner, McCreight and Ukkonen’s algorithm were all linear time only in case of a constant size alphabet, and for general alphabets they required $O(n \log n)$ time. In 1997 Farach [54] showed how to construct the suffix tree in $O(n)$ time for polynomial sized alphabets. Contrary to the previous construction algorithms, Farach’s algorithm used a divide-and-conquer approach by first constructing suffix trees restricted to the odd and even positions of $S$, before merging them in linear time. This approach generalizes to constructing the suffix tree in sorting complexity in other models of computation as well [56].

Applications

The applications of the suffix tree are far too many to list here. Instead, we provide an overview of the common techniques that are most important to our
work. See [44, 46, 73] for examples of the many uses of suffix trees.

At the fundamental level, the suffix tree of $S$ provides a linear space index of the substrings of $S$. We usually assume that each explicit node in the suffix tree stores its outgoing edges in a perfect hash table [61] using the first character on the edge as the key. Given a pattern string $P$ of length $m$, this allows us to search or traverse the suffix tree for $P$ in $O(m)$ time, and implies that we can report all occ substrings of $S$ that match $P$ in $O(m + \text{occ})$ time.

In most applications suffix trees are combined with other data structures. Very often, we preprocess the suffix tree in linear time and space to support constant time nearest common ancestor (NCA) queries\(^2\), also known as lowest common ancestor queries [78]. Such a query takes two explicit nodes $u$ and $v$ and returns the deepest common ancestor of $u$ and $v$. This provides an efficient way of computing the longest common prefix between any two suffixes of $S$, which is a fundamental primitive in many string algorithms. Specifically, it also provides a deterministic (although space consuming) alternative to Karp-Rabin fingerprints, since it allows us to compare substrings of $S$ for equality in $O(1)$ time.

Level and weighted ancestor queries are two other widely used primitives on suffix trees. A level ancestor query takes an explicit node $v$ and an integer $i$ and returns the $i^{\text{th}}$ ancestor of $v$. After $O(n)$ time and space preprocessing, level ancestor queries can be supported in constant time [5, 22, 24, 48]. A weighted ancestor query takes the same input, but returns the (possibly implicit) node in the suffix tree corresponding to the level ancestor of $v$ in the uncompact suffix trie. Weighted ancestor queries can be defined for arbitrary edge weighted rooted trees, and in that general case it is known that any data structure for weighted ancestor queries using $O(n \text{polylog}(n))$ space must have $\Omega(\log \log n)$ query time. However, for suffix trees, Gawrychowski et al. [69] very recently showed that weighted ancestor queries can be supported in constant time after $O(n)$ time and space preprocessing.

The suffix tree is also very often combined with range reporting data structures. Without going into details, the longest common prefix of two suffixes can also be computed in constant time as a one-dimensional range minimum query [63] on the LCP array with the help of the suffix array [94, 121]. In many applications suffix trees are also used in combination with 2D range reporting data structures. See [111] by Lewenstein for a recent comprehensive survey.

\(^2\)Also sometimes known as lowest common ancestor, or LCA queries, in the literature.
1.4 On Chapter 2: Time-Space Trade-Offs for Longest Common Extensions

In this chapter we study the longest common extension (LCE) problem. This is the problem of constructing a data structure for a string \( T \) of length \( n \) that supports LCE queries. Such a query takes a pair \((i, j)\) of indices into \( T \) and returns the length of the longest common prefix of the \( i \)th and \( j \)th suffix of \( T \). We denote this length by \( |LCE(i, j)| \).

LCE queries are also commonly known as LCP (longest common prefix) queries and they are used as a fundamental primitive in a wide range of string matching algorithms. For example, Landau and Vishkin [109] showed that the approximate string matching problem can be solved efficiently using LCE queries. More specifically, this is the problem of finding all approximate occurrences of some pattern \( P \) in \( T \). Here an approximate occurrence of \( P \) in \( T \) is a substring of \( T \) that is within edit or Hamming distance \( k \) of \( P \). Examples of other string matching algorithms that directly use LCE queries include algorithms for finding palindromes and tandem repeats [74, 108, 117].

Motivated by these important applications, we study space-efficient solutions for the LCE problem. That is, we are interested in obtaining a time-space trade-offs between the space usage of the data structure and the query time.

There are two simple and well-known solutions to this problem. At one extreme we can construct a data structure that uses linear space and answers queries in constant time by storing the suffix tree combined with a nearest common ancestor data structure. At the other extreme, we can answer queries on the fly by sequentially comparing characters until we encounter a mismatch. This results in an \( O(1) \) space data structure with query time \( O(|LCE(i, j)|) \) which is \( \Omega(n) \) in the worst-case.

1.4.1 Our Contributions

We show that it is possible to obtain an almost smooth trade-off between these two extreme solutions. We present two different data structures, both parameterized by a trade-off parameter \( \tau \in [1, n] \).

The first solution is a deterministic data structure that uses \( O(n/\sqrt{\tau}) \) space and has query time \( O(\tau) \). Here the main idea is to store a sparse sample \( S \) of \( O(n/\sqrt{\tau}) \) suffixes of \( T \) in a data structure supporting constant time LCE queries. We use a combinatorial construction known as difference covers to choose the sample \( S \) in a way that guarantees that for any pair of indices \( i, j \) in \( T \), there exists some integer \( \delta < \tau \) such that \( S \) contains both the suffix of \( T \) starting at position \( i + \delta \) and \( j + \delta \). This implies that queries can be answered in \( O(\tau) \) time.
Our second solution is a randomized data structure, which uses $O(n/\tau)$ space and supports LCE queries in $O(\tau \log(|LCE(i,j)|/\tau))$ time. The data structure can be constructed in $O(n)$ time, and with high probability\(^3\) it answers all LCE queries on $T$ correctly. We also give a Las-Vegas version of the data structure that with certainty answers all queries correctly and with high probability meets the preprocessing time bound of $O(n \log n)$. The main idea is to store $O(n/\tau)$ Karp-Rabin fingerprints, and use these to answer an LCE query by comparing $O(\log(|LCE(i,j)|/\tau))$ substrings of $T$ in an exponential search.

We demonstrate how these new trade-offs for longest common extensions implies time-space trade-offs for approximate string matching and the problems of finding palindromes or tandem repeats. In particular we obtain new sublinear space solutions for the approximate string matching problem.

We also give a lower bound on the time-space product of LCE data structures in the non-uniform cell-probe model. More precisely, we show that any LCE data structure using $O(n/\tau)$ bits of space in addition to the string $T$ must use at least $\Omega(\tau)$ time to answer an LCE query. We obtain this bound by showing that any LCE data structure can be used to answer range minimum queries on a binary array $A$ in the same time and space bounds.

### 1.4.2 Future Directions

There is a significant gap between the time-space product of our deterministic and randomized data structure. At a high level this gap can be explained in part by the fact that difference covers need to have density $\sqrt{\tau}$, which often limits their practicality in algorithm design. On the other hand, our randomized data structure demonstrates that it is possible to obtain trade-offs with a time-space product that almost matches the $\Omega(n/\log n)$ lower bound. Consequently, an obvious focus of future research on this problem would be on improving our deterministic trade-off using new techniques and ideally obtaining a clean $O(n/\tau)$ space and $O(\tau)$ time trade-off.

### 1.5 On Chapter 3: Fingerprints in Compressed Strings

The enormous volume of digitized textual information of today makes it increasingly important to be able to store text data in a compressed form while still being able to answer questions about the underlying text. This challenge has resulted in a large body of research concerned with designing algorithms that

\(^3\)With probability at least $1 - 1/n^c$ for any constant $c$. 
work directly on the compressed representation of a string. Such algorithms not only save space by avoiding decompression, they also have the potential to solve the problems exponentially faster than algorithms that operate on the uncompressed string.

One of the central problems in this general area is that of finding a pattern $P$ in a compressed text. This problem is known as compressed pattern matching and was originally introduced by Amir and Benson [6] with the study of pattern matching in two-dimensional run-length encoded documents. Subsequently, algorithms for pattern matching in strings compressed by many popular schemes have been invented [65, 100, 114, 120, 129]. For the Lempel-Ziv family, Amir et al. [7] gave an algorithm for LZW compressed strings, and later Farach and Thorup [55] gave a compressed pattern matching algorithm for LZ77. Recently, these results were improved by Gawrychowski [66, 67]. Compressed pattern matching has also been studied for grammar compressed strings [98, 113, 114, 126] and for fully compressed pattern matching, where the pattern $P$ is also given in a compressed form [79, 90, 112, 138]. Furthermore, much recent work has been devoted to solutions for approximate compressed pattern matching, which asks to find approximate occurrences of $P$ in the compressed string [25, 47, 93, 118, 130, 140, 142].

The focus of our work in Chapter 3 is on building strong primitives for use in algorithms on grammar compressed strings. Grammar compression is a widely-studied and general compression scheme that represents a string $S$ of length $N$ as a context-free grammar $G$ of size $n$ that exactly produces $S$. For highly compressible strings the size of $G$ can be exponentially smaller than $S$. Grammar compression provides a powerful paradigm that with little or no overhead captures several popular compression schemes including run-length encoding, the Lempel-Ziv schemes LZ77, LZ78 and LZW [139, 148, 150, 151] and numerous others [15, 101, 110, 114, 132].

1.5.1 Our Contributions

We study the problem of constructing a data structure for a context free grammar $G$ that supports fingerprint queries. Such a query $\text{FINGERPRINT}(i, j)$ returns the Karp-Rabin fingerprint $\phi(S[i, j])$ of the substring $S[i, j]$, where $S$ is the string compressed by $G$.

By storing the Karp-Rabin fingerprints for all prefixes of $S$, $\phi(S[1, i])$ for $i = 1 \ldots N$, a fingerprint query can be answered in $O(1)$ time. However, this data structure uses $O(N)$ space which can be exponential in $n$. Another approach is to use the data structure of Gąsieniec et al. [71] which supports
linear time decompression of a prefix or suffix of the string generated by a node. To answer a query we find the deepest node that generates a string containing \( S[i] \) and \( S[j] \) and decompress the appropriate suffix of its left child and prefix of its right child. Consequently, the space usage is \( O(n) \) and the query time is \( O(h + j - i) \), where \( h \) is the height of the grammar. The \( O(h) \) time to find the correct node can be improved to \( O(\log N) \) using the data structure by Bille et al. [27] giving \( O(\log N + j - i) \) time for a \textsc{Fingerprint}(i, j) query. Note that the query time depends on the length of the decompressed string which can be large. For the case of balanced grammars (by height or weight) Gagie et al. [64] showed how to efficiently compute fingerprints for indexing Lempel-Ziv compressed strings.

We present the first data structures that answer fingerprint queries on general grammar compressed strings without decompressing any characters, and improve all of the above time-space trade-offs. We assume without loss of generality that \( G \) is a \textit{Straight Line Program} (SLP), i.e., \( G \) produces a single string and every nonterminal in \( G \) has exactly two children (Chomsky normal form). Our main result is a data structure for an SLP \( G \) that can answer \textsc{Fingerprint}(i, j) queries in \( O(\log N) \) time. The data structure uses \( O(n) \) space, and can thus be stored together with the SLP at no additional overhead. This matches the best known bounds for supporting random access in grammar compressed strings [27] We also show that for linear SLPs, which is a special variant of SLPs that capture LZ78, we can support fingerprint queries in \( O(\log \log N) \) time and \( O(n) \) space.

As an application, we demonstrate how to efficiently support longest common extension queries on the compressed string \( S \) in \( O(\log N \log \ell) \) time for general SLPs and \( O(\log \ell \log \log \ell + \log \log N) \) time for linear SLPs. Here \( \ell \) denotes the length of the LCE. We also show how obtain a Las Vegas version of both data structures by verifying that the fingerprinting function is collision free.

1.5.2 Future Directions

The generality of grammar compression makes it an ideal target model of fundamental data structures and algorithms on compressed strings. The suffix tree has been incredibly successful in combination with other data structures, and we expect the same could be the case for SLPs in the future. Besides supporting new primitive operations on strings compressed by SLPs, our work leaves open the following interesting question:

For uncompressed strings we know how to support the three fundamental primitives of random access, Karp-Rabin fingerprints and longest common
extensions efficiently. Using linear space, we can in all cases answer a query in constant time. However, for grammar compressed strings the situation is different. Here, using $O(n)$ space, we obtain query times of $O(\log N)$ for random access and Karp-Rabin fingerprints, but $O(\log^2 N)$ for longest common extensions. It would be nice if this apparent asymmetry could be eliminated.

1.6 On Chapter 4: Sparse Text Indexing in Small Space

In this chapter we study the sparse text indexing problem. Given a string $T$ of length $n$ and a list of $b$ interesting positions in $T$, the goal is to construct an index for only those $b$ positions, while using only $O(b)$ space during the construction process (in addition to storing the string $T$). Here, by index we mean a data structure allowing for the quick location of all occurrences of a pattern $P$ starting at interesting positions in $T$ only.

The ideal solution to the sparse text indexing problem would be an algorithm that fully generalizes the linear time and space construction bounds for full text indexes. That is, an algorithm which in $O(n)$ time and $O(b)$ space can construct a sparse index for $b$ arbitrary positions. Moreover the index constructed should support pattern matching queries for a pattern $P$ of length $m$ in $O(m + \text{occ})$ time. However, we are still some way from achieving this goal.

First partial results were only obtained in 1996, where Andersson et al. [10, 11] and Kärkkäinen and Ukkonen [95] considered restricted variants of the sparse text indexing problem: the first [10, 11] assumed that the interesting positions coincide with natural word boundaries of the text, and the authors achieved expected linear running time using $O(b)$ space. The expectancy was later removed [57, 87], and the result was recently generalised to variable length codes such as Huffman code [143]. The second restricted case [95] assumed that the text of interesting positions is evenly spaced; i.e., every $k^{th}$ position in the text. They achieved linear running time and optimal $O(b)$ space. It should be mentioned that the data structure by Kärkkäinen and Ukkonen [95] was not necessarily meant for finding only pattern occurrences starting at the evenly spaced indexed positions, as a large portion of the paper is devoted to recovering all occurrences from the indexed ones. Their technique has recently been refined by Kolpakov et al. [104]. Another restricted case admitting an $O(b)$ space solution is if the interesting positions have the same period $\rho$ (i.e., if position $i$ is interesting then so is position $i + \rho$). In this case the sparse suffix array can be constructed in $O(bp + b \log b)$ time. This was shown by Burkhardt and Kärkkäinen [34], who used it to sort difference cover samples leading to a clever technique for constructing the full suffix array in sublinear space. Interestingly,
their technique also implies a time-space tradeoff for sorting $b$ arbitrary suffixes in $O(v + n/ \sqrt{v})$ space and $O(\sqrt{vn} + (n/\sqrt{v}) \log(n/\sqrt{v}) + vb + b \log b)$ time for any $v \in [2, n]$.

### 1.6.1 Our Contributions

Our work focuses on construction algorithms for three sparse text indexing data structures: sparse suffix trees, sparse suffix arrays and sparse position heaps. For sparse suffix trees (and arrays) we give an $O(n \log^2 b)$ time and $O(b)$ space Monte-Carlo algorithm that with high probability correctly constructs the data structure. For sparse position heaps we show that they can be constructed slightly faster, in $O(n + b \log b)$ time and $O(b)$ space – however then pattern matching queries take $O(m^2 + occ)$ time.

In more detail, our construction for sparse suffix trees implies a general Monte-Carlo time-space trade-off: For any $\alpha \in [2, n]$, we can construct the sparse suffix tree in

$$O\left(n \log^2 b \frac{\log \alpha}{\log \alpha} + \alpha b \log^2 b \frac{\log \alpha}{\log \alpha}\right)$$

time and $O(\alpha b)$ space. Consequently, by using $O(b^{1+\varepsilon})$ space for any constant $\varepsilon > 0$ (i.e., slightly more than the $O(b)$ requirement), we can improve the construction time of the sparse suffix tree from $O(n \log^2 b)$ to $O(n \log b)$.

Finally, we give a deterministic verification algorithm that can verify the correctness of the sparse suffix tree output by our Monte-Carlo algorithm in $O(n \log^2 b + b^2 \log b)$ time and $O(b)$ space. This implies a Las-Vegas algorithm that with certainty constructs the correct sparse suffix tree in $O(b)$ space and uses $O(n \log^2 b + b^2 \log b)$ time with high probability.

The main idea in our construction is to use a new technique, which we call batched longest common extension queries, to efficiently sort the $b$ suffixes and obtain the suffix and LCP array. We show that given a batch of $q$ pairs of indices into $T$, we can compute the longest common extension of all pairs in $O((n + q) \log q)$ time and $O(q)$ space with a Monte-Carlo algorithm. This allows us to sort the $b$ suffixes using a quick-sort approach, where we in each round pick a random pivot suffix, which we compare all other suffixes to using a batched LCE query.

### 1.6.2 Future Directions

I et al. [85] very recently improved upon our $O(n \log^2 b)$ time bound and showed how to construct the sparse suffix tree in $O(n \log b)$ time and $O(b)$ space. They
did so by introducing the clever notion of an \( \ell \)-\textit{strict compact trie}, which relaxes the normal requirement that sibling edges in compact tries must have distinct first characters, and instead allows labels on sibling edges to share a common prefix of length \( \ell - 1 \). They start off with the \( n \)-\textit{strict compact trie} on the \( b \) suffixes (which is easy to construct) and over the course of \( O(\log n) \) rounds gradually refine it to a normal (1-strict) compact trie. The key trick in the refinement process is to use fingerprints to compare and group edge labels, and the challenge is to compute them efficiently using little space.

More precisely, I et al. [85] obtain a Monte-Carlo time-space trade-off and show that for any \( s \in [b, n] \) the sparse suffix tree can be constructed in \( O(n + (bn/s) \log s) \) time and \( O(s) \) space. The construction is correct with high probability. They also give a deterministic \( O(n \log b) \) time and \( O(b) \) space verification algorithm, which improves upon our \( O(n \log^2 b + b^2 \log b) \) time verification algorithm, and implies a Las-Vegas construction algorithm that correctly constructs the sparse suffix tree in \( O(b) \) space and with high probability uses \( O(n \log b) \) time.

Recall that the central open problem in sparse text indexing is whether it is possible to construct sparse text indexes in \( O(n) \) time and \( O(b) \) space that supports queries for patterns of length \( m \) in \( O(m + \text{occ}) \) time. Very interestingly, the time-space trade-off of I et al. [85] implies that using just \( O(b \log b) \) space, the sparse suffix tree can be constructed (correctly with high probability) in \( O(n) \) time. Thus it seems we could be close to achieving the goal of having sparse text index constructions that fully generalize those of full text indexes. However, other interesting questions remain. Specifically, the central role of fingerprints, in both our work and that of I et al. [85], raises the interesting question of finding fast and space-efficient deterministic constructions of sparse indexes.

### 1.7 On Chapters 5 & 6: The Longest Common Substring Problem

In Chapter 5 and Chapter 6 we study time-space trade-offs for the \textit{longest common substring} (LCS) problem. This problem should not be confused with the \textit{longest common subsequence} problem, which is also often abbreviated LCS. We are considering a general version of the problem in which the input consists of \( m \) strings \( T_1, \ldots, T_m \) of total length \( n \) and an integer \( 2 \leq d \leq m \). The output is the longest substring common to at least \( d \) of the input strings.

Notably, the special case where \( d = m = 2 \) captures the simplest form of the problem, where the goal is to find the longest common substring of two strings. Historically, this fundamental problem has received the most attention, and
work on it dates back to the very early days of combinatorial pattern matching. In the seminal paper by Knuth et al. [102], exhibiting a linear time algorithm for exact pattern matching, the authors write the following historical remark about the longest common substring problem:

“It seemed at first that there might be a way to find the longest common substring of two given strings, in time $O(n)$; but the algorithm of this paper does not readily support any such extension, and Knuth conjectured in 1970 that such efficiency would be impossible to achieve.” [102].

However, Knuth’s conjecture did not stand long. In 1972 Karp et al. [96] gave an $O(n \log n)$ time algorithm, and the year after, Weiner published his paper introducing suffix trees and showed that the longest common substring of two strings from a constant size alphabet can be found in $O(n)$ time [147]. The solution was particularly simple: Build the suffix tree over the concatenation of the two strings and find the deepest node that contains a suffix from both strings in its subtree.

The general version of the problem where $2 \leq d \leq m$ was not dealt with until 1992, when Hui [80] showed that a tree on $n$ nodes with colored leaves can be preprocessed in $O(n)$ time so every node stores the number of distinctly colored leaves in its subtree. With this information available, it is easy to find the LCS by traversing the suffix tree of the input strings in linear time to locate the deepest node having $d$ distinctly colored leaves below it.

The assumption of constant size alphabet was eliminated in 1997 when Farach [54] showed how to construct the suffix tree in $O(n)$ time and space for strings from a polynomial sized alphabet, thereby also showing that the general version of the longest common substring problem allows an $O(n)$ time and space solution for strings from such alphabets.

In Chapter 5 and Chapter 6 we revisit the longest common substring problem, specifically focusing on the space complexity of the problem. The suffix tree approach inherently requires $\Omega(n)$ space, which is infeasible in many practical situations where the strings are long. We investigate how the longest common substring problem can be efficiently solved in sublinear space, i.e., $O(n^{1-\varepsilon})$ space for a parameter $\varepsilon > 0$. In the following $\varepsilon$ refers to an arbitrary function of $n$ on the range $[0, 1]$, and thus not necessarily a constant. Before our work, very little was known about the possible time-space trade-offs for this problem.

---

4In [102] the running time is stated as $O(m + n)$ where $m$ and $n$ are the lengths of the two strings.
Space | Time | Trade-Off Interval | Description
--- | --- | --- | ---
$O(1)$ | $O(n^2 |LCS|)$ | | Naive solution
$\frac{1}{uve}$ | $O(n^{1-\varepsilon})$ & $O(n^{2(1+\varepsilon)})$ | $0 \leq \varepsilon \leq \frac{1}{2}$ | Deterministic LCE d.s. [26]
$\frac{2}{uve}$ | $O(n^{1-\varepsilon})$ & $O(n^{2+\varepsilon} \log |LCS|)$ | w.h.p. | $0 \leq \varepsilon \leq 1$ | Fingerprint LCE d.s. [26]
$\frac{3}{uve}$ | $O(n^{1-\varepsilon})$ | $O(n^{1+\varepsilon})$ | $0 < \varepsilon \leq \frac{1}{3}$ | Chapter 5
$\frac{4}{uve}$ | $O(n^{1-\varepsilon})$ & $O(n^{1+\varepsilon} \log |LCS|)$ | | $0 \leq \varepsilon \leq 1$ | Fingerprint. Correct w.h.p.
$\frac{5}{uve}$ | $O(n^{1-\varepsilon})$ & $O(n^{1+\varepsilon} \log^2 n (d \log^2 n + d^2))$ | $0 \leq \varepsilon < \frac{1}{3}$ | Chapter 5
$\frac{6}{uve}$ | $O(n^{1-\varepsilon})$ | $O(n^{1+\varepsilon})$ | $0 \leq \varepsilon \leq 1$ | Chapter 6
$\frac{7}{uve}$ | $O(n)$ | $O(n)$ | | Suffix tree [80,147]

Table 1.1: Overview of the known time-space trade-offs for the longest common substring problem in relation to our results in Chapter 5 and Chapter 6. $|LCS|$ is the length of the longest common substring.

Table 1.1 summarizes the known solutions in comparison to our new results. In the following we provide a brief description of the trade-offs in the table. For more details see Chapter 5.1.1.

In the special case $d = m = 2$, we can obtain an $O(1)$ space solution by naively comparing all pairs of substrings in time $O(n^2 |LCS|)$, where $|LCS|$ is the length of the longest common substring. This solution can generalized into a time-space trade-off by using the deterministic or randomized sublinear space LCE data structure presented in Chapter 2 to perform the $\Theta(n^2)$ LCE queries. For the general case of $2 \leq d \leq m$, we can combine hashing and Karp-Rabin fingerprints to obtain a $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon} \log |LCS|)$ time solution for any $0 \leq \varepsilon \leq 1$. The main idea is to repeatedly consider batches of $O(n^{1-\varepsilon})$ substrings of the same length and use fingerprints and a sliding window to identify the longest substring in the batch that occurs in at least $d$ strings. Additionally, we have to binary search for the length of the LCS, which results in a solution using $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon} \log |LCS|)$ time.

1.7.1 Our Contributions

In Chapter 5 we start by establishing time-space trade-offs for the $d = m = 2$ case as well as the general case. For the special case of $d = m = 2$, we obtain an $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time solution, but for the general case, we obtain a
time bound of $O(n^{1+\varepsilon}\log^2 n(d\log^2 n+d^2))$, thus yielding a rather poor trade-off for large values of $d$. Moreover, both of these trade-offs are restricted in the sense that they only work for $\varepsilon$ (roughly) in the range $[0, \frac{1}{5}]$.

In Chapter 6 we address these shortcomings, and, using a very different approach, we manage to obtain a clean $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time trade-off for the general case $2 \leq d \leq m$ that holds for any $\varepsilon \in [0, 1]$ \(^5\). This provides the first smooth time-space trade-off from constant to linear space matching the time-space product of $O(n^2)$ of the classic suffix tree solution.

In the last part of Chapter 6 we show a time-space trade-off lower bound for the LCS problem. Let $T_1$ and $T_2$ be two arbitrary strings of total length $n$ from an alphabet of size at least $n^2$. We prove that any deterministic RAM algorithm that solves the LCS problem on $T_1$ and $T_2$ using $O(n^{1-\varepsilon})$ space where $\varepsilon \in [\log\log n / \log n, 1]$ must use $\Omega(n^{\sqrt{\varepsilon \log n / \log \varepsilon \log n}})$ time. In particular for $\varepsilon = 1$, this means that any constant space algorithm that solves the LCS problem on two strings must use $\Omega(n^{\sqrt{\log n / \log \log n}})$ time. At the other extreme we obtain that any algorithm using $O(n^{1-\log\log n / \log n})$ space must use $\Omega(n^{\sqrt{\log \log n / \log \log \log n}})$ time. So in a sense, Knuth was right when he conjectured that the problem requires superlinear time, assuming he was thinking of algorithms that use little space.

1.7.2 Future Directions

The main problem left open by our work is to settle the optimal time-space product for the LCS problem. While it is tempting to guess that the answer lies in the vicinity of $\Theta(n^2)$, it seems really difficult to substantially improve our lower bound. Strong time-space product lower bounds have so far only been established in weaker models (e.g., the comparison model) or for multi-output problems (e.g., sorting an array, outputting its distinct elements and various pattern matching problems). Proving an $\Omega(n^2)$ time-space product lower bound in the RAM model for any problem where the output fits in a constant number of words (e.g., the LCS problem) is a major open problem.

Moreover, we draw the attention to a recent result by Beame et al. [18] who gave a randomized algorithm for the element distinctness problem with an $O(n^{3/2} \text{polylog } n)$ time-space product. Although it does not immediately generalize to element bidistinctness, this result shows that one should be careful about ruling out the possibility a major improvement of our $O(n^2)$ upper bound.

\(^5\)In the chapter this trade-off is stated $O(\tau)$ space and $O(n^2/\tau)$ time for $1 \leq \tau \leq n$. Here we substituted $\tau = n^{1-\varepsilon}$ to more easily compare it with the previous work.
In particular one could speculate that a randomized approach using Karp-Rabin fingerprints could lead to an algorithm for the longest common substring problem with a subquadratic time-space product.

Another interesting research direction is to study approximate versions of the longest common substring problem. Given two strings $T_1$ and $T_2$ of total length $n$ and an integer $k$, this problem asks to find longest substrings $S_1$ of $T_1$ and $S_2$ of $T_2$ such that the edit or Hamming distance between $S_1$ and $S_2$ is at most $k$. For Hamming distance, Flouri et al. [60] very recently showed that this problem allows a constant space and $O(n^2)$ time algorithm for any $k$. Moreover, they show that for $k = 1$ the problem can be solved in $O(n \log n)$ time and $O(n)$ space. Notably, their new constant space algorithm for approximate LCS completely generalizes the constant space and $O(n^2)$ time algorithm for $k = 0$ (Corollary 6.1) that we develop as stepping stone to our main result in Chapter 6.

These results introduce a third trade-off dimension to the longest common substring problem and raise a number of interesting questions. In particular, is it possible to obtain a time-space trade-off of the constant space and $O(n^2)$ time solution for any $k$? Furthermore, it would be very interesting to consider edit distance, as well as investigate whether similar solutions and trade-offs can be obtained for the general LCS problem where $2 \leq d \leq m$.

1.8 On Chapter 7: A Suffix Tree or Not A Suffix Tree?

Since their introduction in 1973 [147], suffix trees have been incredibly successful in the field of combinatorial pattern matching. Specifically, all papers in this dissertation use suffix trees in some way or another. But despite their success and the recent celebration of their 40th year anniversary, many structural properties of suffix trees are still not well-understood.

In Chapter 7 we study combinatorial properties of suffix trees, and in particular the problem of characterizing the topological structure of suffix trees. We are focusing on suffix trees constructed for arbitrary strings, i.e., strings that do not necessarily end with a unique symbol. In such suffix trees some suffixes can coincide with internal nodes or end in implicit nodes on the edges.

The nature of this problem can be illustrated by the two similar trees in Figure 1.2. If one constructs the suffix tree of the string acacbacacbacc (see Figure 1.1(a)) it has the same topological structure as the tree shown in Figure 1.2(a). However, by careful inspection and case analysis, one can argue that there is no string that has a suffix tree with the structure shown in Figure 1.2(b). We are interested in understanding why suffix trees can have some topological
structures but not others. Ideally, we want to find a general criteria that characterizes the topological structure of suffix trees. About the search for such a characterization, it has been remarked that

“the problem [of characterizing suffix trees] is so natural that anyone working with suffix trees, eventually will ask themselves this question.”

Amihood Amir, Bar-Ilan University, 2013
(personal communication)

To discuss the problem in a general framework, we introduce the following informal notion of partially specified suffix trees. A partially specified suffix tree $T$ is a specification of a subset of the structure of a suffix tree. For example, $T$ can be an unlabeled ordered rooted tree as in Figure 1.2, but it can also be annotated with suffix links, and partially or fully specify some of the edge labels\(^6\).

Given a partially specified suffix tree $T$, the suffix tree decision problem is to decide if there exists a string $S$ such that the suffix tree of $S$ has the structure specified by $T$. If such a string exists, we say that $T$ is a suffix tree and that $S$ realizes $T$. If $T$ can be realized by a string $S$ having a unique end symbol $\$, we additionally say that $T$ is a $\$-suffix tree. For example, the tree in Figure 1.2(a) is a suffix tree and is realized by the string acacbacbacc. However, the same

---

\(^6\)In Chapter 7 the symbol $\tau$ is used to denote a partially specified suffix tree. Here we use $T$ to avoid confusion with the trade-off parameter $\tau$ used in the previous sections.
tree is not a $\$-suffix tree, since the suffix labeled by the unique character $\$ must correspond to a leaf, which is a child of the root.

One of the challenging aspects of the suffix tree decision problem is that, in general, a string $S$ that realizes a partially specified suffix tree is not unique. For example, the tree in Figure 1.2(a) is also realized by the string $caacabcabcac$. Intuitively, the suffix tree decision problem becomes easier the more information that $T$ specifies. Also, it is generally easier to decide if $T$ is a $\$-suffix tree than a suffix tree, since for $\$-suffix trees, we can infer the length of the string $S$ from the number of leaves in $T$.

In the general case that $T$ is just an unlabeled ordered rooted tree (as in Figure 1.2), a polynomial time algorithm for deciding if $T$ is a suffix tree or a $\$-suffix tree is not known. Obviously, one can decide whether $T$ is a $\$-suffix tree by an exhaustive search that enumerates the suffix trees of all strings of length equal to the number of leaves in $T$. However, when deciding whether $T$ is a suffix tree, the number of leaves in $T$ only provides a lower bound on the length of the string, and without an upper bound, even an exhaustive search algorithm is not obvious. That said, it is easy to find simple and necessary properties that an unlabeled rooted tree $T$ must satisfy in order to be a suffix tree. For instance, nodes in $T$ must have at least two children, and no node can have more children than the root. More strongly it holds that

**Observation 1.1** If an unlabeled rooted tree $T$ is a suffix tree then for all subtrees $T_v$ of $T$, a tree isomorphic to $T_v$ can be obtained from $T$ by the process of repeatedly contracting an edge in $T$ that goes to a node with one or zero children.

Unfortunately, this condition is not sufficient for $T$ to be a suffix tree. For example all subtrees of the tree in Figure 1.2(b) satisfy the above criteria, and yet the tree it is not a suffix tree.

To approach this seemingly difficult problem, I et al. [84] considered the case where $T$ specifies more information about the suffix tree. More precisely, they assume that $T$ is an ordered rooted tree on $n$ nodes, which is annotated with suffix links of the internal nodes as well as the first character of all edge labels. For this case they give an $O(n)$ time algorithm for deciding if $T$ is a $\$-suffix tree. The main idea in their solution is to exploit the suffix links of the internal nodes in $T$ to infer a valid permutation of the leaves. To do so, they define a special graph, the *suffix tour graph*, and show that this graph has an Eulerian cycle if and only if $T$ is a $\$-suffix tree. Moreover, the order in which the Eulerian cycle visits the leaves in the suffix tour graph, defines a string that
realizes $T$. They also show how to remove the assumption that $T$ specifies the first character of all edge labels, for the special case of deciding whether $T$ is a $\$-$suffix tree for a string drawn from a binary alphabet.

1.8.1 Our Contributions

In Chapter 7 we study the same variant of the suffix tree decision problem considered by I et al. [84], but we focus on the general case of deciding whether $T$ is a suffix tree. As previously mentioned, this problem is more challenging, in part, because we cannot infer the length of the string $S$ from the number of leaves in $T$. We start by addressing this issue, and show that a tree on $n$ nodes is a suffix tree if and only if it is realized by a string of length $n - 1$. This bound is tight, since the tree consisting of a root and $n - 1$ leaves, needs a string of length at least $n - 1$. The bound implies an exhaustive search algorithm for deciding whether $T$ is a suffix tree, when $T$ is just an unlabeled ordered rooted tree.

In the case considered by I et al. [84] where $T$ also specifies the suffix links and the first character on every edge, we show how to decide whether $T$ is a suffix tree in $O(n)$ time. If $T$ is a suffix tree, our algorithm also outputs a string that realizes $T$. This provides a generalization over the $O(n)$ time algorithm provided by I et al. [84] for deciding if $T$ is a $\$-$suffix tree. To obtain our linear time algorithm, we extend the suffix tour graph technique to suffix trees. The main challenge is that if $T$ is a suffix tree, but not a $\$-$suffix tree, then suffix tour graph can be disconnected, and we must use non-trivial properties to infer a string that realizes $T$. We show several new properties of suffix trees and use these to characterize the relationship between suffix tour graphs of $\$-$suffix trees and suffix trees.

1.8.2 Future Directions

Cazaux and Rivals [35] very recently studied the $\$-$suffix tree decision problem. They also consider the variant where $T$ contains the suffix links of internal nodes, but they remove the assumption that $T$ specifies the first character of all edge labels. This provides an improvement over the work of I et al. [84], who only showed how to solve the problem without first characters for binary alphabets. The main idea of Cazaux and Rivals is to replace the suffix tour graph with a new graph, which is only defined on the internal nodes of $T$. Similar to the suffix tour graph approach, they show that this graph contains an Eulerian cycle with a special property if and only if $T$ is a $\$-$suffix tree. However,
the efficiency of this approach remains unclear, and Cazaux and Rivals do not explicitly bound the time complexity of their algorithm. In the worst case it seems their algorithm might have to explore an exponential number of Eulerian cycles in the graph.

