2D DICTIONARY MATCHING IN SMALL SPACE

by

SHOSHANA NEUBURGER

A dissertation submitted to the Graduate Faculty in Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy, The City University of New York

2012

© 2012
Shoshana Neuburger
All Rights Reserved

This manuscript has been read and accepted for the Graduate Faculty in Computer Science in satisfaction of the dissertation requirement for the degree of Doctor of Philosophy.

Dina Sokol

Date — Chair of Examining Committee

Theodore Brown

Date — Executive Officer

Amihood Amir

Amotz Bar-Noy

Stathis Zachos

Supervision Committee

THE CITY UNIVERSITY OF NEW YORK

Abstract

2D Dictionary Matching in Small-Space

by

Shoshana Neuburger

Advisor: Professor Dina Sokol

The dictionary matching problem seeks all locations in a given text that match any of the patterns in a given dictionary. Efficient algorithms for dictionary matching scan the text once, searching for all patterns simultaneously. There are many scenarios in which storage capacity is limited or the data sets are exceedingly large. The added constraint of performing efficient dictionary matching using little or no extra space is a challenging and practical problem. This thesis focuses on the problem of performing dictionary matching on two-dimensional data in small space. We have developed the first efficient algorithms for succinct 2D dictionary matching in both static and dynamically changing data. Although time and space optimal dictionary matching algorithms for one-dimensional data have recently been developed, they have not yet been implemented. Since our two-dimensional algorithms employ one-dimensional dictionary matching, we created software to solve one-dimensional dictionary matching in small space. This is a first step towards developing software for succinct dictionary matching in two-dimensional data.

Acknowledgements

I would like to express my gratitute to Prof. Dina Sokol for for being an exceptional mentor. She has made herself available to me and guided me patiently and skillfully through every aspect of the research process. She has become a role model both academically and personally.

I have also been privileged to be a student of Prof. Amotz Bar-Noy. Although I was not his doctoral student, he has shared with me his advice and experience in academia and has inspired me to immerse myself in the world of ideas. I have also been fortunate to take courses with Prof. Stathis Zachos. It has been an honor and inspiration to interact with a researcher of his caliber. Prof. Amihood Amir has graciously given of his time and has a taken a personal interest that helped shape my research.

It has been an exceptional experience to be a member of the graduate program under the direction of Prof. Theodore Brown. My personal academic achievements would not have been possible without the supportive staff at the Graduate Center who administered the grants that funded my graduate studies and travels. I have benefited greatly from the teaching fellowship that funded the majority of my dissertation work. As such, I am grateful to Prof. Aaron Tenenbaum and to Prof. Yedidyah Langsam of the Brooklyn College CIS department for providing me with interesting courses that complemented my

studies and for the favorable teaching schedules that made this all possible.

Over the years, I have been fortunate to learn from and work with professors whose creativity has broadened my thinking and whose guidance has been invaluable. These include Prof. Paula Whitlock, Prof. Noson Yanofsky, and Prof. S. Muthukrishnan. This section would be incomplete if I do not pay tribute to the late Prof. Chaya Gurwitz, who mentored me throughout my undergraduate experience and pushed me to pursue graduate studies in mathematics and in computer science. Ultimately, it was her influence that led me to pursue a career in academia. Her exceptional personal example will always guide me.

Most of all, I would like to thank my parents for their continuous encouragement and motivation. They embody the values of dedication and hard work. They have imparted these values powerfully through their own example. I would also like to give a special thank you to my husband for being supportive and encouraging throughout the dissertation process.

Table of Contents

Abstract **iv**

Acknowledgements **v**

Table of Contents **vi**

List of Tables **viii**

List of Figures **ix**

1 Introduction **1**

2 Preliminaries **5**
2.1 Periodicity 5
2.2 Conjugacy 6
2.3 Empirical Entropy 6

3 Related Work **8**
3.1 1D Dictionary Matching 9
3.1.1 Linear Time and Space 9
3.1.2 Small-Space Algorithms 10
3.1.3 Dynamic Dictionary Matching 12
3.2 2D Dictionary Matching 15
3.2.1 Linear Time and Space 15
3.2.2 Small-Space Algorithms 18
3.2.3 Dynamic Dictionary Matching 18
3.3 Indexing 20
3.3.1 Suffix Tree 21
3.3.2 Suffix Array 23

3.3.3 Compressed Data Structures 24

4 Succinct 2D Dictionary Matching 29
4.1 Overview 29
4.2 Case I: Patterns With Rows of Period Size $\leq m/4$ 31
4.2.1 Pattern Preprocessing 32
4.2.2 Text Scanning 47
4.3 Case II: Patterns With Row of Period Size $> m/4$ 51
4.3.1 Case IIa: $d < m$ 52
4.3.2 Case IIb: $d \geq m$ 60
4.4 Data Compression 62

5 Succinct 2D Dictionary Matching With No Slowdown 64
5.1 Large Number of Patterns 65
5.2 Small Number of Patterns 67
5.2.1 Pattern Preprocessing 69
5.2.2 Text Scanning 71

6 Dynamic 2D Dictionary Matching in Small Space 77
6.1 2D-DDM in Linear Space 79
6.2 2D-DDM in Small-Space For Large Number of Patterns 81
6.3 2D-DDM in Small-Space for Small Number of Patterns 84
6.3.1 Dynamic Witness Tree 85
6.3.2 Group I Patterns 87
6.3.3 Group II Patterns 93

7 Implementation 99
7.1 Software Development 100
7.2 Evaluation 109

8 Conclusion 110

Bibliography 113

List of Tables

3.1 Algorithms for 1D small-space dictionary matching 11

3.2 Comparison of the time complexities of dynamic 2D dictionary matching algorithms. 19

3.3 Suffix Array of the string Mississippi 23

List of Figures

3.1 Suffix tree for the string Mississippi. 21

4.1 2D patterns with their 1D representations 33
4.2 Witness tree 34
4.3 h-periodic matrix 37
4.4 Horizontally consistent patterns 39
4.5 Offset tree 45
4.6 Duel between candidates in the same column 57
4.7 Duel between candidates in the same row 59

5.1 Linearized 2D patterns 70
5.2 Witness tree 70

7.1 Suffix tree for the string Mississippi with several suffix links. 102

Chapter 1

Introduction

Pattern matching is a fundamental problem in computer science with applications in a wide array of domains. In its basic form, a pattern matching solution locates all occurrences of a pattern string within a larger text string. A simple computing task, such as a file search utility employs pattern matching techniques, as does a word processor when it searches a document for a specific word. Computational molecular biology and the World-Wide Web provide additional settings in which efficient pattern matching algorithms are essential.

The *dictionary matching problem* is an extension of the single pattern matching paradigm where the task is to identify a *set* of patterns, called a dictionary, within a given text. Applications for this problem include searching for specific phrases in a book, scanning a file for virus signatures, and network intrusion detection. The problem also has applications in the biological sciences, such as searching through a DNA sequence for a set of motifs. Both pattern matching and dictionary matching generalize to the two-dimensional setting. Image identification software, which identifies smaller images in a large image based on a set of known images, is a direct application of dictionary matching on two-dimensional data.

In recent years, there has been a massive proliferation of digital data. Some of the main contributors to this data explosion are the World-Wide Web, next generation sequencing, and increased use of satellite imaging. Concurrently, industry has been producing equipment with ever-decreasing hardware availability. Thus, researchers are faced with scenarios in which this data growth must be accessible to applications running on devices that have reduced storage capacity, such as mobile and satellite devices. Hardware resources are more limited, yet the consumer's expectations of software capability continue to escalate. This unprecedented rate of digital data accumulation therefore presents a constant challenge to the algorithms and software developers who must work with a shrinking hardware capacity.

A series of succinct dictionary matching algorithms for the one-dimensional setting have in fact been developed. The related problem of small-space dictionary matching in two-dimensional data has not been addressed until now. Existing algorithms for 2D dictionary matching are not suitable for the space-constrained setting. This thesis contributes new algorithms and data structures to fill this void. Our algorithms preprocess the dictionary and then scan the text, searching for all patterns simultaneously. Thus, when a text arrives as input, the running times of our algorithms depend only on the size of the text and are independent of the size of the dictionary.

The main accomplishment presented in this thesis is the development of new techniques that have proven to be extremely useful for solving dictionary matching in small space. These innovations include Lyndon word naming in one dimension, 2D Lyndon words, the witness tree, the offset tree, and dynamic dueling. First, we demonstrate how these tools are generally used. Then, we show how useful these tools are because they can be specifically used to solve other variations of succinct dictionary matching. They allow us to achieve

succinct 2D dictionary matching with no slowdown and dynamic dictionary matching in small space. For each variant of the general 2D dictionary matching problem, we combined the original techniques presented in this thesis with new approaches to the classic problems.

The first achievement of this thesis is in the development of the first algorithms for static succinct 2D dictionary matching, when the dictionary is known in advance. The second contribution of this thesis is the presentation of an efficient algorithm for dynamic 2D dictionary matching, when the dictionary can change over time. Developing an efficient algorithm for dynamic data presents its own challenge. When a pattern is added to or removed from the dictionary, we do not reprocess the entire dictionary. Rather, the indexes are updated in time proportional to the size of the pattern that is entering or leaving the set of patterns in the dictionary. All of our new algorithms use sublinear working space.

Our algorithms for 2D data use succinct data structures to gather information about each pattern and then form a linear representation of the dictionary. Then the text is linearized in the same manner and the linearized text is searched for pattern occurrences. We would like to create software for 2D dictionary matching in small space. However, our algorithms for succinct 2D dictionary matching rely on succinct 1D dictionary matching algorithms. These algorithms for the 1D setting have not yet been implemented. They rely on intricate data structures, whose coding is not a trivial task.

The final contribution of this thesis is the development of software for succinct 1D dictionary matching. Our main challenge lay in combining an algorithm for the generalized suffix tree with the succinct data structures that are readily available to us, and were designed with other purposes in mind. In this thesis we present intuition behind our approach and an overview of our code.

This thesis is organized as follows. Chapter 2 establishes terminology that will be used

liberally throughout this thesis. In Chapter 3, we review relevant background on one and two dimensional dictionary matching as well as recent innovations in text indexing. Chapters 4 - 7 present the accomplishments and contributions of this thesis work. In Chapter 4, we introduce our new techniques for succinct dictionary matching and present the first efficient algorithm for succinct 2D dictionary matching. In Chapter 5, we improve on this algorithm to perform 2D dictionary matching in small space and linear time. In Chapter 6, we extend our focus to the scenario in which the dictionary can change over time and present the first efficient algorithm for dynamic 2D dictionary matching in small space. Chapter 7 delineates the techniques we employed in developing software for succinct 1D dictionary matching. We conclude with a summary and open problems in Chapter 8.

Chapter 2

Preliminaries

2.1 Periodicity

A periodic pattern contains several locations where the pattern can be superimposed on itself without mismatch. We say a pattern is *non-periodic* if the origin is the only position before the midpoint at which the pattern can be superimposed on itself without mismatch.

In a periodic string, a smallest period can be found whose concatenation generates the entire string. A string S is periodic if its longest prefix that is also a suffix is at least half the length of S. A proper suffix that is also a prefix of a string is called a *border*. There is duality between periods and borders. The length of a string with its longest border subtracted corresponds to its shortest period.

More formally, a string S is *periodic* if $S = u^j u'$ where u' is a (possibly null) proper prefix of u, and $j \geq 2$. A periodic string S can be expressed as $u^j u'$ for one unique primitive u. A string S is *primitive* if it cannot be expressed in the form $S = u^j$, for $j > 1$ and a prefix u of S. We refer to both u and $|u|$ as "the period" of S, although S can have several non-primitive periods. The period of S can also be defined as $|S| - b$ where b is the longest

border of S.

For example, consider the periodic string S = *abcabcabcab*. The longest border of S is b =*abcabcab*. Since $|b| \geq \frac{|S|}{2}$, S is periodic. $u =$ *abc* is the period of S. Another way of concluding that S is periodic is by the observation that $|u| < \frac{|S|}{2}$.

2.2 Conjugacy

Two strings, x and y, are said to be *conjugate* if $x = uv$, $y = vu$ for some strings u, v. Two strings are conjugate if they differ only by a cyclic permutation of their characters.

A *Lyndon word* is a primitive string which is strictly smaller than any of its conjugates for the alphabetic ordering. In other terms, a string x is a Lyndon word if for any factorization $x = uv$ with u, v nonempty, one has $uv < vu$.

Any string has a conjugate which is a Lyndon word, namely its least conjugate. Computing the smallest conjugate of a string is a practical way to compute a standard representative of the conjugacy class of a string. This procedure is called *canonization*.

2.3 Empirical Entropy

Empirical entropy is defined in terms of the number of occurrences of each symbol or group of symbols. Therefore, it is defined for any string without requiring any probabilistic assumption and it can be used to establish worst-case results. For $k \geq 0$, the kth order empirical entropy of a string S, $H_k(S)$, provides a lower bound to the compression we can achieve for each symbol using a code which depends on the k symbols preceding it.

Let S be a string of length n over alphabet $\Sigma = \{\alpha_1, \ldots, \alpha_\sigma\}$, and let n_i denote the number of occurrences of the symbol α_i inside S. The 0th order empirical entropy of the

string S is defined as

$$H_0(S) = -\sum_{i=1}^{\sigma} \frac{n_i}{n} \log \frac{n_i}{n}.$$

We can achieve greater compression if the codeword we use for each symbol depends on the k symbols preceding it. For any string w of length k, let w_S denote the string of single characters following the occurrences of w in S, taken from left to right. The kth order empirical entropy of S is defined as

$$H_k(S) = \frac{1}{n} \sum_{w \in \Sigma^k} |w_S| H_0(w_S).$$

The value $nH_k(S)$ represents a lower bound to the compression we can achieve using codes which depend on the k most recently seen symbols.

For any string S and $k \geq 0$ we have the following hierarchy: [49]

$$H_k(S) \leq H_{k-1}(S) \leq \cdots H_0(S) \leq \log |\Sigma|$$

Chapter 3

Related Work

This chapter provides context for the accomplishments presented in this thesis. It portrays relevant background along with the framework upon which this thesis work is built. Since the algorithms we develop in this thesis for succinct 2D dictionary matching employ 1D dictionary matching techniques, we begin in Section 3.1 by presenting efficient 1D dictionary matching algorithms. We begin with the classical linear time and space algorithm and then describe recent developments in succinct 1D dictionary matching. We also review the dynamic dictionary matching algorithms and the few succinct dynamic dictionary matching algorithms that have been developed, albeit with a slowdown. Then, in Section 3.2, we review dictionary matching algorithms for 2D data to highlight the dearth of succinct 2D dictionary matching algorithms. In Section 3.3 we provide an overview of common data structures that index a string. We then refer to them freely throughout this thesis. We focus on the suffix tree, the suffix array, and the recent advances in compressed self-indexing that serve the dual purpose of compressing the underlying data and simultaneously indexing it, all within very limited amounts of space.

3.1 1D Dictionary Matching

3.1.1 Linear Time and Space

The pattern matching problem consists of locating all occurrences of a pattern string in a text string. Efficient algorithms preprocess the pattern once so that the search is completed in time proportional to the length of the text. *Dictionary matching* is a generalization of the pattern matching problem. It seeks to find all occurrences of all elements of a *set* of pattern strings in a text string. The set of patterns $D = \{P_1, P_2, \dots, P_d\}$ is called the *dictionary*.

We can define dictionary matching by:

> INPUT: A set of patterns $P_1, P_2, \dots, P_d$ of total length ℓ and a text $T = t_1 t_2 \dots t_n$ all over an alphabet Σ, with $|\Sigma| = \sigma$.
>
> OUTPUT: All ordered pairs (i, j) such that pattern P_j matches the segment of text beginning at location t_i.

