This electronic thesis or dissertation has been downloaded from the King's Research Portal at https://kclpure.kcl.ac.uk/portal/

Data Structures for Strings in the Internal and Dynamic Settings

Charalampopoulos, Panagiotis

Awarding institution:
King's College London

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without proper acknowledgement.

END USER LICENCE AGREEMENT

Unless another licence is stated on the immediately following page this work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International licence. https://creativecommons.org/licenses/by-nc-nd/4.0/

You are free to copy, distribute and transmit the work

Under the following conditions:

- Attribution: You must attribute the work in the manner specified by the author (but not in any way that suggests that they endorse you or your use of the work).
- Non Commercial: You may not use this work for commercial purposes.
- No Derivative Works - You may not alter, transform, or build upon this work.

Any of these conditions can be waived if you receive permission from the author. Your fair dealings and other rights are in no way affected by the above.

Take down policy

If you believe that this document breaches copyright please contact librarypure@kcl.ac.uk providing details, and we will remove access to the work immediately and investigate your claim.

Download date: 26. Dec. 2024

Data Structures for Strings in the Internal and Dynamic Settings

Panagiotis Charalampopoulos

Department of Informatics

King's College London

UK

Thesis submitted for the degree of Doctor of Philosophy.

2020

Abstract

This thesis is devoted to algorithms and data structures for classical problems in stringology in the *internal* and *dynamic* settings.

In the internal setting, the task is to preprocess a string S so that queries of interest for *substrings* of S can be answered efficiently. Note that a substring of S can be specified in constant time by the indices of an occurrence of it as a *fragment* of S. Efficient data structures for problems in the internal setting have proved useful as building blocks in the design of algorithms and data structures for problems on strings.

Here, we consider the Internal Dictionary Matching problem, where one is given a text T of length n and a dictionary $\mathcal{D}$ consisting of substrings of T, each given as a fragment of T. This way, $\mathcal{D}$ takes space proportional to the number $d = |\mathcal{D}|$ of patterns rather than their total length, which could be $\Theta(n \cdot d)$. We show how to construct in $(n+d) \cdot \log^{\mathcal{O}(1)} n$ time a data structure that answers the following types of queries in $\log^{\mathcal{O}(1)} n + \mathcal{O}(|output|)$ time: reporting/counting *all* occurrences of patterns from $\mathcal{D}$ in a fragment $T[i \mathinner{.\,.} j]$ and reporting *distinct* patterns from $\mathcal{D}$ that occur in $T[i \mathinner{.\,.} j]$. We also present data structures for the problem of counting *distinct* patterns from $\mathcal{D}$ that occur in $T[i \mathinner{.\,.} j]$. Finally, we also address these problems for a dynamic dictionary, in which case queries are interleaved with updates to $\mathcal{D}$.

In the dynamic setting, the task is to maintain a collection of strings under updates, such as insertions and deletions of letters, so that information of interest can be efficiently retrieved. We consider the following problems in such a setting:

- Internal Pattern Matching: We show that the data structure of Gawrychowski et al. [SODA 2018] for maintaining a persistent collection $\mathcal{X}$ of strings, of total length at most N, under operations "split", "concatenate" and "insert a string of length 1 to $\mathcal{X}$", in $\mathcal{O}(\log N)$ time each, can be augmented to return the occurrences of any string $P \in \mathcal{X}$ in any string $T \in \mathcal{X}$ in $\mathcal{O}(|T|/|P| \cdot \log^2 N)$ time.

- Repetition Detection: We show how to maintain the longest square substring of a dynamic string, of length at most n, in $\mathcal{O}(\log^3 n)$ time per update, using

$\mathcal{O}(n)$ space. One of the main ingredients of our algorithm is our aforementioned implementation of internal pattern matching queries in the dynamic setting.

- Longest Common Factor: We show that a longest common factor (equiv. substring) of two dynamic strings, of total length at most n, can be maintained in amortised $\mathcal{O}(\log^8 n)$ time per update, using $\tilde{\mathcal{O}}(n)$ space. An exponentially faster algorithm with polynomial space requirements is ruled out due to a lower bound in the cell-probe model of computation [Charalampopoulos et al., ICALP 2020]. We also show more efficient solutions for the case where one of the two strings is static.

- String Alignment: We consider the problem of dynamically maintaining an optimal alignment of two strings, each of length at most n, as they undergo edit operations. The string alignment problem generalises the longest common subsequence (LCS) problem and the edit distance problem (as long as insertions and deletions cost the same). The conditional lower bound of Backurs and Indyk [SICOMP 2018] for computing the LCS in the static case implies that strongly sublinear update time for the problem in scope is unlikely. We essentially match this lower bound when the alignment weights are constants, by showing how to process each update in $\tilde{\mathcal{O}}(n)$ time. When the weights are integers, bounded in absolute value by some $w = n^{\mathcal{O}(1)}$, we can maintain the alignment in $n \cdot \min\{\sqrt{n}, w\} \cdot \log^{\mathcal{O}(1)} n$ time per update.

Interestingly, we use a wide range of tools in order to obtain our results, such as: string periodicity, locally consistent parsing of strings, orthogonal range queries, efficient $(\min, +)$-multiplication of simple unit-Monge matrices, and data structures for computing distances in planar graphs.

Acknowledgements

First, I thank my primary supervisor, Tomasz Radzik, for his guidance during the last two and a half years of my PhD studies. I also thank my second supervisor and closest collaborator, Solon Pissis, for introducing me to research on string algorithms and for his continuous support. Special thanks go to my colleagues and friends Lorraine Ayad, Ritu Kundu, Manal Mohamed and Steven Watts for their companionship during my time at King's and our stimulating working sessions.

Moreover, I am grateful for having the pleasure to work with and learn from my friend Jakub Radoszewski throughout the last four years. I cherish the time that I spent in the University of Warsaw, during which I also had the opportunity to work with Adam Karczmarz, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba.

Furthermore, I wish to express my deep gratitude to Shay Mozes for hosting me at IDC Herzliya, introducing me to algorithms for planar graphs, and supporting me during my time in Israel and beyond. During this time, I also had the pleasure to work with Benjamin Tebeka and Oren Weimann, and to regularly meet and work with Amihood Amir, Itai Boneh and Eitan Kondratovsky at Bar-Ilan University.

I also especially thank Paweł Gawrychowski and Tomasz Kociumaka who have generously shared parts of their deep knowledge of algorithms with me, and have motivated me by example to become a better researcher.

During my PhD studies I had the chance to work with many people. I thank the following coauthors of mine for our fruitful collaboration: Michał Adamczyk, Hayam Alamro, Yannis Almirantis, Amihood Amir, Lorraine A. K. Ayad, Carl Barton, Itai Boneh, Huiping Chen, Peter Christen, Maxime Crochemore, Gabriele Fici, Jia Gao, Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Eitan Kondratovsky, Chang Liu, Grigorios Loukides, Robert Mercaş, Manal Mohamed, Shay Mozes, Nadia Pisanti, Solon P. Pissis, Karol Pokorski, Dimitris Polychronopoulos, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Wing-Kin Sung, Benjamin Tebeka, Tomasz Waleń, Oren Weimann, Philip Wellnitz, and Wiktor Zuba.

I would also like to thank my examiners Artur Czumaj and Tatiana Starikovskaya for taking the time to read my thesis and for their valuable suggestions.

In addition, I wish to acknowledge financial support from a KCL Faculty of Natural & Mathematical Sciences Studentship (Graduate Teaching Scholarship up to early 2018) and from an A. G. Leventis Foundation grant.

I am indebted to my close friends who have been with me in the ups and the downs.

Finally, I wholeheartedly thank my parents, Dimitris and Chara, and my brother, Konstantinos, for their unconditional love and support. This thesis is dedicated to them.

Papers this Thesis is Based on

This thesis has drawn material from a number of papers I have co-authored.

Chapter 3 is based on [44, 45]:

– P. Charalampopoulos, T. Kociumaka, M. Mohamed, J. Radoszewski, W. Rytter, J. Straszyński, T. Waleń, and W. Zuba: *Counting Distinct Patterns in Internal Dictionary Matching*, In 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020.

– P. Charalampopoulos, T. Kociumaka, M. Mohamed, J. Radoszewski, W. Rytter, and T. Waleń: *Internal Dictionary Matching*, In 30th International Symposium on Algorithms and Computation, ISAAC 2019.

Chapter 4 is based on [47]:

– P. Charalampopoulos, T. Kociumaka, and P. Wellnitz: *Faster Approximate Pattern Matching: A Unified Approach*, In 61st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2020.

Chapter 5 builds upon [8]:

– A. Amir, I. Boneh, P. Charalampopoulos, and E. Kondratovsky: *Repetition Detection in a Dynamic String*, In 27th Annual European Symposium on Algorithms, ESA 2019.

Chapter 6 is based on [42, 10]:

– P. Charalampopoulos, P. Gawrychowski, and K. Pokorski: *Dynamic Longest Common Substring in Polylogarithmic Time*, In 47th International Colloquium on Automata, Languages, and Programming, ICALP 2020.

– A. Amir, P. Charalampopoulos, S. P. Pissis, and J. Radoszewski: *Dynamic and Internal Longest Common Substring*, Algorithmica (2020).

Chapter 7 is based on [46]:

– P. Charalampopoulos, T. Kociumaka, and S. Mozes: *Dynamic String Alignment*, In 31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020.

Contents

1 Introduction **12**

1.1 Internal Queries in Strings 12

1.2 Internal Dictionary Matching 13

1.3 Dynamic Strings 18

1.4 Dynamic Repetition Detection 21

1.5 Dynamic Longest Common Factor 22

1.6 Dynamic String Alignment 26

2 Preliminaries **29**

2.1 Model of Computation 29

2.2 Strings 29

2.3 Periodicity 30

2.4 Tries and Trees 31

2.5 Data Structures for Range Queries 32

2.6 Internal Queries in Strings 34

2.7 Straight-Line Programs and Recompression 36

2.8 Dynamic Strings 39

3 Internal Dictionary Matching **42**

3.1 Exists(i, j) and Report(i, j) Queries 43

3.2 ReportDistinct(i, j) Queries 45

3.2.1 Processing an Aperiodic k-Dictionary 47

3.2.2 Processing a Periodic k-Dictionary 47
3.2.3 Reducing the Space 50
3.3 COUNT(i, j) Queries 51
3.3.1 An Auxiliary Problem 52
3.3.2 Using Recompression 56
3.4 COUNTDISTINCT(i, j) Queries 58
3.4.1 2-Approximate Counting 59
3.4.2 Time-Space Tradeoffs for Exact Counting 65
3.5 Dynamic Dictionaries 68
3.6 Internal Counting of Distinct Squares 75

4 Internal Pattern Matching in a Dynamic Collection of Strings 80
4.1 The Algorithm 80

5 Dynamic Longest Square Substring 85
5.1 Strategy 85
5.2 Implementation 86

6 Internal and Dynamic Longest Common Factor 94
6.1 Internal LCF queries 95
6.1.1 A Lower Bound Based on Set Disjointness 95
6.1.2 Internal Queries for Special Substrings 97
6.1.3 CONCAT LCF queries 100
6.2 Partially Dynamic LCF 102
6.3 Fully Dynamic LCF 106
6.3.1 Locally Consistent Parsing 107
6.3.2 Anchoring the LCF 111
6.3.3 A Problem on Dynamic Bicoloured Trees 115
6.3.4 Dynamic Best Bichromatic Point 120

7 Dynamic String Alignment **123**

7.1 The Alignment Graph and Internal LCS 124

7.2 Main Algorithm 129

7.2.1 Supporting Only Substitutions 129

7.2.2 Supporting Insertions and Deletions 130

7.2.3 Extension to String Alignment Under Integer Weights 134

7.3 Handling Large Weights 135

7.3.1 Data Structures for Planar Graphs 135

7.3.2 Direct Application to String Alignment 136

List of Tables

1.1 Some of our results on Internal Dictionary Matching. 16

1.2 Our results for CountDistinct(i, j) queries. 17

List of Figures

2.1 A straight-line program and its parse tree. 37
2.2 An illustration for Lemma 2.7.2. 39

3.1 A $\mathcal{D}$-modified suffix tree. 43
3.2 An illustration of our use of coloured range reporting queries in our solution for ReportDistinct(i, j). 46
3.3 The construction of rectangles in the proof of Lemma 3.3.3. 54
3.4 The setting in Lemma 3.4.4. 60
3.5 Using basic factors for 2-approximating CountDistinct(i, j). 65
3.6 The setting in Theorem 3.4.14. 66
3.7 Reduction of BSq to BSq'; the case where $|F_1| \leq b$. 78
3.8 The setting in Lemma 3.6.6. 78

5.1 The setting in Lemma 5.2.1. 87
5.2 An illustration of the setting in Case 1 in the proof of Lemma 5.2.4. . . . 89
5.3 A position that is contained in $\Omega(n)$ squares and runs. 91

6.1 An uncompressed parse tree and a pair of layers of nodes. 109
6.2 Best bichromatic points via range trees. 121

7.1 The alignment graph. 125
7.2 The alignment graph and semi-local LCS. 127

Chapter 1

Introduction

Throughout this thesis, we consider strings over an integer alphabet Σ of size σ, polynomially bounded in the total size N of the input (i.e. each letter is in $[1, N^c]$ for some constant c). We use $\tilde{\mathcal{O}}(\cdot)$ notation to suppress $\log^{\mathcal{O}(1)} N$ factors.

1.1 Internal Queries in Strings

Internal queries in strings have received much attention in recent years. In the internal setting, one is to preprocess a string S of length $n := |S|$, so that queries about substrings of S can be answered efficiently. Note that a substring of S can be specified in $\mathcal{O}(1)$ time by the indices i, j of an occurrence of it as a fragment $S[i \mathinner{.\,.} j]$ of S. Data structures for answering internal queries are interesting in their own sake, but also have numerous applications in the design of algorithms and (more sophisticated) data structures in stringology.

The most basic and widely used internal query is that of computing the *longest common prefix* of two suffixes $S[i \mathinner{.\,.} n]$ and $S[j \mathinner{.\,.} n]$ of S, denoted by $\mathsf{LCP}(i, j)$. It has been known for decades, that one can construct the suffix tree of S and a lowest common ancestor data structure for it in $\mathcal{O}(n)$ time [67, 94], thus obtaining an $\mathcal{O}(n)$-space data structure that answers queries in $\mathcal{O}(1)$ time. However, in the word RAM model of computation, with word size $\Theta(\log n)$ this is not necessarily optimal. A sequence of

works [157, 133, 33] has culminated in the recent optimal data structure of Kempa and Kociumaka, that can be built in $\mathcal{O}(n/\log_\sigma n)$ time and answers queries in $\mathcal{O}(1)$ time [107].

The *Internal Pattern Matching* (IPM) problem consists in preprocessing a string S of length n so that we can efficiently compute the occurrences of a substring of S in another substring of S. For the decision version of the problem, a nearly-linear size data structure that allows for sublogarithmic-time IPM queries was presented by Keller et al. [106]. A linear-size data structure allowing for constant-time IPM queries in the case where the ratio between the lengths of the two substrings is constant was presented by Kociumaka et al. [115]. The $\mathcal{O}(n)$-time construction algorithm of the latter data structure was derandomised in [112]. The authors of [115], using their efficient IPM queries as a subroutine, managed to show efficient solutions for other internal problems, such as for computing the periods of a substring (*period queries*, introduced in [114]), and for checking whether two substrings are rotations of one another (*cyclic equivalence queries*). For a variant of the IPM problem called *cross-document pattern matching*, where the input is a collection of texts see [120]. Other queries that have been studied include the computation of the lexicographically minimal or maximal suffix, and the lexicographically minimal rotation of a queried substring [23, 111]. We refer the interested reader to the PhD dissertation of Tomasz Kociumaka [112], which contains an overview of the literature.

As discussed in detail in Section 1.3, in Chapter 4, we show how to efficiently answer IPM queries in the dynamic setting. This implies that other internal queries, such as period queries and cyclic equivalence queries can be also answered efficiently in this setting, due to the (implicit) reductions of [115].

1.2 Internal Dictionary Matching

The dictionary matching problem has been studied for more than 40 years [3]. In this problem, we are to preprocess a dictionary $\mathcal{D}$, consisting of d patterns, in order to be able to efficiently compute the occurrences of the patterns from $\mathcal{D}$ in any given text T.

The Aho-Corasick automaton preprocesses the dictionary in linear time with respect to its total length and then processes each input text T in time $\mathcal{O}(|T| + |output|)$ [3, 64]. Compressed indexes for dictionary matching were presented in [38]. Different variants of the approximate dictionary matching problem [135, 25] have been studied, such as its indexing version [53] and its streaming version [81]. Dynamic dictionary matching in its more general version consists in the problem where a dynamic dictionary is maintained, texts are presented as input, and for each such text all the occurrences of patterns from the dictionary in the text have to be reported; see [11, 12, 85].

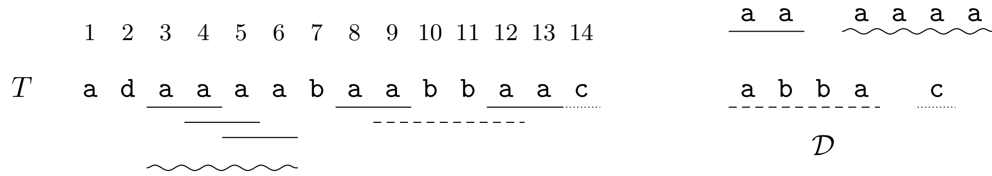
We introduce the *Internal Dictionary Matching* (IDM) problem that consists in answering the following types of queries for an internal dictionary $\mathcal{D}$ consisting of substrings of a text T of length n: given (i, j), report/count all occurrences of patterns from $\mathcal{D}$ in $T[i \mathinner{.\,.} j]$, and report/count the distinct patterns from $\mathcal{D}$ that occur in $T[i \mathinner{.\,.} j]$.

The IDM problem in the special case of the dictionary $\mathcal{D}$ being the set of all palindromes in T has already been studied in [146] by Rubinchik and Shur, who proposed a data structure of size $\mathcal{O}(n \log n)$ that returns the number of all distinct palindromes in $T[i \mathinner{.\,.} j]$ in $\mathcal{O}(\log n)$ time. Let us note that in this case the total length of patterns might be quadratic in n, but the internal dictionary is of size $\mathcal{O}(n)$ and can be constructed in $\mathcal{O}(n)$ time [90]. Our general solution can be applied, in particular, to this and other dictionaries that satisfy these requirements, such as the dictionaries of all square or non-primitive substrings of T [90, 59, 27].

Let us now formally define the problem and the types of queries that we consider.

Internal Dictionary Matching
Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns, each given as a fragment $T[a \mathinner{.\,.} b]$ of T (represented only by integers a, b).
Queries:
$\textsc{Exists}(i, j)$: Decide whether at least one pattern $P \in \mathcal{D}$ occurs in $T[i \mathinner{.\,.} j]$.
$\textsc{Report}(i, j)$: Report all occurrences of all the patterns of $\mathcal{D}$ in $T[i \mathinner{.\,.} j]$.
$\textsc{ReportDistinct}(i, j)$: Report all patterns $P \in \mathcal{D}$ that occur in $T[i \mathinner{.\,.} j]$.
$\textsc{Count}(i, j)$: Count all occurrences of all the patterns of $\mathcal{D}$ in $T[i \mathinner{.\,.} j]$.
$\textsc{CountDistinct}(i, j)$: Count all patterns $P \in \mathcal{D}$ that occur in $T[i \mathinner{.\,.} j]$.

Example 1.2.1. Let us consider the dictionary $\mathcal{D} = \{\texttt{aa}, \texttt{aaaa}, \texttt{abba}, \texttt{c}\}$ and the text $T = \texttt{adaaaabaabbaac}$.



We then have:

$$\begin{aligned}
\textsc{Exists}(2,12) &= \textbf{true} \\
\textsc{Report}(2,12) &= \{(\texttt{aa},3),(\texttt{aaaa},3),(\texttt{aa},4),(\texttt{aa},5),(\texttt{aa},8),(\texttt{abba},9)\} \\
\textsc{Count}(2,12) &= 6 \\
\textsc{ReportDistinct}(2,12) &= \{\texttt{aa},\texttt{aaaa},\texttt{abba}\} \\
\textsc{Exists}(1,3) &= \textbf{false} \\
\textsc{CountDistinct}(2,12) &= 3 \\
\textsc{CountDistinct}(5,12) &= 2.
\end{aligned}$$

In particular, the two distinct patterns from $\mathcal{D}$ that occur in $T[5 \mathinner{.\,.} 12]$ are $\texttt{aa}$ and $\texttt{abba}$.

Let us briefly consider two straightforward approaches for answering $\textsc{Report}(i, j)$ queries. One could answer each such query in time $\mathcal{O}(j - i + |output|)$ by running $T[i \mathinner{.\,.} j]$ over the Aho-Corasick automaton of $\mathcal{D}$ [3] or in time $\tilde{\mathcal{O}}(d + |output|)$ by performing internal

pattern matching [115] for each element of $\mathcal{D}$ individually. None of these approaches is satisfactory as they can require $\Omega(n)$ time in the worst case.

We propose an $\tilde{\mathcal{O}}(n+d)$-size data structure, which can be built in time $\tilde{\mathcal{O}}(n+d)$ and answers all types of IDM queries, apart from $\textsc{CountDistinct}(i,j)$ queries, in time $\tilde{\mathcal{O}}(1+|output|)$. The exact complexities are shown in Table 1.1.

Table 1.1: The complexities of the proposed data structures for queries $\textsc{Exists}(i,j)$, $\textsc{Report}(i,j)$, $\textsc{ReportDistinct}(i,j)$, and $\textsc{Count}(i,j)$.

Query	Preprocessing time	Space	Query time
$\textsc{Exists}(i,j)$	$\mathcal{O}(n+d)$	$\mathcal{O}(n)$	$\mathcal{O}(1)$
$\textsc{Report}(i,j)$	$\mathcal{O}(n+d)$	$\mathcal{O}(n+d)$	$\mathcal{O}(1+\|output\|)$
$\textsc{ReportDistinct}(i,j)$	$\mathcal{O}(n\log n+d)$	$\mathcal{O}(n+d)$	$\mathcal{O}(\log n+\|output\|)$
$\textsc{Count}(i,j)$	$\mathcal{O}(\frac{n\log n}{\log\log n}+d\log^{3/2}n)$	$\mathcal{O}(n+d\log n)$	$\mathcal{O}(\frac{\log^2 n}{\log\log n})$

Our solutions for $\textsc{Exists}(i,j)$ and $\textsc{Report}(i,j)$ are rather simple and make use of the suffix tree of T. For $\textsc{ReportDistinct}(i,j)$ queries, we mostly rely on the periodic structure of strings. Our solution for $\textsc{Count}(i,j)$ is more involved, and is based on a combination of a locally consistent parsing [102, 103, 98] with orthogonal range searching [30].

Our results for $\textsc{CountDistinct}(i,j)$ queries are presented in Table 1.2. Note that we also consider a special case of $\textsc{CountDistinct}(i,j)$ queries, in which the dictionary $\mathcal{D}$ is the set of all squares (i.e. strings of the form UU) in T.

The main ingredients of our data structures are, again, periodicity, orthogonal range queries, as well as other auxiliary internal queries in strings. We leave the problem of whether an $\tilde{\mathcal{O}}(n+d)$-space data structure that answers $\textsc{CountDistinct}(i,j)$ queries in $\tilde{\mathcal{O}}(1+|output|)$ time exists open for further investigation.

Table 1.2: Our results for CountDistinct(i, j) queries—m is an arbitrary parameter.

Variant	Preprocessing time	Space	Query time
2-approximation	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(n+d)$	$\tilde{\mathcal{O}}(1)$
exact	$\tilde{\mathcal{O}}(n^2/m+d)$	$\tilde{\mathcal{O}}(n^2/m^2+d)$	$\tilde{\mathcal{O}}(m)$
exact	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(nd/m+d)$	$\tilde{\mathcal{O}}(m)$
$\mathcal{D} =$ squares, exact	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(n \log^2 n)$	$\mathcal{O}(\log n)$

A natural problem would be to consider a dynamic dictionary, in the sense that one would perform interleaved IDM queries and updates to $\mathcal{D}$ (insertions/deletions of patterns). For some natural dynamic problems, the best known bounds on the query and the update time are of the form $\mathcal{O}(m^\alpha)$, where m is the size of the input and α is some constant. Henzinger et al. [95] introduced the Online Boolean Matrix-Vector Multiplication product (OMv) conjecture that can be used to provide some justification for the polynomial-time hardness of many such dynamic problems in a unified manner.

We reduce the OMv problem to the internal dictionary matching problem with a dynamic dictionary. Interestingly, in our lower bound construction we only add single-letter patterns to an initially empty dictionary.

Theorem 1.2.2. *The OMv conjecture implies that there is no algorithm that preprocesses T and $\mathcal{D}$ in time polynomial in n, performs insertions to $\mathcal{D}$ in time $\mathcal{O}(n^\alpha)$, answers* Exists(i, j) *queries in time $\mathcal{O}(n^\beta)$, in an online manner, such that $\alpha + \beta = 1 - \epsilon$ for $\epsilon > 0$, and has error probability at most 1/3.*

Finally, by building upon our solutions for static dictionaries, we provide algorithms for the case of a dynamic dictionary, where patterns can be added to or removed from $\mathcal{D}$. We show how to process updates in $\tilde{\mathcal{O}}(n^\alpha)$ time and answer each of the following queries in $\tilde{\mathcal{O}}(n^{1-\alpha} + |output|)$ time for any $0 < \alpha < 1$: Exists(i, j), Report(i, j),

REPORTDISTINCT(i,j), COUNT(i,j), and 2-approximate COUNTDISTINCT(i,j). In particular, for all queries, apart from COUNTDISTINCT(i,j) ones, we match—up to subpolynomial factors—our conditional lower bound.

1.3 Dynamic Strings

The main goal in the dynamic setting is to provide algorithms that are faster than recomputing the answer from scratch after every update. Let us review some basic problems on strings in the dynamic setting.

Finding all occurrences of a pattern of length m in a *static* text can be performed in the optimal $\mathcal{O}(m + |output|)$ time using suffix trees, which can be constructed in linear time [170, 67]. A very basic problem is that of maintaining a dynamic text while enabling efficient pattern matching queries, with the pattern being given explicitly at query time. This is clearly motivated by the possible application in a text editor, where the text is dynamic and the user may issue pattern matching queries. A considerable amount of work has been carried out on this problem [91, 68, 69]. The first data structure achieving polylogarithmic update time and optimal query time was designed by Sahinalp and Vishkin [150]. The update time was improved to $\mathcal{O}(\log^2 n \log\log n \log^* n)$ at the cost of randomisation and polylogarithmic additional time per query by Alstrup et al. [4]. Recently, Gawrychowski et al. [82] presented a randomised data structure that requires $\mathcal{O}(\log^2 n)$ time per update and allows for time-optimal queries. The setting of a dynamic text and a static pattern has also been considered [14]. See [13] for another variant.

Another well-studied problem is that of maintaining a dynamic collection $\mathcal{X}$ of strings in order to support $\mathcal{O}(1)$-time string equality operations, under the following update operations:

- $\mathsf{makestring}(U)$: Insert a non-empty string U to $\mathcal{X}$.
- $\mathsf{concat}(U,V)$: Insert UV to $\mathcal{X}$, for $U,V \in \mathcal{X}$.
- $\mathsf{split}(U,i)$: Insert $U[1\mathinner{.\,.} i-1]$ and $U[i\mathinner{.\,.}|U|]$ in $\mathcal{X}$. for $U \in \mathcal{X}$ and $i \in [2,|U|]$.

This line of research was initiated by Sundar and Tarjan [156] who presented a data structure with amortised polynomial update time. The first data structures supporting updates in polylogarithmic time were presented by Mehlhorn et al. [129]; the authors presented a deterministic data structure and a randomised one with better update time. Alstrup et al. [4] presented a randomised data structure with faster update time, which also allowed for efficiently computing the length $\mathsf{LCP}(U, V)$ of the longest common prefix of any two strings U, V in the collection. Finally, Gawrychowski et al. [82] managed to prove the following result.

Theorem 1.3.1 ([82])**.** *A collection $\mathcal{X}$ of non-empty strings of total length at most N can be dynamically maintained with update operations $\mathsf{makestring}(U)$, $\mathsf{concat}(U, V)$, $\mathsf{split}(U, i)$ requiring time $\mathcal{O}(\log N + |U|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, all w.h.p, so that $\mathsf{LCP}(U, V)$ queries for $U, V \in \mathcal{X}$ can be answered in time $\mathcal{O}(1)$.*

The data structure underlying Theorem 1.3.1 is Las Vegas randomised: the answers are correct, but the update times are guaranteed only *with high probability* (*w.h.p.*), i.e. probability $1 - 1/n^{\Omega(1)}$, where n is the total size of the input, including updates. (This is the only type of randomisation we allow throughout this thesis.) Note that the size of a $\mathsf{makestring}(U)$ operation is $|U|$, while the size of $\mathsf{concat}(U, V)$ and $\mathsf{split}(U, i)$ operations is $\mathcal{O}(1)$. The data structure of Theorem 1.3.1 maintains a parse tree for each string in the collection, and the time complexities depend on the height of each such parse tree, which is $\mathcal{O}(\log N)$ w.h.p.

We show that the data structure of Gawrychowski et al. [82] can also efficiently return all exact occurrences of any string $P \in \mathcal{X}$ in any string $T \in \mathcal{X}$, namely in $\mathcal{O}(|T|/|P| \cdot \log^2 N)$ time. This is formalised in the following statement.

Theorem 1.3.2. *A collection $\mathcal{X}$ of non-empty strings of total length at most N can be dynamically maintained with update operations $\mathsf{makestring}(U)$, $\mathsf{concat}(U, V)$, $\mathsf{split}(U, i)$ requiring time $\mathcal{O}(\log N + |U|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, so that the occurrences of a string $P \in \mathcal{X}$ in a string $T \in \mathcal{X}$ can be computed in time $\mathcal{O}(|T|/|P| \cdot \log^2 N)$. All running times hold w.h.p.*

The above theorem actually constitutes in an implementation of internal pattern matching queries in this dynamic setting: upon an IPM query, we can first perform a constant number of split operations to add the relevant substrings in the collection $\mathcal{X}$, and then employ our pattern matching algorithm. This result, can be seen as an analogue of the $\mathcal{O}(n)$-space, $\mathcal{O}(|T|/|P|)$-query-time data structure of Kociumaka et al. for answering IPM queries for a static string of length n [115].

The implementation of IPM queries in this setting has already proved useful. The recent meta-algorithm of [47] for approximate pattern matching (under the Hamming and edit distances), which relies on certain primitive operations, including IPM queries, can be applied to extend the data structure of Gawrychowski et al. [82] to report all occurrences of a string $P \in \mathcal{X}$ in a string $T \in \mathcal{X}$ with up to k mismatches (resp. errors) in time $\mathcal{O}(|T|/|P| \cdot k^2 \log^2 N)$ (resp. $\mathcal{O}(|T|/|P| \cdot k^4 \log^2 N)$). We also utilise this result on IPM queries in Chapter 5 for efficiently detecting repetitions in a dynamic string.

For all problems considered in this thesis in a dynamic setting, other than the IPM queries discussed above, we consider the following more restricted dynamic setting. We maintain a constant number of strings that undergo (a subset of) the following *edit* operations: letter insertions, deletions, and substitutions. Note that, each edit operation can be simulated with a constant number of concat, split, and makestring(a) (for $a \in \Sigma$) operations, and hence, for instance, the data structure of [82] can be of use in this setting as well. We formalise this in Section 2.8

Problems on strings that have been recently studied in the dynamic setting, apart from the ones discussed in the remaining sections of this chapter, include maintaining:

- two dynamic strings T and P, in order to be able to efficiently compute the Hamming distance or the inner product of P and any given fragment of T [51];
- (an approximation of) the length of a longest increasing subsequence of a dynamic string [130, 77, 117];
- a longest palindromic substring of a dynamic string [10, 6];

- the suffix array of a dynamic string T, with applications such as retrieving the letters of the Burrows-Wheeler transform of T [7];
- the dynamic time warping distance of two dynamic strings [139];
- the Lyndon factorisation of a dynamic string [10];
- a dynamic 2D-string T, and a static 2D-string P, allowing for checking whether P equals some 2D-substring of T, and (approximate-matching) variants of this problem [50].

1.4 Dynamic Repetition Detection

A non-empty string UU is called a *square* or a *tandem repeat*. Squares are a fundamental notion in word combinatorics, and algorithms for finding all squares have been sought as early as the 1980's [54, 20, 126].

A *run* (also known as a *maximal repetition*) is a periodic fragment of the text that cannot be extended to either direction without increasing its period [126]. Kolpakov and Kucherov, in their seminal paper [119], showed that there are $\mathcal{O}(n)$ runs in a text of length n, and presented an algorithm to compute them in $\mathcal{O}(n)$ time. They also posed the so-called *runs conjecture*, which stated that a string of length n cannot have more than n runs. After a long line of research [56, 57, 58, 83, 144, 148, 149], the breakthrough result of Bannai et al. [26] positively resolved the runs conjecture. Runs have been extensively used as an algorithmic tool, for example, for extracting the k-powers in a string (note that a square is a 2-power) and for answering period queries [59, 114, 115]. Here, we extensively utilise runs in our solutions for some of the variants of the IDM problem in Chapter 3.

Each position of a string of length n can be contained in as many as $\Omega(n)$ squares/runs. (For squares, it suffices to simply consider string $\mathtt{a}^n$; an analogous example for runs is more cumbersome and is presented in Chapter 5.) Thus, in the dynamic setting, a single update in a string may result in the destruction/creation of $\Omega(n)$ squares/runs.

Fortunately, however, in such "bad" cases, the squares/runs have extra structure. A square UU is *primitively rooted* if U is a primitive string. By the well-known three squares lemma [61], it follows that only $\mathcal{O}(\log n)$ primitively rooted squares can start at any position. Roughly speaking, this implies that $\mathcal{O}(\log n)$ runs that are not "very periodic" can contain each position.

Here, we consider the problem of maintaining the longest square of a dynamic string, and present the following result.

Theorem 1.4.1. *The longest square of a dynamic string of length n that undergoes substitution operations can be maintained in $\mathcal{O}(\log^3 n)$ time per each such operation, using $\mathcal{O}(n)$ space, after an $\mathcal{O}(n \log^2 n)$-time preprocessing. All running times hold w.h.p.*

Our algorithm relies on a novel implicit neat characterisation of all squares of the string in terms of "very periodic" runs, based on the discussed insights, and on being able to efficiently answer LCE and IPM queries in a dynamic string (cf. Theorems 1.3.1 and 1.3.2).

The considered problem actually captures the essence of repetition detection in dynamic strings. Only straightforward modifications to our algorithm are required for the maintenance of all squares, while the developed combinatorial insights also extend for runs [8]. See [8] for $n^{o(1)}$-time dynamic algorithms for the maintenance of all squares/runs.

Let us remark that the algorithm underlying Theorem 1.4.1 is faster than the one presented in [8] for the same problem. This improvement comes from employing the newly developed efficient IPM queries for a dynamic string, discussed in Section 1.3, in the framework of [8].

An interesting future research direction is that of exploring whether more algorithms and data structures can benefit from our novel characterisation of squares.

1.5 Dynamic Longest Common Factor

In the well-known *longest common factor* (LCF) or *longest common substring* problem, we are given two strings S and T, each of length at most n, and are asked to compute

a longest substring X of S that is a substring of T.[1] This problem was conjectured by Knuth to require $\Omega(n \log n)$ time. However, in his seminal paper that introduced the suffix tree, Weiner showed a linear-time solution (for constant-size alphabets) [170]. Knuth declared Weiner's algorithm the "Algorithm of the Year" [19]. Since then, many different versions of this classical question were considered, such as obtaining a tradeoff between the time and the working space [154, 118, 140], and computing an approximate LCF under either the Hamming or the edit distance (see [158, 40, 116, 21, 88] and references therein).

Here, we start by studying the LCF problem in the internal model. That is, we consider the problem of preprocessing two strings S and T, with the aim of being able to efficiently answer the following type of queries: compute an LCF of a substring of S and a substring of T. We show a hardness result for the general case of this problem, conditional on the hardness of the set disjointness problem [87, 122]. However, we manage to show efficient data structures for useful restricted cases, based on ingredients such as the suffix tree and orthogonal range queries.

Then, we turn to the dynamic model. As mentioned in [116], an answer to the LCF problem "is not robust and can vary greatly when the input strings are changed even by one character". This implicitly poses the following question: "Can we compute an LCF after editing S or T in $o(n)$ time?". Formally, we consider the problem of maintaining two strings S and T that undergo edit operations (i.e. letter insertions, deletions, and substitutions), returning, after each update, the length of an LCF and the starting positions of one of its occurrences in each of the strings.

Example 1.5.1. The length of an LCF of S and T below is *doubled* when substitution $S[4] := \mathtt{a}$ is performed. The next substitution, $T[3] := \mathtt{b}$, *halves* the length of an LCF.

$S = \mathtt{c}\underline{\mathtt{aab}}\mathtt{aaa}$ $\quad S[4] := \mathtt{a}$ $\quad S = \mathtt{c}\underline{\mathtt{aaaaaa}}$ $\quad T[3] := \mathtt{b}$ $\quad S = \mathtt{c}\underline{\mathtt{aaa}}\mathtt{aaa}$

$T = \mathtt{aaaa}\underline{\mathtt{aab}}$ $\quad T = \underline{\mathtt{aaaaaa}}\mathtt{b}$ $\quad T = \mathtt{aab}\underline{\mathtt{aaa}}\mathtt{b}$

[1] We choose to call this problem longest common factor and to abbreviate it as LCF in order to avoid confusion with the longest common subsequence problem that we abbreviate as LCS.

In [9], we (Amir et al.) initiated the study of this question in the dynamic setting by considering the problem of constructing a data structure over two strings that returns the LCF after a single edit operation in one of the strings. However, in this solution, after each edit operation, the string is immediately reverted to its original version. Abedin et al. [2] improved the tradeoffs for this problem by designing a more efficient solution for the so-called heaviest induced ancestors (HIA) problem. Amir and Boneh [5] investigated some special cases of the *partially dynamic LCF* problem (in which one of the strings is assumed to be static); namely, the case where the static string is periodic and the case where the substitutions in the dynamic string are substitutions with some letter $\# \notin \Sigma$. Next, in [10], we (Amir et al.) presented the first algorithm for the *fully dynamic LCF* problem (in which both strings are subject to updates) with sublinear update time, namely $\tilde{\mathcal{O}}(n^{2/3})$ time. As a stepping stone towards this result, we designed an algorithm for the partially dynamic LCF problem that processes each edit operation in $\tilde{\mathcal{O}}(\sqrt{n})$ time. This brought the question of determining if the bound on the update time in the dynamic LCF problem should be polynomial or subpolynomial. Here, we settle this question. As a warm-up, we present the following (relatively simple) deterministic solution for the partially dynamic LCF problem.

Theorem 1.5.2. *We can maintain an LCF of a dynamic string S and a static string T, each of length at most n,*

(a) in $\mathcal{O}(\log n \log\log n)$ time per substitution operation using $\mathcal{O}(n \log^2 n)$ space, after an $\mathcal{O}(n\log^2 n)$-time preprocessing, or

(b) in $\mathcal{O}(\log\log n)$ time per substitution operation using $\mathcal{O}(n^{1+\epsilon})$ space, after an $\mathcal{O}(n^{1+\epsilon})$-time preprocessing, for any constant $\epsilon > 0$.

Then, we present a much more involved solution for the fully dynamic case. It relies on exploiting the local consistency of the parsing that the data structure of Gawrychowski et al. [82] underlying Theorem 1.3.1 maintains for the strings in the collection, and on maintaining two dynamic trees with labelled bicoloured leaves, so that after each update

we can report a pair of nodes, one from each tree, of maximum combined weight, which have at least one common leaf-descendant of each colour.

Theorem 1.5.3. *We can maintain an LCF of two initially empty dynamic strings, each of length at most n, in $\mathcal{O}(\log^8 n)$ amortised time w.h.p. per edit operation.*

After having determined that the complexity of fully dynamic LCF is polylogarithmic, the next natural question is whether we can further improve the bound to polyloglogarithmic. By now, we have techniques that can be used to not only distinguish between these two situations but (in some cases) also provide tight bounds. As a prime example, static predecessor for a set of n numbers from $[n^2]$ requires $\Omega(\log\log n)$ time for structures of size $\tilde{\mathcal{O}}(n)$ [142], and dynamic connectivity for forests requires $\Omega(\log n)$ time [141], with both bounds being asymptotically tight. In some cases, seemingly similar problems might have different complexities, as in the orthogonal range emptiness problem: Nekrich [138] showed a data structure of size $\mathcal{O}(n\log^4 n)$ with $\mathcal{O}(\log^2\log n)$ query time for three dimensions, while for the same problem in four dimensions Pătraşcu showed that any polynomial-size data structure requires $\Omega(\log n/\log\log n)$ query time [143].

In [42], the following lower bounds were shown.

- Any data structure of $\tilde{\mathcal{O}}(n)$ size for maintaining an LCF of a dynamic string S and a static string T, each of length at most n, requires $\Omega(\log n/\log\log n)$ time per update operation.

- Any polynomial-size data structure for maintaining an LCF of two dynamic strings, each of length at most n, requires $\Omega(\log n/\log\log n)$ time per update operation.

These lower bounds hold even when both amortisation and Las Vegas randomisation are allowed. The first step in both of these lower bounds is a reduction from the problem of answering reachability queries in butterfly graphs that was considered in the seminal paper of Pătraşcu [143] to the HIA problem. In particular, in order to show that these lower bounds hold even when Las Vegas randomisation is allowed, a generalisation of Pătraşcu's reduction from the information-theoretic lopsided set disjointness problem to the butterfly reachability problem was necessary; see [42].

As can be seen by Theorem 1.5.2, the difference in the allowed space in the above two lower bounds is indeed needed, as partially dynamic LCF admits an $\mathcal{O}(n^{1+\epsilon})$-space, $\mathcal{O}(\log \log n)$-update-time solution, for any constant $\epsilon > 0$.

1.6 Dynamic String Alignment

The problems of computing an optimal string alignment, a longest common subsequence (LCS), or the edit distance of two strings have been studied for more than 50 years [168, 137]. In the string alignment problem, we are given weights w_{match} for aligning a pair of matching letters, w_{mis} for aligning a pair of mismatching letters, and w_{gap} for letters that are not aligned, and the goal is to compute an alignment with maximum weight. The edit distance $d_E(S, T)$ of two strings S and T is the minimum cost of transforming string S to string T using insertions, deletions, and substitutions of letters, under specified costs c_{ins}, c_{del}, and c_{sub}, respectively. When all costs are 1, this is also known as the Levenshtein distance of S and T [124]. Note that if $c_{ins} = c_{del}$, the edit distance problem is a special case of the string alignment problem, with $w_{match} = 0$, $w_{mis} = -c_{sub}$, and $w_{gap} = -c_{ins} = -c_{del}$. In turn, the LCS problem can be seen as a special case of the edit distance problem: Let the length of an LCS of S and T be denoted by $\texttt{LCS}(S, T)$. Then, for $c_{ins} = c_{del} = 1$ and $c_{sub} = 2$, we have $d_E(S, T) = |S| + |T| - 2 \cdot \texttt{LCS}(S, T)$.