While our work together with that of I et al. [84] and Cazaux and Rivals [35] do provide many new non-trivial insights about suffix trees, we are still far from the goal of having a simple characterization of the topological structure of suffix trees, which can be efficiently tested by an algorithm. To this end, the central open problem is to settle the time complexity of deciding whether an unlabeled tree \( T \) is a suffix tree, when neither suffix links nor first characters are specified. Can we exploit properties of suffix links to decide this in polynomial time, or can we prove that the problem is intractable?
CHAPTER 2

TIME-SPACE TRADE-OFFS FOR LONGEST COMMON EXTENSIONS

Philip Bille∗  Inge Li Gørtz∗  Benjamin Sach†  Hjalte Wedel Vildhøj∗

∗ Technical University of Denmark, DTU Compute
† University of Warwick, Department of Computer Science

Abstract

We revisit the longest common extension (LCE) problem, that is, preprocess a string $T$ into a compact data structure that supports fast LCE queries. An LCE query takes a pair $(i,j)$ of indices in $T$ and returns the length of the longest common prefix of the suffixes of $T$ starting at positions $i$ and $j$. We study the time-space trade-offs for the problem, that is, the space used for the data structure vs. the worst-case time for answering an LCE query. Let $n$ be the length of $T$. Given a parameter $\tau$, $1 \leq \tau \leq n$, we show how to achieve either $O(n/\sqrt{\tau})$ space and $O(\tau)$ query time, or $O(n/\tau)$ space and $O(\tau \log(|LCE(i,j)|/\tau))$ query time, where $|LCE(i,j)|$ denotes the length of the LCE returned by the query. These bounds provide the first smooth trade-offs for the LCE problem and almost match the previously known bounds at the extremes when $\tau = 1$ or $\tau = n$. We apply the result to obtain improved bounds for several applications where the LCE problem is the computational bottleneck, including approximate string matching and computing palindromes. We also present an efficient technique to reduce LCE queries on two strings to one string. Finally, we give a lower bound on the time-space product for LCE data structures in the non-uniform cell probe model showing that our second trade-off is nearly optimal.
2.1 Introduction

Given a string \( T \), the longest common extension of suffix \( i \) and \( j \), denoted \( \text{LCE}(i, j) \), is the length of the longest common prefix of the suffixes of \( T \) starting at position \( i \) and \( j \). The longest common extension problem (LCE problem) is to preprocess \( T \) into a compact data structure supporting fast longest common extension queries.

The LCE problem is a basic primitive that appears as a subproblem in a wide range of string matching problems such as approximate string matching and its variations [9, 40, 107, 109, 128], computing exact or approximate tandem repeats [74, 108, 117], and computing palindromes. In many of the applications, the LCE problem is the computational bottleneck.

In this paper we study the time-space trade-offs for the LCE problem, that is, the space used by the preprocessed data structure vs. the worst-case time used by LCE queries. We assume that the input string is given in read-only memory and is not counted in the space complexity. There are essentially only two time-space trade-offs known: At one extreme we can store a suffix tree combined with an efficient nearest common ancestor (NCA) data structure [78] (other combinations of \( \mathcal{O}(n) \) space data structures for the string can also be used to achieve this bound, e.g. [59]). This solution uses \( \mathcal{O}(n) \) space and supports LCE queries in \( \mathcal{O}(1) \) time. At the other extreme we do not store any data structure and instead answer queries simply by comparing characters from left-to-right in \( T \). This solution uses \( \mathcal{O}(1) \) space and answers an LCE\((i, j)\) query in \( \mathcal{O}(|\text{LCE}(i, j)|) = \mathcal{O}(n) \) time. This approach was recently shown to be very practical [86].

2.1.1 Our Results

We show the following main result for the longest common extension problem.

**Theorem 2.1** For a string \( T \) of length \( n \) and any parameter \( \tau \), \( 1 \leq \tau \leq n \), \( T \) can be preprocessed into a data structure supporting LCE\((i, j)\) queries on \( T \). This can be done such that the data structure

1. uses \( \mathcal{O}(\frac{n}{\sqrt{\tau}}) \) space and supports queries in \( \mathcal{O}(\tau) \) time. The preprocessing of \( T \) can be done in \( \mathcal{O}(\frac{n^2}{\sqrt{\tau}}) \) time and \( \mathcal{O}(\frac{n}{\sqrt{\tau}}) \) space.

2. uses \( \mathcal{O}(\frac{n^2}{\tau}) \) space and supports queries in \( \mathcal{O}(\tau \log(\frac{\text{LCE}(i,j)}{\tau})) \) time. The preprocessing of \( T \) can be done in \( \mathcal{O}(n) \) time and \( \mathcal{O}(\frac{n}{\tau}) \) space.
The solution is randomised (Monte-Carlo); with high probability, all queries are answered correctly.

(iii) uses $O\left(\frac{n}{\tau}\right)$ space and supports queries in $O\left(\tau \log \left(\frac{|LCE(i,j)|}{\tau}\right)\right)$ time. The preprocessing of $T$ can be done in $O(n \log n)$ time and $O(n)$ space. The solution is randomised (Las-Vegas); the preprocessing time bound is achieved with high probability.

Unless otherwise stated, the bounds in the theorem are worst-case, and with high probability means with probability at least $1 - \frac{1}{n^c}$ for any constant $c$.

Our results provide a smooth time-space trade-off that allows several new and non-trivial bounds. For instance, with $\tau = \sqrt{n}$ Theorem 2.1(i), gives a solution using $O(n^{3/4})$ space and $O(\sqrt{n})$ time. If we allow randomisation, we can use Theorem 2.1(iii) to further reduce the space to $O(\sqrt{n})$ while using query time $O(\sqrt{n} \log (|LCE(i,j)|/\sqrt{n})) = O(\sqrt{n} \log n)$. Note that at both extremes of the trade-off ($\tau = 1$ or $\tau = n$) we almost match the previously known bounds. In the conference version of this paper [26], we mistakenly claimed the preprocessing space of Theorem 2.1(iii) to be $O(n/\tau)$ but it is in fact $O(n)$. It is possible to obtain $O(n/\tau)$ preprocessing space by using $O(n \log n + n\tau)$ preprocessing time. For most applications, including those mentioned in this paper, this issue have no implications, since the time to perform the LCE queries typically dominates the preprocessing time.

Furthermore, we also consider LCE queries between two strings, i.e. the pair of indices to an LCE query is from different strings. We present a general result that reduces the query on two strings to a single one of them. When one of the strings is significantly smaller than the other, we can combine this reduction with Theorem 2.1 to obtain even better time-space trade-offs.

Finally, we give a reduction from range minimum queries that shows that any data structure using $O(n/\tau)$ bits space in addition to the string $T$ must use at least $\Omega(\tau)$ time to answer an LCE query. Hence, the time-space trade-offs of Theorem 2.1(ii) and Theorem 2.1(iii) are almost optimal.

2.1.2 Techniques

The high-level idea in Theorem 2.1 is to combine and balance out the two extreme solutions for the LCE problem. For Theorem 2.1(i) we use difference covers to sample a set of suffixes of $T$ of size $O(n/\sqrt{\tau})$. We store a compact trie combined with an NCA data structure for this sample using $O(n/\sqrt{\tau})$ space. To answer an LCE query we compare characters from $T$ until we get a mismatch or
reach a pair of sampled suffixes, which we then immediately compute the answer for. By the properties of difference covers we compare at most $O(\tau)$ characters before reaching a pair of sampled suffixes. Similar ideas have previously been used to achieve trade-offs for suffix array and LCP array construction [94,135].

For Theorem 2.1(ii) and Theorem 2.1(iii) we show how to use Rabin-Karp fingerprinting [97] instead of difference covers to reduce the space further. We show how to store a sample of $O(n/\tau)$ fingerprints, and how to use it to answer LCE queries using doubling search combined with directly comparing characters. This leads to the output-sensitive $O(\tau \log(|LCE(i,j)|/\tau))$ query time. We reduce space compared to Theorem 2.1(i) by computing fingerprints on-the-fly as we need them. Initially, we give a Monte-Carlo style randomised data structure (Theorem 2.1(iii)) that may answer queries incorrectly. However, this solution uses only $O(n)$ preprocessing time and is therefore of independent interest in applications that can tolerate errors. To get the error-free Las-Vegas style bound of Theorem 2.1(iii) we need to verify the fingerprints we compute are collision free; i.e. two fingerprints are equal if and only if the corresponding substrings of $T$ are equal. The main challenge is to do this in only $O(n \log n)$ time. We achieve this by showing how to efficiently verify fingerprints of composed samples which we have already verified, and by developing a search strategy that reduces the fingerprints we need to consider.

Finally, the reduction for LCE on two strings to a single string is based on a simple and compact encoding of the larger string using the smaller string. The encoding could be of independent interest in related problems, where we want to take advantage of different length input strings.

### 2.1.3 Applications

With Theorem 2.1 we immediately obtain new results for problems based on LCE queries. We review some of the most important below.

#### Approximate String Matching

Given strings $P$ and $T$ and an error threshold $k$, the approximate string matching problem is to report all ending positions of substrings of $T$ whose edit distance to $P$ is at most $k$. The edit distance between two strings is the minimum number of insertions, deletions, and substitutions needed to convert one string to the other. Let $m$ and $n$ be the lengths of $P$ and $T$. The Landau-Vishkin algorithm [109] solves approximate string matching using $O(nk)$ LCE queries on $P$ and substrings of $T$ of length $O(m)$. Using the standard linear space
and constant time LCE data structure, this leads to a solution using $O(nk)$ time and $O(m)$ space (the $O(m)$ space bound follows by the standard trick of splitting $T$ into overlapping pieces of size $O(m)$). If we plug in the results from Theorem 2.1 we immediately obtain the following result.

**Theorem 2.2** Given strings $P$ and $T$ of lengths $m$ and $n$, respectively, and a parameter $\tau$, $1 \leq \tau \leq m$, we can solve approximate string matching

(i) in $O\left(\frac{m}{\sqrt{\tau}}\right)$ space and $O(nk \cdot \tau + \frac{nm}{\sqrt{\tau}})$ time, or

(ii) in $O\left(\frac{m}{\tau}\right)$ space and $O(nk \cdot \tau \log m)$ time with high probability.

For instance for $\tau = \frac{m^2}{k}$ Theorem 2.2(i) gives a solution using $O(nm^{2/3}k^{1/3})$ time and $O(m^{2/3}k^{1/3})$ space. To the best of our knowledge these are the first non-trivial bounds for approximate string matching using $o(m)$ space.

**Palindromes**

Given a string $T$ the palindrome problem is to report the set of all maximal palindromes in $T$. A substring $T[i \ldots j]$ is a maximal palindrome iff $T[i \ldots j] = T[i \ldots j]^R$ and $T[i - 1 \ldots j + 1] \neq T[i - 1 \ldots j + 1]^R$. Here $T[i \ldots j]^R$ denotes the reverse of $T[i \ldots j]$. Any palindrome in $T$ occurs in the middle of a maximal palindrome, and thus the set of maximal palindromes compactly represents all palindromes in $T$. The palindrome problem appears in a diverse range of applications, see e.g. [4, 29, 73, 89, 103, 115, 122].

We can trivially solve the problem in $O(n^2)$ time and $O(1)$ space by a linear search at each position in $T$ to find the maximal palindrome. With LCE queries we can immediately speed up this search. Using the standard $O(n)$ space and constant time solution to the LCE problem this immediately implies an algorithm for the palindrome problem that uses $O(n)$ time and space (this bound can also be achieved without LCE queries [119]). Using Theorem 2.1 we immediately obtain the following result.

**Theorem 2.3** Given a string of length $n$ and a parameter $\tau$, $1 \leq \tau \leq n$, we can solve the palindrome problem

(i) in $O\left(\frac{n^2}{\sqrt{\tau}}\right)$ space and $O\left(\frac{n^2}{\sqrt{\tau}} + n\tau\right)$ time.

(ii) in $O\left(\frac{n}{\tau}\right)$ space and $O(n \cdot \tau \log n)$ time with high probability.
For $\tau = \omega(1)$, these are the first sublinear space bounds using $o(n^2)$ time. For example, for $\tau = n^{2/3}$ Theorem 2.3(i) gives a solution using $O(n^{5/3})$ time and $O(n^{2/3})$ space. Similarly, we can substitute our LCE data structures in the LCE-based variants of palindrome problems such as complemented palindromes, approximate palindromes, or gapped palindromes, see e.g. [103].

**Tandem Repeats**

Given a string $T$, the **tandem repeats problem** is to report all squares, i.e. consecutive repeated substrings in $T$. Main and Lorentz [117] gave a simple solution for this problem based on LCE queries that achieves $O(n)$ space and $O(n \log n + \text{occ})$ time, where $\text{occ}$ is the number of tandem repeats in $T$. Using different techniques Gąsieniec et al. [75] gave a solution using $O(1)$ space and $O(n \log n + \text{occ})$ time. Using Theorem 2.1 we immediately obtain the following result.

**Theorem 2.4** Given a string of length $n$ and parameter $\tau$, $1 \leq \tau \leq n$, we can solve the tandem repeats problem

(i) in $O(n^{\sqrt{\tau}})$ space and $O(n^{2/\sqrt{\tau}} + n\tau \cdot \log n + \text{occ})$ time.

(ii) in $O(n^{\tau})$ space and $O(n\tau \cdot \log^2 n + \text{occ})$ time with high probability.

While this does not improve the result by Gąsieniec et al. [75] it provides a simple LCE-based solution. Furthermore, our result generalises to the approximate versions of the tandem repeats problem, which also have solutions based on LCE queries [108].

**2.2 The Deterministic Data Structure**

We now show Theorem 2.1(i). Our deterministic time-space trade-off is based on sampling suffixes using difference covers.

**2.2.1 Difference Covers**

A difference cover modulo $\tau$ is a set of integers $D \subseteq \{0, 1, \ldots, \tau - 1\}$ such that for any distance $d \in \{0, 1, \ldots, \tau - 1\}$, $D$ contains two elements separated by distance $d$ modulo $\tau$ (see Example 2.1).

**Example 2.1** The set $D = \{1, 2, 4\}$ is a difference cover modulo 5.
A difference cover $D$ can cover at most $|D|^2$ differences, and hence $D$ must have size at least $\sqrt{\tau}$. We can also efficiently compute a difference cover within a constant factor of this bound.

**Lemma 2.1 (Colbourn and Ling [39])** For any $\tau$, a difference cover modulo $\tau$ of size at most $\sqrt{1.5\tau} + 6$ can be computed in $O(\sqrt{\tau})$ time.

### 2.2.2 The Data Structure

Let $T$ be a string of length $n$ and let $\tau$, $1 \leq \tau \leq n$, be a parameter. Our data structure consists of the compact trie of a sampled set of suffixes from $T$ combined with a NCA data structure. The sampled set of suffixes $S$ is the set of suffixes obtained by overlaying a difference cover on $T$ with period $\tau$, that is,

$$S = \{i \mid 1 \leq i \leq n \land i \mod \tau \in D\}.$$

**Example 2.2** Consider the string $T = dbcaabcabcaabcac$. If we use the difference cover from Example 2.1, we obtain the suffix sample $S = \{1, 2, 4, 6, 7, 9, 11, 12, 14, 16\}$.

By Lemma 2.1 the size of $S$ is $O(n/\sqrt{\tau})$. Hence the compact trie and the NCA data structures use $O(n/\sqrt{\tau})$ space. We construct the data structure in $O(n^2/\sqrt{\tau})$ time by inserting each of the $O(n/\sqrt{\tau})$ sampled suffixes in $O(n)$ time.

To answer an LCE$(i, j)$ query we explicitly compare characters starting from $i$ and $j$ until we either get a mismatch or we encounter a pair of sampled suffixes.
If we get a mismatch we simply report the length of the LCE. Otherwise, we do a NCA query on the sampled suffixes to compute the LCE. Since the distance to a pair of sampled suffixes is at most $\tau$ the total time to answer a query is $O(\tau)$. This concludes the proof of Theorem 2.1(i).

### 2.3 The Monte-Carlo Data Structure

We now show Theorem 2.1(ii) which is an intermediate step towards proving Theorem 2.1(iii) but is also of independent interest, providing a Monte-Carlo time-space trade-off. The technique is based on sampling suffixes using Rabin-Karp fingerprints. These fingerprints will be used to speed up queries with large LCE values while queries with small LCE values will be handled naively.

#### 2.3.1 Rabin-Karp fingerprints

Rabin-Karp fingerprints are defined as follows. Let $2n^{c+4} < p \leq 4n^{c+4}$ be some prime and choose $b \in \mathbb{Z}_p$ uniformly at random. Let $S$ be any substring of $T$, the fingerprint $\phi(S)$ is given by,

$$
\phi(S) = \sum_{k=1}^{|S|} S[k]b^k \mod p.
$$

Lemma 2.2 gives a crucial property of these fingerprints (see e.g. [97] for a proof). That is with high probability we can determine whether any two substrings of $T$ match in constant time by comparing their fingerprints.

**Lemma 2.2** Let $\phi$ be a fingerprinting function picked uniformly at random (as described above). With high probability,

$$
\phi(T[i \ldots i + \alpha - 1]) = \phi(T[j \ldots j + \alpha - 1]) \iff T[i \ldots i + \alpha - 1] = T[j \ldots j + \alpha - 1] \text{ for all } i, j, \alpha.
$$

#### 2.3.2 The Data Structure

The data structure consists of the fingerprint, $\phi_k$, of each suffix of the form $T[k\tau \ldots n]$ for $0 < k < n/\tau$, i.e. $\phi_k = \phi(T[k\tau \ldots n])$. Note that there are $O(n/\tau)$ such suffixes and the starting points of two consecutive suffix are $\tau$ characters apart. Since each fingerprint uses constant space the space usage of the data structure is $O(n/\tau)$. The $n/\tau$ fingerprints can be computed in left-to-right order by a single scan of $T$ in $O(n)$ time.
Queries  The key property we use to answer a query is given by Lemma 2.3.

Lemma 2.3 The fingerprint $\phi(T[i \ldots i + \alpha - 1])$ of any substring $T[i \ldots i + \alpha - 1]$ can be constructed in $O(\tau)$ time. If $i, \alpha$ are divisible by $\tau$, the time becomes $O(1)$.

Proof Let $k_1 = [i/\tau]$ and $k_2 = [(i+\alpha)/\tau]$ and observe that we have $\phi_{k_1}$ and $\phi_{k_2}$ stored. By the definition of $\phi$, we can compute $\phi(T[k_1 \ldots k_2 \tau - 1]) = \phi_{k_1} - \phi_{k_2} \cdot b^{(k_2 - k_1)\tau} \mod p$ in $O(1)$ time. If $i, \alpha = 0 \mod \tau$ then $k_1 \tau = i$ and $k_2 \tau = i + \alpha$ and we are done. Otherwise, similarly we can then convert $\phi(T[k_1 \ldots k_2 \tau - 1])$ into $\phi(T[k_1 \tau - 1 \ldots k_2 \tau - 2])$ in $O(1)$ time by inspecting $T[k_1 \tau - 1]$ and $T[k_2 \tau - 1]$. By repeating this final step we obtain $T[i \ldots i + \alpha - 1]$ in $O(\tau)$ time.

We now describe how to perform a query by using fingerprints to compare substrings. We define $\phi_i^k = \phi(T[k \tau \ldots (k + 2^i)\tau - 1])$ which we can compute in $O(1)$ time for any $k, i$ by Lemma 2.3.

First consider the problem of answering a query of the form $LCE(i\tau, j\tau)$. Find the largest $\ell$ such that $\phi_i^\ell = \phi_j^\ell$. When the correct $\ell$ is found convert the query into a new query $LCE((i + 2^\ell)\tau, (j + 2^\ell)\tau)$ and repeat. If no such $\ell$ exists, explicitly compare $T[i \tau \ldots (i + 1)\tau - 1]$ and $T[j \tau \ldots (j + 1)\tau - 1]$ one character at a time until a mismatch is found. Since no false negatives can occur when comparing fingerprints, such a mismatch exists. Let $\ell_0$ be the value of $\ell$ obtained for the initial query, $LCE(i\tau, j\tau)$, and $\ell_q$ the value obtained during the $q$-th recursion. For the initial query, we search for $\ell_0$ in increasing order, starting with $\ell_0 = 0$. After recursing, we search for $\ell_q$ in descending order, starting with $\ell_{q-1}$. By the maximality of $\ell_{q-1}$, we find the correct $\ell_q$. Summing over all recursions we have $O(\ell_0)$ total searching time and $O(\tau)$ time scanning $T$. The desired query time follows from observing that by the maximality of $\ell_0$, we have that $O(\tau + \ell_0) = O(\tau + \log(|LCE(i\tau, j\tau)|)/\tau))$.

Now consider the problem of answering a query of the form $LCE(i\tau, j\tau + \gamma)$ where $0 < \gamma < \tau$. By Lemma 2.3 we can obtain the fingerprint of any substring in $O(\tau)$ time. This allows us to use a similar approach to the first case. We find the largest $\ell$ such that $\phi(T[j\tau + \gamma \ldots (j + 2^\ell)\tau + \gamma - 1]) = \phi_i^\ell$ and convert the current query into a new query, $LCE((i + 2^\ell)\tau, (j + 2^\ell)\tau + \gamma)$. As we have to apply Lemma 2.3 before every comparison, we obtain a total complexity of $O(\tau \log(|LCE(i\tau, j\tau + \gamma)|)/\tau))$.

In the general case an $LCE(i, j)$ query can be reduced to one of the first two cases by scanning $O(\tau)$ characters in $T$. By Lemma 2.2, all fingerprint comparisons are correct with high probability and the result follows.
2.4 The Las-Vegas Data Structure

We now show Theorem 2.1(iii). The important observation is that when we compare the fingerprints of two strings during a query in Chapter 2.3, one of them is of the form $T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1]$ for some $\ell, j$. Consequently, to ensure all queries are correctly computed, it suffices that $\phi$ is $\tau$-good:

**Definition 2.1** A fingerprinting function, $\phi$ is $\tau$-good on $T$ iff

$$
\phi(T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1]) = \phi(T[i \ldots i + \tau \cdot 2^\ell - 1])
$$

iff $T[j\tau \ldots j\tau + \tau \cdot 2^\ell - 1] = T[i \ldots i + \tau \cdot 2^\ell - 1]$ for all $(i, j, \ell)$.

(2.2)

In this section we give an algorithm which decides whether a given $\phi$ is $\tau$-good on string $T$. The algorithm uses $O(n)$ space and takes $O(n \log n)$ time with high probability. By using $O(n \log n + n\tau)$ preprocessing space the algorithm can also be implemented to use only $O(n/\tau)$ space. By Lemma 2.2, a uniformly chosen $\phi$ is $\tau$-good with high probability and therefore (by repetition) we can generate such a $\phi$ in the same time/space bounds. For brevity we assume that $n$ and $\tau$ are powers-of-two.

2.4.1 The Algorithm

We begin by giving a brief overview of Algorithm 1. For each value of $\ell$ in ascending order (the outermost loop), Algorithm 1 checks (2.2) for all $i, j$. For some outermost loop iteration $\ell$, the algorithm inserts the fingerprint of each block-aligned substring into a dynamic perfect dictionary, $D_\ell$ (lines 3-9). A substring is block-aligned if it is of the form, $T[j\tau \ldots (j + 2^\ell)\tau - 1]$ for some $j$ (and block-unaligned otherwise). If more than one block-aligned substring has the same fingerprint, we insert only the left-most as a representative. For the first iteration, $\ell = 0$ we also build an Aho-Corasick automaton [2], denoted $AC$, with a pattern dictionary containing every block-aligned substring.

The second stage (lines 12-21) uses a sliding window technique, checking each time we slide whether the fingerprint of the current $(2^\ell \cdot \tau)$-length substring occurs in the dynamic dictionary, $D_\ell$. If so we check whether the corresponding substrings match (if not a collision has been revealed and we abort). For $\ell > 0$, we use the fact that (2.2) holds for all $i, j$ with $\ell - 1$ (otherwise, Algorithm 1 would have already aborted) to perform the check in constant time (line 18). I.e. if there is a collision it will be revealed by comparing the fingerprints of the left-half ($L_i' \neq L_k$) or right-half ($R_i' \neq R_k$) of the underlying strings. For $\ell = 0$,
the check is performed using the AC automaton (lines 20-21). We achieve this by feeding \( T \) one character at a time into the AC. By inspecting the state of the AC we can decide whether the current \( \tau \)-length substring of \( T \) matches any block-aligned substring.

**Algorithm 1** Verifying a fingerprinting function, \( \phi \) on string \( T \)

```plaintext
1: // AC is an Aho-Corasick automaton and each \( D_\ell \) is a dynamic dictionary
2: for \( \ell = 0 \ldots \log_2(n/\tau) \) do
3:   // Insert all distinct block-aligned substring fingerprints into \( D_\ell \)
4:   \( j = 1 \ldots n/\tau - 2^\ell \) do
5:     \( f_j \leftarrow \phi(T[j \tau \ldots \tau(j + 2^\ell)\tau - 1]) \)
6:     \( L_j \leftarrow \phi(T[j \tau \ldots \tau(j + 2^\ell - 1)\tau - 1]), R_j \leftarrow \phi(T[(j + 2^\ell - 1)\tau \ldots (j + 2^\ell)\tau - 1]) \)
7:     if \( \exists (f_k, L_k, R_k, k) \in D_\ell \) such that \( f_j = f_k \) then
8:       Insert \( (f_j, L_j, R_j, j) \) into \( D_\ell \) indexed by \( f_j \)
9:     if \( \ell = 0 \) then Insert \( T[j \tau \ldots (j + 1)\tau - 1] \) into AC dictionary
10: // Check for collisions between any block-aligned and unaligned substrings
11: if \( \ell = 0 \) then Feed \( T[1 \ldots \tau - 1] \) into AC
12: for \( i = 1 \ldots n - \tau \cdot 2^\ell + 1 \) do
13:     \( f'_i \leftarrow \phi(T[i \ldots i + \tau \cdot 2^\ell - 1]) \)
14:     \( L'_i \leftarrow \phi(T[i \ldots i + \tau \cdot 2^\ell - 1]), R'_i \leftarrow \phi(T[(i + 2^\ell - 1)\tau \ldots i + \tau \cdot 2^\ell - 1]) \)
15:     if \( \ell = 0 \) then Feed \( T[i + \tau - 1] \) into AC // AC now points at \( T[i \ldots i + \tau - 1] \)
16: if \( \exists (f_k, L_k, R_k, k) \in D_\ell \) such that \( f'_i = f_k \) then
17: if \( \ell > 0 \) then
18:   if \( L'_i \neq L_k \) or \( R'_i \neq R_k \) then abort
19:   else
20:     Compare \( T[i \ldots i + \tau - 1] \) to \( T[k \tau \ldots (k + 1)\tau - 1] \) by inspecting AC
21: if \( T[i \ldots i + \tau - 1] \neq T[k \tau \ldots (k + 1)\tau - 1] \) then abort
```

**Correctness** We first consider all points at which Algorithm 1 may abort. First observe that if line 21 causes an abort then (2.2) is violated for \((i, k, 0)\). Second, if line 18 causes an abort either \( L'_i \neq L_k \) or \( R'_i \neq R_k \). By the definition of \( \phi \), in either case, this implies that \( T[i \ldots i + \tau \cdot 2^\ell - 1] \neq T[k \tau \ldots k \tau + 2^\ell \tau - 1] \). By line 16, we have that \( f'_i = f_k \) and therefore (2.2) is violated for \((i, k, \ell)\). Thus, Algorithm 1 does not abort if \( \phi \) is \( \tau \)-good.
It remains to show that Algorithm 1 always aborts if $\phi$ is not $\tau$-good. Consider the total ordering on triples $(i, j, \ell)$ obtained by stably sorting (non-decreasing) by $\ell$ then $j$ then $i$. E.g. $(1, 3, 1) < (3, 2, 3) < (2, 5, 3) < (4, 5, 3)$. Let $(i^*, j^*, \ell^*)$ be the (unique) smallest triple under this order which violates (2.2). We first argue that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*)$ will be inserted into $D_{\ell^*}$ (and $AC$ if $\ell^* = 0$). For a contradiction assume that when Algorithm 1 checks for $f_{j^*}$ in $D_{\ell^*}$ (line 7, with $j = j^*, \ell = \ell^*$) we find that some $f_k = f_{j^*}$ already exists in $D_{\ell^*}$, implying that $k < j^*$. If $T[j^* \tau \ldots j^* \tau + \tau 2^\ell - 1] \neq T[k \tau \ldots k \tau + \tau 2^\ell - 1]$ then $(j^* \tau, k, \ell^*)$ violates (2.2). Otherwise, $(i^*, k, \ell^*)$ violates (2.2). In either case this contradicts the minimality of $(i^*, j^*, \ell^*)$ under the given order.

We now consider iteration $i = i^*$ of the second inner loop (when $\ell = \ell^*$). We have shown that $(f_{j^*}, L_{j^*}, R_{j^*}, j^*) \in D_{\ell^*}$ and we have that $f_{j^*} = f_{j^*}$ (so $k = j^*$) but the underlying strings are not equal. If $\ell = 0$ then we also have that $T[j^* \tau \ldots (j^* + 1) \tau - 1]$ is in the $AC$ dictionary. Therefore inspecting the current $AC$ state, will cause an abort (lines 20-21). If $\ell > 0$ then as $(i^*, j^*, \ell^*)$ is minimal, either $L_{i^*} \neq L_{j^*}$ or $R_{i^*} \neq R_{j^*}$ which again causes an abort (line 18), concluding the correctness.

**Time-Space Complexity** We begin by upper bounding the space used and the time taken to performs all dictionary operations on $D_\ell$ for any $\ell$. First observe that there are at most $O(n/\tau)$ insertions (line 8) and at most $O(n)$ look-up operations (lines 7,16). We choose the dictionary data structure employed based on the relationship between $n$ and $\tau$. If $\tau > \sqrt{n}$ then we use the deterministic dynamic dictionary of Ružić [136]. Using the correct choice of constants, this dictionary supports look-ups and insert operations in $O(1)$ and $O(\sqrt{n})$ time respectively (and linear space). As there are only $O(n/\tau) = O(\sqrt{n})$ inserts, the total time taken is $O(n)$ and the space used is $O(n/\tau)$. If $\tau \leq \sqrt{n}$ we use the Las-Vegas dynamic dictionary of Dietzfelbinger and Meyer auf der Heide [49]. If $\Theta(\sqrt{n}) = O(n/\tau)$ space is used for $D_\ell$, as we perform $O(n)$ operations, every operation takes $O(1)$ time with high probability. In either case, over all $\ell$ we take $O(n \log n)$ total time processing dictionary operations.

The operations performed on $AC$ fall conceptually into three categories, each totalling $O(n \log n)$ time. First we insert $O(n/\tau)$ $\tau$-length substrings into the $AC$ dictionary (line 9). Second, we feed $T$ into the automaton (line 11,15) and third, we inspect the $AC$ state at most $n$ times (line 20). The space to store $AC$ is $O(n)$, the total length of the substrings.

Finally we bound the time spent constructing fingerprints. We first consider computing $f'_i$ (line 13) for $i > 1$. We can compute $f'_i$ in $O(1)$ time from $f'_{i-1}$, $T[i - 1]$ and $T[i + \tau \cdot 2^\ell]$. This follows immediately from the definition of
We can compute \( L'_i \) and \( R'_i \) analogously. Over all \( i, \ell \), this gives \( O(n \log n) \) time. Similarly we can compute \( f_j \) from \( f_{j-1}, T[(j-1)\tau \ldots j\tau - 1] \) and \( T[(j-1+2^\ell)\tau \ldots (j+2^\ell)-1] \) in \( O(\tau) \) time. Again this is analogous for \( L'_i \) and \( R'_i \). Summing over all \( j, \ell \) this gives \( O(n \log n) \) time again. Finally observe that the algorithm only needs to store the current and previous values for each fingerprint so this does not dominate the space usage.

2.5 Longest Common Extensions on Two Strings

We now show how to efficiently reduce LCE queries between two strings to LCE queries on a single string. We generalise our notation as follows. Let \( P \) and \( T \) be strings of lengths \( m \) and \( n \), respectively. Define \( \text{LCE}_{P,T}(i,j) \) to be the length of the longest common prefix of the substrings of \( P \) and \( T \) starting at \( i \) and \( j \), respectively. For a single string \( P \), we define \( \text{LCE}_P(i,j) \) as usual. We can always trivially solve the LCE problem for \( P \) and \( T \) by solving it for the string obtained by concatenating \( P \) and \( T \). We show the following improved result.

**Theorem 2.5** Let \( P \) and \( T \) be strings of lengths \( m \) and \( n \), respectively. Given a solution to the LCE problem on \( P \) using \( s(m) \) space and \( q(m) \) time and a parameter \( \tau, 1 \leq \tau \leq n \), we can solve the LCE problem on \( P \) and \( T \) using \( O(n \tau + s(m)) \) space and \( O(\tau + q(m)) \) time.

For instance, plugging in Theorem 2.1(i) in Theorem 2.5 we obtain a solution using \( O(\frac{n}{\tau} + \frac{m}{\sqrt{\tau}}) \) space and \( O(\tau) \) time. Compared with the direct solution on the concatenated string that uses \( O(\frac{n+m}{\sqrt{\tau}}) \) we save substantial space when \( m \ll n \).

2.5.1 The Data Structure

The basic idea for our data structure is inspired by a trick for reducing constant factors in the space for the LCE data structures [73, Ch. 9.1.2]. We show how to extend it to obtain asymptotic improvements. Let \( P \) and \( T \) be strings of lengths \( m \) and \( n \), respectively. Our data structure for LCE queries on \( P \) and \( T \) consists of the following information.

- A data structure that supports LCE queries for \( P \) using \( s(m) \) space and \( q(m) \) query time.
- An array \( A \) of length \( \lfloor \frac{n}{\tau} \rfloor \) such that \( A[i] \) is the maximum LCE value between any suffix of \( P \) and the suffix of \( T \) starting at position \( i \cdot \tau \), that is, \( A[i] = \max_{j=1 \ldots m} \text{LCE}_{P,T}(j, i\tau) \).
• An array $B$ of length $\left\lfloor \frac{n}{\tau} \right\rfloor$ such that $B[i]$ is the index in $P$ of a suffix that maximises the LCE value, that is, $B[i] = \arg \max_{j=1 \ldots m} \text{LCE}_{P,T}(j, i\tau)$.

Arrays $A$ and $B$ use $O(n/\tau)$ space and hence the total space is $O(n/\tau + s(m))$. We answer an LCE$_{P,T}$ query as follows. Suppose that LCE$_{P,T}(i, j) < \tau$. In that case we can determine the value of LCE$_{P,T}(i, j)$ in $O(\tau)$ time by explicitly comparing the characters from position $i$ in $P$ and $j$ in $T$ until we encounter the mismatch. If LCE$_{P,T}(i, j) \geq \tau$, we explicitly compare $k < \tau$ characters until $j + k \equiv 0 \pmod{\tau}$. When this occurs we can lookup a suffix of $P$, which the suffix $j + k$ of $T$ follows at least as long as it follows the suffix $i + k$ of $P$. This allows us to reduce the remaining part of the LCE$_{P,T}$ query to an LCE$_{P}$ query between these two suffixes of $P$ as follows.

$$\text{LCE}_{P,T}(i, j) = k + \min \left( A \left[ \frac{j+k}{\tau} \right], \text{LCE}_{P} \left( i + k, B \left[ \frac{j+k}{\tau} \right] \right) \right).$$

We need to take the minimum of the two values, since, as shown by Example 2.3, it can happen that the LCE value between the two suffixes of $P$ is greater than that between suffix $i + k$ of $P$ and suffix $j + k$ of $T$. In total, we use $O(\tau + q(m))$ time to answer a query. This concludes the proof of Theorem 2.5.

**Example 2.3** Consider the query LCE$_{P,T}(2, 13)$ on the string $P$ from Example 2.2 and

$T = \underline{c} a c d \underline{e} a b a a c a a b c a a b c d c a e$

The underlined positions in $T$ indicate the positions divisible by 5. As shown below, we can use the array $A = [0, 6, 4, 2]$ and $B = [16, 3, 11, 10]$.

<table>
<thead>
<tr>
<th>$i$</th>
<th>$iv$</th>
<th>1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16</th>
<th>$A[i]$</th>
<th>$B[i]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>$d b c a a b c a a b c a a b c a a b c$</td>
<td>$\times$</td>
<td>0 16</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>$\hat{c} a a b c a \times$</td>
<td></td>
<td>6 3</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>$\hat{a} a b c \times$</td>
<td></td>
<td>4 11</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>$c a \times$</td>
<td></td>
<td>2 10</td>
</tr>
</tbody>
</table>

To answer the query LCE$_{P,T}(2, 13)$ we make $k = 2$ character comparisons and find that

$$\text{LCE}_{P,T}(2, 13) = 2 + \min \left( A \left[ \frac{13+2}{5} \right], \text{LCE}_{P} \left( 2 + 2, B \left[ \frac{13+2}{5} \right] \right) \right) = 2 + \min(4, 5) = 6.$$
2.6 Lower Bound

In this section we prove the lower bound for LCE data structures.

**Lemma 2.4** In the non-uniform cell probe model any LCE data structure that uses $O(n/\tau)$ bits additional space for an input array of size $n$, requires $\Omega(\tau)$ query time, for any $\tau$, where $1 \leq \tau \leq n$.

### Range Minimum Queries (RMQ)

Given an array $A$ of integers, a range minimum query data structure must support queries of the form:

- **RMQ($l$, $r$):** Return the position of the minimum element in $A[l, r]$.

Brodal et al. [33] proved any algorithm that uses $n/\tau$ bits additional space to solve the RMQ problem for an input array of size $n$ (in any dimension), requires $\Omega(\tau)$ query time, for any $\tau$, where $1 \leq \tau \leq n$, even in a binary array $A$ consisting only of 0s and 1s. Their proof is in the non-uniform cell probe model [125]. In this model, computation is free, and time is counted as the number of cells accessed (probed) by the query algorithm. The algorithm is allowed to be non-uniform, i.e. for different values of input parameter $n$, we can have different algorithms.