Knuth, Morris, and Pratt (KMP) developed a well-known linear-time algorithm for pattern matching [44]. They construct an automaton that maintains a failure link for each prefix of the pattern. The failure link of a position points to its longest suffix that is also a pattern prefix. Aho and Corasick (AC) extended the Knuth-Morris-Pratt algorithm to dictionary matching by forming an automaton of the dictionary [1]. Preprocessing requires time and space proportional to the size of the dictionary. Then, the text is scanned once to identify all pattern occurrences. The search phase runs in time proportional to the length of the text, independent of the size of the dictionary. The AC automaton branches to different patterns with similar prefixes, yielding an overall $O(n \log \sigma)$ time to scan the text.

3.1.2 Small-Space Algorithms

Linear-time *single* pattern matching algorithms have achieved impressively small space complexities. For 1D data, we have pattern matching algorithms that require only constant extra space [29, 20, 57, 30]. The first time-space optimal pattern matching algorithm is from Galil and Seiferas [29]. Crochemore and Perrin developed "Two-Way String Matching" [20] which blends the classical Knuth-Morris-Pratt and Boyer-Moore [12] algorithms but computes pattern shifts as needed. Rytter presented a constant-space, yet linear-time version of the Knuth-Morris-Pratt algorithm [57]. The algorithm relies on small-space computation of both approximate periods and lexicographically maximal suffixes, which leads to the computation of periods in $O(1)$ space. Space-efficient real-time searching is discussed by Gasieniec and Kolpakov [30]. Their innovative algorithm uses a partial *next* function to save space.

Concurrently searching for a set of patterns within limited working space presents a greater challenge than searching for a single pattern in small space. Much effort has recently been devoted to solving 1D dictionary matching in small space [14, 38, 9, 37]. We summarize the state of the art for small-space 1D dictionary matching in Table 3.1 and describe the results in the following paragraphs.

The empirical entropy of a string (H_0 or H_k) describes the minimum number of bits that are needed to encode the string within context. Empirical entropy is often used as a measure of space, as it is in Table 3.1. Precise formulas for H_0 and H_k are included in Section 2.3.

Let $D = \{P_1, P_2, \ldots, P_d\}$ be a dictionary of 1D patterns of total length ℓ, $T = t_1 t_2 \ldots t_n$ a text, and occ the number of pattern occurrences in the text. Aho and Corasick presented the first algorithm that solves the dictionary matching problem in $O(\ell \log \ell)$

Space (bits)	Search Time	Reference
$O(\ell \log \ell)$	$O(n + occ)$	Aho-Corasick [1]
$O(\ell)$	$O((n + occ) \log^2 \ell)$	Chan et al. [14]
$\ell H_k(D) + o(\ell \log \sigma) + O(d \log \ell)$	$O(n(\log^\epsilon \ell + \log d) + occ)$	Hon et al. [38]
$\ell(H_0(D) + O(1)) + O(d \log(\ell/d))$	$O(n + occ)$	Belazzougui [9]
$\ell H_k(D) + O(\ell)$	$O(n + occ)$	Hon et al. [37]

Table 3.1: Algorithms for 1D small-space dictionary matching where ℓ is the size of the dictionary, n is the size of the text, d is the number of patterns in the dictionary, σ is the alphabet size, and occ is the number of occurrences of a dictionary pattern in the text.

preprocessing time and $O(n \log \sigma + occ)$ text scanning time [1]. Hashing techniques can achieve linear time complexity in the Aho-Corasick algorithm. The underlying index of their algorithm occupies $O(\ell)$ words, or $O(\ell \log \ell)$ bits. The first algorithm that improves the space complexity of dictionary matching was presented by Chan et al. [14]. They reduced the size of the dictionary index from $O(\ell)$ words, or $O(\ell \log \ell)$ bits, to $O(\ell)$ bits. Their algorithm relies on a compressed representation of the suffix tree and assumes that the alphabet is of constant size. It can find all pattern occurrences in the text in $O((n + occ) \log^2 \ell)$ time.

More recently, Hon et al. presented a 1D dictionary matching algorithm that uses a sampling technique to compress a suffix tree [38]. The patterns are concatenated, with a delimiter separating them, to form a single string which is stored in a compressed format that allows $O(1)$ time retrieval of any character. This results in an algorithm that requires $\ell H_k(D) + o(\ell \log \sigma) + O(d \log \ell)$ space and searches in $O(n(\log^\epsilon \ell + \log d) + occ)$ time, where $\epsilon > 0$ is any constant. Since the patterns are concatenated before the compressed index is constructed, $H_k(D) = H_k(P_1 P_2 \ldots P_d)$.

The first succinct dictionary matching algorithm with no slowdown was introduced by Belazzougui [9]. His algorithm mimics the Aho-Corasick automaton within smaller space. The algorithm requires $\ell(H_0(D) + O(1)) + O(d \log(\ell/d))$ bits. This new approach encodes

the *goto*, *fail*, and *report* functions separately. Hon et al. combine Belazzougui's work with the XBW transform [21] to store an AC automaton in space that meets kth order empirical entropy bounds of the dictionary with no slowdown [37]. They follow the approach of Belazzougui [9] to compress the AC automaton and store its three functions separately. However, they encode the forward transitions of the trie with the XBW transform [21]. With this new representation, the space meets optimal compression of the dictionary and runtime is linear.

The most recent result of Hon et al. [37] has essentially closed the problem of succinct 1D dictionary matching. Their algorithm runs in linear time within space that meets entropy bounds of the dictionary.

We point out that the AC automaton (whether compressed or not) *replaces* the actual dictionary of patterns. That is, once it is constructed, the actual patterns are not needed for performing the search. The goal of the small-space 1D algorithms in Table 3.1 was to minimize the space needed for this structure, which is in a sense the space needed for the input. When analyzing the space needed by small-space algorithms, we distinguish between the space used by the data structures that *replace* the actual input, and the *extra space* that is needed above the input.

3.1.3 Dynamic Dictionary Matching

It is often the case that the dictionary of patterns will change over time. Efficient dynamic dictionary matching algorithms support insertion of a new pattern to the dictionary and removal of a pattern from the dictionary. They thereby eliminate the need to reprocess the entire dictionary and can adapt to changes as they occur.

Amir and Farach introduced the use of the suffix tree for dictionary matching [3]. They

delimit the dictionary patterns with \$ and then concatenate the dictionary with the text and index $T\$D$, with artificial suffixes mixed among the genuine suffixes. If no pattern can be a substring of another, the suffix tree contains all the information needed to perform dictionary matching. When one pattern can be a substring of another, each internal node is labeled by its nearest ancestor that is a pattern occurrence. This is done by a depth first search after the suffix tree is fully constructed. Modifying the dictionary can trigger the update of many labels on nodes and can thus require relabeling the entire suffix tree, which is costly in terms of time. Amir and Farach use an L-tree on the marked nodes to support efficient reparenting of nodes. Then, all operations (preprocessing the dictionary, adding a pattern, removing a pattern, scanning text) run in linear time with an $O(\log \ell)$ slowdown.

Amir et al. [5] improved the previous algorithm so that it does not require any indexing as the text is processed and can process a text online, as it arrives. The suffix tree is simply traversed as the text is read, using suffix links. Processing the patterns and the text meets the same time complexity as [3], linear with an $O(\log \ell)$ slowdown. To know which pattern is a substring of another, they partition the suffix tree into a forest. The $O(\log \ell)$ slowdown in the algorithm is the upper bound on the time complexity of operations in the dynamic forest. Each marked node in the suffix tree, representing a pattern occurrence, becomes the root of a forest component, by removing the edge that connects the marked node to its parent. Nodes are mapped between the two data structures and the root of each forest component shows which nodes the pattern is a prefix of.

Idury and Schaffer developed a dynamic version of the Aho-Corasick automaton [40]. They update the fail function efficiently but with some slowdown. The initial construction of the automaton requires $O(\ell \log \sigma)$ time; this linear time complexity meets that of Aho and Corasick's algorithm. This is an improvement over [3, 5], which incur an $O(\log \ell)$

slowdown in preprocessing. However, the other phases of the algorithm incur a slight slowdown, as in [3, 5]. Text scanning runs in $O((n + occ)\log \ell)$ time and a pattern P, of length p, is added to or removed from the dictionary in $O(p \log \ell)$ time.

Idury and Schaffer explore alternative representations of their dynamic AC automaton. They point out that other trade-offs between search and update times are possible. Using a different data structure, this algorithm achieves the same search time as AC and update time $O(p(k\ell^{1/k} + \log \sigma))$, for any constant $k \geq 2$.

The dynamic dictionary matching algorithm of Amir et al. [6] mimics the Aho-Corasick automaton but stores the *goto* and *report* transitions separately. Overall, there is an $O(\frac{\log \ell}{\log \log \ell})$ slowdown to update the dictionary or to scan text. Instead of the suffix tree, this algorithm uses balanced parentheses as the underlying index. The fail function is computed by a “find nearest enclosing parentheses” operation. To support pattern removal from the dictionary, a balanced tree is constructed, and preprocessed for lowest common ancestor queries among nodes. If only insertion of a pattern, and not removal, is supported, all operations complete in linear time. For such a scenario, this algorithm meets the linear time complexity of the Aho-Corasick automaton.

Sahinalp and Vishkin achieved dynamic dictionary matching with no slowdown [60]. Preprocessing time is linear in the size of the dictionary, text scanning is linear in the size of the text and a pattern is added or removed in time proportional to the size of the pattern. The time complexity of this algorithm meets the standard set by Aho and Corasick.

Sahinalp and Vishkin’s algorithm relies on compact tries and a new data structure, the fat tree. This is the first dynamic infrastructure for dictionary matching that does not slow down the search or indexing processes. Their algorithm employs a naming technique and identifies cores of each pattern using a compact representation of the fat tree. If a pattern

matches a substring of the text, then the main core of the pattern and the text substring should necessarily be aligned. Conversely, if the main cores do not match, the text is easily filtered to a limited number of positions at which a pattern can occur.

For dynamic dictionary matching in the space constrained application, Chan et al. [14] use the compressed suffix tree for succinct dictionary matching. They build on the work of Amir and Farach [3] to use the suffix tree for dictionary matching. They replace the suffix tree with a compressed suffix tree developed by Sadakane [59], which is stored in $O(\ell)$ bits, and show how to make the data structure dynamic. They describe how to answer lowest marked ancestor queries by a balanced parenthesis representation of the nodes. The time complexity of inserting or removing a pattern and of scanning text has a slowdown of $O(\log^2 \ell)$.

An improved succinct dynamic dictionary matching algorithm was developed by Hon et al. [39]. It uses space that meets kth order empirical entropy bounds of the dictionary. The suffix tree is sampled to save space and an innovative method is proposed for a lowest marked ancestor data structure. They introduce the combination of a dynamic interval tree with a Dietz and Sleator order-maintenance data structure as a framework for answering lowest marked ancestor queries efficiently. Inserting or removing a dictionary pattern P, of length p, requires $O(p \log \sigma + \log \ell)$ time and searching a text of length n requires $O(n \log \ell + occ)$ time.

3.2 2D Dictionary Matching

3.2.1 Linear Time and Space

We can define two-dimensional dictionary matching as:

INPUT: A set of pattern matrices $P_1, P_2, \ldots, P_d$ and a text matrix T, all over an alphabet Σ, with $|\Sigma| = \sigma$.

OUTPUT: All tuples (h, i, j) such that pattern P_h occurs at location (i, j) in T, i.e., $T[i + k, j + l] = P_h[k + 1, l + 1]$, $0 \leq k, l < m$.

We first consider single pattern matching in two dimensions and then shift our focus to two-dimensional dictionary matching. The first linear-time 2D single pattern matching algorithm was developed independently by Bird [11] and by Baker [8]. They translate the 2D pattern matching problem into a 1D pattern matching problem. Rows of the pattern are perceived as metacharacters and named so that distinct rows receive different names. The text is named in a similar fashion and 1D pattern matching is performed over the text columns and the pattern of names. Algorithm 1 is an outline of the Bird / Baker algorithm.

Algorithm 1 Bird / Baker Algorithm

{1} Preprocess Pattern:
  a) Form Aho-Corasick automaton of pattern rows.
  b) Name pattern rows using Aho-Corasick and store 1D pattern.
  c) Construct Knuth-Morris-Pratt automaton of 1D pattern.
{2} Row Matching:
  Run Aho-Corasick on each text row.
  This labels position at which a pattern row ends.
{3} Column Matching:
  Run Knuth-Morris-Pratt on named columns of text.
  Output pattern occurrences.

Although this algorithm was initially developed for a single pattern, it is easily extended to perform dictionary matching by replacing the KMP automaton with another AC automaton. The Bird / Baker algorithm is appropriate for 2D patterns that are of uniform size in at least one dimension, so that the text can be marked. The Bird / Baker method uses linear time and space in both the pattern preprocessing and the text scanning stages. The linear

time complexity of the algorithm depends on the assumption that the label of each state fits into a single word of RAM.

There are several efficient algorithms that perform dictionary matching over square patterns. In the 2D dictionary, $D = \{P_1, P_2, \ldots, P_d\}$, each pattern P_i is a square of size $p_i \times p_i$, $1 \leq i \leq d$, and the text T is of size $n \times n$. The total size of the dictionary is $|D| = \sum_{i=1}^{d} p_i^2$. Let $\overline{D} = \sum_{i=1}^{d} p_i$.

Amir and Farach [4] presented an algorithm for 2D dictionary matching that is suitable for square patterns of different sizes. Their algorithm also deals with metacharacters but converts the patterns to a 1D representation by considering subrow/subcolumn pairs around the diagonals. Then they run Aho-Corasick on text that is linearized along the diagonals. Metacharacters are compared by longest common prefix queries. This is done efficiently with suffix trees of the pattern rows and columns. Text scanning time is $O(n^2 \log d)$, and the extra space used is proportional to the size of the text plus the patterns of names. This is considered a linear-time algorithm since the $O(\log d)$ slowdown stems from branching in the AC automaton.

Giancarlo developed the first 2D suffix tree [31]. At the same time, he introduced a 2D dictionary matching algorithm for square patterns that is based on this data structure, which he calls an Lsuffix tree. The time and space complexities of this algorithm are comparable to Amir and Farach's approach that uses a 1D suffix tree for 2D data. Preprocessing of the pattern builds an Lsuffix tree in $O(|D| + \overline{D} \log \overline{D})$ time and $O(|D|)$ space. Based on it, the text scanning process simulates an automaton in $O(n^2 \log \overline{D} + occ)$ time.

Idury and Schaffer [41] developed an algorithm for dictionary matching in rectangular patterns with different heights, widths, and aspect ratios. Such patterns cannot be aligned at a corner so the notion of comparing prefixes and suffixes of patterns is not defined. They

split patterns into overlapping pieces and apply dictionary matching as well as techniques for multidimensional range searching. Idury and Schaffer's algorithm requires working space proportional to the dictionary size, and has a slight slowdown in the time for text processing.

3.2.2 Small-Space Algorithms

An approach for small-space, yet linear-time *single* pattern matching in 2D was developed by Crochemore et al. [19]. Their algorithm preprocesses an $m \times m$ pattern, of total size m^2, within only $O(\log m)$ working space and scans the text in $O(1)$ extra space. Such an algorithm can be trivially extended to perform dictionary matching but would require $O(dn)$ time to process the text, a time complexity that is dependent on the number of patterns in the dictionary.

None of the existing approaches to 2D dictionary matching are suitable for a space-constrained environment. The main contribution of this thesis is to address this problem, both in the static and dynamic settings.

3.2.3 Dynamic Dictionary Matching

We now turn our attention to the scenario in which the dictionary can change over time. Several different dynamic 2D dictionary matching algorithms exist. Table 3.2 summarizes the different time complexities achieved by the dynamic dictionary matching algorithms for square patterns. These results all incorporate some slowdown in processing text and in updating the dictionary; the question is *how much* slowdown.

We use notation consistent with Section 3.2.1. In the dictionary, $D = \{P_1, P_2, \ldots, P_d\}$, each pattern P_i is a square of size $p_i \times p_i$, $1 \leq i \leq d$, and the text T is of size $n \times n$.