The textbook dynamic programming (DP) $\mathcal{O}(n^2)$-time algorithm for the (static) LCS and edit distance problems has been rediscovered several times, e.g. in [168, 137, 152, 153, 169]. When the desired output is just the edit distance or the length of an LCS, the space required by the DP algorithm is trivially $\mathcal{O}(n)$ as one needs to store just two rows or columns of the DP matrix. Hirschberg showed how to actually retrieve an LCS within $\mathcal{O}(n^2)$ time using only $\mathcal{O}(n)$ space [96]. A line of works has improved the complexity of the classic DP algorithm by factors polylogarithmic with respect to n (see [128, 173, 60, 32, 89]).

On the lower-bound side, Backurs and Indyk showed that an $\mathcal{O}(n^{2-\epsilon})$-time algorithm for computing the edit distance of two strings of length at most n would refute the Strong

Exponential Time Hypothesis (SETH) [24]. Bringmann and Künnemann generalised this conditional lower bound by showing that it holds even for binary strings under any non-trivial assignment of weights c_{ins}, c_{del}, and c_{sub} [36]—an assignment of weights is trivial if it allows one to infer the edit distance in constant time. Further consequences of subquadratic-time algorithms for the edit distance or LCS problems where shown by Abboud et al. [1]; interestingly, they proved that even shaving arbitrarily large polylogarithmic factors from n^2 would have major consequences.

Note that the aforementioned DP algorithm is inherently "dynamic" in the sense that it supports appending a letter and deleting the last letter in either of the strings in linear time. A series of works examined variants of incremental and decremental LCS and edit distance problems [123, 109, 99]. In particular, Tiskin presented a linear-time algorithm for maintaining an LCS in the case where both strings are subject to the following updates: prepending or appending a letter, and deleting the first or the last letter [161]. Tiskin's solution not only maintains the LCS, but implicitly also the *semi-local LCS information*: the LCS lengths between all prefixes of S (resp. T) and all suffixes of T (resp. S), as well as the LCS between S (resp. T) and all fragments (substrings) of T (resp. S). Semi-local LCS is a restricted variant of internal LCS [151], which we briefly discuss in Section 7.1.

One of the main technical contributions of Tiskin in this area is an efficient algorithm for computing the $(\min, +)$*-product* (also known as *distance product*) of two simple unit-Monge matrices [166].[2] The algorithm itself and the ideas behind it have found numerous applications to variants of the LCS and string alignment problems. We refer the reader to Tiskin's monograph [161] as well as to [162, 163, 164, 160, 165].

Here, we consider the dynamic version of the string alignment problem, in which the strings S and T, each of length at most n, undergo insertions, deletions, and substitutions of letters, and we are to report an optimal alignment after each such update. Hyyrö et

[2]A matrix M is a Monge matrix if $M[i,j] + M[i',j'] \leq M[i',j] + M[i,j']$ for all $i < i'$ and $j < j'$ [131]. An $n \times n$ Monge matrix is a simple unit-Monge matrix if its leftmost column and bottommost row consist of zeroes, while its rightmost column and topmost row consist of subsequent integers from 0 to $n-1$ [166].

al. [97] presented a practical algorithm for maintaining the edit distance of two dynamic strings; however the worst-case bound on the update time of their algorithm is $\mathcal{O}(n^2)$, which is not better than recomputing the edit distance from scratch. We are the first to obtain non-trivial worst-case update time for the dynamic string alignment problem. Note that, in light of the conditional lower bounds stated above, an $\mathcal{O}(n^{1-\epsilon})$-time algorithm maintaining an optimal string alignment of two strings of length $\mathcal{O}(n)$ subject to edit operations seems highly unlikely, as it would directly imply an $\mathcal{O}(n^{2-\epsilon})$-time algorithm for the static version of the problem. We show the following result, in particular matching this lower bound—up to subpolynomial factors—for the case where the alignment weights are small positive integers.

Theorem 1.6.1. *Given two strings S and T, of length at most n, and integer alignment weights w_{match}, w_{mis}, and w_{gap}, bounded by $n^{\mathcal{O}(1)}$, the optimal alignment of S and T as they undergo insertions, deletions, and substitutions of letters can be maintained in $n \cdot \min\{\sqrt{n}, w\} \cdot \log^{\mathcal{O}(1)} n$ time per operation after an $\tilde{\mathcal{O}}(n^2)$-time preprocessing.*

For the $\tilde{\mathcal{O}}(nw)$-time algorithm, we heavily rely on Tiskin's work on semi-local LCS, and in particular, in an implicit way, on his algorithm for computing the $(\min, +)$-product of two simple unit-Monge matrices [161, 166]. For the $\tilde{\mathcal{O}}(n\sqrt{n})$-time algorithm, we employ efficient data structures for computing distances in planar graphs. Let us note that the data structure underlying Theorem 1.6.1 can also efficiently answer internal LCS queries.

There has been a recent series of breakthrough papers on approximating the edit distance and length of the LCS; see e.g. [18, 16, 37, 93, 86, 34, 17]. It is natural to ask about the maintenance of an approximation of the edit distance or LCS in the setting of dynamic strings. We leave this problem open for further investigation.

Chapter 2

Preliminaries

2.1 Model of Computation

We work in the standard word RAM model of computation with word-size $\Theta(\log n)$, where n is the total size of the input. We measure the space requirements of our algorithms and data structures in words.

Some of our algorithms are Las Vegas randomised, that is, they always return correct answers but their running times may only hold *with high probability* (*w.h.p.*), i.e. probability $1 - 1/n^{\Omega(1)}$, where n is the total size of the input (including updates).

2.2 Strings

Let $S = S[1]S[2]\cdots S[n]$ be a *string* of length $n := |S|$ over an alphabet Σ of size σ. The elements of Σ are called *letters*. The set of all strings over Σ is denoted by Σ^*. Throughout, we make the standard assumption that Σ consists of non-negative integers polynomially bounded in the size of the input.

By ε we denote an *empty string*. For two positions i and j of S, we denote by $S[i\,..\,j]$ the *fragment* of S that starts at position i and ends at position j (the fragment is empty if $i > j$). A fragment of S is represented in $\mathcal{O}(1)$ space by specifying the indices i and j. The fragment $S[i\,..\,j]$ is an *occurrence* of the underlying *substring* $P = S[i]\cdots S[j]$. We

then say that P occurs at *(starting) position* i in S. A *prefix* of S is a fragment that starts at position 1 (i.e. of the form $S[1\,..\,j]$) and a *suffix* is a fragment that ends at position n (i.e. of the form $S[i\,..\,n]$).

We denote the *reverse string* of S by S^R, i.e. $S^R = S[n]S[n-1]\cdots S[1]$. By UV we denote the concatenation of two strings U and V, and by U^k the concatenation of k copies of U. A string of the form U^2 is called a *square*. A string S is called *primitive* if it cannot be expressed as U^k for a string U and an integer $k > 1$. A *cyclic rotation* of a string U is any string V such that $U = XY$ and $V = YX$ for some strings X and Y.

2.3 Periodicity

Periodicity is one of the main and most elegant notions in stringology. A positive integer p is called a *period* of a string S of length n if $S[i] = S[i+p]$ for all $i \in [1, n-p]$. We refer to the smallest period as *the period* of the string, and denote it by $\mathsf{per}(S)$. A string is called *periodic* if its period is no more than half of its length and *aperiodic* otherwise. Let us state the weak version of the periodicity lemma, a classic combinatorial result on strings.

Lemma 2.3.1 (Periodicity Lemma (weak version), [70])**.** *If p and q are periods of a string S and satisfy $p + q \le |S|$, then $\gcd(p, q)$ is also a period of S.*

The following lemma, which characterises the occurrences of a pattern in a text in terms of the pattern's period, is a prime application of the periodicity lemma.

Lemma 2.3.2 ([35])**.** *Let P be a pattern and T be a text with $|T| \le 2|P|$. The starting positions of occurrences of P in T form an arithmetic progression with difference $\mathsf{per}(P)$.*

Runs. A *run* is a periodic fragment $R = S[a\,..\,b]$ which can be extended neither to the left nor to the right without increasing its period $p = \mathsf{per}(R)$, i.e. $S[a-1] \neq S[a+p-1]$ and $S[b-p+1] \neq S[b+1]$, provided that the respective positions exist. The *exponent* $\mathsf{exp}(R)$ of a run R with period p is $|R|/p$. A string of length n has at most n runs [26] and they can be computed in $\mathcal{O}(n)$ time [119].

The *Lyndon root* of a periodic string S is the lexicographically smallest rotation of its $\mathsf{per}(S)$-length prefix. If L is the Lyndon root of a periodic string S, then S may be represented as (L, r, a, b); here $S = L[|L| - a + 1 \mathinner{.\,.} |L|]L^r L[1 \mathinner{.\,.} b]$, and r is called the *rank* of U. For a periodic fragment U, we write $\mathsf{run}(U)$ for the unique run with period $\mathsf{per}(U)$ that contains U [26, 59]. We say that $\mathsf{run}(U)$ *extends* U.

2.4 Tries and Trees

We define the *trie* of a collection of strings $\mathcal{X} = \{S_1, S_2, \dots, S_k\}$ as follows. It is a rooted tree with edges labeled by single letters. Every string S that is a prefix of some string in $\mathcal{X}$ is represented by exactly one path from the root to some node u of the tree, such that the concatenation of the labels of the edges of the path, the *path-label* of u, is equal to S. The nodes with path-label equal to some $S_i \in \mathcal{X}$ are called terminal. Moreover, the path-label of each node of the trie is equal to a prefix of some $S_i \in \mathcal{X}$.

The *compact trie* of $\mathcal{X}$ is obtained by making all non-terminal nodes of the trie of $\mathcal{X}$ that have exactly one child implicit. For a node u in a (compact) trie, we define its *depth* as the number of edges on the path from the root to u. Analogously, we define the *string-depth* of u as the total length of the labels along the path from the root to u. We denote by $\mathcal{L}(u)$ the *path-label* of a node u, i.e. the path-ordered concatenation of the edge labels along the path from the root to u. The label of each edge is stored in $\mathcal{O}(1)$ space as a fragment of some S_i. In order to access the child of an explicit node by the first letter of its edge label in $\mathcal{O}(1)$ time, perfect hashing [73] can be used.

The *suffix tree* $\mathcal{T}(S)$ of a non-empty string S of length n is the compact trie of all suffixes of $S\$$, where $\$ \notin \Sigma$ is a sentinel letter smaller than all letters of the alphabet Σ. This sentinel letter ensures that all terminal nodes are leaves. $\mathcal{T}(S)$ can be constructed in $\mathcal{O}(n)$ time for polynomially bounded integer alphabets [67]. Each fragment of S is uniquely represented by either an explicit or an implicit (along an edge) node of $\mathcal{T}(S)$, called its *locus*.

We say that a rooted tree is *weighted* if there is an integer weight $w(v)$ associated with

each node v of the tree, such that the weight of the root is zero, and weights along each root-to-leaf path are increasing, that is, for any node u with parent v, $w(u) > w(v)$. We say that a node v is a *weighted ancestor at depth ℓ* of a node u if v is the lowest ancestor of u with weight at least ℓ. This problem was introduced in [66] by Farach-Colton and Muthukrishnan. Their data structure was derandomised by Amir et al. in [14].

Theorem 2.4.1 ([14]). *Weighted ancestor queries for nodes of a weighted tree $\mathcal{T}$ of size n, with weights of size $n^{\mathcal{O}(1)}$, can be answered in $\mathcal{O}(\log\log n)$ time after an $\mathcal{O}(n)$-time preprocessing.*

A suffix tree $\mathcal{T}(S)$ is a weighted tree with $w(v)$ equal to the string-depth of v (which is $\mathcal{O}(n)$), for every node v. Hence, the locus of a fragment $S[i\mathinner{.\,.} j]$ in $\mathcal{T}(S)$ can be computed via computing the weighted ancestor of the terminal node with path-label $S[i\mathinner{.\,.} n]\$$ at string-depth $j-i+1$.

If the weight-function w has the property that the difference of the weights of a child and its parent is always equal to 1, then the value $w(v)$ is called the *level* of v, and the respective queries are called *level ancestor queries.*

Theorem 2.4.2 ([29, 31]). *Level ancestor queries for nodes of a tree $\mathcal{T}$ of size n can be answered in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing.*

Note that given a node v in the suffix tree of a string S, we can compute v's child that is on the path to a given leaf-descendant u of v using a single level ancestor query in $\mathcal{O}(1)$ time after an $\mathcal{O}(|S|)$-time preprocessing.

2.5 Data Structures for Range Queries

In the Range Minimum Query (RMQ) problem, we are given an array A of n numbers and we are asked to answer queries of the following type: for indices i and j, return the index of a minimum element in the subarray $A[i\mathinner{.\,.} j]$. We denote such a query by $\mathsf{RMQ}(i,j)$. The RMQ problem and the linearly equivalent *lowest common ancestor* (LCA) problem on trees are very well-studied and several optimal solutions exist [94, 28, 71].

Theorem 2.5.1 ([94, 28]). *An array of size n can be preprocessed in $\mathcal{O}(n)$ time, so that* RMQ *queries can then be answered in $\mathcal{O}(1)$ time.*

The range maximum queries problem admits the same solution.

The predecessor problem, consists in maintaining a set Y of integers, over an ordered universe U, so that, for any queried integer $x \in U$ one can efficiently return the predecessor $\max\{y \in Y : y \leq x\}$ of x in Y. The successor problem is defined analogously: upon a queried integer $x \in U$, the successor $\min\{y \in Y : y \geq x\}$ of x in Y is to be returned. The following theorem encapsulates the *van Emde Boas tree* data structure [167].

Theorem 2.5.2 ([167]). *We can maintain a subset Y of $[1, U]$, supporting insertions and deletions to Y, as well as predecessor and successor queries in $\mathcal{O}(\log \log |U|)$ time, using $\mathcal{O}(|U|)$ space.*

For a *static* set Y, by combining y-fast tries [171] and deterministic dictionaries [147] using a two-level approach, we can get a deterministic $\mathcal{O}(|Y|)$-space data structure with the same query time complexity; see [159].

Orthogonal range searching. Let $\mathcal{P}$ be a collection of n points in 2D (rank space) with integer weights. Throughout the thesis, we will ensure that the cost of mapping the points to rank space (i.e. maintaining them sorted in both dimensions) is always accounted for. Let the universe be $[1, m] \times [1, m]$, for m polynomially bounded in 2^w. It suffices to use a (deterministic) predecessor structure, with $\mathcal{O}(\log \log m)$ query time in the static case and $\mathcal{O}(\log m)$ time per update in the dynamic case; these complexities are dominated by the ones of the data structures that we employ.

We consider the following queries for an axes-parallel rectangle $R = [a, b] \times [c, d]$.

- Orthogonal range maximum: report the maximum weight of a point from $\mathcal{P}$ in R.
- Orthogonal range reporting: report all points from $\mathcal{P}$ in R.
- Orthogonal range counting: count all points from $\mathcal{P}$ in R.

The following statement formalises the 2D range tree data structure that was devised by Bentley [30]. (For each of the problems encapsulated in the following theorem there are several tradeoffs, but we choose to mostly use the standard 2D range trees in favour of uniformity.)

Theorem 2.5.3 ([30]). *An $\mathcal{O}(n \log n)$-size data structure over a collection at most n integer points with integer weights in 2D (rank space) that answers orthogonal range maximum/reporting/counting queries in time $\mathcal{O}(\log^2 n + |output|)$ can be constructed in time $\mathcal{O}(n \log n)$ and maintained in time $\mathcal{O}(\log^2 n)$ upon each insertion/deletion of a point to/from the collection.*

In particular, for the range counting problem, which we will encounter several times, we also use the following result by Chan and Pătraşcu [39].

Theorem 2.5.4 ([39]). *Orthogonal range counting queries for n integer points in 2D (rank space) can be answered in time $\mathcal{O}(\log n / \log \log n)$ with a data structure of size $\mathcal{O}(n)$ that can be constructed in time $\mathcal{O}(n\sqrt{\log n})$.*

2.6 Internal Queries in Strings

We write $\mathsf{LCP}(S, T)$ to denote the length of the *longest common prefix* of two strings S and T. When some reference string S of length n is clear from the context, we might write $\mathsf{LCP}(i, j)$ instead of $\mathsf{LCP}(S[i\mathinner{.\,.} n], S[j\mathinner{.\,.} n])$. The following result can be obtained by constructing a lowest common ancestor data structure over the suffix tree of string S.

Theorem 2.6.1 ([67]). *A string S of length n can be preprocessed in $\mathcal{O}(n)$ time, so that LCP queries concerning the suffixes of S can be then answered in $\mathcal{O}(1)$ time.*

A symmetric construction on S^R (the reverse of S) can answer so-called *longest common suffix* queries (denoted by $\mathsf{LCP}^{\mathrm{R}}$) for prefixes of S in the same complexity. The LCP and $\mathsf{LCP}^{\mathrm{R}}$ queries are also collectively known as *longest common extension* (LCE) queries.

In an internal pattern matching query for a string S, we are given two fragments P and T of S, and are asked to return the occurrences of P in T. We denote such a query

by $\mathsf{IPM}(P, T)$. The decision (deciding whether P occurs in T) and counting (counting the occurrences of P in T) versions of internal pattern matching queries are also of interest.

One can construct a data structure answering any of these three variants of internal pattern matching using the following relation to orthogonal range queries, which (to the best of our knowledge) first appeared in the conference version of [127]. Let us construct the suffix tree $\mathcal{T}(S)$ and preprocess it so that each node stores the lexicographic range of suffixes of which its path-label is a prefix. We also construct a 2D orthogonal range counting data structure over an $n \times n$ grid $\mathcal{G}$, in which, for each $S[a \,..\, n]$, we insert a point (a, b), where b is the lexicographic rank of this suffix among all suffixes. We then answer a query for P and $T = S[i \,..\, j]$ as follows. We first locate the locus of P in $\mathcal{T}(S)$ using a weighted ancestor query in $\mathcal{O}(\log \log n)$ time using Theorem 2.4.1, and retrieve the associated lexicographic range $[l, r]$. Next, we perform a range counting or reporting query (Theorem 2.5.3) for the range $[i, j - |P| + 1] \times [l, r]$ of $\mathcal{G}$, depending on the variant we are considering. This yields the following result. (See [127, 106] for more efficient solutions for variants of the IPM problem.)

Proposition 2.6.2 (Based on [127])**.** *Given a string of length n, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$-size data structure that answers the decision, counting, and reporting versions of internal pattern matching queries in time $\mathcal{O}(\log^2 n + |output|)$.*

Remark 2.6.3. Actually, in [127] there is an extra $|P|$ additive factor in the query time complexity as the authors consider patterns given explicitly; i.e. this factor corresponds to computing the locus of the pattern in $\mathcal{T}(S)$ using a forward search from the root.

A *2-period query* decides whether a given fragment of the string in scope is periodic and, if so, it also returns its period.

Theorem 2.6.4 ([115, 26, 112])**.** *Given a string of length n, we can construct in $\mathcal{O}(n)$ time an $\mathcal{O}(n)$-size data structure that answers 2-period queries in $\mathcal{O}(1)$ time.*

Finally, we make use of the following internal query for computing the run extending a given periodic fragment.

Theorem 2.6.5 ([112])**.** *A string of length n can be preprocessed in $\mathcal{O}(n)$ time, so that, given a periodic fragment U, $\mathsf{run}(U)$ and its Lyndon root can be computed in $\mathcal{O}(1)$ time.*

2.7 Straight-Line Programs and Recompression

We denote the set of non-terminals of a context-free grammar $\mathcal{G}$ by $N_{\mathcal{G}}$ and call the elements of $\mathcal{S}_G := N_{\mathcal{G}} \cup \Sigma$ symbols. A *straight-line program* (*SLP*) $\mathcal{G}$ is a context-free grammar that consists of a set $N_{\mathcal{G}} = \{A_1, \ldots, A_n\}$ of non-terminals, such that each $A_i \in N_{\mathcal{G}}$ is associated with a unique production rule $A_i \to f_{\mathcal{G}}(A) \in (\Sigma \cup \{A_j : j < i\})^*$.

Every symbol $A \in \mathcal{S}_{\mathcal{G}}$ generates a unique string in Σ^*, which we denote by $\mathsf{gen}(A)$.

The string $\mathsf{gen}(A)$ can be obtained from A by repeatedly replacing each non-terminal by its production. (We also use $\mathsf{gen}(\cdot)$ for sequences of symbols, to denote the concatenation of the strings generated by those symbols.) In addition, A is associated with its *parse tree* $\mathsf{PT}(A)$ (also denoted by $\mathsf{PT}[\mathsf{gen}(A)]$) consisting of a root labeled with A to which zero or more subtrees are attached:

- If A is a terminal, there are no subtrees.
- If A is a non-terminal $A \to B_1 \cdots B_p$, then $\mathsf{PT}(B_i)$ are attached in increasing order of i.

Note that if we traverse the leaves of $\mathsf{PT}(A)$ from left to right, spelling out the corresponding non-terminals, then we obtain $\mathsf{gen}(A)$.

The parse tree PT of $\mathcal{G}$ is the parse tree of the (distinguished) starting symbol $A_n \in N_{\mathcal{G}}$, for which $\mathsf{gen}(A_n) = S$, where S is the unique string generated by $\mathcal{G}$. We write $\mathsf{gen}(\mathcal{G}) := S$. Consult Figure 2.1 for an example of an SLP and its parse tree.

We define the *value* $\mathsf{val}(v)$ of a node v in PT to be the fragment $S[a \mathinner{.\,.} b]$ corresponding to the leaves $S[a], \ldots, S[b]$ in the subtree of v. Note that $\mathsf{val}(v)$ is an occurrence of $\mathsf{gen}(A)$ in $\mathsf{gen}(\mathcal{G})$, where A is the label of v. We define a *layer* to be any sequence $v_1, v_2, \ldots, v_r$ of nodes in PT whose values are consecutive fragments of S, i.e. $\mathsf{val}(v_j) = S[r_{j-1}+1 \mathinner{.\,.} r_j]$ for some increasing sequence of r_i's. The value of a layer C is the concatenation of the values of its elements and is denoted by $\mathsf{val}(C)$.

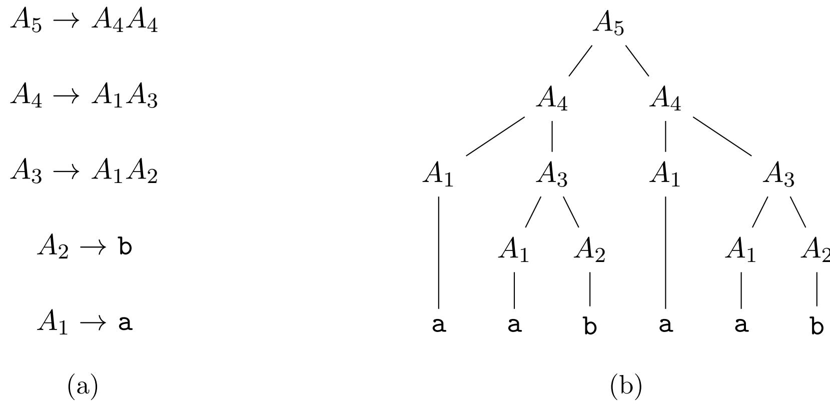


Figure 2.1: (a) An SLP $\mathcal{G}$ generating aabaab. (b) The corresponding parse tree PT.

A *run-length straight-line program* (*RLSLP*) is a straight-line program $\mathcal{G}$ that contains two kinds of non-terminals:

- *Concatenations*: Non-terminals with production rules of the form $A \to BC$ (for symbols B and C).
- *Powers*: Non-terminals with production rules of the form $A \to B^p$ (for a symbol B and an integer $p \geq 2$).

The key idea of the *recompression* technique by Jeż [102, 103] is the construction of a particular RLSLP $\mathcal{H}$ that generates the input string S. Let n be the length of S. Then, the underlying parse tree PT is of depth $\mathcal{O}(\log n)$ and it can be constructed in $\mathcal{O}(n)$ time. In particular, the name of the technique stems from the fact that an SLP $\mathcal{G}$ can be efficiently recompressed to the RLSLP in-place, that is, without first uncompressing $\mathcal{G}$.

The *run-length encoding* (RLE) representation of a string V is a sequence $V_1, \ldots, V_k$ of substrings, such that $V = V_1 \cdots V_k$, and, for all i: $V_i = a_i^{p_i}$ for some $a_i \in \Sigma$, and $a_i \neq a_{i+1}$. We call each fragment of S that corresponds to a block in the RLE representation of S an *RLE run*. (This definition naturally extends to sequences.)

As observed by I [98], the parse tree PT of $\mathcal{H}$ is *locally consistent* in a certain sense. To formalise this property, he introduced the *popped sequence* of every fragment

$S[a\mathinner{.\,.} b]$, which is a sequence of symbols labeling a certain layer of nodes whose values constitute $S[a\mathinner{.\,.} b]$.

Theorem 2.7.1 ([98])**.** *If two fragments of a string of length n are equal, then their popped sequences are equal as well. Moreover, each popped sequence consists of $\mathcal{O}(\log n)$ RLE runs (maximal powers of a single symbol) and can be constructed in $\mathcal{O}(\log n)$ time. The nodes corresponding to symbols in a run share a single parent. Furthermore, the popped sequence consists of a single symbol only for fragments of length 1.*

Let $F_1^{p_1} \cdots F_t^{p_t}$ be the run-length encoding of the popped sequence of a substring U of S. We define

$$L(U) = \{|\mathsf{gen}(F_1)|, |\mathsf{gen}(F_1^{p_1} \cdots F_{t-1}^{p_{t-1}} F_t^{p_t-1})|\} \bigcup \{|\mathsf{gen}(F_1^{p_1} \cdots F_i^{p_i})| : i \in [1, t-1]\}.$$

By Theorem 2.7.1, the set $L(U)$ can be constructed in $\mathcal{O}(\log n)$ time given an occurrence $S[a\mathinner{.\,.} b] = U$.

Lemma 2.7.2. *Let v denote a non-leaf node of a parse tree $\mathsf{PT}[S]$ stemming from recompression and let $S[a\mathinner{.\,.} b]$ denote an occurrence of a string U contained in $\mathsf{val}(v)$, but not contained in $\mathsf{val}(u)$ for any child u of v. If $S[a\mathinner{.\,.} c]$ is the longest prefix of $S[a\mathinner{.\,.} b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\mathinner{.\,.} c]| \in L(U)$. Symmetrically, if $S[c'+1\mathinner{.\,.} b]$ is the longest suffix of $S[a\mathinner{.\,.} b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\mathinner{.\,.} c']| \in L(U)$.*

Proof. Consider the popped sequence $v_1, \ldots, v_p$ of $S[a\mathinner{.\,.} b]$. Each of these nodes is a descendant of a child of v. Note that $S[a\mathinner{.\,.} c] = \mathsf{val}(v_1) \cdots \mathsf{val}(v_q)$, where $v_1, \ldots, v_q$ is the longest prefix of $v_1, \ldots, v_p$ consisting of descendants of the same child of v.

If the labels of v_q and v_{q+1} are distinct, then they belong to distinct runs of symbols and $|S[a\mathinner{.\,.} c]| \in L(U)$. (See Figure 2.2 for an illustration of this case.)

Otherwise, v_q and v_{q+1} share the same parent. As they are descendants of different children of v, their parent must be v. Due to this, and the fact that $\mathsf{val}(v_q)$ is a prefix of $S[a\mathinner{.\,.} b]$, we have $q = 1$. Hence, $|S[a\mathinner{.\,.} c]| = |\mathsf{val}(v_1)| \in L(U)$.

The proof of the second claim is symmetric. $\square$

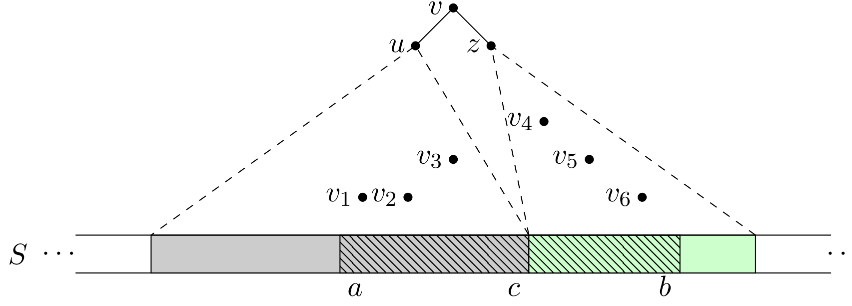


Figure 2.2: Node v has two children, u and z. We denote $val(u)$ by a grey rectangle, and $\mathsf{val}(z)$ by a green rectangle. $S[a\mathinner{.\,.} c]$ is the longest prefix of $S[a\mathinner{.\,.} b]$ that is contained in $\mathsf{val}(u)$. We have $q = 3$. Note that the labels of v_3 and v_4 must be distinct as they do not share a single parent. Hence, $|S[a\mathinner{.\,.} c]| \in L(U)$.

2.8 Dynamic Strings

Lastly, we consider the dynamic setting. In particular, we consider the dynamic maintenance of a collection of non-empty persistent strings $\mathcal{X}$ that is initially empty and undergoes updates specified by the following operations:

- $\mathsf{makestring}(U)$: Insert a non-empty string U to $\mathcal{X}$.
- $\mathsf{concat}(U, V)$: Insert UV to $\mathcal{X}$, for $U, V \in \mathcal{X}$.
- $\mathsf{split}(U, i)$: Insert $U[1\mathinner{.\,.} i-1]$ and $U[i\mathinner{.\,.} |U|]$ in $\mathcal{X}$, for $U \in \mathcal{X}$ and $i \in [2, |U|]$.

The collection is persistent, in the sense that concat and split do not destroy their arguments. Gawrychowski et al. [82] presented a data structure that efficiently maintains such a collection and allows for efficiently answering longest common prefix queries.

Theorem 1.3.1 ([82])**.** *A collection $\mathcal{X}$ of non-empty strings of total length at most N can be dynamically maintained with update operations $\mathsf{makestring}(U)$, $\mathsf{concat}(U, V)$, $\mathsf{split}(U, i)$ requiring time $\mathcal{O}(\log N + |U|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, all w.h.p, so that $\mathsf{LCP}(U, V)$ queries for $U, V \in \mathcal{X}$ can be answered in time $\mathcal{O}(1)$.*

$\mathsf{LCP}^{\mathrm{R}}$ queries can also be answered in the same time as LCP queries. LCE queries for arbitrary fragments of elements of $\mathcal{X}$ can be answered in $\mathcal{O}(\log N)$ time by first performing a constant number of split operations to add the corresponding fragments to the collection and then asking a query for them.

For each string of the collection $\mathcal{X}$, the data structure of [82] implicitly maintains an RLSLP stemming from recompression that is of depth $\mathcal{O}(\log N)$ with high probability.

Further, although the parse trees are not maintained explicitly, we have access to the following pointers. First of all, a pointer to the parse tree of each $S \in \mathcal{X}$. Then, given a pointer to some node v in the parse tree of some $S \in \mathcal{X}$, we can retrieve in $\mathcal{O}(1)$ time the endpoints a, b of the fragment $\mathsf{val}(v) = S[a\,..\,b]$, the degree of v, a pointer to the parent of v, and a pointer to the j-th child of v, provided that such a child exists.

Crucially, the RLSLPs of all $S \in \mathcal{X}$ maintained by the data structure underlying Theorem 1.3.1 are locally consistent with each other, that is, Theorem 2.7.1 and Lemma 2.7.2 are also true for fragments of different strings $S_1, S_2 \in \mathcal{X}$, with each $\log n$ factor in Theorem 2.7.1 replaced by a $\log N$ factor (with high probability)—these factors stem from the depth of the underlying parse trees.

We next discuss how to maintain a dynamic data structure answering LCE and IPM queries for a single dynamic string S of length at most n that undergoes edit operations (i.e. insertions, deletions, and substitutions of letters), employing Theorems 1.3.1 and 1.3.2. First, note that each edit operation can be simulated with a constant number of concat, split, and $\mathsf{makestring}(a)$ operations, for $a \in \Sigma$.

Let us now prove a general lemma, which we will use in order to keep the space occupied by our data structure low.

Lemma 2.8.1. *Suppose that we have a dynamic data structure $\mathcal{A}$ for a problem of size (at all times) at most n, that can be built in $p(n)$ time, can process each update, specified in $\mathcal{O}(1)$ time, in $u(n)$ time, and can answer each query in $q(n)$ time (possibly with the restriction that no more than r updates have been performed since the initialisation of the data structure).*

Then, for any $m \le \min\{r, n\}$, we can design a dynamic data structure $\mathcal{B}$ for the same

problem that can be built in $p(n)$ time, occupies space $\mathcal{O}(p(n) + m \cdot u(n))$, can process each update in $\mathcal{O}(u(n) + p(n)/m)$ time, and can answer each query in $\mathcal{O}(q(n))$ time.

If either of $p(n)$ or $u(n)$ holds w.h.p., then the update time of $\mathcal{B}$ holds with probability $1 - n^{-\Omega(1)}$. If $q(n)$ holds w.h.p., then the query time of $\mathcal{B}$ holds with probability $1 - n^{-\Omega(1)}$.

Proof. We rebuild $\mathcal{A}$ after every m updates. The space occupied by it is upper bounded by the sum of the preprocessing time and the total time to process updates. (We can revert any changes made to the data structure by a query just after answering the query within $\mathcal{O}(q(n))$ time.) Each query clearly then requires time $q(n)$. Each update requires $u(n)$ time, and also gets charged $\mathcal{O}(p(n)/m)$ amortised time for the rebuilding of the data structure. Let us now show how to deamortise the time required for reinitialising $\mathcal{A}$ using the so-called *time-slicing technique.*

We keep two copies of $\mathcal{A}$, switching their roles after (roughly) every $m/2$ updates. One copy is for handling at most $m/2$ updates and answering queries, while the other one is reinitialised in chunks (of either initialisation or updates replayed) in the background.

As for the last statement, it follows from the fact that we reinitialise $\mathcal{A}$ after every $m \leq n$ updates and hence, the total size of the input and updates for any instance of $\mathcal{A}$ is always $\mathcal{O}(n)$. The space is deterministic, as we can allow our data structure to fail if the preprocessing or an update takes too long, and rebuild it from scratch. □

Let us return to our problem. The initialisation of the data structure can be done with a single $\mathsf{makestring}(S)$ operation in an empty collection in $p(n) = \mathcal{O}(n)$ time. We will set $m = n/\log n$. Note that each update increases the total length of the strings in the collection by an $\mathcal{O}(n)$ additive factor, and hence their total length will be polynomial in n at any given point; hence, $N = n^{\mathcal{O}(1)}$. We thus obtain the following statement.

Corollary 2.8.2. *A dynamic string S of length at most n can be maintained in $\mathcal{O}(\log n)$ time per edit operation so that LCE queries concerning substrings of S can be answered in time $\mathcal{O}(\log n)$ and $\mathsf{IPM}(P, T)$ queries can be answered in time $\mathcal{O}(|T|/|P| \cdot \log^2 n)$, after an $\mathcal{O}(n)$-time preprocessing and using $\mathcal{O}(n)$ space. All running times hold w.h.p.*

Chapter 3

Internal Dictionary Matching

In this chapter we consider the Internal Dictionary Matching problem, in which we are given a text T and an internal dictionary $\mathcal{D}$ and are to preprocess them in order to be able to efficiently answer queries concerning occurrences of the elements of $\mathcal{D}$ in queried substrings of T. The elements of the dictionary $\mathcal{D}$ are called *patterns*. Henceforth, we assume that $\varepsilon \notin \mathcal{D}$, i.e. that the length of each $P \in \mathcal{D}$ is at least 1. We also assume that each pattern of $\mathcal{D}$ is given by the starting and ending positions of an occurrence of it in T. Thus, the size of the dictionary $d = |\mathcal{D}|$ refers to the number of patterns in $\mathcal{D}$ and not their total length.

First, in Section 3.1, we present straightforward solutions for queries $\textsc{Exists}(i, j)$ and $\textsc{Report}(i, j)$. In Section 3.2, we describe an involved solution for $\textsc{ReportDistinct}(i, j)$ queries, that heavily relies on the periodic structure of the input text and on coloured range reporting. In Section 3.3, we rely on locally consistent parsing and orthogonal range queries to obtain an efficient solution for $\textsc{Count}(i, j)$ queries. In Section 3.4 we present solutions for answering $\textsc{CountDistinct}(i, j)$ queries. These solutions exploit string periodicity (captured by runs), and use data structures for variants of the (coloured) orthogonal range counting problem, and for some auxiliary internal queries on strings. Finally, in Section 3.5 we extend our solutions for the case of a dynamic dictionary and provide a lower bound conditional on the OMv conjecture.

3.1 Exists(i, j) and Report(i, j) Queries

We first present a convenient modification to the suffix tree with respect to a dictionary $\mathcal{D}$; see Figure 3.1.

Definition 3.1.1. *A* $\mathcal{D}$-modified suffix tree *of a string T is a compact trie with the path-labels of the terminal nodes being in one-to-one correspondence with the non-empty suffixes of $T\$$, and the path-labels of the internal nodes being in one-to-one correspondence with the elements of $\{\varepsilon\} \cup \mathcal{D}$. Each node v stores its depth and its string-depth $w(v)$.*

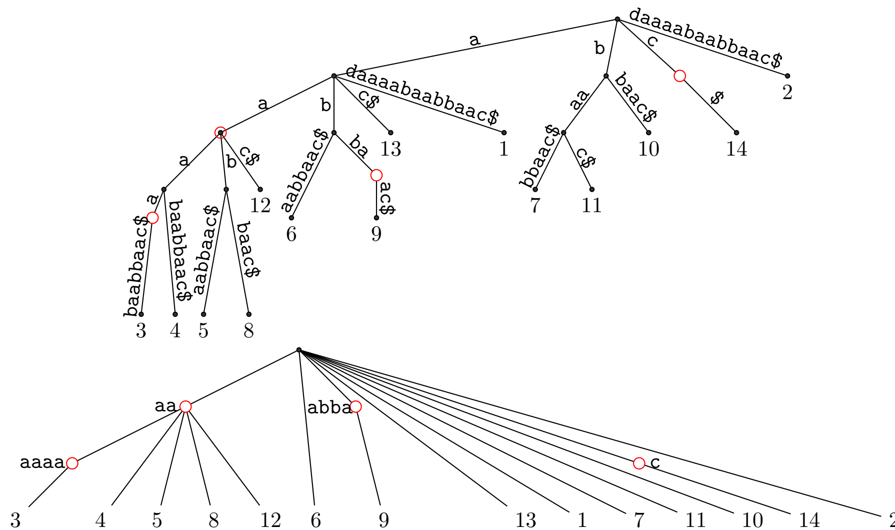


Figure 3.1: Example of a $\mathcal{D}$-modified suffix tree for dictionary $\mathcal{D} = \{\texttt{aa}, \texttt{aaaa}, \texttt{abba}, \texttt{c}\}$ and text $T = \texttt{adaaaabaabbaac}$ from Example 1.2.1. Top: The suffix tree $\mathcal{T}(T)$ with the nodes corresponding to elements of $\mathcal{D}$ annotated in red. Bottom: The $\mathcal{D}$-modified suffix tree of T.

Lemma 3.1.2. *A $\mathcal{D}$-modified suffix tree of T has size $\mathcal{O}(n + d)$ and can be constructed in $\mathcal{O}(n + d)$ time.*

Proof. The $\mathcal{D}$-modified suffix tree is obtained from the suffix tree $\mathcal{T}(T)$ in two steps.

In the first step, we mark all nodes of $\mathcal{T}(T)$ with path-label equal to a pattern $P \in \mathcal{D}$: if any of them are implicit, we first make them explicit; see the annotated nodes

in Figure 3.1 (top). We can find the loci of the patterns in $\mathcal{T}(T)$ in $\mathcal{O}(n+d)$ time by answering the weighted ancestor queries as a batch [113], employing a data structure for a special case of Union-Find [75]. (If many implicit nodes along an edge are to become explicit, we can avoid the local sorting based on string-depth if we sort globally in time $\mathcal{O}(n+d)$ using bucket sort and then add the new explicit nodes in decreasing order with respect to string-depth.)

In the second step, we recursively contract any edge (u,v), with u being the parent of v if:

- both u and v are unmarked, or
- u is marked and v is an unmarked internal node.

The resulting tree is the $\mathcal{D}$-modified suffix tree and has $\mathcal{O}(n)$ terminal nodes and $\mathcal{O}(d)$ internal nodes; see Figure 3.1 (bottom). □

We state the following simple lemma.

Lemma 3.1.3. *With the $\mathcal{D}$-modified suffix tree of T at hand, given positions a, j in T with $a \leq j$, we can compute all $P \in \mathcal{D}$ that occur at position a and are of length at most $j-a+1$ in time $\mathcal{O}(1+|output|)$.*

Proof. We start from the root of the $\mathcal{D}$-modified suffix tree and go down towards the terminal node with path-label $T[a\,..\,n]$. We report the path-labels of all encountered nodes v as long as $w(v) \leq j-a+1$ is satisfied. We stop when this inequality is not satisfied. □

The $\mathcal{D}$-modified suffix tree enables us to answer $\textsc{Exists}(i,j)$ and $\textsc{Report}(i,j)$ queries.

Theorem 3.1.4.

(a) $\textsc{Exists}(i,j)$ *queries can be answered in $\mathcal{O}(1)$ time with a data structure of size $\mathcal{O}(n)$ that can be constructed in $\mathcal{O}(n+d)$ time.*

(b) $\textsc{Report}(i,j)$ *queries can be answered in $\mathcal{O}(1+|output|)$ time with a data structure of size $\mathcal{O}(n+d)$ that can be constructed in $\mathcal{O}(n+d)$ time.*

Proof. (a) Let us define an array $B[a] = \min\{b : T[a\,.\,.\,b] \in \mathcal{D}\}$. If there is no pattern from $\mathcal{D}$ starting in T at position a, then $B[a] = \infty$. It can be readily verified that the answer to query $\textsc{Exists}(i, j)$ is **true** if and only if the minimum element in the subarray $B[i\,.\,.\,j]$ is at most j. Thus, in order to answer $\textsc{Exists}(i, j)$ queries, it suffices to construct the array B and a data structure that answers range minimum queries (RMQ) on B. Using the $\mathcal{D}$-modified suffix tree of T, whose construction time is the bottleneck, array B can be populated in $\mathcal{O}(n)$ time as follows. For each $a \in [1, n]$, we first retrieve the terminal node v with path-label $T[a\,.\,.\,n]\$$. If v is at depth greater than 1, we set $B[a]$ to the string-depth of v's ancestor at depth 1, which can be computed using a level ancestor query in $\mathcal{O}(1)$ time. Else, we set $B[a] = \infty$. We then build the RMQ data structure of Theorem 2.5.1 for B.

(b) We first identify all positions $a \in [i, j]$ that are starting positions of occurrences of some pattern $P \in \mathcal{D}$ in $T[i\,.\,.\,j]$ using RMQs over array B, which has been defined in the proof of part (a), as follows. The first RMQ, is over the range $[i, j]$ and identifies a position a (if any such position exists). The range is then split into two parts, namely $[i, a-1]$ and $[a+1, j]$. We recursively use RMQs to identify the remaining positions in each part. Once we have found all the positions where at least one pattern from $\mathcal{D}$ occurs, we report all the patterns occurring at each of these positions and being contained in $T[i\,.\,.\,j]$. The complexities follow from Lemmas 3.1.2 and 3.1.3. □

3.2 ReportDistinct(i, j) Queries

Below, we present an algorithm that reports patterns from $\mathcal{D}$ occurring in $T[i\,.\,.\,j]$, allowing for $\mathcal{O}(1)$ copies of each pattern on the output. We can then sort these patterns, remove duplicates, and report distinct ones using an additional global array of counters, one for each pattern.