To prove Lemma 2.4, we will show that any LCE data structure can be used to support range minimum queries, using one LCE query and $O(1)$ space additional to the space of the LCE data structure.

**Reduction** Using any data structure supporting LCE queries we can support RMQ queries on a binary array $A$ as follows: In addition to the LCE data structure, store the indices $i$ and $j$ of the longest substring $A[i, j]$ of $A$ consisting of only 1’s. To answer a query RMQ($l$, $r$) compute $res = \text{LCE}(l, i)$. Let $z = j - i + 1$ denote the length of the longest substring of 1’s. Compare $res$ and $z$:

- If $res \leq z$ and $l + res \leq r$, return $l + res$.
- If $res > z$ and $l + z \leq r$, return $l + z$.
- Otherwise return any position in $[l, r]$.

To see that this correctly answers the RMQ query consider the two cases. If $res \leq z$ then $A[l, l + res - 1]$ contains only 1’s, since $A[i, j]$ contains only 1’s. Thus position $l + res$ is the index of the first 0 in $A[l, l + res]$. It follows that if
\( l + res \leq r \), then \( A[l + res] = 0 \) and is a minimum in \( A[l, r] \). Otherwise, \( A[l, r] \) contains only 1’s.

If \( res > z \) then \( A[l, l + z - 1] \) contains only 1’s and position \( A[l + z] = 0 \).

There are two cases: Either \( l + z \leq r \), in which case this position contains the first 0 in \( A[l, r] \). Or \( l + z > r \) in which case \( A[l, r] \) contains only 1’s.

### 2.7 Conclusions and Open Problems

We have presented new deterministic and randomised time-space trade-offs for the Longest Common Extension problem. In particular, we have shown that there is a data structure for LCE queries using \( O(n/\tau) \) space and supporting LCE queries in \( O(\tau \log(|LCE(i, j)|/\tau)) \) time. We have also shown that any LCE data structure using \( O(n/\tau) \) bits of space must have query time \( \Omega(\tau) \). Consequently, the time-space product of our trade-off is essentially a factor \( \log^2 n \) from optimal. It is an interesting open problem whether this gap can be closed.

Another open question, which is also of general interest in applications of error-free fingerprinting, is whether it is possible to find a \( \tau \)-good fingerprinting function on a string of length \( n \) in \( O(n \log n) \) time with high probability and \( O(n/\tau) \) space simultaneously. Moreover, a deterministic way of doing this would provide a strong tool for derandomising solutions using fingerprints, including the results in this paper.

**Acknowledgements** We thank the anonymous referees for their valuable comments, and in particular for pointing out an issue with the preprocessing space of the randomised solution.
Abstract

The Karp-Rabin fingerprint of a string is a type of hash value that due to its strong properties has been used in many string algorithms. In this paper we show how to construct a data structure for a string $S$ of size $N$ compressed by a context-free grammar of size $n$ that answers fingerprint queries. That is, given indices $i$ and $j$, the answer to a query is the fingerprint of the substring $S[i,j]$. We present the first $O(n)$ space data structures that answer fingerprint queries without decompressing any characters. For Straight Line Programs (SLP) we get $O(\log N)$ query time, and for Linear SLPs (an SLP derivative that captures LZ78 compression and its variations) we get $O(\log \log N)$ query time. Hence, our data structures has the same time and space complexity as for random access in SLPs. We utilize the fingerprint data structures to solve the longest common extension problem in query time $O(\log N \log \ell)$ and $O(\log \ell \log \log \ell + \log \log N)$ for SLPs and Linear SLPs, respectively. Here, $\ell$ denotes the length of the LCE.

---

1Supported by a grant from the Danish National Advanced Technology Foundation.
3.1 Introduction

Given a string $S$ of size $N$ and a Karp-Rabin fingerprint function $\phi$, the answer to a FINGERPRINT$(i,j)$ query is the fingerprint $\phi(S[i,j])$ of the substring $S[i,j]$. We consider the problem of constructing a data structure that efficiently answers fingerprint queries when the string is compressed by a context-free grammar of size $n$.

The fingerprint of a string is an alternative representation that is much shorter than the string itself. By choosing the fingerprint function randomly at runtime it exhibits strong guarantees for the probability of two different strings having different fingerprints. Fingerprints were introduced by Karp and Rabin [97] and used to design a randomized string matching algorithm. Since then, they have been used as a central tool to design algorithms for a wide range of problems (see e.g., [8, 13, 41–43, 55, 70, 91, 134]).

A fingerprint requires constant space and it has the useful property that given the fingerprints $\phi(S[1,i-1])$ and $\phi(S[1,j])$, the fingerprint $\phi(S[i,j])$ can be computed in constant time. By storing the fingerprints $\phi(S[1,i])$ for $i = 1 \ldots N$ a query can be answered in $O(1)$ time. However, this data structure uses $O(N)$ space which can be exponential in $n$. Another approach is to use the data structure of Gąsieniec et al. [71] which supports linear time decompression of a prefix or suffix of the string generated by a node. To answer a query we find the deepest node that generates a string containing $S[i]$ and $S[j]$ and decompress the appropriate suffix of its left child and prefix of its right child. Consequently, the space usage is $O(n)$ and the query time is $O(h + j - i)$, where $h$ is the height of the grammar. The $O(h)$ time to find the correct node can be improved to $O(\log N)$ using the data structure by Bille et al. [27] giving $O(\log N + j - i)$ time for a FINGERPRINT$(i,j)$ query. Note that the query time depends on the length of the decompressed string which can be large. For the case of balanced grammars (by height or weight) Gagie et al. [64] showed how to efficiently compute fingerprints for indexing Lempel-Ziv compressed strings.

We present the first data structures that answer fingerprint queries on general grammar compressed strings without decompressing any characters, and improve all of the above time-space trade-offs. Assume without loss of generality that the compressed string is given as a Straight Line Program (SLP). An SLP is a grammar in Chomsky normal form, i.e., each nonterminal has exactly two children. A Linear SLP is an SLP where the root is allowed to have more than two children, and for all other internal nodes, the right child must be a leaf. Linear SLPS capture the LZ78 compression scheme [151] and its variations. Our data structures give the following theorem.
**Theorem 3.1** Let $S$ be a string of length $N$ compressed into an SLP $G$ of size $n$. We can construct data structures that support fingerprint queries in:

(i) $O(n)$ space and query time $O(\log N)$

(ii) $O(n)$ space and query time $O(\log \log N)$ if $G$ is a Linear SLP

Hence, we show a data structure for fingerprint queries that has the same time and space complexity as for random access in SLPs.

Our fingerprint data structures are based on the idea that a random access query for $i$ produces a path from the root to a leaf labelled $S[i]$. The concatenation of the substrings produced by the left children of the nodes on this path produce the prefix $S[1,i]$. We store the fingerprints of the strings produced by each node and concatenate these to get the fingerprint of the prefix instead. For Theorem 3.1(i), we combine this with the fast random access data structure by Bille et al. [27]. For Linear SLPs we use the fact that the production rules form a tree to do large jumps in the SLP in constant time using a level ancestor data structure. Then a random access query is dominated by finding the node that produces $S[i]$ among the children of the root, which can be modelled as the predecessor problem.

Furthermore, we show how to obtain faster query time in Linear SLPs using finger searching techniques. Specifically, a finger for position $i$ in a Linear SLP is a pointer to the child of the root that produces $S[i]$.

**Theorem 3.2** Let $S$ be a string of length $N$ compressed into an SLP $G$ of size $n$. We can construct an $O(n)$ space data structure such that given a finger $f$ for position $i$ or $j$, we can answer a fingerprint$(i, j)$ query in time $O(\log \log D)$ where $D = |i - j|$.

Along the way we give a new and simple reduction for solving the finger predecessor problem on integers using any predecessor data structure as a black box.

In compliance with all related work on grammar compressed strings, we assume that the model of computation is the RAM model with a word size of $\log N$ bits.
3.1.1 Longest common extension in compressed strings

As an application we show how to efficiently solve the longest common extension problem (LCE). Given two indices $i, j$ in a string $S$, the answer to the LCE($i, j$) query is the length $\ell$ of the maximum substring such that $S[i, i + \ell] = S[j, j + \ell]$. The compressed LCE problem is to preprocess a compressed string to support LCE queries. On uncompressed strings this is solvable in $O(N)$ preprocessing time, $O(N)$ space, and $O(1)$ query time with a nearest common ancestor data structure on the suffix tree for $S$ [78]. Other trade-offs are obtained by doing an exponential search over the fingerprints of strings starting in $i$ and $j$ [26]. Using the exponential search in combination with the previously mentioned methods for obtaining fingerprints without decompressing the entire string we get $O((h + \ell) \log \ell)$ or $O((\log N + \ell) \log \ell)$ time using $O(n)$ space for an LCE query. Using our new (finger) fingerprint data structures and the exponential search we obtain Theorem 3.3.

**Theorem 3.3** Let $G$ be an SLP of size $n$ that produces a string $S$ of length $N$. The SLP $G$ can be preprocessed in $O(N)$ time into a Monte Carlo data structure of size $O(n)$ that supports LCE queries on $S$ in

(i) $O(\log \ell \log N)$ time

(ii) $O(\log \ell \log \log \ell + \log \log N)$ time if $G$ is a Linear SLP.

Here $\ell$ denotes the LCE value and queries are answered correctly with high probability. Moreover, a Las Vegas version of both data structures that always answers queries correctly can be obtained with $O(N^2/n \log N)$ preprocessing time with high probability.

Furthermore, when all the internal nodes in the Linear SLP are children of the root (which is the case in LZ78), we show how to reduce the Las Vegas preprocessing time to $O(N \log N \log \log N)$.

The following corollary follows immediately because an LZ77 compression [150] consisting of $n$ phrases can be transformed to an SLP with $O(n \log \frac{N}{n})$ production rules [36, 139].

**Corollary 3.1** We can solve the LCE problem in $O(n \log \frac{N}{n})$ space and $O(\log \ell \log N)$ query time for LZ77 compression.

Finally, the LZ78 compression can be modelled by a Linear SLP $G_L$ with constant overhead. Consider an LZ78 compression with $n$ phrases, denoted $r_1, \ldots, r_n$. A
terminal phrase corresponds to a leaf in $G_L$, and each phrase $r_j = (r_i, a)$, $i < j$, corresponds to a node $v \in G_L$ with $r_i$ corresponding to the left child of $v$ and the right child of $v$ being the leaf corresponding to $a$. Therefore, we get the following corollary.

**Corollary 3.2** We can solve the LCE problem in $O(n)$ space and $O(\log \ell \log \log \ell + \log \log N)$ query time for LZ78 compression.

### 3.2 Preliminaries

Let $S = S[1, |S|]$ be a string of length $|S|$. Denote by $S[i]$ the character in $S$ at index $i$ and let $S[i, j]$ be the substring of $S$ of length $j - i + 1$ from index $i \geq 1$ to $|S| \geq j \geq i$, both indices included.

A Straight Line Program (SLP) $G$ is a context-free grammar in Chomsky normal form that we represent as a node-labeled and ordered directed acyclic graph. Each leaf in $G$ is labelled with a character, and corresponds to a terminal grammar production rule. Each internal node in $G$ is labeled with a nonterminal rule from the grammar. The unique string $S(v)$ of length $\text{size}(v) = |S(v)|$ is produced by a depth-first left-to-right traversal of $v \in G$ and consist of the characters on the leaves in the order they are visited. We let $\text{root}(G)$ denote the root of $G$, and $\text{left}(v)$ and $\text{right}(v)$ denote the left and right child of an internal node $v \in G$, respectively.

A Linear SLP $G_L$ is an SLP where we allow $\text{root}(G_L)$ to have more than two children. All other internal nodes $v \in G_L$ have a leaf as right$(v)$. Although similar, this is not the same definition as given for the Relaxed SLP by Claude and Navarro [37]. The Linear SLP is more restricted since the right child of any node (except the root) must be a leaf. Any Linear SLP can be transformed into an SLP of at most double size by adding a new rule for each child of the root.

We extend the classic heavy path decomposition of Harel and Tarjan [78] to SLPs as in [27]. For each node $v \in G$, we select one edge from $v$ to a child with maximum size and call it the heavy edge. The remaining edges are light edges. Observe that $\text{size}(u) \leq \text{size}(v)/2$ if $v$ is a parent of $u$ and the edge connecting them is light. Thus, the number of light edges on any path from the root to a leaf is at most $O(\log N)$. A heavy path is a path where all edges are heavy. The heavy path of a node $v$, denoted $H(v)$, is the unique path of heavy edges starting at $v$. Since all nodes only have a single outgoing heavy edge, the heavy path $H(v)$ and its leaf $\text{leaf}(H(v))$, is well-defined for each node $v \in G$.

A predecessor data structure supports predecessor and successor queries on a set $R \subseteq U = \{0, \ldots, N - 1\}$ of $n$ integers from a universe $U$ of size $N$. The
answer to a predecessor query $\text{pred}(q)$ is the largest integer $r^- \in R$ such that $r^- \leq q$, while the answer to a successor query $\text{succ}(q)$ is the smallest integer $r^+ \in R$ such that $r^+ \geq q$. There exist predecessor data structures achieving a query time of $\mathcal{O}(\log \log N)$ using space $\mathcal{O}(n)$ [124, 145, 149].

Given a rooted tree $T$ with $n$ vertices, we let $\text{depth}(v)$ denote the length of the path from the root of $T$ to a node $v \in T$. A level ancestor data structure on $T$ supports level ancestor queries $\text{LA}(v, i)$, asking for the ancestor $u$ of $v \in T$ such that $\text{depth}(u) = \text{depth}(v) - i$. There is a level ancestor data structure answering queries in $\mathcal{O}(1)$ time using $\mathcal{O}(n)$ space [48] (see also [5, 22, 24]).

3.2.1 Fingerprinting

The Karp-Rabin fingerprint [97] of a string $x$ is defined as $\phi(x) = \sum_{i=1}^{\mid x \mid} x[i] \cdot c^i \mod p$, where $c$ is a randomly chosen positive integer, and $2N^{c+4} \leq p \leq 4N^{c+4}$ is a prime. Karp-Rabin fingerprints guarantee that given two strings $x$ and $y$, if $x = y$ then $\phi(x) = \phi(y)$. Furthermore, if $x \neq y$, then with high probability $\phi(x) \neq \phi(y)$. Fingerprints can be composed and subtracted as follows.

**Lemma 3.1** Let $x = yz$ be a string decomposable into a prefix $y$ and suffix $z$. Let $N$ be the maximum length of $x$, $c$ be a random integer and $2N^{c+4} \leq p \leq 4N^{c+4}$ be a prime. Given any two of the Karp-Rabin fingerprints $\phi(x)$, $\phi(y)$ and $\phi(z)$, it is possible to calculate the remaining fingerprint in constant time as follows:

\[
\phi(x) = \phi(y) \oplus \phi(z) = \phi(y) + c^{\mid y \mid} \cdot \phi(z) \mod p
\]

\[
\phi(y) = \phi(x) \ominus_s \phi(z) = \phi(x) - \frac{c^{\mid y \mid}}{c^{\mid z \mid}} \cdot \phi(z) \mod p
\]

\[
\phi(z) = \phi(x) \ominus_p \phi(y) = \frac{\phi(x) - \phi(y)}{c^{\mid y \mid}} \mod p
\]

In order to calculate the fingerprints of Lemma 3.1 in constant time, each fingerprint for a string $x$ must also store the associated exponent $c^{\mid x \mid} \mod p$, and we will assume this is always the case. Observe that a fingerprint for any substring $\phi(S[i, j])$ of a string can be calculated by subtracting the two fingerprints for the prefixes $\phi(S[1, i - 1])$ and $\phi(S[1, j])$. Hence, we will only show how to find fingerprints for prefixes in this paper.
3.3 Basic fingerprint queries in SLPs

We now describe a simple data structure for answering \textsc{Fingerprint}(1, i) queries for a string $S$ compressed into a SLP $G$ in time $O(h)$, where $h$ is the height of the parse tree for $S$. This method does not unpack the string to obtain the fingerprint, instead the fingerprint is generated by traversing $G$.

The data structure stores $\text{size}(v)$ and the fingerprint $\phi(S(v))$ of the string produced by each node $v \in G$. To compose the fingerprint $f = \phi(S[1, i])$ we start from the root of $G$ and do the following. Let $v'$ denote the currently visited node, and let $p = 0$ be a variable denoting the size the concatenation of strings produced by left children of visited nodes. We follow an edge to the right child of $v'$ if $p + \text{size}(\text{left}(v')) < i$, and follow a left edge otherwise. If following a right edge, update $f = f \oplus \phi(S(\text{left}(v')))$ such that the fingerprint of the full string generated by the left child of $v'$ is added to $f$, and set $p = p + \text{size}(\text{left}(v'))$. When following a left edge, $f$ and $p$ remains unchanged. When a leaf is reached, let $f = f \oplus \phi(S(v'))$ to include the fingerprint of the terminal character. Aside from the concatenation of fingerprints for substrings, this procedure resembles a random access query for the character in position $i$ of $S$.

The procedure correctly composes $f = \phi(S[1, i])$ because the order in which the fingerprints for the substrings are added to $f$ is identical to the order in which the substrings are decompressed when decompressing $S[1, i]$.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the height of the parse tree for $S[i]$, denoted $O(h)$. Only constant additional space is spent for each node in $G$, so the space usage is $O(n)$.

3.4 Faster fingerprints in SLPs

Using the data structure of Bille et al. [27] to perform random access queries allows for a faster way to answer \textsc{Fingerprint}(1, i) queries.

\textbf{Lemma 3.2 ( [27])} Let $S$ be a string of length $N$ compressed into a SLP $G$ of size $n$. Given a node $v \in G$, we can support random access in $S(v)$ in $O(\log(\text{size}(v)))$ time, at the same time reporting the sequence of heavy paths and their entry- and exit points in the corresponding depth-first traversal of $G(v)$.

The main idea is to compose the final fingerprint from substring fingerprints by performing a constant number of fingerprint additions per heavy path visited.
In order to describe the data structure, we will use the following notation. Let $V(v)$ be the left children of the nodes in $H(v)$ where the heavy path was extended to the right child, ordered by increasing depth. The order of nodes in $V(v)$ is equal to the sequence in which they occur when decompressing $S(v)$, so the concatenation of the strings produced by nodes in $V(v)$ yields the prefix $P(v) = S(v)[1, L(v)]$, where $L(v) = \sum_{u \in V(v)} \text{size}(u)$. Observe that $P(u)$ is a suffix of $P(v)$ if $u \in H(v)$. See Figure 3.1 for the relationship between $u, v$ and the defined strings.
Let each node $v \in G$ store its unique outgoing heavy path $H(v)$, the length $L(v)$, size$(v)$, and the fingerprints $\phi(P(v))$ and $\phi(S(v))$. By forming heavy path trees of total size $O(n)$ as in [27], we can store $H(v)$ as a pointer to a node in a heavy path tree (instead of each node storing the full sequence).

The fingerprint $f = \phi(S[1,i])$ is composed from the sequence of heavy paths visited when performing a single random access query for $S[i]$ using Lemma 3.2. Instead of adding all left-children of the path towards $S[i]$ to $f$ individually, we show how to add all left-children hanging from each visited heavy path in constant time per heavy path. Thus, the time taken to compose $f$ is $O(\log N)$.

More precisely, for the pair of entry- and exit-nodes $v,u$ on each heavy path $H$ traversed from the root to $S[i]$, we set $f = f \oplus (\phi(P(v)) \ominus \phi(P(u)))$ (which is allowed because $P(u)$ is a suffix of $P(v)$). If we leave $u$ by following a right-pointer, we additionally set $f = f \oplus \phi(S(\text{left}(u)))$. If $u$ is a leaf, set $f = f \oplus \phi(S(u))$ to include the fingerprint of the terminal character.

Remember that $P(v)$ is exactly the string generated from $v$ along $H$, produced by the left children of nodes on $H$ where the heavy path was extended to the right child. Thus, this method corresponds exactly to adding the fingerprint for the substrings generated by all left children of nodes on $H$ between the entry- and exit-nodes in depth-first order, and the argument for correctness from the slower fingerprint generation also applies here.

Since the fingerprint composition takes constant time per addition, the time spent generating a fingerprint using this method is bounded by the number of heavy paths traversed, which is $O(\log N)$. Only constant additional space is spent for each node in $G$, so the space usage is $O(n)$. This concludes the proof of Theorem 3.1(i).

3.5 Faster fingerprints in Linear SLPs

In this section we show how to quickly answer $\text{FINGERPRINT}(1, i)$ queries on a Linear SLP $G_L$. In the following we denote the sequence of $k$ children of root$(G_L)$ from left to right by $r_1, \ldots, r_k$. Also, let $R(j) = \sum_{m=1}^{j} \text{size}(r_m)$ for $j = 0, \ldots, k$. That is, $R(j)$ is the length of the prefix of $S$ produced by $G_L$ including $r_j$ (and $R(0)$ is the empty prefix).

We also define the dictionary tree $F$ over $G_L$ as follows. Each node $v \in G_L$ corresponds to a single vertex $v^F \in F$. There is an edge $(u^F, v^F)$ labeled $c$ if $u = \text{left}(v)$ and $c = S(\text{right}(v))$. If $v$ is a leaf, there is an edge $(\text{root}(F), v^F)$ labeled $S(v)$. That is, a left child edge of $v \in G_L$ is converted to a parent edge of $v^F \in F$ labeled like the right child leaf of $v$. Note that for any node $v \in G_L$ except the root, producing $S(v)$ is equivalent to following edges and reporting
edge labels on the path from root\( (F) \) to \( v^F \). Thus, the prefix of length \( a \) of \( S(v) \) may be produced by reporting the edge labels on the path from root\( (F) \) until reaching the ancestor of \( v^F \) at depth \( a \).

The data structure stores a predecessor data structure over the prefix lengths \( R(j) \) and the associated node \( r_j \) and fingerprint \( \phi(S[1, R(j)]) \) for \( j = 0, \ldots, k \). We also have a doubly linked list of all \( r_j \)'s with bidirectional pointers to the predecessor data structure and \( G_L \). We store the dictionary tree \( F \) over \( G_L \), augment it with a level ancestor data structure, and add bidirectional pointers between \( v \in G_L \) and \( v^F \in F \). Finally, for each node \( v \in G_L \), we store the fingerprint of the string it produces, \( \phi(S(v)) \).

A query FINGERPRINT\((1, i)\) is answered as follows. Let \( R(m) \) be the predecessor of \( i \) among \( R(0), R(1), \ldots, R(k) \). Compose the answer to FINGERPRINT\((1, i)\) from the two fingerprints \( \phi(S[1, R(m)]) \oplus \phi(S[R(m) + 1, i]) \). The first fingerprint \( \phi(S[1, R(m)]) \) is stored in the data structure and the second fingerprint \( \phi(S[R(m) + 1, i]) \) can be found as follows. Observe that \( S[R(m) + 1, i] \) is fully generated by \( r_{m+1} \) and hence a prefix of \( S(r_{m+1}) \) of length \( i - R(m) \). We can get \( r_{m+1} \) in constant time from \( r_m \) using the doubly linked list. We use a level ancestor query \( u^F = LA(r^F_{m+1}, i - R(m)) \) to determine the ancestor of \( r^F_{m+1} \) at depth \( i - R(m) \), corresponding to a prefix of \( r_{m+1} \) of the correct length. From \( u \) we can find \( \phi(S(u)) = \phi(S[R(m) + 1, i]) \).

It takes constant time to find \( \phi(S[R(m) + 1, i]) \) using a single level ancestor query and following pointers. Thus, the time to answer a query is bounded by the time spent determining \( \phi(S[1, R(m)]) \), which requires a predecessor query among \( k \) elements (i.e. the number of children of root\( (G_L) \)) from a universe.
of size $N$. The data structure uses $O(n)$ space, as there is a bijection between nodes in $G_L$ and vertices in $F$, and we only spend constant additional space per node in $G_L$ and vertex in $F$. This concludes the proof of Theorem 3.1(ii).

### 3.6 Finger fingerprints in Linear SLPs

The $O(\log\log N)$ running time of a $\text{FINGERPRINT}(1, i)$ query is dominated by having to find the predecessor $R(m)$ of $i$ among $R(0), R(1), \ldots, R(k)$. Given $R(m)$ the rest of the query takes constant time. In the following, we show how to improve the running time of a $\text{FINGERPRINT}(1, i)$ query to $O(\log\log |j - i|)$ given a finger for position $j$. Recall that a finger $f$ for a position $j$ is a pointer to the node $r_m$ producing $S[j]$. To achieve this, we present a simple linear space finger predecessor data structure that is interchangeable with any other predecessor data structure.

#### 3.6.1 Finger Predecessor

Let $R \subseteq U = \{0, \ldots, N - 1\}$ be a set of $n$ integers from a universe $U$ of size $N$. Given a finger $f \in R$ and a query point $q \in U$, the finger predecessor problem is to answer finger predecessor or successor queries in time depending on the universe distance $D = |f - q|$ from the finger to the query point. Belazzougui et al. [20] present a succinct solution for solving the finger predecessor problem relying on a modification of z-fast tries. Other previous work present dynamic finger search trees on the word RAM [12, 92]. Here, we use a simple reduction for solving the finger predecessor problem using any predecessor data structure as a black box.

**Lemma 3.3** Let $R \subseteq U = \{0, \ldots, N - 1\}$ be a set of $n$ integers from a universe $U$ of size $N$. Given a predecessor data structure with query time $t(N, n)$ using $s(N, n)$ space, we can solve the finger predecessor problem in worst case time $O(t(D, n))$ using space $O(s(N, \frac{n}{\log N}) \log N)$.

**Proof** Construct a complete balanced binary search tree $T$ over the universe $U$. The leaves of $T$ represent the integers in $U$, and we say that a vertex span the range of $U$ represented by the leaves in its subtree. Mark the leaves of $T$ representing the integers in $R$. We remove all vertices in $T$ where the subtree contains no marked vertices. Observe that a vertex at height $j$ span a universe range of size $O(2^j)$. We augment $T$ with a level ancestor data structure answering queries in constant time. Finally, left- and right-neighbour pointers are added for all nodes in $T$. 
Each internal node \( v \in T \) at height \( j \) store an instance of the given predecessor data structure for the set of marked leaves in the subtree of \( v \). The size of the universe for the predecessor data structure equals the span of the vertex and is \( \mathcal{O}(2^j)^1 \).

Given a finger \( f \in R \) and a query point \( q \in U \), we will now describe how to find both \( \text{SUCC}(q) \) and \( \text{PRED}(q) \) when \( q < f \). The case \( q > f \) is symmetric. Observe that \( f \) corresponds to a leaf in \( T \), denoted \( f_i \). We answer a query by determining the ancestor \( v \) of \( f_i \) at height \( h = \lceil \log(|f - q|) \rceil \) and its left neighbour \( v_L \) (if it exists). We query for \( \text{SUCC}(q) \) in the predecessor data structures of both \( v \) and \( v_L \), finding at least one leaf in \( T \) (since \( v \) spans \( f \) and \( q < f \)). We return the leaf representing the smallest result as \( \text{SUCC}(q) \) and its left neighbour in \( T \) as \( \text{PRED}(q) \).

Observe that the predecessor data structures in \( v \) and \( v_L \) each span a universe of size \( O(2^h) = O(|f - q|) = O(D) \). All other operations performed take constant time. Thus, for a predecessor data structure with query time \( t(N, n) \), we can answer finger predecessor queries in time \( O(t(D, n)) \).

The height of \( T \) is \( O(\log N) \), and there are \( O(n \log N) \) vertices in \( T \) (since vertices spanning no elements from \( R \) are removed). Each element from \( R \) is stored in \( O(\log N) \) predecessor data structures. Hence, given a predecessor data structure with space usage \( s(N, n) \), the total space usage of the data structure is \( O(s(N, n) \log N) \).

We reduce the size of the data structure by reducing the number of elements it stores to \( O\left(\frac{n}{\log N}\right) \). This is done by partitioning \( R \) into \( O\left(\frac{n}{\log N}\right) \) sets of consecutive elements \( R_i \) of size \( O(\log N) \). We choose the largest integer in each \( R_i \) set as the representative \( g_i \) for that set, and store that in the data structure described above. We store the integers in set \( R_i \) in an atomic heap [62, 76] capable of answering predecessor queries in \( O(1) \) time and linear space for a set of size \( O(\log N) \). Each element in \( R \) keep a pointer to the set \( R_i \) it belongs to, and each set left- and right-neighbour pointers.

Given a finger \( f \in R \) and a query point \( q \in U \), we describe how to determine \( \text{PRED}(q) \) and \( \text{SUCC}(q) \) when \( q < f \). The case \( q > f \) is symmetric. We first determine the closest representatives \( g_l \) and \( g_r \) on the left and right of \( f \), respectively. Assuming \( q < g_l \), we proceed as before using \( g_l \) as the finger into \( T \) and query point \( q \). This gives \( p = \text{PRED}(q) \) and \( s = \text{SUCC}(q) \) among the representatives. If \( g_l \) is undefined or \( g_l < q < f \leq g_r \), we select \( p = g_l \) and \( s = g_r \). To produce the final answers, we perform at most 4 queries in the atomic heaps that \( p \) and \( s \) are representatives for.

---

1The integers stored by the data structure may be shifted by some constant \( k \cdot 2^j \) for a vertex at height \( j \), but we can shift all queries by the same constant and thus the size of the universe is \( 2^j \).
All queries in the atomic heaps take constant time, and we can find \( g_l \) and \( g_r \) in constant time by following pointers. If we query a predecessor data structure, we know that the range it spans is \( O(|g_l - q|) = O(|f - q|) = O(D) \) since \( q < g_l < f \). Thus, given a predecessor data structure with query time \( t(N, n) \), we can solve the finger predecessor problem in time \( O(t(D, n)) \).

The total space spent on the atomic heaps is \( O(n) \) since they partition \( R \). The number of representatives is \( O(n \log N) \). Thus, given a predecessor data structure with space usage \( s(N, n) \), we can solve the finger predecessor problem in space \( O(s(N, n) \log N) \).

Using the van Emde Boas predecessor data structure [124, 145, 149] with \( t(N, n) = O(\log \log N) \) query time using \( s(N, n) = O(n) \) space, we obtain the following corollary.

**Corollary 3.3** Let \( R \subseteq U = \{0, \ldots, N - 1\} \) be a set of \( n \) integers from a universe \( U \) of size \( N \). Given a finger \( f \in R \) and a query point \( q \in U \), we can solve the finger predecessor problem in worst case time \( O(\log \log |f - q|) \) and space \( O(n) \).

### 3.6.2 Finger Fingerprints

We can now prove Theorem 3.2. Assume wlog that we have a finger for \( i \), i.e., we are given a finger \( f \) to the node \( r_m \) generating \( S[i] \). From this we can in constant time get a pointer to \( r_{m+1} \) in the doubly linked list and from this a pointer to \( R(m + 1) \) in the predecessor data structure. If \( R(m + 1) > j \) then \( R(m) \) is the predecessor of \( j \). Otherwise, using Corollary 3.3 we can in time \( O(\log \log |R(m + 1) - j|) \) find the predecessor of \( j \). Since \( R(m + 1) \geq i \) and the rest of the query takes constant time, the total time for the query is \( O(\log \log |i - j|) \).

### 3.7 Longest Common Extensions in Compressed Strings

Given an SLP \( G \), the longest common extension (LCE) problem is to build a data structure for \( G \) that supports longest common extension queries \( LCE(i, j) \). In this section we show how to use our fingerprint data structures as a tool for doing LCE queries and hereby obtain Theorem 3.3.
3.7.1 Computing Longest Common Extensions with Fingers

We start by showing the following general lemma that establishes the connection between LCE and fingerprint queries.

**Lemma 3.4** For any string $S$ and any partition $S = s_1 s_2 \cdots s_t$ of $S$ into $k$ non-empty substrings called phrases, $\ell = \text{LCE}(i, j)$ can be found by comparing $O(\log \ell)$ pairs of substrings of $S$ for equality. Furthermore, all substring comparisons $x = y$ are of one of the following two types:

**Type 1** Both $x$ and $y$ are fully contained in (possibly different) phrase substrings.

**Type 2** $|x| = |y| = 2^p$ for some $p = 0, \ldots, \log(\ell) + 1$ and for $x$ or $y$ it holds that

(a) The start position is also the start position of a phrase substring, or

(b) The end position is also the end position of a phrase substring.

**Proof** Let a position of $S$ be a start (end) position if a phrase starts (ends) at that position. Moreover, let a comparison of two substrings be of type 1 (type 2) if it satisfies the first (second) property in the lemma. We now describe how to find $\ell = \text{LCE}(i, j)$ by using $O(\log \ell)$ type 1 or 2 comparisons.

If $i$ or $j$ is not a start position, we first check if $S[i, i+k] = S[j, j+k]$ (type 1), where $k \geq 0$ is the minimum integer such that $i+k$ or $j+k$ is an end position. If the comparison fails, we have restricted the search for $\ell$ to two phrase substrings, and we can find the exact value using $O(\log \ell)$ type 1 comparisons.

Otherwise, $\text{LCE}(i, j) = k + \text{LCE}(i+k+1, j+k+1)$ and either $i+k+1$ or $j+k+1$ is a start position. This leaves us with the task of describing how to answer LCE$(i, j)$, assuming that either $i$ or $j$ is a start position.

We first use $p = O(\log \ell)$ type 2 comparisons to determine the biggest integer $q$ such that $S[i, i+2^p] = S[j, j+2^p]$. It follows that $\ell \in [2^p, 2^{p+1}]$. Now let $q < 2^p$ denote the length of the longest common prefix of the substrings $x = S[i+2^p+1, i+2^{p+1}]$ and $y = S[j+2^p+1, j+2^{p+1}]$, both of length $2^p$. Clearly, $\ell = 2^p + q$. By comparing the first half $x'$ of $x$ to the first half $y'$ of $y$, we can determine if $q \in [0, 2^{p-1}]$ or $q \in [2^{p-1}+1, 2^p - 1]$. By recursing we obtain the exact value of $q$ after $\log 2^p = O(\log \ell)$ comparisons.

However, comparing $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ directly is not guaranteed to be of type 1 or 2. To fix this, we compare them indirectly
using a type 1 and type 2 comparison as follows. Let \( k < 2^p \) be the minimum integer such that \( b_1 - k \) or \( b_2 - k \) is a start position. If there is no such \( k \) then we can compare \( x' \) and \( y' \) directly as a type 1 comparison. Otherwise, it holds that \( x' = y' \) if and only if \( S[b_1 - k, b_1] = S[b_2 - k, b_2] \) (type 1) and \( S[a_1 - k - 1, b_1 - k - 1] = S[a_2 - k - 1, b_2 - k - 1] \) (type 2).

Theorem 3.3 follows by using fingerprints to perform the substring comparisons. In particular, we obtain a Monte Carlo data structure that can answer a LCE query in \( O(\log \ell \log N) \) time for SLPs and in \( O(\log \ell \log \log N) \) time for Linear SLPs. In the latter case, we can use Theorem 3.2 to reduce the query time to \( O(\log \ell \log \log \ell + \log \log N) \) by observing that for all but the first fingerprint query, we have a finger into the data structure.

### 3.7.2 Verifying the Fingerprint Function

Since the data structure is Monte Carlo, there may be collisions among the fingerprints used to determine the LCE, and consequently the answer to a query may be incorrect. We now describe how to obtain a Las Vegas data structure that always answers LCE queries correctly. We do so by showing how to efficiently verify that the fingerprint function \( \phi \) is good, i.e., collision-free on all substrings compared in the computation of LCE \((i, j)\). We give two verification algorithms. One that works for LCE queries in SLPs, and a faster one that works for Linear SLPs where all internal nodes are children of the root (e.g. LZ78).

### SLPs

If we let the phrases of \( S \) be its individual characters, we can assume that all fingerprint comparisons are of type 2 (see Lemma 3.4). We thus only have to check that \( \phi \) is collision-free among all substrings of length \( 2^p, p = 0, \ldots, \log N \). We verify this in \( \log N \) rounds. In round \( p \) we maintain the fingerprint of a sliding window of length \( 2^p \) over \( S \). For each substring \( x \) we insert \( \phi(x) \) into a dictionary. If the dictionary already contains a fingerprint \( \phi(y) = \phi(x) \), we verify that \( x = y \) in constant time by checking if \( \phi(x[1, 2^p-1]) = \phi(y[1, 2^p-1]) \) and \( \phi(x[2^p-1+1, 2^p]) = \phi(y[2^p-1+1, 2^p]) \). This works because we have already verified that the fingerprinting function is collision-free for substrings of length \( 2^{p-1} \). Note that we can assume that for any fingerprint \( \phi(x) \) the fingerprints of the first and last half of \( x \) are available in constant time, since we can store and maintain these at no extra cost. In the first round \( p = 0 \), we check that \( x = y \) by comparing the two characters explicitly. If \( x \neq y \) we have found a collision and
we abort and report that $\phi$ is not good. If all rounds are successfully verified, we report that $\phi$ is good.