Dictionary Update Time	Text Searching Time	Reference
$O(p^2 \log \lvert D \rvert)$	$O((n^2 + occ) \log \lvert D \rvert)$	Amir et al. [6]
$O(p^2 \log^2 \lvert D \rvert)$	$O((n^2 \log \overline{D} + occ) \log \lvert D \rvert)$	Giancarlo [31]
$O(p^2 + p \log \overline{D})$	$O((n^2 + occ) \log \overline{D})$	Choi and Lam [15]

Table 3.2: Comparison of the time complexities of dynamic 2D dictionary matching algorithms.

Let P, of size $p \times p$, denote a square pattern that will be inserted to or removed from the dictionary. The total size of the dictionary is $|D| = \sum_{i=1}^{d} p_i^2$. Let $\overline{D} = \sum_{i=1}^{d} p_i$.

Amir et al. [6] extended Amir and Farach's approach for 2D static dictionary matching of square patterns to the setting in which the dictionary can change. For a static dictionary, they use the suffix tree with lowest common ancestor queries to form an automaton that recognizes patterns in the text. However, they could not efficiently update the precomputed lowest common ancestor information upon modification of the dictionary. Instead, they devised a creative workaround for the fail function to work. Amir et al. use Idury and Schaffer's dynamic version of Aho-Corasick [40] to index the pattern substrings. The text is marked along its diagonals for subrow and subcolumn occurrences separately. The text is labeled as in the Bird / Baker algorithm, but each position is given two labels, each stored in a separate matrix. Then, pattern occurrences are announced by running the dynamic version of the Aho-Corasick algorithm over the marked text. Every operation, including text scanning and dictionary preprocessing is close to linear; only an $O(\log |D|)$ slowdown is incurred.

Giancarlo's 2D suffix tree can be used for dictionary matching both in the case that the dictionary is static and in the case that the dictionary is dynamic [31]. There is a slowdown in the text scanning stage of Giancarlo's algorithm that can handle a dynamic 2D dictionary of square patterns. For the dynamic case, Amir et al. [6] achieved slightly better time complexity.

Choi and Lam [15] set out to demonstrate that Giancarlo's suffix tree based approach to dictionary matching is just as good as Amir et al.ś automaton based approach. Their algorithm maintains two augmented suffix trees, an adapted version of Giancarlo's Lsuffix tree, and a forest of dynamic trees. They point out that even without their results, for a dictionary of 2D patterns that are all the same size, Bird and Baker's algorithm can be extended to insert and delete a pattern in $O(p^2 \log dp^2)$ time, and search a text in $O((n^2 + occ) \log dp^2)$ time.

Idury and Schaffer developed a dynamic dictionary matching algorithm for rectangular patterns of different sizes [41]. This algorithm is based on several applications of the Bird / Baker algorithm, by dividing the dictionary into groups of uniform height. There is an $O(\log^4 |D|)$ slowdown in each part of the algorithm, preprocessing the dictionary, text scanning, and updating the dictionary.

3.3 Indexing

Indexing is an important paradigm in searching. The text is preprocessed so that queries of the form "*does pattern P occur in text T?*" are answered in time proportional to the pattern, rather than the text. Two popular indexing structures are the suffix tree and the suffix array. These data structures enable efficient solutions to many common string problems. Recent work has compressed these data structures, formed dynamic data structures and developed full-text indexes. A full-text index gathers all the relevant information about text so that the actual text can be discarded. It often attains better space complexity than the original text.

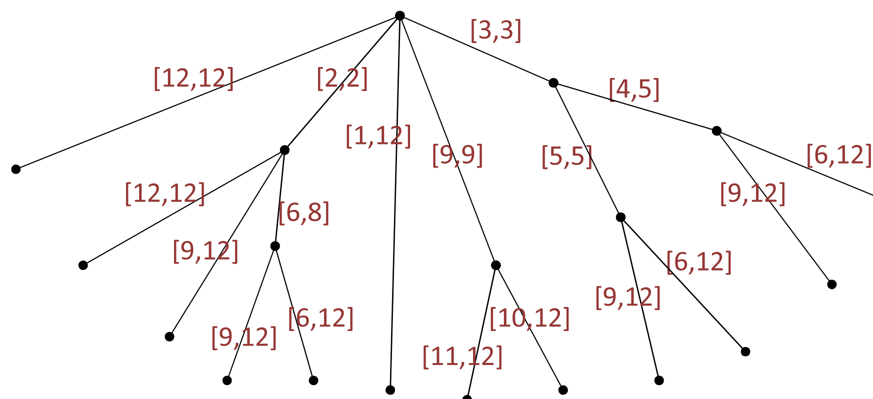


Figure 3.1: Suffix tree for the string Mississippi.

3.3.1 Suffix Tree

The suffix tree is a compact trie that represents all suffixes of the underlying text. The suffix tree for $T = t_1 t_2 \cdots t_n$ is a rooted, directed tree with n leaves, one for each suffix. Each internal node, except the root, has at least two children. Each edge is labeled with a nonempty substring of T and no two edges emanating from a node begin with the same character. The path from the root to leaf i spells out suffix $T[i \ldots n]$. A special character is appended to the string before construction of the suffix tree to guarantee that each suffix ends at a leaf in the tree. The suffix tree for a string of size n is represented in $O(n)$ words, or $O(n \log n)$ bits, by using indexes of constant size, rather than substrings of arbitrary length, to label the edges of the tree. As an example, the suffix tree for Mississippi is shown in Figure 3.1.

A suffix tree can be used to index several patterns. Either the patterns can be concatenated with unique characters separating them or a generalized suffix tree can be constructed. The generalized suffix tree, as described by Gusfield [35], does not mark artificial suffixes that span several patterns. It combines the suffixes of the individual patterns in a single data structure.

The straightforward approach to suffix tree construction inserts each suffix by a sequence of comparisons beginning at the root, in quadratic time with respect to the size of the input. Linear-time construction of the suffix tree, then called the *position tree*, was introduced by Weiner in 1973 [63]. McCreight simplified the construction process in 1976 [50]. Weiner's algorithm scans the text from right to left and begins with the shortest suffix, while McCreight's scans the text from left to right and initializes the data structure with the longest suffix. In the 1990's, Ukkonen provided an online algorithm that constructs the suffix tree in linear time [61]. Ukkonen overcame the problem of extending each suffix on each iteration by introducing a special edge pointer, *, to represent the current end of the string. The development of the linear-time suffix tree construction algorithms and the distinctions between them are described by Giegerich and Kurtz [32].

Suffix links are an implementation trick necessary to achieve linear time and space complexity in suffix tree construction algorithms. Suffix links allow an algorithm to move quickly to a distant part of the tree. A suffix link is a pointer from an internal node labeled xS to another internal node labeled S, where x is an arbitrary character and S is a possibly empty substring. The suffix links of Weiner's algorithm are alphabet dependent as they work in reverse to represent the insertion of any character before a suffix.

Instances arise in which the text indexed by the suffix tree changes over time. The dynamic suffix tree accommodates the insertion or removal of substrings from the underlying

text. The dynamic suffix tree generalizes to represent a set of strings in such a way that strings can be added and removed efficiently. In Choi and Lam's dynamic suffix tree [16], the update operations take time proportional to the string being inserted or removed from the tree. Yet, the tree never stores a reference to a string that has been removed, and the space complexity is bounded by the total length of the strings stored in the tree. They use a two-way pointer for each edge, which is stored in linear space.

3.3.2 Suffix Array

The suffix array indexes text by storing the lexicographic order of its suffixes. The suffix array occupies less space than the suffix tree does. As an example, the suffix array for Mississippi is shown in Table 3.3. Augmented with an LCP array to store the longest common prefix between adjacent suffixes, efficient pattern search is performed in the text.

Suffix	11	8	5	2	1	10	9	7	4	6	3
Index	1	2	3	4	5	6	7	8	9	10	11

Table 3.3: Suffix Array of the string Mississippi

The naive approach to suffix array construction sorts the suffixes using a string sorting algorithm. This ignores the underlying relationship among the suffixes. The worst-case time complexity of such an algorithm is quadratic in the size of the input. A suffix array can be built by preorder traversal of a suffix tree for the same text, and the LCP array by constant-time lowest common ancestor queries on the tree. However, this indirect construction does not achieve better space complexity than the suffix tree, which occupies $O(n \log n)$ bits. Manber and Myers employ a sort and search paradigm to directly construct the suffix array in $O(n \log n)$ time and $O(n)$ bits of space [48]. Three algorithms were introduced in 2003 to directly construct the suffix array in linear time and space [43, 42, 45].

Once the suffix array is available, the longest common prefix (LCP) array can be created in linear time and space using range minimum queries.

3.3.3 Compressed Data Structures

A recent trend in pattern matching algorithms has been to succinctly encode data structures so that they occupy no more space than the data they are built on. These compressed data structures replace the original data, and allow the same queries as their uncompressed counterparts with a very minor time penalty. This research has extended to dynamic ordered trees, suffix trees, and suffix arrays, among other data structures.

Many compressed data structures state their space requirement as a function of the empirical entropy of the indexed text. This is useful because it gives a measure of the index size with respect to the size achieved by the best kth-order compressor, thus relating the index size to the compressibility of the text. Empirical entropy is defined in Section 2.3.

Compressed Suffix Array

Since the suffix array is an array of n indexes, it can be stored in $n \log n$ bits. The compressed suffix array was introduced by Grossi and Vitter [34] to reduce the size of the suffix array from $n \log n$ bits to $O(n \log \sigma)$ bits. This is at the expense of increasing access time from $O(1)$ to $O(\log^{\epsilon} n)$, where ϵ is any constant with $0 < \epsilon < 1$.

Sadakane modified the compressed suffix array so that it is a self-index [58]. That is, the text can be discarded and the index suffices to answer queries as well as to access substrings of the text. He also reduced the size of the structure to $O(n \log H_0(T))$ bits. Pattern matching using his compressed suffix array has the same time complexity as in the uncompressed suffix array. Grossi et al. further reduced its size to $(1+\frac{1}{\epsilon})nH_k(T)+o(n)$ bits, with

character lookup time of $O(\log^{\epsilon} n)$, assuming an alphabet size $\sigma = O(polylog(n))$ [33].

Ferragina and Manzini developed the first compressed suffix array to encode the index size with respect to the high-order empirical entropy [22]. Their self-indexing data structure is known as the FM-index. The FM-index is based on the Burrows-Wheeler transform and uses backward searching. The compressed self-index exploits the compressibility of the text so its size is a function of the compressed text length. Yet, an index supports more functionality than standard text compression. Navarro and Makinen improved the FM-index [51]. They compiled a survey of the various compressed suffix arrays; each offers a different trade-off between the space it requires and the lookup-times it provides [51]. There are also several options for compressing the LCP array.

Compressed Suffix Tree

Recent innovations in succinct full-text indexing provide us with the ability to compress a suffix tree, using no more space than the entropy of the original data it is built upon. These self-indexes can replace the original text, as they support retrieval of the original text, in addition to answering queries about the data, very quickly.

The suffix tree for a string T of length n occupies $O(n)$ words or $O(n \log n)$ bits of space. Several compressed suffix tree representations have been designed and implemented, each with its particular time-space trade-off.

1. Sadakane introduced the first linear representation of suffix trees that supports all navigation operations efficiently [59]. His data structures form a compressed suffix tree in $O(n \log \sigma)$ bits of space, eliminating the $\log n$ factor in the space representation. Any algorithm that runs on a suffix tree will run on this compressed suffix tree with an $O(polylog(n))$ slowdown. It is based on a compressed suffix array (CSA)

that occupies $O(n)$ bits. An implementation has been made publicly available by Välimäki et al. [62].

This compressed suffix tree was adapted for a dynamically changing set of patterns by Chan et al. [14]. They represent the compressed suffix tree as the combination of balanced parentheses, LCP array, CSA and FM index. An edge label is retrieved in $O(\log^2 n)$ time and a substring of size p is inserted or removed from the index in $O(p \log^2 n)$ time.

2. Russo et al. [56] achieved a fully-compressed suffix tree requiring $nH_k(T)+o(n \log \sigma)$ bits of space, which is essentially the space required by the smallest compressed suffix array, and asymptotically optimal under kth order empirical entropy. Although some operations can be executed more quickly, all operations meet $O(\log n)$ time complexity. The data structure reaches an optimal lower-bound of space occupancy. However, traversal is slower than in other compressed suffix trees. The static version of this data structure has been implemented and evaluated by Cánovas and Navarro [13].

   Russo et al. also [56] developed a dynamic fully-compressed suffix tree but it has not yet been implemented. All operations are performed within $O(\log^2 n)$ time. This dynamic compressed suffix tree supports a larger set of suffix tree navigation operations than the compressed suffix tree proposed by Chan et al. [14]. It also reaches a better space complexity and can perform basic operations more quickly.

3. Fischer et al. achieve faster traversal in their compressed suffix tree [25]. Instead

of sampling the nodes, they store the suffix tree as a compressed suffix array (representing the order of the leaves), a longest-common-prefix (LCP) array (representing the string-depth of consecutive leaves) and data structures for range minimum and previous/next smaller value queries. In its original exposition, Fischer et al.'s fully-compressed suffix tree occupies $2H_k(T)(2\log\frac{1}{H_k(T)}+\frac{1}{\epsilon}+O(1))+o(n)$ bits of space [25]. This data structure has been implemented and evaluated by Cánovas and Navarro [13].

With Fischer's new compressed representation of the LCP array [24], the compressed suffix tree of Fischer et al. [25] can be stored in even smaller space. That is, the suffix tree can be stored in $(1+\frac{1}{\epsilon})nH_k(T)+o(n\log\sigma)$ bits of space with all operations computed in sub-logarithmic time. Navigation operations are dominated by the time required to access an element of the compressed suffix array and by the time required to access an entry in the compressed LCP array, both of which are bounded by $O(\log^{\epsilon} n)$, $0<\epsilon\leq 1$.

4. Ohlebusch et al. developed an improved compressed suffix tree representation that can answer queries more quickly [55]. They point out that the compressed suffix tree generally consists of three separate parts: the lexicographical information in a compressed suffix array (CSA), the information about common substrings in the longest common prefix array (LCP), and the tree topology combined with a navigational structure (NAV). Each of these three components functions independently from the others and is stored separately. The fully-functional compressed suffix tree of Russo et al. [56] stores the sampled nodes in addition to these components.

   Ohlebusch et al. developed a mechanism that stores NAV in $3n+o(n)$ bits so that some traversal operations can be performed in constant time, i.e., PARENT, SUFFIX

LINK, and LCA. In addition, they were able to further compress the LCP array. These results have been implemented by Simon Gog and the code is available as part of the Succinct Data Structures Library.

Representations of compressed suffix arrays and compressed LCP arrays are interchangeable in compressed suffix trees. Combining the different variants yields a rich variety of compressed suffix trees, although some compressed suffix trees favor certain compressed suffix array or compressed LCP array implementations [55].

Chapter 4

Succinct 2D Dictionary Matching

In this chapter we present the first algorithm that solves the *Small-Space 2D Dictionary Matching Problem*. Given a dictionary of patterns, $P_1, P_2, \ldots, P_d$, each of size $m \times m$, and a text of size $n \times n$, we find all occurrences of patterns in the text. We discuss patterns that are all of size $m \times m$ for ease of exposition, but as with Bird / Baker, our algorithm can be extended to patterns that are the same size in only one dimension with the complexity dependent on the size of the largest dimension. The initial part of this algorithm appeared in CPM 2010 [53] and the overall techniques were published in Algorithmica [52].

4.1 Overview

In the preprocessing phase, the dictionary is linearized by concatenating the rows of each pattern, with a delimiter separating them, and then concatenating the patterns to form a single string. The linearized dictionary is then stored in an entropy-compressed self-index, allowing the original dictionary to be discarded. The preprocessing phase runs in $O(dm^2)$ time and uses $O(dm \log dm)$ bits of extra space. Let τ be an upper bound on the time

complexity of operations in the self-index and let σ be the size of the alphabet. The text scanning phase takes $O(n^2\tau \log \sigma)$ time and uses $O(dm \log dm)$ bits of extra space.