Let us first partition $\mathcal{D}$ into $\mathcal{D}_0, \ldots, \mathcal{D}_{\lfloor \log n \rfloor}$ such that $\mathcal{D}_k = \{P \in \mathcal{D} : \lfloor \log |P| \rfloor = k\}$. We call $\mathcal{D}_k$ a *k-dictionary*. We now show how to process a single k-dictionary $\mathcal{D}_k$; the query procedure may clearly assume $k \le \log |T[i\,.\,.\,j]|$.

We first build the $\mathcal{D}_k$-modified suffix tree of T. Then, we compute an array $L_k[1\,..\,n]$ such that $T[a\,..\,L_k[a]]$ is the longest pattern in $\mathcal{D}_k$ that is a prefix of $T[a\,..\,n]$. We can do this in $\mathcal{O}(n)$ time by inspecting the terminal nodes in the $\mathcal{D}_k$-modified suffix tree. Next, we assign to all the patterns of $\mathcal{D}_k$ equal to some $T[a\,..\,L_k[a]]$ integer identifiers id (or colours) in $[1\,..\,n]$, and construct an array $I_k[a] = \mathsf{id}(P)$, where $P = T[a\,..\,L_k[a]]$. We then construct the data structure specified in the following theorem for I_k; this data structure allows for efficient coloured range reporting queries, and is due to Muthukrishnan [136].

Theorem 3.2.1 ([136])**.** *Given an array $A[1\,..\,N]$ of elements from $[1, U]$, we can construct a data structure of size $\mathcal{O}(N)$ in $\mathcal{O}(N + U)$ time, so that upon query $[i, j]$ all distinct elements in $A[i\,..\,j]$ can be reported in $\mathcal{O}(1 + |output|)$ time.*

We now describe the query algorithm.

Let $t = \max\{i, j-2^{k+1}+1\}$. First, we perform a coloured range reporting query on the range $[i, t]$ of array I_k and obtain a set $\mathcal{C}_k$ of distinct patterns, employing Theorem 3.2.1. (An illustration is provided in Figure 3.2.) We observe the following.

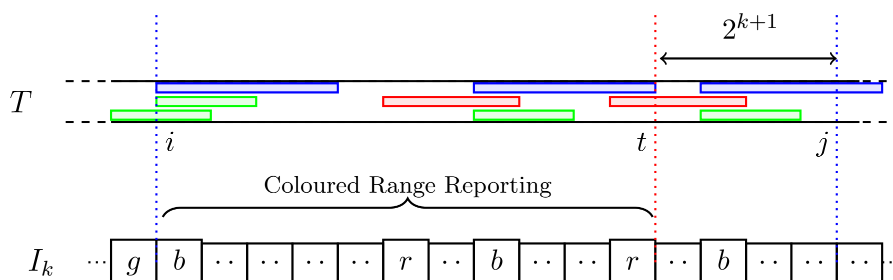


Figure 3.2: An illustration of array I_k and the coloured range reporting query that we perform. The k-dictionary consists of the blue, green and red patterns, whereas $\mathcal{C}_k$ consists of the blue and red patterns. Note that patterns that start in $[i, t]$ cannot end in a position to the right of j.

Observation 3.2.2. *Any pattern of a k-dictionary $\mathcal{D}_k$ occurring in T at position $p \in [i, t]$ is a prefix of a pattern $P \in \mathcal{C}_k$.*

Based on this observation, we will report the remaining patterns that start in $[i, t]$

using the $\mathcal{D}_k$-modified suffix tree, in $\mathcal{O}(1 + |output|)$ time, following parent pointers and temporarily marking the loci of reported patterns to avoid double-reporting. We thus now only have to report the patterns from $\mathcal{D}_k$ that occur in $T[t\,..\,j]$.

We further partition $\mathcal{D}_k$ to a *periodic k-dictionary* and an *aperiodic k-dictionary*:

$$\mathcal{D}_k^p = \{P \in \mathcal{D}_k : \mathsf{per}(P) \le 2^k/3\} \quad \text{and} \quad \mathcal{D}_k^a = \{P \in \mathcal{D}_k : \mathsf{per}(P) > 2^k/3\}.$$

Note that we can partition $\mathcal{D}_k$ in $\mathcal{O}(|\mathcal{D}_k|)$ time using 2-period queries (cf. Theorem 2.6.4).

3.2.1 Processing an Aperiodic k-Dictionary

We make use of the following sparsity property, which is analogous to Lemma 2.3.2.

Fact 3.2.3 (Sparsity of occurrences)**.** *Two occurrences of a pattern P of an aperiodic k-dictionary $\mathcal{D}_k^a$ in T start at least $\frac{1}{6}|P|$ positions apart.*

Proof. If two occurrences of P started $d \le \frac{2^k}{3}$ positions apart, then d would be a period of P, contradicting $P \in \mathcal{D}_k^a$. Then, since $2^k \le |P| < 2^{k+1}$, we have that $2^k/3 \ge \frac{1}{6}|P|$. □

Lemma 3.2.4. ReportDistinct(t, j) *queries for the aperiodic k-dictionary $\mathcal{D}_k^a$ and $j - t \le 2^{k+1}$ can be answered in $\mathcal{O}(1 + |output|)$ time with a data structure of size $\mathcal{O}(n + |\mathcal{D}_k^a|)$, that can be constructed in $\mathcal{O}(n + |\mathcal{D}_k^a|)$ time.*

Proof. Since the fragment $T[t\,..\,j]$ is of length at most 2^{k+1}, it may only contain a constant number of occurrences of each pattern in $\mathcal{D}_k^a$ by Fact 3.2.3. We can thus simply use a Report(t, j) query for dictionary $\mathcal{D}_k^a$ and then remove duplicates. The complexities follow from Theorem 3.1.4(b). □

3.2.2 Processing a Periodic k-Dictionary

Our solution for periodic patterns relies on the well-studied theory of *runs* in strings. Let $\mathcal{R}$ be the set of all runs in T. Following [112], we construct for all $k \in [0, \lfloor \log n \rfloor]$ the sets of runs $\mathcal{R}_k = \{R \in \mathcal{R} : \mathsf{per}(R) \le \frac{2^k}{3}, |R| \ge 2^k\}$ in $\mathcal{O}(n)$ time overall. Note that these sets are not disjoint; however, $|\mathcal{R}_k| = \mathcal{O}(\frac{n}{2^k})$ (cf. Lemma 3.2.5 below) and thus their total

size is $\mathcal{O}(n)$. We denote the overlap $T[\max\{i_1, i_2\} \mathinner{.\,.} \min\{j_1, j_2\}]$ of U and V by $U \cap V$. If U is a fragment of T, by $\mathcal{R}_k(U) \subseteq \mathcal{R}_k$ we denote the set of all runs $R \in \mathcal{R}_k$ such that $|R \cap U| \geq 2^k$.

Lemma 3.2.5 ([112, Lemma 4.4.7])**.** $|\mathcal{R}_k(U)| = \mathcal{O}\big(\frac{1}{2^k}|U|\big)$.

Strategy. Given a fragment $U = T[t \mathinner{.\,.} j]$, we will first identify all runs $\mathcal{R}_k(U)$ of $\mathcal{R}_k$ that have a sufficient overlap with U. There is a constant number of them by Lemma 3.2.5. For an occurrence of a pattern $P \in \mathcal{D}_k^p$ in U, the unique run R extending this occurrence of P must be in $\mathcal{R}_k(U)$. We say that such an occurrence of P is *induced* by run R. We will preprocess the runs in order to be able to compute a unique (the leftmost) occurrence induced by run R for each such pattern P.

Lemma 3.2.6. *Let U be a fragment of T of length at most 2^{k+1}. Then $\mathcal{R}_k(U)$ can be retrieved in $\mathcal{O}(1)$ time after an $\mathcal{O}(n)$-time preprocessing.*

Proof. Recall that, given a periodic fragment V of the text T, we can compute the run extending V in $\mathcal{O}(1)$ time after $\mathcal{O}(n)$-time preprocessing (cf. Theorem 2.6.5).

Let us cover all positions of U using $\mathcal{O}(\frac{1}{2^k}|U|)$ fragments of length $\frac{2^{k+1}}{3}$ with overlaps of at least $\frac{2^k}{3}$. For each fragment in the cover, we first check if it is periodic using a 2-period query (cf. Theorem 2.6.4), and, if so, compute the run extending it using Theorem 2.6.5. For each run $R \in \mathcal{R}_k(U)$ with sufficient overlap, $R \cap U$ must contain a fragment V in the cover and its periodic extension must be R since $|V| \geq 2 \cdot \mathsf{per}(R)$. $\square$

Preprocessing. We first build the $\mathcal{D}_p^k$-modified suffix tree of T in $\mathcal{O}(n + |\mathcal{D}_p^k|)$ time. Then, we construct an array $\ell_k[1 \mathinner{.\,.} n]$ such that $T[i \mathinner{.\,.} \ell_k[i]]$ is the shortest pattern $P \in \mathcal{D}_k^p$ that occurs at position i. Note that $\ell_k[i]$ can be retrieved in $\mathcal{O}(1)$ time using a level ancestor query in the $\mathcal{D}_p^k$-modified suffix tree, as in the proof of Theorem 3.1.4(a). We then preprocess the array ℓ_k for RMQ queries.

Processing a run at query. Let us begin with a consequence of the fact that any $|\mathsf{per}(U)|$-length fragment of a periodic string U is primitive, and the *primitivity lemma*,

which states that a non-empty string V is primitive if and only if it occurs only twice in VV: as a prefix and as a suffix [55].

Lemma 3.2.7. *If a pattern P occurs in a text Q and satisfies $|P| \geq \mathsf{per}(Q)$, then P has exactly one occurrence starting within the first $\mathsf{per}(Q)$ positions of Q.*

Proof. First, let us prove that P has at least one occurrence in the first $q := \mathsf{per}(Q)$ positions of Q. Suppose, for the sake of contradiction, that the leftmost occurrence of P in Q is at some position $j > q$. Then, by periodicity, we have that $Q[j \mathinner{.\,.} j + |P| - 1] = Q[j - q \mathinner{.\,.} j + |P| - 1 - q] = P$, a contradiction.

We now proceed to showing that P has exactly one occurrence in the first q positions of Q. Let us assume, towards a contradiction, that this is not the case and that P occurs at positions $i, j \in [1, q]$, with $i < j$. Now, let us consider $V := Q[i \mathinner{.\,.} i+q-1] = Q[j \mathinner{.\,.} j+q-1]$. $V = Q[j \mathinner{.\,.} j + q - 1]$ occurs in $VV = Q[i \mathinner{.\,.} i + 2q - 1]$. Hence, V is not primitive, a contradiction. □

We use RMQs repeatedly, as in the proof of Theorem 3.1.4(b), for the subarray of ℓ_k corresponding to the first $\mathsf{per}(R)$ positions of $R \cap U$. This way, due to Lemma 3.2.7, we compute precisely the positions where a pattern $P \in \mathcal{D}_k^p$ has its leftmost occurrence in $R \cap U$. The number of positions identified for a single run $R \in \mathcal{R}_k(U)$ is therefore upper bounded by the number of distinct patterns occurring within $R \cap U$. We then report all distinct patterns occurring within $R \cap U$ by processing each such starting position using Lemma 3.1.3. There is no double-reporting while processing a single run, by Lemma 3.2.7 and hence the time required to process each run is $\mathcal{O}(1 + |output|)$—$|output|$ here refers to the number of distinct patterns from $\mathcal{D}_k^p$ occurring within U. Since $|\mathcal{R}_k(U)| = \mathcal{O}(1)$, we report each pattern a constant number of times and the overall time required is $\mathcal{O}(1 + |output|)$.

We have thus proved the following lemma.

Lemma 3.2.8. $\textsc{ReportDistinct}(t, j)$ *queries for the periodic k-dictionary $\mathcal{D}_k^p$ and $j - t \leq 2^{k+1}$ can be answered in $\mathcal{O}(1 + |output|)$ time with a data structure of size $\mathcal{O}(n + |\mathcal{D}_k^p|)$, that can be constructed in $\mathcal{O}(n + |\mathcal{D}_k^p|)$ time.*

Overall, by summing over all $k \in [0, \lfloor \log n \rfloor]$, we obtain a data structure of size $\mathcal{O}(n \log n + d)$, that can be built in time $\mathcal{O}(n \log n + d)$, and answers $\textsc{ReportDistinct}(i, j)$ queries in $\mathcal{O}(\log n + |output|)$ time. In the next subsection, we reduce the space occupied by our data structure.

3.2.3 Reducing the Space

The space occupied by our data structure can be reduced to $\mathcal{O}(n + d)$. We store the $\mathcal{D}$-modified suffix tree and mark all nodes from each $\mathcal{D}_k^i$, for $k \in [0, \lfloor \log n \rfloor]$ and $i \in \{a, p\}$, with a different colour. For each dictionary $\mathcal{D}' = \mathcal{D}_k^i$, we further store, using $\mathcal{O}(|\mathcal{D}'|)$ space, the $\mathcal{D}'$-modified suffix tree without its leaves.

We will show below that the only additional operation we now need to support is determining the parent of a given leaf in the original $\mathcal{D}'$-modified suffix tree (before the leaves were chopped). This can be done using the nearest coloured ancestor data structure of [79] over the $\mathcal{D}$-modified suffix tree. For a tree of size N, it achieves $\mathcal{O}(\log \log N)$ time per query after $\mathcal{O}(N)$-time preprocessing. We can, however, exploit the fact that we only have $\mathsf{palette} = \mathcal{O}(\log n)$ colours to obtain constant-time queries within the same construction time.

It is shown in [79] that, in order to answer nearest coloured ancestor queries in a tree with N nodes, it is enough to store some arrays of total size $\mathcal{O}(N)$ and predecessor data structures for $\mathcal{O}(\mathsf{palette})$ subsets of $[1, 2N]$ whose total size is $\mathcal{O}(N)$. The time needed to compute the sets for the predecessor data structures and the arrays is $\mathcal{O}(N)$. The time complexity of the query is proportional to the time required for answering a constant number of predecessor queries over the aforementioned sets. We implement a predecessor data structure for a set $S \subseteq [1, 2N]$ using $\mathcal{O}(N)$ bits of space as follows. We store a bitmap that has the i-th bit set if and only if $i \in S$ and augment it with a data structure that answers rank and select queries in $\mathcal{O}(1)$ time and requires $o(N)$ additional bits of space [101, 49]. Such a component can be constructed in $\mathcal{O}(N/\log N)$ time [22, 134]. Note that $\mathsf{pred}_S(i) = \mathsf{select}(\mathsf{rank}(i))$. We thus use $\mathcal{O}((n + d) \log n)$ bits, i.e. $\mathcal{O}(n + d)$ machine words in total for the part of the data structure responsible for

reporting occurrences starting at given positions.

For coloured range reporting, the main component of the data structure underlying Theorem 3.2.1 from [136] is an RMQ data structure over array $J[i] = \max\{j : j < i,\ A[i] = A[j]\}$. It was shown in [71] that we can implement an $\mathcal{O}(1)$-query-time RMQ data structure for an array of size N using $\mathcal{O}(N)$ bits. This data structure only returns the index of the minimum value in the given range. We build an $\mathcal{O}(|J|)$-bits RMQ data structure over J. The query procedure however, needs access to A, i.e. the colours. We can retrieve the value of the i-th element in our array of colours using our representation of the $\mathcal{D}'$-modified suffix tree, since the its colour corresponds to the parent of the respective leaf in the D'-modified suffix tree.

Then, filtering $\mathcal{O}(|output|)$ starting positions in the periodic case, is based on RMQ queries over multiple arrays of total length $\mathcal{O}(n \log n)$. To construct them, we build the $\mathcal{D}'$-modified suffix trees one by one and build the relevant RMQ data structures that require $\mathcal{O}(n \log n)$ bits in total before chopping the leaves. The actual value at the indices returned by RMQ queries can, as above, be determined using our representation of the $\mathcal{D}'$-modified suffix tree.

We arrive at the main result of this section.

Theorem 3.2.9. *Given a text T of length n and an internal dictionary $\mathcal{D}$ of size d, we can construct, in $\mathcal{O}(n \log n + d)$ time, a data structure of size $\mathcal{O}(n + d)$ that answers* ReportDistinct(i, j) *queries in $\mathcal{O}(\log n + |output|)$ time.*

3.3 Count(i, j) Queries

We first solve an auxiliary problem and show how it can be employed to give an unsatisfactory solution for Count(i, j). We then refine our approach using recompression and obtain the following result.

Theorem 3.3.1. *Given a text T of length n and an internal dictionary $\mathcal{D}$ of size d, we can construct, in $\mathcal{O}(n \log n / \log\log n + d \log^{3/2} n)$ time, a data structure of size $\mathcal{O}(n + d \log n)$ that answers* Count(i, j) *queries in $\mathcal{O}(\log^2 n / \log\log n)$ time.*

3.3.1 An Auxiliary Problem

By inter-position $i+1/2$ we refer to a location between positions i and $i+1$ in T. We also refer to inter-positions $1/2$ and $n+1/2$. We consider the following auxiliary problem, in which we are given a set of inter-positions (*breakpoints*) B of P and upon query we are to compute all fragments of $T[i\,..\,j]$ that match P and align a specific inter-position (*anchor*) β of the text with some inter-position in B.

> Breakpoints-Anchor IPM
>
> **Input:** A text T of length n, a substring P of T of length m, and a set B of inter-positions (breakpoints) of P.
>
> **Query:** BA-Count(β, i, j): the number of fragments $T[r\,..\,r+m-1]$ of $T[i\,..\,j]$ that match P such that $\beta - r + 1 \in B$ (β is an anchor).

Recall that in the 2D orthogonal range counting problem, one is to preprocess an $n \times n$ grid with $\mathcal{O}(n)$ marked points so that upon query $[x_1, y_1] \times [x_2, y_2]$, the number of points in this rectangle can be computed efficiently. In the (dual) 2D range stabbing counting problem, one is to preprocess the grid with $\mathcal{O}(n)$ rectangles so that upon query (x, y) the number of (stabbed) rectangles that contain (x, y) can be retrieved efficiently. The counting version of range stabbing queries in 2D reduces to two-sided range counting queries in 2D as follows (cf. [143]). For each rectangle $[x_1, y_1] \times [x_2, y_2]$ in grid $\mathcal{G}$, we add points (x_1, y_1) and (x_2+1, y_2+1) with weight 1 and points (x_1, y_2+1) and (x_2, y_1+1) with weight -1 in a grid G'. Then the number of rectangles stabbed by point (a, b) in $\mathcal{G}$ is equal to the sum of weights of points in the quarterplane $(-\infty, a] \times (-\infty, b]$ in $\mathcal{G}'$. We will use Theorem 2.5.4 (for orthogonal range counting) in our solution for Breakpoints-Anchor IPM.

Data structure. Let $W_1 = \{P[\lceil b \rceil\,..\,m] : b \in B\}$ and consider the set W_2 obtained by adding $U\$$ and $U\#$ for each element U of W_1 to an initially empty set, where \$ is a letter smaller (resp. # is larger) than all the letters in Σ. Let W be the compact trie for the set of strings W_2. For each internal node v of W that does not have an outgoing edge with label \$, we add such a (leftmost) edge with a leaf attached to its

endpoint. W can be constructed in $\mathcal{O}(|B|)$ time after an $\mathcal{O}(n)$-time preprocessing of T, allowing for constant-time longest common prefix queries; cf. [55]. We also build the W_1-modified suffix tree of T and preprocess it for weighted ancestor queries. We keep two-sided pointers between nodes of W and of the W_1-modified suffix tree of T that have the same path-label.

Similarly, let W^R be the compact trie for set Z_2 consisting of elements $U\$$ and $U\#$ for each $U \in Z_1 = \{(P[1 \mathinner{.\,.} \lfloor b \rfloor])^R : b \in B\}$. We preprocess the pair (W^R, Z_1) analogously to how we preprocess the pair (W, W_1)—explained in the previous paragraph.

Note that each of the tries has at most $k = \mathcal{O}(|B|)$ leaves. Let us now consider a 2D grid of size $k \times k$, whose x-coordinates (resp. y-coordinates) correspond to the leaves of W (resp. W^R). For each $b \in B$ we do the following. Let x_1 and x_2 be the leaves with path-label $P[\lceil b \rceil \mathinner{.\,.} m]\$$ and $P[\lceil b \rceil \mathinner{.\,.} m]\#$ in W, respectively. Similarly, let y_1 and y_2 be the leaves with path-label $(P[1 \mathinner{.\,.} \lfloor b \rfloor])^R\$$ and $(P[1 \mathinner{.\,.} \lfloor b \rfloor])^R\#$ in W^R, respectively. We add the rectangle $R_b = [x_1, y_1] \times [x_2, y_2]$ to the grid. An illustration is provided in Figure 3.3. We then preprocess the grid for the counting version of 2D range stabbing queries, employing Theorem 2.5.4.

Query. Let the longest prefix of $T[\lceil \beta \rceil \mathinner{.\,.} j]$ that is a prefix of an element of W_1 be U and its locus in W be u. This can be computed in $\mathcal{O}(\log \log n)$ time using a weighted ancestor query in the W_1-modified suffix tree of T and following the pointer to W. If u is an explicit node, we follow the edge with label \$, while if it is implicit along edge (p, q), we follow the edge with label \$ from p. In either case, we reach a leaf u'. We do the symmetric procedure with $(T[i \mathinner{.\,.} \lfloor \beta \rfloor])^R$ in the Z_1-modified suffix tree of T and obtain a leaf v' in W^R.

Observation 3.3.2. *The number of fragments $T[r \mathinner{.\,.} t] = P$ with $r, t \in [i \mathinner{.\,.} j]$ and $\beta - r + 1 \in B$ is equal to the number of rectangles stabbed by the point of the grid defined by u' and v'.*

The observation holds because this point is inside rectangle R_b for $b \in B$ if and only if $P[\lceil b \rceil \mathinner{.\,.} m]$ is a prefix of $T[\lceil \beta \rceil \mathinner{.\,.} j]$ and $P[1 \mathinner{.\,.} \lfloor b \rfloor]$ is a suffix of $T[i \mathinner{.\,.} \lfloor \beta \rfloor]$. This concludes

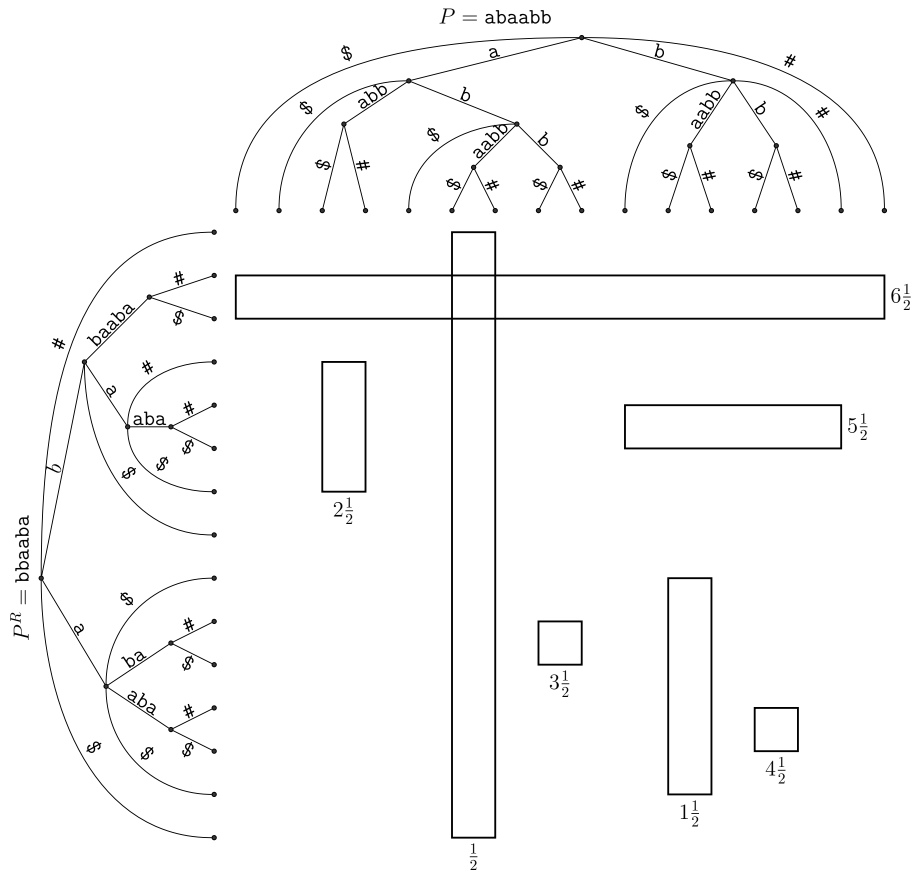


Figure 3.3: Example of the construction of rectangles in the proof of Lemma 3.3.3 for $P =$ abaabb and breakpoints $i + 1/2$ for $i = 0, 1, 2, 3, 4, 5, 6$. Each rectangle is annotated with its breakpoint.

the proof of the following result.

Lemma 3.3.3. Breakpoints-Anchor IPM *queries can be answered in* $\mathcal{O}(\frac{\log n}{\log \log n})$ *time with a data structure of size* $\mathcal{O}(n+|B|)$ *that can be constructed in time* $\mathcal{O}(n+|B|\sqrt{\log |B|})$.

Let us now define problem Breakpoints-Anchor IDM, which is a generalisation of the Breakpoints-Anchor IPM problem.

Breakpoints-Anchor IDM **Input:** A text T of length n, a dictionary $\mathcal{D}$ of substrings of T, and a set B_P of inter-positions (breakpoints) for each $P \in \mathcal{D}$. **Query:** BA-Count(β, i, j): the number of fragments $T[r \mathinner{.\,.} r+m-1]$ of $T[i \mathinner{.\,.} j]$ that match some $P \in \mathcal{D}$ such that $\beta - r + 1 \in B_P$ (β is an anchor).

We obtain the following lemma by a straightforward generalisation of the proof of Lemma 3.3.3: we build trie W for the union of the sets W_2 defined in the above proof for each pattern (similarly for W^R), and add all rectangles to a single grid; the query procedure is identical.

Lemma 3.3.4. Breakpoints-Anchor IDM *queries can be answered in* $\mathcal{O}(\frac{\log n}{\log\log n})$ *time with a data structure of size* $\mathcal{O}(n + \sum_{P\in\mathcal{D}} |B_P|)$*. The data structure can be constructed in time* $\mathcal{O}(n + \sqrt{\log n}\sum_{P\in\mathcal{D}} |B_P|)$*.*

A warm-up solution for Count(i, j). Lemma 3.3.4 can be applied naively to answer Count(i, j) queries as follows. Let us set $B_P = \{p + 1/2 : p \in [1, |P| - 1]\}$ for each pattern $P \in \mathcal{D}$ and construct the data structure of Lemma 3.3.4. We build a balanced binary tree BT on top of the text and for each node v in BT define $\mathsf{val}(v)$ to be the fragment consisting of the characters corresponding to the leaves in the subtree of v. Note that if v is a leaf, then $|\mathsf{val}(v)| = 1$; otherwise, $\mathsf{val}(v) = \mathsf{val}(u_\ell)\mathsf{val}(u_r)$, where u_ℓ and u_r are the children of v. For each node v in BT, we precompute and store the count for $\mathsf{val}(v)$, defined as the number of occurrences of patterns from $\mathcal{D}$ in $\mathsf{val}(v)$. If v is a leaf, this count can be determined easily. Otherwise, each occurrence is contained in $\mathsf{val}(u_\ell)$, is contained in $\mathsf{val}(u_r)$, or spans both $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. Hence, we sum the answers for the children u_ℓ and u_r of v and add the result of a Breakpoints-Anchor IDM query in $\mathsf{val}(v)$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$.

To answer a query concerning $T[i \mathinner{.\,.} j]$, we recursively count the occurrences in the intersection of $\mathsf{val}(v)$ with $T[i \mathinner{.\,.} j]$, starting from the root r of BT as it satisfies $\mathsf{val}(r) = T[1 \mathinner{.\,.} n]$. If the intersection is empty, the result is 0, and if $\mathsf{val}(v)$ is contained in $T[i \mathinner{.\,.} j]$, we can use the precomputed count. Otherwise, we recurse on the children u_ℓ

and u_r of v and sum the resulting counts. It remains to add the number of occurrences spanning across both $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. This value is non-zero only if $T[i\,..\,j]$ spans both these fragments, and it can be determined from a BREAKPOINTS-ANCHOR IDM query in the intersection of $\mathsf{val}(v)$ and $T[i\,..\,j]$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$.

The query time is $\mathcal{O}(\log^2 n/\log\log n)$ since non-trivial recursive calls are made only for nodes on the paths from the root r to the leaves representing $T[i]$ and $T[j]$. Nevertheless, the space required for this solution can be $\Omega(nd)$, which is unacceptable. Below, we refine this technique using a locally consistent parsing; our goal is to decrease the size of each set B_P from $\Theta(|P|)$ to $\mathcal{O}(\log n)$.

3.3.2 Using Recompression

Data Structure. First, we build an RLSLP generating T using the recompression technique [102, 103, 98]. Let us denote the parse tree of this RLSLP by PT. We then construct the set $L(P)$ of size $\mathcal{O}(\log n)$ for each pattern $P \in \mathcal{D}$ (see Section 2.7). These sets have the following crucial property, restated here for convenience.

Lemma 2.7.2. *Let v denote a non-leaf node of a parse tree $\mathsf{PT}[S]$ stemming from recompression and let $S[a\,..\,b]$ denote an occurrence of a string U contained in $\mathsf{val}(v)$, but not contained in $\mathsf{val}(u)$ for any child u of v. If $S[a\,..\,c]$ is the longest prefix of $S[a\,..\,b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\,..\,c]| \in L(U)$. Symmetrically, if $S[c'+1\,..\,b]$ is the longest suffix of $S[a\,..\,b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\,..\,c']| \in L(U)$.*

Then, we also construct the component of Lemma 3.3.4 with $B_P = \{i + \frac{1}{2} : i \in L(P)\}$ for each pattern $P \in \mathcal{D}$. Moreover, for every symbol A we store the number of occurrences of patterns from $\mathcal{D}$ in $\mathsf{gen}(A)$. Additionally, if $A \to B^k$ is a power, we also store the number of occurrences in $\mathsf{gen}(B^i)$ for $i \in [1,k]$. The space consumption is $\mathcal{O}(n + d\log n)$ since $|B_P| = \mathcal{O}(\log n)$ for each $P \in \mathcal{D}$.

Efficient preprocessing. The RLSLP and the parse tree are built in $\mathcal{O}(n)$ time, and the sets B_P are computed in $\mathcal{O}(d\log n)$ time using Theorem 2.7.1. The data structure

of Lemma 3.3.4 is then constructed in $\mathcal{O}(n+d\log^{3/2} n)$ time. Next, we process the RLSLP in a bottom-up fashion. If A is a terminal, its count is easily determined. If $A \to BC$ is a concatenation, we sum the counts for B and C and the number of occurrences spanning both $\mathsf{gen}(B)$ and $\mathsf{gen}(C)$. To determine the latter value, we fix an arbitrary node v with label A and denote its children u_ℓ, u_r. By Lemma 2.7.2, any occurrence of P intersecting both $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$ has a breakpoint aligned to the inter-position between the two fragments. Hence, the third summand is the result of a BREAKPOINTS-ANCHOR IDM query in $\mathsf{val}(v)$ with the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. Finally, if $A \to B^k$, then to determine the count in $\mathsf{gen}(B^i)$, we add the count for B, the count in $\mathsf{gen}(B^{i-1})$, and the number of occurrences in B^i spanning both the prefix B and the suffix B^{i-1}. To find the latter value, we fix an arbitrary node v with label A, denote its children $u_1, \dots, u_k$, and make a BREAKPOINTS-ANCHOR IDM query in $\mathsf{val}(u_1)\cdots\mathsf{val}(u_i)$ with the anchor between $\mathsf{val}(u_1)$ and $\mathsf{val}(u_2)$. The correctness of this step follows from Lemma 2.7.2. The running time of the last phase is $\mathcal{O}(n\log n/\log\log n)$, so the overall construction time is $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$.

Query. Upon a query COUNT(i, j), we proceed essentially as in the warm-up solution: we recursively count the occurrences contained in the intersection of $T[i\mathinner{.\,.} j]$ with $\mathsf{val}(v)$ for nodes v in PT, starting from the root of PT. If the two fragments are disjoint, the result is 0, and if $\mathsf{val}(v)$ is contained in $T[i\mathinner{.\,.} j]$, it is the count precomputed for the label of v. Otherwise, the label of v is a non-terminal. If it is a concatenation symbol, we recurse on both children u_ℓ, u_r of v and sum the obtained counts. If $T[i\mathinner{.\,.} j]$ spans both $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$, we also add the result of a BREAKPOINTS-ANCHOR IDM query in the intersection of $T[i\mathinner{.\,.} j]$ with $\mathsf{val}(v)$ and the anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_r)$. If the label is a power symbol $A \to B^k$, we determine which of the children $u_1, \dots, u_k$ of v are spanned by $T[i\mathinner{.\,.} j]$. We denote these children by $u_\ell, \dots, u_r$ and recurse on u_ℓ and on u_r. If $r > \ell$, we also make a BREAKPOINTS-ANCHOR IDM query in the intersection of $T[i\mathinner{.\,.} j]$ with $\mathsf{val}(u_\ell)\cdots\mathsf{val}(u_r)$ and anchor between $\mathsf{val}(u_\ell)$ and $\mathsf{val}(u_{\ell+1})$. If $r > \ell + 1$, we further add the precomputed value for $\mathsf{gen}(B^{r-\ell-1})$ to account for the occurrences

contained in $\mathsf{val}(u_{\ell+1})\cdots\mathsf{val}(u_{r-1})$ and make a BREAKPOINTS-ANCHOR IDM query in the intersection of $T[i\,..\,j]$ with $\mathsf{val}(u_{\ell+1})\cdots\mathsf{val}(u_r)$ and anchor between u_{r-1} and u_r. By Lemma 2.7.2, the answer is the sum of the up to five values computed. The overall query time is $\mathcal{O}(\log^2 n/\log\log n)$, since we make $\mathcal{O}(\log n)$ non-trivial recursive calls and each of them is processed in $\mathcal{O}(\log n/\log\log n)$ time.

3.4 CountDistinct(i, j) Queries

We first show how to apply geometric methods to a special variant of the COUNTDISTINCT problem, where we are interested in a small subset of occurrences of each pattern.

A quarterplane is a range of the form $(-\infty, x_1] \times (-\infty, x_2]$. By reversing coordinates we can also consider quarterplanes with some dimensions of the form $[x_i, \infty)$. Let us state the following result on orthant colour range counting due to Kaplan et al. [105] in the special case of two dimensions.

Theorem 3.4.1 ([105, Theorem 2.3])**.** *Given n coloured integer points in 2D, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$-size data structure that, given any quarterplane Q, counts the number of distinct colours with at least one point in Q in $\mathcal{O}(\log n)$ time.*

Let $\mathcal{D} = \{P_1, P_2, \dots, P_d\}$ and $\mathcal{S}$ be a family of sets $S_1, \dots, S_d$ such that $S_k \subseteq \mathsf{Occ}(P_k)$, where $\mathsf{Occ}(P_k)$ is the set of positions of T where P_k occurs. Let $\|\mathcal{S}\| = \sum_k |S_k|$. For each pattern P_k, we call the positions in the set S_k the *special positions of* P_k. Counting distinct patterns occurring at their special positions in $T[i\,..\,j]$ is called COUNTDISTINCT$_{\mathcal{S}}(i, j)$.

Lemma 3.4.2. COUNTDISTINCT$_{\mathcal{S}}(i, j)$ *queries can be answered in $\mathcal{O}(\log n)$ time with a data structure of size $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ that can be constructed in $\mathcal{O}(n + \|\mathcal{S}\| \log n)$ time.*

Proof. We assign a different integer colour c_k to every pattern $P_k \in \mathcal{D}$. Then, for each fragment $T[a\,..\,b] = P_k$ such that $a \in S_k$, we add point (a, b) with colour c_k to an initially empty 2D grid $\mathcal{G}$. A COUNTDISTINCT$_{\mathcal{S}}(i, j)$ query reduces to counting different colours in the range $[i, \infty) \times (-\infty, j]$ of $\mathcal{G}$. The complexities follow from Theorem 3.4.1. □

3.4.1 2-Approximate Counting

CountDistinct for Extended or Contracted Fragments. For two positions ℓ and r, we define $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ as the longest prefix of $T[\ell \mathinner{.\,.} r]$ that matches some pattern $P \in \mathcal{D}$; the length of such prefix is at most $r - \ell + 1$. Let us show how to compute the locus of $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$ in the $\mathcal{D}$-modified suffix tree $\mathcal{T}_{T,\mathcal{D}}$. To this end, we preprocess $\mathcal{T}_{T,\mathcal{D}}$ for weighted ancestor queries and store at every node v of $\mathcal{T}_{T,\mathcal{D}}$ a pointer $p(v)$ to the nearest ancestor u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$. To return $\mathsf{Pref}_{\mathcal{D}}(\ell, r)$, we find the locus u of $T[\ell \mathinner{.\,.} r]$ in the $\mathcal{D}$-modified suffix tree. We return $p(u)$ if $|\mathcal{L}(u)| = |T[\ell \mathinner{.\,.} r]|$ and $p(v)$, where v is the parent of u, otherwise.

Lemma 3.4.4 applies the $\mathcal{D}$-modified suffix tree and the following internal queries to the problem of maintaining the count of distinct patterns occurring in a fragment subject to extending or shrinking the fragment.

In a *bounded LCP query*, one is given two fragments U and V of T and needs to return the longest prefix of U that occurs in V; we denote such a query by $\mathsf{BLCP}(U, V)$. Kociumaka et al. [115] presented several tradeoffs for this problem, including the following.

Theorem 3.4.3 ([115],[112, Corollary 7.3.4])**.** *Given a text T of length n, one can construct in $\mathcal{O}(n\sqrt{\log n})$ time an $\mathcal{O}(n)$-size data structure that answers BLCP queries in $\mathcal{O}(\log^{\epsilon} n)$ time, for any constant $\epsilon > 0$.*

Lemma 3.4.4. *For any constant $\epsilon > 0$, given* $\textsc{CountDistinct}(i, j)$*, one can compute* $\textsc{CountDistinct}(i \pm 1, j)$ *and* $\textsc{CountDistinct}(i, j \pm 1)$ *in $\mathcal{O}(\log^{\epsilon} n)$ time with a data structure of size $\mathcal{O}(n + d)$ that can be constructed in $\mathcal{O}(n\sqrt{\log n} + d)$ time.*

Proof. We only present a data structure for $\textsc{CountDistinct}(i \pm 1, j)$ queries. Queries $\textsc{CountDistinct}(i, j \pm 1)$ can be handled analogously by building the same data structure for the reverses of all the strings in scope.

We show how to compute the number of patterns $P \in \mathcal{D}$ whose only occurrence in some fragment $T[\ell \mathinner{.\,.} r]$ starts at position ℓ. The computation of $\textsc{CountDistinct}(i \pm 1, j)$ follows directly by setting $j = r$ and ℓ equal to $i - 1$ or i.

Data structure. We preprocess T for BLCP queries (using Theorem 3.4.3) and construct the $\mathcal{D}$-modified suffix tree of T. In addition, we preprocess this tree for weighted ancestor queries and store at every node v of the tree the number $\#(v)$ of the ancestors u (including v) of v such that $\mathcal{L}(u) \in \mathcal{D}$.

Query. We want to count patterns longer than $k = |\mathsf{BLCP}(T[\ell \mathinner{.\,.} r], T[\ell+1 \mathinner{.\,.} r])|$. Let $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell+k-1)$ and $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. The desired number of patterns is equal to $\#(v) - \#(u)$. See Figure 3.4 for a visualisation. □

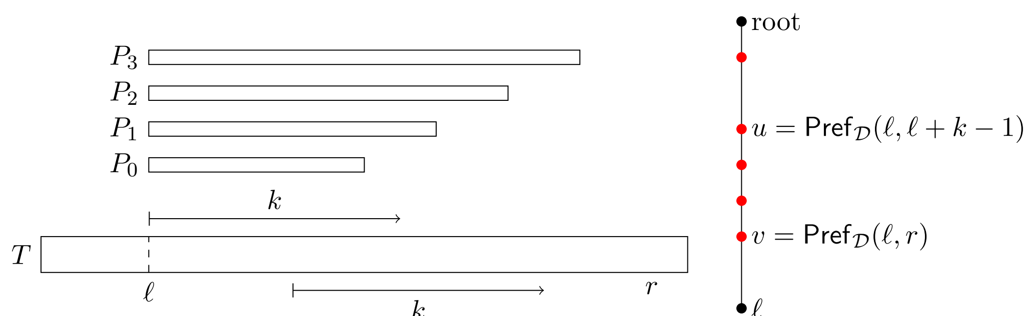


Figure 3.4: The setting in Lemma 3.4.4. Left: Text T. Right: The path from the root of the $\mathcal{D}$-modified suffix tree of T to the leaf with path-label $T[\ell \mathinner{.\,.} n]\$$. The nodes of the path whose path-labels match some patterns from $\mathcal{D}$ are drawn in red. Here, P_0 is the longest pattern that occurs at ℓ and also has an occurrence in $T[\ell+1 \mathinner{.\,.} r]$; its locus in the tree is $u = \mathsf{Pref}_{\mathcal{D}}(\ell, \ell+k-1)$. The patterns that occur in $T[\ell \mathinner{.\,.} r]$ only at position ℓ are P_1, P_2 and P_3. The locus of P_3 is $v = \mathsf{Pref}_{\mathcal{D}}(\ell, r)$. Then, $\#(v) - \#(u) = 5 - 2 = 3$.

An Auxiliary Operation. Two fragments $U = T[i_1 \mathinner{.\,.} j_1]$ and $V = T[i_2 \mathinner{.\,.} j_2]$ are called *consecutive* if $i_2 = j_1 + 1$.

3-Fragments-Counting

Input: A text T of length n and a dictionary $\mathcal{D}$ consisting of d patterns

Query: Given three consecutive fragments F_1, F_2, F_3 in T such that $|F_1| = |F_3|$ and $|F_2| \geq 8 \cdot |F_1|$, count distinct patterns P from $\mathcal{D}$ that have an occurrence starting in F_1 and ending in F_3 and do not occur in either F_1F_2 or F_2F_3

Let us fix $|F_1| = |F_3| = x$ and $|F_2| = y \geq 8x$. Additionally, let us call an occurrence of $P \in \mathcal{D}$ that starts in fragment F_a and ends in fragment F_b an (F_a, F_b)-occurrence. We will call an (F_1, F_3)-occurrence an *essential occurrence.*

We say that a string S is *highly periodic* if $\mathsf{per}(S) \leq \frac{1}{4}|S|$. We first consider the case where all patterns in $\mathcal{D}$ are not highly periodic.

Lemma 3.4.5. *If each $P \in \mathcal{D}$ is not highly periodic, then*

$$\textsc{3-Fragments-Counting}(F_1, F_2, F_3) = \\ \textsc{Count}(F_1F_2F_3) - \textsc{Count}(F_1F_2) - \textsc{Count}(F_2F_3) + \textsc{Count}(F_2).$$

Proof. Let us start with the following claim.

Claim 3.4.6. *Any $P \in \mathcal{D}$ that has an essential occurrence occurs exactly once in $F_1F_2F_3$.*

Proof. We have $|F_1F_2F_3| = x + y + x = 2x + y$. String P has an essential occurrence, so $|P| \geq y$. Therefore, if there are two occurrences of P in $F_1F_2F_3$, then they overlap in

$$2|P| - (2x + y) \geq 2|P| - (\tfrac{1}{4}|P| + |P|) = \tfrac{3}{4}|P|$$

positions. This implies that P is highly periodic, which is a contradiction. □

Claim 3.4.6 shows that $\textsc{3-Fragments-Counting}(F_1, F_2, F_3)$ is equal to the number of essential occurrences. Let us prove that the stated formula does not count any (F_a, F_b)-occurrences other than (F_1, F_3)-occurrences.