For the analysis, observe that computing all fingerprints of length $2^p$ in the sliding window can be implemented by a single traversal of the SLP parse tree in $O(N)$ time. Thus, the algorithm correctly decides whether $\phi$ is good in $O(N \log N)$ time and $O(N)$ space. We can easily reduce the space to $O(n)$ by carrying out each round in $O(N/n)$ iterations, where no more than $n$ fingerprints are stored in the dictionary in each iteration. So, alternatively, $\phi$ can be verified in $O(N^2/n \log N)$ time and $O(n)$ space.

**Linear SLPs**

In Linear SLPs where all internal nodes are children of the root, we can reduce the verification time to $O(N \log N \log \log N)$, while still using $O(n)$ space. To do so, we use Lemma 3.4 with the partition of $S$ being the root substrings. We verify that $\phi$ is collision-free for type 1 and type 2 comparisons separately.

**Type 1 Comparisons.** We carry out the verification in rounds. In round $p$ we check that no collisions occur among the $p$-length substrings of the root substrings as follows: We traverse the SLP maintaining the fingerprint of all $p$-length substrings. For each substring $x$ of length $p$, we insert $\phi(x)$ into a dictionary. If the dictionary already contains a fingerprint $\phi(y) = \phi(x)$ we verify that $x = y$ in constant time by checking if $x[1] = y[1]$ and $\phi(x[2,|x|]) = \phi(y[2,|y|])$ (type 1).

Every substring of a root substring ends in a leaf in the SLP and is thus a suffix of a root substring. Consequently, they can be generated by a bottom up traversal of the SLP. The substrings of length 1 are exactly the leaves. Having generated the substrings of length $p$, the substrings of length $p+1$ are obtained by following the parents left child to another root node and prepending its right child. In each round the $p$ length substrings correspond to a subset of the root nodes, so the dictionary never holds more than $n$ fingerprints. Furthermore, since each substring is a suffix of a root substring, and the root substrings have at most $N$ suffixes in total, the algorithm will terminate in $O(N)$ time.

**Type 2 Comparisons.** We adopt an approach similar to that for SLPs and verify $\phi$ in $O(\log N)$ rounds. In round $p$ we store the fingerprints of the substrings of length $2^p$ that start or end at a phrase boundary in a dictionary. We then slide a window of length $2^p$ over $S$ to find the substrings whose fingerprint equals one of those in the dictionary. Suppose the dictionary in round $p$ contains the
fingerprint $\phi(y)$, and we detect a substring $x$ such that $\phi(x) = \phi(y)$. To verify that $x = y$, assume that $y$ starts at a phrase boundary (the case when it ends in a phrase boundary is symmetric). As before, we first check that the first half of $x$ is equal to the first half of $y$ using fingerprints of length $2^{p-1}$, which we know are collision-free. Let $x' = S[a_1, b_1]$ and $y' = S[a_2, b_2]$ be the second half of $x$ and $y$. Contrary to before, we can not directly compare $\phi(x') = \phi(y')$, since neither $x'$ nor $y'$ is guaranteed to start or end at a phrase boundary. Instead, we compare them indirectly using a type 1 and type 2 comparison as follows: Let $k < 2^{p-1}$ be the minimum integer such that $b_1 - k$ or $b_2 - k$ is a start position. If there is no such $k$ then we can compare $x'$ and $y'$ directly as a type 1 comparison. Otherwise, it holds that $x' = y'$ if and only if $\phi(S[b_1 - k, b_1]) = \phi(S[b_2 - k, b_2])$ (type 1) and $\phi(S[a_1 - k - 1, b_1 - k - 1]) = \phi(S[a_2 - k - 1, b_2 - k - 1])$ (type 2), since we have already verified that $\phi$ is collision-free for type 1 comparisons and type 2 comparisons of length $2^{p-1}$.

The analysis is similar to that for SLPs. The sliding window can be implemented in $\mathcal{O}(N)$ time, but for each window position we now need $\mathcal{O}(\log \log N)$ time to retrieve the fingerprints, so the total time to verify $\phi$ for type 2 collisions becomes $\mathcal{O}(N \log N \log \log N)$. The space is $\mathcal{O}(n)$ since in each round the dictionary stores at most $\mathcal{O}(n)$ fingerprints.
CHAPTER 4

SPARSE TEXT INDEXING IN SMALL SPACE

Philip Bille*, Johannes Fischer†, Inge Li Gørtz*, Hjalte Wedel Vildhøj*
Tsvi Kopelowitz‡, Benjamin Sach§

* Technical University of Denmark, DTU Compute
† TU Dortmund, Department of Computer Science
‡ Weizmann Institute of Science, Faculty of Mathematics and Computer Science
§ University of Bristol, Department of Computer Science

Abstract

In this work we present efficient algorithms for constructing sparse suffix trees, sparse suffix arrays and sparse positions heaps for \( b \) arbitrary positions of a text \( T \) of length \( n \) while using only \( O(b) \) words of space during the construction.

Attempts at breaking the naive bound of \( \Omega(nb) \) time for constructing sparse suffix trees in \( O(b) \) space can be traced back to the origins of string indexing in 1968. First results were only obtained in 1996, but only for the case where the \( b \) suffixes were evenly spaced in \( T \). In this paper there is no constraint on the locations of the suffixes.

Our main contribution is to show that the sparse suffix tree (and array) can be constructed in \( O(n \log^2 b) \) time. To achieve this we develop a technique, that allows to efficiently answer \( b \) longest common prefix queries on suffixes of \( T \), using only \( O(b) \) space. We expect that this technique

---

*Supported partly by a grant from the Danish Council for Independent Research | Natural Sciences
will prove useful in many other applications in which space usage is a concern. Our first solution is Monte-Carlo and outputs the correct tree with high probability. We then give a Las-Vegas algorithm which also uses $O(b)$ space and runs in the same time bounds with high probability when $b = O(\sqrt{n})$. Furthermore, additional tradeoffs between the space usage and the construction time for the Monte-Carlo algorithm are given.

Finally, we show that at the expense of slower pattern queries, it is possible to construct sparse position heaps in $O(n + b \log b)$ time and $O(b)$ space.

4.1 Introduction

In the sparse text indexing problem we are given a string $T = t_1 \ldots t_n$ of length $n$, and a list of $b$ interesting positions in $T$. The goal is to construct an index for only those $b$ positions, while using only $O(b)$ words of space during the construction process (in addition to storing the text $T$). Here, by index we mean a data structure allowing for the quick location of all occurrences of patterns starting at interesting positions only. A natural application comes from computational biology, where the string would be a sequence of nucleotides or amino acids, and additional biological knowledge rules out many positions where patterns could potentially start. Another application is indexing far eastern languages, where one might be interested in indexing only those positions where words start, but natural word boundaries do not exist.

Examples of suitable $O(b)$ space indexes include suffix trees [147], suffix arrays [121] and positions heaps [53] built on only those suffixes starting at interesting positions. Of course, one can always first compute a full-text suffix tree or array in linear time, and then postprocess it to include the interesting positions only. The problem of this approach is that it needs $O(n)$ words of intermediate working space, which may be much more than the $O(b)$ words needed for the final result, and also much more than the space needed for storing $T$ itself. In situations where the RAM is large enough for the string itself, but not for an index on all positions, a more space efficient solution is desirable. Another situation is where the text is held in read-only memory and only a small amount of read-write memory is available. Such situations often arise in embedded systems or in networks, where the text may be held remotely.

A “straightforward” space-saving solution would be to sort the interesting suffixes by an arbitrary string sorter, for example, by inserting them one after the other into a compacted trie. However, such an approach is doomed to take $\Omega(nb + n \log n)$ time [23], since it takes no advantage of the fact that the strings are suffixes of one large text, so it cannot be faster than a general string sorter.
Breaking these naive bounds has been a problem that can be traced back to — according to Kärkkäinen and Ukkonen [95] — the origins of string indexing in 1968 [127]. First results were only obtained in 1996, where Andersson et al. [10, 11] and Kärkkäinen and Ukkonen [95] considered restricted variants of the problem: the first [10, 11] assumed that the interesting positions coincide with natural word boundaries of the text, and the authors achieved expected linear running time using $O(b)$ space. The expectancy was later removed [57, 87], and the result was recently generalised to variable length codes such as Huffman code [143]. The second restricted case [95] assumed that the text of interesting positions is evenly spaced; i.e., every $k^{th}$ position in the text. They achieved linear running time and optimal $O(b)$ space. It should be mentioned that the data structure by Kärkkäinen and Ukkonen [95] was not necessarily meant for finding only pattern occurrences starting at the evenly spaced indexed positions, as a large portion of the paper is devoted to recovering all occurrences from the indexed ones. Their technique has recently been refined by Kolpakov et al. [104]. Another restricted case admitting an $O(b)$ space solution is if the interesting positions have the same period $\rho$ (i.e., if position $i$ is interesting then so is position $i + \rho$). In this case the sparse suffix array can be constructed in $O(bp + b \log b)$ time. This was shown by Burkhardt and Kärkkäinen [34], who used it to sort difference cover samples leading to a clever technique for constructing the full suffix array in sublinear space. Interestingly, their technique also implies a time-space tradeoff for sorting $b$ arbitrary suffixes in $O(v + n/\sqrt{v})$ space and $O(\sqrt{vn} + (n/\sqrt{v}) \log(n/\sqrt{v}) + vb + b \log b)$ time for any $v \in [2, n]$.

4.1.1 Our Results

We present the first improvements over the naive $O(nb)$ time algorithm for general sparse suffix trees, by showing how to construct a sparse suffix tree in $O(n \log^2 b)$ time, using only $O(b)$ words of space. To achieve this, we develop a novel technique for performing efficient batched longest common prefix (LCP) queries, using little space. In particular, in Chapter 4.3, we show that a batch of $b$ LCP queries can be answered using only $O(b)$ words of space, in $O(n \log b)$ time. This technique may be of independent interest, and we expect it to be helpful in other applications in which space usage is a factor. Both algorithms are Monte-Carlo and output correct answers with high probability, i.e., at least $1 - 1/n^c$ for any constant $c$.

In Chapter 4.5 we give a Las-Vegas version of our sparse suffix tree algorithm. This is achieved by developing a deterministic verifier for the answers to a batch of $b$ longest common prefix queries in $O(n \log^2 b + b^2 \log b)$ time, using
$O(b)$ space. We show that this verifier can be used to obtain the sparse suffix tree with certainty within the same time and space bounds. For example for $b = O(\sqrt{n})$ we can construct the sparse suffix tree correctly in $O(n \log^2 b)$ time with high probability using $O(b)$ space in the worst case. This follows because, for verification, a single batch of $b$ LCP queries suffices to check the sparse suffix tree. The verifier we develop encodes the relevant structure of the text in a graph with $O(b)$ edges. We then exploit novel properties of this graph to verify the answers to the LCP queries efficiently.

In Chapter 4.6, we show some tradeoffs of construction time and space usage of our Monte-Carlo algorithm, which are based on time-space tradeoffs of the batched LCP queries. In particular we show that using $O(b\alpha)$ space the construction time is reduced to $O\left(n \log^2 b + \frac{ab \log^2 b}{\log \alpha}\right)$. So, for example, the cost for constructing the sparse suffix tree can be reduced to $O(n \log b)$ time, using $O(b^{1+\epsilon})$ words of space where $\epsilon > 0$ is any constant.

Finally, in Chapter 4.7 we show that an entirely different data structure, the position heap of Ehrenfeucht et al. [53], yields a completely different tradeoff for indexing a sparse set of positions. Position heaps are in a sense “easier” to compute than suffix trees or suffix arrays, since it is not necessary to sort the entire suffixes. The price is that, in their plain form, pattern matching is slower than with suffix trees, namely $O(m^2)$ for a pattern of length $m$. Using this approach, we show how to index $b$ positions from a text of length $n$ using $O(n + b \log b)$ time and $O(b)$ space, such that subsequent pattern matching queries (finding the $k$ occurrences starting at one of the $b$ positions) can be answered in $O(m^2 + k)$ time, for patterns of length $m$. Again, this algorithm is Monte-Carlo and outputs correct answers with high probability.

4.2 Preliminaries

For a string $T = t_1 \cdots t_n$ of length $n$, denote by $T_i = t_i \cdots t_n$ the $i$th suffix of $T$. The LCP of two suffixes $T_i$ and $T_j$ is denoted by $LCP(T_i, T_j)$, but we will slightly abuse notation and write $LCP(i, j) = LCP(T_i, T_j)$. We denote by $T_{i,j}$ the substring $t_i \cdots t_j$. We say that $T_{i,j}$ has period $\rho > 0$ iff $T_{i+\rho,j} = T_{i,j} - \rho$. Note that $\rho$ is a period of $T_{i,j}$ and not necessarily the unique minimal period of $T_{i,j}$, commonly referred to as the period. Logarithms are given in base two.

We assume the reader is familiar with both the suffix tree data structure [147] as well as suffix and LCP arrays [121].

Fingerprinting We make use of the fingerprinting techniques of Karp and Rabin [97]. Our algorithms are in the word-RAM model with word size $\Theta(\log n)$.
and we assume that each character in $T$ fits in a constant number of words. Hence each character can be interpreted as a positive integer, no larger than $n^{O(1)}$. Let $p$ be a prime between $n^c$ and $2n^c$ (where $c > 0$ is a constant picked below) and choose $r \in \mathbb{Z}_p$ uniformly at random. A fingerprint for a substring $T_{i,j}$, denoted by $FP[i,j]$, is the number $\sum_{k=i}^{j} r^{j-k} \cdot t_k \mod p$. Two equal substrings will always have the same fingerprint, however the converse is not true. Fortunately, as each character fits in $O(1)$ words, the probability of any two different substrings having the same fingerprint is at most $\frac{1}{n-\Omega(1)}$.

By making a suitable choice of $c$ and applying the union bound we can ensure that with probability at least $1 - \frac{1}{n-\Omega(1)}$, all fingerprints of substring of $T$ are collision free. I.e. for every pair of substrings $T_{i_1,j_1}$ and $T_{i_2,j_2}$ we have that $T_{i_1,j_1} = T_{i_2,j_2}$ iff $FP[i_1,j_1] = FP[i_2,j_2]$. The exponent in the probability can be amplified by increasing the value of $c$. As $c$ is a constant, any fingerprint fits into a constant number of words.

We utilize two important properties of fingerprints. The first is that $FP[i,j+1]$ can be computed from $FP[i,j]$ in constant time. This is done by the formula $FP[i,j+1] = FP[i,j] \cdot r + t_{j+1} \mod p$. The second is that the fingerprint of $T_{i,k}$ can be computed in $O(1)$ time from the fingerprint of $T_{i,j}$ and $T_{i,k}$, for $i \leq k \leq j$. This is done by the formula $FP[k,j] = FP[i,j] - FP[i,k] \cdot r^{j-k} \mod p$. Notice however that in order to perform this computation, we must have stored $r^{j-k} \mod p$ as computing it on the fly may be costly.

### 4.3 Batched LCP Queries

#### 4.3.1 The Algorithm

Given a string $T$ of length $n$ and a list of $q$ pairs of indices $P$, we wish to compute $LCP(i,j)$ for all $(i,j) \in P$. To do this we perform $\log q$ rounds of computation, where at the $k^{th}$ round the input is a set of $q$ pairs denoted by $P_k$, where we are guaranteed that for any $(i,j) \in P_k$, $LCP(i,j) \leq 2^{\log n - (k-1)}$. The goal of the $k^{th}$ iteration is to decide for any $(i,j) \in P_k$ whether $LCP(i,j) \leq 2^{\log n - k}$ or not. In addition, the $k^{th}$ round will prepare $P_{k+1}$, which is the input for the $(k+1)^{th}$ round. To begin the execution of the procedure we set $P_0 = P$, as we are always guaranteed that for any $(i,j) \in P$, $LCP(i,j) \leq n = 2^{\log n}$.

We will first provide a description of what happens during each of the $\log q$ rounds, and after we will explain how the algorithm uses $P_{\log q}$ to derive $LCP(i,j)$ for all $(i,j) \in P$. 
A Single Round  The $k$th round, for $1 \leq k \leq \log q$, is executed as follows. We begin by constructing the set $L = \bigcup_{(i,j) \in P_k} \{i - 1, j - 1, i + 2\log n - k, j + 2\log n - k\}$ of size $4q$, and construct a perfect hash table for the values in $L$, using a 2-wise independent hash function into a world of size $q^c$ for some constant $c$ (which with high probability guarantees that there are no collisions). Notice if two elements in $L$ have the same value, then we store them in a list at their hashed value. In addition, for every value in $L$ we store which index created it, so for example, for $i - 1$ and $i + 2\log n - k$ we remember they were created from $i$.

Next, we scan $T$ from $t_1$ to $t_n$. When we reach $t_\ell$ we compute $FP[1, \ell]$ in constant time from $FP[1, \ell - 1]$. In addition, if $\ell \in L$ then we store $FP[1, \ell]$ together with $\ell$ in the hash table. Once the scan of $T$ is completed, for every $(i, j) \in P_k$ we compute $FP[i, i + 2\log n - k]$ in constant time from $FP[1, i - 1]$ and $FP[1, i + 2\log n - k]$, which we have stored. Similarly we compute $FP[j, j + 2\log n - k]$. Notice that to do this we need to compute $r^{2\log n - k} \mod p = r^{\frac{n}{q}}$ in $O(\log n - k)$ time, which can be easily afforded within our bounds, as one computation suffices for all pairs.

If $FP[i, i + 2\log n - k] \neq FP[j, j + 2\log n - k]$ then $LCP(i, j) < 2\log n - k$, and so we add $(i, j)$ to $P_{k+1}$. Otherwise, with high probability $LCP(i, j) \geq 2\log n - k$ and so we add $(i + 2\log n + k, j + 2\log n + k)$ to $P_{k+1}$. Notice there is a natural bijection between pairs in $P_{k-1}$ and pairs in $P$ following from the method of constructing the pairs for the next round. For each pair in $P_{k+1}$ we will remember which pair in $P$ originated it, which can be easily transferred when $P_{k+1}$ is constructed from $P_k$.

LCP on Small Strings  After the log $q$ rounds have taken place, we know that for every $(i, j) \in P_{\log q}$, $LCP(i, j) \leq 2\log n - \log q = \frac{n}{q}$. For each such pair, we spend $O(\frac{n}{q})$ time in order to exactly compute $LCP(i, j)$. Notice that this is performed for $q$ pairs, so the total cost is $O(n)$ for this last phase. We then construct $P_{\text{final}} = \{(i + LCP(i, j), j + LCP(i, j)) : (i, j) \in P_{\log q}\}$. For each $(i, j) \in P_{\text{final}}$ denote by $(i_0, j_0) \in P$ the pair which originated $(i, j)$. We claim that for any $(i, j) \in P_{\text{final}}$, $LCP(i_0, j_0) = i - i_0$.

4.3.2 Runtime and Correctness

Each round takes $O(n + q)$ time, and the number of rounds is $O(\log q)$ for a total of $O((n + q) \log q)$ time for all rounds. The work executed for computing $P_{\text{final}}$ is an additional $O(n)$.

The following lemma on LCPs, which follows directly from the definition, will be helpful in proving the correctness of the batched LCP query.
Lemma 4.1 For any \(1 \leq i, j \leq n\), for any \(0 \leq m \leq \text{LCP}(i, j)\), it holds that 
\[ \text{LCP}(i + m, j + m) + m = \text{LCP}(i, j). \]

We now proceed on to prove that for any \((i, j) \in P_{\text{final}}\), \(\text{LCP}(i_0, j_0) = i - i_0\). Lemma 4.2 shows that the algorithm behaves as expected during the \(\log q\) rounds, and Lemma 4.3 proves that the work done in the final round suffices for computing the LCPs.

Lemma 4.2 At round \(k\), for any \((i_k, j_k) \in P_k\), \(i_k - i_0 \leq \text{LCP}(i_0, j_0) \leq i_k - i_0 + 2^{\log n - k}\), assuming the fingerprints do not give a false positive.

Proof The proof is by induction on \(k\). For the base, \(k = 0\) and so \(P_0 = P\) meaning that \(i_k = i_0\). Therefore, \(i_k - i_0 = 0 \leq \text{LCP}(i_0, j_0) \leq \log n = n\), which is always true. For the inductive step, we assume correctness for \(k - 1\) and we prove for \(k\) as follows. By the induction hypothesis, for any \((i_{k-1}, j_{k-1}) \in P_{k-1}\), \(i - i_0 \leq \text{LCP}(i_0, j_0) \leq i - i_0 + 2^{\log n - k + 1}\). Let \((i_k, j_k)\) be the pair in \(P_k\) corresponding to \((i_{k-1}, j_{k-1})\) in \(P_{k-1}\). If \(i_k = i_{k-1}\) then \(\text{LCP}(i_{k-1}, j_{k-1}) < 2^{\log n - k}\). Therefore,

\[ i_k - i_0 = i_{k-1} - i_0 \leq \text{LCP}(i_0, j_0) \leq i_{k-1} - i_0 + 2^{\log n - k}. \]

If \(i_k = i_{k-1} + 2^{\log n - k}\) then \(\text{FP}[i, i + 2^{\log n - k}] = \text{FP}[j, j + 2^{\log n - k}]\), and because we assume that the fingerprints do not produce false positives, \(\text{LCP}(i_{k-1}, j_{k-1}) \geq 2^{\log n - k}\). Therefore,

\[ i_k - i_0 = i_{k-1} + 2^{\log n - k} - i_0 \leq i_{k-1} - i_0 + \text{LCP}(i_{k-1}, j_{k-1}) \leq i_{k-1} - i_0 + 2^{\log n - k + 1} \leq i_k - i_0 + 2^{\log n - k}, \]

where the third inequality holds from Lemma 4.1, and the fourth inequality holds as \(\text{LCP}(i_0, j_0) = i_{k-1} - i_0 + \text{LCP}(i_{k-1}, j_{k-1})\) (which is the third inequality), and \(\text{LCP}(i_{k-1}, j_{k-1}) \leq 2^{\log n - k + 1}\) by the induction hypothesis.

Lemma 4.3 For any \((i, j) \in P_{\text{final}}\), \(\text{LCP}(i_0, j_0) = i - i_0(= j - j_0)\).

Proof Using Lemma 4.2 with \(k = \log q\) we have that for any \((i_{\log q}, j_{\log q}) \in P_{\log q}\), \(i_{\log q} - i_0 \leq \text{LCP}(i_0, j_0) \leq i_{\log q} - i_0 + 2^{\log n - \log q} = i_{\log q} - i_0 + \frac{n}{q}\).

Because \(\text{LCP}(i_{\log q}, j_{\log q}) \leq 2^{\log n - \log q}\) it must be that \(\text{LCP}(i_0, j_0) = i_{\log q} - i_0 + \text{LCP}(i_{\log q}, j_{\log q})\). Notice that \(i_{\text{final}} = i_{\log q} + \text{LCP}(i_{\log q}, j_{\log q})\). Therefore, \(\text{LCP}(i_0, j_0) = i_{\text{final}} - i_0\) as required.
Notice that the space used in each round is the set of pairs and the hash table for $L$, both of which require only $O(q)$ words of space. Thus, we have obtained the following. We discuss several other time/space tradeoffs in Chapter 4.6.

**Theorem 4.1** There exists a randomized Monte-Carlo algorithm that with high probability correctly answers a batch of $q$ LCP queries on suffixes from a string of length $n$. The algorithm uses $O((n + q) \log q)$ time and $O(q)$ space in the worst case.

### 4.4 Constructing the Sparse Suffix Tree

We now describe a Monte-Carlo algorithm for constructing the sparse suffix tree on any $b$ suffixes of $T$ in $O(n \log^2 b)$ time and $O(b)$ space. The main idea is to use batched LCP queries in order to sort the $b$ suffixes, as once the LCP of two suffixes is known, deciding which is lexicographically smaller than the other takes constant time by examining the first two characters that differ in said suffixes.

To arrive at the claimed complexity bounds, we will group the LCP queries into $O(\log b)$ batches each containing $q = O(b)$ queries on pairs of suffixes. One way to do this is to simulate a sorting network on the $b$ suffixes of depth $\log b$ [3]. Unfortunately, such known networks have very large constants hidden in them, and are generally considered impractical [133]. There are some practical networks with depth $\log^2 b$ such as [17], however, we wish to do better.

Consequently, we choose to simulate the quick-sort algorithm by each time picking a random suffix called the pivot, and lexicographically comparing all of the other $b - 1$ suffixes to the pivot. Once a partition is made to the set of suffixes which are lexicographically smaller than the pivot, and the set of suffixes which are lexicographically larger than the pivot, we recursively sort each set in the partition with the following modification. Each level of the recursion tree is performed concurrently using one single batch of $q = O(b)$ LCP queries for the entire level. Thus, by Theorem 4.1 a level can be computed in $O(n \log b)$ time and $O(b)$ space. Furthermore, with high probability, the number of levels in the randomized quicksort is $O(\log b)$, so the total amount of time spent is $O(n \log^2 b)$ with high probability. The time bound can immediately be made worst-case by aborting if the number of levels becomes too large, since the algorithm is still guaranteed to return the correct answer with high probability.

Notice that once the suffixes have been sorted, then we have in fact computed the sparse suffix array for the $b$ suffixes. Moreover, the corresponding sparse LCP array can be obtained as a by-product or computed subsequently by a
answering a single batch of \( q = \mathcal{O}(b) \) LCP queries in \( \mathcal{O}(n \log b) \) time. Hence we have obtained the following.

**Theorem 4.2** There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse suffix array and the sparse LCP array for any \( b \) suffixes from a string of length \( n \). The algorithm uses \( \mathcal{O}(n \log^2 b) \) time and \( \mathcal{O}(b) \) space in the worst case.

Having obtained the sparse suffix and LCP arrays, the sparse suffix tree can be constructed deterministically in \( \mathcal{O}(b) \) time and space using well-known techniques, e.g. by simulating a bottom-up traversal of the tree [99].

**Corollary 4.1** There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse suffix tree on \( b \) suffixes from a string of length \( n \). The algorithm uses \( \mathcal{O}(n \log^2 b) \) time and \( \mathcal{O}(b) \) space in the worst case.

**4.5 Verifying the Sparse Suffix and LCP Arrays**

In this section we give a deterministic algorithm which verifies the correctness of the sparse suffix and LCP arrays constructed in Theorem 4.2. This immediately gives a Las-Vegas algorithm for constructing either the sparse suffix array or sparse suffix tree with certainty.

In fact our main contribution here is an efficient algorithm which solves the general problem of verifying that \( b \) arbitrary pairs of substrings of \( T \) match. As we will see below, this suffices to verify the correctness of the sparse suffix and LCP arrays. A naive approach to verifying that \( b \) arbitrary substring pairs of \( T \) match would be to verify each pair separately in \( \Omega(n b) \) time. However, by exploiting the way the pairs overlap in \( T \), we show how to do much better. In the statement of our result in Lemma 4.4, each substring \((w_i, w'_i)\) in the input is provided by giving the indices of its leftmost and rightmost character (i.e. in \( \mathcal{O}(1) \) words).

**Lemma 4.4** Given (the locations of) \( b \) pairs of substrings \((w_1, w'_1), \ldots, (w_b, w'_b)\) of a string \( T \) of length \( n \), there is a deterministic algorithm that decides whether all pairs match, i.e. whether \( w_i = w'_i \) for all \( i = 1, \ldots, b \), in time \( \mathcal{O}(n \log^2 b + b^2 \log b) \) and \( \mathcal{O}(b) \) working space.

Before we prove Lemma 4.4, we first discuss how it can be used to verify a batch of LCP queries and then in turn to verify the sparse suffix array. Consider some LCP query \((i, j)\) for which the answer \( LCP(i, j) \) has
been computed (perhaps incorrectly). By definition, it suffices to check that
\( T_{i,i+LCP(i,j)-1} = T_{j,j+LCP(i,j)-1} \) and \( t_{i+LCP(i,j)} \neq t_{j+LCP(i,j)} \). The latter
check takes \( O(1) \) time per query while the former is exactly the problem solved
in Lemma 4.4. Lemma 4.5 then follows immediately from Lemma 4.4 and the
Monte-Carlo algorithm for batched LCP queries we gave in Theorem 4.1.

**Lemma 4.5** There exists a randomized Las-Vegas algorithm that correctly
answers a batch of \( b \) LCP queries on suffixes from a string of length \( n \). The
algorithm runs in \( O(n \log^2 b + b^2 \log b) \) time with high probability and uses
\( O(b) \) space in the worst case.

Finally observe that as lexicographical ordering is transitive it suffices to
verify the correct ordering of each pair of indices which are adjacent in the
sparse suffix array. The correct ordering of any two suffixes \( T_i \) and \( T_j \) can be de-
cided deterministically in constant time by comparing \( t_{i+LCP(i,j)} \) to \( t_{j+LCP(i,j)} \).
Therefore the problem reduces to checking the LCP of each pair of indices which
are adjacent in the sparse suffix array and the result then follows.

**Theorem 4.3** There exists a randomized Las-Vegas algorithm that correctly
constructs the sparse suffix array and the sparse LCP array for any \( b \) suffixes
from a string of length \( n \). The algorithm uses \( O(n \log^2 b + b^2 \log b) \) time with high probability and \( O(b) \) space in the worst case.

### 4.5.1 Proof of Lemma 4.4

As before, our algorithm performs \( O(\log b) \) rounds of computation. The rounds
occur in decreasing order. In round \( k \) the input is a set of (at most) \( b \) pairs of
substrings to be verified. Every substring considered in round \( k \) has length \( m_k = 2^k \).
Therefore they can be described as a pair of indices \( \{x,y\} \), corresponding
to a pair of substrings \( T_{x,x+m_k-1} \) and \( T_{y,y+m_k-1} \) where \( m_k = 2^k \). We say that
\( \{x,y\} \) matches iff \( T_{x,x+m_k-1} = T_{y,y+m_k-1} \). The initial, largest value of \( k \) is the
largest integer such that \( m_k < n \). We perform \( O(\log b) \) rounds, halting when
\( n/b < m_k < 2n/b \) after which point we can verify all pairs by scanning \( T \) in
\( O(m_k \cdot b) = O(n) \) time.

Of course in Lemma 4.4, substring pairs can have arbitrary lengths. This is
resolved by inserting two overlapping pairs into the appropriate round. I.e. if
the original input contains a pair of substrings \( (T_{x,x+d-1}, T_{y,x+d-1}) \) we insert
two index pairs into round \( k = \lceil \log d \rceil \):

\[
\{x, y\} \text{ and } \{x + d - 1 - m_k, y + d - 1 - m_k\}.
\]
In round $k$ we will replace each pair $\{x,y\}$ with a new pair $\{x',y'\}$ to be inserted into round $(k-1)$ such that $T_{x,x+m_k-1} = T_{y,y+m_k-1}$ iff $T_{x',x'+m_k-1-1} = T_{y',y'+m_k-1-1}$. Each new pair will in fact always correspond to substrings of the old pair. In some cases we may choose to directly verify some $\{x,y\}$, in which case no new pair is inserted into the next round.

We now focus on an arbitrary round $k$ and for brevity we let $m = m_k$ when $k$ is clear from the context.

**The Suffix Implication Graph**  We start each round by building a graph $(V,E)$ which encodes the overlap structure of the pairs we are to verify. We build the vertex set $V$ greedily. Consider each text index $1 \leq x \leq n$ in ascending order. We include index $x$ as a vertex in $V$ iff it occurs in some pair $\{x,y\}$ (or $\{y,x\}$) and the last index included in $V$ was at least $m/(9 \cdot \log b)$ characters ago. Observe that $|V| \leq 9 \cdot (n/m) \log b$ and also $|V| \leq b$ as it contains at most one vertex per index pair. It is simple to build the suffix implication graph in $O(b \log b)$ time by traversing the pairs in sorted order. As $|E| \leq b$ we can store the graph in $O(b)$ space.

See Figure 4.1 for an example of the suffix implication graph. Each pair of indices $\{x,y\}$ corresponds to an edge between vertices $v(x)$ and $v(y)$. Here $v(x)$ is the unique vertex such that $v(x) \leq x < v(x) + m/(9 \cdot \log b)$. The vertex $v(y)$ is defined analogously. Where it is clear from context, we will abuse notation by using $\{x,y\}$ to refer to the match and the corresponding edge in the graph. Note that the graph can have multiple edges and self-loops, which we are going to treat as cycles of length 2 and 1, respectively.

We now discuss the structure of the graph constructed and show how it can be exploited to efficiently verify the pairs in a round. The following simple lemma will be essential to our algorithm and underpins the main arguments below. The result is folklore but we include a proof for completeness.

**Lemma 4.6** Let $(V,E)$ be an undirected graph in which every vertex has degree at least three. There is a (simple) cycle in $(V,E)$ of length at most $2 \log |V| + 1$.

**Proof** Let $v$ be any vertex in $V$. First observe that as each vertex has degree at least three, there must be a cycle (keep walking until you get back to somewhere you’ve been before). Perform a breadth first search from $v$. Every time you increase the distance from $v$ by one, either the number of different vertices seen doubles or a cycle is found. This is because each vertex has
degree at least three (but you arrived via one of the edges). Two of these 
edges must lead to new, undiscovered vertices (or a cycle has been found). 
Therefore when a cycle is discovered the length of the path from \( v \) is at most 
\( \log |V| \). Note that this cycle may not contain \( v \). However as the distance 
from \( v \) to any vertex found is at most \( \log |V'| \), the cycle length is at most 
\( 2 \log |V| + 1 \). As \( v \) was arbitrary, the lemma follows.

The graph we have constructed may have vertices with degree less than 
three, preventing us from applying Lemma 4.6. For each vertex \( v(x) \) with degree 
less than three, we verify every index pair \( \{x, y\} \) (which corresponds to a unique 
edge). By directly scanning the corresponding text portions this takes \( O(|V|m) \) 
time. We can then safely remove all such vertices and the corresponding edges.
This may introduce new low degree vertices which are then verified iteratively in the same manner. As $|V| \leq 9 \cdot (n/m) \log b$, this takes a total of $\mathcal{O}(n \log b)$ time. In the remainder we continue under the assumption that every vertex has degree at least three.

**Algorithm Summary** Consider the graph $(V, E)$. As every vertex has degree at least three, there is a short cycle of length at most $2 \log |V| + 1 \leq 2 \log b + 1$ by Lemma 4.6. We begin by finding such a cycle in $\mathcal{O}(b)$ time by performing a BFS of $(V, E)$ starting at any vertex (this follows immediately from the proof of Lemma 4.6). Having located such a cycle, we will distinguish two cases. The first case is when the cycle is lock-stepped (defined below) and the other when it is unlocked. In both cases we will show below that we can exploit the structure of the text to safely delete an edge from the cycle, breaking the cycle. The index pair corresponding to the deleted edge will be replaced by a new index pair to be inserted into the next round where $m \leftarrow m_{k-1} = m_k/2$. Observe that both cases reduce the number of edges in the graph by one. Whenever we delete an edge we may reduce the degree of some vertex to below three. In this case we immediately directly process this vertex in $\mathcal{O}(m)$ time as discussed above (iterating if necessary). As we do this at most once per vertex (and $\mathcal{O}(|V|m) = \mathcal{O}(n \log b)$), this does not increase the overall complexity. We then continue by finding and processing the next short cycle. The algorithm therefore searches for a cycle at most $|E| \leq b$ times, contributing an $\mathcal{O}(b^2)$ time additive term. In the remainder we will explain the two cycle cases in more detail and prove that the time complexity for round $k$ is upper bounded by $\mathcal{O}(n \log b)$ (excluding finding the cycles). Recall we only perform $\mathcal{O}(\log b)$ rounds (after which $m = \mathcal{O}(n/b)$ and we verify all pairs naively), so the final time complexity is $\mathcal{O}(n \log^2 b + b^2 \log b)$ and the space is $\mathcal{O}(b)$.

**Cycles** We now define a lock-stepped cycle. Let the pairs $\{x_i, y_i\}$ for $i \in \{1, \ldots, \ell\}$ define a cycle in the graph of length at most $2 \log b + 1$, i.e. $v(y_i) = v(x_{i+1})$ for $1 \leq i < \ell$ and $v(y_\ell) = v(x_1)$. Let $d_i = x_i - y_{i-1}$ for $1 < i < \ell$, $d_1 = x_1 - y_\ell$ and let $\rho = \sum_{i=1}^\ell d_i$. We say that the cycle is lock-stepped iff $\rho = 0$ (and unlocked otherwise). Intuitively, lock-stepped cycles are ones where all the underlying pairs are in sync. See Figure 4.2 for an example of a lock-stepped and an unlocked cycle in the suffix implication graph of Figure 4.1. We will discuss these examples in more detail below.