Our algorithm preprocesses the dictionary of patterns before searching the text once for all patterns in the dictionary. The text scanning stage initially filters the text to a limited number of *candidate* positions and then verifies which of these positions are actual pattern occurrences. We allow $O(dm \log dm)$ bits of working space to process the text and locate patterns in the dictionary. The text scanning stage does not depend on the size of the dictionary. The data structures we use for indexing are dynamic during the pattern preprocessing stage and static during the text scanning stage.

A known technique for minimizing space is to work with small overlapping text blocks of size $3m/2 \times 3m/2$. The potential starts all lie in the upper-left $m/2 \times m/2$ square. This way, the size of our working space relies on the size of the dictionary, not on the size of the text.

We divide patterns into two groups based on 1D periodicity. Our algorithm considers each of these cases separately. A pattern can consist of rows that are periodic with period $\leq m/4$. Alternatively, a pattern can have one or more possibly aperiodic rows whose periods are larger than $m/4$. In each of these cases, the bottlenecks are quite different. In the case of highly periodic pattern rows, a single pattern can overlap itself with several occurrences in close proximity to each other and we can easily have more candidates than the space we allow. In the case of an aperiodic row, there is a more limited number of pattern occurrences, but several patterns can overlap each other in both directions.

A pattern can have only periodic rows with all periods $\leq m/4$ (Case I) or have at least one aperiodic row or a row with a period $> m/4$ (Case II). Case I is addressed in Section 4.2 and Case II is addressed in Section 4.3.

In the case that $d \geq m$, i.e., when $dm = \Omega(m^2)$, we have more space to work with as the text is processed. We can store $O(m^2)$ information for a text block and present a different algorithm for that case in Section 4.3.2.

4.2 Case I: Patterns With Rows of Period Size $\leq m/4$

We store the linearized dictionary in an entropy-compressed form that allows constant time random access to any character in the original data, such as the compression scheme of Ferragina and Venturini [23] or of Fredriksson and Nikitin [27]. For Case I patterns we do not need additional functionality in the self-index, thus we do not construct a compressed suffix tree or suffix array. The space needed for storing the dictionary D in entropy-compressed form is $\ell H_k(D) + \gamma$ where γ is the low-order term,[1] and depends on the particular compression scheme that is employed.

We overcome the extra space requirement of traditional 2D dictionary matching algorithms with an innovative preprocessing scheme that names 2D patterns to represent them in 1D. The pattern rows are initially classified into groups, with each group having a single representative. We store a *witness*, or position of mismatch, between the group representatives. A 2D pattern is named by the group representative for each of its rows. This is a generalization of the naming technique used by Bird and by Baker to name 2D data in 1D. The preprocessing is performed in a single pass over the patterns. O(1) of information is stored per pattern row, occupying a total of O($dm \log dm$) bits of space. Details of the preprocessing stage can be found in Section 4.2.1.

In the text scanning phase, we name the rows of the text to form a 1D representation of the 2D text. Then, we use an Aho-Corasick (AC) automaton to mark candidates of possible

[1] For example, using Ferragina and Venturini [23], $\gamma = O(\frac{\ell}{\log_\sigma \ell}(k \log \sigma + \log \log \ell))$.

pattern occurrences in the 1D text in $O(n^2 \log \sigma)$ time. In this part of the algorithm, σ can be viewed as the size of the alphabet of names if it is smaller than the original alphabet; $\sigma \leq dm$. Since similar pattern rows are grouped together, we need a verification stage to determine if the candidates are actual pattern occurrences. With additional preprocessing of the 1D pattern representations, a single pass suffices to verify potential pattern occurrences in the text. The details of the text scanning stage are described in Section 4.2.2.

4.2.1 Pattern Preprocessing

A dictionary of patterns with highly periodic rows can occur $\Omega(dm)$ times in a text block. It is difficult to search for these patterns in small space since the output can be larger than the amount of extra space we allow. We take advantage of the periodicity of pattern rows to succinctly represent pattern occurrences. The distance between any two overlapping occurrences of P_i in the same row is the Least Common Multiple (LCM) of the periods of all rows of P_i. We precompute the LCM of each pattern so that $O(1)$ space suffices to store all occurrences of a pattern in a row, and $O(dm \log dm)$ bits of space suffice to store all occurrences of patterns whose rows are periodic with periods $\leq m/4$.

We introduce two new data structures, the witness tree and the offset tree. The witness tree facilitates the linear-time preprocessing of pattern rows. The offset tree allows the text scanning stage to achieve linear time complexity, independent of the number of patterns in the dictionary. They are described later in this section.

Lyndon Word Naming

Since conjugacy is an equivalence relation, we can partition the pattern rows into disjoint groups based on the conjugacy of their periods. We use the same name to represent all

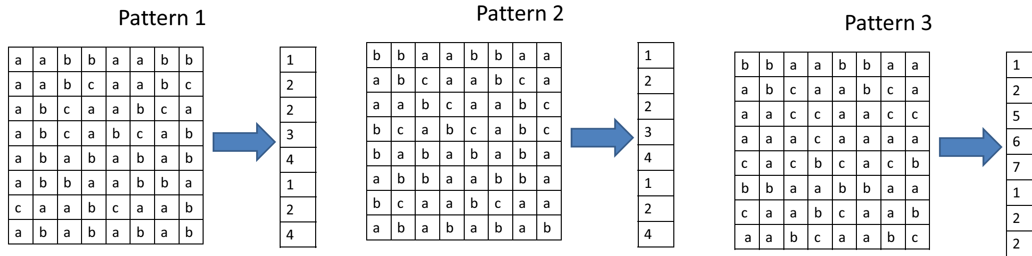


Figure 4.1: Three 2D patterns with their 1D representations. We use these patterns to illustrate the concepts, although their periods are larger than $m/4$. Patterns 1 and 2 are different, yet their 1D representations are the same.

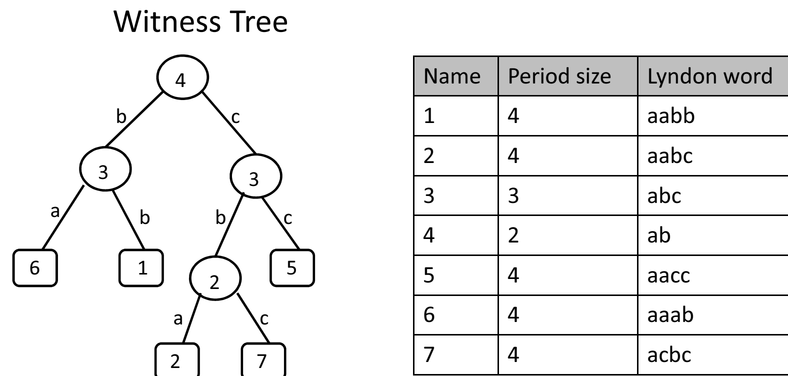
rows whose periods are conjugate. The smallest conjugate of a word, i.e., its Lyndon word, is the standard representation of its conjugacy class. *Canonization* is the process of computing a Lyndon word, and can be done in linear time and space [46]. We name one pattern row at a time by finding its period and canonizing it. If a new Lyndon word or a new period *size* is encountered, the row is given a new name. Otherwise, the row adopts the name already given to another member of its conjugacy class. Each 2D pattern obtains a 1D representation of names in a similar manner to the Bird / Baker algorithm, but using Lyndon word naming. The extra space needed to store the 1D patterns of names is $O(dm \log dm)$ bits.

Three 2D patterns and their 1D representations are shown in Figure 4.1. To understand the naming process, we will look at Pattern 1. The period of the first row is *aabb*, which is four characters long. It is given the name *1*. When the second row is examined, its period is found to be *aabc*, which is also four characters long. *aabb* and *aabc* are both Lyndon words of size four, but they are different, so the second row is named *2*. The period of the third row is *abca*, which is represented by the Lyndon word *aabc*. Thus, the second and third rows are given the same name even though they are not identical.

When naming a pattern row, its period is identified using known techniques in linear

time and space, e.g., using a KMP automaton [44] of the string. Then, we compute and store several discrete pieces of information per row: period size (in $\log m/4$ bits), name (in $\log dm$ bits), and position of the first Lyndon word occurrence in the period, which we call *LYpos* (in $\log m/4$ bits).

We use the *witness tree*, described in the following subsection, to name the pattern rows. A separate witness tree is constructed for each period size. The witness tree allows linear time naming of each Lyndon word by keeping track of failures in Lyndon word character comparisons.



Name	Period size	Lyndon word
1	4	aabb
2	4	aabc
3	3	abc
4	2	ab
5	4	aacc
6	4	aaab
7	4	acbc

Figure 4.2: A witness tree for the Lyndon words of length 4 that are in the table of names.

Witness Tree

Components of witness tree:

- *Internal node:* position of a character mismatch. The position is an integer $\in [1, m]$.

- *Edge:* labeled with a character in the alphabet. Two edges emanating from a node must have different labels.

- *Leaf:* an equivalence class representing one or more pattern rows.

When a new row is examined, we need to determine if the Lyndon word of its period has already been named. The witness tree allows us to identify the only named string of the same size that has no recorded position of mismatch with the new string. Then, the found string is compared sequentially to the new row. A witness tree for Lyndon words of length four is depicted in Figure 4.2.

The witness tree is used as it is constructed in the pattern preprocessing stage. As strings of the same size are compared, points of distinction between the representatives of 1D names are identified and stored in a tree structure. When a mismatch is found between strings that have no recorded distinction, comparison halts, and the point of failure is added to the tree. Characters of a new string are examined in the order dictated by traversal of the witness tree, possibly out of sequence. If traversal halts at an internal node, the string receives a new name. Otherwise, traversal halts at a leaf, and the new string is sequentially compared to the string represented by the leaf.

As an example, we explain how the name *7* became a leaf in the witness tree of Figure 4.2. We seek to classify the Lyndon word *acbc*, using the witness tree for Lyndon words of size four. Since the root represents position 4, the first comparison finds that *c*, the fourth character in *acbc*, matches the edge connecting the root to its right child. This brings us to the right child of the root, which tells us to look at position 3. Since there is a *b* at the third position of *acbc*, we reach the leaf labeled *2*. Thus, we compare the Lyndon words *acbc* and *aabc*. They differ at the second position, so we create an internal node for position 2, with children leading to leaves labeled *2* and *7*, and their edges labeled *a* and *c*, respectively.

Lemma 4.2.1. *Of the named strings that are the same size as a new string, i, there is at most one equivalence class, j, that has no recorded mismatch against i.*

Proof. The proof is by contradiction. Suppose we have two such classes, h and j. Both h and j have the same size as i and neither has a recorded mismatch with i. By transitivity of the equivalence relation, we have not recorded a mismatch between h and j. This means that h and j should have received the same name. This contradicts the assumption that h and j are different classes. □

Lemma 4.2.2. *The witness trees for the rows of* d *patterns, each of size* $m \times m$*, occupies* $O(dm \log dm)$ *bits of space.*

Proof. The proof is by induction. The first time a string of size u is encountered, the tree for strings of size u is initialized to a single leaf. The subsequent examination of a string of size u will contribute either zero or one new node (with an accompanying edge) to the tree. Either the string is given a name that has already been used or it is given a new name. If the string is given a name already used, the tree remains unchanged. If the string is given a new name, it mismatched another string of the same size. There are two possibilities to consider.

(i) A leaf is replaced with an internal node to represent the position of mismatch. The new internal node has two leaves as its children. One leaf represents the new name, and the other represents the string to which it was compared. The new edges are labeled with the characters that mismatched.

(ii) A new leaf is created by adding an edge to an existing internal node. The new edge represents the character that mismatched and the new leaf represents the new name. □

Corollary 4.2.3. *The witness tree for Lyndon words of length* u *has depth* $\leq u$*.*

Lemma 4.2.4. *A pattern row of size* $O(m)$ *is named in* $O(m)$ *time using the appropriate witness tree.*

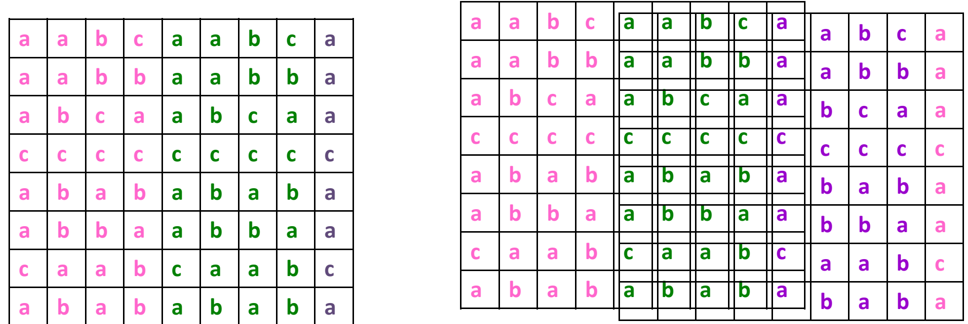


Figure 4.3: An h-periodic matrix with h-period of size 4. The figure on the right shows that the matrix can overlap itself at this distance, with no mismatch in the region of overlap.

Proof. By Lemma 4.2.1, a new string is compared to at most one other string, j. A witness tree is traversed from the root to identify j. Traversal of a witness tree ceases either at an internal node or at a leaf. The time spent traversing a tree is bounded by its depth. By Corollary 4.2.3, the tree-depth is $O(m)$, so the tree is traversed in $O(m)$ comparisons. Thus, a new string is classified with $O(m)$ comparisons.

□

Preprocessing the 1D Patterns

We initially focus on h-periodic patterns in this section. We discuss h-aperiodic at the end of the section.

Definition 4.2.1. *[18] An $m \times m$ matrix is* h-periodic*, or horizontally periodic, if two copies of the matrix can be aligned in the top row so that there is no mismatch in the region of overlap and the number of overlapping columns is $\geq m/2$.*

An h-periodic pattern is depicted in Figure 4.3.

Definition 4.2.2. *The* h-period *of an h-periodic matrix is the minimum column number at*

which the matrix can be aligned over itself.

Observation 1. *If a 2D pattern is h-periodic then each of its rows is periodic. However, the converse is not necessarily true.*

Once the pattern rows are named, an Aho-Corasick (AC) automaton is constructed for the 1D patterns of names. (See Figure 4.1 for the 1D names of three patterns.) Several different patterns have the same 1D name if their rows belong to the same equivalence class. This is easily detected in the AC automaton since the patterns occur at the same terminal state.

The next preprocessing step computes a Least Common Multiple (LCM) table for each distinct 1D pattern. The LCM table stores the LCM of the periods of the first i rows of the pattern in entry $LCM[i]$, for $1 \leq i \leq m$. Each LCM entry can be computed incrementally, one row at a time. $LCM[i]$ is computed from $LCM[i-1]$ and the period of row i. The LCM of two numbers x and y can be found by multiplying x by y and dividing the product by the greatest common divisor of x and y, which can be found in $O(min(x, y))$ time. Thus, each value takes $O(m)$ time and $O(1)$ space to compute and the entire table is constructed in linear time with respect to the size of a pattern.

The LCM of an h-periodic pattern reveals the horizontal distance between its potential occurrences in a text block. This conserves space as there are fewer candidates to maintain. In addition, we use this to conserve verification time. The LCM table of each 1D pattern can be stored in $O(m \log m)$ bits of space since the LCM of an h-periodic pattern must be $\leq m/2$. Thus, the LCM tables occupy $O(dm \log m)$ bits overall.

We say that two distinct patterns are *horizontally consistent* if one pattern can be placed on the other in the first row so that they overlap in at least $m/2$ columns and their overlap is identical. Note that simply having the same 1D representation does not render candidates

horizontally consistent, although it is a necessary condition. Horizontally consistent patterns can be obtained from one another by removing several columns from one end and then extending each row at the other end by continuing its period by the same number of characters. Figure 4.4 depicts a pair of horizontally consistent patterns. Horizontal consistency is determined by the periods of the pattern rows, i.e., their 1D names, and the Lyndon word alignment between the pattern rows. Two distinct patterns can be horizontally consistent even if they are not h-periodic. We discuss this case at the end of the section.