- Each (F_1, F_2)-occurrence is registered when we add $\textsc{Count}(F_1F_2F_3)$ and unregistered when we subtract $\textsc{Count}(F_1F_2)$. Similarly for (F_2, F_3)-occurrences.

- Each (F_2, F_2)-occurrence is registered when we add $\textsc{Count}(F_1F_2F_3)$, $\textsc{Count}(F_2)$ and unregistered when we subtract $\textsc{Count}(F_1F_2)$, $\textsc{Count}(F_2F_3)$.
- Each (F_1, F_1)-occurrence is registered when we add $\textsc{Count}(F_1F_2F_3)$ and unregistered when we subtract $\textsc{Count}(F_1F_2)$. Similarly for (F_3, F_3)-occurrences. $\square$

We now proceed with answering 3-Fragments-Counting queries for the dictionary of highly periodic patterns.

Lemma 3.4.7. *If F_2 is aperiodic, then there are no essential occurrences of highly periodic patterns. Otherwise, all essential occurrences of highly periodic patterns are generated by the same run, that is,* $\mathsf{run}(F_2)$.

Proof. The first claim follows from the fact that such an occurrence of a pattern $P \in \mathcal{D}$ has an overlap of length at least $2\mathsf{per}(P)$ with F_2 and hence $\mathsf{per}(P) \le \frac{1}{2}|F_2|$ is a period of F_2.

As for the second claim, it suffices to show that, for any pattern $P \in \mathcal{D}$ that has an essential occurrence, we have $\mathsf{per}(P) = \mathsf{per}(F_2)$. The inequalities $|F_2| \ge 2\mathsf{per}(F_2)$ and $|F_2| \ge 2\mathsf{per}(P)$ imply $|F_2| \ge \mathsf{per}(F_2) + \mathsf{per}(P)$. Hence, by the periodicity lemma (Lemma 2.3.1), $q = \gcd(\mathsf{per}(P), \mathsf{per}(F_2))$ is a period of F_2. As $q \le \mathsf{per}(F_2)$, we conclude that $q = \mathsf{per}(F_2)$. Thus, $\mathsf{per}(F_2)$ divides $\mathsf{per}(P)$, and therefore $\mathsf{per}(P) = \mathsf{per}(F_2)$. This concludes the proof. $\square$

For a periodic factor U of T, let $\textsc{Periodic}(U)$ denote the set of distinct patterns from $\mathcal{D}$ that occur in U and have the same shortest period. Let us make the following observation.

Observation 3.4.8. *If all $P \in \mathcal{D}$ are highly periodic, F_2 is periodic, and $R = \mathsf{run}(F_2)$, then we have*

$$\text{3-}\textsc{Fragments-Counting}(F_1, F_2, F_3) = |\textsc{Periodic}(F_1F_2F_3 \cap R)| - |\textsc{Periodic}(F_1F_2 \cap R) \cup \textsc{Periodic}(F_2F_3 \cap R)|.$$

Next, we show how to efficiently evaluate the right-hand side of the formula in the above observation, using the orthogonal range counting data structure specified in Theorem 2.5.4.

We group all highly periodic patterns by Lyndon root and rank; for a Lyndon root L and a rank r, we denote by $\mathcal{D}^p_{L,r}$ the corresponding set of patterns. Then, we build the data structure of Theorem 2.5.4 for the set of points obtained by adding the point (a, b) for each $(L, r, a, b) \in \mathcal{D}^p_{L,r}$. We refer to the 2D grid underlying this data structure as $\mathcal{G}_{L,r}$. Note that the total number of points in the data structures over all Lyndon roots and ranks is $\mathcal{O}(d)$.

Each occurrence of a pattern (L, r, a, b) lies within some run in $\mathcal{R}$ with Lyndon root L. Let us state a simple fact.

Fact 3.4.9. *A periodic string (L, r, a, b) occurs in a periodic string (L, r', a', b') if and only if at least one of the following conditions is met:*

(a) $r = r'$, $a \le a'$, *and* $b \le b'$;

(b) $r = r' - 1$ *and* $a \le a'$;

(c) $r = r' - 1$ *and* $b \le b'$;

(d) $r \le r' - 2$.

Lemma 3.4.10. *One can compute $|\textsc{Periodic}(U)|$ for any periodic fragment U in time $\mathcal{O}(\log n/ \log\log n)$ using a data structure of size $\mathcal{O}(n + d)$ that can be constructed in time $\mathcal{O}(n + d\sqrt{\log n})$.*

Proof. For $U = (L, r, a, b)$, we count points contained in at least one of the rectangles

(a) $(-\infty, a] \times (-\infty, b]$ in $\mathcal{G}_{L,r}$,

(b) $(-\infty, a] \times (-\infty, |L|]$ in $\mathcal{G}_{L,r-1}$,

(c) $(-\infty, |L|] \times (-\infty, b]$ in $\mathcal{G}_{L,r-1}$,

and we add to the count the number of patterns of the form (L, r', a, b) with $r' < r-1$. For the latter term, it suffices to store an array $X_L[1\,..\,t]$ such that $X_L[r] = \sum_{i=1}^{r} |\mathcal{D}^p_{L,i}|$, where t is the maximum rank of a pattern with Lyndon root L. The total size of these arrays is $\mathcal{O}(n)$ by the linearity of the sum of exponents of runs in a string [26, 119]. $\square$

Remark 3.4.11. In particular, in the proof of the above lemma, we count points that are contained within at least one out of a constant number of rectangles. Therefore, not only we can easily compute $|\textsc{Periodic}(U)|$, but similarly we are able to compute $|\textsc{Periodic}(U_1) \cup \textsc{Periodic}(U_2)|$ for some periodic factors U_1, U_2 of T.

We are now ready to prove the main result of this subsection.

Lemma 3.4.12. *The* 3-FRAGMENTS-COUNTING(F_1, F_2, F_3) *queries can be answered in time* $\mathcal{O}(\log^2 n/\log\log n)$ *with a data structure of size* $\mathcal{O}(n+d\log n)$ *that can be constructed in* $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$ *time.*

Proof. By Lemma 3.4.5, in order to count the patterns that are not highly periodic, it suffices to perform three COUNT queries. To this end, we employ the data structure of Theorem 3.3.1 which answers COUNT queries in $\mathcal{O}(\log^2 n/\log\log n)$ time, occupies space $\mathcal{O}(n + d\log n)$, and can be constructed in time $\mathcal{O}(n\log n/\log\log n + d\log^{3/2} n)$.

We now proceed to counting highly periodic patterns. First, we check whether F_2 is periodic using Theorem 2.6.4. If F_2 is not periodic, then by Lemma 3.4.7 no highly periodic pattern has an essential occurrence, and we are thus done. If F_2 is periodic, three $|\textsc{Periodic}(U)|$ queries suffice to obtain the answer due to Observation 3.4.8. They can be efficiently answered due to Lemma 3.4.10 and Remark 3.4.11; the complexities are dominated by those for building the data structure for COUNT queries. $\square$

2-Approximation Algorithm. Let us fix $\delta = \frac{1}{9}$. A fragment of length $\lfloor (1+\delta)^p \rfloor$ for any positive integer p will be called a *p-basic fragment.* Our data structure stores $\textsc{CountDistinct}(i, j)$ for every basic fragment $T[i\,..\,j]$. Using Lemma 3.4.4, these values can be computed in $\mathcal{O}(n\log^{1+\epsilon} n + d)$ time with a sliding window approach. The space requirement is $\mathcal{O}(n\log n + d)$.

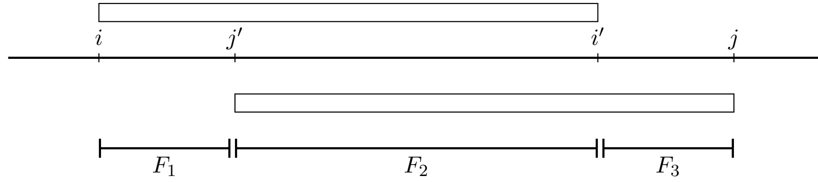


Figure 3.5: A 2-approximation of CountDistinct(i, j) is achieved using precomputed counts for basic factors $T[i\,..\,i']$ and $T[j'\,..\,j]$.

In order to answer an arbitrary CountDistinct(i, j) query, let $T[i\,..\,i']$ and $T[j'\,..\,j]$ be the longest prefix and suffix of $T[i\,..\,j]$ being a basic factor; see Figure 3.5. We sum up CountDistinct(i, i') and CountDistinct(j', j) and the result of a 3-Fragments-Counting query for $F_1 = T[i\,..\,j'-1]$, $F_2 = T[j'\,..\,i']$, $F_3 = T[i'+1\,..\,j]$. (Note that $(|F_1|+|F_2|)\cdot(1+\delta) > |F_1|+|F_2|+|F_3|$ implies $\delta(|F_1|+|F_2|) > |F_3|$, and since $|F_1| = |F_3|$, we have that $|F_1| = |F_3| \leq \frac{1}{8}|F_2|$.) Now, a pattern $P \in \mathcal{D}$ is counted at least once if and only if it occurs in $T[i\,..\,j]$. Also, a pattern $P \in \mathcal{D}$ is counted at most twice (exactly twice if and only if it occurs in both F_1F_2 and F_2F_3). The above discussion and Lemma 3.4.12 yield the following result.

Theorem 3.4.13. *Given a text T of length n and an internal dictionary $\mathcal{D}$ of size d, we can construct, in $\mathcal{O}(n\log^{1+\epsilon} n + d\log^{3/2} n)$ time, for any constant $\epsilon > 0$, a data structure of size $\mathcal{O}((n+d)\log n)$ that answers* CountDistinct(i, j) *queries 2-approximately in $\mathcal{O}(\log^2 n/\log\log n)$ time.*

3.4.2 Time-Space Tradeoffs for Exact Counting

Tradeoff for Large Dictionaries. The following result is yet another application of Lemma 3.4.4.

Theorem 3.4.14. *Given a text T of length n and an internal dictionary $\mathcal{D}$ of size d for any $m \in [1, n]$ and any constant $\epsilon > 0$, we can construct, in $\mathcal{O}((n^2\log^{\epsilon} n)/m + n\sqrt{\log n} + d)$ time, a data structure of size $\mathcal{O}(n^2/m^2 + n + d)$ that answers* CountDistinct(i, j) *queries in $\mathcal{O}(m\log^{\epsilon} n)$ time.*

Proof. A fragment of the form $T[c_1m + 1 \mathinner{.\,.} c_2m]$ for integers c_1 and c_2 will be called a *canonical fragment*. Our data structure stores $\textsc{CountDistinct}(i', j')$ for every canonical fragment $T[i' \mathinner{.\,.} j']$ and the data structure of Lemma 3.4.4. Hence, the space complexity is $\mathcal{O}(n^2/m^2 + n + d)$.

We can compute in $\mathcal{O}(n \log^\epsilon n)$ time $\textsc{CountDistinct}(i', j)$ for a given i' and all j using Lemma 3.4.4. There are $\mathcal{O}(n/m)$ starting positions of canonical fragments and hence the counts for all canonical fragments can be computed in $\mathcal{O}((n^2 \log^\epsilon n)/m)$ time. An additional additive $\mathcal{O}(n\sqrt{\log n} + d)$ factor in the preprocessing time originates from Lemma 3.4.4.

We can answer a $\textsc{CountDistinct}(i, j)$ query in $\mathcal{O}(m \log^\epsilon n)$ time as follows. Let $T[i' \mathinner{.\,.} j']$ be the maximal canonical fragment contained in $T[i \mathinner{.\,.} j]$. We first retrieve $\textsc{CountDistinct}(i', j')$ for $T[i' \mathinner{.\,.} j']$. Then, we apply Lemma 3.4.4 $\mathcal{O}(m)$ times; each time we extend the fragment for which we count, until we obtain $\textsc{CountDistinct}(i, j)$. See Figure 3.6 for an illustration. □

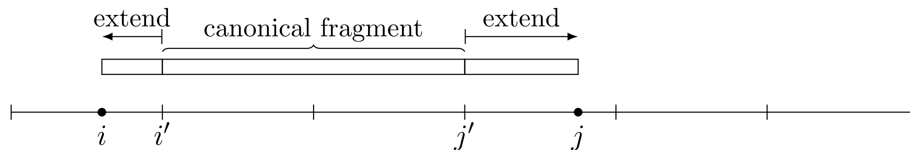


Figure 3.6: An illustration of the setting in the query algorithm underlying Theorem 3.4.14.

Tradeoff for Small Dictionaries. We call a set of strings Π a *path-set* if all elements of Π are prefixes of its longest element. We now show how to efficiently handle dictionaries that do not contain large path-sets.

Lemma 3.4.15. *If $\mathcal{D}$ does not have any subset of size greater than k that is a path-set, then we can construct in $\mathcal{O}(kn \log n)$ time an $\mathcal{O}(kn \log n)$-size data structure that answers $\textsc{CountDistinct}(i, j)$ queries in $\mathcal{O}(\log n)$ time.*

Proof. Let $\mathcal{D} = \{P_1, \ldots, P_d\}$ and $\mathcal{S} = \{\mathsf{Occ}(P_1), \ldots, \mathsf{Occ}(P_d)\}$. Every position of T contains at most k occurrences of patterns from $\mathcal{D}$. This implies that $\|\mathcal{S}\| \leq kn$. We can obviously treat a $\textsc{CountDistinct}(i, j)$ query as a $\textsc{CountDistinct}_{\mathcal{S}}(i, j)$ query. The complexities follow from Lemma 3.4.2. □

Lemma 3.4.16. *For any $k \in [1, n]$, we can compute a maximal family $\mathcal{F}$ of pairwise-disjoint path-sets in $\mathcal{D}$, each consisting of at least k elements, in $\mathcal{O}(n + d)$ time.*

Proof. Let us consider the $\mathcal{D}$-modified suffix tree of T and call every internal node that has no descendant internal nodes a *bottom node*. As the considered path-sets are maximal, the longest string in any path-set $\Pi \in \mathcal{F}$ is the path-label of a bottom node. We preprocess the tree, so that for each bottom node u we store a counter $C(u)$ equal to the number of non-root nodes on the root-to-u path.

Then, we perform a preorder traversal of the tree. This way all bottom nodes are considered in a left-to-right order. When adding a path-set to $\mathcal{F}$, we mark all nodes of that path-set. During our traversal we can easily maintain the number N of ancestors of the node that we are visiting that have been marked. When we visit some bottom node u, we check whether $r = C(u) - N$ is at least k. If yes, we add the path-set consisting of u and its unmarked ancestors to $\mathcal{F}$. Note that, throughout the above process we maintain that if a node is marked, then all its (non-root) ancestors are also marked. Hence, we can efficiently find the r unmarked ancestors of u by following parent pointers. □

We now combine Lemmas 3.4.15 and 3.4.16 and Theorem 3.4.3 to get the following result.

Theorem 3.4.17. *Given a text T of length n and an internal dictionary $\mathcal{D}$ of size d for any $m \in [1, n]$ and any constant $\epsilon > 0$, we can construct, in $\mathcal{O}((nd \log n)/m + d)$ time, a data structure of size $\mathcal{O}((nd \log n)/m + d)$ that answers $\textsc{CountDistinct}(i, j)$ queries in $\mathcal{O}(m \log^{\epsilon} n + \log n)$ time.*

Proof. We first apply Lemma 3.4.16 for $k = \lceil d/m \rceil$. We then have a decomposition of $\mathcal{D}$ to a family $\mathcal{F}$ of at most m path-sets and a set $\mathcal{D}'$ with no path-set of size greater than $\lfloor d/m \rfloor$. We directly apply Lemma 3.4.15 for $\mathcal{D}'$. In order to handle path-sets, we build

the data structure of Theorem 3.4.3. Then, upon a $\textsc{CountDistinct}(i, j)$ query, for each path-set $\Pi \in \mathcal{F}$, we compute the longest pattern in Π that occurs in $T[i\,..\,j]$ using a BLCP query followed by a predecessor query in a structure that stores the lengths of the elements of Π, with the lexicographic rank in Π stored as satellite information. $\square$

Remark 3.4.18. Let us fix the query time to be $\mathcal{O}(m \log^{\epsilon} n)$ for $m = \Omega(\log n)$. Then, Theorem 3.4.17 outperforms Theorem 3.4.14 in terms of the required space in the case where $d = o(n/(m \log n))$. For example, for $m = d = n^{1/4}$, the data structure of Theorem 3.4.17 requires space $\tilde{\mathcal{O}}(n)$ while the one of Theorem 3.4.14 requires space $\tilde{\mathcal{O}}(n\sqrt{n})$.

We leave open the problem of whether an $\tilde{\mathcal{O}}(n + d)$-size data structure answering $\textsc{CountDistinct}(i, j)$ queries exactly in time $\tilde{\mathcal{O}}(1)$ exists.

3.5 Dynamic Dictionaries

In the Online Boolean Matrix-Vector Multiplication (OMv) problem, we are given as input an $n \times n$ boolean matrix M. Then, we are given in an online fashion a sequence of n vectors $r_1, \ldots, r_n$, each of size n. For each such vector r_i, we are required to output Mr_i before receiving r_{i+1}.

Conjecture 3.5.1 (OMv Conjecture [95])**.** *For any constant $\epsilon > 0$, there is no $\mathcal{O}(n^{3-\epsilon})$-time algorithm that solves OMv correctly with probability at least 2/3.*

We now state a restricted, but sufficient for our purposes, version of [95, Theorem 2.2].

Theorem 3.5.2 ([95])**.** *The OMv conjecture implies that there is no algorithm, for a fixed $\gamma > 0$, that given as input an $r_1 \times r_2$ matrix M, with $r_1 = \lfloor r_2^{\gamma} \rfloor$, preprocesses M in time polynomial in $r_1 + r_2$ and, then, presented with a vector v of size r_2, computes Mv in time $\mathcal{O}(r_2^{1+\gamma-\epsilon})$ for $\epsilon > 0$, and has error probability at most 1/3.*

We proceed to obtain a conditional lower bound for IDM in the case of a dynamic dictionary. This lower bound clearly carries over to the other considered problems.

Theorem 1.2.2. *The OMv conjecture implies that there is no algorithm that preprocesses T and $\mathcal{D}$ in time polynomial in n, performs insertions to $\mathcal{D}$ in time $\mathcal{O}(n^{\alpha})$, answers* $\text{EXISTS}(i, j)$ *queries in time $\mathcal{O}(n^{\beta})$, in an online manner, such that $\alpha + \beta = 1 - \epsilon$ for $\epsilon > 0$, and has error probability at most 1/3.*

Proof. Let us suppose that there is such an algorithm and set $\gamma = (\alpha + \epsilon/2)/(\beta + \epsilon/2)$, where $\epsilon = 1 - \alpha - \beta$. Given an $r_1 \times r_2$ matrix M satisfying $r_1 = \lfloor r_2^{\gamma} \rfloor$, we construct a text T of length $n = r_1 r_2 + r_2$ as follows. Let T' be a text created by concatenating the rows of M in increasing order. Then T is obtained by assigning to each non-zero element of T' the column index of the matrix entry it originates from, and appending one by one the integers in $[1, r_2]$ in increasing order. Formally, for $i \in [1, r_1 r_2]$, let $a[i] = \lceil i/r_2 \rceil$ and $b[i] = 1 + (i - 1) \bmod r_2$, and set $T[i] = b[i] \cdot M[a[i], b[i]]$; for $i \in [r_1 r_2 + 1, r_1 r_2 + r_2]$, set $T[i] = i - r_1 r_2$. (We append these letters in order to ensure that the dictionary that we construct below is internal, and that we can specify in $\mathcal{O}(1)$ time each of the inserted/deleted elements by an occurrence of them in T.)

We compute Mv as follows. We add the indices of v's non-zero entries into an initially empty dictionary. We then perform queries $\text{EXISTS}(1 + (t - 1)r_2, tr_2)$ for $t \in [1, r_1]$. The answer to the query $\text{EXISTS}(1 + (t - 1)r_2, tr_2)$ is equal to the boolean dot product of the t-th row of M with v. We thus obtain Mv, with each entry correct with probability at least 2/3. We can guarantee that the whole vector Mv is correct with probability at least $1 - n^{-\Omega(1)} \geq 2/3$ by maintaining $\Theta(\log n)$ independent instances of the algorithm and taking the dominant answer to each EXISTS query.

In total, we perform $\tilde{\mathcal{O}}(r_2)$ insertions to $\mathcal{D}$ and $\tilde{\mathcal{O}}(r_1)$ EXISTS queries. Thus, the total time required is $\tilde{\mathcal{O}}(r_2 n^{\alpha} + r_1 n^{\beta}) = \tilde{\mathcal{O}}(n^{\beta+\epsilon/2} n^{\alpha} + n^{\alpha+\epsilon/2} n^{\beta}) = \tilde{\mathcal{O}}(n^{1-\epsilon/2}) = \mathcal{O}(r_2^{1+\gamma-\epsilon'})$ for $\epsilon' > 0$. Conjecture 3.5.1 would be disproved due to Theorem 3.5.2. $\square$

Example 3.5.3. For the matrix

$$M = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix},$$

we construct the text $T = 1\,0\,3\,0\,0\,0\,3\,4\,0\,2\,0\,4\,1\,2\,3\,4$. For the vector $v = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}^T$, the dictionary is $\mathcal{D} = \{1, 2\}$. The answers to $\textsc{Exists}(1,4)$, $\textsc{Exists}(5,8)$, $\textsc{Exists}(9,12)$ queries are **true**, **false**, **true**, respectively, which corresponds to $Mv = \begin{bmatrix} 1 & 0 & 1 \end{bmatrix}^T$.

Remark 3.5.4. The proof of the above theorem requires a large alphabet. Let us describe a straightforward modification to this proof that yields the same lower bound for ternary alphabets. We add a $ between every pair of consecutive letters in T and then replace each integer in T with its binary representation. Similarly, the patterns we add to or remove from $\mathcal{D}$ are the same integers as above, but represented in binary. The obtained text is now longer than the one in the above proof by a multiplicative $\mathcal{O}(\log r_2)$ factor. This factor is hidden by the $\tilde{\mathcal{O}}(\cdot)$ notation in the analysis.

In the remainder of this section we mainly focus on providing algorithms that essentially match this lower bound. We extensively use the data structure underlying Proposition 2.6.2, restated here for convenience.

Proposition 2.6.2 (Based on [127]). *Given a string of length n, we can construct in $\mathcal{O}(n \log n)$ time an $\mathcal{O}(n \log n)$-size data structure that answers the decision, counting, and reporting versions of internal pattern matching queries in time $\mathcal{O}(\log^2 n + |output|)$.*

Let us denote the dictionary we start with by $\mathcal{D}^0$. Further, let $u_1, u_2, \ldots$ be the sequence of dictionary updates and $\mathcal{D}^r$ be the dictionary after update u_r. Each update is an insertion or a deletion of a pattern in $\mathcal{D}$. We first discuss how to answer $\textsc{ReportDistinct}$ queries.

ReportDistinct(i, j). We maintain the invariant that after update u_t we have access to the static data structure of Section 3.2, encapsulated in Theorem 3.2.9, for answering

ReportDistinct queries in T with respect to dictionary $\mathcal{D}^r$, for some $r = t - \mathcal{O}(m)$; note that m will depend on $n+d$. This can be achieved by rebuilding the data structure of Section 3.2 every m updates in $\mathcal{O}((n+d)\log n)$ time, which amortises to $\mathcal{O}((n+d)\log n/m)$ time per update. The time complexity can be made worst-case by applying the time-slicing technique (cf. the proof of Lemma 2.8.1). We also store updates $u_{r+1}, \dots, u_t$ (or the differences between $\mathcal{D}^r$ and $\mathcal{D}^t$).

To answer a ReportDistinct query, we:

1. use the static data structure to answer the ReportDistinct query for $\mathcal{D}^r$;
2. filter out the $\mathcal{O}(m)$ reported patterns that are in $\mathcal{D}^r \setminus \mathcal{D}^t$;
3. search for the $\mathcal{O}(m)$ patterns in $\mathcal{D}^t \setminus \mathcal{D}^r$ individually in $\mathcal{O}(\log^2 n)$ time per pattern by performing internal pattern matching queries, employing Theorem 2.6.2.

Each query thus requires time $\mathcal{O}(\log n + m\log^2 n + |output|)$. We obtain the following proposition.

Proposition 3.5.5. *Given a text T of length n, an internal dictionary $\mathcal{D}$ of size d, and a parameter $m \in [1, n+d]$, after an $\mathcal{O}((n+d)\log n)$-time preprocessing, we can support* ReportDistinct(i, j) *queries in $\mathcal{O}(m\log^2 n + |output|)$ time and process updates to the dictionary in $\mathcal{O}((n+d)\log n/m)$ time, using space $\mathcal{O}(n\log n + d)$, provided that the size of the dictionary remains $\mathcal{O}(d)$.*

We next show how to attain $\tilde{\mathcal{O}}(n^{\alpha})$ update time and $\tilde{\mathcal{O}}(n^{1-\alpha} + |output|)$ query time for any $0 < \alpha < 1$. In other words, we show how to avoid the direct dependency on the size of the dictionary.

We store $\mathcal{D}$ in an array D of size n, which consists in collections of total size d. $D[p]$ stores the elements of $\mathcal{D}$ whose leftmost occurrence in T is at position p in a min heap with respect to their lengths. In fact, as all elements stored in $D[p]$ are prefixes of $T[p\mathinner{.\,.} n]$, it suffices to store the length of each element. We can find the desired position p for a pattern P in $\mathcal{O}(\log\log n)$ time by locating its locus on $\mathcal{T}(T)$ using a weighted ancestor query; we can have precomputed the leftmost occurrence of the path-label of each explicit node of $\mathcal{T}(T)$ in a DFS traversal. We can initialise D in $\mathcal{O}(n + |\mathcal{D}^0|)$ time

by answering all weighted ancestor queries as a batch [113].

The dictionary $\mathcal{D}' = \{\min D[p] : 1 \le p \le n\}$, where $\min D[p]$ is the shortest element stored in $D[p]$, is of size $\mathcal{O}(n)$. An insertion in $\mathcal{D}$ corresponds to a possible insertion followed by a possible deletion in $\mathcal{D}'$, while a deletion in $\mathcal{D}$ corresponds to a possible deletion in $\mathcal{D}'$, followed by an insertion if the collection in D where the deletion occurs is non-empty. We observe that if some $P \in \mathcal{D} \setminus \mathcal{D}'$ occurs in $T[i\,..\,j]$, then the shortest element in the collection in which P belongs also occurs in $T[i\,..\,j]$.

We use the solution for $\text{ReportDistinct}(i,j)$ from Proposition 3.5.5 for $\mathcal{D}'$. We then iterate over the elements of each collection from which an element has been reported in the order of increasing length, while they occur in $T[i\,..\,j]$; we check whether this is the case using Theorem 2.6.2. The update time now becomes $\mathcal{O}(n \log n/m)$, while the query time becomes $\mathcal{O}((m + |output|) \cdot \log^2 n)$, for any $m \in [1,n]$.

Report(i,j)**.** We first perform a $\text{ReportDistinct}(i,j)$ query and then find all occurrences of each returned pattern in $T[i\,..\,j]$ in time $\mathcal{O}(\log^2 n + |output|)$ using Theorem 2.6.2. The complexities are identical with those for $\text{ReportDistinct}(i,j)$ queries.

Theorem 3.5.6. *Given a text T of length n, an internal dictionary $\mathcal{D}^0$, and a parameter $m \in [1,n]$, after an $\mathcal{O}(n \log^{3/2} n + |\mathcal{D}^0|)$-time preprocessing, we can answer* $\text{Report}(i,j)$ *and* $\text{ReportDistinct}(i,j)$ *queries in $\mathcal{O}((m + |output|) \cdot \log^2 n)$ time and process updates to the dictionary in $\mathcal{O}(n \log n/m)$ time, using space $\mathcal{O}(n \log n + d)$, where d is the size of the dictionary.*

Exists(i,j)**.** We again use $\mathcal{D}'$. We first use the static version of $\text{Count}(i,j)$ for $\mathcal{D}'$, presented in Section 3.3, encapsulated in Theorem 3.3.1, and then the counting version of internal pattern matching for removed/added patterns using Theorem 2.6.2, incrementing/decrementing the counter appropriately. We rebuild the data structure underlying Theorem 3.3.1 after every m updates. For any $m \in [1,n]$, we can thus achieve preprocessing time $\mathcal{O}(n \log n/ \log\log n + |\mathcal{D}^0| \log^{3/2} n)$, space $\mathcal{O}(n + d \log n)$, query time $\mathcal{O}(m \log^2 n)$, and update time $\mathcal{O}(n \log^{3/2} n/m)$.

Theorem 3.5.7. *Given a text T of length n, an internal dictionary $\mathcal{D}^0$, and a parameter $m \in [1, n]$, after an $\mathcal{O}(n \log n / \log \log n + |\mathcal{D}^0| \log^{3/2} n)$-time preprocessing, we can answer* EXISTS(i, j) *queries in $\mathcal{O}(m \log^2 n)$ time and process updates to the dictionary in $\mathcal{O}(n \log^{3/2} n/m)$ time, using space $\mathcal{O}(n \log n + d)$, where d is the size of the dictionary.*

Count(i, j). We first build the data structure of Section 3.3 for COUNT(i, j) queries for dictionary $\mathcal{D}^0$, with each 2D range stabbing data structure implemented with a 2D range tree (i.e. using Theorem 2.5.3), so that we can efficiently insert or delete rectangles. This change affects the complexities of Lemma 3.3.4 in a straightforward manner. For the COUNT(i, j) problem, the preprocessing time becomes $\mathcal{O}((n + |\mathcal{D}^0|) \log^2 n)$, the space used is $\mathcal{O}(n + d \log^2 n)$, while the query time is $\mathcal{O}(\log^3 n)$. For the subsequent m updates, we answer COUNT(i, j) queries, using this data structure and treating individually the added/removed patterns using Theorem 2.6.2. Queries are thus answered in $\mathcal{O}(m \log^2 n + \log^3 n)$ time.

After every m updates, we update our data structure to refer to the current dictionary as follows. (We focus on $\mathcal{D}^0$ and $\mathcal{D}^m$ for notational simplicity.) We update the counts of occurrences for all nodes of PT by computing the counts for the set of added and the set of removed patterns in $\mathcal{O}(n \log n / \log \log n + m \log^{3/2} n)$ time and updating the previously stored counts accordingly.

As for BREAKPOINTS-ANCHOR IDM, we have to do something smarter than simply recomputing the whole data structure from scratch, as we do not want to spend $\Omega(d)$ time. At preprocessing, we set our grid $\mathcal{G}$ to be of size $K \times K$ for $K = \mathcal{O}(n^2)$ and identify x-coordinate i with the i-th smallest element of the set $W = \{Ux : U \text{ a substring of } T \text{ and } x \in \{\$, \#\}\}$. (Similarly for y-coordinates and T^R.)

We can preprocess the suffix tree $\mathcal{T}(T)$ in $\mathcal{O}(n)$ time so that the lexicographic rank of a given $T[a \mathinner{.\,.} b]\$$ or $T[a \mathinner{.\,.} b]\#$ in W can be computed in $\mathcal{O}(\log \log n)$ time. Let us assume that $\mathcal{T}(T)$ has been built for T, without \$ appended to it. We make a DFS traversal of $\mathcal{T}(T)$, maintaining a global counter cr, which is initialised to zero at the root. The DFS visits the children of a node in a left-to-right order. When traversing an edge, we

increment cr by the size of the path-label of this edge. When an explicit node v is visited for the *first* time we set the rank of $\mathcal{L}(v)\$$ equal to cr; if v is a leaf then cr is incremented by one. The rank of $\mathcal{L}(v)\#$ is set to cr when v is visited for the *last* time. Let q be the locus of $T[a\mathinner{.\,.} b]$ in $\mathcal{T}(T)$, which can be computed in $\mathcal{O}(\log\log n)$ time using a weighted ancestor query. If q is an explicit node, the ranks of $T[a\mathinner{.\,.} b]\$$ and $T[a\mathinner{.\,.} b]\#$ are already stored at q. Otherwise, these ranks can be inferred from the ranks of $\mathcal{L}(v)\$$ and $\mathcal{L}(v)\#$ stored at the nearest explicit descendant v of q by, respectively, subtracting and adding the distance between v and q.

Thus, instead of explicitly building trees W and W^R as in the proof of Lemma 3.3.3, we use $\mathcal{T}(T)$ and $\mathcal{T}(T^R)$ and maintain rectangles in $\mathcal{G}$. After m updates, we remove (resp. add) the $\mathcal{O}(m\log n)$ rectangles corresponding to patterns in $\mathcal{D}^0\setminus D^m$ (resp. $\mathcal{D}^m\setminus D^0$). Since we are using 2D range trees, this can be done in $\mathcal{O}(m\log^3 n)$ time.

To wrap up, each query is answered in $\mathcal{O}(m\log^2 n+\log^3 n)$ time and each update is processed in $\mathcal{O}(n\log n/(m\log\log n)+\log^3 n)$ amortised time. We can deamortise this time complexity using the time-slicing technique (cf. the proof of Lemma 2.8.1). See Theorem 3.5.8 for a formal statement encapsulating the above discussion.

2-approximate CountDistinct(i,j). We build our static data structure for computing a 2-approximation of CountDistinct(i,j), specified in Theorem 3.4.13, for dictionary $\mathcal{D}^0$, using 2D range trees for all 2D range counting/stabbing data structures. The preprocessing time becomes $\mathcal{O}((n+|\mathcal{D}^0|)\log^2 n)$, the space used is $\mathcal{O}(n+d\log^2 n)$, while the query time is $\mathcal{O}(\log^3 n)$. For the subsequent m updates to the dictionary we answer CountDistinct(i,j) queries 2-approximately in $\mathcal{O}(m\log^2 n+\log^3 n)$ time as follows. We first ask a CountDistinct(i,j) for $\mathcal{D}^0$, and then rely on Proposition 2.6.2. We use an internal pattern matching query for each pattern in $\mathcal{D}^m\setminus D^0$. For each pattern in $\mathcal{D}^0\setminus D^m$, we need to check whether it has been counted once or twice by the static data structure for $\mathcal{D}^0$: for this, it suffices to query whether such a pattern occurs in $T[i\mathinner{.\,.} j]$ and in the two relevant basic factors.

We update our data structure after every m updates to the dictionary as follows.

- We adjust $\textsc{CountDistinct}(i, j)$ for each basic factor in $\mathcal{O}(n \log^{1+\epsilon} n + m)$ time, for any $\epsilon > 0$, by counting distinct patterns of $\mathcal{D}^m \setminus D^0$ and $\mathcal{D}^0 \setminus D^m$ in each of them, as in the preprocessing of Theorem 3.4.13.
- We update our collections of points on grids $\mathcal{G}_{L,r}$ in $\mathcal{O}(m \log^2 n)$ time using 2D range trees. As for the values $\sum_{i=1}^{r} |\mathcal{D}^p_{L,i}|$, we use an augmented balanced binary search tree, which can be updated and queried in $\mathcal{O}(\log n)$ time; this time is dominated by the other parts of the data structure.
- Finally, we update our static data structure for $\textsc{Count}(i, j)$ as above.

Theorem 3.5.8. *Given a text T of length n, an internal dictionary $\mathcal{D}^0$, and a parameter $m \in [1, n]$, after an $\mathcal{O}((n + |\mathcal{D}^0|) \log^2 n)$-time preprocessing, we can answer* $\textsc{Count}(i, j)$ *queries and 2-approximate* $\textsc{CountDistinct}(i, j)$ *queries in $\mathcal{O}(m \log^2 n + \log^3 n)$ time and process updates to the dictionary in $\mathcal{O}(n \log n/(m \log \log n) + \log^3 n)$ time, using space $\mathcal{O}(n + d \log^2 n)$, where d is the size of the dictionary.*

3.6 Internal Counting of Distinct Squares

We now consider the $\textsc{CountDistinct}$ problem in the special case where the dictionary $\mathcal{D}$ is the set of all squares in T. By the following theorem, $d = \mathcal{O}(n)$ and $\mathcal{D}$ can be computed in $\mathcal{O}(n)$ time.

Theorem 3.6.1 ([59, 63, 72, 92])**.** *A string T of length n contains $\mathcal{O}(n)$ distinct square factors and they can all be computed in $\mathcal{O}(n)$ time.*

Similarly to before, we say that an occurrence of a square U^2 is *induced* by a run R if it is contained in R and the shortest periods of U and R are the same. Every occurrence of a square is induced by exactly one run.

The number of occurrences of squares can be quadratic in n, but we can construct a much smaller $\mathcal{O}(n \log n)$-size subset of these occurrences (called *boundary occurrences*) that, from the point of view of $\textsc{CountDistinct}$ queries, gives almost the same answers. This is the main trick in this section. Distinct squares with a boundary occurrence in

a given fragment can be counted in $\mathcal{O}(\log n)$ time due to Lemma 3.4.2. The remaining squares can be counted based on their structure: we show that they are all generated by the same run.

We need the following lemma (note that it is false for the set of *all* runs; see [84]).

Lemma 3.6.2. *The sum of the lengths of all highly periodic runs is $\mathcal{O}(n \log n)$.*

Proof. We will prove that each position in T is contained in $\mathcal{O}(\log n)$ highly periodic runs. Let us consider all highly periodic runs R containing some position i, such that $m \le \mathsf{per}(R) < \frac{3}{2}m$ for some even integer m. Suppose for the sake of contradiction that there are at least 5 such runs. Note that each such run fully contains one of the fragments $T[i-3m+1+t\,..\,i+t]$ for $t \in \{0, m, 2m, 3m\}$. By the pigeonhole principle, one of these four fragments is contained in at least two runs, say R_1 and R_2. In particular, the overlap of these runs is at least $3m \ge \mathsf{per}(R_1) + \mathsf{per}(R_2)$, which is a contradiction by the periodicity lemma (Lemma 2.3.1). ☐

We define a family of occurrences $\mathcal{B} = \{B_1, \dots, B_d\}$ such that, for each square U_i^2, the set B_i contains the leftmost and the rightmost occurrence of U_i^2 in every run. We call these *boundary occurrences*. Boundary occurrences of squares have the following property.

Lemma 3.6.3. $\|\mathcal{B}\| = \mathcal{O}(n \log n)$ *and $\mathcal{B}$ can be computed in $\mathcal{O}(n \log n)$ time.*

Proof. Let us define the *root* of a square U^2 to be U. A square is primitively rooted if its root is a primitive string. Let *p-squares* be primitively rooted squares, *np-squares* be the remaining ones. The number of occurrences of p-squares in a string of length n is $\mathcal{O}(n \log n)$ and they can all be computed in $\mathcal{O}(n \log n)$ time; see [54, 155].

We now proceed to np-squares. Note that for any highly periodic run R, the leftmost occurrence of each np-square induced by R starts in one of the first $\mathsf{per}(R)$ positions of R; a symmetric property holds for rightmost occurrences and last $\mathsf{per}(R)$ positions. In addition, it can be readily verified that such a position is the starting (resp. ending) position of at most $\mathsf{exp}(R)$ squares induced by R. It thus suffices to bound the sum

of $\mathsf{exp}(R) \cdot \mathsf{per}(R)$ over all highly periodic runs R. The fact that $\mathsf{exp}(R) \cdot \mathsf{per}(R) = |R|$ concludes the proof of the combinatorial part by Lemma 3.6.2.

For the algorithmic part, it suffices to iterate over the $\mathcal{O}(n)$ runs of T. □

Lemma 3.6.4. *If $T[i\mathinner{.\,.} j]$ is aperiodic,* $\textsc{CountDistinct}(i,j) = \textsc{CountDistinct}_{\mathcal{B}}(i,j)$.

Proof. Let us consider an occurrence of a square U^2 inside $T[i\mathinner{.\,.} j]$. Let R be the run that induces this occurrence. By the assumption of the lemma, R does not contain $T[i\mathinner{.\,.} j]$. Then at least one of the boundary occurrences of U^2 in R is contained in $T[i\mathinner{.\,.} j]$. □

For a periodic fragment F of T, by $\mathit{RunSquares}(F)$ we denote the number of distinct squares that are induced by F (being a run if interpreted as a standalone string). The value $\mathit{RunSquares}(F)$ can be computed in $\mathcal{O}(1)$ time, as it was shown in e.g. [59].

Let F_1 be a prefix and F_2 be a suffix of a periodic fragment F, such that each of F_1 and F_2 is of length at most $\mathsf{per}(F)$, and hence they are disjoint. By $\mathit{BSq}(F, F_1, F_2)$ ("bounded squares") we denote the number of distinct squares induced by F which have an occurrence starting in F_1 or ending in F_2.

Lemma 3.6.5. *Given $\mathsf{per}(F)$, the $\mathit{BSq}(F, F_1, F_2)$ queries can be answered in $\mathcal{O}(1)$ time.*

Proof. We are to count distinct squares induced by F that start in F_1 or end in F_2.

We introduce an easier version of BSq queries. Let $\mathit{BSq}'(F, F_1) = \mathit{BSq}(F, F_1, \varepsilon)$ be the number of squares induced by F which start in its prefix F_1 of length at most $p := \mathsf{per}(F)$.

Reduction of BSq to BSq'. First, observe that the set of squares induced by F starting at some position $q \in [1, p]$ and the set of squares induced by F ending at some position $q' \in [|F| - p + 1, |F|]$ are equal if $q \equiv q' + 1 \pmod p$ and disjoint otherwise. Also note that $F_2 = UV$ for some prefix V and some suffix U of $F[p]F[1\mathinner{.\,.} p-1]$; we consider this rotation of $F[1\mathinner{.\,.} p]$ to offset the $+1$ factor in the above modular equation. Let $|U| = a$ and $|V| = b$.

Then, by the aforementioned observation, we are to count distinct squares that start in some position in the set $[1, |F_1|] \cup [1, b] \cup [p - a + 1, p]$; see Figure 3.7.

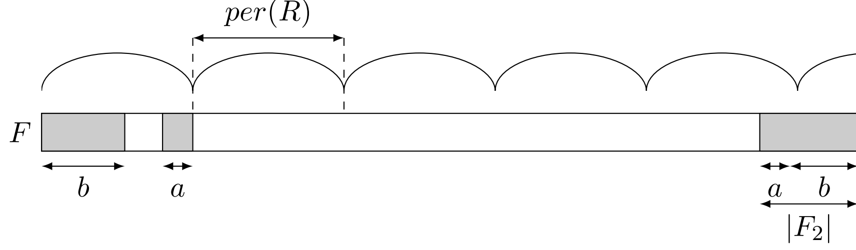


Figure 3.7: Reduction of BSq to BSq'; the case where $|F_1| \le b$.

Hence the computation of $BSq(F, F_1, F_2)$ is reduced to at most two instances of the special case when F_2 is the empty string.