Observe that the definition given is well-defined in the event of multi-edges ($\ell = 2$) and self-loops ($\ell = 1$). For example in Figure 4.1, there is
Figure 4.2: Illustrating two cases of cycles in the suffix implication graph shown in Figure 4.1. The lock-stepped cycle consists of the pairs \( \{p_5, p_{10}\}, \{p_8, p_1\} \) and \( \{p_2, p_4\} \). The unlocked cycle consists of the pairs \( \{p_9, p_3\}, \{p_2, p_4\} \) and \( \{p_5, p_{10}\} \). The symbol \( \rho \) is used between the characters that are known to be equal because the strings overlap in \( T \), and ? is used between characters that do not necessarily match.

multi-edge cycle formed by \( \{x_1, y_1\} = \{p_1, p_8\} \) and \( \{x_2, y_2\} = \{p_9, p_3\} \) where \( v(p_8) = v(p_9) = v_3 \) and \( v'(p_3) = v(p_1) = v_1 \). There is also a self-loop cycle formed by \( \{x_1, y_1\} = \{p_6, p_7\} \) where \( v(p_6) = v(p_7) = v_2 \) which conforms with the definition as \( \ell = 1 \). Note that self-loops are always unlocked since \( \rho = d_1 = x_1 - y_1 \neq 0 \) (we can assume that \( x_1 \neq y_1 \), otherwise a match is trivial).

As discussed above our algorithm proceeds by repeatedly finding a short cycle in the suffix implication graph. Let \( \{x_i, y_i\} \) for \( i = 1 \ldots \ell \) define a cycle uncovered in \( (V, E) \) by the algorithm. There are now two cases, either the cycle is lock-stepped or, by definition, it is unlocked. We now discuss the key properties we use in each case to process this cycle. We begin with the simpler, lock-stepped cycle case.

**Lock-stepped Cycles**

For a lock-stepped cycle the key property we use is given in Lemma 4.7 below. In the lemma we prove that in any lock-stepped cycle there is a particular pair \( \{x_j, y_j\} \) such that if every other pair \( \{x_i, y_i\} \) matches (i.e. when \( i \neq j \)) then the left half of \( \{x_j, y_j\} \) matches. Therefore to determine whether \( \{x_j, x_j\} \) matches we need only check whether the right half of \( \{x_j, y_j\} \) matches.
Before formally proving Lemma 4.7 we give an overview of the proof technique with reference to the example in Figure 4.2(a). The Figure 4.2(a) gives an illustration of the structure of the text underlying the cycle given by the nodes \( v_1, v_2 \) and \( v_3 \) in the suffix implication graph in Figure 4.1. More formally, it illustrates the cycle \( \{x_1, y_1\} = \{p_5, p_{10}\}, \{x_2, y_2\} = \{p_8, p_1\} \) and \( \{x_3, y_3\} = \{p_2, p_4\} \) where \( v(p_{10}) = v(p_8) = v_3, v(p_1) = v(p_2) = v_1 \) and \( v(p_4) = v(p_5) = v_2 \). Notice that in Figure 4.1 the substrings \( T_{p_8, p_8 + m - 1} \) and \( T_{p_{10}, p_{10} + m - 1} \) overlap in \( T \) and so overlap by the same amount in Figure 4.2(a). Further because they overlap in \( T \) we know that a portion of them is equal - this is indicated with \( \| \) symbols (drawn as a rotated \( = \)). Next consider the substrings \( T_{p_8, p_8 + m - 1} \) and \( T_{p_{10}, p_{10} + m - 1} \) which correspond to a pair \( \{p_8, p_1\} \) which should be verified for equality. To illustrate this we draw them with one directly above the other with ? symbols. The diagram then proceeds in this fashion for the other edges. Notice that because \( \rho = 0 \) it follows that the top substring \( T_{p_{10}, p_{10} + m - 1} \) is aligned directly above the bottom substring \( T_{p_5, p_5 + m - 1} \) and also forms a pair \( \{p_5, p_{10}\} \) to be verified.

Consider the string \( S \) (the grey box in the diagram), which is a prefix of \( T_{p_{10}, p_{10} + m - 1} \). As \( T_{p_{10}, p_{10} + m - 1} \) and \( T_{p_8, p_8 + m - 1} \) overlap in \( T \), the string \( S \) also occurs as a suffix of \( T_{p_8, p_8 + m - 1} \). Now assume (as in the Lemma below) that both \( \{p_8, p_1\} \) and \( \{p_2, p_4\} \) match. This is equivalent to saying that all the ? symbols are equal. We therefore have (as illustrated) that \( S \) occurs as a substring of \( T_{p_{10}, p_{10} + m - 1} \) as well. Continuing this argument we conclude that \( S \) is a prefix of \( T_{p_5, p_5 + m - 1} \). As we demonstrate in the proof, \( |S| > m/2 \) and thus we have, for free, that the first half of \( \{p_5, p_{10}\} \) matches. Lemma 4.7 formalises this intuition:

**Lemma 4.7** Let \( \{x_i, y_i\} \) for \( i = 1 \ldots \ell \) be the edges in a lock-stepped cycle. Further let \( j = \arg \max \sum_{i=1}^j d_i \). If \( \{x_i, y_i\} \) match for all \( i \neq j \) then \( \{x_j, y_j\} \) matches iff

\[
T_{x_j + m/2, x_j + m - 1} = T_{y_j + m/2, y_j + m - 1}.
\]

**Proof** In the following we will work with indices modulo \( \ell \), i.e. \( x_{i+1} \) is \( x_i \). As observed above, as the cycle is lock-stepped, it cannot be a self-loop and thus \( \ell \geq 2 \).

By assumption we have that \( \{x_i, y_i\} \) matches for all \( i \neq j \). This means that \( T_{x_i, x_i + m - 1} = T_{y_i, y_i + m - 1} \) for all \( i \neq j \). Let \( \gamma = \sum_{i=1}^j d_i \) and observe that as \( \rho = \sum_{i=1}^\ell d_i = 0 \), by the maximality of \( j \), we have that \( \gamma \geq 0 \). More generally we define \( \gamma_i' = \gamma - \sum_{i'=1}^i d_{i'} \). We first show that for all \( i \in \{1, 2, \ldots, \ell\} \) we have that \( \gamma_i' + d_i = \gamma_{i-1}' \). This fact will be required below.
Observe that $\gamma'_i \geq 0$ for all $i \neq j$ (or the maximality of $\gamma$ is contradicted) and $\gamma'_j = 0$. For $i > 1$, it is immediate that,

$$\gamma'_i + d_i = \left( \gamma - \sum_{i' = 1}^{i} d_{i'} \right) + d_i = \gamma - \sum_{i' = 1}^{i-1} d_{i'} = \gamma'_{i-1}.$$

For $i = 1$ we have $\gamma'_1 = \gamma - \sum_{i' = 1}^{1} d_{i'} = \gamma$ and $\gamma'_1 = \gamma - d_1$ therefore, $\gamma'_1 + d_1 = \gamma = \gamma'_1$ (which is $\gamma'_0$ as we are working with indices modulo $\ell$).

For notational simplicity let $S = T_{x_j, x_j + m - \gamma - 1}$. We will show that for all $i \in \{1, 2, \ldots, \ell\}$, there is an occurrence of $S$ in $T_{y_i, y_i + m - 1}$ starting at offset $\gamma'_i$, i.e. that $T_{y_i, y_i + \gamma'_i, y_i + \gamma'_i + m - \gamma - 1} = S$. The result then follows almost immediately as $\gamma'_j = 0$ and we will show that $\gamma \leq m/4$.

We proceed by induction on $i$ in decreasing order modulo $\ell$, starting with $i = (j - 1) \mod \ell$ and ending with $i = j \mod \ell$. That is we consider $i = (j - 1)$ down to $i = 1$ followed by $i = \ell$, down to $i = j$.

We first show that in both the base case and the inductive step there is an occurrence of $S$ in $T_{x_{i+1}, x_{i+1} + m - 1}$ starting at offset $\gamma'_{i+1}$. For the base case, $i = (j - 1) \mod \ell$, by the definition of $S$, we immediately have that $T_{x_{i+1} + \gamma'_{i+1}, x_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = S$ as $i + 1 = j$ and $\gamma'_j = 0$. For the inductive step where $i \neq (j - 1) \mod \ell$, by the inductive hypothesis we have that $T_{y_{i+1}, y_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = S$. Further as $i + 1 \neq j$, $\{x_{i+1}, y_{i+1}\}$ matches and therefore,

$$T_{x_{i+1} + \gamma'_{i+1}, x_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = T_{y_{i+1} + \gamma'_{i+1}, y_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = S.$$

Both the base case and the inductive step are now proven in an identical manner.

As the edges $\{x_i, y_i\}$ form a cycle, we have that have that $v(x_{i+1}) = v(y_i)$ and further that $x_{i+1} = y_i + d_{i+1}$. This means that $T_{y_i, y_i + m - 1}$ and $T_{x_{i+1}, x_{i+1} + m - 1}$ overlap in $T$ by $m - |d_{i+1}|$ characters. I.e. there is a substring of $T$ of length $m - |d_{i+1}|$ which is a prefix of $T_{x_{i+1}, x_{i+1} + m - 1}$ and a suffix of $T_{y_i, y_i + m - 1}$ or visa-versa (depending on the sign of $d_{i+1}$). In particular this implies that,

$$S = T_{x_{i+1} + \gamma'_{i+1}, x_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = T_{y_i + d_{i+1} + \gamma'_{i+1}, y_i + d_{i+1} + \gamma'_{i+1} + m - \gamma - 1} = T_{y_i + \gamma'_{i+1}, y_i + \gamma'_{i+1} + m - \gamma - 1}.$$

The first equality follows because $x_{i+1} = y_i + d_{i+1}$ and the second because $\gamma'_{i+1} + d_{i+1} = \gamma'_i$. This completes the inductive argument.
Finally observe that when \(i = j\), we have that \(S\) occurs in \(T_{y_j,y_j+m-1}\) starting at offset \(\gamma_j' = 0\). I.e. \(T_{x_j,x_j+m-\gamma_j-1} = T_{y_j,y_j+m-\gamma_j-1}\). As \(|d_j| < m/(9\cdot \log b)\) and further \(\ell \leq 2\log b + 1\) we have that \(\gamma = \sum_{i=1}^{\ell} d_j \leq \sum_{i=1}^{\ell} |d_i| \leq m/4\), completing the proof.

**Processing lock-stepped Cycles** We use Lemma 4.7 as follows. First we identify the edge \(\{x_j, y_j\}\). This can be achieved by calculating \(\sum_{i=1}^{\ell} d_i\) for all \(i\) by traversing the cycle in \(\mathcal{O}(\log b)\) time. We then delete this edge from the graph, breaking the cycle. However we still need to check that \(T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}\). This is achieved by inserting a new pair, \(\{x_j + m/2, y_j + m/2\}\) into the next round where \(m \leftarrow m_{k-1} = m_k/2\). Processing all lock-stepped cycles in this way takes \(\mathcal{O}(b \log b)\) time in total as we remove an edge each time.

**Unlocked Cycles**

The remaining case is when we find an unlocked cycle in the graph \((V, E)\). For an unlocked cycle, the key property is given in Lemma 4.8. This lemma is similar to the previous lemma for lock-stepped cycles in that it identifies a particular pair, \(\{x_j, y_j\}\) such that if every other pair \(\{x_i, y_i\}, i \neq j\) matches then \(\{x_j, y_j\}\) matches if and only if two conditions hold. The first condition is the same as the condition in the previous Lemma. The second condition requires that the first three-quarters of both the strings have a small period. This second condition may seem nonintuitive but, when viewed in the correct light, follows fairly straightforwardly from the fact that the cycle is unlocked.

Again, we begin with an overview of the proof technique. We focus on the forward direction, that is we assume that all \(\{x_i, y_i\}\) match (including \(i = j\)) and show that the two properties required indeed hold. The reverse direction follows from the observation that \(T_{x_j+m/2,x_j+m-1}\) contains a full period from \(T_{x_j+m/2,x_j+m-1}\). This overview is again given with reference to the example in Figure 4.2(b). This illustration is constructed in the same manner as the illustration for a lock-stepped cycle in Figure 4.2(a). However this time it illustrates the unlocked cycle \(\{p_0, p_3\}, \{p_2, p_4\} \text{ and } \{p_5, p_{10}\}\) where \(v(p_3) = v(p_2) = v_1, v(p_4) = v(p_5) = v_2\) and \(v(p_{10}) = v(p_9) = v_3\). See Chapter 4.5.1 for an explanation of the diagram.

Again consider the string \(S\), which is a prefix of \(T_{p_3,p_3+m-1}\). Assume that all three pairs \(\{p_0, p_3\}, \{p_2, p_4\} \text{ and } \{p_5, p_{10}\}\) match. Similarly to for unlocked cycles, we can then show (as is illustrated) that the string \(S\) occurs as a substring of each of \(T_{p_2,p_2+m-1}, T_{p_4,p_4+m-1}, T_{p_5,p_5+m-1}, T_{p_{10},p_{10}+m-1}\) and in particular \(T_{p_9,p_9+m-1}\). Further as (by assumption), \(T_{p_9,p_9+m-1}\) equals \(T_{p_3,p_3+m-1}\) and so
we have found two occurrences of $S$ in $T_{p_3,p_3+m-1}$. Crucially we show in the
proof that as the cycle is unlocked these are two distinct occurrences of $S$, which
are $|\rho|$ characters apart. This in turn implies that $T_{p_3,p_3+m-1}$ (and hence also
$T_{p_0,p_0+m-1}$) has a long, periodic prefix as required.

**Lemma 4.8** Let $\{x_i, y_i\}$ for $i = 1 \ldots \ell$ be the edges in an unlocked cycle
Further let $j = \arg \max \sum_j d_i$. If $\{x_i, y_i\}$ match for all $i \neq j$ then $\{x_j, y_j\}$
matches iff both the following hold:

1. $T_{x_j+m/2,x_j+m-1} = T_{y_j+m/2,y_j+m-1}$
2. $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ both have period $|\rho| \leq m/4$

**Proof** In the following we will work with indices modulo $\ell$, i.e. $x_{\ell+1}$ equals
$x_1$. We begin proving the forward direction. That is we assume that $\{x_i, y_i\}$
matches for $i = j$ (as well as for all $i \neq j$) and prove that both conditions hold.
The first condition is immediate and hence we focus on the second.

As in the proof of Lemma 4.7, let $\gamma = \sum_i d_i$ and for all $i$, let $\gamma'_i =
\gamma - \sum_{i'} d_i$. Recall that $\gamma'_j = 0$ and $\gamma'_i \geq 0$ for all $i \neq j$ (or the maximality
of $\gamma$ is contradicted). For $i > 1$, it is easily check that, as in the proof of
Lemma 4.7, $\gamma'_i + d_i = \gamma'_{i-1}$. However, unlike in Lemma 4.7, we do not have
that $\gamma'_1 + d_1 = \gamma'_\ell$. In Lemma 4.7 this followed because $\delta = \sum_{i=1}^\ell d_i = 0$
which is not true here. The first portion of this proof is similar to the proof
of Lemma 4.7 but with some small modifications.

We will begin by showing that for all $i \in \{1, 2, \ldots, \ell\}$, $S = T_{x_i,x_i+m-1}$
ocurs in each $T_{x_i,x_i+m-1}$ at offset $\gamma_i$ i.e. that the string $T_{x_i+x_i+\gamma_i+x_i+\gamma_i-1} =
S$. We first proceed by induction on $i$ in decreasing order, starting with $i = j$
and halting with $i = 1$. The base case, $i = j$ is immediate as $\gamma'_j = 0$.

We now assume by the inductive hypothesis that $T_{x_i+x_i+\gamma_i+x_i+\gamma_i+m-1} =
S$. As the edges $\{x_i, y_i\}$ form a cycle, we have that $v(x_i) = v(y_{i-1})$ for all $i$.

By the definition of $v(x_i)$ and $v(y_{i-1})$ we have that $d_i = x_i - y_{i-1}$. In other
words, $T_{y_{i-1},y_{i-1}+m-1}$ and $T_{x_i,x_i+m-1}$ overlap in $T$ by $m - |d_i|$ characters.

Analogously to in the proof of Lemma 4.7, this implies an occurrence of $S$ in
$T_{y_{i-1},y_{i-1}+m-1}$ starting at $\gamma'_i + d_i = \gamma'_{i-1} \geq 0$. Finally observe that for all $i$,
we have that $T_{y_{i-1},y_{i-1}+m-1} = T_{x_{i-1},x_{i-1}+m-1}$ so $S$ occurs in $T_{x_{i-1},x_{i-1}+m-1}$
starting at $\gamma'_i$, completing the inductive case.

We now repeat the induction on $i$ in increasing order, starting with $i = j$
and halting with $i = \ell$. We now assume by the inductive hypothesis that
$T_{x_i+x_i+\gamma_i+x_i+\gamma_i+m-1} = S$. The argument is almost identical but in reverse.

We provide the argument for completeness.
We have that $T_{y_i,y_i+m-1} = T_{x_i,x_i+m-1}$ so by the inductive hypothesis, $S$ occurs in $T_{y_i,y_i+m-1}$, at offset $\gamma_i$. As the edges $\{x_i,y_i\}$ form a cycle, we have that $v(x_{i+1}) = v(y_i)$ for all $i$. As $d_{i+1} = x_{i+1} - y_i$, the substrings $T_{y_i,y_i+m-1}$ and $T_{x_{i+1},x_{i+1}+m-1}$ overlap in $T$ by $m - |d_{i+1}|$ characters. Again, this implies an occurrence of $S$ in $T_{x_{i+1},x_{i+1}+m-1}$ starting at $\gamma_{i+1}' - d_{i+1} = \gamma_{i+1}' + 0$. Finally observe that for all $i$, we have that $T_{y_{i+1},y_{i+1}+m-1} = T_{x_{i+1},x_{i+1}+m-1}$ so $S$ occurs in $T_{x_{i+1},x_{i+1}+m-1}$ starting at $\gamma_i'$, completing the inductive case.

We now have that $S = T_{x_j,x_j+m-\gamma-1}$ occurs in each $T_{x_i,x_i+m-1}$ at offset $\gamma_i$. In particular $S$ occurs in $T_{y_i,y_i+m-1}$ at offset $\gamma_i' = \gamma_i'$ - that is, starting at $x_1 + \gamma_i'$ in $T$. There is also an occurrence of $S$ in $T_{x_\ell,x_\ell+m-1}$ at offset $\gamma_{\ell}'$ - that is, starting at $x_\ell + \gamma_{\ell}'$ in $T$. However we have that $\{x_\ell,y_i\}$ form a cycle, we have that $v(x_\ell) = v(y_i)$ and hence $d_\ell = x_\ell - y_i$. These two occurrences are therefore $\{(x_\ell + \gamma_{\ell}') - (x_\ell + \gamma_{i}')\} = |d_\ell| = 1$ characters apart in $T$. Therefore $S$ has period $|\rho|$. As $|d_j| < m/(9 \cdot \log b)$ and further $\ell \leq 2 \log b + 1$ we have that $\gamma = \sum_{i\leq j} d_j \leq \sum_{i=1}^\ell |d_i| = \rho \leq m/4$. In conclusion, $S$ has period $|\rho| \leq m/4$, length at least $3m/4$ and occurs at the start of $T_{x_j,x_j+m-1}$ (and $T_{y_j,y_j+m-1}$) as required.

We now prove the reverse direction. That is that if both conditions hold that $\{x_j,y_j\}$ matches. By condition 2, both $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ are periodic with period at most $m/4$. Further, by condition 1 we have that $T_{x_j,x_j+m/2,x_j+m-1} = T_{y_j,y_j+m/2,y_j+m-1}$. Observe that $T_{x_j,x_j+m/2,x_j+m-1}$ contains at least a full period of characters from $T_{x_j,x_j+3m/4-1}$, and similarly with $T_{y_j,y_j+m/2,y_j+m-1}$ and $T_{y_j,y_j+3m/4-1}$ analogously. In other words, the first full period of $T_{x_j,x_j+3m/4-1}$ matches the first full period of $T_{y_j,y_j+3m/4-1}$. By the definition of periodicity we have that $T_{x_j,x_j+3m/4-1} = T_{y_j,y_j+3m/4-1}$ and hence that $T_{x_j,x_j+m-1} = T_{y_j,y_j+m-1}$, i.e. $\{x_j,x_j\}$ matches.

**Processing unlocked cycles** We can again identify edge $\{x_j,y_j\}$ as well as $\rho$ in $O(\log b)$ time by inspecting the cycle. This follows immediately from the statement of the lemma and the definition of $\rho$. We again delete the pair, $\{x_j,y_j\}$ (along with the edge in the graph) and insert a new pair, $\{x_{k+1},y_{k+1}\}$ into the next round where $m \leftarrow m_{k-1} = m/2$. This checks the first property.

We also need to check the second property, i.e. that both strings $T_{x_j,x_j+3m/4-1}$ and $T_{y_j,y_j+3m/4-1}$ have $|\rho|$ as a period. We do not immediately check the periodicity, we instead delay computation until the end of round $k$, after all cycles have been processed. At the current point in the algorithm, we simply add the tuple $(\{x,y\}, \rho)$ to a list, $\Pi$ of text substrings to be checked later for periodicity. This list uses $O(b)$ space as at most $b$ edges are considered. Excluding checking for periodicity, processing all unlocked cycles takes $O(b \log b)$ time in total.
4.6 Time-Space Tradeoffs for Batched LCP Queries

We provide an overview of the techniques used to obtain the time-space tradeoff for the batched LCP process, as it closely follows those of Chapter 4.3. In Chapter 4.3 the algorithm simulates concurrent binary searches in order to determine the LCP of each input pair (with some extra work at the end). The idea for obtaining the tradeoff is to generalize the binary search to an \( \alpha \)-ary search. So in the \( k^{th} \) round the input is a set of \( q \) pairs denoted by \( P_k \), where
we are guaranteed that for any \((i, j) \in P_k\), 
\(LCP(i, j) \leq 2^\log n - (k-1) \log \alpha\), and the goal of the \(k\)th iteration is to decide for any \((i, j) \in P_k\) if 
\(LCP(i, j) \leq 2^\log n - k \log \alpha\) or not. From a space perspective, this means we need 
\(O(\alpha q)\) space in order to compute \(\alpha\) fingerprints for each index in any \((i, j) \in P_k\). From a 
time perspective, we only need to perform \(O(\log \alpha q)\) rounds before we may 
begin the final round. However, each round now costs \(O(n + \alpha q)\), so we have the 
following trade-off.

**Theorem 4.4** Let \(2 \leq \alpha \leq n\). There exists a randomized Monte-Carlo 
algorithm that with high probability correctly answers a batch of \(q\) LCP 
queries on suffixes from a string of length \(n\). The algorithm uses 
\(O((n + \alpha q) \log \alpha q)\) time and \(O(\alpha q)\) space in the worst case.

In particular, for \(\alpha = 2\), we obtain Theorem 4.1 as a corollary. Consequently, the 
total time cost for constructing the sparse suffix tree in \(O(ab)\) space becomes

\[
O \left( \frac{n \log^2 b}{\log \alpha} + \frac{\alpha b \log^2 b}{\log \alpha} \right).
\]

If, for example, \(\alpha = b^\varepsilon\) for a small constant \(\varepsilon > 0\), the cost for constructing the 
sparse suffix tree becomes \(O\left(\frac{1}{\varepsilon}(n \log b + b^{1+\varepsilon} \log b)\right)\), using \(O(b^{1+\varepsilon})\) words of 
space. Finally by minimizing with the standard \(O(n)\) time, \(O(n)\) space algorithm we 
achieve the stated result of \(O(n \log b)\) time, using \(O(b^{1+\varepsilon})\) space.

4.7 Sparse Position Heaps

So far our focus has been on sparse suffix trees and arrays. In this section, we 
consider another sparse index, the sparse position heap, and show that it can 
be constructed much faster than the sparse suffix tree or array. However, the 
faster construction time comes at the cost of slower pattern matching queries.

4.7.1 Position Heaps

We start by reviewing position heaps. The position heap \(H_T\) over a text \(T_{1,n}\) is 
a blend of a trie over \(T\)'s suffixes and a heap over its indices [53]:

- The nodes are exactly the \(n\) indices (positions) from \(T\) such that the 
  *max-heap property* is satisfied: a node \(i\) is larger than all nodes below \(i\).
- The edges are labeled with single letters from \(T\) as in a usual trie such 
  that the following constraint is satisfied: for every node \(i\), the letters on 
  the root-to-\(i\) path are a *prefix* of the suffix \(T^i\).
See Figure 4.3 for an example. This definition almost directly results in the following *naive construction algorithm*: start with a tree consisting of a single node $n$ only. Now assume that $\mathcal{H}_{i+1}^T$, the position heap for $T_{i+1,n}$ for some $i < n$, has already been constructed. Then $\mathcal{H}_i^T$ is obtained by first matching $T_{i}^i$ in $\mathcal{H}_{i+1}^T$ as long as possible, thereby finding the longest prefix $T_{i,j}$ of $T_{i}$ that is a root-to-node path in $\mathcal{H}_{i+1}^T$. Then a new node $i$ is appended as a child to the node $h$ where the search ended, and the new edge $(h, i)$ is labeled with letter $t_{j+1}$. In the end, $\mathcal{H}_1^T$ is the desired result $\mathcal{H}_T$.

Finding all $k$ occurrences of a pattern $P_{1,m}$ using $\mathcal{H}_T$ works as follows: first try matching $P$ as long as possible in $\mathcal{H}_T$, thereby finding the longest prefix $P_{1,j}$ of $P$ that is a root-to-node path $i_1 \rightarrow i_2 \rightarrow \cdots \rightarrow i_j$ in $\mathcal{H}_T$. The indices $\{i_1, \ldots, i_j\}$ are potential candidates for matches of $P$; since $j \leq m$, they can be checked naively (meaning one-by-one character comparisons between $P$ and the text) in total time $O(m^2)$. Further, if $j = m$ (the pattern has been fully matched), then all nodes below $i_j$ match $P$ for sure; they can be returned by a simple traversal of the subtree below $i_j$. The total time is $O(m^2 + k)$.

The position heap can also be enhanced with *maximal reach pointers* that allow for optimal $O(m + k)$ search time; however, we do not review this technique here, since it does not seem to generalize for a sparse set of positions.

The definition suggests that position heaps should be “easier” to compute than suffix trees and suffix arrays, since they do not sort the entire suffixes, but still allow for fast pattern matching queries. This is particularly evident when only a sparse set $I$ of $b$ suffixes is to be indexed and those $b$ suffixes are inserted

![Diagram](image)

**Figure 4.3**: (a) Position heap for the text $T_{1,16} = \texttt{a b b a a b a b a b a b a b a b b a}$
(b) Sparse position heap for positions $I = \{1, 2, 4, 5, 10, 11, 13, 16\}$ in $T$. 
using the naive construction algorithm above. Since the time of inserting a suffix $i \in I$ can be upper bounded by the number of indexed suffixes to the right of $i$, the running time of the naive algorithm is $O(b^2)$, in particular optimal $O(n)$ for $b = O(\sqrt{n})$. Recall that the naive suffix tree insertion algorithm, on the other hand, would result in $O(nb)$ time, which is never linear unless $b = O(1)$.

Hence, from now on we can assume $n/b \leq b$.

### 4.7.2 A Monte-Carlo Construction Algorithm

Our basis is the naive construction algorithm sketched in Chapter 4.7.1: we scan $T$ from right to left, and whenever we encounter an indexed suffix from $I$, we insert it into the already existing heap to obtain the new one. To formalize this, assume that $I$ is given as a sorted array $I[1, b]$. Assume that $H_{k+1}^T$, the sparse position heap for suffixes $I[k+1, b]$, has already been constructed. Let $i = I[k]$ be the next position in $I$. Our aim is to insert the new suffix $T^i$ and thereby obtain $H_k^T$, faster than in $O(b)$ time.

Recall that the task upon insertion of $T^i$ is to find the longest prefix $T_{i,j}$ of $T^i$ that is already present in $H_{k+1}^T$. Now instead of matching the suffix from start to end, we would like to binary search over all possible prefixes of $T^i$ to find the longest prefix that already exists.\(^1\) If such an existential test could be done in $O(1)$ time and since the longest possible such prefix is of length $O(b)$, the binary search would run in $O(\log b)$ time. This would result in $O(b \log b)$ running time.

The problem is to check (in constant time) if and where the prefixes $T_{i,i+\ell}$ occur in $H_{k+1}^T$. This is where the fingerprints come into play, again. First, we use a hash table that stores the nodes of $H_{k+1}^T$ and is indexed by the fingerprints of the respective root-to-node paths. Now assuming that we have the fingerprint for some prefix $T_{i,i+\ell}$, then we could easily check the existence of a node representing the same string in $O(1)$ time. If, finally, $T_{i,j}$ is the longest such prefix and $i$ is inserted into $H_{k+1}^T$ as a child of $h$, we can also insert $i$ into the hash table (using the fingerprint $FP[i, j]$).

The remaining problem is how to compute the fingerprints. Storing the values $FP[1, i]$ for all $1 \leq i \leq n$ would do the trick, but is prohibitive due to the extra space of $O(n)$. Instead, we do the following indirection step: in a preprocessing step, store only the values $FP[1, i]$ at $b$ regularly spaced positions $i \in \{n/b, 2n/b, \ldots\}$; this takes $O(b)$ space and $O(n)$ overall time. We also precompute all powers $r^\ell \mod p$ for $x \leq b$ in $O(b)$ time and space ($r$ and $p$ are

\(^1\)This can be regarded as an x-fast trie-like search [149].
the constants needed for the fingerprints, see Chapter 4.2). Then finding the correct node $h$ in $H_{T}^{k+1}$ to which $i$ should be appended amounts to the following steps (see Figure 4.4):

1. Compute $FP[1, i - 1]$, by scanning from $i$ backwards until finding the next multiple $j$ of $n/b$ to the left of $i$ (for which $FP[1, j]$ has been precomputed), and using the formula

   $$FP[1, i - 1] = FP[1, j] \cdot r^{i-j-1} + FP[j + 1, i - 1] \mod p.$$  

   Note that $i - j \leq n/b \leq b$, hence all necessary powers of $r$ are precomputed.

2. Perform a binary search over the prefixes $T_{i, i + \ell}$, for $i + \ell$ being a multiple of $n/b$. Using the result from step (1) and the precomputed fingerprints, the desired fingerprints can be computed in $O(1)$ time by

   $$FP[i, i + \ell] = FP[1, i + \ell] - FP[1, i - 1] \cdot r^{\ell} \mod p,$$

   and hence the binary search takes $O(\log b)$ time (note again $\ell \leq b$).

3. Let $h'$ be the node where the binary search ended with prefix $T_{i, j'}$. From $h'$, continue matching $t_{j'+1}, t_{j'+2}, \ldots$ in the trie until no further match is possible. This yields node $h$, the longest prefix of $T^n$ that is present in $H_{T}^{k+1}$.

   The time for steps (1) and (3) is $O(n/b)$. Since there are $b$ suffixes to be inserted, these steps take overall $O(n)$ time. The time for step (2) is $O(b \log b)$ in total. The fingerprint needed for the insertion of $i$ into the hash table can be either computed along with step (3) in $O(n/b)$ time, or from the fingerprint of the parent node $h$ in constant time. The claim follows.
**Theorem 4.5** There exists a randomized Monte-Carlo algorithm that with high probability correctly constructs the sparse position heap on $b$ suffixes from a string of length $n$. The algorithm uses $O(n + b \log b)$ time and $O(b)$ space in the worst case, and finding the $k$ occurrences of any pattern of length $m$ takes $O(m^2 + k)$ time.

### 4.8 Conclusions

The main open problem in sparse text indexing is whether we can obtain complexity bounds that completely generalise those of full text indexing. More specifically, is it possible to construct a sparse text index for arbitrary positions in $O(b)$ space and $O(n)$ time (for integer alphabets) that support pattern matching queries in $O(m + k)$ time, where $m$ is the length of the pattern and $k$ is the number of occurrences?

In this paper we have shown an $O(n \log^2 b)$ time construction algorithm for sparse suffix trees and arrays. This makes significant progress towards the desired $O(n)$ bound, but closing the generalisation gap entirely remains an open question.

As an intermediate step, it might be advantageous to consider trade-offs between the construction time and space as well as the query time of the index. In this context we showed that the sparse position heap can be constructed in $O(n + b \log b)$ time while supporting pattern matching queries in $O(m^2 + k)$ time. This indicates that relaxing the query time constraint makes the problem more approachable, and thus it might be possible to construct a sparse text index in $O(n)$ time and $O(b)$ space for slower pattern matching queries.

Fingerprints play a fundamental role in our results, and it would be interesting if this technique can be further improved, e.g. by constructing a faster deterministic verifier for batched LCP queries. However, it would perhaps be of even greater interest if deterministic solutions similar to the well-known suffix tree and suffix array construction algorithms exist.
CHAPTER 5

TIME-SPACE TRADE-OFFS FOR THE LONGEST COMMON SUBSTRING PROBLEM

Tatiana Starikovskaya* Hjalte Wedel Vildhøj†

* Moscow State University, Department of Mechanics and Mathematics
† Technical University of Denmark, DTU Compute

Abstract

The Longest Common Substring problem is to compute the longest substring which occurs in at least \( d \geq 2 \) of \( m \) strings of total length \( n \). In this paper we ask the question whether this problem allows a deterministic time-space trade-off using \( O(n^{1+\varepsilon}) \) time and \( O(n^{1-\varepsilon}) \) space for \( 0 \leq \varepsilon \leq 1 \). We give a positive answer in the case of two strings \( (d = m = 2) \) and \( 0 < \varepsilon \leq 1/3 \). In the general case where \( 2 \leq d \leq m \), we show that the problem can be solved in \( O(n^{1-\varepsilon}) \) space and \( O(n^{1+\varepsilon} \log^2 n (d \log^2 n + d^2)) \) time for any \( 0 \leq \varepsilon < 1/3 \).

5.1 Introduction

The Longest Common Substring (LCS) Problem is among the fundamental and classic problems in combinatorial pattern matching [73]. Given two strings \( T_1 \) and \( T_2 \) of total length \( n \), this is the problem of finding the longest substring that occurs in both strings. In 1970 Knuth conjectured that it was not possible to solve the problem in linear time [102], but today it is well-known that the LCS
can be found in $O(n)$ time by constructing and traversing a suffix tree for $T_1$ and $T_2$ [73]. However, obtaining linear time comes at the cost of using $\Theta(n)$ space, which in real-world applications might be infeasible.

In this paper we explore solutions to the LCS problem that achieve sublinear, i.e., $o(n)$, space usage\(^1\) at the expense of using superlinear time. For example, our results imply that the LCS of two strings can be found deterministically in $O(n^{4/3})$ time while using only $O(n^{2/3})$ space. We will also study the time-space trade-offs for the more general version of the LCS problem, where we are given $m$ strings $T_1, T_2, \ldots, T_m$ of total length $n$, and the goal is to find the longest common substring that occurs in at least $d$ of these strings, $2 \leq d \leq m$.

5.1.1 Known Solutions

For $m = d = 2$ the LCS is the longest common prefix between any pair of suffixes from $T_1$ and $T_2$. Naively comparing all pairs leads to an $O(n^2|\text{LCS}|)$ time and $O(1)$ space solution, where $|\text{LCS}|$ denotes the length of the LCS. As already mentioned we can also find the LCS in $O(n)$ time and space by finding the deepest node in the suffix tree that has a suffix from both $T_1$ and $T_2$ in its subtree. Alternatively, we can build a data structure that for any pair of suffixes can be queried for the value of their longest common prefix. Building such a data structure is known as the Longest Common Extension (LCE) Problem and it has several known solutions [26, 86]. If a data structure for a string of length $n$ with query time $q(n)$ and space usage $s(n)$ can be built in time $p(n)$, then this implies a solution for the LCS problem using $O(q(n)n^2 + p(n))$ time and $O(s(n))$ space. For example using the deterministic data structure of Bille et al. [26], the LCS problem can be solved in $O(n^{2(1+\varepsilon)})$ time and $O(n^{1-\varepsilon})$ space for any $0 \leq \varepsilon \leq 1/2$.