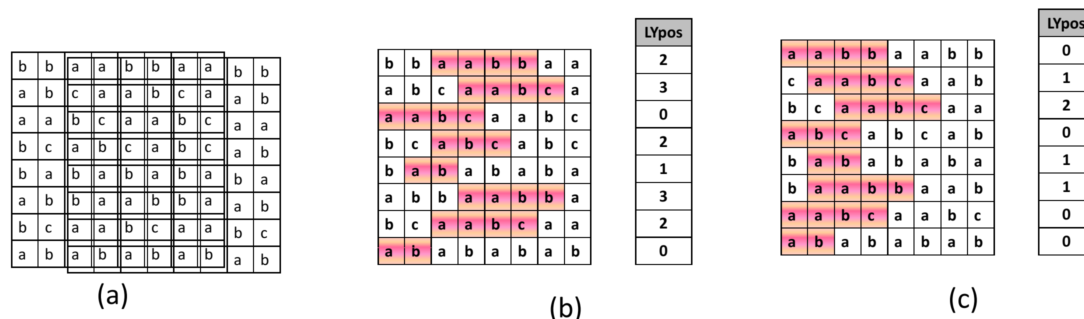


Figure 4.4: (a) Horizontally consistent patterns have overlapping columns: one is a horizontal shift of the other. Each matrix is shown with its *LYpos* array and the first occurrence of the Lyndon word in each row is highlighted. (b) The matrix on the left is Pattern 2 of Figure 4.1. (c) In the matrix on the right, Pattern 2 is shifted left by two columns.

Each row of a pattern is represented by its name and its *LYpos*. To convert a pattern to one that is horizontally consistent with it, its rows are shifted by the same constant, but the *LYpos* of its rows may not be. However, the shift is the same across the rows, relative to the period size of each row. Figure 4.4 shows an example of horizontally consistent patterns and the relative shifts of their rows. Notice that (c) can be obtained from (b) by shifting two columns towards the left. The first occurrence of the Lyndon word of the first row is at the third position in (b) and at the first position in (c). This shift seems to reverse in the third

row, since the Lyndon word first occurs at the first position in (b) and at the third position in (c). However, the relative shift remains the same across all rows, with respect to the period size of each row. We summarize this relationship in the following lemma.

Lemma 4.2.5. *Two patterns with the same 1D representation are* horizontally consistent *iff the* LYPos *of all their rows are shifted by* C mod period size of the row*, where* C *is an integer.*

Proof. Let patterns P_i and P_j be horizontally consistent. Then, their corresponding rows are shifted by the same constant. Matrix P_i is obtained from P_j by removing C columns from the beginning of P_j and then extending each row of P_j by C characters in its period. The *LYpos* of a row is between 1 and the period size of a row. On a row with period size u, a shift of C columns translates to a shift of C mod u. Similarly, if we know that the shift of each row is C mod u, the 2D patterns must be horizontally consistent.

□

We say that a matrix P has a *horizontal prefix* (resp. suffix) U if U is an initial (resp. ending) sequence of contiguous columns in P.

Definition 4.2.3. *Two matrices, P_1 and P_2, are* horizontal 2D conjugate *if $P_1 = UV$, $P_2 = VU$ for some horizontal prefix U and horizontal suffix V of P_1 and the period of every row of P_2 is a 1D conjugate of the period of its corresponding row in P_1.*

Two matrices are horizontal 2D conjugate if they differ only by a cyclic permutation of their columns and the Lyndon word representing each pair of corresponding rows is identical. When it is clear from the context, we simply use the word conjugate to refer to horizontal 2D conjugate.

Observation 2. *If the h-periods of two patterns are horizontal 2D conjugate, then the 2D patterns are horizontally consistent.*

We have shown that two periodic strings that can overlap by more than half their length have periods that are conjugate. There we used the Lyndon word of the period as the representative of each class of consistent periodic strings. In the same way, two h-periodic patterns that are horizontally consistent must have h-periods that are horizontal 2D conjugate. We use horizontal 2D conjugacy as an equivalence relation among h-periodic patterns. Furthermore, we define the *2D Lyndon word* and we use it as the representative of each horizontal consistency class, in a similar manner to the 1D equivalence relation.

We represent a horizontal 2D conjugate as a sequence $c_1, c_2, \ldots, c_m$ where c_i represents the position in row i of the first occurrence of the Lyndon word of the period of row i. The *LYpos* array of a pattern P is the horizontal 2D conjugate sequence of the h-period of P. Each conjugate of the h-period of P will have a distinct sequence which we refer to as the conjugate's *LYpos* array.

Definition 4.2.4. *A* 2D Lyndon word *of a matrix is the* LYpos *array that is the smallest over all the horizontal 2D conjugates of the matrix, for the numerical ordering.*

We use the 2D Lyndon word of the h-period of P to classify pattern P as belonging to exactly one horizontal consistency class. We can compute the 2D Lyndon word of the pattern by computing the *LYpos* array for each conjugate of the h-period and then finding the minimum. This computation can be done in $O(m^2)$ time by generating each *LYpos* array from the pattern's *LYpos* array and the periods of the rows. We maintain only the running minimum and the current sequence as the sequences are generated, and thus $O(m)$ space suffices.

For the preprocessing, these time and space complexities are acceptable. However, when it comes to classifying text blocks, the process of calculating a 2D Lyndon word will have to be more efficient. Therefore, we present an $O(m)$ time algorithm to calculate the 2D Lyndon word for a pattern of size $O(m^2)$. This procedure is delineated in Algorithm 2 and described in the following paragraphs.

We examine one row at a time, and focus on a shrinking subset of shifts at which the 2D Lyndon word can occur. For row i, we begin by shifting the first *LYpos* entry to the first column we are considering. Suppose the first such column is z. Let $u = LCM[i-1]$. Columns $z, z+u, z+2u, \ldots$ are columns at which the 2D Lyndon word can possibly occur. We systematically find the numerically smallest sequence, row by row.

For each row, we use the row's *LYpos* entry to calculate the shifts at the columns that can be the 2D Lyndon word. For each row i there are two possibilities. If its period is a factor of $LCM[i-1]$, the shifted *LYpos* entry will be identical in all columns that we are considering. Otherwise, when $LCM[i]$ is larger than $LCM[i-1]$, we calculate the shifted *LYpos* entry in each column that may be the class representative. Then, we identify the minimum among these values and discard all columns in which the shift in row i is not the minimum. We store the shift at which this minimum value first occurs. The remaining columns that we consider for the 2D Lyndon word are at a distance of $LCM[i]$ from each other. As long as there are several columns we are considering, this continues until either the last row is reached or $LCM[i] = LCM[m]$. Once this occurs, only one shifted *LYpos* value will be computed for each subsequent row.

Observation 3. *Let x and y be distinct integers. If neither x nor y is a factor of the other, $LCM(x, y) \geq 2x$.*

Lemma 4.2.6. *The 2D LYndon word for the h-period of a pattern of size $m \times m$, is computed*

Algorithm 2 Computing a 2D Lyndon Word

Input: $LYpos[1...m], period[1...m], LCM[1...m]$ for matrix M.
Output: 2D Lyndon word, $LW[1...m]$, and its shift z (i.e. column number in M).

$LW[1] \leftarrow 0$
$z \leftarrow$ ***LYpos[1]***
$\{$*LYpos[1]* is first column of shift 0$\}$
$\{$columns z, $z + period[1]$, $z + 2 * period[1]$, $\ldots$ can be 2D Lyndon word$\}$
for $i \leftarrow 2$ to m **do**
  if $LCM[i-1]$ MOD $period[i] = 0$ **then**
    $\{$if period of row i is a factor of cumulative LCM$\}$
    $LW[i] \leftarrow$ (*LYpos[i]*-z) MOD *period[i]*
  else
    $\{LCM[i] > LCM[i-1]\}$
    firstLYshift $\leftarrow$ (*LYpos[i]*-z) MOD *period[i]*
    $\{$shift *LYpos[i]* to $z\}$
    $LW[i] \leftarrow$ min ((*firstLYshift*$-j * LCM[i-1]$) MOD *period[i]*)
    $\{$minimize over $j \geq 0$ such that $z + j * LCM[i-1] \leq LCM[m]\}$
    $z += j * LCM[i-1]$
    $\{$adjust z by j that minimizes shift in previous equation$\}$
  end if
  i++
end for

in $O(m)$ time and $O(m \log m)$ bits of extra space.

Proof. The h-period of a pattern has width $LCM[m] < m$. Thus, we begin with a set of at most $LCM[m]$ columns as possibilities for the 2D Lyndon word. As row i is examined, the if statement in Algorithm 2 has two possibilities:

(i) Its period is a factor of $LCM[i-1]$: computation is done in $O(1)$ time and space.

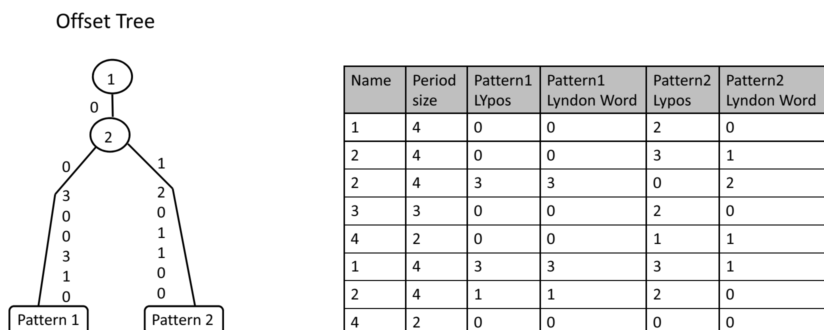
(ii) $LCM[i] > LCM[i-1]$: *LYpos* is shifted for several columns. The values are compared and all but the shifts of minimum value are discarded. When the LCM value is adjusted, the number of columns that we consider is shortened. By Observation 3, at least half the possibilities are discarded. We can charge the computations for shifting *LYpos* values in row i to the set of shifts that are eliminated. Over all rows, at most m columns can be eliminated and thus the time remains $O(m)$.

The 2D Lyndon word of a pattern is stored in $O(m \log m)$ bits of space. Along the way, the only extra information we store are the shifted *LYpos* values for a small number of columns ($\leq m/4$) in one row i at a time, at some initial shift and then at regular intervals of $LCM[i]$.

□

Offset Tree

If several patterns share a 1D name, and their h-periods are not horizontal 2D conjugate, they will each have a unique 2D Lyndon word. We need an efficient way to compare the text to all of these patterns. Thus, we construct a compressed trie labeled by the 2D Lyndon words, which we call an *offset tree*. An example of an offset tree is shown in Figure 4.5.



Name	Period size	Pattern1 LYpos	Pattern1 Lyndon Word	Pattern2 Lypos	Pattern2 Lyndon Word
1	4	0	0	2	0
2	4	0	0	3	1
2	4	3	3	0	2
3	3	0	0	2	0
4	2	0	0	1	1
1	4	3	3	3	1
2	4	1	1	2	0
4	2	0	0	0	0

Figure 4.5: Offset tree for Pattern 1 and Pattern 2 (the first two patterns of Figure 4.1) which have the same 1D name. The *LYpos* entries of Pattern 1 are not shifted since its actual shift is its 2D Lyndon word, while the *Lypos* entries of Pattern 2 are shifted by *2 mod period size of row*, to match the 2D Lyndon word shown in Figure 4.4(c).

Components of offset tree:

- *Root:* represents the first row of a pattern.
- *Internal node:* represents a row index from 1 to m, strictly larger than its parent's.
- *Edge:* labeled by a subsequence of the 2D Lyndon word.
- *Leaf:* represents a consistency class of dictionary patterns.

We construct an offset tree for each set of patterns with the same 1D name. We classify one pattern at a time. Once we have computed the 2D Lyndon word of each pattern, we traverse the tree and compare the 2D Lyndon words in sequential order until either a mismatch is found or we reach a leaf. If a mismatch occurs at an edge leading to a leaf, a new internal node with a leaf are created, to represent the position of mismatch and the new consistency class, respectively. If a mismatch occurs at an edge leading to an internal node, a new branch is created with a new leaf to represent the new consistency class.

Observation 4. *The offset trees for d 1D patterns, each of size m, have $O(d)$ nodes and thus can be stored in $O(d \log d)$ bits of space.*

Lemma 4.2.7. *Given a set of 2D Lyndon words that are representative of different consistency classes, we can classify a 2D Lyndon word in $O(m)$ time.*

Proof. The offset tree for a 1D pattern of length m has string-depth $\leq m$. This is because each node represents a position from 1 to m and each node represents a position strictly greater than that of its parent. A pattern is classified by traversing the offset tree and comparing the Lyndon word offsets labeling the edges until either a point of failure or a leaf is reached. Since a tree of string-depth $\leq m$ is traversed from the root in $O(m)$ time, a 2D Lyndon word of length m is classified in $O(m)$ time.

□

h-Aperiodic patterns

Thus far we have classified only h-periodic patterns into horizontal consistency classes. If a pattern is not h-periodic it can still overlap another non h-periodic pattern with more than $m/2$ columns. The LCM of the periods of all rows, $LCM[m]$, tells us whether a pattern is h-periodic or not so we can easily split Case I patterns into those that are h-periodic and those that are not h-periodic. If two h-aperiodic patterns are horizontally consistent, their periods have the same relative shifts over all their rows. Since the 2D Lyndon word captures the relative shifts of all pattern rows, we can use a similar technique to classify h-aperiodic patterns whose rows are highly periodic.

We assume that each dictionary pattern whose rows are highly periodic has an LCM that is $O(m)$. This ensures that in the standard RAM model, standard arithmetic on the LCM table can be performed in constant time. Note that the 2D Lyndon word may not

be the *LYpos* array for any conjugate of an h-aperiodic pattern. However, if we artificially enlarge the pattern by extending the periods, we will eventually reach a column that is minimal over all possible columns with these relative shifts. We will never need to enlarge a pattern beyond $LCM[m] = O(m)$ and therefore the time complexity remains as stated.

In summary, pattern preprocessing in $O(dm^2)$ time and $O(dm \log dm)$ bits of space:

1. For each pattern row,
   (a) Compute period and canonize.
   (b) Store period size, name, first Lyndon word occurrence (*LYpos*).
2. Construct AC automaton of 1D patterns.
3. Compress dictionary. Can discard original dictionary.
4. For each 1D pattern of names,
   (a) Compute LCM table.
   (b) Compute 2D Lyndon word. Store shift at which it occurs.
5. Build offset tree for multiple patterns of same 1D name.

4.2.2 Text Scanning

The text scanning stage has three steps.

1. Name rows of text.
2. Identify candidates with a 1D dictionary matching algorithm, e.g. AC.

3. Verify candidates separately for each text row using the offset trees of the 1D patterns.

Step 1. Name Text Rows

We search a 2D text for a 1D dictionary of patterns using a 1D Aho-Corasick (AC) automaton. A 1D pattern can begin at any of the first $m/2$ positions of a text block row. The AC automaton can branch to one of several characters; we can't afford the time or space to search for each of them in the text block row. Thus, we name the rows of a text block before searching for patterns. The divide-and-conquer algorithm of Main and Lorentz finds all maximal repetitions in linear time, searching for repetitions to the right and to the left of the midpoint of a string [47]. Repetitions of length $\geq m$ that cross the midpoint and have a period size $\leq m/4$ are the only ones that are of interest to our algorithm.

Lemma 4.2.8. *At most one maximal periodic substring of length $\geq m$ with period $\leq m/4$ can occur in a text block row of size $3m/2$.*

Proof. The proof is by contradiction. Suppose that two maximal periodic substrings of length m, with period $\leq m/4$ occur in a row. Call the periods of these strings u and v. Since we are looking at periodic substrings that begin within an $m/2 \times m/2$ square, the two substrings overlap by at least $m/2$ characters. Since u and v are no larger than $m/4$, at least two adjacent copies of both u and v occur in the overlap. This contradicts the fact that both u and v are primitive.