Computation of $BSq'(F, F_1)$. The number of squares induced by F starting at $F[i]$ is $\lfloor(|F| - i + 1)/(2p)\rfloor$. Consequently, $BSq'(F, F_1) = \sum_{i=1}^{|F_1|} \lfloor(|F| - i + 1)/(2p)\rfloor = |F_1| \cdot t - \max\{0, |F_1| - k - 1\}$, where $t = \lfloor|F|/(2p)\rfloor$ and $k = |F| \bmod (2p)$. □

Lemma 3.6.6. *Assume that $F = T[i \mathinner{.\,.} j]$ is periodic and $R = T[a \mathinner{.\,.} b] = \mathsf{run}(T[i \mathinner{.\,.} j])$. Let $F_1 = T[i \mathinner{.\,.} a + p - 1]$ and $F_2 = T[b - p + 1 \mathinner{.\,.} j]$, where $\mathsf{per}(R) = p$. Then:*

$$\textsc{CountDistinct}(i, j) = \textsc{CountDistinct}_{\mathcal{B}}(i, j) + \mathit{RunSquares}(F) - \mathit{BSq}(F, F_1, F_2). \tag{3.1}$$

Proof. In the sum $\textsc{CountDistinct}_{\mathcal{B}}(i, j) + \mathit{RunSquares}(F)$, all squares are counted once except for squares whose boundary occurrences are induced by R, which are counted twice. They are exactly counted in the term $BSq(F, F_1, F_2)$; see Figure 3.8. □

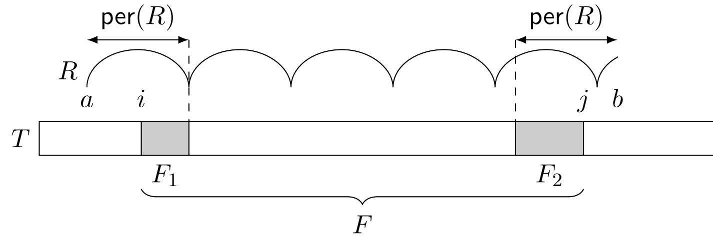


Figure 3.8: The setting in Lemma 3.6.6. F_1 is empty if $i \ge a + \mathsf{per}(R)$; similarly for F_2.

Theorem 3.6.7. *If $\mathcal{D}$ is the set of all square factors of T, then* $\textsc{CountDistinct}(i, j)$ *queries can be answered in $\mathcal{O}(\log n)$ time using a data structure of size $\mathcal{O}(n \log^2 n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We precompute the set $\mathcal{B}$ in $\mathcal{O}(n \log n)$ time using Lemma 3.6.3 and perform an $\mathcal{O}(n \log^2 n)$-time and space preprocessing for $\textsc{CountDistinct}_{\mathcal{B}}(i, j)$ queries, using Lemma 3.4.2.

In order to answer a $\textsc{CountDistinct}(i, j)$ query, first we ask a 2-period query, employing Theorem 2.6.4, to check if $T[i \mathinner{.\,.} j]$ is periodic, and, if so, we compute $\mathsf{run}(T[i \mathinner{.\,.} j])$ by employing Theorem 2.6.5; this requires $\mathcal{O}(1)$ time.

We compute $\textsc{CountDistinct}_{\mathcal{B}}(i, j)$ which takes $\mathcal{O}(\log n)$ time due to Lemma 3.4.2. If $T[i \mathinner{.\,.} j]$ is aperiodic, then it is the final result due to Lemma 3.6.4.

Otherwise, $T[i \mathinner{.\,.} j]$ is periodic. Let F, F_1, F_2 be as in Lemma 3.6.6. We answer $\mathit{RunSquares}(F)$ and $\mathit{BSq}(F, F_1, F_2)$ queries in $\mathcal{O}(1)$ time using the algorithms from [59] and Lemma 3.6.5, respectively. Finally, $\textsc{CountDistinct}(i, j)$ is computed using Equation (3.1). $\square$

Chapter 4

Internal Pattern Matching in a Dynamic Collection of Strings

In this chapter, we show how to extend the data structure of Gawrychowski et al. [82] for the maintenance of a dynamic collection of strings (cf. Theorem 1.3.1) to also support internal pattern matching queries. Formally, we obtain the following result.

Theorem 1.3.2. *A collection $\mathcal{X}$ of non-empty strings of total length at most N can be dynamically maintained with update operations $\mathsf{makestring}(U)$, $\mathsf{concat}(U,V)$, $\mathsf{split}(U,i)$ requiring time $\mathcal{O}(\log N + |U|)$, $\mathcal{O}(\log N)$, and $\mathcal{O}(\log N)$, respectively, so that the occurrences of a string $P \in \mathcal{X}$ in a string $T \in \mathcal{X}$ can be computed in time $\mathcal{O}(|T|/|P| \cdot \log^2 N)$. All running times hold w.h.p.*

Internal, here, is meant in the sense that both P and T must be elements of $\mathcal{X}$. Notice, however, that P and T can be arbitrary substrings of strings in the collection: we can add these substrings to the collection using a constant number of split operations.

4.1 The Algorithm

Recall that the data structure underlying Theorem 1.3.1 for the maintenance of a dynamic collection, maintains a parse tree $\mathsf{PT}[S]$ (stemming from recompression) for

each string in the collection S, and that the popped sequence of a fragment U of S is the sequence labeling a certain layer of nodes of $\mathsf{PT}[S]$, whose values constitute U. Let us restate Theorem 2.7.1, with $\log n$ factors replaced by $\log N$ ones with high probability; these factors stem from the heights of the respective parse trees.

Theorem 4.1.1 ([98])**.** *If two fragments of strings in $\mathcal{X}$ are equal, then their popped sequences are equal as well. Moreover, w.h.p., each popped sequence consists of $\mathcal{O}(\log N)$ RLE runs (maximal powers of a single symbol) and can be constructed in $\mathcal{O}(\log N)$ time. The nodes corresponding to symbols in an RLE run share a single parent. Furthermore, the popped sequence consists of a single symbol only for fragments of length 1.*

Let us also recall that if $F_1^{p_1} \cdots F_t^{p_t}$ is the run-length encoding of the popped sequence of a substring U of some string in $\mathcal{X}$, and

$$L(U) = \{|\mathsf{gen}(F_1)|, |\mathsf{gen}(F_1^{p_1} \cdots F_{t-1}^{p_{t-1}} F_t^{p_t - 1})|\} \bigcup \{|\mathsf{gen}(F_1^{p_1} \cdots F_i^{p_i})| : i \in [1, t-1]\},$$

then the following lemma holds for each $S \in \mathcal{X}$, restated here for convenience.

Lemma 2.7.2. *Let v denote a non-leaf node of a parse tree $\mathsf{PT}[S]$ stemming from recompression and let $S[a\mathinner{.\,.} b]$ denote an occurrence of a string U contained in $\mathsf{val}(v)$, but not contained in $\mathsf{val}(u)$ for any child u of v. If $S[a\mathinner{.\,.} c]$ is the longest prefix of $S[a\mathinner{.\,.} b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\mathinner{.\,.} c]| \in L(U)$. Symmetrically, if $S[c'+1\mathinner{.\,.} b]$ is the longest suffix of $S[a\mathinner{.\,.} b]$ contained in $\mathsf{val}(u)$ for a child u of v, then $|S[a\mathinner{.\,.} c']| \in L(U)$.*

We are now ready to prove the main technical lemma of this chapter. In this lemma, we make the assumption that $|T| < 2|P|$, which we then lift.

Lemma 4.1.2. *Given strings P and T from a collection $\mathcal{X}$, maintained as in Theorem 1.3.1, with $|T| < 2|P|$, we can compute $\mathsf{IPM}(P, T)$ in time $\mathcal{O}(\log^2 N)$ w.h.p. The positions of T where P occurs are returned as a (possibly empty) arithmetic progression with difference $\mathsf{per}(P)$.*

Proof. Let $n := |T|$ and $m := |P|$. We can assume that $n, m > 1$; otherwise it suffices to perform a constant number of letter comparisons, which can be done in $\mathcal{O}(1)$ time.

We first compute the popped sequence of P and $L(P)$ in $\mathcal{O}(\log N)$ time using Theorem 4.1.1. In addition, we perform $\mathcal{O}(\log N)$ split operations, in a total of $\mathcal{O}(\log^2 N)$ time in order to add $P[1\..q-1]$ and $P[q\..m]$ to $\mathcal{X}$, for all $q \in L(P)$.

Let v denote the root of the parse tree $\mathsf{PT}[T]$. Our first aim is to compute all occurrences of P in T that are not contained in $\mathsf{val}(u)$ for any child u of v; we then appropriately recurse on the children of v that may contain sought occurrences.

Let us first analyse the case where the label of v is a concatenation symbol $A \rightarrow BC$. Let us denote by v_ℓ the left child of v and by v_r the right child of v. Further, let $T_\ell = \mathsf{val}(v_\ell)$ and $T_r = \mathsf{val}(v_r)$. Suppose that there is a fragment $U = T[a\..b]$ of $\mathsf{val}(v)$ that equals P and overlaps with both $\mathsf{val}(v_\ell)$ and $\mathsf{val}(v_r)$. The fragment U can then be naturally decomposed into a non-empty suffix U_ℓ of T_ℓ and a non-empty prefix U_r of T_r. Lemma 2.7.2 implies that $|U_\ell| \in L(P)$. It thus suffices to check for each $q \in L(P)$ whether $P[1\..q-1]$ is a suffix of T_ℓ and $P[q\..m]$ is a prefix of T_r. We have $|L(P)| = \mathcal{O}(\log N)$ choices for q. For each of them we can perform the check using operations $\mathsf{LCP}^{\mathrm{R}}(P[1\..q-1], T_\ell)$ and $\mathsf{LCP}(P[q\..m], T_r)$. These operations can be performed in $\mathcal{O}(\log N)$ total time, by first adding T_ℓ and T_r to $\mathcal{X}$ using a split operation, and then answering each LCE query in $\mathcal{O}(1)$ time. If the value of each child of v is of length less than m we terminate the algorithm—up to some postprocessing to be discussed below. Otherwise, at most one of v's children has value of length at least m. In that case, we recurse on this child.

We now consider the case where the label of v is a power symbol $A \rightarrow B^p$ and denote the children of v in the left-to-right order by $v_1, \ldots, v_p$. If $p = 2$, then we can process v as in the previous case. We can thus assume that $p \geq 3$. In that case, for all i, we have $\mathsf{val}(v_i) < m$, and hence no occurrence of P in T can be completely contained in $\mathsf{val}(v_i)$, for any i. We set $T_\ell := \mathsf{val}(v_1)$ and $T_r := \mathsf{val}(v_2) \cdots \mathsf{val}(v_p)$. Using a single split operation, $|L(P)|$ many LCP operations and $|L(P)|$ many $\mathsf{LCP}^{\mathrm{R}}$ operations, we can compute, in $\mathcal{O}(\log N)$ time, the set Y of occurrences of P in T which can be decomposed into a prefix U_ℓ that is a suffix of T_ℓ and a suffix U_r that is a prefix of T_r. Then, by the periodicity

of $\mathsf{val}(v) = \mathsf{gen}(B)^p$, the desired set of occurrences is

$$Z := \{i + j \cdot |\mathsf{gen}(B)| : i \in Y, j \in [0, p-1]\} \cap [1, |\mathsf{val}(v)| - m + 1].$$

We represent Z by $\mathcal{O}(\log N)$ arithmetic progressions. As the value of each child of v has length less than m, we then proceed to the postprocessing step.

The overall time required by the procedure described above is $\mathcal{O}(\log^2 N)$, since the depth of $\mathsf{PT}[T]$ is $\mathcal{O}(\log N)$ w.h.p., we process each node in $\mathcal{O}(\log N)$ time, and all processed nodes lie in single root-to-leaf path.

In the end, the occurrences of P in T are represented by $\mathcal{O}(\log N)$ arithmetic progressions and $\mathcal{O}(\log^2 N)$ single occurrences. We postprocess this representation in $\mathcal{O}(\log^2 N)$ time, in order to represent all occurrences by a single arithmetic progression with difference $\mathsf{per}(P)$. In particular, we compute the first, second, and last positions a, b, and c of T, respectively, where P occurs. Then, we return $\{a + i \cdot d : i \in [0, (c-a)/d]\}$ for $d = b - a = \mathsf{per}(P)$. □

In order to generalise our result for the case that $|T| \geq 2|P|$, it suffices to do the following standard trick. Given arbitrary T and P of lengths n and m, respectively, we apply Lemma 4.1.2 to find the occurrences of P in each of the following $\mathcal{O}(n/m)$ strings: $T[1\mathinner{.\,.} 2m-1], T[m\mathinner{.\,.} 3m-1], \ldots, T[(\lfloor n/m \rfloor - 1) \cdot m \mathinner{.\,.} n-1]$. This concludes the proof of Theorem 1.3.2.

We conclude this chapter by discussing some applications of IPM queries. The following reductions to LCP and IPM queries were (implicitly) shown in [115, 112].

- A *cyclic equivalence query* takes as input two equal-length substrings U and V of a text, and returns all rotations of U that are equal to V. Any cyclic equivalence query reduces to $\mathcal{O}(1)$ LCP queries and $\mathcal{O}(1)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = \mathcal{O}(1)$.

- A *period query* takes as input a substring U of a text, and returns all periods of U. Such a period query reduces to $\mathcal{O}(\log |U|)$ LCP queries and $\mathcal{O}(\log |U|)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = \mathcal{O}(1)$.

- A *2-period* query takes as input a substring U of a text, checks if U is periodic and, if so, it also returns U's period. Such a query reduces to $\mathcal{O}(1)$ LCP queries and $\mathcal{O}(1)$ $\mathsf{IPM}(P, T)$ queries with $|T|/|P| = \mathcal{O}(1)$.

Thus, in the setting of Theorem 1.3.2, cyclic equivalence queries and 2-period queries can be answered in time $\mathcal{O}(\log^2 N)$ w.h.p., while period queries can be answered in time $\mathcal{O}(\log^3 N)$ w.h.p.

Remark 4.1.3. Recall that, as discussed in Section 2.7, a straight-line program can be "recompressed" in-place to an RLSLP with the desired properties. An adaptation of the algorithm underlying Lemma 4.1.2 can be employed for efficiently answering internal pattern matching queries in straight-line programs; see [47, 108].

Chapter 5

Dynamic Longest Square Substring

In this chapter we present an algorithm that maintains the longest square of a dynamic string S, of length n, that undergoes substitution operations in time $\mathcal{O}(\log^3 n)$ per update. For simplicity, we assume—without loss of generality—that n is a power of 2.

5.1 Strategy

We first make the following simple, but crucial observation.

Observation 5.1.1. *In a square-substring UU of S, with $|UU| \geq 4m$ for some positive integer m, the first occurrence of U contains $S[i \mathinner{.\,.} i+m-1]$ for some $i = 1 \pmod m$.*

For simplicity, we ignore squares of length smaller than 8; the created/destroyed such squares can be trivially recomputed from scratch in $\mathcal{O}(1)$ time per update. This allows us to rely on Observation 5.1.1. The idea is to use fragments of the form $S[i{\cdot}2^j{+}1 \mathinner{.\,.} (i{+}1){\cdot}2^j]$ for all i, j as anchors. Then, Observation 5.1.1 guarantees that for every square UU, the first implied occurrence of U contains an anchor whose length is in $[|U|/4, |U|)$. In light of this, we will do the following for each anchor:

- We will maintain all of its occurrences in a fragment starting at the same position as the anchor, but longer than the anchor by a multiplicative constant factor.
- For each such occurrence V we will check whether there is any square UU that "aligns" the anchor and V in the two occurrences of U implied by UU. "Aligns", here, means that the starting position of the anchor is $|U|$ positions to the left of the starting position of V.

If an anchor has a super-constant number of such occurrences then it must be periodic; we will exploit the periodic structure to treat them in batches.

5.2 Implementation

Let us focus at a level $\ell \in [1, \log n)$. Let $k = 2^\ell$. We consider all fragments $S[i \mathinner{.\,.} i+k-1]$ with $i = 1 \pmod{k}$ as anchors. Now, each anchor's goal is to capture squares that contain it and are at most four times longer than it. To this end, for each such anchor $P = S[i \mathinner{.\,.} i+k-1]$, we maintain the set of its *relevant occurrences* $\mathcal{A}(P)$ in $T = S[i \mathinner{.\,.} \min\{i+4k-1, n\}]$; note that $|T|/|P| = \mathcal{O}(1)$. We call this problem *anchor pattern matching.*

Now, each substitution operation, only affects a constant number of *anchor pattern matching* per level. For each affected instance, we simply recompute the occurrences of the anchor from scratch in $\mathcal{O}(\log^2 n)$ time by asking a constant number of IPM queries, employing Corollary 2.8.2. We thus have the relevant occurrences represented by a constant number of (possibly empty) arithmetic progressions. Hence, the total time required by this step is $\mathcal{O}(\log^3 n)$.

We are left with showing how to compute the longest square implied by an anchor and its relevant occurrences. This suffices, as it is then enough to maintain a representative longest square-substring for each anchor, resolving ties arbitrarily in a global max heap (with lengths as the keys).

Computing squares. Consider an anchor $P = S[i \mathinner{.\,.} i+k-1]$ and let $j \in \mathcal{A}(P)$. We want to check whether a square $S[a \mathinner{.\,.} b] = UU$, such that $a \le i < j \le b$ and $j - i = |U|$, exists. We call each such a square an (i,j)-*square*. The following lemma shows how to perform the described check efficiently.

Lemma 5.2.1. *Given two positions $i < j$, we can check whether an (i,j)-square exists and report all (i,j)-squares compactly in $\mathcal{O}(\log n)$ time w.h.p.*

Proof. The following observation reduces computing all (i,j)-squares to answering two LCE queries, which can be answered in $\mathcal{O}(\log n)$ time due to Corollary 2.8.2. Inspect Figure 5.1 for an illustration.

Observation 5.2.2. *An (i,j)-square UU, where i is the t-th letter of the first occurrence of U exists if and only if $\mathsf{LCP}^{\mathrm{R}}(i,j) \ge t$ and $\mathsf{LCP}(i,j) \ge |U| - t + 1$.*

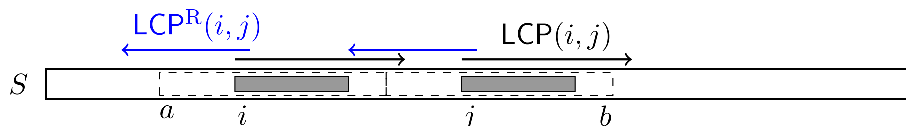


Figure 5.1: The setting in the proof of Lemma 5.2.1. The two occurrences of U in an (i,j)-square UU are denoted by dashed rectangles. The two equal k-length fragments starting at positions i and j are denoted by grey rectangles.

Now $1 \le t \le |U|$ and $t = i - a + 1$, where a is the starting position of such a square. Hence $a = i+1-t$ for $1 \le t \le |U|$ such that $\mathsf{LCP}^{\mathrm{R}}(i,j) \ge t$ and $\mathsf{LCP}(i,j) \ge |U|-t+1$ are the starting positions of all (i,j)-squares. Equivalently, the (i,j)-squares are the fragments $S[a \mathinner{.\,.} a+2|U|-1]$, with $a \in [i+1-\min\{\mathsf{LCP}^{\mathrm{R}}(i,j), |U|\}, i+\min\{\mathsf{LCP}(i,j)-|U|, 0\}]$. $\square$

If the anchor $P = S[i \mathinner{.\,.} i+k-1]$ to be processed is aperiodic, then $\mathcal{A}(P)$ is of constant size (cf. Lemma 2.3.2), and we can thus afford to employ Lemma 5.2.1 for i and each $j \in \mathcal{A}(P)$. Note that, strictly speaking, we are interested in the subset of the (i,j)-squares for which the anchor is fully contained in the first occurrence of U in UU. These are exactly the (i,j)-squares $S[a \mathinner{.\,.} b] = UU$ satisfying $i + k \le a + |U|$. To avoid

clutter, we do not impose this extra condition, as computing the longest square over a superset of the squares in scope is fine for our purposes.

Moreover, for each anchor P, we have the set of relevant occurrences represented by a constant number of (possibly empty) arithmetic progressions. In the case where P is periodic, each non-empty arithmetic progression has difference $\mathsf{per}(P)$. If there is only a constant number of relevant occurrences of P, then we treat each of them individually. Otherwise, there is at least one arithmetic progression with at least two elements and difference $\mathsf{per}(P)$.

Periodic Anchors. If the anchor is periodic, and has a super-constant number of occurrences (there could be $\Omega(n)$ of them), then processing each pair individually would be too costly. To overcome this, we exploit periodicity to process the pairs in batches.

We call a set of positions $C = \{j + t \cdot p | t = 0, \dots, r\}$ a p-cluster of a string P in S if $p = \mathsf{per}(P)$, $S[a \mathinner{.\,.} a + k - 1] = P$ for all $a \in C$ and $S[j - p \mathinner{.\,.} j - p + k - 1] \neq P \neq S[j + (r + 1)p \mathinner{.\,.} j + (r + 1)p + k - 1]$. The p-cluster containing an occurrence V of a periodic string P is the set of the occurrences of P in the run $\mathsf{run}(V)$ extending V.

Lemma 5.2.3. *Given a periodic fragment $V = S[i \mathinner{.\,.} j]$ and $p = \mathsf{per}(V)$, the run R that extends V can be computed using a constant number of LCE queries. $R = S[i - a + 1 \mathinner{.\,.} i + p + b - 1]$, where $a = \mathsf{LCP}^R(i, i + p)$ and $b = \mathsf{LCP}(i, i + p)$.*

First, we extend each such arithmetic progression to a p-cluster, using Lemma 5.2.3. Next, we process all pairs in $\{i\} \times C$ at once, relying on the following lemma.

Lemma 5.2.4. *Given a position i in S, where a string $P = S[i \mathinner{.\,.} i + k - 1]$ occurs, and a p-cluster C of P in S, we can compute a longest (i, j)-square over all $j \in C$ in $\mathcal{O}(\log n)$ time w.h.p. In particular, if $i \notin C$, we return a superset of all (i, j)-squares for $j \in C$ that are of length at least $2k$ in a compact form.*

Proof. If it so happens that $i \in C$, then the longest (i, j)-square can be easily retrieved as it must lie entirely within the run $R = S[a \mathinner{.\,.} b]$ corresponding to C. Let $r = b - a$

(mod $2p$). It can be readily verified that either $S[a+r \mathinner{.\,.} b]$ or $S[a \mathinner{.\,.} b-r]$ is a longest (i,j)-square over all $j \in C$. (See also [59].)

In the other case, that is $i \notin C$, we first compute the unique run $R_1 = S[s_1 \mathinner{.\,.} e_1]$ that extends the occurrence of P at position i, and similarly the run $R_2 = S[s_2 \mathinner{.\,.} e_2]$ corresponding to the occurrences of P in C. This can be done in $\mathcal{O}(\log n)$ time by performing a constant number of LCE queries due to Lemma 5.2.3 and Corollary 2.8.2.

We assume without loss of generality that $i < \min C$, which implies that $s_1 < s_2$. First, we treat the pair $(i, \min C)$ individually using Lemma 5.2.1 in $\mathcal{O}(\log n)$ time. Now, let UU be an (i,j)-square with $j \in C \setminus \{\min C\}$. The following fact implies that $S[e_1+1]$ lies in UU, as $j \geq \min C + p \geq s_2 + p > e_1$.

Fact 5.2.5 ([112]). *Two runs with period p cannot overlap by more than $p-1$ positions.*

We have the following cases for the occurrence of U in which $S[e_1+1]$ lies.

1. The first occurrence, in which case the endpoints $S[e_1]$ and $S[e_2]$ of the two runs must be aligned (i.e. be at distance $|U|$), since $\mathsf{LCP}(i,j) > e_1 + 2 - i$. In other words, $S[e_1]$ and $S[e_2]$ must both occur as the t-th letter of an occurrence of U in the square for some t; inspect Figure 5.2 for an illustration. In this case we compute the longest (e_1, e_2)-square (or all (e_1, e_2)-squares) in $\mathcal{O}(\log n)$ time using Lemma 5.2.1.

2. The second occurrence, in which case, the situation is more interesting. We have the following two subcases.

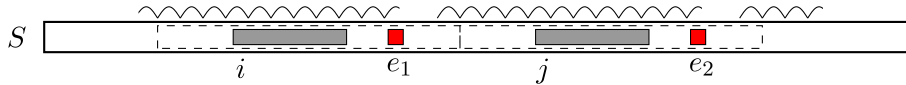


Figure 5.2: An illustration of the setting in Case 1 in the proof of Lemma 5.2.4. As before, the two occurrences of U in an (i,j)-square UU are denoted by dashed rectangles and the two equal k-length fragments starting at positions i and j are denoted by grey rectangles.

(a) If $e_1 + 1 < s_2$, by an argument symmetric to that for the first case, the starting points $S[s_1]$ and $S[s_2]$ of the two runs must be aligned—one can think of Figure 5.2 reversed. As in Case 1, we can compute the longest (all) (s_1, s_2)-square(s) in $\mathcal{O}(\log n)$ time using Lemma 5.2.1.

(b) Else, we have that the first and second occurrences of U are fragments of runs R_1 and R_2, respectively.

We now look into the structure yielded by the condition in Case (b) and show how to efficiently compute and represent all (possibly many) squares that satisfy it, and are hence captured by runs R_1 and R_2. For two runs R_1 and R_2, with period $\mathsf{per}(R_1) = \mathsf{per}(R_2) = p$ that overlap, we define $Sq(R_1, R_2)$ to be the set of squares UU of length at least $4p$ such that the first and second occurrences of U lie entirely within R_1 and R_2, respectively.

In what follows, we show how to compute $Sq(R_1, R_2)$, which is a superset of the (i, j)-squares of length at least $2k$ for $j \in C$ since $4p \leq 4k/2 \leq 2k$. We obtain a constant number of arithmetic progressions that represent all such squares. Let us start with an example that captures the structure of $Sq(R_1, R_2)$.

Example 5.2.6. Consider string $(\mathtt{baa})^4\mathtt{a}(\mathtt{baa})^3$. There are two runs with period $p = 3$, namely $R_1 = S[1\,..\,12]$ and $R_2 = S[12\,..\,22]$. See Figure 5.3 for an illustration and for the squares that satisfy the condition of Case (b). One can see that we can get $\Omega(n)$ such squares for a string of length $\mathcal{O}(n)$, by extending this paradigm and considering string $(\mathtt{baa})^n\mathtt{a}(\mathtt{baa})^n$. This example shows that a single substitution can create/destroy $\Omega(n)$ squares; think of first setting $S[n+1] := \mathtt{c}$ and then $S[n+1] := \mathtt{a}$.

Claim 5.2.7. *Let us suppose that we are given two runs $R_1 = S[s_1\,..\,e_1]$ and $R_2 = S_2[s_2\,..\,e_2]$, with $\mathsf{per}(R_1) = \mathsf{per}(R_2) = p$, such that $R_1[f\,..\,f+p-1] = R_2[1\,..\,p-1]$ for some given $f \leq s_1 + p - 1$ and such that $s_1 \leq s_2 \leq e_1 \leq e_2$. We can compute a representation of $Sq(R_1, R_2)$ in $\mathcal{O}(1)$ time.*

Proof. Fact 5.2.5 implies that Example 5.2.6 resembles the structure of the problem.

Due to the condition that the first and second occurrences of U must be fragments of runs R_1 and R_2, respectively, we have that the second occurrence of U can only start

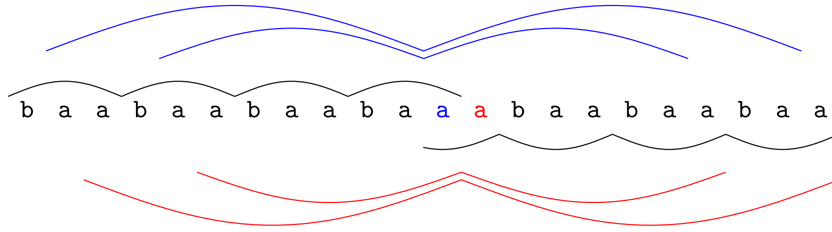


Figure 5.3: The two runs with period 3 are represented by black. The squares UU of length at least $4p$, such that the two occurrences of U are fully contained in the two runs are shown in red and blue, partitioned with respect to the first letter of the second occurrence of U.

at one of the positions in $M = \{s_2, \dots, e_1 + 1\}$, where $|M| \leq p$ by Fact 5.2.5. Let us consider some $x \in M$ and characterise all squares $S[a \,..\, b] = UU$ with $x = a + |U|$ and $|U| \geq 2p$.

$S[x-p \,..\, x-1]$ is a rotation of $S[x \,..\, x+p-1]$, i.e. there exists some $\delta < p$ such that $S[x-p \,..\, x-1] = S[x+\delta \,..\, x+p-1]S[x \,..\, x+\delta-1]$. In particular, $\delta = s_2 - f \pmod p$.

$|U|$ must equal $t \cdot p + \delta$ in order for the two occurrences of U to start at the same offset mod p from f and s_2; this is necessary, since otherwise we would have two different rotations of $R_2[1 \,..\, p-1]$ matching, which is impossible as it would imply that $\mathsf{per}(R_2) < p$. In addition, all $|U|$'s of the form $t \cdot p + \delta$ for $t \geq 2$ and for which the two occurrences of U lie entirely within runs R_1 and R_2, respectively, define valid squares. We can thus compute all these squares in $\mathcal{O}(1)$ time and represent them as an arithmetic progression with respect to $|U|$.

Example 5.2.8 (Continued). For position 12 of $(\mathtt{baa})^4\mathtt{a}(\mathtt{baa})^3$, the blue $\mathtt{a}$ in Figure 5.3, we have $\delta = 1$ and hence the squares UU that we obtain with this as starting position of the second occurrence of U are for $|U| = 1 + 3t$, for $t = 2, 3$.

Iterating over $x \in M$ in increasing order, we only have to (a) shift all squares by 1 position each time, and (b) identify the—at most two—shifts that yield an increment/decrement in the length of the arithmetic progression due to one more/less square being allowed after the shift. We can infer the values of x for which we must incre-

ment/decrement in $\mathcal{O}(1)$ time from the endpoints of the two runs and δ. These values, p, and the arithmetic progression for $x = s_2$ are our representation of $Sq(R_1, R_2)$. $\square$

We can straightforwardly extract the longest (i, j)-square for $j \in C$ if it is of length at least $2k$ from this representation, and this concludes the proof of the theorem. $\square$

To summarise, each string update affects $\mathcal{O}(\log n)$ anchor pattern matching instances; the corresponding sets of relevant occurrences are recomputed in $\mathcal{O}(\log^3 n)$ time in total. Then, for each anchor belonging to an affected instance, we invoke each of Lemma 5.2.1 and Theorem 5.2.4 at most a constant number of times. The overall time complexity of our algorithm is thus $\mathcal{O}(\log^3 n)$. The space required throughout the execution of our algorithm is $\mathcal{O}(n)$: apart from the space required for the data structure of Corollary 2.8.2, for each $\ell \in [1, \log n)$ for each of the $\mathcal{O}(n/2^\ell)$ anchor pattern matching instances we simply store at most one square in the global max heap. Similarly, in order to initialise the data structure, we need $\mathcal{O}(n \log^2 n/2^\ell)$ time for each level ℓ.

Theorem 1.4.1. *The longest square of a dynamic string of length n that undergoes substitution operations can be maintained in $\mathcal{O}(\log^3 n)$ time per each such operation, using $\mathcal{O}(n)$ space, after an $\mathcal{O}(n \log^2 n)$-time preprocessing. All running times hold w.h.p.*

In order to allow for arbitrary edit operations, i.e. also allow for insertions and deletions of letters, it is enough to maintain, for each level ℓ, anchors of length 2^ℓ, such that every interval $[i+1, i+2^\ell] \subseteq [1, |S|]$ contains the starting position of at least one of them. In order to make sure that an edit operation does not affect too many anchor pattern matching instances, one has to also impose the condition that no two anchors are closer than some constant fraction of 2^ℓ. This can be implemented, in each level, with the aid of an augmented balanced binary search tree storing the starting positions of the anchors, and allowing us to keep track of the offsets in indices due to the insertions and deletions of letters. This incurs no extra cost asymptotically.

As for maintaining all squares of string S, it is enough to ensure that each one of them is captured by a unique anchor; say the leftmost anchor of the topmost level for which it is computed.

Analogues of Lemmas 5.2.1 and 5.2.4 for computing runs can be found in [8], as well as $n^{o(1)}$-time algorithms for maintaining all squares and runs of a dynamic string. Let us note here, that Example 5.2.6 shows that there are positions contained in $\Omega(n)$ runs, as each square denoted by red extends to a unique run. However, these can be compactly represented by the two runs denoted by black.

Chapter 6

Internal and Dynamic Longest Common Factor

In this chapter, we first consider LCF queries in the internal setting. In the most general version of the problem, we are given two strings S and T for preprocessing, and upon a query we are to report an LCF between a substring of S and a substring of T. We first show a conditional lower bound via reducing the Set Disjointness problem to the internal LCF problem. We then explore restricted versions of internal LCF queries and design efficient solutions for them.

Then, in Section 6.2, we consider the partially dynamic LCF problem, where updates are only allowed in one of the strings. In this problem, we use the static string T as a reference point. We maintain a partition of the dynamic string S into blocks (i.e. substrings of S whose concatenation equals S), such that each block is a substring of T, but the concatenation of any two consecutive blocks is not. This is similar to the approach of [14] and other works that consider one dynamic and one static string. The improvement upon the $\tilde{\mathcal{O}}(\sqrt{n})$-time algorithm presented in [10] comes exactly from imposing the aforementioned maximality property, which guarantees that the sought LCF is a substring of the concatenation of at most three consecutive blocks and contains the first letter of one of these blocks. The latter property allows us to anchor the LCF in

S. Upon an update, we can maintain the block decomposition, by updating a constant number of blocks. It then suffices to show how to efficiently compute the longest substring of T that contains the first letter of a given block.

Finally, in Section 6.3 we move to the fully dynamic LCF problem. We try to anchor the LCF in both strings as follows. For each of the strings S and T we show how to maintain, in $\tilde{\mathcal{O}}(1)$ time, a collection of pairs of consecutive fragments (e.g. $(S[i \mathinner{.\,.} j-1], S[j \mathinner{.\,.} k])$), denoted by J_S for S and J_T for T, with the following property. For any common substring X of S and T there exists a partition $X = X_\ell X_r$ for which there exists a pair $(U_\ell, U_r) \in J_S$ and a pair $(V_\ell, V_r) \in J_T$ such that X_ℓ is a suffix of both U_ℓ and V_ℓ, while X_r is a prefix of both U_r and V_r. We can maintain this collection by exploiting the properties of the locally consistent parsing underlying Theorem 1.3.1 ([82]). We maintain tries for fragments in the collections J_S and J_T, and reduce the dynamic LCF problem to a problem on dynamic bicoloured trees, which we solve by using dynamic heavy-light decompositions and 2D range trees.

6.1 Internal LCF queries

6.1.1 A Lower Bound Based on Set Disjointness

In the Set Disjointness problem, we are given a collection of m sets $A_1, A_2, \ldots, A_m$ of total size N from some universe U for preprocessing in order to answer queries on the emptiness of the intersection of some two query sets from the collection. Set Disjointness has been used to obtain lower bounds for problems such as distance oracles and reachability, see e.g. [87, 52]. Goldstein et al. focused on the hardness of Set Disjointness with regard to its space-query time tradeoff [87]. Specifically, they stated the following conjecture.

Conjecture 6.1.1 (Strong Set Disjointness Conjecture [87]). *Any data structure for the Set Disjointness problem that answers queries in time t must use space $N^2/(t^2 \cdot \log^{\mathcal{O}(1)} N)$.*

Conjecture 6.1.1 is a generalisation of the Set Disjointness conjecture stating that

any data structure for the Set Disjointness problem with constant query time must use $N^{2-o(1)}$ space [87, 145, 52].

Theorem 6.1.2. *Any data structure answering internal LCF queries for two strings, each of length at most n, in time t must use $n^2/(t^2 \cdot \log^{\mathcal{O}(1)} n)$ space, unless the Strong Set Disjointness Conjecture is false.*

Proof. We reduce the Set Disjointness problem to that of answering internal LCF queries as follows. Given sets $A_1, \ldots, A_m$ of total cardinality N, we construct a string T of length $n := N$ that consists of the concatenation of the elements of the sets, so that each set A_i corresponds to the substring $T[a_i \mathinner{.\,.} b_i]$ of T and $\{T[a_i], T[a_i+1], \ldots, T[b_i]\} = A_i$. Further, consider a copy T' of T. Then, for any two sets A_i and A_j, $A_i \cap A_j$ is empty if and only if the length of an LCF of $T[a_i \mathinner{.\,.} b_i]$ and $T'[a_j \mathinner{.\,.} b_j]$ is 0. □

Conditional lower bounds for the preprocessing time-query time tradeoff are also known [121, 122], and these also carry over to our problem due to the reduction in the above lemma. In particular, consider a data structure for the Set Disjointness problem, and let the preprocessing time be N^p and the query time be N^q. Then, Kopelowitz et al. [121] showed that $p + 2q \geq 2$ conditional on the 3SUM hypothesis. Very recently, Kopelowitz and Vassilevska Williams [122] showed that if $1/3 \leq q < 1$, then $2p + q \geq 3$, conditional on the so-called Unbalanced Triangle Detection hypothesis.

The proof of Theorem 6.1.2 mimics the proof of Amir et al. [15] for the hardness of so-called two-range-LCP queries. In the two-range-LCP problem, one is to preprocess a string so that queries of the following type can be answered: given two ranges I and J, return $\max_{i \in I, j \in J} \mathsf{LCP}(i, j)$. Amir et al. [15] presented, for any $t \in [1, \sqrt{n}]$ and constant $\epsilon > 0$, an $\mathcal{O}(n + n^2/(t^2 \cdot \log n))$-size data structure that answers two-range-LCP queries in $\mathcal{O}(t \cdot \log^{\epsilon} n)$ time.

Now, note that a general internal LCF query can be reduced via binary search to $\mathcal{O}(\log n)$ two-range-LCP queries as follows. The length of an LCF between $S[a_1 \mathinner{.\,.} b_1]$ and $T[a_2 \mathinner{.\,.} b_2]$ is at least m if and only if the two-range-LCP on the concatenation of S and T with intervals $[a_1 \mathinner{.\,.} b_1 - m + 1]$ and $[|S| + a_2 \mathinner{.\,.} |S| + b_2 - m + 1]$ is at least m. We

summarise the above discussion in the following statement.

Proposition 6.1.3. *Given two strings of total length n, a parameter $t \in [1, \sqrt{n}]$ and a constant $\epsilon > 0$, there is an $\mathcal{O}(n + n^2/(t^2 \log n))$-size data structure that answers internal LCF queries in $\mathcal{O}(t \cdot \log^{1+\epsilon} n)$ time.*

6.1.2 Internal Queries for Special Substrings

We now show efficient data structures for answering simpler types of internal LCF queries. In particular, we show how to answer internal LCF queries for a prefix or suffix of S and a prefix or suffix of T and for a substring of S and T.

Let us denote the length of an LCF of two strings U and V by $\texttt{LCF}(U, V)$. In our solutions we use the formula:

$$\texttt{LCF}(S[a \mathinner{..} b], T[c \mathinner{..} d]) = \max_{\substack{i=a,\dots,b,\\ j=c,\dots,d}} \{\min\{\mathsf{LCP}(S[i \mathinner{..} |S|], T[j \mathinner{..} |T|]),\, b-i+1,\, d-j+1\}\}. \quad (6.1)$$

We also apply the following observation to create range maximum queries data structures over points constructed from explicit nodes of the suffix tree of $S\#T$, where $\# \notin \Sigma \cup \{\$\}$. (As before, we denote the string-depth of a node v by $w(v)$.)

Observation 6.1.4. *Let S and T be two strings of length n each. We have*

$$\{\mathit{LCP}(S[i \mathinner{..} n], T[j \mathinner{..} n] \,:\, i, j \in [1, n]\} \subseteq \{w(v) \,:\, v \text{ is explicit in } \mathcal{T}(S\#T)\}.$$

Lemma 6.1.5. *Let S and T be two strings of length at most n. After $\mathcal{O}(n \log n)$-time and $\mathcal{O}(n \log n)$-space preprocessing, an LCF between any prefix or suffix of S and any prefix or suffix of T can be computed in $\mathcal{O}(\log^2 n)$ time.*

Proof. For a node v of $\mathcal{T}(S\#T)$ and $U \in \{S, T\}$ we define:

$$\mathit{minPref}(v, U) = \min\{i \,:\, \mathcal{L}(v) \text{ is a prefix of } U[i \mathinner{..} |U|]\},$$
$$\mathit{maxPref}(v, U) = \max\{i \,:\, \mathcal{L}(v) \text{ is a prefix of } U[i \mathinner{..} |U|]\}.$$

We assume that $\min\emptyset = \infty$ and $\max\emptyset = -\infty$. These values can be computed for all explicit nodes of $\mathcal{T}(S\#T)$ in $\mathcal{O}(n)$ time in a bottom-up traversal of the tree.

We only consider computing $\mathtt{LCF}(S[a\mathinner{.\,.}|S|], T[b\mathinner{.\,.}|T|])$ and $\mathtt{LCF}(S[1\mathinner{.\,.}a], T[b\mathinner{.\,.}|T|])$ as the remaining cases can be solved by considering the reversed strings.

In the first case, formula (6.1) has an especially simple form:

$$\mathtt{LCF}(S[a\mathinner{.\,.}|S|], T[b\mathinner{.\,.}|T|]) = \max_{i\ge a,\, j\ge b} \mathsf{LCP}(S[i\mathinner{.\,.}|S|], T[j\mathinner{.\,.}|T|])$$

which lets us use orthogonal range maximum queries to evaluate it. For each explicit node v of $\mathcal{T}(S\#T)$ with descendants from both S and T we create a point (x,y) with weight $w(v)$, where $x = \mathit{maxPref}(v,S)$ and $y = \mathit{maxPref}(v,T)$. By Observation 6.1.4, the sought LCF length is the maximum weight of a point in the rectangle $[a,n]\times[b,n]$. This lets us also recover the LCF itself. The complexity follows from Theorem 2.5.3.

In the second case, formula (6.1) becomes:

$$\mathtt{LCF}(S[1\mathinner{.\,.}a], T[b\mathinner{.\,.}|T|]) = \max_{i\le a,\, j\ge b} \min(\mathsf{LCP}(S[i\mathinner{.\,.}|S|], T[j\mathinner{.\,.}|T|]),\ a-i+1).$$

The result is computed in one of two steps depending on which of the two terms produces the minimum. First let us consider the case where $\mathsf{LCP}(S[i\mathinner{.\,.}|S|], T[j\mathinner{.\,.}|T|]) < a-i+1$. For each explicit node v of $\mathcal{T}(S\#T)$ with descendants from both S and T we create a point (x,y) with weight $w(v)$, where $x = \mathit{minPref}(v,S)+w(v)-1$ and $y = \mathit{maxPref}(v,T)$. The answer r_1 is the maximum weight of a point in the rectangle $[1,a-1]\times[b,n]$.

In the opposite case we can assume that the resulting internal LCF is a suffix of $S[1\mathinner{.\,.}a]$ that does not occur earlier in S. For each explicit node v of $\mathcal{T}(S\#T)$ we create a point (x,y) with weight x', where $x' = \mathit{minPref}(v,S)$, $x = x'+w(v)-1$, and $y = \mathit{maxPref}(v,T)$. Let i be the minimum weight of a point in the rectangle $[a,n]\times[b,n]$. If $i \le a$, then we set $r_2 = a-i+1$. Otherwise, we set $r_2 = -\infty$.

In both cases we use the 2D RMQ data structure of Theorem 2.5.3. In the end, we return $\max(r_1, r_2)$ and the corresponding LCF. $\square$

The following lemma provides an efficient solution for the other special case of internal LCF that we consider.

Lemma 6.1.6. *Let S and T be two strings of length at most n. After $\mathcal{O}(n)$-time preprocessing, one can compute an LCF between T and any substring of S in $\mathcal{O}(\log n)$ time.*

Proof. We define $B[i] = \max_{j=1,\dots,|T|}\{\mathsf{LCP}(S[i\,..\,|S|], T[j\,..\,|T|])\}$. The following fact was shown in [9]. Here we give a proof for completeness.