In the general case where $2 \leq d \leq m$, the LCS can still be found in $O(n)$ time and space using the suffix tree approach. Using Rabin-Karp fingerprints [97] we can also obtain an efficient randomised algorithm using sublinear space. The algorithm is based on the following useful trick: Suppose that we have an efficient algorithm for deciding if there is a substring of length $i$ that occurs in at least $d$ of the $m$ strings. Moreover, assume that the algorithm outputs such a string of length $i$ if it exists. Then we can find the LCS by repeating the algorithm $O(\log |\text{LCS}|)$ times in an exponential search for the maximum value of $i$. To determine if there is a substring of length $i$ that occurs in at least $d$ strings, we start by checking if any of the $n^{1-\varepsilon}$ first substrings of length $i$ occurs

\(^1\)We assume the input is in read-only memory and not counted in the space usage.
at least $d$ times. We can check this efficiently by storing their fingerprints in a hash table and sliding a window of length $i$ over the strings $T_j$, $j = 1, \ldots, m$. For each substring we look up its fingerprint in the hash table and increment an associated counter if it is the first time we see this fingerprint in $T_j$. If at any time a counter exceeds $d$, we stop and output the window. In this way we can check all $i$ length substrings in $\mathcal{O}(n^c)$ rounds each taking time $\mathcal{O}(n)$. Thus, this gives a Monte Carlo algorithm for the general LCS problem using $\mathcal{O}(n^{1+c} \log |\text{LCS}|)$ time and $\mathcal{O}(n^{1-c})$ space for all $0 \leq c \leq 1$. From the properties of fingerprinting we know that the algorithm succeeds with high probability. The algorithm can also be turned into a Las Vegas algorithm by verifying that the fingerprinting function is collision free in $\mathcal{O}(n^2)$ time. Table 5.1 summarises the solutions.

5.1.2 Our Results

We show the following main result:

**Theorem 5.1** Given $m$ strings $T_1, T_2, \ldots, T_m$ of total length $n$, an integer $2 \leq d \leq m$ and a trade-off parameter $\varepsilon$, the longest common substring that occurs in at least $d$ of the $m$ strings can be found in
(i) $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time for $d=m=2$ and $0<\varepsilon \leq \frac{1}{3}$, or in

(ii) $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon} \log^2 n(d \log n + d^2))$ time for $2 \leq d \leq m$, $0 \leq \varepsilon < \frac{1}{3}$.

The main innovation in these results is that they are both deterministic. Moreover, our first solution improves over the randomised fingerprinting trade-off by removing the $\log |\text{LCS}|$ factor. The basis of both solutions is a sparse suffix array determining the lexicographic order on $O(n^{1-\varepsilon})$ suffixes sampled from the strings $T_1, T_2, \ldots, T_m$ using difference covers.

5.2 Preliminaries

Throughout the paper all logarithms are base 2, and positions in strings are numbered from 1. Notation $T[i..j]$ stands for a substring $T[i]T[i+1] \cdots T[j]$ of $T$, and $T[i..]$ denotes the suffix of $T$ starting at position $i$. The longest common prefix of strings $T_1$ and $T_2$ is denoted by $lcp(T_1, T_2)$.

5.2.1 Suffix Trees

We assume a basic knowledge of suffix trees. In order to traverse and construct suffix trees in linear time and space, we will assume that the size of the alphabet is constant. Thus, the suffix tree for a set of strings $S$, denoted $ST(S)$, together with suffix links, can be built in $O(n)$ time and space, where $n$ is the total length of strings in $S$ [73]. We remind that a suffix link of a node labelled by a string $\ell$ points to the node labelled by $\ell[2..]$ and that suffix links exist for all inner nodes of a suffix tree. We need the following lemma:

**Lemma 5.1** Let $ST(S)$ be the suffix tree for a set of strings $S$, and $A$ be a set of all nodes (explicit or implicit) of $ST(S)$ labelled by substrings of another string $T$. I.e., the labels of the nodes in $A$ are exactly all common substrings of $T$ and strings from $S$. Then $ST(S)$ can be traversed in $O(|T|)$ time so that

(i) All nodes visited during the traversal will belong to $A$, and

(ii) Every node in $A$ will have at least one visited descendant.
Proof We first explain how the tree is traversed. We traverse $ST(S)$ with $T$ starting at the root. If a mismatch occurs or the end of $T$ is reached at a node $v$ (either explicit or implicit) labelled by a string $\ell$ we first jump to a node $v'$ labelled by $\ell[2..]$. We do that in three steps: 1) walk up to the higher end $u$ of the edge $v$ belongs to; 2) follow the suffix link from $u$ to a node $u'$; 3) descend from $u'$ to $v'$ comparing only the first characters of the labels of the edges with the corresponding characters of $\ell[2..]$ in $O(1)$ time. Then we proceed the traversal from the position of $T$ at which the mismatch occurred. The traversal will end at the root of the suffix tree.

All nodes visited during the traversal are labelled by substrings of $T$, and thus belong to $A$. For each $i$ the traversal visits the deepest node of $ST(S)$ labelled by a prefix of $T[i..]$. Hence, conditions (i) and (ii) of the lemma hold.

We now estimate the running time. Obviously, the number of successful matches is no more than $|T|$. We estimate the number of operations made due to unsuccessful matches by amortised analysis. During the traversal we follow at most $|T|$ suffix links and each time the depth of the current node decreases by at most one [73]. Hence, the number of up-walks is also bounded by $|T|$. Each up-walk decreases the current node-depth by one as well. On the contrary, traversal of an edge at step 3) increases the current node-depth by one. Since the maximal depth of a node visited by the traversal is at most $|T|$, the total number of down-walks is $O(|T|)$. □

5.2.2 Difference Cover Sparse Suffix Arrays

A difference cover modulo $\tau$ is a set of integers $DC_\tau \subseteq \{0, 1, \ldots, \tau - 1\}$ which for any $i, j$ contains two elements $i', j'$ such that $j - i \equiv j' - i' \pmod{\tau}$. For any $\tau$ a difference cover $DC_\tau$ of size at most $\sqrt{1.5\tau + 6}$ can be computed in $O(\sqrt{\tau})$ time [39]. Note that this size is optimal to within constant factors, since any difference cover modulo $\tau$ must contain at least $\sqrt{\tau}$ elements.

For a string $T$ of length $n$ and a fixed difference cover modulo $\tau$, $DC_\tau$, we define a difference cover sample $DC_\tau(T)$ as the subset of $T$’s positions that are in the difference cover modulo $\tau$, i.e.,

$$DC_\tau(T) = \{i \mid 1 \leq i \leq n \land i \mod \tau \in DC_\tau\}.$$

The following lemma captures two important properties of difference cover samples that we will use throughout the paper. The proof follows immediately from the above definitions.
We will consider difference cover samples of the string \( T \) ending in \( \tau \) and denote by \( DC_\tau \) the reversed blocks, and the latter stores the reversed blocks. The first contains the sampled positions sorted according to the lexicographic ordering of the reversed blocks, and the latter stores the sampled suffixes, sorted lexicographically. Similarly, we define the difference cover sparse \( LCP \) array, denoted \( LCP_\tau \), as the array storing the longest common prefix (lcp) values of neighbouring suffixes in \( SA_\tau \). As for the sampled suffixes, we define arrays \( SA_\tau^R \) and \( LCP_\tau^R \) are in \( DC_\tau(T) \) and \( LCP_\tau^R \) are shown. Sampled positions in \( T_1 \) and \( T_2 \) are marked by white and black dots, respectively.

**Lemma 5.2** The size of \( DC_\tau(T) \) is \( O(n/\sqrt{\tau}) \), and for any pair \( p_1, p_2 \) of positions in \( T \) there is an integer \( 0 \leq i < \tau \) such that both \( (p_1 + i) \) and \( (p_2 + i) \) are in \( DC_\tau(T) \).

We will consider difference cover samples of the string \( T = T_1 \$_1 T_2 \$_2 \cdots T_k \$_k \), i.e., the string obtained by concatenating and delimiting the input strings with unique characters \( \$_1, \ldots, \$_k \). See Figure 5.1 for an example of a difference cover sample of two input strings.

The difference cover sparse suffix array, denoted \( SA_\tau \) is the suffix array restricted to the positions of \( T \) sampled by the difference cover, i.e., it is an array of length \( n/\sqrt{\tau} \) containing the positions of the sampled suffixes, sorted lexicographically. Similarly, we define the difference cover sparse \( LCP \) array, denoted \( LCP_\tau \), as the array storing the longest common prefix (lcp) values of neighbouring suffixes in \( SA_\tau \). Moreover, for a sampled position \( p \in DC_\tau(T) \) we denote by \( RB(p) \) the reversed substring of length \( \tau \) ending in \( p \), i.e., \( RB(p) = T[p]T[p-1] \cdots T[p-\tau+1] \), and we refer to this string as the reversed block ending in \( p \). As for the sampled suffixes, we define arrays \( SA_\tau^R \) and \( LCP_\tau^R \) for the reversed blocks. The first contains the sampled positions sorted according to the lexicographic ordering of the reversed blocks, and the latter stores the sampled suffixes, sorted lexicographically.
corresponding longest common prefix values. See Figure 5.1 for an example of the arrays $SA_\tau$, $LCP_\tau$, $SA^R_\tau$ and $LCP^R_\tau$.

The four arrays can be constructed in $O(n\sqrt{\tau} + (n/\sqrt{\tau}) \log(n/\sqrt{\tau}))$ time and $O(n/\sqrt{\tau})$ space [34, 135]. To be able to compute the longest common prefix between pairs of sampled suffixes and pairs of reversed blocks in constant time, we use the well-known technique of constructing a linear space range minimum query data structure [21, 63] for the arrays $LCP_\tau$ and $LCP^R_\tau$.

5.3 Longest Common Substring of Two Strings

In this section we prove Theorem 5.1(i). We do so by providing two algorithms both using $O(n/\sqrt{\tau})$ space which are then combined to obtain the desired trade-off. The first one correctly computes the LCS if it has length at least $\tau$, while the second one works if the length of the LCS is less than $\tau$. In the second algorithm we must assume that $\tau \leq n^{2/3}$, which translates into the $\varepsilon \leq 1/3$ bound on the trade-off interval.

5.3.1 A Solution for Long LCS

We first compute a difference cover sample with parameter $\tau$ for the string $T = T_1 \$ \_1 T_2 \$ \_2$, where \$ \_1, \$ \_2 are special characters that do not occur in $T_1$ or $T_2$. We then construct the arrays and the range minimum query data structures described in Chapter 5.2.2 for computing longest common prefixes between pairs of sampled suffixes or pairs of reversed blocks in constant time.

The LCS is the longest common prefix of suffixes $T[p_1\ldots]$ and $T[p_2\ldots]$ for some $p_1 \leq |T_1|$ and $p_2 > |T_1| + 1$. If $|LCS| \geq \tau$ then from the property of difference cover samples (Lemma 5.2) it follows that there is an integer $r < \tau$ such that $p_1' = p_1 + r$ and $p_2' = p_2 + r$ are both in $DC_\tau(T)$, and the length of the LCS is thus $r + \text{lcp}(T[p_1\ldots],T[p_2\ldots]) - 1$. In particular, this implies that

$$|LCS| = \max_{\begin{array}{c} p_1' \leq |T_1| \\ p_2' > |T_1| + 1 \end{array}} \left( \text{lcp}(RB(p_1'), RB(p_2')) + \text{lcp}(T[p_1\ldots], T[p_2\ldots]) - 1 \right).$$

The $-1$ is necessary since a sampled suffix overlaps with the reversed block in one position. We will find the LCS by computing a pair of sampled positions $p_1^* \leq |T_1|, p_2^* > |T_1| + 1$ that maximises the above expression. Obviously, this can be done by performing two constant time range minimum queries for all $O((n/\sqrt{\tau})^2)$ pairs of sampled positions, but we want to do better.
We then scan where \( \ell \) is the rank of a reversed block
simultaneously. To do so, we first allocate an array
maximum. See Figure 5.2 for an example.

The candidate pairs with a longest common prefix of length at least
are located in disjoint intervals \( I_1, I_2, \ldots, I_k \) of \( SA_\tau \). We compute these intervals
by scanning \( LCP_\tau \) to identify the maximal contiguous ranges with lcp values
greater than or equal to \( \ell \). For each interval \( I_j \), we will find a pair \( p'_1 \leq |T_1|, p'_2 > |T_1| + 1 \) such that the length of the longest common prefix
of \( T[p'_1..] \) and \( T[p'_2..] \) is at least \( \ell = \ell_{\text{max}} - i \). Among these pairs we select the
one maximising \( \text{lcp}(RB(p'_1), RB(p'_2)) \).

The candidate pairs with a longest common prefix of length at least \( \ell \) are
located in disjoint intervals \( I_1, I_2, \ldots, I_k \) of \( SA_\tau \). We compute these intervals
by scanning \( LCP_\tau \) to identify the maximal contiguous ranges with lcp values
greater than or equal to \( \ell \). For each interval \( I_j \), we find a pair \( p'_1 \leq |T_1|, p'_2 > |T_1| + 1 \) in \( I_j \) that maximises \( \text{lcp}(RB(p'_1), RB(p'_2)) \). If \( \text{lcp}(RB(p'_1), RB(p'_2)) + \ell - 1 \)
is greater than the maximum value seen so far, we store this value as the new
maximum. See Figure 5.2 for an example.

Instead of searching the \( k \) intervals one by one, we process all intervals
simultaneously. To do so, we first allocate an array \( A \) of size \( n/\sqrt{\tau} \) and if \( r \)
is the rank of a reversed block \( RB(p) \), \( p \in I_j \), we set \( A[r] \) to be equal to \( j \).
We then scan \( A \) once and compute the longest common prefixes of every two
consecutive reversed blocks ending at positions \( p'_1 \leq |T_1|, p'_2 > |T_1| + 1 \) from
the same interval. We can do this if we for each interval \( I_j \) keep track of the
rightmost \( r \) such that \( A[r] = j \).

The intervals considered in each round are disjoint so each round takes
\( O(n/\sqrt{\tau}) \) time and never uses more than \( O(n/\sqrt{\tau}) \) space. The total time is
\( O(n/\sqrt{\tau}) \) in addition to the \( O(n/\sqrt{\tau} + (n/\sqrt{\tau}) \log(n/\sqrt{\tau})) \) time for the construction.
Hence we have showed the following lemma:

**Lemma 5.3** Let \( 1 \leq \tau \leq n \). If the length of the longest common substring
of \( T_1 \) and \( T_2 \) is at least \( \tau \), it can be computed in \( O(n/\sqrt{\tau}) \) space and \( O(n/\sqrt{\tau} + (n/\sqrt{\tau}) \log n) \) time, where \( n \) is the total length of \( T_1 \) and \( T_2 \).
5.3.2 A Solution for Short LCS

In the following we require that \( \tau \leq n^{2/3} \), or, equivalently, that \( \tau \leq n/\sqrt{\tau} \).

Let us assume, for simplicity, that \( n_1 = |T_1| \) is a multiple of \( \tau \). Note that if \( |LCS| \leq \tau \) then the LCS is a substring of one of the following strings: \( T_1[1..2\tau], T_1[\tau + 1..3\tau], \ldots, T_1[n_1 - 2\tau + 1..n_1] \). Therefore, we can reduce the problem of computing the LCS to the problem of computing the longest substring of \( T_2 \) which occurs in at least one of these strings.

We divide the set \( S = \{ T_1[1..2\tau], T_1[\tau + 1..3\tau], \ldots, T_1[n_1 - 2\tau + 1..n_1] \} \) into disjoint subsets \( S_i \), \( i = 1, \ldots, \sqrt{\tau} \), such that the total length of strings in \( S_i \) is no more than \( 2n/\sqrt{\tau} \). For each \( S_i \) we compute the longest substring \( t^*_i \) of \( T_2 \) which occurs in one of the strings in \( S_i \), and take the one of the maximal length.

To compute \( t^*_i \) for \( S_i \) we build the generalised suffix tree \( ST(S_i) \) for the strings in \( S_i \). We traverse \( ST(S_i) \) with \( T_2 \) as described in Lemma 5.1. Any common substring of \( T_2 \) and one of the strings in \( S_i \) will be a prefix of the label of some visited node in \( ST(S_i) \). It follows that \( t^*_i \) is the label of the node of maximal string depth visited during the traversal.

We now analyse the time and space complexity of the algorithm. Since the total length of the strings in \( S_i \) is at most \( 2n/\sqrt{\tau} \), the suffix tree can be built in \( O(n/\sqrt{\tau}) \) space and time. The traversal takes \( O(n) \) time (see Lemma 5.1). Consequently, \( t^*_i \) can be found in \( O(n/\sqrt{\tau}) \) space and \( O(n) \) time. By repeating for all \( i = 1, \ldots, \sqrt{\tau} \), we obtain the following lemma:

**Lemma 5.4** Let \( 1 \leq \tau \leq n^{2/3} \). If the length of longest common substring of \( T_1 \) and \( T_2 \) is at most \( \tau \), it can be computed in \( O(n/\sqrt{\tau}) \) space and \( O(n\sqrt{\tau}) \) time, where \( n \) is the total length of \( T_1 \) and \( T_2 \).

**Combining the Solutions.**

By combining Lemma 5.3 and Lemma 5.4, we see that the LCS can be computed in \( O(n/\sqrt{\tau}) \) space and \( O(n\sqrt{\tau} + (n/\sqrt{\tau}) \log n) \) time for \( 1 \leq \tau \leq n^{2/3} \). Substituting \( \tau = n^{2\varepsilon} \) the space bound becomes \( O(n^{1-\varepsilon}) \) and the time \( O(n^{1+\varepsilon} + n^{1-\varepsilon} \log n) \), which is \( O(n^{1+\varepsilon}) \) for \( \varepsilon > 0 \). This concludes the proof of Theorem 5.1(i).

5.4 Longest Common Substring of Multiple Strings

In this section we prove Theorem 5.1(ii). Similar to the case of two strings, the algorithm consists of two procedures that both use space \( O(n/\sqrt{\tau}) \). The first
one correctly computes the LCS if its length is at least $\tau' = \frac{1}{11} \tau \log^2 n$, while the second works if the length of the LCS is at most $\tau'$. We then combine the solutions to obtain the desired trade-off. The choice of the specific separation value $\tau'$ comes from the fact that we need $\tau' \leq n$, and since the general solution for long LCS requires a data structure with a superlinear space bound.

5.4.1 A General Solution for Long LCS

Note that we cannot use the same idea that we use in the case of two strings since the property of difference cover samples (Lemma 5.2) does not necessarily hold for $d$ positions. Instead we propose a different approach described below.

If $d > \frac{n}{\sqrt{\tau}}$, the algorithm returns an empty string and stops. This can be justified by the following simple observation.

**Lemma 5.5** If $d > \frac{n}{\sqrt{\tau}}$ then $|\text{LCS}| < \tau$.

**Proof** From $d > \frac{n}{\sqrt{\tau}}$ it follows that among any $d$ strings from $T_1, T_2, \ldots, T_m$ there is at least one string shorter than $\sqrt{\tau}$. Therefore, the length of LCS is smaller than $\sqrt{\tau} < \tau$.

This leaves us with the case where $d \leq \frac{n}{\sqrt{\tau}}$. We first construct the difference cover sample with parameter $\tau'$ for the string $T = T_1\$T_2\$\ldots\$T_m\$, where $\$, $1 \leq i \leq m$, are special characters that do not occur in $T_1, T_2, \ldots, T_m$. We also construct the arrays and the range minimum query data structures described in Chapter 5.2.2 for computing longest common prefixes between pairs of sampled suffixes or pairs of reversed blocks in constant time.

Suppose that the LCS is a prefix of $T_i[p_i..]$, for some $1 \leq i \leq m$, $1 \leq p_i \leq |T_i|$. Then to compute $|\text{LCS}|$ it is enough to find $(d - 1)$ suffixes of distinct strings from $T_1, T_2, \ldots, T_m$ such that the lcp values for them and $T_i[p_i..]$ are maximal. The length of the LCS will be equal to the minimum of the lcp values. Below we show how to compute the minimum.

Let $N_1$ stand for zero, and $N_i$, $i \geq 2$, stand for the length of $T_1\$T_2\$\ldots\$T_{i-1}\$. Consider the sampled positions $p^k_i, p^2_i, \ldots, p^z_i$ in an interval $[N_i + p_i, N_i + p_i + \tau']$ (see Figure 5.3).

From the property of the difference cover samples it follows that there is an integer $r < \tau'$ such that both $p'_i = (N_i + p_i) + r$ and $p'_j = (N_j + p_j) + r$ are in $DC_{\tau'}(T)$ — in particular, $p'_i = p^k_i$ for some $k$. Moreover, if $\text{lcp}(T_i[p_i..], T_j[p_j..]) \geq \tau'$, then the length of the longest common prefix of $\text{RB}(p^k_i)$ and $\text{RB}(p'_j)$ is at least $r = (p^k_i - N_i) - p_i$. 
We need only to consider the positions \( p_i \), \( p_i^2 \), ..., \( p_i^z \) of \( T \) in an interval \([N_i + p_i, N_i + p_i + \tau']\), and a reversed block \( RB(p_i^k) \).

Let \( \text{lcp}^k \) be the maximum length of the longest common prefix of \( T_i[p_i^k - N_i..] \) and \( T_j[p_j^k - N_j..] \), taken over all possible choices of \( p_j \), \( N_j < p_j \leq N_{j+1} \), such that \( \text{lcp}(RB(p_i^k), RB(p_j^k)) \geq ((p_i^k - N_i) - p_i) \). For each \( k \) we define a list \( L^k \) to contain values \((p_i^k - N_i) - p_i) + \text{lcp}^k - 1, j \neq i\), in decreasing order. Note that since the number of the sampled positions in \([N_i + p_i, N_i + p_i + \tau']\) is at most \( \sqrt{1.5\tau^2} + 6 \) (see Chapter 5.2.2), the number of the lists does not exceed \( \sqrt{1.5\tau^2} + 6 \) as well.

We first explain how we use the lists to obtain the answer and then how their elements are retrieved. The lists \( L^k \) are merged into a sorted list \( L \) until it contains values corresponding to suffixes of \((d - 1)\) distinct strings from \( T_1, T_2, \ldots, T_m \). The algorithm maintains a heap \( H_{\text{val}} \) on the values stored in the heads of the lists and a heap \( H_{\text{id}} \) on the distinct identifiers of strings already added to \( L \). At each step it takes the maximum value in \( H_{\text{val}} \) and moves it from its list to \( L \). Then it updates \( H_{\text{val}} \) and \( H_{\text{id}} \) and proceeds. The last value added to \( L \) will be equal to the length of the LCS.

We now explain how to retrieve values from \( L^k \). Consider a set \( S \) of \( |DC_{\tau'}(T)| \) coloured points in the plane, where a point corresponding to a position \( p \in DC_{\tau'}(T) \) will have \( x \)-coordinate equal to the rank of \( T[p..] \) in the lexicographic ordering of the sampled suffixes, \( y \)-coordinate equal to the rank of \( RB(p) \) in the lexicographic ordering of the reversed blocks, and colour equal to the number of the string \( T[p..] \) starts within.

We will show that after having retrieved the first \( \ell - 1 \) elements from \( L^k \), the next element can be retrieved using \( O(\log n) \) coloured orthogonal range reporting queries on the set \( S \). For an integer \( \ell \) and an axis-parallel rectangle \([a_1, b_1] \times [a_2, b_2]\), such a query reports \( \ell \) points of distinct colours lying in the rectangle. We need only to consider the positions \( p \) such that \( \text{lcp}(RB(p_i^k), RB(p)) \geq ((p_i^k - N_i) - p_i) \). These positions form an interval \( I^k \) of the reversed block array, \( \text{SA}_R^T \). For each \( L^k \) we maintain a rectangle \( R = [x_1, x_2] \times I^k \) such that \( x_1 \leq x \leq x_2 \), where \( x \) is the \( x \)-coordinate of the point corresponding to the position \( p_i^k \). After the first \( \ell - 1 \) elements of \( L^k \) have been retrieved, \( R \) contains points of \( (\ell - 1) \) colours besides \( i \) and \( L^k[\ell - 1] = ((p_i^k - N_i) - p_i) + \text{lcp}(x_1, x_2) - 1 \),
where \( \text{lcp}(x_1, x_2) \) is the longest common prefix of suffixes of \( T \) with ranks \( x_1 \) and \( x_2 \) (see Figure 5.4). To retrieve the next element we extend \( R \) until it contains points of \( \ell \) colours not equal to \( i \). We do this by extending either its left or right side until it includes a point of a new colour. We keep the rectangle that maximises \( \text{lcp}(x_1, x_2) \). Finding the two candidate rectangles can be done by performing two separate binary searches for the right and left sides using \( O(\log n) \) coloured orthogonal range queries. Note that in each query at most \( \ell \) points are to be reported.

The procedure described above is repeated for all \( 1 \leq i \leq m \) and \( 1 \leq p_i \leq |T_i| \). The maximum of the retrieved values will be equal to the length of the LCS. We can compute the LCS itself, too, if we remember \( i \) and \( p_i \) on which the maximum is achieved.

**Lemma 5.6** Let \( 1 \leq \tau \leq 11n/\log^2 n \), and let LCS be the longest substring that appears in at least \( d \) of the strings \( T_1, \ldots, T_m \) of total length \( n \). In case \( |\text{LCS}| \geq \frac{1}{11} \tau \log^2 n \), the LCS can be found in \( O(n/\sqrt{\tau}) \) space and \( O(nd\sqrt{\tau} \log^2 n (\log^2 n + d)) \) time.

**Proof** If \( d > n/\sqrt{\tau} \), the algorithm returns an empty string and thus is correct. Otherwise, \( \tau' = \frac{1}{11} \tau \log^2 n \leq n \), and correctness of the algorithm follows from its description. The data structures for performing constant time \( \text{lcp} \) computations require \( O(n/\sqrt{\tau}) \) space and can be built in \( O(n\sqrt{\tau} \log n) \) time.

Suppose that \( i \) and \( p_i \) are fixed. Each interval \( I^k \) can be found using \( O(\log n) \) \( \text{lcp} \) computations. To perform coloured orthogonal range
queries on the set \( S \) of size \(|DC_{\tau'}(T)| = \mathcal{O}(n/(\sqrt{\tau} \log n))\), we use the data structure [88] that can be constructed in \( \mathcal{O}(|S| \log^2 |S|) = \mathcal{O}(n \log n)/\sqrt{\tau} \) time and \( \mathcal{O}(|S| \log |S|) = \mathcal{O}(n/\sqrt{\tau}) \) space and allows to report \( \ell \) points of distinct colours in time \( \mathcal{O}(\log^2 |S| + \ell) = \mathcal{O}(\log^2 n + \ell) \). Thus retrieving \( L^k[\ell] \) takes time \( \mathcal{O}(\log(n/\log^2 n + \ell)) \). The merge stops after retrieving at most \( d \) elements from each of the \( \mathcal{O}(\sqrt{\tau}) \) lists, which will take \( \mathcal{O}(d\sqrt{\tau} \log n (\log^2 n + d)) = \mathcal{O}(d\sqrt{\tau} \log^2 n (\log^2 n + d)) \) time.

Merging the lists into \( L \) will take \( \mathcal{O}(\log \tau' + \log d) \) time per element, and thus, \( \mathcal{O}(d\sqrt{\tau'}(\log \tau' + \log d)) = \mathcal{O}(d\sqrt{\tau'} \log^{3/2} n) \) time in total, and \( \mathcal{O}(\sqrt{\tau'} + d) = \mathcal{O}(n/\sqrt{\tau}) \) space (remember that we are in the case \( d \leq n/\sqrt{\tau} \)). Therefore, computing the longest prefix of \( T_i[p_i..] \) which occurs in at least \((d - 1)\) other strings will take \( \mathcal{O}(d\sqrt{\tau} \log^2 n (\log^2 n + d)) \) time. The lemma follows. \( \blacksquare \)

### 5.4.2 A General Solution for Short LCS

We start by proving the following lemma:

**Lemma 5.7** Given input strings \( T_1, T_2, \ldots, T_m \) of total length \( n \) and a string \( S \) of length \( |S| \). The longest substring \( t \) of \( S \) that appears in at least \( d \) of the input strings can be found in \( \mathcal{O}((|S| + n) \log |t|) \) time and \( \mathcal{O}(|S|) \) space.

**Proof** We prove that there is an algorithm that takes an integer \( i \), and in \( \mathcal{O}(|S| + n) \) time and \( \mathcal{O}(|S|) \) space either finds an \( i \)-length substring of \( S \) that occurs in at least \( d \) input strings, or reports that no such substring exists. The lemma then follows, since by running the algorithm \( \mathcal{O}(\log |t|) \) times we can do an exponential search for the maximum value of \( i \).

We construct the algorithm as follows. First we build the suffix tree \( ST(S) \) for the string \( S \), together with all suffix links. For every node of the suffix tree we store a pointer to its ancestor of string depth \( i \) (all such pointers can be computed in \( \mathcal{O}(|S|) \) time by post-processing the tree). Besides, for every node \( v \in ST(S) \) of string depth \( i \) (explicit or implicit), we store a counter \( c(v) \) and an integer \( id(v) \), both initially set to zero. These nodes correspond exactly to the \( i \)-length substrings of \( S \), and we will use \( c(v) \) to count the number of distinct input strings that the label of \( v \) occurs in. To do this, we traverse \( ST(S) \) with the input strings \( T_1, T_2, \ldots, T_m \) one at a time as described in Lemma 5.1. When matching a character \( a \) of \( T_j \), we always check if a node \( v \) of string depth \( i \) above our current location has \( id(v) < j \). In that case, we increment the counter \( c(v) \) and set \( id(v) = j \) to ensure that the counter is only incremented once for \( T_j \).
To prove the correctness note that for any $i$-length substring $\ell$ of $T_j$ that also occurs in $S$ there exists a node of $ST(T)$ labelled by it, and one of the descendants of this node will be visited during the matching process of $T_j$ (see Lemma 5.1). The converse is also true, because any node $v' \in ST(T)$ visited during the traversal implies that all prefixes of the label of $v'$ occur in $T_j$.

The suffix tree for $S$ can be constructed in $O(|S|)$ time and space. The traversal with $T_j$ can be implemented to take time $O(|T_j|)$, i.e., $O(n)$ time for all the input strings. In addition to the suffix tree, at most $|S|$ constant space counters are stored. Thus the algorithm requires $O(n + |S|)$ time and $O(|S|)$ space.

We now describe the algorithm for finding the LCS when $|LCS| \leq \tau' = 1 \frac{1}{11} \tau \log^2 n$. Consider the partition of $T$ into substrings of length $\delta n/\sqrt{\tau}$ overlapping in $\tau'$ positions, where $\delta$ is a suitable constant. Assuming that $\tau \leq n^{2/3-\gamma}$ for some constant $\gamma > 0$, implies that these strings will have length at least $2\tau'$, and thus the LCS will be a substring of one of them. We examine the strings one by one and apply Lemma 5.7 to find the longest substring that occurs in at least $d$ input strings. It follows that we can check one string in $O(n/\sqrt{\tau})$ space and $O(n \log n)$ time, so by repeating for all $O(\sqrt{\tau})$ strings, we have:

**Lemma 5.8** Let $1 \leq \tau \leq n^{2/3-\gamma}$ for some constant $\gamma > 0$, and let LCS denote the longest substring that appear in at least $d$ of the strings $T_1, T_2, \ldots, T_m$ of total length $n$. If $|LCS| \leq \frac{1}{11} \tau \log^2 n$, the LCS can be found in $O(n/\sqrt{\tau})$ space and $O(\sqrt{\tau} n \log n)$ time.

**Combining the Solutions.**

Our specific choice of separation value ensures that the assumption on $\tau$ of Lemma 5.8 implies the assumption of Lemma 5.6 (because $n^{2/3-\gamma} \leq 11n/\log^2 n$ for all $n$ and $\gamma > 0$). Thus by combining the two solutions the LCS can be computed in $O(n/\sqrt{\tau})$ space and $O(d\sqrt{\tau} n \log^2 n \log^2 n + d)$ time for $1 \leq \tau \leq n^{2/3-\gamma}$, $\gamma > 0$. Substituting $\tau = n^{2\varepsilon}$, we obtain the bound stated by Theorem 5.1(ii) with the requirement that $0 \leq \varepsilon < 1/3$.

**5.5 Open Problems**

We conclude with some open problems. Is it possible to extend the trade-off range of our solutions to ideally $0 \leq \varepsilon \leq 1/2$? Can the time bound for
the general LCS problem be improved so it fully generalises the solution for two strings? The difference cover technique requires $\Omega(\sqrt{n})$ space, so the most interesting question is perhaps whether the LCS problem can be solved deterministically in $O(n^{1-\varepsilon})$ space and $O(n^{1+\varepsilon})$ time for any $0 \leq \varepsilon \leq 1$?

**Acknowledgements.** T. Starikovskaya has been partly supported by a grant 10-01-93109-CNRS-a of the Russian Foundation for Basic Research and by Dynasty foundation.
CHAPTER 6

SUBLINEAR SPACE ALGORITHMS FOR THE LONGEST COMMON SUBSTRING PROBLEM

Tomasz Kociumaka*, Tatiana Starikovskaya†, Hjalte Wedel Vildhøj‡

* Institute of Informatics, University of Warsaw
† National Research University Higher School of Economics (HSE)
‡ Technical University of Denmark, DTU Compute

Abstract

Given $m$ documents of total length $n$, we consider the problem of finding a longest string common to at least $d \geq 2$ of the documents. This problem is known as the longest common substring (LCS) problem and has a classic $O(n)$ space and $O(n)$ time solution (Weiner [FOCS’73], Hui [CPM’92]). However, the use of linear space is impractical in many applications. In this paper we show that for any trade-off parameter $1 \leq \tau \leq n$, the LCS problem can be solved in $O(\tau)$ space and $O(n^2/\tau)$ time, thus providing the first smooth deterministic time-space trade-off from constant to linear space. The result uses a new and very simple algorithm, which computes a $\tau$-additive approximation to the LCS in $O(n^2/\tau)$ time and $O(1)$ space. We also show a time-space trade-off lower bound for deterministic branching programs, which implies that any deterministic RAM algorithm solving the

---

1Supported by Polish budget funds for science in 2013-2017 as a research project under the ‘Diamond Grant’ program

Partly supported by Dynasty Foundation
LCS problem on documents from a sufficiently large alphabet in $O(\tau)$ space must use $\Omega(n\sqrt{\log(n/\tau \log n)})/\log \log(n/(\tau \log n))$ time.

6.1 Introduction

The longest common substring (LCS) problem is a fundamental and classic string problem with numerous applications. Given $m$ strings $T_1, T_2, \ldots, T_m$ (the documents) from an alphabet $\Sigma$ and a parameter $2 \leq d \leq m$, the LCS problem is to compute a longest string occurring in least $d$ of the $m$ documents. We denote such a string by $LCS$ and use $n = \sum_{i=1}^{m} |T_i|$ to refer to the total length of the documents.

The classic text-book solution to this problem is to build the (generalized) suffix tree of the documents and find the node that corresponds to $LCS$ [73, 80, 147]. While this can be achieved in linear time, it comes at the cost of using $\Omega(n)$ space$^1$ to store the suffix tree. In applications with large amounts of data or strict space constraints, this renders the classic solution impractical. A recent example of this challenge is automatic generation of signatures for identifying zero-day worms by solving the LCS problem on internet packet data [1, 105, 146]. The same challenge is faced if the length of the longest common substring is used as a measure for plagiarism detection in large document collections.

To overcome the space challenge of suffix trees, succinct and compressed data structures have been subject to extensive research [72, 131]. Nevertheless, these data structures still use $\Omega(n)$ bits of space in the worst-case, and are thus not capable of providing truly sublinear space solutions to the LCS problem.

6.1.1 Our Results

We give new sublinear space algorithms for the LCS problem. They are designed for the word-RAM model with word size $w = \Omega(\log n)$, and work for integer alphabets $\Sigma = \{1, 2, \ldots, \sigma\}$ with $\sigma = n^{O(1)}$. Throughout the paper, we regard the output to the LCS problem as a pair of integers referring to a substring in the input documents, and thus the output fits in $O(1)$ machine words.

As a stepping stone to our main result, we first show that an additive approximation of $LCS$ can be computed in constant space. We use $|LCS|$ to denote the length of the longest common substring.

$^1$Throughout the paper, we measure space as the number of words in the standard unit-cost word-RAM model with word size $w = \Theta(\log n)$ bits.
Theorem 6.1  There is an algorithm that given a parameter $\tau$, $1 \leq \tau \leq n$, runs in $O(n^2/\tau)$ time and $O(1)$ space, and outputs a string, which is common to at least $d$ documents and has length at least $|LCS| - \tau + 1$.

The solution is very simple and essentially only relies on a constant space pattern matching algorithm as a black-box. We expect that it could be of interest in applications where an approximation of LCS suffices.

For $\tau = 1$ we obtain the corollary:

Corollary 6.1  LCS can be computed in $O(1)$ space and $O(n^2)$ time.

To the best of our knowledge, this is the first constant space $O(n^2)$-time algorithm for the LCS problem. Given that it is a simple application of a constant space pattern matching algorithm, it is an interesting result on its own.

Using Theorem 6.1 we are able to establish our main result, which gives the first deterministic time-space trade-off from constant to linear space:

Theorem 6.2  There is an algorithm that given a parameter $\tau$, $1 \leq \tau \leq n$, computes LCS in $O(\tau)$ space and $O(n^2/\tau)$ time.