□

After finding the only maximal periodic substring of length $\geq m$ with period $\leq m/4$, the text block rows are named in much the same way as the pattern rows are named. The

period of the maximal run is found and canonized. Then, the appropriate witness tree is used to name the text block row. We use the witness tree constructed during pattern preprocessing since we are only interested in identifying text block rows that correspond to Lyndon words found in the pattern rows. At most one pattern row will be examined to classify the conjugacy class of a text block row. In addition to the name, period size, and *LYpos*, we maintain a *left* and a *right* pointer for each row of a text block. *left* and *right* mark the endpoints of the periodic substring in the text. This process is repeated for each row, and $O(m)$ information is obtained for the text block.

Complexity of Step 1: The largest periodic substring of a row of width $3m/2$, if it exists, can be found in $O(m)$ time and space [47]. Its period can be found and canonized in linear time and space [46]. The row is named in $O(m)$ time and space using the appropriate witness tree (Lemma 4.2.4). Overall, $O(m^2)$ time and $O(m \log dm)$ bits of space are needed to name the rows of a text block.

Step 2. Identify Candidates

After Step 1 completes, a 1D text remains, each row labeled with a name, period size, *LYpos*, and *left*, *right* boundaries. A 1D dictionary matching algorithm, such as AC, is used to mark occurrences of 1D patterns. We call a text row at which a 1D pattern occurs a *candidate row*. The occurrence of a 1D pattern indicates the potential occurrence of one or more 2D patterns since several 2D dictionary patterns can have the same 1D name. We do not store individual candidate positions within a candidate row since the set of positions is $O(m^2)$ in size and can. occupy too much space.

Complexity of Step 2: 1D dictionary matching in a string of size m is accomplished in $O(m \log \sigma)$ time with $O(dm \log dm)$ bits of space for the AC automaton.

Step 3. Verify Candidates

The occurrence of a 1D pattern is not sufficient evidence that a 2D pattern actually occurs since several patterns can share a 1D representation. We need to verify the alignment of the periods among rows as well as the overall width of the 1D names. We verify each candidate row in $O(m)$ time using the concept of horizontal consistency and the offset tree of 2D Lyndon words.

A segment of the text block that is m rows long is classified by its 2D Lyndon word in the same way that the patterns are. This classification determines horizontal consistency of candidates in a row. The 2D Lyndon word of m rows in a text block is computed and classified with the offset tree in $O(m)$ time. This allows the text scanning stage to complete in time proportional to the text size, independent of the dictionary size.

We must confirm that the labeled periodic string extends over at least m columns in each of the m rows that follow a candidate row. We are interested in the minimum of all *right* pointers, *minRight*, as well as the maximum of all *left* pointers, *maxLeft*, as this is the range of positions in which the pattern(s) can occur. If the pattern will not fit between *minRight* and *maxLeft*, i.e., *minRight* $-$ *maxLeft* $< m$, the candidate row is eliminated.

At this point, we know which patterns are horizontally consistent with the text block row. The last step is to locate the positions at which each pattern begins, within the row. Since we store the shift at which the 2D Lyndon word occurs in a pattern, we can reverse the shift to find the location at which each pattern begins in the text. The reverse shift is computed for each pattern that is horizontally consistent with the text. Let z be the position at which we expect a pattern to begin in the text block row. As long as $z < maxLeft$, we increment z by the LCM of the pattern. Then, we announce position z as a pattern occurrence when $minRight - z \geq m$. Subsequent occurrences of the pattern in the same

row are separated by a distance of $LCM[m]$ columns.

Complexity of Step 3: $O(m)$ rows in a text block can contain candidates. At each candidate row, the 2D Lyndon word is computed and classified in $O(m)$ time, by Lemmas 4.2.6 and 4.2.7. The computation of *maxLeft* and *minRight* for the m rows that a pattern can span takes $O(m)$ time. The actual locations of a pattern occurrence are also determined in $O(m)$ time. Overall, a text block is verified in $O(m^2)$ time, proportional to the size of a text block. The verification process requires $O(m \log dm)$ extra bits of space.

Complexity of Text Scanning Stage: Each block of text is processed separately in $O(m)$ space and in $O(m^2 \log \sigma)$ time. Since the text blocks are $O(m^2)$ in size, there are $O(n^2/m^2)$ blocks of text. Overall, $O(n^2 \log \sigma)$ time and $O(m \log dm)$ extra bits of space are required to process a text of size $n \times n$.

4.3 Case II: Patterns With Row of Period Size $> m/4$

We consider the case of a dictionary of patterns in which each pattern has at least one aperiodic row. The case of a pattern having a row that is periodic with period size between $m/4$ and $m/2$ can be treated similarly, since each pattern can occur only $O(1)$ times on one row of a text block.

In the case of one or more aperiodic pattern rows in the patterns, many different patterns can overlap in a text block row. As a result, it is difficult to employ a naming scheme to find all occurrences of patterns. However, it is straightforward to initially identify a limited number of candidates of pattern occurrences. Verification of these candidates in one pass over the text presented a difficulty.

We allow $O(dm \log dm)$ bits of space to process a block of text. In the event that $d < m$, Case IIa, this limit on space is a significant constraint. We address this case in

Section 4.3.1. When $d \geq m$, Case IIb, the number of candidates for pattern occurrences can exceed the size of a text block. It is difficult to verify such a large number of candidates in time proportional to the size of a text block. Because we allow working space larger than the size of a text block, there is no need to begin by filtering the text and identifying a limited set of candidate positions. We present a different algorithm to handle this case in Section 4.3.2.

For Case II patterns, we again *linearize* the dictionary by concatenating the rows of all patterns, inserting a delimiter at the end of each row. We then replace the original dictionary by storing an entropy-compressed self-index of the linearized dictionary. For Case IIa, a compressed suffix array (CSA) and compressed LCP array encapsulate sufficient information for our dictionary matching algorithm. However, in Case IIb, we need the ability to traverse the compressed suffix tree. For consistency, we discuss the usage of a compressed suffix tree in both cases. The time-space trade-offs of various compressed suffix tree representations are described in Section 3.3.3. Any compressed suffix tree representation can be used in this algorithm. We use τ to refer to the time complexity of operations in the compressed suffix tree.

4.3.1 Case IIa: $d < m$

The aperiodic row (or row with period $> m/4$) of each pattern can only occur $O(1)$ times in a text block row. Thus, we use an aperiodic row of each pattern to filter the text block. The text scanning stage first identifies a small set of positions that are candidates for pattern occurrences. Then, the verification stage determines which of these candidates are actual pattern occurrences. After preprocessing the dictionary, text scanning proceeds in time proportional to the text block size.

Pattern Preprocessing

We form an AC automaton of one aperiodic row of each pattern, say, the first aperiodic row of each pattern. There can be $O(1)$ candidates for any non-periodic row in a text block row. In total, there can be $O(dm)$ candidates in a text block, with candidates for several distinct 1D patterns on a single row of text. If the same aperiodic row occurs in several patterns, we can even find several candidates at the same text position.

The pattern rows are named to form a 1D dictionary of patterns. Distinct rows are given different names, much the same way that Bird and Baker convert a 2D pattern to a 1D representation. However, Bird and Baker form an AC automaton of all pattern rows. We do not allow that much space. Instead, we use a witness tree, described in Section 4.2.1, to store distinctions between the pattern rows, which are all strings of length m. The witness tree of the row names is preprocessed for Lowest Common Ancestor (LCA) to provide a witness between any pair of distinct pattern rows in constant time.

Preprocessing proceeds by indexing the 1D patterns. We form a generalized suffix tree of the 1D patterns of names, complete with suffix links. The suffix tree is preprocessed for LCA to allow $O(1)$ time Longest Common Prefix (LCP) queries between suffixes of the 1D patterns.

In summary, pattern preprocessing is as follows:

1. Construct AC automaton of first aperiodic row of each pattern. Store row number of each of these aperiodic rows.

2. Name pattern rows using a single witness tree. Store 1D patterns of names.

3. Preprocess witness tree for LCA.

4. Construct generalized suffix tree of 1D patterns. Preprocess for LCA.

Lemma 4.3.1. *The pattern preprocessing stage for Case IIa completes in $O(dm^2)$ time and $O(dm \log dm)$ extra bits of space.*

Proof. 1. The AC automaton of the first non-periodic row of each pattern is constructed in $O(dm)$ time and is stored in $O(dm \log dm)$ bits.
2. By Lemma 4.2.2, the witness tree occupies $O(dm \log dm)$ bits of space. By Lemma 4.2.4, pattern rows are named with the witness tree in $O(dm^2)$ time.
3. The witness tree and generalized suffix tree are preprocessed in linear time to answer LCA queries in $O(1)$ time [36, 10].
4. The 1D dictionary of names is stored in $O(dm \log dm)$ bits of space and its generalized suffix tree is constructed and stored in time and space proportional to this 1D representation.

□

Text Scanning

The text scanning stage has three steps.

1. Identify candidates in text block with 1D dictionary matching of a non-periodic row of each pattern.

2. Duel to eliminate vertically inconsistent candidates.

3. Verify pattern occurrences at surviving candidate positions.

Step 1. Identify Candidates

We do not name all text positions as Bird and Baker do, since this would require $O(m^2)$ space per text block. Neither do we use the witness tree to name the text block rows as we do for the patterns whose rows are highly periodic, since many names can overlap in a text block row. Instead, text scanning begins by identifying a limited set of positions that are candidates for pattern occurrences. Unlike patterns in the first group (Case I), each pattern can only occur $O(1)$ times in the text block.

We locate the first aperiodic row of each pattern and consider this set of strings as a 1D dictionary of patterns. $O(dm)$ candidates are found by performing 1-D dictionary matching, e.g. AC, on this limited set of pattern rows over the text block, row by row. Then we update each candidate to point to the position at which we expect a 1D pattern name to begin. This is done by subtracting the row number of the selected aperiodic row within the pattern from the row number of the candidate in the text block.

Complexity of Step 1: 1D dictionary matching on a text block takes $O(m^2 \log \sigma)$ time with the AC method. Marking the positions at which patterns can begin is done in constant time per candidate found; overall, this requires $O(dm)$ time. The AC 1D dictionary matching algorithm uses extra space proportional to the dictionary, $O(dm \log dm)$ bits of space. The candidates can also be stored in $O(dm \log dm)$ bits of space.

Step 2. Eliminate Vertically Inconsistent Candidates

We call a pair of patterns *consistent* if they can overlap in a single text block. Overlapping segments of consistent candidates can be verified simultaneously. In this stage we eliminate inconsistent candidates with a dueling technique inspired by the 2D *single* pattern matching algorithm of Amir et al. [2]. In the single pattern matching algorithm, a witness table is

computed in advance, and duels are performed between candidates of the same pattern. In dictionary matching, we want to perform duels between candidates for *different* patterns. It would be inefficient both in terms of time and space to store witnesses between all locations in all patterns.

We call two patterns *vertically consistent* if they can overlap in the same column. Note that vertically consistent patterns have a suffix/prefix match in their 1D representations. Thus, we duel between candidates within each column using *dynamic dueling*. In dynamic dueling, no witness locations are computed in advance. We are given two candidate patterns and their locations, candidate A at location (i, j) in the text and candidate B at location (k, j) in the text, $i \leq k$. Since all of our candidates are in an $m/2 \times m/2$ square, we know that there is overlap between the two candidates.

A dynamic duel consists of two steps. In the first step, the 1D representation of names is used for A and B, denoted by A' and B'. An LCP query between the suffix $k-i+1$ of A' against B' returns the number of overlapping rows that match. If this number is $\geq i+m-k$ then the two candidates are consistent. Otherwise, we are given a "row-witness," i.e. the LCP points to the first row at which the patterns differ. In the second step of the duel, an LCA query in the witness tree provides a position of mismatch between the two different pattern rows, and we use that position to eliminate one or both candidates.

Text block columns are scanned top-down, one at a time, to determine vertical consistency of candidates. We confirm consistency pairwise over the candidates within a column, since consistency is a transitive relation. A duel eliminates at least one element of a pair of inconsistent candidates. If only the lower candidate is killed, this does not affect the consistent candidates above it in the same column, as they are still consistent with the text

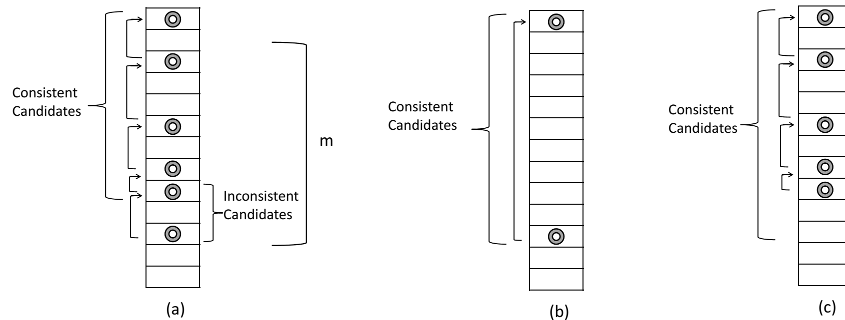


Figure 4.6: (a) Duel between vertically inconsistent candidates in a column. (b) Surviving candidates if the lower candidate wins the duel. (c) Surviving candidates if the upper candidate wins the duel.

character. However, if the lower candidate survives, it triggers the elimination of all candidates within m rows above it. Pointers link consecutive candidates in each column. This way, a duel eliminates the set of consistent candidates that are within range of the mismatch. This is shown in Figure 4.6. Distinct patterns have different 1D representations. Thus, the same method can be used when two (or more) candidates occur at a single text position.

The pass over the text to check for consistency ensures that candidates within each column are vertically consistent. Consistency in other directions (including horizontal consistency) is established in Step 3 while comparing characters sequentially against the text.

Complexity of Step 2: The consistency of a pair of candidates is determined by an LCP query followed by a duel between characters. We use data structures that can answer LCP queries in $O(1)$ time over the 1D patterns of names. Duels are performed with witnesses generated by an LCA query in the witness tree over the pattern rows in $O(1)$ time. Due to transitivity, the number of duels will be no more than the number of candidates. There are $O(dm)$ candidate positions, with $d < m$, so this step completes in $O(m^2\tau)$ time. The only

slowdown is in retrieving pattern characters for the duel. (This is true even in the event that several candidates occur at the same text position.)

Step 3. Verify Surviving Candidates

After eliminating vertically inconsistent candidates, we verify pattern occurrences in a single scan of the text block. Beginning at the first candidate position, characters in the text block are compared sequentially to the expected characters in the appropriate pattern. If two candidates overlap in a text block, we compare the overlapping text characters to a substring of only one pattern row, to verify them simultaneously.

Before we scan a text block row, we mark the positions at which we expect to find a pattern row, by carrying candidates from one row to the next and merging this with the list of candidates that begin on the new row. Then, the text block row is scanned sequentially, comparing one text character to one pattern character at a time, until a pattern row of another candidate is encountered. Then we perform an LCP query over the pattern row that is currently being used for verification and the pattern row that is expected to begin. If the distance between the candidates is smaller than the LCP, a duel resolves the inconsistency among candidates.

Since consistency is transitive, duels are performed on pairs of candidates. Yet, there are times at which the detection of an inconsistency must eliminate several candidates. If several LCP queries have already succeeded in a row (that is, we have a set of consistent patterns), and then we encounter a failure, we eliminate all candidates that are consistent with the candidate that lost and are within range of the mismatch. As in the search for vertical consistency, we chain candidates to facilitate this process.