Claim 6.1.7 ([9])**.** *The values $B[i]$ for all $i = 1, \dots, |S|$ can be computed in $\mathcal{O}(n)$ time.*

Proof. For every explicit node v of $\mathcal{T}(S\#T)$ let us compute the length $\ell(v)$ of the longest common prefix of $\mathcal{L}(v)$ and any suffix of T. The values $\ell(v)$ are computed in a top-down manner. If v has as a descendant a leaf from T, then clearly $\ell(v) = w(v)$. Otherwise, we set $\ell(v)$ to the value computed for v's parent. Finally, the values $B[i]$ can be read at the leaves of $\mathcal{T}(S\#T)$. □

The formula (6.1) can be written as:

$$\mathtt{LCF}(S[a\,..\,b], T) = \max_{a\le k\le b}\{\min(B[k], b-k+1)\}.$$

The function $f(k) = b - k + 1$ is decreasing. We are thus interested in the smallest $k_0 \in [a, b]$ such that $B[k_0] \ge b - k_0 + 1$. If there is no such k_0, we set $k_0 = b + 1$. This lets us restate the previous formula as follows:

$$\mathtt{LCF}(S[a\,..\,b], T) = \max(\max_{a\le k<k_0}\{B[k]\},\, b - k_0 + 1).$$

Indeed, for $a \le k < k_0$ we know that $\min(B[k], b-k+1) = B[k]$, for $k = k_0$ we have $\min(B[k], b-k+1) = b-k_0+1$, and for $k_0 < k \le b$ we have $\min(B[k], b-k+1) \le b-k+1 \le b-k_0+1$.

The final formula for LCF length can be evaluated in $\mathcal{O}(1)$ time with a data structure for range maximum queries over B, provided that k_0 is known. This lets us also recover the LCF itself.

Computation of k_0. The condition for k_0 can be stated equivalently as $B[k_0]+k_0 \geq b+1$. We create an auxiliary array $B'[i] = B[i] + i$. To find k_0, we need to find the smallest index $k \in [a, b]$ such that $B'[k] \geq b + 1$. We can do this in time $\mathcal{O}(\log n)$ by performing a binary search for k in the range $[a, b]$ of B' using $\mathcal{O}(n)$-time preprocessing and $\mathcal{O}(1)$-time range maximum queries (cf. Theorem 2.5.1). □

6.1.3 Concat LCF queries

In [9] we (Amir et al.) observed that the problem of computing an LCF after a single edit operation at position i can be decomposed into two queries out of which we choose the one with the maximal answer: an occurrence of an LCF either avoids i or it covers i. The former case can be precomputed. The latter, reduces to answering CONCAT LCF queries, formalised below.

> CONCAT LCF
>
> **Input:** A string T of length n.
>
> **Query:** Given two substrings U and V of T, compute the longest substring XY of T such that X is a suffix of U and Y is a prefix of V.

The CONCAT LCF can be reduced to the so-called HEAVIEST INDUCED ANCESTORS (HIA) problem, which we now proceed to define.

We say that a tree is labelled if each of its leaves is given a distinct label. For rooted, weighted, labelled trees $\mathcal{T}_1$ and $\mathcal{T}_2$, we say that two nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$ are *induced* (by a label ℓ) if and only if there are leaves x and y with the same label (ℓ), such that x is a descendant of u and y is a descendant of v.

> HEAVIEST INDUCED ANCESTORS
>
> **Input:** Two rooted, weighted, labelled trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size n.
>
> **Query:** Given a pair of nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$, return a pair of nodes u', v' such that u' is ancestor of u, v' is ancestor of v, u' and v' are induced and they have the largest total combined weight $w(u') + w(v')$.

This problem was introduced in [76]; further advances were made in [2, 42]. In the

following theorem, we use the variant of the data structure from Section 2.1 in [76]. Although the construction time is not stated explicitly in [76], the data structure can be straightforwardly constructed in the time specified below.

Theorem 6.1.8 ([76]). *There is an $\mathcal{O}(n \log^2 n)$-size data structure for the* Heaviest Induced Ancestors *problem, that can be built in $\mathcal{O}(n \log^2 n)$ time and answers queries in $\mathcal{O}(\log n \log\log n)$ time.*

The following result was recently shown in [42].

Theorem 6.1.9 ([42]). *For any constant $\epsilon > 0$, there is an $\mathcal{O}(n^{1+\epsilon})$-size structure for the* Heaviest Induced Ancestors *problem, that can be built in $\mathcal{O}(n^{1+\epsilon})$ time and answers queries in $\mathcal{O}(1)$ time.*

The following lemma was (implicitly) shown in [9] and explicitly in [2]. We include its proof, as we will need it in our solution for the partially dynamic LCF problem, and it can help the reader build intuition for our solution for the fully dynamic LCF problem.

Lemma 6.1.10. *Given a string T of length n, we can construct in $\mathcal{O}(n)$ time two trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size $\mathcal{O}(n)$, so that, each* Concat LCF *query for T can be reduced in $\mathcal{O}(\log\log n)$ time to a constant number of* Heaviest Induced Ancestors *queries over $\mathcal{T}_1$ and $\mathcal{T}_2$.*

Proof. We first construct $\mathcal{T}_1 = \mathcal{T}(T^R)$ and $\mathcal{T}_2 = \mathcal{T}(T)$ and the weighted ancestor data structure of Theorem 2.4.1 for efficiently computing loci of substrings. The leaf corresponding to $(T[1\mathinner{.\,.} i-1])^R$ in $\mathcal{T}_1$ and to suffix $T[i\mathinner{.\,.}|T|]$ in $\mathcal{T}_2$ are labelled with i. For the sake of HIA queries, we treat $\mathcal{T}_1$ and $\mathcal{T}_2$ as weighted trees over the set of explicit nodes. (Recall that each node has a string-depth and this is its weight.) We make the following observation.

Observation 6.1.11. *Explicit nodes u_1 of $\mathcal{T}_1$ and u_2 of $\mathcal{T}_2$ are induced if and only if $\mathcal{L}(u_1)^R\mathcal{L}(u_2)$ is a substring of T.*

Let u be the locus of U^R in $\mathcal{T}_1$ and v be the locus of V in $\mathcal{T}_2$. We can compute these loci in time $\mathcal{O}(\log\log n)$. By the claim, if both u and v are explicit nodes, then the

problem reduces to a HIA query for u and v. If only one of u and v is implicit, say u, then we do the following. Let u be an implicit node along the edge from u_1 to u_2. We ask a HIA query for u_1 and v, and a HIA query for u_2 and v. If u_2 is the node of $\mathcal{T}_1$ returned by the second query, we decrease the combined weight of this pair of nodes by $|\mathcal{L}(u_2)| - |U|$. Then, we return the answer with maximum combined weight. The case where both u and v are implicit nodes can be taken care of similarly; in particular, it reduces to three HIA queries. □

6.2 Partially Dynamic LCF

In this section, we describe an algorithm for solving the partially dynamic variant of the LCF problem, where updates are only allowed in one of the strings, say S, while T is given in advance and is not subject to change.

Let us assume for now that all the letters of S throughout the execution of the algorithm occur at least once in T; we will waive this assumption later. Also, for simplicity, we assume that S is initially equal to $\$^{|S|}$, for $\$ \notin \Sigma$. We can always obtain any other initial S by performing an appropriate sequence of updates in the beginning.

Definition 6.2.1. *A* block decomposition *of string S with respect to string T is a sequence of strings $(S_1, S_2, \ldots, S_k)$ such that $S = S_1 S_2 \cdots S_k$ and every S_i is a fragment of T. An element of the sequence is called a* block *of the decomposition. A decomposition is* maximal *if and only if $S_i S_{i+1}$ is not a substring of T for every $i \in [1, k-1]$.*

Maximal block decompositions are not necessarily unique and may have different lengths, but all admit the following useful property.

Lemma 6.2.2. *For any maximal block decomposition of S with respect to T, any substring of S that occurs in T is contained in at most three consecutive blocks. Furthermore, any occurrence of an LCF of S and T in S must contain the first letter of some block.*

Proof. We prove the first claim by contradiction. If $(S_1, S_2, \ldots, S_k)$ is a maximal block decomposition of S with respect to T and a fragment of S that occurs in T spans at

least four consecutive blocks $S_i, S_{i+1}, S_{i+2}, \ldots, S_j$, then $S_{i+1}S_{i+2}$ is a substring of T, a contradiction.

As for the second claim, it is enough to observe that if an occurrence of an LCF in S starts in some other than the first position of a block S_i, then it must contain the first letter of the next block, as otherwise its length would be smaller than $|S_i|$, which is a common substring of S and T. $\square$

We will show that an update in S can be processed by considering a constant number of blocks in a maximal block decomposition of S with respect to T. We first summarise the basic building block needed for efficiently maintaining such a maximal block decomposition.

Lemma 6.2.3. *Let T be a string of length at most n. After $\mathcal{O}(n \log^2 n)$-time and $\mathcal{O}(n)$-space preprocessing, given two fragments U and V of T, one can compute a longest fragment of T that is equal to a prefix of UV in $\mathcal{O}(\log\log n)$ time.*

Proof. We build a weighted ancestor queries structure over the suffix tree of T. We also build a data structure for answering unrooted LCP queries over the suffix tree of T. In our setting, such queries can be defined as follows: given nodes u and v of the suffix tree of T, we want to compute the (implicit or explicit) node where the search for the path-label of v starting from node u ends. Cole et al. [53] showed how to construct in $\mathcal{O}(n \log^2 n)$ time a data structure of size $\mathcal{O}(n \log n)$ that answers unrooted LCP queries in $\mathcal{O}(\log\log n)$ time. With these data structures at hand, the longest prefix of UV that is a fragment of T can be computed as follows. First, we retrieve the nodes of the suffix tree of T corresponding to U and V using weighted ancestor queries in $\mathcal{O}(\log\log n)$ time. Second, we ask an unrooted LCP query to obtain the node corresponding to the sought prefix of UV. $\square$

Lemma 6.2.4. *A maximal block decomposition of a dynamic string S, with respect to a static string T, can be maintained in $\mathcal{O}(\log\log n)$ time per substitution operation with a data structure of size $\mathcal{O}(n \log n)$ that can be constructed in $\mathcal{O}(n \log^2 n)$ time.*

Proof. We keep the blocks on a doubly-linked list and we store the starting positions of blocks in an $\mathcal{O}(n)$-size predecessor/successor data structure over $[1,n]$ that supports updates and queries in $\mathcal{O}(\log\log n)$ time using Theorem 2.5.2. This allows us to navigate in the structure of blocks, and in particular to be able to compute the block in which the edit occurred and its neighbours.

Suppose that we have a maximal block decomposition $B=(S_1,\ldots,S_k)$ of S with respect to T. Let us consider a substitution operation with letter b at position t of block S_i, and let $S_i=S_i^{\mathrm{l}}aS_i^{\mathrm{r}}$, with $|S_i^{\mathrm{l}}a|=t$. Consider the following block decomposition of string S' after the update: $B'=(S_1,S_2,\ldots,S_{i-1},S_i^{\mathrm{l}},b,S_i^{\mathrm{r}},S_{i+1},\ldots,S_k)$. Note that both S_i^{l} and S_i^{r} may be empty. This block decomposition does not need to be maximal. However, since B is a maximal block decomposition of S, none of the strings S_1S_2, S_2S_3, …, $S_{i-2}S_{i-1}$, $S_{i+1}S_{i+2}$, $S_{i+2}S_{i+3}$, …, $S_{k-1}S_k$ occurs in T. Thus, given B', we repeatedly merge any two consecutive blocks from $(S_{i-1},S_i^{\mathrm{l}},b,S_i^{\mathrm{r}},S_{i+1})$ whose concatenation is a substring of T into one, until this is no longer possible. We have at most four merges before obtaining a maximal block decomposition B' of string S'. Each merge is implemented with Lemma 6.2.3 in $\mathcal{O}(\log\log n)$ time. □

As for allowing substitutions of letters that do not occur in T, we simply allow blocks of length 1 that are not substrings of T in block decompositions, corresponding to such letters. It is readily verified that all the statements above still hold.

Due to Lemma 6.2.2, for a maximal block decomposition $(S_1,S_2,\ldots,S_k)$ of S with respect to T, we know that any occurrence of an LCF of S and T in S must contain the first letter of some block of the decomposition and cannot span more than three blocks. In other words, it is the concatenation of a potentially empty suffix of $S_{i-1}S_i$ and a potentially empty prefix of $S_{i+1}S_{i+2}$ for some $i\in[1,k]$ (for convenience we consider the non-existent S_is to be equal to ε). We call an LCF that can be decomposed in such way a candidate of S_i. Our goal is to maintain the candidate proposed by each S_i in a max-heap with the length as the key. We also store a pointer to it from block S_i. The max-heap can be implemented with an $\mathcal{O}(n)$-size data structure that supports updates and queries in $\mathcal{O}(\log\log n)$ time using Theorem 2.5.2. We assume that each block S_i

stores a pointer to its candidate in the max-heap.

After an update, the candidate of each block S_i that satisfies the following two conditions remains unchanged: (a) S_i did not change and (b) neither of S_i's neighbours at distance at most 2 changed. For the $\mathcal{O}(1)$ blocks that changed, we proceed as follows. First, we remove from the max-heap any candidates proposed by the deleted blocks or blocks whose neighbours at distance at most 2 have changed. Then, for each new block and for each block whose neighbours at distance at most 2 have changed, we compute its candidate and insert it to the max-heap. To compute the candidate of a block S_i, we proceed as follows. We first compute the longest suffix U of $S_{i-1}S_i$ and the longest prefix V of $S_{i+1}S_{i+2}$ that occur in T in $\mathcal{O}(\log \log n)$ time using Lemma 6.2.3. Then, we have to ask a CONCAT LCF query for U and V. This reduces to a constant number of HIA queries as per Lemma 6.1.10. Finally, for HIA queries we employ either the data structure of Theorem 6.1.8 or the data structure of Theorem 6.1.9. We thus obtain the main result of this section.

Theorem 1.5.2. *We can maintain an LCF of a dynamic string S and a static string T, each of length at most n,*

(a) in $\mathcal{O}(\log n \log \log n)$ time per substitution operation using $\mathcal{O}(n \log^2 n)$ space, after an $\mathcal{O}(n \log^2 n)$-time preprocessing, or

(b) in $\mathcal{O}(\log \log n)$ time per substitution operation using $\mathcal{O}(n^{1+\epsilon})$ space, after an $\mathcal{O}(n^{1+\epsilon})$-time preprocessing, for any constant $\epsilon > 0$.

Let us note that our solution can be easily extended to allow for arbitrary edit operations, i.e. insertions and deletions of letters as well, with an additive $\mathcal{O}(\log n)$ factor for processing each update. We just need to maintain an augmented balanced binary search tree over the blocks, in order to be able to offset their start and end indices upon insertions and deletions of letters.

6.3 Fully Dynamic LCF

In this section, we prove the main result of this chapter.

Theorem 1.5.3. *We can maintain an LCF of two initially empty dynamic strings, each of length at most n, in $\mathcal{O}(\log^8 n)$ amortised time w.h.p. per edit operation.*

We start with some intuition. Let us suppose that we can maintain a decomposition of each string in level-k blocks of length roughly 2^k for each level $k = 0, 1, \dots, \log n$ with the following property: any two equal fragments $U = S[i \mathinner{.\,.} j]$ and $V = T[i' \mathinner{.\,.} j']$ are "aligned" by a pair of equal blocks B_1 in S and B_2 in T at some level k such that $2^k = \Theta(|U|)$. In other words, the decomposition of U (resp. V) at level k consists of a constant number of blocks, where the first and last blocks are potentially trimmed, including B_1 (resp. B_2), and the distance of the starting position of B_1 from position i in S equals the distance of the starting position of B_2 from position i' in T. The idea is that we can use such blocks as anchors for the LCF. For each level, for each string B appearing as a block in this level, we would like to design a data structure that:

(a) supports insertions and deletions of strings corresponding to sequences of a constant number of level-k blocks, each containing a specified block equal to B and a boolean variable indicating the string this sequence originates from (S or T), and

(b) can return the LCF among pairs of elements originating from different strings that is aligned by a pair of specified blocks (that are equal to B).

For each edit operation in either of the strings, we would only need to update $\mathcal{O}(\log n)$ entries in our data structures: a constant number of them per level.

Unfortunately, it is not clear how to maintain a decomposition with these properties. We resort to the dynamic maintenance of a *locally consistent parsing* of the two strings, due to Gawrychowski et al. [82]. We exploit the structure of this parsing in order to apply the high-level idea outlined above in a much more technically demanding setting.

6.3.1 Locally Consistent Parsing

As also discussed in earlier chapters, at the heart of Theorem 1.3.1 due to Gawrychowski et al. for maintaining a dynamic collection of strings ([82]) lies a locally consistent parsing of the strings in the collection that can be maintained efficiently while the strings undergo updates. It can be interpreted as a dynamic version of the recompression technique of Jeż [103, 102] (see also [98]) for a static string T. We now describe in more detail the structure of this parsing for a static string T and then extend the description to the dynamic variant for a collection of strings.

Recall that a run-length straight-line program (RLSLP) is a context-free grammar which generates exactly one string and contains two kinds of non-terminals: *concatenations* with production rule of the form $A \to BC$ (for symbols B, C) and *powers* with production rule of the form $A \to B^k$ (for a symbol B and an integer $k \geq 2$).

Let $T = T_0$. We can compute strings $T_1, \ldots, T_H$, where $H = \mathcal{O}(\log n)$ and $|T_H| = 1$ in $\mathcal{O}(n)$ time using interleaved calls to the following two auxiliary procedures:

RunCompress applied if h is even: for each B^r, $r > 1$, replace all occurrences of B^r as an RLE run by a new letter A. There are no RLE runs of length greater than one after an application of this procedure.

HalfCompress applied if h is odd: first partition Σ into Σ_ℓ and Σ_r; then, for each pair of letters $B \in \Sigma_\ell$ and $C \in \Sigma_r$ such that BC occurs in T_h replace all occurrences of BC by a new letter A.

We can interpret strings $T = T_0, T_1, \ldots, T_H$ as an *uncompressed* parse tree $\mathsf{UPT}[T]$, by considering their letters as nodes, so that the parent of $T_h[i]$ is the letter of T_{h+1} that either (a) corresponds to $T_h[i]$ or (b) replaced a fragment of T_h containing $T_h[i]$.[1] We say that the node representing $T_h[i]$ is the node left (resp. right) of the node representing $T_h[i+1]$ (resp. $T_h[i-1]$). Every node v of $\mathsf{UPT}[T]$ is labelled with the symbol it represents,

[1]Uncompressed here means that nodes are allowed to have a single child, as opposed to our definition of parse trees used in earlier chapters. The actual parse tree can be obtained by contracting such edges.

denoted by $\mathcal{L}(v)$. For a node v corresponding to a letter of T_h, we say that the level of v, denoted by $\mathsf{lev}(v)$, is h.

As before, the value $\mathsf{val}(v)$ of a node v is defined as the fragment of T corresponding to the leaf descendants of v and it is an occurrence of $\mathsf{gen}(A)$ for $A = \mathcal{L}(v)$. A sequence of nodes in UPT is a layer if the values of those nodes are consecutive fragments of T. We call a layer $v_1 v_2 \cdots v_r$ an *up-layer* when $\mathsf{lev}(v_i) \leq \mathsf{lev}(v_{i+1})$ for all i, and a *down-layer* when $\mathsf{lev}(v_i) \geq \mathsf{lev}(v_{i+1})$ for all i.

In [82], the authors show how to maintain an RLSLP $\mathsf{UPT}[T]$ for each string T in the collection $\mathcal{X}$, that is consistent with the above constructive description [103, 102]. Each such RLSLP is guaranteed to have at most $c' \log N$ levels with high probability, where N is the total length of the strings in $\mathcal{X}$ and c' is a global constant. As also discussed in Section 2.8, we can reinitialise our data structure after every $\mathcal{O}(n)$ operations, so that $N = n^{\mathcal{O}(1)}$, and hence $c' \log N \leq c \log n$ for some global constant c, throughout the execution of the algorithm.

Let T be a string in the collection $\mathcal{X}$. We have defined the popped sequence of a fragment $T[a \mathinner{.\,.} b]$ as a sequence labeling a certain layer of nodes of the parse tree of T, whose values constitute $T[a \mathinner{.\,.} b]$. Naturally, this sequence also labels a layer of nodes of $\mathsf{UPT}[T]$. We now look more closely at the internals of Theorem 4.1.1, restated here for convenience.

Theorem 4.1.1 ([98])**.** *If two fragments of strings in $\mathcal{X}$ are equal, then their popped sequences are equal as well. Moreover, w.h.p., each popped sequence consists of $\mathcal{O}(\log N)$ RLE runs (maximal powers of a single symbol) and can be constructed in $\mathcal{O}(\log N)$ time. The nodes corresponding to symbols in an RLE run share a single parent. Furthermore, the popped sequence consists of a single symbol only for fragments of length* 1*.*

For each fragment $U = T[a \mathinner{.\,.} b]$ of T, one can compute in $\mathcal{O}(\log n)$ time a layer $\mathsf{C}(U)$ of nodes in $\mathsf{UPT}[T]$ with value $T[a \mathinner{.\,.} b]$ and has the following property. It can be decomposed into an up-layer $\mathsf{C}_{\mathsf{up}}(U)$ and a down-layer $\mathsf{C}_{\mathsf{down}}(U)$ such that:

- The sequence of the labels of the nodes in $\mathsf{C}_{\mathsf{up}}(U)$ can be expressed as a sequence of

at most $c \log n$ symbols and powers of symbols $\mathsf{d}_{\mathsf{up}}(U) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ such that, for all i, $A_i^{r_i}$ corresponds to r_i consecutive nodes at level i of $\mathsf{UPT}[T]$; r_i can be 0 for $i < m$.

- Similarly, the sequence of the labels of the nodes in $\mathsf{C}_{\mathsf{down}}(U)$ can be expressed as a sequence of at most $c \log n$ symbols and powers of symbols $\mathsf{d}_{\mathsf{down}}(U) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$ such that, for all i, $B_i^{t_i}$ corresponds to t_i consecutive nodes at level i of $\mathsf{UPT}[T]$; t_i can be equal to 0.

The concatenation $\mathsf{d}(U)$ of $\mathsf{d}_{\mathsf{up}}(U)$ and $\mathsf{d}_{\mathsf{down}}(U)$ is the popped sequence of U. Note that $U = \mathsf{gen}(\mathsf{d}(U)) = \mathsf{gen}(A_0)^{r_0} \cdots \mathsf{gen}(A_m)^{r_m} \mathsf{gen}(B_m)^{t_m} \cdots \mathsf{gen}(B_0)^{t_0}$. See Figure 6.1 for a visualisation.

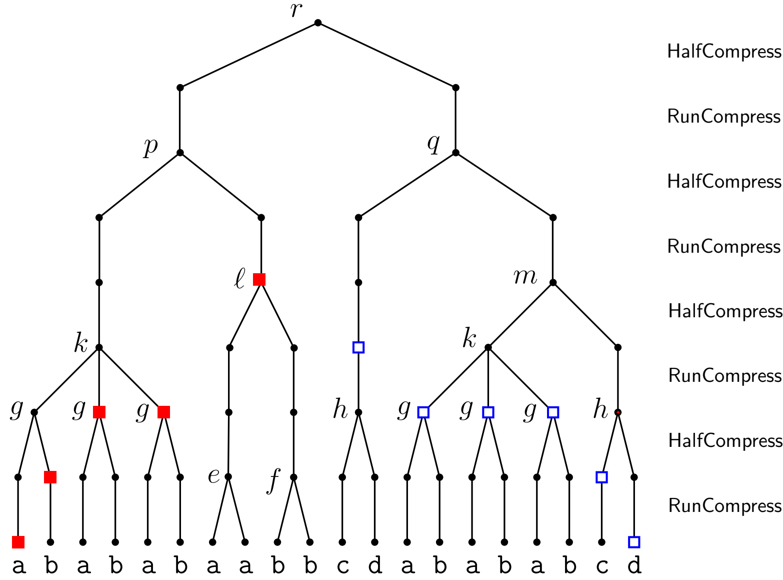


Figure 6.1: An example $\mathsf{UPT}[T]$ for $T = T_0 = \mathtt{abababaabbcdabababcd}$. We omit the label of each node v with a single child u; $\mathcal{L}(v) = \mathcal{L}(u)$. $T_3 = kefhkh$ and $T_6 = pq$. We denote the nodes $\mathsf{C}_{\mathsf{up}}(T)$ by red (filled) squares and the nodes of $\mathsf{C}_{\mathsf{down}}(T)$ with blue (unfilled) squares. $\mathsf{d}_{\mathsf{up}}(T) = \mathtt{ab}g^2\ell$, $\mathsf{d}_{\mathsf{down}}(T) = hg^3\mathtt{cd}$ and hence $\mathsf{d}(T) = \mathtt{ab}g^2\ell hg^3\mathtt{cd}$.

Then, Theorem 4.1.1 states that $\mathsf{d}(U) = \mathsf{d}(V)$ for any fragment V of any string in the collection such that $U = V$. We will use $\mathsf{d}(\cdot)$ for substrings and not just for fragments.

Let us consider any sequence of nodes corresponding, for some $j < m$, to $A_j^{r_j}$ with $r_j > 1$ or $B_j^{t_j}$ with $t_j > 1$. We note that T_j must have been obtained from T_{j-1} by an application of $\mathsf{HalfCompress}$, since there are no RLE runs of length greater than one after an application of procedure $\mathsf{RunCompress}$. Thus, at level $j+1$ in $\mathsf{UPT}[T]$, i.e. the one corresponding to T_{j+1}, all of these nodes collapse to a single one: their parent in $\mathsf{UPT}[T]$. Hence, we have the following lemma.

Lemma 6.3.1. *Let us consider a fragment U of T with $\mathsf{d}_{\mathsf{up}}(U) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ and $\mathsf{d}_{\mathsf{down}}(U) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$. Then we have the following:*

- *The value of $\mathsf{C}_{\mathsf{up}}(U)$ is a suffix of the value of a layer of (at most) $c \log n + r_m - 1$ level-m nodes, such that the two layers have the same rightmost node. The last r_m nodes of both layers are consecutive siblings with label A_m.*
- *The value of $\mathsf{C}_{\mathsf{down}}(U)$ is a prefix of the value of the layer consisting of the subsequent (at most) $c \log n + \max(t_m - 1, 0)$ level-m nodes. If $t_m \neq 0$, then the first t_m nodes of both of these layers are consecutive siblings with label $B_m \neq A_m$.*

While the (uncompressed) parse trees of the strings in the collection are not maintained explicitly, we have access to the following pointers and functions, among others, which allow us to efficiently navigate through them. First, we can get a pointer to the root of $\mathsf{UPT}[T]$ for any string T in the collection. Given a pointer to some node v in $\mathsf{UPT}[T]$ we can get $\deg(v)$ and pointers to the parent of v, the k-th child of v and the nodes to the left/right of v.

Let us now briefly explain how the dynamic data structure of [82] processes a substitution in T at some position i, that yields a string T'. First, the $\mathsf{C}(T[1\,.\,.\,i-1])$ and $\mathsf{C}(T[i+1\,.\,.\,|T|])$ are retrieved. These, together with the new letter at position i form a layer of $\mathsf{UPT}[T']$. The sequence of the labels of the nodes of this layer can be expressed as a sequence of $\mathcal{O}(\log n)$ symbols and powers of symbols. Then, only the portion of $\mathsf{UPT}[T]$ that lies above this layer needs to be (implicitly) computed, and the authors of [82] show how to do this in $\mathcal{O}(\log n)$ time. In total, we get $\mathsf{UPT}[T']$ from $\mathsf{UPT}[T]$ through $\mathcal{O}(\log^2 n)$ insertions and deletions of nodes and layers that consist of consecutive

siblings. Insertions and deletions of letters in T can be processed analogously in $\mathcal{O}(\log n)$ time, and also result in $\mathcal{O}(\log^2 n)$ insertions and deletions of nodes and layers that consist of consecutive siblings.

6.3.2 Anchoring the LCF

We maintain S and T as they undergo edit operations using Theorem 1.3.1. We will rely on Lemma 6.3.1 in order to identify an LCF $S[i \,..\, j] = T[i' \,..\, j']$ at a pair of topmost nodes of $\mathsf{C}(S[i \,..\, j])$ and $\mathsf{C}(T[i' \,..\, j'])$ in $\mathsf{UPT}[S]$ and $\mathsf{UPT}[T]$, respectively. In order to develop some intuition, let us first sketch a solution for the case where $\mathsf{UPT}[S]$ and $\mathsf{UPT}[T]$ do not contain any power symbols throughout the execution of our algorithm. For each node v in one of the parse trees, let $\mathsf{Z}_\ell(v)$ be the value of the layer consisting of the (at most) $c \log n$ level-$\mathsf{lev}(v)$ nodes, with v being the layer's rightmost node, and $\mathsf{Z}_\mathsf{r}(v)$ be the value of the layer consisting of the (at most) $c \log n$ subsequent level-$\mathsf{lev}(v)$ nodes. Now, consider a common substring X of S and T and partition it into the prefix $X_\ell = \mathsf{gen}(\mathsf{d}_\mathsf{up}(X))$ and the suffix $X_r = \mathsf{gen}(\mathsf{d}_\mathsf{down}(X))$. For any fragment U of S that equals X, $\mathsf{C}_\mathsf{up}(U)$ is an up-layer of the form $v_1 \cdots v_m$. Hence, by Lemma 6.3.1, X_ℓ is a suffix of $\mathsf{Z}_\ell(v_m)$. Similarly, X_r is a prefix of $\mathsf{Z}_\mathsf{r}(v_m)$. Thus, it suffices to maintain pairs $(\mathsf{Z}_\ell(v), \mathsf{Z}_\mathsf{r}(v))$ for all nodes v in $\mathsf{UPT}[S]$ and $\mathsf{UPT}[T]$, and, in particular, a pair of nodes $u \in \mathsf{UPT}[S]$ and $v \in \mathsf{UPT}[T]$ that maximises $\mathsf{LCP}(\mathsf{Z}_\ell(u)^R, \mathsf{Z}_\ell(v)^R) + \mathsf{LCP}(\mathsf{Z}_\mathsf{r}(u), \mathsf{Z}_\mathsf{r}(v))$. The existence of power symbols poses some technical challenges which we overcome below.

For each node of $\mathsf{UPT}[T]$, we consider at most one pair consisting of an up-layer and a down-layer. The treatment of nodes differs, based on their parent. We have two cases.

1. For each node z with $\deg(z) = 2$ and $\mathcal{L}(z)$ being a concatenation symbol, for each child v of z, we consider the following layers:

   - The layer $\mathsf{J}_\mathsf{up}(v)$ of the (at most) $c \log n$ level-$\mathsf{lev}(v)$ consecutive nodes of $\mathsf{UPT}[T]$ with v a rightmost node.

- The layer $\mathsf{J}_{\mathsf{down}}(v)$ of the (at most) $c \log n + \deg(w)$ subsequent level-$\mathsf{lev}(v)$ nodes of $\mathsf{UPT}[T]$, where w is the parent of the node to the right of v.

2. For each node z of $\mathsf{UPT}[T]$ whose label is a power symbol and has more than one child, we will consider $\mathcal{O}(\log n)$ pairs of layers. In particular, for each v, being one of the $c \log n + 1$ leftmost or $c \log n + 1$ rightmost children of z, we consider the following layers:

   - The layer $\mathsf{J}_{\mathsf{up}}(v)$ consisting of (a) the (at most) $c \log n$ level-$\mathsf{lev}(v)$ consecutive nodes of $\mathsf{UPT}[T]$ preceding the leftmost child of z and (b) all the children of z that lie weakly to the left of v, i.e. including v.
   - The layer $\mathsf{J}_{\mathsf{down}}(v)$ consisting of the (at most) $c \log n$ subsequent level-$\mathsf{lev}(v)$ nodes of $\mathsf{UPT}[T]$—with one exception. If v is the rightmost child of z and the node to its right is a child of a node w with more than two children, then $\mathsf{J}_{\mathsf{down}}(v)$ consists of the $c \log n + \deg(w)$ subsequent level-$\mathsf{lev}(v)$ nodes.

In particular, we create at most one pair $(\mathsf{J}_{\mathsf{up}}(v), \mathsf{J}_{\mathsf{down}}(v))$ of layers for each node v of $\mathsf{UPT}[T]$. Let $\mathsf{Y}_{\ell}(v) = \mathsf{val}(\mathsf{J}_{\mathsf{up}}(v))$ and $\mathsf{Y}_{\mathsf{r}}(v) = \mathsf{val}(\mathsf{J}_{\mathsf{down}}(v))$. Given a pointer to a node z in $\mathsf{UPT}[T]$, we can compute the indices of the fragments corresponding to those layers with a straightforward use of the pointers at hand in $\mathcal{O}(\log n)$ time. With a constant number of split operations, we can then add the string $\mathsf{Y}_{\mathsf{r}}(v)$ to our collection within $\mathcal{O}(\log n)$ time. Similarly, if we also maintain T^R in our collection of strings, we can add the reverse of $\mathsf{Y}_{\ell}(v)$ to the collection within $\mathcal{O}(\log n)$ time. We maintain pointers between v and these strings. Note that each node of $\mathsf{UPT}[T]$ takes part in $\mathcal{O}(\log n)$ pairs of layers and these pairs can be retrieved in $\mathcal{O}(\log n)$ time. Similarly, for each node whose label is a power symbol, subsets of its children appear in $\mathcal{O}(\log n)$ pairs of layers; these can also be retrieved in $\mathcal{O}(\log n)$ time. These pairs of layers (or rather the pairs of their corresponding strings maintained in a dynamic collection) will be stored in an abstract structure presented in the next subsection.

Recall that each update on T is processed in $\mathcal{O}(\log n)$ time, while it deletes and inserts $\mathcal{O}(\log^2 n)$ nodes and layers of consecutive siblings. Each of those inserted/deleted nodes

and layers affects $\mathcal{O}(\log n)$ pairs of layers as described above, for a total of $\mathcal{O}(\log^3 n)$. The total time required to add/remove the affected layers is thus $\mathcal{O}(\log^4 n)$. We summarise the above discussion in the following lemma.

Lemma 6.3.2. *We can maintain pairs $(\mathsf{Y}_\ell(v)^R, \mathsf{Y}_\mathsf{r}(v))$ for all v in $\mathsf{UPT}[T]$ and $\mathsf{UPT}[S]$, with each string given as a handle from the dynamic collection, in $\mathcal{O}(\log^4 n)$ time w.h.p. per edit operation. The number of deleted and inserted pairs per edit operation is $\mathcal{O}(\log^3 n)$ w.h.p.*

The following lemma gives us an anchoring property, which is crucial for our approach.

Lemma 6.3.3. *For any common substring X of S and T, there exists a partition $X = X_\ell X_r$ for which there exist nodes $u \in \mathsf{UPT}[S]$ and $v \in \mathsf{UPT}[T]$ such that:*

1. *X_ℓ is a suffix of $\mathsf{Y}_\ell(u)$ and $\mathsf{Y}_\ell(v)$, and*

2. *X_r is a prefix of $\mathsf{Y}_\mathsf{r}(u)$ and $\mathsf{Y}_\mathsf{r}(v)$.*

Proof. Let $\mathsf{d_{up}}(X) = A_0^{r_0} A_1^{r_1} \cdots A_m^{r_m}$ and $\mathsf{d_{down}}(X) = B_m^{t_m} B_{m-1}^{t_{m-1}} \cdots B_0^{t_0}$.

Claim 6.3.4. *Either $r_m > 1$, $t_m = 0$, and $\mathsf{gen}(\mathsf{d_{up}}(X))$ is not a suffix of $A_m^{c\log n + r_m}$, or there exists a node $v \in \mathsf{UPT}[T]$ such that:*

1. *$\mathsf{gen}(\mathsf{d_{up}}(X))$ is a suffix of $\mathsf{Y}_\ell(v)$, and*

2. *$\mathsf{gen}(\mathsf{d_{down}}(X))$ is a prefix of $\mathsf{Y}_\mathsf{r}(v)$.*

Proof. We assume that $r_m = 1$ or $\mathsf{gen}(\mathsf{d_{up}}(X))$ is a suffix of $A_m^{c\log n + r_m}$ or $t_m \neq 0$ and distinguish between the following cases.

Case 1. There exists an occurrence Y of X in T, where the label of the parent of the rightmost node u of $\mathsf{C_{up}}(Y)$ is not a power symbol. (In this case $r_m = 1$.) Recall here, that we did not construct any pairs of layers for nodes whose parent has a single child. Let v be the lowest ancestor of u with label A_m. If $u \neq v$ then all nodes that are descendants of v and strict ancestors of u have a single child, while the parent of v does not. In addition, the label of the parent of v must be a concatenation symbol, since only

new letters are introduced at each level and thus we cannot have new nodes with label A_m appearing to the left/right of any strict ancestor of u. Finally, note that a layer of k level-$\mathsf{lev}(v)$ nodes with v a leftmost (resp. rightmost) node contains an ancestor of each of the nodes in a layer of k level-$\mathsf{lev}(u)$ nodes with u a leftmost (resp. rightmost) node. Thus, an application of Lemma 6.3.1 for u straightforwardly implies our claim for v.

Case 2. There exists an occurrence Y of X in T, where the label of the parent z of the rightmost node u of $\mathsf{C}_{\mathsf{up}}(Y)$ is a power symbol. Let W be the rightmost occurrence of X in T with the rightmost node w of $\mathsf{C}_{\mathsf{up}}(W)$ being a child of z. We have three subcases.

(a) We first consider the case $r_m = 1$. Let us assume towards a contradiction that u is not one of the $c \log n + 1$ leftmost or the $c \log n + 1$ rightmost children of z. Then, by Lemma 6.3.1 we have that $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $A_m^{c \log n}$ and $\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(X))$ is a prefix of $A_m^{c \log n}$. Hence, there is another occurrence of X $|\mathsf{gen}(A_m)|$ positions to the right of Y, contradicting our assumption that Y is a rightmost occurrence.

(b) In the case where $t_m \neq 0$, u must be the rightmost child of z since $A_m \neq B_m$.

(c) In the remaining case where $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is a suffix of $A_m^{c \log n + r_m}$, either $t_m > 0$ and we are done, or $\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(Y))$ is a prefix of the value of the (at most) $c \log n$ level-m nodes to the right of u. In the latter case, either u is already among the rightmost $c \log n + 1$ children of z or there is another occurrence of X $|\mathsf{gen}(A_m)|$ positions to the right of Y, contradicting our assumptions on Y. □

We have to treat a final case.

Claim 6.3.5. *If $r_m > 1$, $t_m = 0$ and $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is not a suffix of $A_m^{c \log n + r_m}$ then there exists a node $v \in \mathsf{UPT}[T]$ such that:*

1. *$\mathsf{gen}(A_0^{r_0} A_1^{r_1} \cdots A_{m-1}^{r_{m-1}} A_m)$ is a suffix of $\mathsf{Y}_\ell(v)$, and*

2. *$\mathsf{gen}(A_m^{r_m - 1})\mathsf{gen}(\mathsf{d}_{\mathsf{down}}(X))$ is a prefix of $\mathsf{Y}_\mathsf{r}(v)$.*

Proof. In any occurrence Y of X in T, the label of the parent z of the rightmost node of $\mathsf{C}_{\mathsf{up}}(Y)$ is a power symbol. Let u be the r_m-th rightmost node of $\mathsf{C}_{\mathsf{up}}(Y)$. By the

assumption that $\mathsf{gen}(\mathsf{d}_{\mathsf{up}}(X))$ is not a suffix of $A_m^{c\log n+r_m}$ and Lemma 6.3.1, u must be one of the $c\log n$ leftmost children of z. $\square$

The combination of the two claims applied to both S and T yields the lemma. $\square$

6.3.3 A Problem on Dynamic Bicoloured Trees

Due to Lemmas 6.3.2 and 6.3.3, our task reduces to solving the problem defined below in polylogarithmic time per update, as we can directly apply it to $\mathcal{R} = \{(\mathsf{Y}_\ell(u)^R, \mathsf{Y}_\mathsf{r}(u)) : u \in \mathsf{UPT}[S]\}$ and $\mathcal{B} = \{(\mathsf{Y}_\ell(v)^R, \mathsf{Y}_\mathsf{r}(v)) : v \in \mathsf{UPT}[T]\}$. Note that $|\mathcal{R}| + |\mathcal{B}| = \tilde{\mathcal{O}}(n)$ throughout the execution of our algorithm.

LCP for Two Families of Pairs of Strings
Input: Two families $\mathcal{R}$ and $\mathcal{B}$, each consisting of pairs of strings, where each string is given as a handle from a dynamic collection.
Update: Insertion or deletion of an element in $\mathcal{R}$ or $\mathcal{B}$.
Query: Return $(P,Q) \in \mathcal{R}$ and $(P',Q') \in \mathcal{B}$ that maximise $\mathsf{LCP}(P,P')+\mathsf{LCP}(Q,Q')$.

Each element of $\mathcal{B}$ and $\mathcal{R}$ is given a unique identifier. We maintain two compact tries $\mathcal{T}_P$ and $\mathcal{T}_Q$. By appending unique letters, we can assume that no string is a prefix of another string. $\mathcal{T}_P$ (resp. $\mathcal{T}_Q$) stores the string P (resp. Q) for every $(P,Q) \in \mathcal{R}$, with the corresponding leaf coloured red and labelled by the identifier of the pair and the string P' (resp. Q') for every $(P',Q') \in \mathcal{B}$, with the corresponding leaf coloured blue and labelled by the identifier of the pair. Then, the sought result corresponds to a pair of nodes $u \in \mathcal{T}_P$ and $v \in \mathcal{T}_Q$ returned by a query to a data structure for the Dynamic Bicoloured Trees Problem defined below for $\mathcal{T}_1 = \mathcal{T}_P$ and $\mathcal{T}_2 = \mathcal{T}_Q$, with node weights being their string-depths.

Dynamic Bicoloured Trees Problem
Input: Two weighted trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of total size at most m, whose leaves are bicoloured and labelled, so that each label is an integer in $\mathcal{O}(m)$ and corresponds to exactly one leaf of each tree.
Update: Split an edge into two, attach a new leaf to a node, or delete a leaf.
Query: Return a pair of nodes $u \in \mathcal{T}_1$ and $v \in \mathcal{T}_2$ with the maximum combined weight that have at least one red descendant with the same label, and at least one blue descendant with the same label.

Remark 6.3.6. A static version of this problem has been used for approximate LCF under the Hamming distance, e.g. in [40, 140].

To complete the reduction, we have to show how to translate an update in $\mathcal{R}$ or $\mathcal{B}$ into updates in $\mathcal{T}_P$ and $\mathcal{T}_Q$. Let us first explain how to represent $\mathcal{T}_P$ and $\mathcal{T}_Q$. For each edge, we store a handle to a string from the dynamic collection, and indices for a fragment of this string which represents the edge's label. For each explicit node, we store edges leading to its children in a dictionary structure indexed by the first letters of the edges' labels. For every leaf, we store its label and colour. An insert operation receives a string (given as a handle from a dynamic collection), together with its label and colour, and should create its corresponding leaf. A delete operation does not actually remove a leaf, but simply removes its label. However, in order to not increase the space complexity, we rebuild the whole data structure from scratch after every m updates. This rebuilding does not incur any extra cost asymptotically; the time required for it can be deamortised using the time-slicing technique (cf. the proof of Lemma 2.8.1).