Previously, no deterministic trade-off was known except in the restricted setting where $n^{2/3} < \tau \leq n$. In this case two of the authors showed that the problem allows an $O((n^2/\tau)d \log^2 n(|\log^2 n + d|))$-time and $O(\tau)$-space trade-off [141]. Our new solution is also strictly better than the $O((n^2/\tau) \log n)$-time and $O(\tau)$-space randomized trade-off, which correctly outputs LCS with high probability (see [141] for a description).

Finally, we prove a time-space trade-off lower bound for the LCS problem over large-enough alphabets, which remains valid even restricted to two documents.

Theorem 6.3  Given two documents of total length $n$ from an alphabet $\Sigma$ of size at least $n^2$, any deterministic RAM algorithm which computes the longest common substring of the two documents must use $\Omega(n^{\sqrt{\log(n/(\tau \log n))}}/\log \log (n/(\tau \log n)))$ time.

We prove the bound for non-uniform deterministic branching programs, which are known to simulate deterministic RAM algorithms with constant overhead. The lower bound of Theorem 6.3 implies that the classic linear-time solution is close to asymptotically optimal in the sense that there is no hope for a linear-time and $o(n/ \log n)$-space algorithm that solves the LCS problem on polynomial-sized alphabets.
6.2 Upper Bounds

Let $T$ be a string of length $n > 0$. Throughout the paper, we use the notation $T[i..j]$, $1 \leq i \leq j \leq n$, to denote the substring of $T$ starting at position $i$ and ending at position $j$ (both inclusive). We use the shorthand $T[..i]$ and $T[i..]$ to denote $T[1..i]$ and $T[i..n]$ respectively.

A suffix tree of $T$ is a compacted trie on suffixes of $T$ appended with a unique letter (sentinel) $\$$ to guarantee one-to-one correspondence between suffixes and leaves of the tree. The suffix tree occupies linear space. Moreover, if the size of the alphabet is polynomial in the length of $T$, then the suffix tree can be constructed in linear time [54]. We refer to nodes of the suffix tree as explicit nodes, and to nodes of the underlying trie, which are not preserved in the suffix tree, as implicit nodes. Note that each substring of $T$ corresponds to a unique explicit or implicit node, the latter can be specified by the edge it belongs to and its distance to the upper endpoint of the edge.

A generalized suffix tree of strings $T_1, T_2, \ldots, T_m$ is a trie on all suffixes of these strings appended with sentinels $\$$. It occupies linear space and for polynomial-sized alphabets can also be constructed in linear time.

**Classic solution.** As a warm-up, we briefly recall how to solve the LCS problem in linear time and space. Consider the generalized suffix tree of the documents $T_1, T_2, \ldots, T_m$, where leaves corresponding to suffixes of $T_i$, $i = 1, 2, \ldots, m$, are painted with color $i$. The main observation is that $LCS$ is the label of a deepest explicit node with leaves of at least $d$ distinct colors in its subtree. Hui [80] showed that given a tree with $O(n)$ nodes where some leaves are colored, it is possible to compute the number of distinctly colored leaves below all nodes in $O(n)$ time. Consequently, we can locate the node corresponding to $LCS$ in $O(n)$ time and $O(n)$ space.

6.2.1 Approximating LCS in Constant Space

Given a pattern and a string, it is possible to find all occurrences of the pattern in the string using constant space and linear time (see [30] and references therein). We use this result in the following $O(1)$-space additive approximation algorithm.

**Lemma 6.1** There is an algorithm that given integer parameters $\ell$, $r$ satisfying $1 \leq \ell < r \leq n$, runs in $O\left(\frac{n^2}{r-\ell}\right)$ time and constant space, and returns NO if $|LCS| < \ell$, YES if $|LCS| \geq r$, and an arbitrary answer otherwise.
**Proof** Let $S = T_1S_1T_2S_2...T_mS_m$ and $\tau = r - \ell$. Consider substrings $S_k = S[k\tau + 1..k\tau + \ell]$ for $k = 0,...,\lfloor \frac{|S|}{\tau} \rfloor$. For each $S_k$ we use a constant-space pattern matching algorithm to count the number of documents $T_i$ containing an occurrence of $S_k$. We return YES if for any $S_k$ this value is at least $d$ and NO otherwise.

If $|LCS| < \ell$, then any substring of $S$ of length $\ell$ — in particular, any $S_k$ — occurs in less than $d$ documents. Consequently, in this case the algorithm will return NO. On the other hand, any substring of $S$ of length $r$ contains some $S_k$, so if $|LCS| \geq r$, then some $S_k$ occurs in at least $d$ documents, and in this case the algorithm will return YES.

To establish Theorem 6.1 we perform a ternary search using Lemma 6.1 with the modification that if the algorithm returns YES, it also outputs a string of length $\ell$ common to at least $d$ documents. We maintain an interval $R$ containing $|LCS|$; initially $R = [1,n]$. In each step we set $\ell$ and $r$ (approximately) in $1/3$ and $2/3$ of $R$, so that we can reduce $R$ by $\lfloor |R|/3 \rfloor$. We stop when $|R| \leq \tau$. The time complexity bound forms a geometric progression dominated by the last term, which is $O(n^2/\tau)$. This concludes the proof of the following result.

**Theorem 6.1.** There is an algorithm that given a parameter $\tau$, $1 \leq \tau \leq n$, runs in $O(n^2/\tau)$ time and $O(1)$ space, and outputs a string, which is common to at least $d$ documents and has length at least $|LCS| - \tau + 1$.

### 6.2.2 An $O(\tau)$-Space and $O(n^2/\tau)$-Time Solution

We now return to the main goal of this section. Using Theorem 6.1, we can assume to know $\ell$ such that $\ell \leq |LCS| < \ell + \tau$. Organization of the text below is as follows. First, we explain how to compute $LCS$ if $\ell = 1$. Then we extend our solution so that it works with larger values of $\ell$. Here we additionally assume that the alphabet size is constant and later, in Chapter 6.2.3, we remove this assumption.

**Case $\ell = 1$.**

From the documents $T_1, T_2, ..., T_m$ we compose two lists of strings. First, we consider “short” documents $T_j$ with $|T_j| < \tau$. We split them into groups of total length in $[\tau, 1 + 2\tau]$ (except for the last group, possibly). For each group we add a concatenation of the documents in this group, appended with sentinels $S_j$, to a list $L_1$. Separately, we consider “long” documents $T_j$ with $|T_j| \geq \tau$. For each of them we add to a list $L_2$ its substrings starting at positions of the form...
topics in combinatorial pattern matching

$k\tau + 1$ for integer $k$ and in total covering $T_j$. These substrings are chosen to have length $2\tau$, except for the last whose length is in $[\tau, 2\tau]$. We assume that substrings of the same document $T_j$ occur contiguously in $\mathcal{L}_2$ and append them with $\$$. The lists $\mathcal{L}_1$ and $\mathcal{L}_2$ will not be stored explicitly but will be generated on the fly while scanning the input. Note that $|\mathcal{L}_1 \cup \mathcal{L}_2| = O(n/\tau)$.

Observation 6.1 Since the length of LCS is between 1 and $\tau$, LCS is a substring of some string $S_k \in \mathcal{L}_1 \cup \mathcal{L}_2$. Moreover, it is a label of an explicit node of the suffix tree of $S_k$ or of a node where a suffix of some $S_i \in \mathcal{L}_1 \cup \mathcal{L}_2$ branches out of the suffix tree of $S_k$.

We process candidate substrings in groups of $\tau$, using the two lemmas below.

Lemma 6.2 Consider a suffix tree of $S_k$ with $\tau$ marked nodes (explicit or implicit). There is an $O(n)$-time and $O(\tau)$-space algorithm that counts the number of short documents containing an occurrence of the label of each marked node.

Proof For each marked node we maintain a counter $c(v)$ storing the number of short documents the label of $v$ occurs in. Counters are initialized with zeros. We add each string $S_i \in \mathcal{L}_1$ to the suffix tree of $S_k$ in $O(\tau)$ time. By adding a string to the suffix tree of another string, we mean constructing the generalized suffix tree of both strings and establishing pointers from explicit nodes of the generalized suffix tree to the corresponding nodes of the original suffix tree. We then paint leaves representing suffixes of $S_i$; namely, we paint a leaf with color $j$ if the corresponding suffix of $S_i$ starts within a document $T_j$ (remember that $S_i$ is a concatenation of short documents). Then the label of a marked node occurs in $T_j$ iff this node has a leaf of color $j$ in its subtree. Using Hui’s algorithm we compute the number of distinctly colored leaves in the subtree of each marked node $v$ and add this number to $c(v)$. After updating the counters, we remove colors and newly added nodes from the tree. Since all sentinels in the strings in $\mathcal{L}_1$ are distinct, the algorithm is correct. It runs in $O(|\mathcal{L}_1|\tau + \tau) = O(n)$ time.

Lemma 6.3 Consider a suffix tree of $S_k$ with $\tau$ marked nodes (explicit or implicit). There is an $O(n)$-time and $O(\tau)$-space algorithm that counts the number of long documents containing an occurrence of the label of each marked node.
Proof For each of the marked nodes we maintain a variable \( c(v) \) counting the documents where the label of \( v \) occurs. A single document might correspond to several strings \( S_i \), so we also keep an additional variable \( m(v) \), which prevents increasing \( c(v) \) several times for a single document. As in Lemma 6.2, we add each string \( S_i \in L_2 \) to the suffix tree of \( S_k \). For each marked node \( v \) whose subtree contains a suffix of \( S_i \) ending with \( $j \), we compare \( m(v) \) with \( j \). We increase \( c(v) \) only if \( m(v) \neq j \), also setting \( m(v) = j \) to prevent further increases for the same document. Since strings corresponding to the \( T_j \) occur contiguously in \( L_2 \), the algorithm is correct. Its running time is \( O(|L_2|\tau + \tau) = O(n) \).

Let \( L = L_1 \cup L_2 \). If \( LCS \) is a substring of \( S_k \in L \), we can find it as follows: we construct the suffix tree of \( S_k \), mark its explicit nodes and nodes where suffixes of \( S_i \in L \) \((i \neq k)\) branch out, and determine the deepest of them which occurs in at least \( d \) documents. Repeating for all \( S_k \in L \), this allows us to determine \( LCS \). To reduce the space usage to \( O(\tau) \), we use Lemma 6.2 and Lemma 6.3 for batches of \( O(\tau) \) marked nodes in the suffix tree of \( S_k \) at a time. Labels of all marked node are also labels of explicit nodes in the generalized suffix tree of \( T_1, \ldots, T_m \). In order to achieve good running time we will make sure that marked nodes have, over all \( S_k \in L \), distinct labels. This will imply that we use Lemma 6.2 and Lemma 6.3 only \( O(n/\tau) \) times, and hence spend \( O(n^2/\tau) \) time overall.

We consider each of the substrings \( S_k \in L \) in order. We start by constructing a suffix tree for \( S_k \). To make sure the labels of marked nodes are distinct, we shall exclude some (explicit and implicit) nodes of \( S_k \). Each node is going to be excluded together with all its ancestors or descendants, so that it is easy to test whether a particular node is excluded. (It suffices to remember the highest and the lowest non-excluded node on each edge, if any, \( O(\tau) \) nodes in total.)

First of all, we do not need to consider substrings of \( S_1, \ldots, S_{k-1} \). Therefore we add each of strings \( S_1, S_2, \ldots, S_{k-1} \) to the suffix tree (one by one) and exclude nodes common to \( S_k \) and these strings from consideration. Note that in this case a node is excluded with all its ancestors.

Then we consider all strings \( S_k, S_{k+1}, S_{k+2}, \ldots \) in turn. For each string we construct the generalized suffix tree of \( S_k \) and the current \( S_i \) and iterate over explicit nodes of the tree whose labels are substrings of \( S_k \). If a node has not been excluded, we mark it. Once we have \( \tau \) marked nodes (and if any marked nodes are left at the end), we apply Lemma 6.2 and Lemma 6.3. If the label of a marked node occurs in at least \( d \) documents, then we can exclude the marked node and all its ancestors. Otherwise, we can exclude it with all its descendants.
Recall that LCS is a label of one of the explicit inner nodes of the generalized suffix tree of $T_1, T_2, \ldots, T_m$, i.e., there are $O(n)$ possible candidates for LCS. Moreover, we are only interested in candidates of length at most $\tau$, and each such candidate corresponds to an explicit node of the generalized suffix tree of a pair of strings from $L$. The algorithm process each such candidate exactly once due to node exclusion. Thus, its running time is $O(\frac{2\tau}{\tau}n + \tau) = O(n^2/\tau)$. At any moment it uses $O(\tau)$ space.

**General case.**

If $\ell < 10\tau$ we can still use the technique above, adjusting the multiplicative constants in the complexity bounds. Thus, we can assume $\ell > 10\tau$.

Documents shorter than $\ell$ cannot contain LCS and we ignore them. For each of the remaining documents $T_j$ we add to a list $L$ its substrings starting at positions of the form $k\tau + 1$ for integer $k$ and in total covering $T_j$. The substrings are chosen to have length $\ell + 2\tau$, except for the last whose length is in the interval $[\ell, \ell + 2\tau]$. Each substring is appended with $j$, and we assume that the substrings of the same document occur contiguously.

**Observation 6.2** Since the length of LCS is between $\ell$ and $\ell + \tau$, LCS is a substring of some string $S_k \in L$. Moreover, it is the label of a node of the suffix tree of $S_k$ where a suffix of another string $S_i \in L$ branches out. (We do not need to consider explicit nodes of the suffix tree as there are no short documents.)

As before, we consider strings $S_k \in L$ in order and check all candidates which are substrings of $S_k$ but not any $S_i$ for $i < k$. However, in order to make the algorithm efficient, we replace all strings $S_i$, including $S_k$, with strings $r_k(S_i)$, each of length $O(\tau)$. To define the mapping $r_k$ we first introduce some necessary notions.

We say that $S[1..p]$ is a period of a string $S$ if $S[i] = S[i + p]$, $1 \leq i \leq |S| - p$. The length of the shortest period of $S$ is denoted as $\text{per}(S)$. We say that a string $S$ is primitive if its shortest period is not a proper divisor of $|S|$. Note that $\rho = S[1..\text{per}(S)]$ is primitive and therefore satisfies the following lemma:

**Lemma 6.4 (Primitivity Lemma [44])** Let $\rho$ be a primitive string. Then $\rho$ has exactly two occurrences in a string $\rho\rho$.

Let $Q_k = S_k[1 + 2\tau..\ell]$; note that $|Q_k| = \ell - 2\tau \geq 8\tau$. Let $\text{per}(Q_k)$ be the length of the shortest period $\rho$ of $Q_k$. If $\text{per}(Q_k) > 4\tau$, we define $Q'_k = \#$,
where \# is a special letter that does not belong to the main alphabet. Otherwise \( Q_k \) can be represented as \( \rho' \rho' \), where \( \rho' \) is a prefix of \( \rho \). We set \( Q'_k = \rho' \rho' \) for \( t' \leq t \) chosen so that \( 8 \tau \leq |Q'_k| < 12 \tau \). For any string \( S \) we define \( r_k(S) = \varepsilon \) if \( S \) does not contain \( Q_k \), and a string obtained from \( S \) by replacing the first occurrence of \( Q_k \) with \( Q'_k \) otherwise. Below we explain how to compute \( Q'_k \).

**Lemma 6.5** One can decide in linear time and constant space if \( \text{per}(Q_k) \leq 4 \tau \) and provided that this condition holds, compute \( \text{per}(Q_k) \).

**Proof** Let \( P \) be the prefix of \( Q_k \) of length \( \lfloor |Q_k|/2 \rfloor \) and \( p \) be the starting position of the second occurrence of \( P \) in \( Q_k \), if any. The position \( p \) can be found in \( O(|Q_k|) \) time by a constant-space pattern matching algorithm.

We claim that if \( \text{per}(Q_k) \leq 4 \tau \leq \lfloor |Q_k|/2 \rfloor \), then \( p = \text{per}(Q_k) + 1 \). Observe first that in this case \( P \) occurs at a position \( \text{per}(Q_k) + 1 \), and hence \( p \leq \text{per}(Q_k) + 1 \). Furthermore, \( p \) cannot be smaller than \( \text{per}(Q_k) + 1 \), because otherwise \( \rho = Q_k[1..\text{per}(Q_k)] \) would occur in \( \rho\rho = Q_k[1..2\text{per}(Q_k)] \) at the position \( p \). The shortest period \( \rho \) is primitive, so this is a contradiction with Lemma 6.4.

The algorithm compares \( p \) and \( 4 \tau + 1 \). If \( p \leq 4 \tau + 1 \), it uses letter-by-letter comparison to determine whether \( Q_k[1..p-1] \) is a period of \( Q_k \). If so, by the discussion above \( \text{per}(Q_k) = p - 1 \), and the algorithm returns it. Otherwise \( \text{per}(Q_k) > 4 \tau \). The algorithm runs in \( O(|Q_k|) \) time and uses constant space. \( \blacksquare \)

**Fact 6.1** Suppose that a string \( S \), \( |S| = |Q_k| + 4 \tau \), contains \( Q_k \) as a substring. Then

(a) replacing with \( Q'_k \) any occurrence of \( Q_k \) in \( S \) results in \( r_k(S) \),

(b) replacing with \( Q'_k \) any occurrence of \( Q'_k \) in \( r_k(S) \) results in \( S \).

**Proof** We start with (a). Let \( i \) and \( i' \) be the positions of the first and last occurrence of \( Q_k \) in \( S \). We have \( 1 \leq i \leq i' \leq |S| - |Q_k| + 1 \), so \( i' - i \leq |S| - |Q_k| \leq 4 \tau \). If \( \text{per}(Q_k) > 4 \tau \) this implies that \( i' - i = 0 \), or, in other words, that \( Q_k \) has just one occurrence in \( S \).

On the other hand, if \( \text{per}(Q_k) \leq 4 \tau \), we observe that \( i' - i \leq 4 \tau = 8 \tau - 4 \tau \leq |Q_k| - \text{per}(Q_k) \). Therefore the string \( \rho = S[i'..i' + \text{per}(Q_k) - 1] \) fits within \( Q_k = S[i..i + |Q_k| - 1] \). It is primitive and Lemma 6.4 implies that \( \rho \) occurs in \( \rho' \rho' \) only \( t \) times, so \( i' = i + j \cdot \text{per}(Q_k) \) for some integer \( j \leq t \). Therefore all occurrences of \( Q_k \) lie in the substring of \( S \) of the form \( \rho^s \rho' \) for some \( s \geq t \). Thus, replacing any of these occurrences with \( Q'_k \) leads to the same result, \( r_k(S) \).
Now, let us prove (b). Note that if we replace an occurrence of \( Q'_k \) in \( r_k(S) \) with \( Q_k \), by (a) we obtain a string \( S' \) such that \( r_k(S') = r_k(S) \). Moreover all such strings \( S' \) can be obtained by replacing some occurrence of \( Q'_k \), in particular this is true for \( S \).

If \( \text{per}(Q_k) > 4\tau \), since \( \# \) does not belong to the main alphabet, \( Q'_k \) has exactly one occurrence in \( r_k(S) \) and the statement holds trivially. For the other case we proceed as in the proof of (a) showing that all occurrences of \( Q'_k \) are in fact substrings of a longer substring of \( S \) of the form \( \rho \rho' \) for some \( s' \geq t' \).

**Lemma 6.6** Consider strings \( P \) and \( S \), such that \( |S| \leq |Q_k| + 4\tau \) and \( P \) contains \( Q_k \) as a substring. Then \( P \) occurs in \( S \) at position \( p \) if and only if \( r_k(P) \) occurs in \( r_k(S) \) at position \( p \).

**Proof** First, assume that \( P \) occurs in \( S \) at a position \( p \). This induces an occurrence of \( Q_k \) in \( S \) within the occurrence of \( P \), and replacing this occurrence of \( Q_k \) with \( Q'_k \) gives \( r_k(S) \) by Fact 6.1(a). This replacement also turns the occurrence of \( P \) at the position \( p \) into an occurrence of \( r_k(P) \).

Now, assume \( r_k(P) \) occurs in \( r_k(S) \) at the position \( p \). Since \( r_k(P) \neq \varepsilon \), this means that \( r_k(S) \neq \varepsilon \) and that \( Q'_k \) occurs in \( r_k(S) \) (within the occurrence of \( r_k(P) \)). By Fact 6.1(b) replacing this occurrence of \( Q'_k \) with \( Q_k \) turns \( r_k(S) \) into \( S \) and the occurrence of \( r_k(P) \) at the position \( p \) into an occurrence of \( P \).

Observe that applied for \( S = S_k \), Lemma 6.6 implies that \( r_k \) gives a bijection between substrings of \( S_k \) of length \( \geq \ell = |Q_k| + 2\tau \) and substrings of \( r_k(S_k) \) of length \( \geq |Q'_k| + 2\tau \). Moreover, it shows that any substring of \( S_k \) of length \( \geq \ell \) occurs in \( S \) iff the corresponding substring of \( r_k(S_k) \) occurs in \( r_k(S) \).

This lets us apply the technique described in the previous section to find \( \text{LCS} \) provided that it occurs in \( S_k \) but not \( S_i \) with \( i < k \). Strings \( r_k(S_i) \) are computed in parallel with \( \rho \rho' \) a constant-space pattern matching algorithm for a pattern \( Q_k \) in the documents of length \( \ell \) or more, which takes \( \mathcal{O}(n) \) time in total. The list \( \mathcal{L} \) is composed \( r_k(S_i) \) obtained from long documents, and we use Lemma 6.3 to compute the number of documents each candidates occurs in.

Compared to the arguments of the previous section, we additionally exclude nodes of depth less than \( |Q'_k| + 2\tau \) to make sure that each marked node is indeed \( r_k(P) \) for some substring \( P \) of \( S_k \) of length at least \( \ell = |Q_k| + 2\tau \). This lets us use the amortization by the number of explicit nodes in the generalized suffix tree of \( T_1, \ldots, T_m \). More precisely, if a node with label \( r_k(P) \) is marked, we charge \( P \), which is guaranteed to be explicit in the generalized suffix tree. This implies \( \mathcal{O}(n^2/\tau) \)-time and \( \mathcal{O}(\tau) \)-space bounds.
6.2.3 Large alphabets

In this section we describe how to adapt our solution so that it works for alphabets of size $n^{O(1)}$. Note that we have used the constant-alphabet assumption only to make sure that suffix trees can be efficiently constructed. If the alphabet is not constant, a suffix tree of a string can be constructed in linear time plus the time of sorting its letters [54]. If $\tau > \sqrt{n}$, the size of the alphabet is $n^{O(1)} = \tau^{O(1)}$ and hence any suffix tree used by the algorithm can be constructed in $O(\tau)$ time.

Suppose now that $\tau \leq \sqrt{n}$ and $\ell = 1$. Our algorithm uses suffix trees in a specific pattern: in a single phase it builds the suffix tree of $S_k$ and then constructs the generalized suffix tree of $S_k$ and $S_i$ for each $i$. Note that the algorithm only needs information about the nodes of the suffix tree of $S_k$, the nodes where suffixes of $S_i \in \mathcal{L}$ branch out, and leaves of the generalized suffix tree. None of these changes if we replace each letter of $\Sigma$ occurring in $S_i$, but not in $S_k$, with a special letter which does not belong to $\Sigma$.

Thus our approach is as follows: first we build a deterministic dictionary, mapping letters of $S_k$ to integers of magnitude $O(|S_k|) = O(\tau)$ and any other letter of the main alphabet to the special letter. The dictionary can be constructed in $O(\tau \log^2 \log \tau)$ time [77, 137]. Then instead of building the generalized suffix tree of $S_k$ and $S_i$ we build it for the corresponding strings with letters mapped using the dictionary. In general, when $\ell$ is large, we apply the same idea with $r_k(S_k)$ and $r_k(S_i)$ instead of $S_k$ and $S_i$ respectively.

In total, the running time is $O(n^2/\tau + n \log^2 \log \tau)$. For $\tau \leq \sqrt{n}$ the first term dominates the other, i.e. we obtain an $O(n^2/\tau)$-time solution.

**Theorem 6.2.** There is an algorithm that given a parameter $\tau$, $1 \leq \tau \leq n$, computes LCS in $O(n^2/\tau)$ time using $O(\tau)$ space.

6.3 A Time-Space Trade-Off Lower Bound

Given $n$ elements over a domain $D$, the element distinctness problem is to decide whether all $n$ elements are distinct. Beame et al. [19] showed that if $|D| \geq n^2$, then any RAM algorithm solving the element distinctness problem in $\tau$ space, must use at least $\Omega(n \sqrt{\log(n/(\tau \log n))/\log \log(n/(\tau \log n))})$ time.\footnote{Note that in [19, 28] the space consumption is measured in bits. The version of RAM used there is unit-cost with respect to time and log-cost with respect to space.}

The element distinctness (ED) problem can be seen as a special case of the LCS problem where we have $m = n$ documents of length 1 and want to find
the longest string common to at least \( d = 2 \) documents. Thus, the lower bound for ED also holds for this rather artificial case of the LCS problem. Below we show that the same bound holds with just \( m = 2 \) documents. The main idea is to show an analogous bound for a two-dimensional variant of the element distinctness problem, which we call the element bidistinctness problem. The LCS problem on two documents naturally captures this problem. The steps are similar to those for the ED lower bound by Beame et al. [19], but the details differ. We start by introducing the necessary definitions of branching programs and embedded rectangles. We refer to [19] for a thorough overview of this proof technique.

**Branching Programs.** A \( n \)-variate branching program \( \mathcal{P} \) over domain \( D \) is an acyclic directed graph with the following properties: (1) there is a unique source node denoted \( s \), (2) there are two sink nodes, one labelled by 0 and one labelled by 1, (3) each nonsink node \( v \) is assigned an index \( i(v) \in [1, n] \) of a variable, and (4) there are exactly \( |D| \) arcs out of each nonsink node, labelled by distinct elements of \( D \). A branching program is executed on an input \( x \in D^n \) by starting at \( s \), reading the variable \( x_{i(s)} \) and following the unique arc labelled by \( x_{i(s)} \). This process is continued until a sink is reached and the output of the computation is the label of the sink. For a branching program \( \mathcal{P} \), we define its size as the number of nodes, and its length as the length of the longest path from \( s \) to a sink node.

**Lemma 6.7 (see page 2 of [28])** If \( f : D^n \rightarrow \{0, 1\} \) has a word-RAM algorithm with running time \( T(n) \) using \( S(n) \) \( w \)-bit words, then there exists an \( n \)-variate branching program \( \mathcal{P} \) over \( D \) computing \( f \), of length \( O(T(n)) \) and size \( 2^{O(wS(n)+\log n)} \).

**Embedded Rectangles.** If \( A \subseteq [1, n] \), a point \( \tau \in D^A \) (i.e. a function \( \tau : A \rightarrow D \)) is called a partial input on \( A \). If \( \tau_1, \tau_2 \) are partial inputs on \( A_1, A_2 \subseteq [1, n] \), \( A_1 \cap A_2 = \emptyset \), then \( \tau_1 \tau_2 \) is the partial input on \( A_1 \cup A_2 \) agreeing with \( \tau_1 \) on \( A_1 \) and with \( \tau_2 \) on \( A_2 \). For sets \( B \subseteq D^{[1,n]} \) and \( A \subseteq [1, n] \) we define \( B_A \), the projection of \( B \) onto \( A \), as the set of all partial inputs on \( A \) which agree with some input in \( B \). An embedded rectangle \( R \) is a triple \((B, A_1, A_2)\), where \( A_1 \) and \( A_2 \) are disjoint subsets of \([1, n]\), and \( B \subseteq D^{[1,n]} \) satisfies: (i) \( B_{[1,n]\setminus A_1 \cup A_2} \) consists of a single partial input \( \sigma \), (ii) if \( \tau_1 \in B_{A_1} \), and \( \tau_2 \in B_{A_2} \), then \( \tau_1 \tau_2 \sigma \in B \). For an embedded rectangle \( R = (B, A_1, A_2) \), and \( j \in \{1,2\} \) we define:

\[
\begin{align*}
m_j(R) &= |A_j| \\
\alpha_j(R) &= |B_{A_j}|/|D|^{|A_j|}
\end{align*}
\]

\[
\begin{align*}
m(R) &= \min(m_1(R), m_2(R)) \\
\alpha(R) &= \min(\alpha_1(R), \alpha_2(R))
\end{align*}
\]
Given a small branching program $P$ it can be shown that $P^{-1}(1)$, the set of all YES-inputs, contains a relatively large embedded rectangle. Namely,

**Lemma 6.8 (Corollary 5.4 (i) [19])** Let $k \geq 8$ be an integer, $q \leq 2^{-40}k^{-8}$, $n \geq r \geq q^{-5k^2}$. Let $P$ be a $n$-variate branching program over domain $D$ of length at most $(k - 2)n$ and size $2^S$. Then there is an embedded rectangle $R$ contained in $P^{-1}(1)$ satisfying $m(R) = m_1(R) = m_2(R) \geq q^{2k^2}n/2$ and $\alpha(R) \geq 2^{-q^{1/2}m(R)} - \sum_{\tau \in P^{-1}(1)} |D^n|$. 

**Element Bidistinctness.** We say that two elements $x = (x_1, x_2)$ and $y = (y_1, y_2)$ of the Cartesian product $D \times D$ are bidistinct if both $x_1 \neq y_2$ and $x_2 \neq y_1$. The element bidistinctness function $EB : (D \times D)^n \rightarrow \{0, 1\}$ is defined to be 1 iff for every pair of indices $1 \leq i, j \leq n$ the $i$-th and $j$-th pair are bidistinct. Note that computing $EB$ for $(s_1, t_1), \ldots, (s_n, t_n)$ is equivalent to deciding if $LCS(s_1 \ldots s_n, t_1 \ldots t_n) \geq 1$. Thus the problem of computing the longest common substring of two strings over $\Sigma = D$ is at least as hard as the EB problem. Below we show a time-space trade-off lower bound for element bidistinctness.

**Lemma 6.9** If $|D| \geq 2n^2$, at least a fraction $1/e$ of inputs belong to $EB^{-1}(1)$.

**Proof** The size of $EB^{-1}(1)$ is at least $(|D| - 1)^2 \cdot (|D| - 2)^2 \cdot \ldots \cdot (|D| - n)^2$. Hence, $|EB^{-1}(1)| = |D|^{2n} \prod_{i=1}^{n} (1 - \frac{i}{|D|^2})^2 \geq |D|^{2n} (1 - \frac{1}{2n})^{2n} \geq |D|^{2n}/e$. 

**Lemma 6.10** For any embedded rectangle $R = (B, A_1, A_2) \subseteq EB^{-1}(1)$ we have $\alpha(R) \leq 2^{-2m(R)}$.

**Proof** Let $S_j$ be the subset of $D \times D$ that appear on indices in $A_j$, i.e., $S_j = \bigcup_{\tau \in B_{A_j}} \{\tau(i) : i \in A_j\}$, $j = 1, 2$. Clearly, all elements in $S_1$ must be bidistinct from all elements in $S_2$. If this was not the case $B$ would contain a vector with two non-bidistinct elements of $D \times D$. We will prove that $\min(|S_1|, |S_2|) \leq |D|^2/4$. Let us first argue that this implies the lemma. For $j = 1$ or $j = 2$, we get that $|B_{A_j}| \leq (|D|^2/4)^{|A_j|}$, and thus $\alpha_j(R) \leq (|D|^2/4)^{|A_j|}/(|D|^2)^{|A_j|} = 4^{-|A_j|} \leq 4^{-m(R)} = 2^{-2m(R)}$.

It remains to prove that $\min(|S_1|, |S_2|) \leq |D|^2/4$. For $j \in \{1, 2\}$ let $X_j$ and $Y_j$ denote the set of first and second coordinates that appear in $S_j$. Note that by bidistinctness $X_1 \cap Y_2 = X_2 \cap Y_1 = \emptyset$. Moreover $|S_j| \leq |X_j||Y_j|$ and therefore $\sqrt{|S_j|} \leq \sqrt{|X_j||Y_j|} \leq \frac{1}{2}(|X_j| + |Y_j|)$. Consequently $2(\sqrt{|S_1|} + \sqrt{|S_2|}) \leq |D|^2/4$. Therefore $\sqrt{|S_1|} + \sqrt{|S_2|} \leq |D|^2/8$. Since $|D| \geq 2n^2$, we have $\min(|S_1|, |S_2|) \leq |D|^2/4$.
\( \sqrt{|S_2|} \leq |X_1| + |Y_1| + |X_2| + |Y_2| = (|X_1| + |Y_1|) + (|Y_1| + |X_2|) \leq 2|D| \) and thus \( \min(\sqrt{|S_1|}, \sqrt{|S_2|}) \leq |D|/2 \), i.e., \( \min(|S_1|, |S_2|) \leq |D|^2/4 \) as claimed. \( \blacksquare \)

**Theorem 6.4** Any \( n \)-variate branching program \( P \) of length \( T \) and size \( 2^S \) over domain \( D \), \( |D| \geq 2n^2 \), which computes the element bidistinctness function \( EB \), requires \( T = \Omega(n\sqrt{\log(n/S)} / \log \log(n/S)) \) time.

**Proof** The proof repeats the proof of Theorem 6.13 [19]. We restore the details omitted in [19] for the sake of completeness. Suppose that the length of \( P \) is \( T = (k-2)n/2 \) and size \( 2^S \). Apply Lemma 6.8 with \( q = 2^{-40}k^{-8} \) and \( r = \left\lceil q^{-5k^2} \right\rceil \). We then obtain an embedded rectangle \( R \in EB^{-1}(1) \) such that \( m(R) \geq q^{2k^2}n/4 \) and \( \alpha(R) \geq 2^{-q^{1/2}m(R) - Sr/e} = 2^{-q^{1/2}m(R) - Sr - \log e} \). From Lemma 6.10 we have \( 2^{-2m(R)} \geq 2^{-q^{1/2}m(R) - Sr - \log e} \) and thus \( Sr \geq m(R)(2 - q^{1/2}) - \log e \geq m(R)/2 \). Consequently, \( S \geq q^{2k^2}n/(8r) \). Remember that \( q = 2^{-40}k^{-8} \) and \( r = \left\lceil q^{-5k^2} \right\rceil \), which means that \( P \) requires at least \( k^{-ck^2}n \) space for some constant \( c > 0 \). That is, \( k^{-ck^2} \geq n/S \), which implies \( k = \Omega(\sqrt{\log(n/S)} / \log \log(n/S)) \). Substituting \( k = 2T/n + 2 \), we obtain the claimed bound. \( \blacksquare \)

**Corollary 6.2** Any deterministic RAM algorithm that solves the element bidistinctness (EB) problem on inputs in \((D \times D)^n\), \( |D| \geq 2n^2 \), using \( \tau \leq \frac{n}{\log n} \) space, must use at least \( \Omega(n\sqrt{\log(n/(\tau \log n))} / \log \log(n/(\tau \log n))) \) time.

**Corollary 6.3** (Theorem 6.3) Given two documents of total length \( n \) from an alphabet \( \Sigma \) of size at least \( n^2 \), any deterministic RAM algorithm, which uses \( \tau \leq \frac{n}{\log n} \) space to compute the longest common substring of both documents, must use time \( \Omega(n\sqrt{\log(n/(\tau \log n))} / \log \log(n/(\tau \log n))) \).

### 6.4 Conclusions

The main problem left open by our work is to settle the optimal time-space product for the LCS problem. While it is tempting to guess that the answer lies in the vicinity of \( \Theta(n^2) \), it seems really difficult to substantially improve our lower bound. Strong time-space product lower bounds have so far only been established in weaker models (e.g., the comparison model) or for multi-output problems (e.g., sorting an array, outputting its distinct elements and various pattern matching problems). Proving an \( \Omega(n^2) \) time-space product lower bound
in the RAM model for any problem where the output fits in a constant number of words (e.g., the LCS problem) is a major open problem.
# A Suffix Tree Or Not A Suffix Tree?

**Tatiana Starikovskaya**

**Hjalte Wedel Vildhøj**

* National Research University, Higher School of Economics (HSE)
† Technical University of Denmark, DTU Compute

## Abstract

In this paper we study the structure of suffix trees. Given an unlabeled tree $\tau$ on $n$ nodes and suffix links of its internal nodes, we ask the question “Is $\tau$ a suffix tree?”, i.e., is there a string $S$ whose suffix tree has the same topological structure as $\tau$? We place no restrictions on $S$, in particular we do not require that $S$ ends with a unique symbol. This corresponds to considering the more general definition of implicit or extended suffix trees. Such general suffix trees have many applications and are for example needed to allow efficient updates when suffix trees are built online. We prove that $\tau$ is a suffix tree if and only if it is realized by a string $S$ of length $n - 1$, and we give a linear-time algorithm for inferring $S$ when the first letter on each edge is known. This generalizes the work of I et al. [Discrete Appl. Math. 163, 2014].