Consider Figure 4.7. Suppose we are at position π in row α of pattern P_1 and we

approach the expected beginning of row β in pattern P_2. An LCP query on suffix π of α and the entire β determines if they can overlap at this distance. Let the LCP of these substrings be l_1 and the distance between α and β in the text be l_2. Suppose $l_1 < m - l_2$. That is, the mismatch is within the expected overlap. Then we can duel between the candidates using the text character at $\pi + l_1$ to eliminate one or both candidates. However, if $l_1 = m - l_2$, the text can be compared to a substring of either pattern row since these substrings are identical.

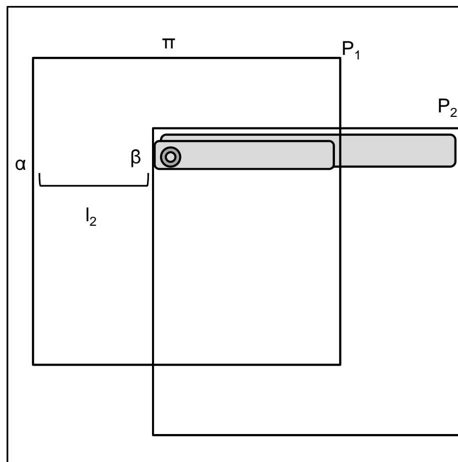


Figure 4.7: Consistency is determined by LCP queries and duels.

Complexity of Step 3: Time Complexity: Each text block character that is within an anticipated pattern occurrence is scanned once and compared to a pattern character, yielding $O(m^2\tau)$ time. When a new label is encountered on a row, a duel is performed. Each duel consists of an LCP query on the compressed suffix tree, which is done in $O(\tau)$ time. Since each candidate can only be eliminated once, transitivity of dueling ensures that the number of duels is $O(dm)$, which is strictly smaller than the size of the text block when $d < m$.

Space Complexity: When a text block row is verified, we mark positions at which a pattern row (1D name) is expected to begin. These labels can be discarded after the row has been verified and the information is carried to the next row. Thus, the space needed is proportional to the number of candidates, plus the labels for one text row, $O(dm \log dm)$ bits.

Lemma 4.3.2. *The text scanning stage for Case IIa, when $d < m$, completes in $O(n^2\tau \log \sigma)$ time and $O(dm \log dm)$ bits of space, in addition to the entropy compressed self-index of the linearized dictionary.*

Proof. This follows from the complexity of Steps 1, 2, and 3.

□

4.3.2 Case IIb: $d \geq m$

Since $d \geq m$ and our algorithm allows $O(dm \log dm)$ extra bits of space, we have $\Omega(m^2)$ space available. This allows us to store information proportional to the size of the text block. In its original form, the Bird / Baker algorithm uses an Aho-Corasick automaton to name the pattern rows and the text positions. We can implement a similar algorithm to name the pattern rows and the text positions if we use a smaller-space mechanism to determine the names.

We can name the text positions using the compressed suffix tree of pattern rows in much the same way as an AC automaton. With suffix links, we name the positions of the text block, row by row, according to the names of pattern rows. Beginning at the root of the tree, we traverse the edge whose label matches the first character of the text block row. When m consecutive characters trace a path from the root, and traversal reaches a leaf, the position is named with the appropriate pattern row. At a mismatch, we traverse suffix links

to find the longest suffix of the already matched string that matches a prefix of a pattern row and compare the next text character to that labeled edge of the tree. With suffix links, this is done in time proportional to the number of characters that have already matched a path from the root of the tree. This is done in the spirit of Ukkonen's online suffix tree construction algorithm which runs in linear time [61].

After naming text positions at which a pattern row occurs, 1D dictionary matching is used to find actual occurrences of the 2D patterns in the text block. We mention the usage of an Aho-Corasick (AC) automaton of the linearized patterns but any 1D dictionary matching algorithm can be used as a black box.

Lemma 4.3.3. *The algorithm for 2D dictionary matching in Case IIb, when $d \geq m$, completes in $O(n^2\tau \log \sigma)$ time and $O(dm \log dm)$ bits of space, after constructing and storing the entropy compressed self-index of the linearized dictionary in $O(m^2\tau)$ time.*

Proof. It suffices to show that the procedure completes in $O(m^2\tau \log \sigma)$ time for a text block of size $3m/2 \times 3m/2$. The algorithm names the text positions by traversing the compressed suffix tree of the dictionary in $O(m^2\tau \log \sigma)$ time and then locates occurrences of the 1D patterns of names with 1D dictionary matching in $O(m^2)$ time. Our algorithm uses an AC automaton of the dictionary of 1D pattern names and a compressed suffix tree of the linearized dictionary. $O(dm \log dm)$ bits of space suffice to store an AC automaton of the 1D patterns of names. A compressed self-index and compressed suffix tree can be stored in entropy compressed space [24]. After forming the two data structures, $O(m^2 \log dm)$=$O(dm \log dm)$ bits of space are used to name a text block. □

Theorem 4.3.4. *Our algorithm for 2D dictionary matching completes in $O(dm^2+n^2\tau \log \sigma)$ time and $O(dm \log dm)$ bits of extra space.*

Proof. Our algorithm is divided into several cases.

Case I: pattern rows are all periodic with period $\leq m/4$.

The complexity of the pattern preprocessing stage is summarized in Section 4.2.1 and the complexity of the text scanning stage is summarized in Section 4.2.2. Both of them meet the bounds specified by this theorem.

Case II: at least one pattern row is aperiodic or has period $> m/4$.

Case IIa: $d < m$. The complexity is summarized in Lemmas 4.3.1 and 4.3.2.

Case IIb: $d \geq m$. The complexity is summarized in Lemma 4.3.3.

□

4.4 Data Compression

The *compressed pattern matching problem* seeks all occurrences of a pattern in text, and works with a pattern and text that are stored in compressed form. Amir et. al. presented an algorithm for strongly-inplace single pattern matching in 2D LZ78-compressed data [7]. They define an algorithm as *strongly inplace* if the extra space it uses is proportional to the optimal compression of the data. Their algorithm preprocesses the pattern of uncompressed size $m \times m$ in $O(m^3)$ time and searches a text of uncompressed size $n \times n$ in $O(n^2)$ time. Our preprocessing scheme can be applied to their algorithm to achieve an optimal $O(m^2)$ preprocessing time, resulting in an overall time complexity of $O(m^2 + n^2)$.

In the *compressed dictionary matching* problem, the input is in compressed form and one would like to search the text for all occurrences of any element of a *set* of patterns. Case I of our algorithm, for patterns with rows of periods $\leq m/4$, is both linear time and strongly inplace. It can be used for 2D compressed dictionary matching when the patterns and text are compressed by a scheme that can be sequentially decompressed in small space.

For example, LZ78 [64] has this property.

Our algorithm is strongly inplace since it uses $O(dm \log dm)$ bits of space and this is the best that can be achieved by a scheme that linearizes each 2D pattern row-by-row. Case I of our algorithm requires only $O(1)$ rows of the pattern or text to be decompressed at a time so it is suitable for a compressed context. A strongly-inplace 2D dictionary matching algorithm for the case in which a pattern row is aperiodic remains an open problem.

Chapter 5

Succinct 2D Dictionary Matching With No Slowdown

In this chapter we achieve succinct 2D dictionary matching in strictly linear time, with no slowdown. We extend new developments in succinct 1D dictionary matching to the two-dimensional setting, in a way similar to the Bird and Baker (BB) extension of the Aho-Corasick 1D dictionary matching algorithm (AC). This problem is not trivial, due to the small space that we allow to index the dictionary and the necessity to label each position of the text. However, we modify the technique of dynamic dueling to make use of recent achievements in 1D dictionary matching, thus eliminating the slowdown in searching the text for patterns whose rows are not highly periodic. We indeed achieve a linear time algorithm that solves the *Small-Space 2D Dictionary Matching Problem*. This algorithm was published in WADS 2011 [54].

Given a dictionary D of d patterns, $D = \{P_1, \ldots, P_d\}$, each of size $m \times m$, and a text T of size $n \times n$, our algorithm finds all occurrences of P_i, $1 \leq i \leq d$, in T. During the preprocessing stage, the patterns are stored in entropy compressed form, in

$|D|H_k(D) + O(|D|)$ bits. $H_k(D)$ denotes the kth order empirical entropy of the string formed by concatenating all the patterns in D, row by row. Preprocessing completes in $O(|D|)$ time using $O(dm \log dm)$ bits of extra space. Then, the text is searched in $O(|T|)$ time using $O(dm \log dm)$ bits of extra space. For ease of exposition, we discuss patterns that are all of size $m \times m$, however, our algorithm generalizes to patterns that are the same size in only one dimension, and the complexity would depend on the size of the largest dimension. As in [9] and [37], the alphabet can be non-constant in size.

This chapter is organized as follows. We begin by reviewing the dictionary matching version of the Bird / Baker algorithm [11, 8] and Hon et al.'s succinct 1D dictionary matching algorithm with no slowdown [37]. We outline how it is possible to combine these algorithms to yield a linear time yet small space 2D dictionary matching algorithm for certain types of patterns. Then, in Section 5.2, we distinguish between highly periodic patterns and non-periodic patterns. We follow the approach we developed in Chapter 4 for the periodic case since it is a suitable linear time algorithm. In Sections 5.2.1 and 5.2.2 we deal with the non-periodic case, and introduce an algorithm that achieves linear time complexity.

5.1 Large Number of Patterns

The first linear-time 2D pattern matching algorithm was developed independently by Bird [11] and by Baker [8], which we henceforth refer to as BB. Although the BB algorithm was initially presented for a single pattern, it is easily extended to perform dictionary matching. Algorithm 3 is an outline of the dictionary matching version of BB.

Space Complexity of BB: We use s to denote the number of states in the Aho-Corasick automaton of D, AC1. Note that $s \leq |D| = dm^2$. $O(s \log s)$ bits of space are needed to

Algorithm 3 Dictionary Matching Version of Bird / Baker Algorithm

{1} Preprocess Pattern:
a) Form Aho-Corasick automaton of pattern rows, called AC1.
Let s denote the number of states in AC1.
b) Name pattern rows using AC1, and store a 1D pattern of names
for each pattern in D, called D'.
c) Construct AC automaton of D', called AC2.
Let s' denote the number of states in AC2.
{2} Row Matching:
Run Aho-Corasick on each text row using AC1.
This labels positions at which a pattern row ends.
{3} Column Matching:
Run Aho-Corasick on named text columns using AC2.
Output pattern occurrences.

store AC1, and labeling all text locations uses $O(n^2 \log dm)$ bits of space. Overall, the BB algorithm uses $O(s \log s + n^2 \log dm)$ bits of space.

Our objective is to improve upon this space requirement. As a first attempt to conserve space, we replace the traditional AC algorithm in Step 1 of BB with the compressed AC automaton of Hon et al. [37]. The algorithm of [37] indexes the 1D dictionary in space that meets the information-theoretic lower bounds of the dictionary. At the same time, it achieves optimal time complexity. We can use their algorithm as a black box replacement for the AC automata in both Steps 1a and 1c of the BB algorithm.

To reduce the algorithm's working space, we work with small overlapping text blocks of size $3m/2 \times 3m/2$. This way, we can replace the $O(n^2 \log dm)$ bits of space used to label the text in Step 3 with $O(m^2 \log dm)$ bits of space, relating the working space to the size of the dictionary, rather than the size of the entire text.

Theorem 5.1.1. *We can solve the 2D dictionary matching problem in linear* $O(dm^2 + n^2)$ *time and* $sH_k(D) + s'H_k(D') + O(s + m^2 \log dm)$ *bits of space.*

Proof. Since the algorithm of Hon et al. [37] has no slowdown, replacing the AC automata in BB with compressed AC automata preserves the linear time complexity. Preprocessing uses $sH_k(D) + O(s) + s'H_k(D') + O(s')$ bits of space. The compressed AC1 automaton uses $sH_k(D) + O(s)$ bits of space and it replaces the original dictionary, while the compressed AC2 automaton uses $s'H_k(D') + O(s')$ *extra* bits of space. Text scanning uses $O(m^2 \log dm)$ extra bits of space to label each location of a text block.

□

Although this is an improvement over the space required by the uncompressed version of BB, we would like to improve on this further. Our aim is to reduce the working space to $O(dm \log dm)$ bits, thus completely eliminating the dependence of the working space on the size of the given text. Yet, note that this constraint still allows us to store $O(1)$ information per pattern row to linearize the dictionary in the preprocessing. In addition, we will have the ability to store $O(1)$ information about each pattern per text row to allow linearity in text scanning.

The following corollary restates Theorem 5.1.1 in terms of $O(dm \log dm)$ for the case of a dictionary with many patterns. It also omits the term $s'H_k(D')+O(s')$, since $s'H_k(D')+O(s') = O(dm \log dm)$.

Corollary 5.1.2. *If $d \geq m$, we can solve the 2D dictionary matching problem in linear $O(dm^2 + n^2)$ time and $sH_k(D) + O(s) + O(dm \log dm)$ bits of space.*

5.2 Small Number of Patterns

The rest of this chapter deals with the case in which the number of patterns is smaller than the dimension of the patterns, i.e., $d = o(m)$. For this case, we cannot label each text

location and therefore the Bird and Baker algorithm cannot be applied trivially. We present several clever space-saving tricks to preserve the spirit of Bird and Baker's algorithm without incurring the necessary storage overhead.

There are two types of patterns, and each one presents its own difficulty. In the first type, which we call Case 1 patterns, all rows are periodic, with periods $\leq m/4$. The difficulty in this case is that many overlapping occurrences can appear in the text in close proximity to each other, and we can easily have more candidates than the working space we allow. The second type, Case 2 patterns, have at least one aperiodic row or one row whose period is larger than $m/4$. Here, each pattern can occur only $O(1)$ times in a text block. Since several patterns can overlap each other in both directions, a difficulty arises in the text scanning stage. We do not allow the time to verify different candidates separately, nor do we allow space to keep track of the possible overlaps for different patterns.

In the initial preprocessing step, we divide the patterns into two groups based on 1D periodicity. For Case 1 patterns, we use the algorithm we developed in Chapter 4. The following lemma summarizes its complexity.

Lemma 5.2.1. *2D dictionary matching for Case 1 patterns can be done in* $O(dm^2 + n^2)$ *time and* $O(dm \log m)$ *bits of space, aside from the* $sH_k(D) + O(s)$ *bits of space that store compressed self-index of the dictionary.*

For Case 2 patterns, candidates are identified by following the same framework we introduced in Chapter 4. However, we use a different method for verification. We make use of new developments in succinct 1D dictionary matching. We use the latest compressed AC automaton for two purposes, to represent the dictionary and to index the data at the same time. Thus, we eliminate the slowdown incurred by a compressed suffix tree representation of the pattern rows. This allows our algorithm to perform a *dynamic duel* between a pair of

candidates in constant time, resulting in an algorithm that runs in linear time.

In the remainder of this section, we assume that each pattern has at least one aperiodic row. The case of a pattern having a row that is periodic with period size between $m/4$ and $m/2$ will add only a small constant to the stated complexities.

5.2.1 Pattern Preprocessing

This is an outline of the steps in pattern preprocessing. Notice that we use the compressed AC automaton as the representation and index of the pattern rows, after which the original dictionary can be discarded.

1. Construct (compressed) AC automaton of first aperiodic row of each pattern. Store row number of each of these rows within the patterns.

2. Form a compressed AC automaton of the pattern rows.

3. Construct witness tree of pattern rows and preprocess for LCA.

4. Name pattern rows. Index the 1D patterns of names in a suffix tree.

In the first step, we form an AC automaton of one aperiodic row of each pattern, say, the first aperiodic row of each pattern. This will allow us to filter the text and limit the number of potential pattern occurrences to consider. Since we use only one row from each pattern, using a compressed version of the AC automaton is optional.

In the second step, the pattern rows are named as in BB to form a 1D dictionary of patterns. Here we use a compressed AC automaton of the pattern rows. An example of two patterns and their 1D representation is shown in Figure 5.1.