Lemma 6.3.7. *Each update in $\mathcal{R}$ or $\mathcal{B}$ implies $\mathcal{O}(1)$ updates in $\mathcal{T}_P$ and $\mathcal{T}_Q$ that can be computed in $\mathcal{O}(\log n)$ time.*

Proof. Inserting a new leaf, corresponding to string U, to $\mathcal{T}_P$ requires possibly splitting an edge into two by creating a new explicit node, and then attaching a new leaf to an explicit node. To implement this efficiently, we maintain the set C of path-labels of explicit nodes of $\mathcal{T}_P$ in a balanced search tree, sorted in lexicographic order. Using LCP

queries (cf. Theorem 1.3.1), we binary search for the longest prefix U' of U that equals the path-label of some implicit or explicit node of $\mathcal{T}_P$. If this node is explicit, then we attach a leaf to it. Otherwise, let the successor of U' in C be the path-label of node v. We split the edge $(\mathsf{parent}(v), v)$ appropriately and attach a leaf to the newly created node. This allows us to maintain $\mathcal{T}_P$ after each insert operation in $\mathcal{O}(\log n)$ time.

For a delete operation, we can access the leaf corresponding to the deleted string in $\mathcal{O}(\log n)$ time using the balanced search tree. □

It thus suffices to show a solution for the Dynamic Bicoloured Trees Problem that processes each update in polylogarithmic time.

We will maintain a heavy-light decomposition of both $\mathcal{T}_1$ and $\mathcal{T}_2$. This can be done by using a standard method of rebuilding as used by Gabow [74]. Let $L(u)$ be the number of leaves in the subtree of u, including the leaves without labels, when the subtree was last rebuilt. Each internal node u of a tree selects at most one child v and the edge (u, v) is *heavy*. All other edges are *light*. Maximal sequences of consecutive heavy edges are called *heavy paths*. The node $r(p)$ closest to the root of the tree is called the *root* of the heavy path p and the node $e(p)$ furthest from the root of the tree is called the *end* of the heavy path. The following procedure receives a node u of the tree and recursively rebuilds the heavy paths in its subtree.

```
1: function DECOMPOSE(u, r)            ▷ r is the root of the heavy path containing u.
2:     S ← children(u)
3:     v ← argmax_{v∈S} L(v)
4:     if L(v) ≥ 5/6 · L(u) then
5:         edge (u, v) is heavy
6:         DECOMPOSE(v, r)
7:         S ← S \ {v}
8:     for v ∈ S do
9:         DECOMPOSE(v, v)
```

Every root u of a heavy path maintains the number of insertions $I(u)$ in its subtree

since it was last rebuilt. When $I(u) \geq \frac{1}{6} \cdot L(u)$, we recalculate the values of $L(v)$ for nodes v in the subtree of u and call DECOMPOSE(u, u). This maintains the property that $L(e(p)) \geq \frac{2}{3}L(r(p))$ for each heavy path p and leads to the following.

Proposition 6.3.8. *There are $\mathcal{O}(\log m)$ heavy paths above any node.*

As rebuilding a subtree of size s takes $\mathcal{O}(s)$ time, by a standard potential argument, we get the following. (The bottleneck is in updating $I(u)$s.)

Lemma 6.3.9. *The heavy-light decompositions of $\mathcal{T}_1$ and $\mathcal{T}_2$ can be maintained in $\mathcal{O}(\log m)$ amortised time per update.*

The main ingredient of our data structure is a collection of additional data structures, each storing a dynamic set of points. Each such point structure sends its current result to a max-heap, and after each update we return the largest element stored in the heap. The problem each of these point structures are designed for is the following.

DYNAMIC BEST BICHROMATIC POINT

Input: A multiset of at most m bicoloured points from $[m] \times [m]$.

Update: Insertions and deletions of points from $[m] \times [m]$.

Query: Return a pair of points $R = (x, y)$ and $B = (x', y')$ such that R is red, B is blue, and $\min(x, x') + \min(y, y')$ is as large as possible.

We call the pair of points sought in this problem the *best bichromatic pair of points.* In Section 6.3.4, we show the following result by modifying 2D range trees.

Lemma 6.3.10. *There is a data structure for* DYNAMIC BEST BICHROMATIC POINT *that processes each update in $\mathcal{O}(\log^2 m)$ amortised time.*

Conceptually, we maintain a point structure for every pair of heavy paths from $\mathcal{T}_P$ and $\mathcal{T}_Q$. However, the total number of points stored in all point structures at any moment is only $\mathcal{O}(m \log^2 m)$ and the empty structures are not actually created. Consider heavy paths p of $\mathcal{T}_1$ and q of $\mathcal{T}_2$. Let ℓ be a label such that there are leaves u in the subtree of $r(p)$ in $\mathcal{T}_1$ and v in the subtree of $r(q)$ in $\mathcal{T}_2$ with the same colour and both labelled by ℓ.

Then, the point structure should contain a point (x, y) with this colour, where x and y are the string-depths of the nodes of p and q containing u and v in their light subtrees, respectively. It can be verified then that the answer extracted from the point structure is equal to the sought result, assuming that the corresponding pair of nodes belongs to p and q, respectively. It remains to explain how to maintain this invariant when both trees undergo modifications.

Splitting an edge does not require any changes to the point structures. Each label appears only once in $\mathcal{T}_1$ and $\mathcal{T}_2$, and hence by Proposition 6.3.8 contributes to only $\mathcal{O}(\log^2 n)$ point structures. Furthermore, by navigating the heavy path decompositions we can access these point structures efficiently. This allows us to implement each deletion in $\mathcal{O}(\log^4 n)$ amortised time, employing Lemma 6.3.10. To implement the insertions, we need to additionally explain what to do after rebuilding a subtree of u. In this case, we first remove all points corresponding to leaves in the subtree of u, then rebuild the subtree, and then proceed to insert points to existing and potentially new point structures. As each insertion affects $\mathcal{O}(\log n)$ heavy paths, it affects $\mathcal{O}(\log n)$ rebuilding instances. By the same standard potential argument as above, each insertion costs $\mathcal{O}(\log^4 n)$ amortised time per such instance: we add a point, in $\mathcal{O}(\log^2 n)$ time, in $\mathcal{O}(\log^2 n)$ point structures. Hence insertions require $\mathcal{O}(\log^5 n)$ amortised time.

Wrap-up. Lemma 6.3.3 reduces our problem to the LCP for Two Families of Pairs of Strings problem for sets $\mathcal{R}$ and $\mathcal{B}$ of size $\tilde{\mathcal{O}}(n)$, so that each edit operation in S or T yields $\mathcal{O}(\log^3 n)$ updates to $\mathcal{R}$ and $\mathcal{B}$, which can be performed in $\mathcal{O}(\log^4 n)$ time due to Lemma 6.3.2. The LCP for Two Families of Pairs of Strings problem is then reduced to the Dynamic Bicoloured Trees Problem for trees $\mathcal{T}_1$ and $\mathcal{T}_2$ of size $\tilde{\mathcal{O}}(n)$, so that each update in $\mathcal{R}$ or $\mathcal{B}$ yields $\mathcal{O}(1)$ updates to the trees, which can be computed in $\mathcal{O}(\log n)$ time (Lemma 6.3.7). We solve the latter problem by maintaining a heavy-light decomposition of each of the trees in $\mathcal{O}(\log n)$ amortised time per update (Lemma 6.3.9), and an instance of a data structure for the Dynamic Best Bichromatic Point problem for each pair of heavy paths. For each update to the

trees, we spend $\mathcal{O}(\log^5 n)$ amortised time to update the point structures. Thus, each update in one of the strings costs a total of $\mathcal{O}(\log^8 n)$ amortised time. Finally, note that we can employ Lemma 2.8.1 to keep the space usage of our data structure $\tilde{\mathcal{O}}(n)$. (Here, the preprocessing step would be to employ $\mathcal{O}(n)$ edit operations on two empty strings in order to obtain S and T.)

6.3.4 Dynamic Best Bichromatic Point

In this section we prove Lemma 6.3.10, i.e. we design an efficient data structure for the DYNAMIC BEST BICHROMATIC POINT problem.

First, we show that we can assume that all x and y coordinates of points are distinct. Let us replace identical points of the same colour with a single point, with which we store its multiplicity as satellite information. Then, we perform the following standard perturbation. Namely, we can (implicitly) replace each red (resp. blue) point (x, y) with $((x|y|0), (y|x|0))$ (resp. $((x|y|1), (y|x|1))$), and use the lexicographic order to perform comparisons for each coordinate (cf [62, Section 5.5]). As the data structures that we use are comparison based, the above transformation does not affect the complexities.

We maintain an augmented dynamic 2D range tree [172] over the multiset of points. This is a balanced search tree $\mathcal{T}$ (called primary) over the x coordinates of all points in the multiset in which every x coordinate corresponds to a leaf and, more generally, every node $u \in \mathcal{T}$ corresponds to a range of x coordinates denoted by $x(u)$. Additionally, every $u \in \mathcal{T}$ stores another balanced search tree $\mathcal{T}_u$ (called secondary) over the y coordinates of all points $(x, y) \in S$ such that $x \in x(u)$. Thus, the leaves of $\mathcal{T}_u$ correspond to y coordinates of such points, and every $v \in \mathcal{T}_u$ corresponds to a range of y coordinates denoted by $y(v)$. We interpret every $v \in \mathcal{T}_u$ as the rectangular region of the plane $x(u) \times y(v)$, and, in particular, each leaf $v \in \mathcal{T}_u$ corresponds to a single point in the multiset. Each node $v \in \mathcal{T}_u$ will be augmented with some extra information that can be computed in constant time from the extra information stored in its children. Similarly, each node $u \in \mathcal{T}$ will be augmented with some extra information that can be computed in constant time from the extra information stored in its children together with the extra

information stored in the root of the secondary tree $\mathcal{T}_u$. Irrespectively of what this extra information is, as explained by Willard and Lueker [172], if we implement the primary tree as a $BB(\alpha)$ tree and each secondary tree as a balanced search tree, each insertion and deletion can be implemented in $\mathcal{O}(\log^2 m)$ amortised time.

Before we explain what is the extra information, we need the following notion. Consider a non-leaf node $u \in \mathcal{T}$ and let $u_\ell, u_r \in \mathcal{T}$ be its children. Let $v \in \mathcal{T}_u$ be a non-leaf node with children $v_\ell, v_r \in \mathcal{T}_u$. The regions $A = x(u_\ell) \times y(v_\ell)$, $B = x(u_\ell) \times y(v_r)$, $C = x(u_r) \times y(v_\ell)$ and $D = x(u_r) \times y(v_r)$ partition $x(u) \times y(v)$ into four parts. We say that two points $p = (x, y)$ and $q = (x', y')$ with $x < x'$ are shattered by $v \in \mathcal{T}_u$ if and only if $p \in A$ and $q \in D$ or $p \in B$ and $q \in C$ (note that the former is only possible when $y < y'$ while the latter can only hold when $y > y'$). See Figure 6.2 for an illustration.

Proposition 6.3.11. *Any pair of points in the multiset is shattered by a unique $v \in \mathcal{T}_u$ (for a unique u).*

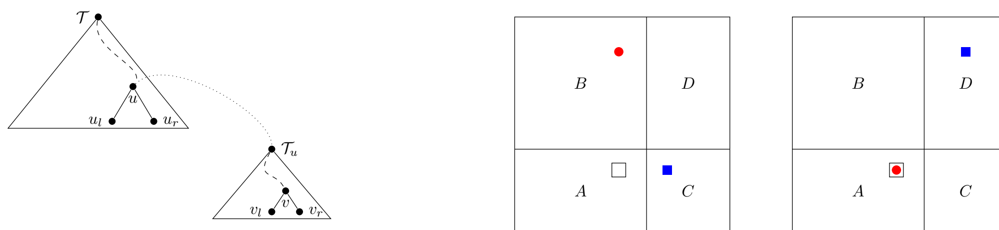


Figure 6.2: Left: A 2D range tree. Right: The regions A, B, C, D; the best bichromatic point for each case is denoted by a small square.

Now we are ready to describe the extra information. Each node $u \in \mathcal{T}$ stores the best bichromatic pair with x coordinates from $x(u)$. Each node $v \in \mathcal{T}_u$ stores the best bichromatic pair shattered by one of its descendants $v' \in \mathcal{T}_u$ (possibly v itself). Additionally, each node $v \in \mathcal{T}_u$ stores the following information about points of each colour in its region:

1. the point with the maximum x,

2. the point with the maximum y,

3. a point with the maximum $x + y$.

We need to verify that such extra information can be indeed computed in constant time from the extra information stored in the children.

Lemma 6.3.12. *Let $v \in \mathcal{T}_u$ be a non-leaf node, and v_ℓ, v_r be its children. The extra information of v can be computed in constant time given the extra information stored in v_ℓ and v_r.*

Proof. This is clear for the maximum x, y and $x + y$ of each colour, as we can take the maximum of the corresponding values stored in the children. For the best bichromatic pair shattered by a descendant v' of v, we start with considering the best bichromatic pair shattered by a descendant v'_ℓ of v_ℓ and v'_r of v_r. The remaining case is that the best bichromatic pair is shattered by v itself. Let A, B, C, D be as in the definition of shattering. Without losing generality we assume that the sought pair is $p = (x, y)$ and $q = (x', y')$ with $x < x'$, red p and blue q. We consider two cases:

1. $p \in A$ and $q \in D$: the best such pair is obtained by taking p with the maximum $x + y$ and any q,

2. $p \in B$ and $q \in C$: the best such pair is obtained by taking p with the maximum x and q with the maximum y.

In both cases, we are able to compute the best bichromatic pair shattered by v using the extra information stored at the children of v. See Figure 6.2. $\square$

Lemma 6.3.13. *Let $u \in \mathcal{T}$ be a non-leaf node, and u_ℓ, u_r be its children. The extra information of u can be computed in constant time given the extra information stored in u_ℓ, u_r and the root of $\mathcal{T}_u$.*

Proof. We seek the best bichromatic pair with x coordinates from $x(u)$. If the x coordinates are in fact from $x(u_\ell)$ or $x(u_r)$, we obtain the pair from the children of u. Otherwise, the pair must be shattered by some $v \in \mathcal{T}_u$ that is a descendant of the root of $\mathcal{T}_u$, so we obtain the pair from the root of $\mathcal{T}_u$. $\square$

Chapter 7

Dynamic String Alignment

Our first algorithm heavily relies on Tiskin's work on efficient distance multiplication of simple unit-Monge matrices and its applications to the string alignment problem. Specifically for the LCS problem, Tiskin showed that the semi-local LCS (a restricted variant of internal LCS) information of two strings of length at most n can be efficiently retrieved from an $\tilde{\mathcal{O}}(n)$-size representation as a *permutation matrix* $P_{S,T}$. Based on his efficient algorithm for computing the $(\min, +)$-product of two simple unit-Monge matrices, he showed that given permutation matrices $P_{S,T}$ and $P_{S,T'}$, one can efficiently compute $P_{S,TT'}$. We formalise this in Section 7.1.

In Section 7.2, we first describe our algorithm for maintaining an LCS of two strings S and T in $\tilde{\mathcal{O}}(n)$ time per edit operation, and then we extend it to maintaining a string alignment under integer weights. Our algorithm maintains a hierarchical partition of strings S and T to fragments of length roughly 2^s for each scale $s \in [0, \log n]$, and permutation matrices P_{S_i,T_j} for all pairs of fragments (S_i, T_j) at each scale. Then, upon an update to S or T, we need to update $\Theta(n/2^s)$ permutation matrices at each scale s. This is in contrast with the sequential approach of combining the permutation matrices in Tiskin's work.

In Section 7.3, we show that efficient data structures for computing distances in planar graphs outperform the approach outlined above when the alignment weights cannot be expressed as small integers.

7.1 The Alignment Graph and Internal LCS

A *longest common subsequence* (LCS) of two strings S and T is a longest string that is a subsequence of both S and T. We denote the length of an LCS of S and T by $\texttt{LCS}(S,T)$.

Example 7.1.1. An LCS of $S = $ `acbcddaaea` and $T = $ `abbbccdec` is `abcde`; $\texttt{LCS}(S,T) = 5$.

The Alignment Graph. For strings S and T, of length m and n respectively, the alignment graph $G_{S,T}$ of S and T is a directed acyclic graph with vertex set $\{v_{i,j} : 0 \leq i \leq m, 0 \leq j \leq n\}$. For every $0 \leq i \leq m$ and $0 \leq j \leq n$, the graph $G_{S,T}$ has the following edges (defined only if both endpoints exist):

- $v_{i,j}v_{i+1,j}$ and $v_{i,j}v_{i,j+1}$ of length 0,
- $v_{i,j}v_{i+1,j+1}$ of length 1, present if and only if $S[i+1] = T[j+1]$.

Intuitively, $G_{S,T}$ is an $(m+1) \times (n+1)$ grid graph (with length-0 edges) augmented with length-1 diagonal edges corresponding to matching letters of S and T. We think of the vertex $v_{0,0}$ as the top left vertex of the grid and the vertex $v_{m,n}$ as the bottom right vertex of the grid. We shall refer to the rows and columns of $G_{S,T}$ in a natural way. It is easy to see that $\texttt{LCS}(S,T)$ equals the length of the highest scoring path between $v_{0,0}$ and $v_{m,n}$ in $G_{S,T}$. An illustration is presented in Figure 7.1.

Internal LCS. Let us now briefly discuss internal LCS queries. In our reduction from the Set Disjointness problem to the internal LCF problem (Theorem 6.1.2), we constructed two strings and reduced each Set Disjointness query to checking whether some (internal) LCF is of length at least 1. This reduction can be straightforwardly adapted to give a reduction from the Set Disjointness problem to the internal LCS problem, since, for any two strings U and V, $\texttt{LCF}(U,V) \geq 1$ if and only if $\texttt{LCS}(U,V) \geq 1$. As we did in Section 6.1.2 for the LCF problem, Tiskin considered an analogously restricted variant of the internal LCS problem, called *semi-local* LCS, for which he obtained an $\tilde{\mathcal{O}}(n)$-size data structure, that can be constructed in $\tilde{\mathcal{O}}(n^2)$ time, and answers queries in $\mathcal{O}(\log n/\log\log n)$ time. We will discuss this data structure in detail below.

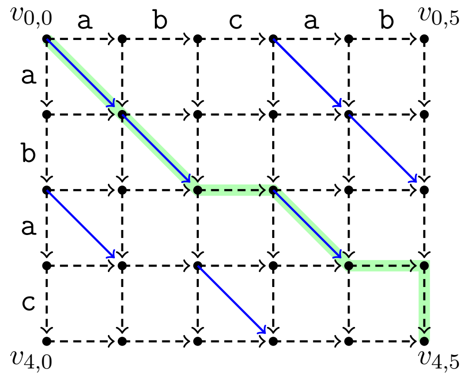


Figure 7.1: The alignment graph for $S = \texttt{abac}$ and $T = \texttt{abcab}$. We represent 0-length edges by dashed black arrows, and 1-length edges by blue arrows. A highest scoring $v_{0,0}$-to-$v_{4,5}$ path is highlighted in green, and corresponds to the LCS aba of length three.

As for the general internal LCS problem, Sakai, building on Tiskin's work, showed how to construct in $\mathcal{O}(n^2)$ time a data structure answering queries in $\mathcal{O}(n)$ time [151]. Furthermore, observe that the alignment graph is planar, and that, by slightly modifying the edge-weights, we can end up with the problem of computing lowest scoring paths instead of highest scoring ones; see Section 7.3 for details. Thus, we can build a distance oracle for planar graphs over the alignment graph, and answer each internal LCS query by querying this oracle. For instance, by using the state-of-the-art distance oracle of Long and Pettie [125], which was obtained by building upon [80, 41], we obtain an $n^{2+o(1)}$-size data structure that answers queries in $\log^{2+o(1)} n$ time; this data structure can be built in $\mathcal{O}(n^{3+\epsilon})$, for any constant $\epsilon > 0$. An interesting research direction is to investigate the tradeoff between the construction time and the query time for the internal LCS problem—as well as for distance oracles for planar graphs.[1]

[1]After the submission of this thesis, we have presented an almost optimal data structure for the internal LCS problem [43].

Simple unit-Monge Matrices and Semi-local LCS. We index matrices from 0. Let us define some matrices of interest.

Definition 7.1.2. *The distribution matrix $\sigma(M)$ of an $m \times n$ matrix M is the $(m+1) \times (n+1)$ matrix satisfying $\sigma(M)[i,j] = \sum_{r \geq i, c < j} M[r,c]$.*

Definition 7.1.3. *An $n \times n$ binary matrix is a permutation matrix if it has exactly one 1 entry in each row and each column. Such a matrix can be represented in $\mathcal{O}(n)$ space.*

By constructing the 2D orthogonal range counting data structure of Chan and Pătraşcu (Theorem 2.5.4) over the non-zero entries of a permutation matrix, one obtains the following lemma.

Lemma 7.1.4 ([161, Theorem 2.15])**.** *An $n \times n$ permutation matrix P can be preprocessed $\mathcal{O}(n\sqrt{\log n})$ time so that any entry of $\sigma(P)$ can be retrieved in time $\mathcal{O}(\log n / \log\log n)$.*

Let $\diamond$ be a wildcard letter, i.e. a letter that matches all letters. Tiskin [161] defines an $(m+n+1) \times (m+n+1)$ distance matrix $H_{S,T}$ over $G_{S,\diamond^m T \diamond^m}$ so that $H_{S,T}[i,j]$ equals the highest weight of a path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S,\diamond^m T \diamond^m}$. Note that if $j = i - m$, then $H_{S,T}[i,j] = 0$. By convention, if $j < i - m$, then $H_{S,T}[i,j] = j - (i-m) < 0$. The matrix $H_{S,T}$ captures so-called *semi-local LCS* values as follows; see Figure 7.2 for an illustration.

$$H_{S,T}[i,j] = \begin{cases} \texttt{LCS}(S[m-i+1\mathinner{.\,.} m], T[1\mathinner{.\,.} j]) + m - i & \text{if } i \leq m \text{ and } j \leq n, \\ \texttt{LCS}(S[1\mathinner{.\,.} m+n-j], T[i-m+1\mathinner{.\,.} n]) + j - n & \text{if } i \geq m \text{ and } j \geq n, \\ \texttt{LCS}(S[m-i+1\mathinner{.\,.} m+n-j], T) + m - i + j - n & \text{if } i \leq m \text{ and } j \in [n, n+i], \\ m & \text{if } n + i \leq j, \\ \texttt{LCS}(S, T[i-m+1\mathinner{.\,.} j]) & \text{if } i \geq m \text{ and } j \in [i-m, n], \\ j + m - i & \text{if } j \leq i - m. \end{cases}$$

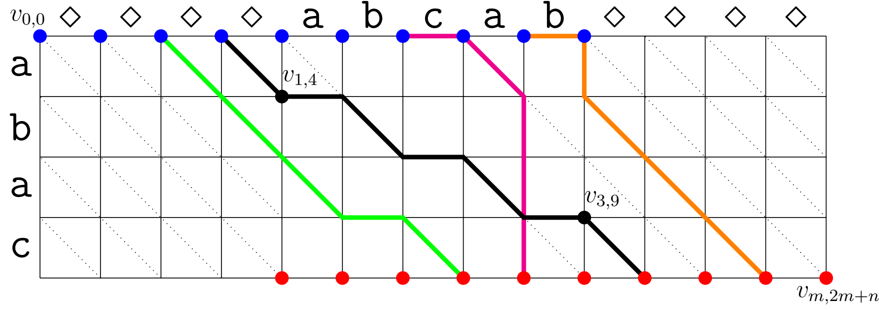


Figure 7.2: The figure illustrates how $H_{S,T}$ captures semi-local LCS information for $S = \texttt{abac}$ and $T = \texttt{abcab}$. We have $m = 4$ and $n = 5$. The value $H_{S,T}(i, j)$ captures the length of the highest scoring path from the i-th blue node to the j-th red node in the above figure (in the left-to-right order). The underlying idea is that when there are wildcards $\diamond$ involved, one may always choose to use the diagonal edges corresponding to them and then fill in the rest of the path. Let us analyse one of the cases thoroughly, the analysis of the other cases is analogous.

- The highest weight of a path from $v_{0,3}$ to $v_{4,4+6}$ is 4, which corresponds to $H_{S,T}(3, 6) = 4 = \texttt{LCS}(S[4-3+1\,..\,9-6], T) + 4 - 3 + 6 - 5 = \texttt{LCS}(S[2\,..\,3], T) + 2$ (case 3 of the equation above). The highest scoring path (in black), after trimming diagonal edges corresponding to wildcards, yields a highest scoring path from $v_{1,4}$ to $v_{3,4+5}$. Its weight indeed corresponds to $\texttt{LCS}(S[2\,..\,3], T) = 2$.

- The $v_{0,2}$-to-$v_{4,4+3}$ highest scoring path (in green) illustrates case 1 of the equation above: $H_{S,T}(2, 3) = 4 = \texttt{LCS}(S[4-2+1\,..\,4], T[1\,..\,3]) + 4 - 2 = \texttt{LCS}(S[3\,..\,4], T[1\,..\,3]) + 2$.

- The $v_{0,8}$-to-$v_{4,4+8}$ highest scoring path (in orange) illustrates case 2 of the equation above: $H_{S,T}(8, 8) = 3 = \texttt{LCS}(S[1\,..\,9-8], T[8-4+1\,..\,5]) + 8 - 5 = \texttt{LCS}(S[1], T[5]) + 3$.

- The $v_{0,6}$-to-$v_{4,4+4}$ highest scoring path (in magenta) illustrates case 5 of the equation above: $H_{S,T}(6, 4) = 1 = \texttt{LCS}(S, T[6-4+1\,..\,4]) = \texttt{LCS}(S, T[3\,..\,4])$.

Remark 7.1.5. Let us try to provide some extra intuition by considering the *indel distance*, for which we get a more uniform formula. The indel distance of two strings, denoted $\delta(S,T)$, is the minimum number of insertions and deletions that are needed to transform S to T. In other words, $\delta(S,T) = |S| + |T| - 2\texttt{LCS}(S,T)$. Then $2m + j - i - 2H_{S,T}[i,j]$, which can be interpreted as the number of length-0 edges on the highest scoring path from $v_{0,i}$ to $v_{m,m+j}$ in $G_{S,\diamond^m T \diamond^m}$, admits a more uniform formula:

$$2m + j - i - 2H_{S,T}[i,j] = \begin{cases} \delta(S[m-i+1\,..\,m], T[1\,..\,j]) & \text{if } i \le m \text{ and } j \le n, \\ \delta(S[1\,..\,m+n-j], T[i-m+1\,..\,n]) & \text{if } i \ge m \text{ and } j \ge n, \\ \delta(S[m-i+1\,..\,m+n-j], T) & \text{if } i \le m \text{ and } j \in [n, n+i], \\ j - i & \text{if } n + i \le j, \\ \delta(S, T[i-m+1\,..\,j]) & \text{if } i \ge m \text{ and } j \in [i-m, n], \\ i - j & \text{if } j \le i - m. \end{cases}$$

We now return to the LCS problem. Tiskin shows that the $(n+m) \times (n+m)$ matrix $P_{S,T}$ defined as

$$P_{S,T}[i,j] = H_{S,T}[i,j] + H_{S,T}[i+1,j+1] - H_{S,T}[i+1,j] - H_{S,T}[i,j+1], \tag{7.1}$$

is a permutation matrix and satisfies $H_{S,T}[i,j] = j + m - i - \sigma(P_{S,T})[i,j]$. Note that for constant-length strings S and T, the matrix $P_{S,T}$ can be computed naively in constant time from $H_{S,T}$. Conversely, each entry of $H_{S,T}$ can be retrieved in time $\mathcal{O}(\log(n+m)/\log\log(n+m))$ after an $\mathcal{O}((n+m)\sqrt{\log(n+m)})$-time preprocessing of $P_{S,T}$ by Lemma 7.1.4. Crucially for our approach, Tiskin shows the following result.

Theorem 7.1.6 ([161, Theorem 4.21])**.** *(a) Given $P_{S,T}$ and $P_{S,T'}$ for three strings S, T, T', each of length at most n, one can compute $P_{S,TT'}$ in $\mathcal{O}(n \log n)$ time.*

(b) Given $P_{T,S}$ and $P_{T',S}$ for three strings S, T, T', each of length at most n, one can compute $P_{TT',S}$ in $\mathcal{O}(n \log n)$ time.

Actually, only part (a) of the above theorem is stated explicitly in [161]. Part (b) can be derived by symmetry as follows; see also [161, Lemma 4.14]. One can

check using the characterisation of $H_{S,T}$ in terms of the semi-local LCS values that $H_{S,T}[i,j] = H_{T,S}[n+m-i, n+m-j] + m - i + j - n$. In particular, this means that $H_{S,T}$ can be obtained from $H_{T,S}$ by first performing a 180-degree rotation and then off-setting the values in every row i by $m - i$ and the values in every column j by $j - n$. This, in turn, means that $P_{T,S}$ can be obtained from $P_{S,T}$ just through a 180-degree rotation, as the offsets are cancelled out in the computation of $P_{S,T}[i,j]$ from $H_{T,S}$; see Equation (7.1). Thus, we can rotate $P_{T,S}$ and $P_{T',S}$ to obtain $P_{S,T}$ and $P_{S,T'}$, compute $P_{S,TT'}$ using Theorem 7.1.6(a), and then rotate $P_{S,TT'}$ to obtain $P_{TT',S}$.

7.2 Main Algorithm

We show how to maintain the permutation matrix $P_{S,T}$ in $\mathcal{O}((m+n)\log(m+n))$ time per update when the strings S and T undergo substitutions, insertions, and deletions of single letters. Within the stated update time we can recompute the orthogonal range counting data structure that allows us to report, in $\mathcal{O}(\log(m+n)/\log\log(m+n))$ time, any element of the matrix $H_{S,T}$.

The high-level idea is to maintain the permutation matrices $P_{A,B}$ for fragments A of S and B of T, at exponentially growing scales. Local changes to S and T, such as substitutions, insertions, and deletions, only affect a single fragment at each scale. We can therefore use Theorem 7.1.6 to recompute the affected matrices efficiently in a bottom-up fashion.

We first describe the maintenance of a data structure that can only support substitutions in order to demonstrate the general approach. We will then describe how to also support insertions and deletions.

7.2.1 Supporting Only Substitutions

We can assume that both S and T are of length n and that n is a power of two; otherwise, we pad S with \$ characters and T with # characters such that $\$ \neq \#$ and neither \$ nor # is in the alphabet. We define $\log n + 1$ scales, where at scale s, each of S and T is

partitioned into non-overlapping fragments of length 2^s. At every scale, and for every pair of fragments S_i and T_j of S and T, respectively, we store the permutation matrix P_{S_i,T_j} corresponding to H_{S_i,T_i}. At scale s, there are $(n/2^s)^2$ matrices, each stored in $\mathcal{O}(2^s)$ space. Thus, the overall space required by the data structure is $\mathcal{O}(n^2)$. Building the data structure in a bottom-up manner requires time $\sum_{s=0}^{\log n}(n/2^s)^2 \cdot 2^s \cdot s = \mathcal{O}(n^2)$ by Theorem 7.1.6.

Suppose, without loss of generality, that a letter of S is substituted (the other case is symmetric). We work in order of increasing scales $s = 0, 1, \ldots, \log n$. Let S_i be the unique fragment of S in scale s that contains the substituted letter. We recompute the matrices P_{S_i,T_j} for each one of the $n/2^s$ fragments T_j of T at scale s. At scale $s = 0$, both S_i and T_j consist of single letters, and we recompute the constant-size permutation matrices P_{S_i,T_j} from scratch in total $\mathcal{O}(n)$ time. (In fact, there are only two types of matrices, one corresponding to the case where the letter S_i matches the letter T_j, and the other corresponding to a mismatch.) To recompute a matrix P_{S_i,T_j} at scale $s > 0$, let S'_i, S''_i be the two fragments of S at scale $s-1$ such that $S_i = S'_i S''_i$. Similarly, let T'_j, T''_j be the two fragments of T at scale $s-1$ such that $T_j = T'_j T''_j$. We repeatedly apply Theorem 7.1.6 to $P_{S'_i,T'_j}, P_{S'_i,T''_j}, P_{S''_i,T'_j}, P_{S''_i,T''_j}$ to obtain P_{S_i,T_j} in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to update all affected permutation matrices at all scales (and, in particular, to obtain the matrix $P_{S,T}$) is $\sum_{s=0}^{\log n} \frac{n}{2^s} \cdot s \cdot 2^s = \mathcal{O}(n \log^2 n)$.

7.2.2 Supporting Insertions and Deletions

To support insertions and deletions we use the same approach. However, as each update increases or decreases the length of the string it is applied to, we can no longer use fixed-length fragments at each scale. At each scale s, we maintain a partition of each string into consecutive fragments, each of length between $\frac{1}{4} \cdot 2^s$ and $2 \cdot 2^s$, such that the partition at scale s is a refinement of the partition at scale $s+1$. Let us denote by R_s (resp. C_s) the partition of S (resp. T) at level s. We only describe the process for S; the string T is handled analogously. The *refinement property* for R_s can be stated formally as follows. For any $s' > s$, for each fragment $S[a\,..\,b] \in R_s$ there exists a

fragment $S[a'\,..\,b'] \in R_{s'}$ with $a' \le a \le b \le b'$. We maintain each R_s as a linked list of the fragments, which are represented by their start and end indices, sorted by the start indices in increasing order. Upon an update in S, we update the partitions in a bottom-up manner.

Let us first describe how to insert a letter in S after the letter at position k. We first scan R_s for all s and increment by 1 all the start indices that are greater than k and all the end indices that are at least k. This way, the newly inserted letter is assigned to a unique fragment in each partition. Then, we process the scales in increasing order, starting from scale 0. If the fragment $U_0 \in R_0$ that contains the newly inserted letter has just become of length greater than $2 \cdot 2^0 = 2$, then we split U_0 into two fragments of length at most 2. Note that this potential split does not violate the refinement property. Then, we proceed to the next scale. Generally, at scale s, if the length of the fragment $U_s \in R_s$ that contains the newly inserted letter does not exceed $2 \cdot 2^s$, we just proceed to the next scale. Otherwise, we need to make adjustments, ensuring that the refinement property is not violated. Note that, if $|U_s| = 2 \cdot 2^s + 1$, then, since fragments at scale $s-1$ have been already processed and respect the length constraint, the refinement property implies that U_s is the concatenation of at least three (and at most nine) fragments $V_1, V_2, \ldots, V_t$ at scale $s-1$. Let the middle letter of U_s belong to V_i. Then, either $\sum_{g<i} |V_g| \ge 2^s/4$ or $\sum_{g>i} |V_g| \ge 2^s/4$; let us assume without loss of generality that we are in the first case. We replace U at scale s by $V_1 \cdots V_{i-1}$ and $V_i \cdots V_t$. If such a replacement happens at the highest scale s (with $U_s = S$), then we create a new level $s+1$ with $R_{s+1} = \{U_s\} = \{S\}$. Note that the refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We now treat the complementary case of deleting $S[k]$. Again, we first scan R_s for all scales s and decrement by 1 all the start/end indices that are at least k—ensuring that none of them becomes negative. If some fragment becomes of length 0, then we remove it. We again process levels in increasing order. Suppose that the fragment $U_s \in R_s$ that contained the deleted letter has just become shorter than $\frac{1}{4} \cdot 2^s$. If R_s is the top level of the decomposition, then we simply remove this level. Otherwise, consider the fragment

$U_{s+1} \in R_{s+1}$ that contains U_s. Note that, $1 \leq |U_s| = \frac{1}{4} \cdot 2^s - 1$ implies that the length $|U_s| + 1$ of the fragment corresponding to U_s prior to the deletion is smaller than $\frac{1}{4} \cdot 2^{s+1}$, and hence $|U_{s+1}| > |U_s|$. Thus, there exists a fragment V at scale s that is adjacent to U_s and is also a subfragment of U_{s+1}. Let us assume without loss of generality that V lies to the right of U_s—the other case is symmetric. If $|V| < \frac{7}{4} \cdot 2^s$, then we can just replace U_s and V in R_s by their concatenation, U_sV. Otherwise, let the first element of the decomposition of V at scale $s-1$ be X. In this case, we can replace U_s and V in R_s by U_sX and $Y = V[|X| \mathinner{..} |V|-1]$, since $\frac{1}{4} \cdot 2^s \leq |U_sX| < \frac{1}{4} \cdot 2^s + 2 \cdot 2^{s-1} < 2 \cdot 2^s$ and $\frac{1}{4} \cdot 2^s < \frac{7}{4} \cdot 2^s - 2 \cdot 2^{s-1} \leq |V| - |X| = |Y| < |V| \leq 2 \cdot 2^s$. The refinement property is maintained and the whole procedure requires $\mathcal{O}(m)$ time.

We maintain $P_{A,B}$ for each pair of fragments $(A, B) \in R_s \times C_s$ at scale s. In the case where R_s simply consists of S at scale s, while T is still fragmented, we consider R_j for any $j > s$ to simply consist of S. (Symmetrically for the opposite case.) The number of pairs of fragments that are affected at scale s is $\mathcal{O}((n+m)/2^s)$. We compute $P_{A,B}$, for each such pair (A, B), using a constant number of applications of Theorem 7.1.6 in $\mathcal{O}(s \cdot 2^s)$ time. Thus, the total time to handle scale s is $\mathcal{O}((n+m)s)$ and the total time to handle all scales is $\mathcal{O}((m+n)\log^2(m+n))$.

Remark 7.2.1. Deletions of fragments of either of the strings can also be processed within the same time complexity with a straightforward generalisation of the above process.

Obtaining the longest common subsequence. We now describe how one can obtain the longest common subsequence, and not just its length, within $\tilde{\mathcal{O}}(n+m)$ time. Let us consider the following auxiliary problem: given some pair of fragments S_i, T_j at scale $s > 0$, compute the longest common subsequence of either some prefix of S_i (resp. T_j) and some suffix of T_j (resp. S_i), or some fragment of S_i (resp. T_j) and T_j (resp. S_i). Consider the refinement, at scale $s-1$, of S_i to $U_1, \ldots, U_k$ and of T_j to $V_1, \ldots, V_\ell$. Let $G_{S,T}(S[i_1 \mathinner{..} i_2], T[j_1 \mathinner{..} j_2])$ be the subgraph of $G_{S,T}$ induced by the set of vertices $\{v_{i',j'} : i_1 - 1 \leq i' \leq i_2, j_1 - 1 \leq j' \leq j_2\}$. Our aim is to decompose the highest scoring path in scope (say $v_{a,b}$-to-$v_{c,d}$) into subpaths, each lying entirely on some

$G_{S,T}(U_r, V_t)$. We can then apply this procedure recursively.

P_{S_i,T_j} was obtained from the $k \times \ell$ matrices P_{U_t,V_r} through some order of applications of Theorem 7.1.6. We can store such intermediate matrices, preprocessed as in Lemma 7.1.4, without any extra asymptotic cost in the complexities. We refine the path by considering the reverse order. For clarity of presentation, let us assume that $k = \ell = 2$ and the intermediate matrices were $P_{U_1U_2,V_1}$ and $P_{U_1U_2,V_2}$. We can decompose the path to at most two subpaths, one lying entirely on $J_1 = G_{S,T}(U_1U_2, V_1)$ and one lying entirely on $J_2 = G_{S,T}(U_1U_2, V_2)$. The case where both $v_{a,b}$ and $v_{c,d}$ lie on one of J_1 or J_2 is trivial. In the other case, we wish to find a node that lies on both J_1 and J_2 and is on the path. To this end, we query $P_{U_1U_2,V_1}$ (resp. $P_{U_1U_2,V_2}$) for the length of the highest scoring $v_{a,b}$-to-u (resp. u-to-$v_{c,d}$) path for all nodes u that belong to both J_1 and J_2. Using Lemma 7.1.4, this can be done in $\mathcal{O}(2^s \cdot s/\log s)$ time (for $s > 0$). Any u for which the sum of these values equals the length of the highest scoring $v_{a,b}$-to-$v_{c,d}$ path is a valid vertex to decompose the path. We then recurse, further refining the path. Note that the $v_{a,b}$-to-$v_{c,d}$ path gets decomposed into $\mathcal{O}((n+m)/2^s)$ pieces at scale s, for all s. Hence, by summing over all scales, the total time required for applying this procedure is $\mathcal{O}((n+m)\log^2(n+m))$.

Internal LCS queries. Our data structure also enables us to answer queries of the type $\texttt{LCS}(S[i_1 \mathinner{..} i_2], T[j_1 \mathinner{..} j_2])$ in time $\tilde{\mathcal{O}}(n+m)$ (in fact, in time $\tilde{\mathcal{O}}(1 + i_2 - i_1 + j_2 - j_1)$). Note that $G_{S,T}(S[i_1 \mathinner{..} i_2], T[j_1 \mathinner{..} j_2])$ can be decomposed in $\mathcal{O}((n+m)\log(n+m))$ time to multiple pieces $G_{S,T}(U, V)$, overlapping at their boundaries, such that U and V are of the same scale and there are $\mathcal{O}((n+m)/2^s)$ pairs (U, V) of scale s. This can be done, intuitively, using a greedy approach, that each time uses a piece from the largest possible scale. One can also think of this as extending a rectangle using a constant number of layers consisting of pieces corresponding to pairs of strings at scale s, in order of decreasing s. Finally, a repeated application of Theorem 7.1.6 yields that the total time complexity is $\mathcal{O}((n+m)\log^2(n+m))$.

7.2.3 Extension to String Alignment Under Integer Weights

Let us now consider the problem of computing an alignment of two strings S and T under integer weights w_{match}, w_{mis} and w_{gap}. Recall that w_{match} is the weight for aligning a pair of matching letters, w_{mis} is the weight for aligning a pair of mismatching letters, and w_{gap} is the weight for letters that are not aligned, and the goal is to compute an alignment with maximum weight. We may assume that $2w_{match} > 2w_{mis} \geq w_{gap}$ [161]. In this problem, the goal is to compute a highest scoring path from $v_{0,0}$ to $v_{m,n}$ in the following modification $\hat{G}_{S,T}$ of $G_{S,T}$. Edges of the form $v_{i,j}v_{i+1,j}$ and $v_{i,j}v_{i,j+1}$ have weight w_{gap}, while edges of the form $v_{i,j}v_{i+1,j+1}$ have weight w_{match} if $T[i+1] = S[i+1]$ and w_{mis} otherwise.

Tiskin shows in Section 6.1 of his monograph [161] that the alignment problem between strings S and T, can be reduced to the LCS problem between strings S' and T', obtained as follows. First, replace every letter a in S or in T by the string $\$^{\mu}a^{\nu-\mu}$, where $\$ \notin \Sigma$ and

$$\frac{\mu}{\nu} = \frac{w_{mis} - 2w_{gap}}{w_{match} - 2w_{gap}}.$$

Then, if one defines matrix $\hat{H}_{S,T}$ over $\hat{G}_{S,T}$ analogously to the definition of $H_{S,T}$ over $G_{S,T}$, we have that $\hat{H}_{S,T}(i,j) = \frac{1}{\nu} \cdot H_{S',T'}(\nu i, \nu j)$.

We maintain the same information as in the previous subsections, making sure that each fragment of each partition is a multiple of ν. At scale 0, we have only two options about how $P_{A,B}$ can look like, despite it being a $\nu \times \nu$ matrix; its structure only depends on whether $A = B$ or not. We precompute all such possible $P_{A,B}$'s. This way, upon an update on S or T, updating scale 0 requires $\mathcal{O}(n\nu)$ time. Over all scales, the total update time is

$$\mathcal{O}\left(\sum_{s=0}^{\log n} \frac{n\nu}{2^s\nu} \cdot 2^s\nu \cdot \log(2^s\nu)\right) = \mathcal{O}\left(n\nu \cdot \sum_{s=0}^{\log n}(s + \log\nu)\right) = \mathcal{O}(n\nu \log n \log(n\nu)).$$

The same reasoning shows that the preprocessing time is

$$\mathcal{O}\left(\sum_{s=0}^{\log n} \left(\frac{n\nu}{2^s\nu}\right)^2 \cdot 2^s\nu \cdot \log(2^s\nu)\right) = \mathcal{O}\left(n^2\nu \cdot \sum_{s=0}^{\log n}\left(\frac{s+\log\nu}{2^s}\right)\right) = \mathcal{O}(n^2\nu\log\nu).$$

As for retrieving the actual alignment, we process $\mathcal{O}(n\nu/2^s\nu)$ pieces at scale s, each in time $2^s \log(2^s\nu)/\log\log(2^s\nu)$, as we only try to decompose the path at nodes of the form $v_{i\nu,j\nu}$. The total time over all scales is thus dominated by $\mathcal{O}(n\nu \log n \log(n\nu))$.