## 7.1 Introduction

The suffix tree was introduced by Peter Weiner in 1973 [147] and remains one of the most popular and widely used text indexing data structures (see [14] and references therein). In static applications it is commonly assumed that
suffix trees are built only for strings with a unique end symbol (often denoted $\$), thus ensuring the useful one-to-one correspondence between leaves and suffixes. In this paper we view such suffix trees as a special case and refer to them as $-suffix trees. Our focus is on suffix trees of arbitrary strings, which we simply call suffix trees to emphasize that they are more general than $-suffix trees\textsuperscript{1}. Contrary to $-suffix trees, the suffixes in a suffix tree can end in internal non-branching locations of the tree, called implicit suffix nodes.

Suffix trees for arbitrary strings are not only a nice generalization, but are required in many applications. For example in online algorithms that construct the suffix tree of a left-to-right streaming text (e.g., Ukkonen’s algorithm [144]), it is necessary to maintain the implicit suffix nodes to allow efficient updates. Despite their essential role, the structure of suffix trees is still not well understood. For instance, it was only recently proved that each internal edge in a suffix tree can contain at most one implicit suffix node [32].

In this paper we prove some new properties of suffix trees and show how to decide whether suffix trees can have a particular structure. Structural properties of suffix trees are not only of theoretical interest, but are essential for analyzing the complexity and correctness of algorithms using suffix trees.

Given an unlabeled ordered rooted tree $\tau$ and suffix links of its internal nodes, the suffix tree decision problem is to decide if there exists a string $S$ such that the suffix tree of $S$ is isomorphic to $\tau$. If such a string exists, we say that $\tau$ is a suffix tree and that $S$ realizes $\tau$. If $\tau$ can be realized by a string $S$ having a unique end symbol $\$$, we additionally say that $\tau$ is a $-suffix tree. See Figure 7.1 for examples of a $-suffix tree, a suffix tree, and a tree which is not a suffix tree. In all figures in this paper leaves are black and internal nodes are white.

I et al. [84] recently considered the suffix tree decision problem and showed how to decide if $\tau$ is a $-suffix tree in $O(n)$ time, assuming that the first letter on each edge of $\tau$ is also known. Concurrently with our work, another approach was developed in [35]. There the authors show how to decide if $\tau$ is a $-suffix tree without knowing the first letter on each edge, but also introduce the assumption that $\tau$ is an unordered tree.

Deciding if $\tau$ is a suffix tree is much more involved than deciding if it is a $-suffix tree, mainly because we can no longer infer the length of a string that realizes $\tau$ from the number of leaves. Without an upper bound on the length of such a string, it is not even clear how to solve the problem by an exhaustive search. In this paper, we give such an upper bound, show that it is tight, and

\textsuperscript{1} In the literature the standard terminology is suffix trees for $-suffix trees and extended/implicit suffix trees [31,73] for suffix trees of strings not ending with $\$.
Figure 7.1: Three potential suffix trees. (a) is a $-$suffix tree, e.g. for \texttt{ababa$}$. (b) is not a $-$suffix tree, but it is a suffix tree, e.g. for \texttt{abaabab}. (c) is not a suffix tree.

give a linear time algorithm for deciding whether $\tau$ is a suffix tree when the first letter on each edge is known.

7.1.1 Our Results

In Chapter 7.2, we start by settling the question of the sufficient length of a string that realizes $\tau$.

**Theorem 7.1** An unlabeled tree $\tau$ on $n$ nodes is a suffix tree if and only if it is realized by a string of length $n - 1$.

As far as we are aware, there were no previous upper bounds on the length of a shortest string realizing $\tau$. The bound implies an exhaustive search algorithm for solving the suffix tree decision problem, even when the suffix links are not provided. In terms of $n$, this upper bound is tight, since e.g. stars on $n$ nodes are realized only by strings of length at least $n - 1$.

The main part of the paper is devoted to the suffix tree decision problem. We generalize the work of I et al. [84] and show in Chapter 7.4 how to decide if $\tau$ is a suffix tree.

**Theorem 7.2** Let $\tau$ be a tree with $n$ nodes, annotated with suffix links of internal nodes and the first letter on each edge. There is an $O(n)$ time algorithm for deciding if $\tau$ is a suffix tree.
In case $\tau$ is a suffix tree, the algorithm also outputs a string $S$ that realizes $\tau$. To obtain the result, we show several new properties of suffix trees, which may be of independent interest.

### 7.1.2 Related Work

The problem of revealing structural properties and exploiting them to recover a string realizing a data structure has received a lot of attention in the literature. Besides $\$-$suffix trees, the problem has been considered for border arrays [50, 116], parameterized border arrays [81–83], suffix arrays [16, 52, 106], KMP failure tables [51, 68], prefix tables [38], cover arrays [45], directed acyclic word graphs [16], and directed acyclic subsequence graphs [16].

### 7.2 Suffix Trees

In this section we prove Theorem 7.1 and some new properties of suffix trees, which we will need to prove Theorem 7.2. We start by briefly recapitulating the most important definitions.

The suffix tree of a string $S$ is a compacted trie on suffixes of $S$ [73]. Branching nodes and leaves of the tree are called explicit nodes, and positions on edges are called implicit nodes. The label of a node $v$ is the string on the path from the root to $v$, and the length of this label is called the string depth of $v$. The suffix link of an internal explicit node $v$ labeled by $a_1a_2\ldots a_m$ is a pointer to the node $u$ labeled by $a_2a_3\ldots a_m$. We use the notation $v \rightarrow u$ and extend the definition of suffix links to leaves and implicit nodes as well. We will refer to nodes that are labeled by suffixes of $S$ as suffix nodes. All leaves of the suffix tree are suffix nodes, and unless $S$ ends with a unique symbol $\$$, some implicit nodes and internal explicit nodes can be suffix nodes as well. Suffix links for suffix nodes form a path starting at the leaf labeled by $S$ and ending at the root. Following [32], we call this path the suffix chain.

**Lemma 7.1** ([32]) The suffix chain of the suffix tree can be partitioned into the following consecutive segments: (1) Leaves; (2) Implicit suffix nodes on leaf edges; (3) Implicit suffix nodes on internal edges; and (4) Suffix nodes that coincide with internal explicit nodes. (See Figure 7.2(a).)

The string $S$ is fully specified by the order in which the suffix chain visits the subtrees hanging off the root. More precisely,
A SUFFIX TREE OR NOT A SUFFIX TREE?

Figure 7.2: (a) The suffix tree $\tau$ of a string $S = abaababaababaa$ with suffix nodes and the suffix chain. (b) The suffix tree of a prefix $S' = abaab$ of $S$. Suffix links of internal nodes are not shown, but they are the same in both trees.

**Observation 7.1** If $y_0 \rightarrow y_1 \rightarrow \ldots \rightarrow y_l = \text{root}$ is the suffix chain in the suffix tree of a string $S$, then $|S| = l$ and $S[i] = a_i$, where $a_i$ is the first letter on the edge going from the root to the subtree containing $y_{i-1}$, $i = 1, \ldots, l$.

We define the parent $\text{par}(x)$ of a node $x$ to be the deepest explicit node on the path from the root to $x$ (excluding $x$). The distance between a node and one of its ancestors is defined to be the difference between the string depths of these nodes.

**Lemma 7.2** If $x_1 \rightarrow x_2$ is a suffix link, then the distance from $x_1$ to $\text{par}(x_1)$ cannot be less than the distance from $x_2$ to $\text{par}(x_2)$.

**Proof** If $d$ is the distance between $x_1$ and $\text{par}(x_1)$, then the suffix link of $\text{par}(x_1)$ points to an explicit ancestor $d$ characters above $x_2$.

**Lemma 7.3** Let $x$ be an implicit suffix node. The distance between $x$ and $\text{par}(x)$ is not bigger than the length of any leaf edge.

**Proof** It follows from Lemma 7.2 that as the suffix chain $y_0 \rightarrow y_1 \rightarrow \ldots \rightarrow y_l = \text{root}$ is traversed, the distance from each node to its parent is non-increasing. Since the leaves are visited first, the distance between any implicit suffix node and its parent cannot exceed the length of a leaf edge.
Lemma 7.4 If $\tau$ is a suffix tree, then it can be realized by some string such that

1. The minimal length of a leaf edge of $\tau$ will be equal to one;
2. Any edge of $\tau$ will contain at most one implicit suffix node at the distance one from its upper end.

**Proof** Let $S$ be a string realizing $\tau$, and $m$ be the minimal length of a leaf edge of $\tau$. Consider a prefix $S'$ of $S$ obtained by deleting its last $(m-1)$ letters. Its suffix tree is exactly $\tau$ trimmed at height $m-1$. (See Figure 7.2(b).) The minimal length of a leaf edge of this tree is one. Applying Lemma 7.3, we obtain that the distance between any implicit suffix node $x$ of this tree and $\text{par}(x)$ is one, and, consequently, any edge contains at most one implicit suffix node.

Lemma 7.5 If $\tau$ is realized by a string of length $l$, then it is also realized by strings of length $l+1, l+2, l+3$, and so on.

**Proof** Let $y_0 \rightarrow y_1 \rightarrow \ldots \rightarrow y_l = \text{root}$ be the suffix chain for a string $S$ that realizes $\tau$. Moreover let $\text{letters}(y_i)$ be the set of first letters immediately below node $y_i$. Then $\text{letters}(y_{i-1}) \subseteq \text{letters}(y_i), i = 1, \ldots, l$. Let $y_j$ be the first non-leaf node in the suffix chain (possibly the root). It follows that $Sa$ also realizes $\tau$, where $a$ is any letter in $\text{letters}(y_j)$.

We now prove Theorem 7.1 by showing that if $\tau$ is a suffix tree then a string of length $n-1$ realizes it. By Lemma 7.4, $\tau$ can be realized by a string $S'$ so that the minimal length of a leaf edge is 1. Consider the last leaf $\ell$ visited by the suffix chain in the suffix tree of $S'$. By the property of $S'$ the length of the edge $(\text{par}(\ell) \to \ell)$ is 1. Remember that a suffix link of an internal node always points to an internal node and that suffix links cannot form cycles. Moreover, upon transition by a suffix link the string depth decreases exactly by one. Hence if $\tau$ has $I$ internal nodes then the string depth of the parent of $\ell$ is at most $I - 1$ and the string depth of $\ell$ is at most $I$. Consequently, if $L$ is the number of leaves in $\tau$, the length of the suffix chain and thus the length of $S'$ is at most $L + I - 1 = n - 1$, so by Lemma 7.5 there is a string of this length that realizes $\tau$. 


The Suffix Tour Graph

In their work [84] I et al. introduced a notion of suffix tour graphs. They showed that suffix tour graphs of $\$-suffix trees must have a nice structure which ties together the suffix links of the internal explicit nodes, the first letters on edges, and the order of leaves of $\tau$ — i.e., which leaf corresponds to the longest suffix, which leaf corresponds to the second longest suffix, and so on. Knowing this order and the first letters on edges outgoing from the root, it is easy to infer a string realizing $\tau$. We study the structure of suffix tour graphs of suffix trees.

Let us first formalize the input to the problem. Consider a tree $\tau = (V, E)$ annotated with a set of suffix links $\sigma : V \rightarrow V$ between internal explicit nodes, and the first letter on each edge, given by a labelling function $\lambda : E \rightarrow \Sigma$ for some alphabet $\Sigma$. For ease of description, we will always augment $\tau$ with an auxiliary node $\bot$, the parent of the root. We add the suffix link $(\text{root} \rightarrow \bot)$ to $\sigma$ and label the edge $(\bot \rightarrow \text{root})$ with a symbol ”?” , which matches any letter of the alphabet.

To construct the suffix tour graph of $\tau$, we first compute values $\ell(x)$ and $d(x)$ for every explicit node $x$ in $\tau$. The value $\ell(x)$ is equal to the number of leaves $y$ where $\text{par}(y) \rightarrow \text{par}(x)$ is a suffix link in $\sigma$, and $\lambda(\text{par}(y) \rightarrow y) = \lambda(\text{par}(x) \rightarrow x)$. See Figure 7.3(a) for an example. Let $L_x$ and $V_x$ be the sets of leaves and nodes, respectively, of the subtree of $\tau$ rooted at a node $x$. Note that $L_x$ is a subset of $V_x$. We define $d(x) = |L_x| - \sum_{y \in V_x} \ell(y)$. See Figure 7.3(b) for an example.

**Definition 7.1** The suffix tour graph of a tree $\tau = (V, E)$ is a directed graph $G = (V, E_G)$, where $E_G = \{(y \rightarrow x)^k \mid (y \rightarrow x) \in E, k = d(x)\} \cup \{(y \rightarrow x) \mid y$ is a leaf contributing to $\ell(x)\}$. Here $(y \rightarrow x)^k$ means the edge $y \rightarrow x$ with multiplicity $k$. If $k = d(x) < 0$, we define $(y \rightarrow x)^k$ to be $(x \rightarrow y)^{|k|}$.
Figure 7.3: (a) An example of a node \( x \) with \( \ell(x) = 1 \). The leaf \( y \) contributes to \( \ell(x) \) since \( \lambda(\text{par}(y) \to y) = \lambda(\text{par}(x) \to x) = a \). (b) An input consisting of a tree, suffix links and the first letter on each edge. The tree has been extended with the node \( \perp \), and each node is assigned values \((\ell(x), d(x))\). For the node \( x \), \( \ell(x) = 1 \), \(|L_x| = 2 \), and hence \( d(x) = 2 - (1 + 0 + 0) = 1 \).

\( y \) in Figure 7.3(a) would point to the subtree rooted in \( x \). The value \( \ell(x) \) is simply the number of leaves that points to \( x \). It can happen that the outgoing edge of \( y \) points to another leaf, in which case we then know the successor suffix of \( y \) with certainty. The remaining edges in the suffix tour graph are introduced to make the graph Eulerian. The subtree rooted in a node \( x \) will have \(|L_x|\) outgoing pointers, and \( \sum_{y \in V_x} \ell(y) \) incoming pointers, and hence we create \( d(x) = |L_x| - \sum_{y \in V_x} \ell(y) \) edges from \( \text{par}(x) \) to \( x \). The main idea, which we will elaborate on in the next section, is that if the graph is Eulerian (and connected), we can reconstruct the suffix chain on the leaves by finding an Eulerian cycle through the leaves of the suffix tour graph. See Figure 7.4 for an example of the suffix tour graph.

Lemma 7.6 ([84]) The suffix tour graph \( G \) of a suffix tree \( \tau \) is an Eulerian graph (possibly disconnected).

Proof I et al. [84] only proved the lemma for \( $ \)-suffix trees, but the proof holds for suffix trees as well. We give the proof here for completeness and because I et al. use different notation. To prove the lemma it suffices to show that for every node the number of incoming edges equals the number of outgoing edges.
Consider an internal node $x$ of $\tau$. It has $\sum_{z \in \text{children}(x)} d(z)$ outgoing edges and $\ell(x) + d(x)$ incoming edges. But, $\ell(x) + d(x)$ equals

$$|L_x| - \sum_{y \in V \setminus \{x\}} \ell(y) = \sum_{z \in \text{children}(x)} (|L_z| - \sum_{y \in V_z} \ell(y)) = \sum_{z \in \text{children}(x)} d(z)$$

Now consider a leaf $y$ of $\tau$. The outdegree of $y$ is one, and the indegree is equal to $\ell(x) + d(x) = \ell(x) + 1 - \ell(x) = 1$.

7.3.1 Suffix tour graph of a $\$-$suffix tree

The following proposition follows from the definition of a $\$-$suffix tree.

**Proposition 7.1 ([84])** If $\tau$ is a $\$-$suffix tree with a set of suffix links $\sigma$ and first letters on edges defined by a labelling function $\lambda$, then

1. For every internal explicit node $x$ in $\tau$ there exists a unique path $x = x_0 \rightarrow x_1 \rightarrow \ldots \rightarrow x_k = \text{root}$ such that $x_i \rightarrow x_{i+1}$ belongs to $\sigma$ for all $i$;

2. If $y$ is the end of the suffix link for $\text{par}(x)$, there is a child $z$ of $y$ such that $\lambda(\text{par}(x) \rightarrow x) = \lambda(y \rightarrow z)$, and the end of the suffix link for $x$ belongs to the subtree of $\tau$ rooted at $y$;
(3) For any node \( x \in V \) the value \( d(x) \geq 0 \).

If all tree conditions hold, it can be shown that

**Lemma 7.7 ([84])** The tree \( \tau \) is a \$-suffix tree iff its suffix tour graph \( G \) contains a cycle \( C \) which goes through the root and all leaves of \( \tau \). Moreover, a string realizing \( \tau \) can be inferred from \( C \) in linear time.

In more detail, the authors proved that the order of leaves in the cycle \( C \) corresponds to the order of suffixes. That is, the \( i^{th} \) leaf after the root corresponds to the \( i^{th} \) longest suffix. Thus, the string can be reconstructed in linear time: its \( i^{th} \) letter will be equal to the first letter on the edge in the path from the root to the \( i^{th} \) leaf. Note that the cycle and hence the string is not necessarily unique. See Figure 7.4 for an example.

7.3.2 Suffix tour graph of a suffix tree

We now focus on suffix tour graphs of general input trees, which are not necessarily \$-suffix trees. If the input tree \( \tau \) is a suffix tree, but not a \$-suffix tree, the suffix tour graph does not necessarily contain a cycle through the root and the leaves. This is illustrated by the example in Figure 7.5. We therefore have to devise a new approach.

The high level idea of our solution is to try to augment the input tree so that the augmented tree is a \$-suffix tree. More precisely, we will try to augment the suffix tour graph of the tree to obtain a suffix tour graph of a \$-suffix tree. It will be essential to understand how the suffix tour graphs of suffix trees and \$-suffix trees are related.

Let \( ST \) and \( ST_\$ \) be the suffix tree and the \$-suffix tree of a string. We call a leaf of \( ST_\$ \) a \$-leaf if the edge ending at it is labeled by a single letter \$. Note that to obtain \( ST_\$ \) from \( ST \) we must add all \$-leaves, their parents, and suffix links between the consecutive parents to \( ST \). We denote the deepest \$-leaf by \( s \).

An internal node \( x \) of a suffix tour graph has \( d(x) \) incoming arcs produced from edges and \( \ell(x) \) incoming arcs produced from suffix links. All arcs outgoing from \( x \) are produced from edges, and there are \( d(x) + \ell(x) \) of them since suffix tour graphs are Eulerian graphs. A leaf \( x \) of a suffix tour graph has \( d(x) \) incoming arcs produced from edges, \( \ell(x) \) incoming arcs produced from suffix links, and one outgoing arc produced from a suffix link. Below we describe what happens to the values \( d(x) \) and \( \ell(x) \), and to the outgoing arcs produced from suffix links. These two things define the changes to the suffix tour graph.
A SUFFIX TREE OR NOT A SUFFIX TREE?

Lemma 7.8 For the deepest $\$-$leaf $s$ we have $\ell(s) = 0$ and $d(s) = 1$. The $\ell$-values of other $\$-$leaves are equal to one, and their $d$-values are equal to zero.

Proof Suppose that $\ell(s) = 1$. Then there is a leaf $y$ such that $par_s(y) \rightarrow par_s(s)$ is a suffix link in $\sigma$, and the first letter on the edge from $par_s(y)$ to $y$ is $\$. That is, $y$ is a $\$-$leaf and its string depth is bigger than the string depth of $s$, which is a contradiction. Hence, $\ell(s) = 0$ and therefore $d(s) = 1$. The parent of any other $\$-$leaf $y$ will have an incoming suffix link from the parent of the previous $\$-$leaf and hence $\ell(y) = 1$ and $d(y) = 0$.

The important consequence of Lemma 7.8 is that in the suffix tour graph of $ST_s$ all the $\$-$leaves are connected by a path starting in the deepest $\$-$leaf and ending in the root.

Next, we consider nodes that are explicit in $ST$ and $ST_s$. If a node $x$ is explicit in both trees, we denote its (explicit) parent in $ST$ by $par(x)$ and in $ST_s$ — by $par_s(x)$. Below in this section we assume that each edge of $ST$ contains at most one implicit suffix node at distance one from its parent.
Lemma 7.9 Consider a node $x$ of $ST$. If a leaf $y$ contributes to $\ell(x)$ either in $ST$ or $ST_\$s$, and $\text{par}_\$(y)$ and $\text{par}_\$(x)$ are either both explicit or both implicit in $ST$, then $y$ contributes to $\ell(x)$ in both trees.

Proof If $\text{par}_\$(y)$ and $\text{par}_\$(x)$ are explicit, the claim follows straightforwardly.

Consider now the case when $\text{par}_\$(y)$ and $\text{par}_\$(x)$ are implicit. Suppose first that $y$ contributes to $\ell(x)$ in $ST_\$. Then the labels of $\text{par}_\$(y)$ and $\text{par}_\$(x)$ are $La$ and $L[2..]a$ for some string $L$ and a letter $a$. Remember that distances between $\text{par}_\$(y)$ and $\text{par}(y)$ and between $\text{par}_\$(x)$ and $\text{par}(x)$ are equal to one. Therefore, labels of $\text{par}(y)$ and $\text{par}(x)$ are $L$ and $L[2..]$, and the first letters on edges $\text{par}(x) \rightarrow x$ and $\text{par}(y) \rightarrow y$ are equal to $a$. Consequently, $\text{par}(y) \rightarrow \text{par}(x)$ is a suffix link, and $y$ contributes to $\ell(x)$ in $ST$ as well.

Now suppose that $y$ contributes to $\ell(x)$ in $ST$. Then the labels of $\text{par}(y)$ and $\text{par}(x)$ are $L$ and $L[2..]$, and the first letters on the edges $\text{par}(y) \rightarrow y$ and $\text{par}(x) \rightarrow x$ are equal to some letter $a$. This means that the labels of $\text{par}_\$(y)$ and $\text{par}_\$(x)$ are $La$ and $L[2..]a$, and hence there is a suffix link from $\text{par}_\$(y)$ to $\text{par}_\$(x)$. Since $y$ and $x$ are not $\$-leaves, $y$ contributes to $\ell(x)$ in $ST_\$.

Before we defined the deepest $\$-leaf $s$. If the parent of $s$ is implicit in $ST$, the changes between $ST$ and $ST_\$ are more involved. To describe them, we first need to define the twist node. Let $p$ be the deepest explicit parent of any $\$-leaf in $ST$. The node that precedes $p$ in the suffix chain is thus an implicit node in $ST$, i.e., it has two children in $ST_\$, one which is a $\$-leaf and another node $y$, which is either a leaf or an internal node. If $y$ is a leaf, let $t$ be the child of $p$ such that $y$ contributes to $\ell(t)$. We refer to $t$ as the twist node.

Lemma 7.10 Let $x$ be a node of $ST$. Upon transition from $ST$ to $ST_\$, the $\ell$-value of $x = t$ increases by one and the $\ell$-value of its parent decreases by one. If $\text{par}_\$(x)$ is an implicit node of $ST$, then $\ell(x)$ decreases by $\ell(\text{par}_\$(x))$. Otherwise, $\ell(x)$ does not change.

Proof The value $\ell(x)$ can change when (1) A leaf $y$ contributes to $\ell(x)$ in $ST_\$, but not in $ST$; or (2) A leaf $y$ contributes to $\ell(x)$ in $ST$, but not in $ST_\$.

In the first case the nodes $\text{par}_\$(y)$ and $\text{par}_\$(x)$ cannot be both explicit or both implicit. Moreover, from the properties of suffix links we know that if $\text{par}_\$(y)$ is explicit in $ST$, then $\text{par}_\$(x)$ is explicit as well [73]. Consequently, $\text{par}_\$(x)$ is implicit in $ST$, and $\text{par}_\$(x)$ is explicit. Since $\text{par}_\$(x)$ is the first
explicit suffix node and \(y\) is a leaf that contributes to \(\ell(x)\), we have \(x = t\), and \(\ell(x) = \ell(t)\) in \(ST_s\) is bigger than \(\ell(t)\) in \(ST\) by one (see Figure 7.6(a)).

Consider one of the leaves \(y\) satisfying (2). In this case \(\text{par}(y) \rightarrow \text{par}(x)\) is a suffix link, and the first letters on the edges \(\text{par}(y) \rightarrow y\) and \(\text{par}(x) \rightarrow x\) are equal. Since \(y\) does not contribute to \(\ell(x)\) in \(ST_s\), exactly one of the nodes \(\text{par}_s(y)\) and \(\text{par}_s(x)\) must be implicit in \(ST\). Hence, we have two subcases: (2a) \(\text{par}_s(y)\) is implicit in \(ST\), and \(\text{par}_s(x)\) is explicit; (2b) \(\text{par}_s(y)\) is explicit in \(ST\), and \(\text{par}_s(x)\) is implicit.

In the subcase (2a) the distance between \(\text{par}(y)\) and \(\text{par}(x)\) is one. The end of the suffix link for \(\text{par}_s(y)\) must belong to the subtree rooted at \(x\). From the other hand, the string distance from \(\text{par}(x)\) to the end of the suffix link is one. This means that the end of the suffix link is \(x\). Consequently, \(x\) is the parent of the twist node \(t\), and the value \(\ell(x) = \ell(\text{par}_s(t))\) is smaller by one in \(ST_s\) (see Figure 7.6(a)).

In the subcase (2b) the \(\ell\)-value of \(x\) in \(ST\) is bigger than the \(\ell\)-value of \(x\) in \(ST_s\) by \(\ell(\text{par}_s(x))\), as all leaves contributing to \(\text{par}_s(x)\) in \(ST_s\), e.g. \(y\), switch to \(x\) in \(ST\) (see Figure 7.6(b)).

**Lemma 7.11** Let \(x\) be a node of \(ST\). Upon transition from \(ST\) to \(ST_s\), the value \(d(x)\) of a node \(x\) such that \(\text{par}_s(x)\) is implicit in \(ST\) increases by \(\ell(\text{par}_s(x))\). If \(x\) is the twist node \(t\), its \(d\)-value decreases by one. Finally, the \(d\)-values of all ancestors of the deepest \$-leaf \(s\) increase by one.

**Proof** Remember that \(d(x) = |L_x| - \sum_{y \in V_x} \ell(y)\). If \(\text{par}_s(x)\) is implicit in \(ST\), \(\ell(x)\) decreases by \(\ell(\text{par}_s(x))\), i.e. \(d(x)\) increases by \(\ell(\text{par}_s(x))\). Note
that \(d\)-values of ancestors of \(x\) are not affected since for them the decrease of \(\ell(x)\) is compensated by the presence of \(par_S(x)\). The value \(\ell(t)\) increases by one and results in decrease of \(d(t)\) by one, but for other ancestors of \(t\) increase of \(\ell(t)\) will be compensated by decrease of \(\ell(par_S(t))\).

The value \(\ell(s) = 0\) and the \(\ell\)-values of other \$-leaves are equal to one. Consequently, when we add the \$-leaves to \(ST\), \(d\)-values of ancestors of \(s\) increase by one, and \(d\)-values of ancestors of other \$-leaves are not affected. 

**Lemma 7.12** Let \(par_S(x)\) be an implicit parent of a node \(x \in ST\). Then \(d(par_S(x))\) in \(ST_S\) is equal to \(d(x)\) in \(ST\) if the node \(par_S(x)\) is not an ancestor of \(s\), and \(d(x) + 1\) otherwise.

**Proof** First consider the case when \(par_S(x)\) is not an ancestor of \(s\). Remember that the suffix tour graph is an Eulerian graph. The node \(par_S(x)\) has \(\ell(par_S(x))\) incoming arcs produced from suffix links and \(d(x)\) outgoing arcs produced from edges. Hence it must have \(d(x) - \ell(par_S(x))\) incoming arcs produces from edges, and this is equal to \(d(x)\) in \(ST\). If \(par_S(x)\) is an ancestor of \(s\), the \(d\)-value must be increased by one as in the previous lemma.

Speaking in terms of suffix tour graphs, we make local changes when the node is the twist node \(t\) or when the parent of a node is implicit in \(ST\), and add a cycle from the root to \(s\) (increase of \(d\)-values of ancestors of \(s\)) and back via all \$-leaves.

### 7.4 A Suffix Tree Decision Algorithm

Given a tree \(\tau = (V,E)\) annotated with a set of suffix links and a labelling function, we want to decide whether there is a string \(S\) such that \(\tau\) is the suffix tree of \(S\) and it has all the properties described in Lemma 7.4.

We assume that \(\tau\) satisfies Proposition 7.1(1) and Proposition 7.1(2), which can be verified in linear time. We will not violate this while augmenting \(\tau\). If \(\tau\) is a suffix tree, the string depth of a node equals the length of the suffix link path starting at it. Consequently, string depths of all explicit internal nodes and lengths of all internal edges can be found in linear time.

We replace the original problem with the following one: Can \(\tau\) be augmented to become a \$-suffix tree? The deepest \$-leaf \(s\) can either hang from a node of \(\tau\), or from an implicit suffix node \(par_S(s)\) on an edge of \(\tau\). In the latter case the distance from \(par_S(s)\) to the upper end of the edge is equal to one. That is,
there are $O(n)$ possible locations of $s$. For each of the locations we consider a suffix link path starting at its parent. The suffix link paths form a tree which we refer to as the suffix link tree. The suffix link tree can be built in linear time: For explicit locations the paths already exist, and for implicit locations we can build the paths following the suffix link path from the upper end of the edge containing a location and exploiting the knowledge about lengths of internal edges. (Of course, if we see a node encountered before, we stop.)

If $\tau$ is a suffix tree, then it is possible to augment it so that its suffix tour graph will satisfy Proposition 7.1(3) and Lemma 7.7. We remind that Proposition 7.1(3) says that for any node $x$ of the suffix tour graph $d(x) \geq 0$, and Lemma 7.7 says that the suffix tour graph contains a cycle going through the root and all leaves. We show that each of the conditions can be verified for all possible ways to augment $\tau$ by a linear time traverse of $\tau$ or the suffix link tree. We start with Proposition 7.1(3).

**Lemma 7.13** If $\tau$ can be augmented to become a $\$-suffix tree, then $\forall x d(x) \geq -1$.

**Proof** The value $d(x)$ increases only when $x$ is an ancestor of $s$ or when $\text{par}_s(x)$ is implicit in $ST$. In the first case it increases by one. Consider the second case. Remember that $d(\text{par}_s(x))$ is equal to $d(x)$ or to $d(x) + 1$ if it is an ancestor of $s$. Since in a $\$-suffix tree all $d$-values are non-negative, we have $d(x) \geq -1$ for any node $x$.

**Step 1.** We first compute all $d$-values and all $\ell$-values. If $d(x) \leq -2$ for some node $x$ of $\tau$, then $\tau$ cannot be augmented to become a $\$-suffix tree and hence it is not a suffix tree. From now on we assume that $\tau$ does not contain such nodes. All nodes $x$ with $d(x) = -1$, except for at most one, must be ancestors of $s$. If there is a node with a negative $d$-value that is not an ancestor of $s$, then it must be the lower end of the edge containing $\text{par}_s(s)$, and the $d$-value must become non-negative after we augment $\tau$.

We find the deepest node $x$ with $d(x) = -1$ by a linear time traverse of $\tau$. All nodes with negative $d$-values must be its ancestors, which can be verified in linear time. If this is not the case, $\tau$ is not a suffix tree. Otherwise, the possible locations for the parent of $s$ are descendants of $x$ and the implicit location on the edge to $x$ if $d(x) + \ell(x)$, the $d$-value of $x$ after augmentation, is at least zero. We cross out all other locations.
Step 2. For each of the remaining locations we consider the suffix link path starting at its parent. If the implicit node \(q\) preceding the first explicit node \(p\) in the path belongs to a leaf edge then the twist node \(t\) is present in \(\tau\) and will be a child of \(p\). We cannot tell which child though, since we do not know the first letter on the leaf edge outgoing from \(q\). However, we know that \(d(t)\) decreases by 1 after augmentation, and hence \(d(t)\) must be at least 0. Moreover, if \(d(t) = 0\) the twist node \(t\) must be an ancestor of \(s\) to compensate for the decrease of \(d(t)\).

In other words, a possible location of \(s\) is crossed out if the twist node \(t\) is present but \(p\) has no child \(t\) that satisfies \(d(t) > 0\) or \(d(t) = 0\) and \(t\) is ancestor of \(s\). For each of the locations of \(s\) we check if \(t\) exists, and if it does, we find \(p(i1)\). This can be done in linear time in total by a traverse of the suffix link tree. We also compute for every node if it has a child \(u\) such that \(d(u) > 0\) (i2). Finally, we traverse \(\tau\) in the depth-first order while testing the current location of \(s\). During the traverse we remember, for any node on the path to \(s\), its child which is an ancestor of \(s\) (i3). With the information (i1), (i2), and (i3), we can determine if we cross out a location of \(s\) in constant time, and hence the whole computation takes linear time.

Step 3. We assume that the suffix tour graph of \(\tau\) is an Eulerian graph, otherwise \(\tau\) is not a suffix tree by Lemma 7.7. This condition can be verified in linear time. When we augment \(\tau\), we add a cycle \(C\) from the root to the deepest $-$leaf \(s\) and back via $-$leaves. The resulting graph will be an Eulerian graph as well, and one of its connected components (cycles) must contain the root and all leaves of \(\tau\).

We divide \(C\) into three segments: the path from the root to the parent \(par(x)\) of the deepest node \(x\) with \(d(x) = -1\), the path from \(par(x)\) to \(s\), and the path from \(s\) to the root. We start by adding the first segment to the suffix tour graph. This segment is present in the cycle \(C\) for any choice of \(s\), and it might actually increase the number of connected components in the graph. (Remember that if \(C\) contains an edge \(x \rightarrow y\) and the graph contains an edge \(y \rightarrow x\), then the edges eliminate each other.)

The second segment cannot eliminate any edges of the graph, and if it touches a connected component then all its nodes are added to the component containing the root of \(\tau\). Since the third segment contains the $-$leaves only, the second segment must go through all connected components that contain leaves of \(\tau\). We paint nodes of each of the components into some color. And then we perform a depth-first traverse of \(\tau\) maintaining a counter for each color and the total number of distinct colors on the path from the root to the current node. When a color counter becomes equal to zero, we decrease the total number of
colors by one, and when a color counter becomes positive, we increase the total number of colors by one. If a possible location of $s$ has ancestors of all colors, we keep it.

**Lemma 7.14** The tree $\tau$ is a suffix tree iff there is a survived location of $s$.

**Proof** If there is such a location, then for any $x$ in the suffix tour graph of the augmented tree we have $d(x) \geq 0$ and there is a cycle containing the root and all leaves. We are still to apply the local changes caused by implicit parents. Namely, for each node $x$ with an implicit parent the edge from $y$ to $x$ is to be replaced by the path $y, \operatorname{pars}(x), x$ (see Figure 7.6(b)). The cycle can be re-routed to go via the new paths instead of the edges, and it will contain the root and the leaves of $\tau$. Hence, the augmented tree is a $\$-$suffix tree and $\tau$ is a suffix tree.

If $\tau$ is a suffix tree, then it can be augmented to become a $\$-$suffix tree. The parent of $s$ will survive the selection process.

Suppose that there is such a location. Then we can find the parent of the twist node if it exists. The parent must have a child $t$ such that either $d(t) > 0$ or $d(t) = 0$ and $t$ is an ancestor of $s$, and we choose $t$ as the twist node. Let the first letter on the edge to the twist node be $a$. Then we put the first letter on all new leaf edges caused by the implicit nodes equal to $a$. The resulting graph will be the suffix tour graph of a $\$-$suffix tree. We can use the solution of I et al. [84] to reconstruct a string $S\$ realizing this $\$-$suffix tree in linear time. The tree $\tau$ will be a suffix tree of the string $S$. This completes the proof of Theorem 7.2.

### 7.5 Conclusion and Open Problems

We have proved several new properties of suffix trees, including an upper bound of $n - 1$ on the length of a shortest string $S$ realizing a suffix tree $\tau$ with $n$ nodes. As noted this bound is tight in terms of $n$, since the number of leaves in $\tau$, which can be $n - 1$, provides a trivial lower bound on the length of $S$.

Using these properties, we have shown how to decide if a tree $\tau$ with $n$ nodes is a suffix tree in $O(n)$ time, provided that the suffix links of internal nodes and the first letter on each edge is specified. It remains an interesting open question whether the problem can be solved without first letters or, even, without suffix links (i.e., given only the tree structure).

Our results imply that the set of all $\$-$suffix trees is a proper subset of the set all of suffix trees (e.g., the suffix tree of a string $abaabab$ is not a $\$-$suffix tree.
by Lemma 7.7), which in turn is a proper subset of the set of all trees (consider, e.g., Figure 7.1(c) or simply a path of length 2).
BIBLIOGRAPHY


136  

**TOPICS IN COMBINATORIAL PATTERN MATCHING**