The analysis of the time required to answer internal queries is identical to that of handling an update.

We summarise the results of this section in the following theorem.

Theorem 7.2.2. *For integer alignment weights w_{match}, w_{mis} and w_{gap}, bounded by w, the alignment score of two dynamic strings S and T, of total length at most n, as they undergo edit operations, can be maintained in $\mathcal{O}(nw \log n \log(nw))$ time per operation, after an $\mathcal{O}(n^2 w \log w)$-time preprocessing. An optimal alignment of S and T after each edit operation can be reported at no extra asymptotic cost. In addition, the following queries are supported:*

- *Semi-local alignment scores can be computed in $\mathcal{O}(\log(nw)/\log\log(nw))$ time.*
- *Fragment-to-fragment alignment scores can be computed in $\mathcal{O}(nw \log n \log(nw))$ time.*

7.3 Handling Large Weights

In this section, we describe an algorithm for string alignment that only relies on the planarity of $\hat{G}_{S,T}$. This algorithm outperforms the one from Theorem 7.2.2 when the alignment weights cannot be transformed to integers bounded by (roughly) $\sqrt{n}$.

Instead of computing a highest scoring path, we can reduce the problem to computing a shortest path in the alignment DAG. Given w_{match}, w_{mis} and w_{gap}, we define $w'_{match} = 0$, $w'_{mis} = w_{match} - w_{mis}$ and $w'_{gap} = \frac{1}{2}w_{match} - w_{gap}$. Then, a shortest path with respect to the new weights (of length W), corresponds to a highest scoring path with respect to the original weights (of score $\frac{1}{2}(m+n)w_{match} - W$).

7.3.1 Data Structures for Planar Graphs

Let us first introduce some data structures for shortest paths in planar graphs.

MSSP. The *multiple-source shortest paths* (MSSP) data structure of Klein [110] represents all shortest path trees rooted at the vertices of a single face f in a planar graph G of size n using a persistent dynamic tree. It can be constructed in $\mathcal{O}(n \log n)$ time, requires $\mathcal{O}(n \log n)$ space, and can report the distance between any vertex of f and any other vertex in G in $\mathcal{O}(\log n)$ time. The actual shortest path p can be retrieved in time $\mathcal{O}(\rho \log \log n)$, where ρ is the number of edges of p.

FR-Dijkstra. Let us consider a subgraph P of a planar graph G, and a face f of P. The *dense distance graph* (DDG) of P with respect to f, denoted $\mathsf{DDG}_{P,f}$ is a complete directed graph on the set of vertices F that lie on f. Each edge (u, v) has weight $d_P(u, v)$, equal to the length of the shortest u-to-v path in P. $\mathsf{DDG}_{P,f}$ can be computed in time $\mathcal{O}((|F|^2 + |P|) \log |P|)$ using MSSP. In their seminal paper, Fakcharoenphol and Rao [65] designed an efficient implementation of Dijkstra's algorithm on any union of DDGs—this algorithm is nicknamed FR-Dijkstra. The algorithm exploits the fact that, due to planarity, certain submatrices of the adjacency matrix of $\mathsf{DDG}_{P,f}$ satisfy the Monge property. We next give a—convenient for our purposes—interface for FR-Dijkstra, which was essentially proved in [65], with some additional components and details from [104, 132].

Theorem 7.3.1 ([65, 104, 132])**.** *Dijkstra's algorithm can be run on the union of a set of DDGs with $\mathcal{O}(N)$ vertices in total (with multiplicities) and an arbitrary set of $\mathcal{O}(N)$ extra edges in time $\mathcal{O}(N \log^2 N)$.*

Remark 7.3.2. An improvement in the time complexity of Theorem 7.3.1 was shown in [78].

7.3.2 Direct Application to String Alignment

Our approach is essentially the same as the one for dynamic distance oracles in planar graphs due to Klein [110], with extensions in [100, 104, 48]. We want to maintain a data structure that enables us to compute the length of the shortest $v_{0,0}$-to-$v_{m,n}$ path. However, instead of a single update to the graph, we have a batch of $\mathcal{O}(m + n)$ updates

for each update to one of the strings. We rely on the fact that the updates to the graphs are clustered in a constant number of rows/columns of $\hat{G}_{S,T}$ in order to process them more efficiently compared to simply using dynamic distance oracles for planar graphs in a black-box manner.

Let us consider a partition of $\hat{G}_{S,T}$ into $\mathcal{O}((n/r)^2)$ pieces of size $\Theta(r) \times \Theta(r)$ each. We consider the vertices that lie on the infinite face of each piece as its boundary nodes. Then, as each piece has $\mathcal{O}(r)$ boundary vertices, the total number of boundary vertices is $\mathcal{O}(n^2/r)$. We compute the MSSP data structure and the DDG for each piece with respect to its outer face in $\mathcal{O}(\frac{n^2}{r^2} \cdot r^2 \log n) = \mathcal{O}(n^2 \log n)$ time in total. Note that the shortest path from $v_{0,0}$ to $v_{m,n}$ can be decomposed to subpaths $p_1, \dots, p_k$ such that each p_i lies entirely within some piece P_i and p_i's endpoints are boundary nodes of P_i. Thus, we can compute the length of the shortest $v_{0,0}$-to-$v_{m,n}$ path by running FR-Dijkstra from $v_{0,0}$ in the union of all DDGs in $\mathcal{O}(\frac{n^2}{r} \cdot \log^2 n)$ time. In order to retrieve the actual shortest path, we can refine the DDG edges of the shortest $v_{0,0}$-to-$v_{m,n}$ path to the actual underlying edges using the MSSP data structures for the respective pieces.

Each update to one of the strings affects a constant number of rows or of columns of the original matrix and these are covered by $\mathcal{O}(n/r)$ pieces. The MSSP data structures and DDGs for these pieces can be recomputed using MSSP in $\mathcal{O}(\frac{n}{r} \cdot r^2 \log n) = \mathcal{O}(nr \log n)$ time. The balance is at $\frac{n^2}{r} \cdot \log^2 n = nr \log n$, which yields $r = \sqrt{n \log n}$, so the time per operation is $\mathcal{O}(n^{3/2} \log^{3/2} n)$. If a piece grows (resp. shrinks) too much, we break it into two pieces (resp. merge it with an adjacent piece and split in the middle) and recompute the DDGs for the affected pieces.

An optimal alignment of two arbitrary substrings of S and T can be computed in $\mathcal{O}(n^{3/2} \log^{3/2} n)$ time as follows. First, if the source and the target are not boundary vertices and belong to the same piece, we can simply compute the optimal alignment in $\mathcal{O}(r^2)$ time using the standard dynamic programming solution. Otherwise, we run FR-Dijkstra on the collection of DDGs, and, possibly, the following extra edges. In the case where the source (resp. target) is not a boundary vertex, we include $\mathcal{O}(\sqrt{n \log n})$ additional edges: for each boundary vertex of the piece containing it, an edge with length

equal to that of the shortest path from the source to (resp. to the target from) that boundary vertex. The weights of such edges can be computed in $\mathcal{O}(\sqrt{n \log n} \cdot \log n)$ time using the MSSP data structures at hand.

We summarise the results of this section in the following theorem.

Theorem 7.3.3. *Given two strings S and T and alignment weights w_{match}, w_{mis}, and w_{gap}, the optimal alignment of S and T as they undergo insertions, deletions, and substitutions of letters can be maintained in $\mathcal{O}(n^{3/2} \log^{3/2} n)$ time per operation after an $\mathcal{O}(n^2 \log n)$-time preprocessing. In addition, fragment-to-fragment alignment scores can be computed in $\mathcal{O}(n^{3/2} \log^{3/2} n)$ time.*

Bibliography

[1] Amir Abboud, Thomas Dueholm Hansen, Virginia Vassilevska Williams, and Ryan Williams. Simulating branching programs with edit distance and friends: or: a polylog shaved is a lower bound made. In *48th Annual ACM Symposium on Theory of Computing, STOC 2016*, pages 375–388, 2016. doi:10.1145/2897518.2897653.

[2] Paniz Abedin, Sahar Hooshmand, Arnab Ganguly, and Sharma V. Thankachan. The heaviest induced ancestors problem revisited. In *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 20:1–20:13, 2018. doi:10.4230/LIPIcs.CPM.2018.20.

[3] Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975. doi:10.1145/360825.360855.

[4] Stephen Alstrup, Gerth Stølting Brodal, and Theis Rauhe. Pattern matching in dynamic texts. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2000*, pages 819–828, 2000. URL: http://dl.acm.org/citation.cfm?id=338219.338645.

[5] Amihood Amir and Itai Boneh. Locally maximal common factors as a tool for efficient dynamic string algorithms. In *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 11:1–11:13, 2018. doi:10.4230/LIPIcs.CPM.2018.11.

[6] Amihood Amir and Itai Boneh. Dynamic palindrome detection. *CoRR*, 2019. arXiv:1906.09732.

[7] Amihood Amir and Itai Boneh. Update query time trade-off for dynamic suffix arrays. In *31st International Symposium on Algorithms and Computation, ISAAC 2020*, pages 63:1–63:16, 2020. doi:10.4230/LIPIcs.ISAAC.2020.63.

[8] Amihood Amir, Itai Boneh, Panagiotis Charalampopoulos, and Eitan Kondratovsky. Repetition Detection in a Dynamic String. In *27th Annual European Symposium on Algorithms, ESA 2019*, pages 5:1–5:18, 2019. doi:10.4230/LIPIcs.ESA.2019.5.

[9] Amihood Amir, Panagiotis Charalampopoulos, Costas S. Iliopoulos, Solon P. Pissis, and Jakub Radoszewski. Longest common factor after one edit operation. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, pages 14–26, 2017. doi:10.1007/978-3-319-67428-5_2.

[10] Amihood Amir, Panagiotis Charalampopoulos, Solon P. Pissis, and Jakub Radoszewski. Dynamic and internal longest common substring. *Algorithmica*, 82(12):3707–3743, 2020. doi:10.1007/s00453-020-00744-0.

[11] Amihood Amir, Martin Farach, Zvi Galil, Raffaele Giancarlo, and Kunsoo Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994. doi:10.1016/S0022-0000(05)80047-9.

[12] Amihood Amir, Martin Farach, Ramana M. Idury, Johannes A. La Poutré, and Alejandro A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995. doi:10.1006/inco.1995.1090.

[13] Amihood Amir and Eitan Kondratovsky. Searching for a modified pattern in a changing text. In *25th International Symposium on String Processing and Information Retrieval, SPIRE 2018*, pages 241–253, 2018. doi:10.1007/978-3-030-00479-8_20.

[14] Amihood Amir, Gad M. Landau, Moshe Lewenstein, and Dina Sokol. Dynamic text and static pattern matching. *ACM Transactions on Algorithms*, 3(2):19, 2007. doi:10.1145/1240233.1240242.

[15] Amihood Amir, Moshe Lewenstein, and Sharma V. Thankachan. Range LCP queries revisited. In *22nd International Symposium on String Processing and Information Retrieval, SPIRE 2015*, pages 350–361, 2015. doi:10.1007/978-3-319-23826-5_33.

[16] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Polylogarithmic approximation for edit distance and the asymmetric query complexity. In *51st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010*, pages 377–386, 2010. doi:10.1109/FOCS.2010.43.

[17] Alexandr Andoni and Negev Shekel Nosatzki. Edit distance in near-linear time: it's a constant factor. In *61st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2020*, pages 990–1001, 2020. doi:10.1109/FOCS46700.2020.00096.

[18] Alexandr Andoni and Krzysztof Onak. Approximating edit distance in near-linear time. *SIAM Journal on Computing*, 41(6):1635–1648, 2012. doi:10.1137/090767182.

[19] Alberto Apostolico, Maxime Crochemore, Martin Farach-Colton, Zvi Galil, and S. Muthukrishnan. Forty years of text indexing. In *24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, pages 1–10, 2013. doi:10.1007/978-3-642-38905-4_1.

[20] Alberto Apostolico and Franco P. Preparata. Optimal off-line detection of repetitions in a string. *Theoretical Computer Science*, 22:297–315, 1983. doi:10.1016/0304-3975(83)90109-3.

[21] Lorraine A. K. Ayad, Carl Barton, Panagiotis Charalampopoulos, Costas S. Iliopoulos, and Solon P. Pissis. Longest common prefixes with k-errors and applications.

In *25th International Symposium on String Processing and Information Retrieval, SPIRE 2018*, pages 27–41, 2018. doi:10.1007/978-3-030-00479-8_3.

[22] Maxim Babenko, Paweł Gawrychowski, Tomasz Kociumaka, and Tatiana Starikovskaya. Wavelet trees meet suffix trees. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 572–591, 2015. doi:10.1137/1.9781611973730.39.

[23] Maxim A. Babenko, Paweł Gawrychowski, Tomasz Kociumaka, Ignat I. Kolesnichenko, and Tatiana Starikovskaya. Computing minimal and maximal suffixes of a substring. *Theoretical Computer Science*, 638:112–121, 2016. doi:10.1016/j.tcs.2015.08.023.

[24] Arturs Backurs and Piotr Indyk. Edit distance cannot be computed in strongly subquadratic time (unless SETH is false). *SIAM Journal on Computing*, 47(3):1087–1097, 2018. doi:10.1137/15M1053128.

[25] Ricardo A. Baeza-Yates and Gonzalo Navarro. New and faster filters for multiple approximate string matching. *Random Structures & Algorithms*, 20(1):23–49, 2002. doi:10.1002/rsa.10014.

[26] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta. The "runs" theorem. *SIAM Journal on Computing*, 46(5):1501–1514, 2017. doi:10.1137/15M1011032.

[27] Hideo Bannai, Shunsuke Inenaga, and Dominik Köppl. Computing all distinct squares in linear time for integer alphabets. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 22:1–22:18, 2017. doi:10.4230/LIPIcs.CPM.2017.22.

[28] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *4th Latin American Symposium on Theoretical Informatics, LATIN 2000*, pages 88–94, 2000. doi:10.1007/10719839_9.

[29] Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theoretical Computer Science*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

[30] Jon Louis Bentley. Decomposable searching problems. *Information Processing Letters*, 8(5):244 – 251, 1979. doi:10.1016/0020-0190(79)90117-0.

[31] Omer Berkman and Uzi Vishkin. Finding level-ancestors in trees. *Journal of Computer and System Sciences*, 48(2):214–230, 1994. doi:10.1016/S0022-0000(05)80002-9.

[32] Philip Bille and Martin Farach-Colton. Fast and compact regular expression matching. *Theoretical Computer Science*, 409(3):486–496, 2008. doi:10.1016/j.tcs.2008.08.042.

[33] Or Birenzwige, Shay Golan, and Ely Porat. Locally consistent parsing for text indexing in small space. In *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 607–626, 2020. doi:10.1137/1.9781611975994.37.

[34] Mahdi Boroujeni, Masoud Seddighin, and Saeed Seddighin. Improved algorithms for edit distance and LCS: beyond worst case. In *31st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2020*, pages 1601–1620, 2020. doi:10.1137/1.9781611975994.99.

[35] Dany Breslauer and Zvi Galil. Finding all periods and initial palindromes of a string in parallel. *Algorithmica*, 14(4):355–366, 1995. doi:10.1007/BF01294132.

[36] Karl Bringmann and Marvin Künnemann. Quadratic conditional lower bounds for string problems and dynamic time warping. In *56th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2015*, pages 79–97, 2015. doi:10.1109/FOCS.2015.15.

[37] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks. Approximating edit distance within constant factor in truly sub-quadratic time. In *59th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2018*, pages 979–990, 2018. doi:10.1109/FOCS.2018.00096.

[38] Ho-Leung Chan, Wing-Kai Hon, Tak Wah Lam, and Kunihiko Sadakane. Dynamic dictionary matching and compressed suffix trees. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 13–22, 2005.

[39] Timothy M. Chan and Mihai Pătraşcu. Counting inversions, offline orthogonal range counting, and related problems. In *21st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 161–173, 2010. doi:10.1137/1.9781611973075.15.

[40] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Linear-time algorithm for long LCF with k mismatches. In *29th Annual Symposium on Combinatorial Pattern Matching, CPM 2018*, pages 23:1–23:16, 2018. doi:10.4230/LIPIcs.CPM.2018.23.

[41] Panagiotis Charalampopoulos, Paweł Gawrychowski, Shay Mozes, and Oren Weimann. Almost optimal distance oracles for planar graphs. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 138–151, 2019. doi:10.1145/3313276.3316316.

[42] Panagiotis Charalampopoulos, Paweł Gawrychowski, and Karol Pokorski. Dynamic longest common substring in polylogarithmic time. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*, pages 27:1–27:19, 2020. doi:10.4230/LIPIcs.ICALP.2020.27.

[43] Panagiotis Charalampopoulos, Pawel Gawrychowski, Shay Mozes, and Oren Weimann. An almost optimal edit distance oracle. *CoRR*, 2021. arXiv:2103.03294.

[44] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba. Counting distinct patterns in internal dictionary matching. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, pages 8:1–8:15, 2020. doi:10.4230/LIPIcs.CPM.2020.8.

[45] Panagiotis Charalampopoulos, Tomasz Kociumaka, Manal Mohamed, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal dictionary matching. In *30th International Symposium on Algorithms and Computation, ISAAC 2019*, pages 22:1–22:17, 2019. doi:10.4230/LIPIcs.ISAAC.2019.22.

[46] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Shay Mozes. Dynamic string alignment. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, pages 9:1–9:13, 2020. doi:10.4230/LIPIcs.CPM.2020.9.

[47] Panagiotis Charalampopoulos, Tomasz Kociumaka, and Philip Wellnitz. Faster approximate pattern matching: A unified approach. In *61st Annual IEEE Symposium on Foundations of Computer Science, FOCS 2020*, pages 978–989, 2020. Full version: arXiv:2004.08350. doi:10.1109/FOCS46700.2020.00095.

[48] Panagiotis Charalampopoulos, Shay Mozes, and Benjamin Tebeka. Exact distance oracles for planar graphs with failing vertices. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 2110–2123, 2019. doi:10.1137/1.9781611975482.127.

[49] David Clark. *Compact Pat trees*. PhD thesis, University of Waterloo, 1996. URL: http://hdl.handle.net/10012/64.

[50] Raphaël Clifford, Allyx Fontaine, Tatiana Starikovskaya, and Hjalte Wedel Vildhøj. Dynamic and approximate pattern matching in 2d. In *23rd International Symposium on String Processing and Information Retrieva, SPIRE 2016*, pages 133–144, 2016. doi:10.1007/978-3-319-46049-9_13.

[51] Raphaël Clifford, Allan Grønlund, Kasper Green Larsen, and Tatiana A. Starikovskaya. Upper and lower bounds for dynamic data structures on strings. In *35th Symposium on Theoretical Aspects of Computer Science, STACS 2018*, pages 22:1–22:14, 2018. doi:10.4230/LIPIcs.STACS.2018.22.

[52] Hagai Cohen and Ely Porat. On the hardness of distance oracle for sparse graph. *CoRR*, 2010. arXiv:1006.1117.

[53] Richard Cole, Lee-Ad Gottlieb, and Moshe Lewenstein. Dictionary matching and indexing with errors and don't cares. In *36th Annual ACM Symposium on Theory of Computing, STOC 2004*, pages 91–100, 2004. doi:10.1145/1007352.1007374.

[54] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Information Processing Letters*, 12(5):244–250, 1981. doi:10.1016/0020-0190(81)90024-7.

[55] Maxime Crochemore, Christophe Hancart, and Thierry Lecroq. *Algorithms on Strings*. Cambridge University Press, 2007. doi:10.1017/cbo9780511546853.

[56] Maxime Crochemore and Lucian Ilie. Analysis of maximal repetitions in strings. In *32nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2007*, pages 465–476, 2007. doi:10.1007/978-3-540-74456-6_42.

[57] Maxime Crochemore and Lucian Ilie. Maximal repetitions in strings. *Journal of Computer and System Sciences*, 74(5):796–807, 2008. doi:10.1016/j.jcss.2007.09.003.

[58] Maxime Crochemore, Lucian Ilie, and Liviu Tinta. The "runs" conjecture. *Theoretical Computer Science*, 412(27):2931–2941, 2011. doi:10.1016/j.tcs.2010.06.019.

[59] Maxime Crochemore, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Extracting powers and periods in a word

from its runs structure. *Theoretical Computer Science*, 521:29–41, 2014. doi:10.1016/j.tcs.2013.11.018.

[60] Maxime Crochemore, Gad M. Landau, and Michal Ziv-Ukelson. A subquadratic sequence alignment algorithm for unrestricted scoring matrices. *SIAM Journal on Computing*, 32(6):1654–1673, 2003. doi:10.1137/S0097539702402007.

[61] Maxime Crochemore and Wojciech Rytter. Squares, cubes, and time-space efficient string searching. *Algorithmica*, 13(5):405–425, 1995. doi:10.1007/BF01190846.

[62] Mark de Berg, Otfried Cheong, Marc J. van Kreveld, and Mark H. Overmars. *Computational geometry: algorithms and applications, 3rd Edition*. Springer, 2008.

[63] Antoine Deza, Frantisek Franek, and Adrien Thierry. How many double squares can a string contain? *Discrete Applied Mathematics*, 180:52–69, 2015. doi:10.1016/j.dam.2014.08.016.

[64] Shiri Dori and Gad M. Landau. Construction of Aho Corasick automaton in linear time for integer alphabets. *Information Processing Letters*, 98(2):66–72, 2006. doi:10.1016/j.ipl.2005.11.019.

[65] Jittat Fakcharoenphol and Satish Rao. Planar graphs, negative weight edges, shortest paths, and near linear time. *Journal of Computer and System Sciences*, 72(5):868–889, 2006. doi:10.1016/j.jcss.2005.05.007.

[66] Martin Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In *7th Annual Symposium on Combinatorial Pattern Matching, CPM 1996*, pages 130–140, 1996. doi:10.1007/3-540-61258-0_11.

[67] Martin Farach-Colton. Optimal suffix tree construction with large alphabets. In *38th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1997*, pages 137–143, 1997. doi:10.1109/SFCS.1997.646102.

[68] Paolo Ferragina. Dynamic text indexing under string updates. *Journal of Algorithms*, 22(2):296–328, 1997. doi:10.1006/jagm.1996.0814.

[69] Paolo Ferragina and Roberto Grossi. Optimal on-line search and sublinear time update in string matching. *SIAM Journal on Computing*, 27(3):713–736, 1998. doi:10.1137/S0097539795286119.

[70] Nathan J. Fine and Herbert S. Wilf. Uniqueness theorems for periodic functions. *Proceedings of the American Mathematical Society*, 16(1):109–114, 1965. doi:10.2307/2034009.

[71] Johannes Fischer and Volker Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM Journal on Computing*, 40(2):465–492, 2011. doi:10.1137/090779759.

[72] Aviezri S. Fraenkel and Jamie Simpson. How many squares can a string contain? *Journal of Combinatorial Theory, Series A*, 82(1):112–120, 1998. doi:10.1006/jcta.1997.2843.

[73] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the ACM*, 31(3):538–544, 1984. doi:10.1145/828.1884.

[74] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990*, pages 434–443, 1990. URL: http://dl.acm.org/citation.cfm?id=320176.320229.

[75] Harold N. Gabow and Robert Endre Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985. doi:10.1016/0022-0000(85)90014-5.

[76] Travis Gagie, Paweł Gawrychowski, and Yakov Nekrich. Heaviest induced ancestors and longest common substrings. In *Proceedings of the 25th Canadian Conference on Computational Geometry, CCCG 2013*, 2013.

[77] Paweł Gawrychowski and Wojciech Janczewski. Fully dynamic approximation of LIS in polylogarithmic time. *CoRR*, 2020. arXiv:2011.09761.

[78] Paweł Gawrychowski and Adam Karczmarz. Improved bounds for shortest paths in dense distance graphs. In *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018*, pages 61:1–61:15, 2018. doi:10.4230/LIPIcs.ICALP.2018.61.

[79] Paweł Gawrychowski, Gad M. Landau, Shay Mozes, and Oren Weimann. The nearest colored node in a tree. *Theoretical Computer Science*, 710:66–73, 2018. doi:10.1016/j.tcs.2017.08.021.

[80] Paweł Gawrychowski, Shay Mozes, Oren Weimann, and Christian Wulff-Nilsen. Better tradeoffs for exact distance oracles in planar graphs. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 515–529, 2018. doi:10.1137/1.9781611975031.34.

[81] Paweł Gawrychowski and Tatiana Starikovskaya. Streaming dictionary matching with mismatches. In *30th Annual Symposium on Combinatorial Pattern Matching, CPM 2019*, pages 21:1–21:15, 2019. doi:10.4230/LIPIcs.CPM.2019.21.

[82] Paweł Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski. Optimal dynamic strings. In *29th ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 1509–1528, 2018. doi:10.1137/1.9781611975031.99.

[83] Mathieu Giraud. Not so many runs in strings. In *2nd International Conference on Language and Automata Theory and Applications, LATA 2008*, pages 232–239, 2008. doi:10.1007/978-3-540-88282-4_22.

[84] Amy Glen and Jamie Simpson. The total run length of a word. *Theoretical Computer Science*, 501:41–48, 2013. doi:10.1016/j.tcs.2013.06.004.

[85] Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat. Dynamic dictionary matching in the online model. In *16th International Symposium on Algorithms and Data Structures, WADS 2019*, pages 409–422, 2019. doi:10.1007/978-3-030-24766-9_30.

[86] Elazar Goldenberg, Robert Krauthgamer, and Barna Saha. Sublinear algorithms for gap edit distance. In *60th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2019*, pages 1101–1120, 2019. doi:10.1109/FOCS.2019.00070.

[87] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *15th International Symposium Algorithms and Data Structures, WADS 2017*, pages 421–436, 2017. doi:10.1007/978-3-319-62127-2_36.

[88] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya. Approximating longest common substring with k mismatches: Theory and practice. In *31st Annual Symposium on Combinatorial Pattern Matching, CPM 2020*, pages 16:1–16:15, 2020. doi:10.4230/LIPIcs.CPM.2020.16.

[89] Szymon Grabowski. New tabulation and sparse dynamic programming based techniques for sequence similarity problems. *Discrete Applied Mathematics*, 212:96–103, 2016. doi:10.1016/j.dam.2015.10.040.

[90] Richard Groult, Élise Prieur, and Gwénaël Richomme. Counting distinct palindromes in a word in linear time. *Information Processing Letters*, 110(20):908–912, 2010. doi:10.1016/j.ipl.2010.07.018.

[91] Ming Gu, Martin Farach, and Richard Beigel. An efficient algorithm for dynamic text indexing. In *5th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1994*, pages 697–704, 1994. URL: http://dl.acm.org/citation.cfm?id=314464.314675.

[92] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *Journal of Computer and System Sciences*, 69(4):525–546, 2004. doi:10.1016/j.jcss.2004.03.004.

[93] MohammadTaghi Hajiaghayi, Masoud Seddighin, Saeed Seddighin, and Xiaorui Sun. Approximating LCS in linear time: Beating the $\sqrt{n}$ barrier. In *30th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2019*, pages 1181–1200, 2019. doi:10.1137/1.9781611975482.72.

[94] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, 1984. doi:10.1137/0213024.

[95] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and strengthening hardness for dynamic problems via the online matrix-vector multiplication conjecture. In *47th Annual ACM Symposium on Theory of Computing, STOC 2015*, pages 21–30, 2015. doi:10.1145/2746539.2746609.

[96] Daniel S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Communications of the ACM*, 18(6):341–343, 1975. doi:10.1145/360825.360861.

[97] Heikki Hyyrö, Kazuyuki Narisawa, and Shunsuke Inenaga. Dynamic edit distance table under a general weighted cost function. *Journal of Discrete Algorithms*, 34:2–17, 2015. doi:10.1016/j.jda.2015.05.007.

[98] Tomohiro I. Longest common extensions with recompression. In *28th Annual Symposium on Combinatorial Pattern Matching, CPM 2017*, pages 18:1–18:15, 2017. doi:10.4230/LIPIcs.CPM.2017.18.

[99] Yusuke Ishida, Shunsuke Inenaga, Ayumi Shinohara, and Masayuki Takeda. Fully incremental LCS computation. In *15th International Symposium on Fundamentals of Computation Theory, FCT 2005*, pages 563–574, 2005. doi:10.1007/11537311_49.

[100] Giuseppe F. Italiano, Yahav Nussbaum, Piotr Sankowski, and Christian Wulff-Nilsen. Improved algorithms for min cut and max flow in undirected planar graphs. In *43rd ACM Symposium on Theory of Computing, STOC 2011*, pages 313–322, 2011. doi:10.1145/1993636.1993679.

[101] Guy Jacobson. Space-efficient static trees and graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*, pages 549–554, 1989. doi:10.1109/SFCS.1989.63533.

[102] Artur Jeż. Faster fully compressed pattern matching by recompression. *ACM Transactions on Algorithms*, 11(3):20:1–20:43, 2015. doi:10.1145/2631920.

[103] Artur Jeż. Recompression: A simple and powerful technique for word equations. *Journal of the ACM*, 63(1):4:1–4:51, 2016. doi:10.1145/2743014.

[104] Haim Kaplan, Shay Mozes, Yahav Nussbaum, and Micha Sharir. Submatrix maximum queries in Monge matrices and partial Monge matrices, and their applications. *ACM Transactions on Algorithms*, 13(2):26:1–26:42, 2017. doi:10.1145/3039873.

[105] Haim Kaplan, Natan Rubin, Micha Sharir, and Elad Verbin. Efficient colored orthogonal range counting. *SIAM Journal on Computing*, 38(3):982–1011, 2008. doi:10.1137/070684483.

[106] Orgad Keller, Tsvi Kopelowitz, Shir Landau Feibish, and Moshe Lewenstein. Generalized substring compression. *Theoretical Computer Science*, 525:42–54, 2014. doi:10.1016/j.tcs.2013.10.010.

[107] Dominik Kempa and Tomasz Kociumaka. String synchronizing sets: sublinear-time BWT construction and optimal LCE data structure. In *51st Annual ACM SIGACT Symposium on Theory of Computing, STOC 2019*, pages 756–767, 2019. doi:10.1145/3313276.3316368.

[108] Dominik Kempa and Tomasz Kociumaka. Resolution of the Burrows–Wheeler transform conjecture. In *61st Annual IEEE Symposium on Foundations of Computer*

Science, FOCS 2020, pages 1002–1013, 2020. Full version: arXiv:1910.10631. doi:10.1109/FOCS46700.2020.00097.

[109] Sung-Ryul Kim and Kunsoo Park. A dynamic edit distance table. *Journal of Discrete Algorithms*, 2(2):303–312, 2004. doi:10.1016/S1570-8667(03)00082-0.

[110] Philip N. Klein. Multiple-source shortest paths in planar graphs. In *16th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005*, pages 146–155, 2005. URL: http://dl.acm.org/citation.cfm?id=1070432.1070454.

[111] Tomasz Kociumaka. Minimal suffix and rotation of a substring in optimal time. In *27th Annual Symposium on Combinatorial Pattern Matching, CPM 2016*, pages 28:1–28:12, 2016. doi:10.4230/LIPIcs.CPM.2016.28.

[112] Tomasz Kociumaka. *Efficient Data Structures for Internal Queries in Texts*. PhD thesis, University of Warsaw, 2018. URL: https://mimuw.edu.pl/~kociumaka/files/phd.pdf.

[113] Tomasz Kociumaka, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. A linear-time algorithm for seeds computation. *ACM Transactions on Algorithms*, 16(2):27:1–27:23, 2020. doi:10.1145/3386369.

[114] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Efficient data structures for the factor periodicity problem. In *19th International Symposium on String Processing and Information Retrieval, SPIRE 2012*, pages 284–294, 2012. doi:10.1007/978-3-642-34109-0_30.

[115] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń. Internal pattern matching queries in a text and applications. In *26th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015*, pages 532–551, 2015. doi:10.1137/1.9781611973730.36.

[116] Tomasz Kociumaka, Jakub Radoszewski, and Tatiana A. Starikovskaya. Longest common substring with approximately k mismatches. *Algorithmica*, 81(6):2633–2652, 2019. doi:10.1007/s00453-019-00548-x.

[117] Tomasz Kociumaka and Saeed Seddighin. Improved dynamic algorithms for longest increasing subsequence. *CoRR*, 2020. arXiv:2011.10874.

[118] Tomasz Kociumaka, Tatiana A. Starikovskaya, and Hjalte Wedel Vildhøj. Sublinear space algorithms for the longest common substring problem. In *22nd Annual European Symposium on Algorithms, ESA 2014*, pages 605–617, 2014. doi:10.1007/978-3-662-44777-2_50.

[119] Roman M. Kolpakov and Gregory Kucherov. Finding maximal repetitions in a word in linear time. In *40th Annual Symposium on Foundations of Computer Science, FOCS 1999*, pages 596–604, 1999. doi:10.1109/SFFCS.1999.814634.

[120] Tsvi Kopelowitz, Gregory Kucherov, Yakov Nekrich, and Tatiana Starikovskaya. Cross-document pattern matching. *Journal of Discrete Algorithms*, 24:40–47, 2014. doi:10.1016/j.jda.2013.05.002.

[121] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *27th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016*, pages 1272–1287, 2016. doi:10.1137/1.9781611974331.ch89.

[122] Tsvi Kopelowitz and Virginia Vassilevska Williams. Towards optimal set-disjointness and set-intersection data structures. In *47th International Colloquium on Automata, Languages, and Programming, ICALP 2020*, pages 74:1–74:16, 2020. doi:10.4230/LIPIcs.ICALP.2020.74.

[123] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt. Incremental string comparison. *SIAM Journal on Computing*, 27(2):557–582, 1998. doi:10.1137/S0097539794264810.

[124] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics. Doklady*, 10:707–710, 1966.

[125] Yaowei Long and Seth Pettie. Planar distance oracles with better time-space tradeoffs. In *32nd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 2517–2537, 2021. doi:10.1137/1.9781611976465.149.

[126] Michael G. Main and Richard J. Lorentz. An $O(n \log n)$ algorithm for finding all repetitions in a string. *Journal of Algorithms*, 5(3):422–432, 1984. doi: 10.1016/0196-6774(84)90021-X.

[127] Veli Mäkinen and Gonzalo Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007. doi:10.1016/j.tcs.2007.07.013.

[128] William J. Masek and Mike Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980. doi: 10.1016/0022-0000(80)90002-1.

[129] Kurt Mehlhorn, Rajamani Sundar, and Christian Uhrig. Maintaining Dynamic Sequences under Equality Tests in Polylogarithmic Time. *Algorithmica*, 17(2):183–198, 1997. doi:10.1007/BF02522825.

[130] Michael Mitzenmacher and Saeed Seddighin. Dynamic algorithms for LIS and distance to monotonicity. In *Proccedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing, STOC 2020*, pages 671–684, 2020. doi: 10.1145/3357713.3384240.

[131] Gaspard Monge. *Mémoire sur la théorie des déblais et des remblais.* De l'Imprimerie Royale, 1781.

[132] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n \log^2 n / \log \log n)$ time. In *18th Annual European Symposium on*

Algorithms, ESA 2010, pages 206–217, 2010. doi:10.1007/978-3-642-15781-3_18.

[133] J. Ian Munro, Gonzalo Navarro, and Yakov Nekrich. Text indexing and searching in sublinear time. *CoRR*, 2017. arXiv:1712.07431.

[134] J. Ian Munro, Yakov Nekrich, and Jeffrey Scott Vitter. Fast construction of wavelet trees. *Theoretical Computer Science*, 638:91–97, 2016. doi:10.1016/j.tcs.2015.11.011.

[135] Robert Muth and Udi Manber. Approximate multiple strings search. In *7th Annual Symposium on Combinatorial Pattern Matching, CPM 1996*, pages 75–86, 1996. doi:10.1007/3-540-61258-0_7.

[136] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *13th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2002*, pages 657–666, 2002. URL: http://dl.acm.org/citation.cfm?id=545381.545469.

[137] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, 1970. doi:10.1016/0022-2836(70)90057-4.

[138] Yakov Nekrich. A data structure for multi-dimensional range reporting. In *23rd International Symposium on Computational Geometry, SoCG 2007*, pages 344–353, 2007. doi:10.1145/1247069.1247130.

[139] Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Towards efficient interactive computation of dynamic time warping distance. In *27th International Symposium on String Processing and Information Retrieval, SPIRE 2020*, pages 27–41, 2020. doi:10.1007/978-3-030-59212-7_3.

[140] Stav Ben Nun, Shay Golan, Tomasz Kociumaka, and Matan Kraus. Time-space tradeoffs for finding a long common substring. In *31st Annual Symposium on*

Combinatorial Pattern Matching, CPM 2020, pages 5:1–5:14, 2020. doi:10.4230/LIPIcs.CPM.2020.5.

[141] Mihai Pătraşcu and Erik D. Demaine. Logarithmic lower bounds in the cell-probe model. *SIAM Journal on Computing*, 35(4):932–963, 2006. doi:10.1137/S0097539705447256.

[142] Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *38th Annual ACM Symposium on Theory of Computing, STOC 2006*, pages 232–240, 2006. doi:10.1145/1132516.1132551.

[143] Mihai Pătraşcu. Unifying the landscape of cell-probe lower bounds. *SIAM Journal on Computing*, 40(3):827–847, 2011. doi:10.1137/09075336X.

[144] Simon J. Puglisi, Jamie Simpson, and William F. Smyth. How many runs can a string contain? *Theoretical Computer Science*, 401(1-3):165–171, 2008. doi:10.1016/j.tcs.2008.04.020.

[145] Mihai Pătraşcu and Liam Roditty. Distance oracles beyond the Thorup-Zwick bound. *SIAM Journal on Computing*, 43(1):300–311, 2014. doi:10.1137/11084128X.

[146] Mikhail Rubinchik and Arseny M. Shur. Counting palindromes in substrings. In *24th International Symposium on String Processing and Information Retrieval, SPIRE 2017*, pages 290–303, 2017. doi:10.1007/978-3-319-67428-5_25.

[147] Milan Ružić. Constructing efficient dictionaries in close to sorting time. In *35th International Colloquium on Automata, Languages and Programming, ICALP 2008*, pages 84–95, 2008. doi:10.1007/978-3-540-70575-8_8.

[148] Wojciech Rytter. The number of runs in a string: Improved analysis of the linear upper bound. In *23rd Annual Symposium on Theoretical Aspects of Computer Science, STACS 2006*, pages 184–195, 2006. doi:10.1007/11672142_14.

[149] Wojciech Rytter. The number of runs in a string. *Information and Computation*, 205(9):1459–1469, 2007. doi:10.1016/j.ic.2007.01.007.

[150] Süleyman Cenk Sahinalp and Uzi Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *37th Annual IEEE Symposium on Foundations of Computer Science, FOCS 1996*, pages 320–328, 1996. doi:10.1109/SFCS.1996.548491.

[151] Yoshifumi Sakai. A substring-substring LCS data structure. *Theoretical Computer Science*, 753:16–34, 2019. doi:10.1016/j.tcs.2018.06.034.

[152] David Sankoff. Matching sequences under deletion/insertion constraints. *Proceedings of the National Academy of Sciences of the United States of America*, 69(1):4–6, 1972. doi:10.1073/pnas.69.1.4.

[153] Peter H. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974. doi:10.1137/0126070.

[154] Tatiana A. Starikovskaya and Hjalte Wedel Vildhøj. Time-space trade-offs for the longest common substring problem. In *24th Annual Symposium on Combinatorial Pattern Matching, CPM 2013*, pages 223–234, 2013. doi:10.1007/978-3-642-38905-4_22.

[155] Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree. *Theoretical Computer Science*, 270(1-2):843–856, 2002. doi:10.1016/S0304-3975(01)00121-9.

[156] Rajamani Sundar and Robert E Tarjan. Unique binary-search-tree representations and equality testing of sets and sequences. *SIAM Journal on Computing*, 23(1):24–44, 1994. doi:10.1145/100216.100219.

[157] Yuka Tanimura, Takaaki Nishimoto, Hideo Bannai, Shunsuke Inenaga, and Masayuki Takeda. Small-space LCE data structure with constant-time queries. In *42nd International Symposium on Mathematical Foundations of Computer Science, MFCS 2017*, pages 10:1–10:15, 2017. doi:10.4230/LIPIcs.MFCS.2017.10.

[158] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru. A provably efficient algorithm for the k-mismatch average common substring problem. *Journal of Computational Biology*, 23(6):472–482, 2016. doi:10.1089/cmb.2015.0235.

[159] Mikkel Thorup. Space efficient dynamic stabbing with fast queries. In *35th Annual ACM Symposium on Theory of Computing, STOC 2003*, pages 649–658, 2003. doi:10.1145/780542.780636.

[160] Alexander Tiskin. Longest common subsequences in permutations and maximum cliques in circle graphs. In *17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006*, pages 270–281, 2006. doi:10.1007/11780441_25.

[161] Alexander Tiskin. Semi-local string comparison: algorithmic techniques and applications. *CoRR*, 2007. arXiv:0707.3619.

[162] Alexander Tiskin. Semi-local longest common subsequences in subquadratic time. *Journal of Discrete Algorithms*, 6(4):570–581, 2008. doi:10.1016/j.jda.2008.07.001.

[163] Alexander Tiskin. Semi-local string comparison: Algorithmic techniques and applications. *Mathematics in Computer Science*, 1(4):571–603, 2008. doi:10.1007/s11786-007-0033-3.

[164] Alexander Tiskin. Faster subsequence recognition in compressed strings. *Journal of Mathematical Sciences*, 158(5):759–769, 2009. doi:10.1007/s10958-009-9396-0.

[165] Alexander Tiskin. Periodic string comparison. In *20th Annual Symposium on Combinatorial Pattern Matching, CPM 2009*, pages 193–206, 2009. doi:10.1007/978-3-642-02441-2_18.

[166] Alexander Tiskin. Fast distance multiplication of unit-Monge matrices. *Algorithmica*, 71(4):859–888, 2015. doi:10.1007/s00453-013-9830-z.

[167] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977. `doi:10.1016/0020-0190(77)90031-X`.

[168] Taras K. Vintsyuk. Speech discrimination by dynamic programming. *Cybernetics*, 4:52–57, 1968. `doi:10.1007/BF01074755`.

[169] Robert A. Wagner and Michael J. Fischer. The string-to-string correction problem. *Journal of the ACM*, 21(1):168–173, 1974. `doi:10.1145/321796.321811`.

[170] Peter Weiner. Linear pattern matching algorithms. In *14th Annual Symposium on Switching and Automata Theory, SWAT (FOCS) 1973*, pages 1–11. IEEE, 1973. `doi:10.1109/SWAT.1973.13`.

[171] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Information Processing Letters*, 17(2):81–84, 1983. `doi:10.1016/0020-0190(83)90075-3`.

[172] Dan E. Willard and George S. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985. `doi:10.1145/3828.3839`.

[173] Sun Wu, Udi Manber, and Eugene W. Myers. A subquadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996. `doi:10.1007/BF01942606`.