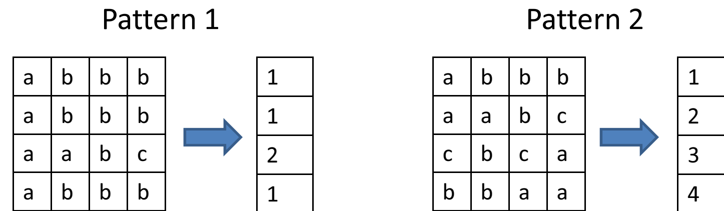
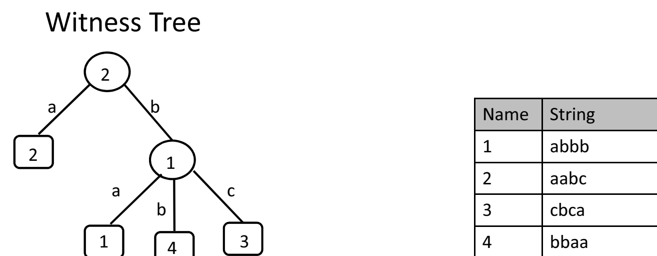


Figure 5.1: Two linearized 2D patterns with their 1D names.

Another necessary data structure is the witness tree, which we introduced in [53] and described in Chapter 4. A witness tree is used to store pairwise distinctions between different patterns, or pattern rows, of the same length. A witness tree provides a witness between pattern rows in constant time if it is preprocessed for Lowest Common Ancestor (LCA). Figure 5.2 depicts a witness tree of the row names used in Figure 5.1.



Name	String
1	abbb
2	aabc
3	cbca
4	bbaa

Figure 5.2: A witness tree for several strings of length 4.

Preprocessing proceeds by indexing the 1D patterns of names. We form a suffix tree of the 1D patterns to allow efficient computation of longest common prefix (LCP) queries between substrings of the 1D patterns.

Lemma 5.2.2. *The pattern preprocessing stage for Case 2 patterns completes in* $O(dm^2)$ *time and* $O(dm \log m)$ *extra bits of space.*

Proof. The AC automaton of the first non-periodic row of each pattern is constructed in $O(dm)$ time and is stored in $O(dm \log m)$ bits, in its uncompressed form. A compressed AC automaton of all pattern rows occupies $sH_k(D) + O(s)$ bits of space and can then become the sole representation of the dictionary [37]. The witness tree occupies $O(dm \log m)$ bits of space. A rooted tree can be preprocessed in linear time and space to answer LCA queries in $O(1)$ time [36, 10]. The patterns are converted to a 1D representation in $O(dm^2)$ time. A suffix tree of the 1D dictionary of names can be constructed and stored in linear time and space, e.g., [61].

□

5.2.2 Text Scanning

The text scanning stage has three steps.

1. Identify candidates in text block with 1D dictionary matching of a non-periodic row of each pattern.

2. Duel to eliminate inconsistent candidates within each column.

3. Verify pattern occurrences at surviving candidate positions.

Step 1. Identify Candidates

We identify a limited set of candidates in the text block using 1D dictionary matching on the first aperiodic row of each pattern. There can be only one occurrence of any non-periodic pattern row in a text block row. Each occurrence of an aperiodic pattern row demarcates a candidate, at most d per row. In total, there can be up to dm candidates in a text block, with candidates for several distinct 1D patterns on a single row of text. If the same aperiodic

row occurs in several patterns, several candidates can occur at the same text position, but candidates are still limited to d per row.

We run the Aho-Corasick algorithm over the text block, row by row, to find up to dm candidates. Then we update each candidate to reflect the position at which we expect a pattern to begin. This is done by subtracting the row number of the selected aperiodic row within the pattern from the row number at which it is found in the text block.

Complexity of Step 1: 1D dictionary matching on a text block takes $O(m^2)$ time with the AC method. Marking the positions at which patterns can begin is done in constant time per candidate found; overall, this requires $O(dm) = o(m^2)$ time. The AC algorithm uses extra space proportional to the dictionary, which is $O(dm \log m)$ bits of space for this limited set of pattern rows. The dm candidates can also be stored in $O(dm \log m)$ bits of space.

Step 2. Eliminate Vertically Inconsistent Candidates

Recall that we call a pair of candidates for pattern occurrences *consistent* if all positions of overlap match. Vertically consistent candidates are two candidates that appear in the same column, and have a suffix/prefix match in their 1D representations. For our purposes, we need only eliminate vertically inconsistent candidates before comparing text and pattern characters at which patterns are expected to occur. Thus, we perform duels between the candidates in a given column, pairwise. In order to verify candidates in a single pass over the text, we take advantage of the fact that overlapping segments of consistent candidates can be verified simultaneously.

We perform duels between one pair of candidates at a time and eliminate inconsistent candidates. Then we remain with a set of consistent candidates to verify in the text. Until

this thesis work, the dueling paradigm had not been applied to dictionary matching since it is prohibitive to precompute and store witnesses for all possible overlaps of all candidate patterns in a set of patterns. However, we developed an innovative way of performing duels for a set of 2D patterns. In *dynamic dueling*, no witness locations are computed in advance. We store a minimal amount of information that allows us to efficiently generate witnesses on the fly, as they are needed in a duel.

A duel consists of two steps. In the first step, an LCP query on the 1D representation of the patterns is used to generate a "row-witness," the first row at which the candidates differ. In the second step of the duel, we use the witness tree to locate the position of mismatch between the two different pattern rows, and we use that position to eliminate one or both candidates.

To demonstrate how a witness is found and the duel is performed, we return to the patterns in Figure 5.1. Assume two candidates exist; directly below a candidate for Pattern 1, we have a candidate for Pattern 2. The LCP of *121*, (second suffix of linearized Pattern 1) and *1234* (linearized Pattern 2) is 2. Since $2 < 3$, the LCP query reveals that the patterns are inconsistent, and that a witness exists between the fourth row of Pattern 1 (name 1) and the third row of Pattern 2 (name 3). We then procure a witness from the witness tree shown in Figure 5.2 by taking the LCA of the leaves that represent names 1 and 3. The result of this query shows that the first position is a point of distinction between names 1 and 3. If the text has an 'a' at that position, Pattern 1 survives the duel. Otherwise, if the character is a 'c', Pattern 2 survives the duel. If neither 'a' nor 'c' occur at the text location, both candidates are eliminated.

Lemma 5.2.3. *A duel between two candidate patterns A and B in a given column j of the text can be performed in constant time.*

Proof. The suffix tree constructed in Step 4 of the pattern preprocessing answers LCP queries in the 1D patterns of names in $O(1)$ time. The witness tree gives a position of a row-witness in $O(1)$ time, and retrieving the text and pattern characters to perform the actual duel takes constant time.

□

Complexity of Step 2: Step 2 begins with at most dm candidate positions. Each candidate is involved in exactly one duel, and is either killed or survives. If a candidate survives, it may be visited exactly one more time to be eliminated by a duel beneath it. Since a duel is performed in constant time, by Lemma 5.2.3, this step completes in $O(dm)$ time. Recall that $d < m$. Hence, the time for Step 2 is $O(m^2)$.

Step 3. Verify Surviving Candidates

After eliminating vertically inconsistent candidates, we verify pattern occurrences in a single scan of the text block. We process one text block row at a time to conserve space. Before scanning the current text block row, we label the positions at which we expect to find a pattern row. This is done by merging the labels from the previous row with the list of candidates that begin on the new row. If a new candidate is introduced in a column that already has a label, we keep only the label of the lower candidate. This is permissible since the label must be from a consistent candidate in the same column. Thus, each position in the text has at most one label.

The text block row is then scanned sequentially, to mark actual occurrences of pattern rows. This is done by running AC on the text row with the compressed AC automaton of all pattern rows. The lists of expected row names and actual row names are then compared sequentially. If every expected row name appears in the text block row, the candidate list

remains unchanged. If an expected row name does not appear, a candidate is eliminated. The pointers that connect candidates are used to eliminate candidates in the same column that also include the label that was not found.

After all rows are verified in this manner, all surviving candidates in the text are pattern occurrences of their respective patterns.

Complexity of Step 3: When a text block row is verified, we mark each position at which a pattern row (1D name) is expected to begin. This list is limited by $m/2$ due to the vertical consistency of the candidates. We also mark actual pattern row occurrences in the text block row which are again no more than $m/2$ due to distinctness of the row names. Thus, the space complexity for Step 3 is $O(m)$. The time complexity is also linear, since AC is run on the row in linear time, and then two sorted lists of pattern row names are merged. Over all $3m/2$ rows in the text block, the complexity of Step 3 is $O(m^2)$.

Lemma 5.2.4. *The algorithm for 2D dictionary matching, when pattern rows are not highly periodic and $d < m$, completes in $O(n^2)$ time and $O(dm \log m)$ bits of space, in addition to $sH_k(D) + O(s)$ bits of space to store the compressed AC automaton of the dictionary.*

Proof. This follows from Lemma 5.2.2 and the complexities of Steps 1, 2, and 3.

□

Theorem 5.2.5. *Our algorithm for 2D dictionary matching completes in $O(dm^2+n^2)$ time and $O(dm \log dm)$ bits of extra space.*

Proof. For $d > m$, this is stated in Corollary 1 of Theorem 5.1.1.
For $d \leq m$, the patterns are split into groups according to the periodicity of their rows.
A pattern is classified in linear time by finding the period of each of its rows, e.g., using a KMP automaton [44].

For Case 1 patterns, this is proven in Lemma 5.2.1.

For Case 2 patterns, this is proven in Lemma 5.2.4.

□

Chapter 6

Dynamic 2D Dictionary Matching in Small Space

This chapter develops the first efficient dynamic dictionary matching algorithm for two-dimensional data in the space-constrained environment. The algorithm is a succinct and dynamic version of the classic Bird / Baker algorithm. Since we follow their labeling paradigm, our algorithm is well-suited for a dictionary of rectangular patterns that are the same size in at least one dimension. Our algorithm uses a dynamic compressed suffix tree as a compressed self-index to represent the dictionary in entropy-compressed space. All tasks are completed by our algorithm in linear time, overlooking the slowdown in querying the compressed suffix tree.

The static succinct 2D dictionary matching algorithm with no slowdown presented in Chapter 5 is not suitable for the dynamic setting. It relies on the succinct 1D dictionary matching algorithm of Hon et al. [37], which does not readily admit changes to the dictionary. Instead, we adapt the succinct 2D dictionary matching algorithm of Chapter 4 to the

dynamic setting. We develop a dynamic algorithm that meets the time and space complexities that were achieved in the static version of the algorithm. The dictionary is initially processed in time proportional to the size of the dictionary. Subsequently, a pattern is inserted or removed in time proportional to the single pattern's size. We modify the witness tree (Section 4.2.1) to form a dynamic data structure that meets the space and time complexities achieved by the static version. The dynamic witness tree accommodates insertion or removal of any string in time proportional to the string's length.

We define *Two-Dimensional Dynamic Dictionary Matching* (2D-DDM) as follows. Our algorithm is given a dictionary of d patterns, $D = \{P_1, \ldots, P_d\}$, of total size ℓ. Each pattern P_i is of size $m_i \times \overline{m}$, $1 \leq i \leq d$. We use τ to denote the time it takes to access a character or perform other queries in the compressed self-index of the dictionary. The dictionary is initially preprocessed in $O(\ell\tau)$ time. A pattern P, of size $p \times \overline{m}$, is inserted to or removed from the dictionary in $O(p\overline{m}\tau)$ time. Our algorithm searches a text T of size $n_1 \times n_2$ in $O(n_1 n_2 \tau)$ time for all occurrences of P_i, $1 \leq i \leq d$. Using recent results, τ is at most $\log^2 \ell$. Our algorithm uses $O(d\overline{m} \log d\overline{m} + dm' \log dm')$ bits of extra space, where $m' = max\{m_1, \ldots, m_d\}$.

The succinct 2D dictionary matching algorithm of Chapter 4 was presented in terms of a static dictionary in which all patterns are the same size in both dimensions, resulting in a dictionary of size dm^2. In this chapter, we work with a dictionary of patterns that are of uniform width, but of varying heights. We perform a more detailed analysis and distinguish between the sources of time complexities. Specifically, we analyze which time complexities are proportional to the uniform width of the patterns[1], $\overline{m}$, which are proportional to the height of the largest pattern, m', and which are proportional to the actual dictionary size, ℓ.

[1]We chose this notation since it is visual. The bar represents a uniform width, while the prime is vertical, representing a uniform height.

While doing this, we discovered the need for more efficient techniques in the verification process in order for the text scanning to remain linear in the size of the text, in the case that m' is an order of magnitude larger than $\overline{m}$. Herein lies one of the contributions of this chapter.

We begin by presenting a linear-time dynamic 2D dictionary matching algorithm that uses extra space proportional to the size of the input. In Section 6.2, we describe a succinct variation of this linear space algorithm for a dictionary with a large number of patterns. For all other dictionaries, patterns are divided into two groups; patterns in each group are searched for separately and in different ways. We describe our approach to dynamic dictionary matching for each group of patterns in Section 6.3.

6.1 2D-DDM in Linear Space

In this section we present a linear-time dynamic 2D dictionary matching algorithm that uses extra space proportional to the size of the input. It is a dynamic succinct variant of the Bird / Baker algorithm. In the multiple pattern matching version of the Bird / Baker algorithm, 1D dictionary matching is used in two different ways. First, the pattern rows are seen as a 1D dictionary and this set of "patterns" is used to linearize the dictionary and then to label text positions. A separate 1D dictionary is formed of the linearized 2D patterns. The Bird / Baker algorithm is suitable for 2D patterns that are of uniform size in at least one dimension, so that the text can be marked with at most one name at each text location. The Bird / Baker method uses linear time and space in both the pattern preprocessing and the text scanning stages.

Sahinalp and Vishkin's dynamic 1D dictionary matching algorithm (SV) uses a naming

technique rather than a dictionary-matching automaton [60]. Yet, it is a suitable replacement for the Aho-Corasick automata in the Bird / Baker algorithm. Thus, the combination of these techniques, one for dynamic dictionary matching in 1D and another for static 2D dictionary matching, yields a dynamic 2D dictionary matching algorithm that runs in linear time. This modification extends the Bird / Baker algorithm to a accommodate a changing dictionary, yet it does not introduce any slowdown. This algorithm is outlined in Algorithm 4.

Algorithm 4 Dynamic Version of Bird / Baker Algorithm

{1} Preprocess Pattern:
  a) Name pattern rows using SV [60].
  b) Store 1D pattern of names for each pattern in D, called D'.
  c) Preprocess D' using SV to later perform 1D dynamic dictionary matching.
{2} Row Matching:
  Use SV on each row of text to find occurrences of D's pattern rows.
  This labels positions at which a pattern row ends.
{3} Column Matching:
  Run SV on named columns of text to find occurrences of patterns from D' in the text.
  Output pattern occurrences.

Initially, the dictionary of pattern rows is empty. One 2D pattern is linearized at a time, row by row. As a pattern row is examined, it can be viewed as a text on which to perform dictionary matching. If a pattern row is identified in the new pattern row, then it is given the same name as the matching row. Otherwise, this new row is seen as a new 1D pattern and added to the dictionary of pattern rows. Once the pattern rows have been given names, the 1D patterns of names in D' are preprocessed separately.

Whenever a pattern is added to or removed from the 2D dictionary, the precomputed information about the patterns can be adjusted in time proportional to the size of the 2D pattern that is entering or leaving the dictionary. That is, Sahinalp and Vishkin's framework for dictionary matching allows both 1D dictionaries to efficiently react to a change in the

2D dictionary that they represent.

Space complexity of Algorithm 4: The dynamic version we present of the Bird / Baker algorithm uses extra space proportional to the size of the input. It uses $O(\ell \log \ell)$ bits of extra space to name the pattern rows using SV [60] and $O(dm' \log dm')$ bits of extra space to store and index the 1D representation of the patterns. During text scanning, $O(n_2 \log n_2)$ bits of space are used to run SV on each row of text and $O(n_1 \log n_1)$ bits of space are used to run SV on the named columns of text, one at a time. $O(n_1 n_2 \log dm')$ bits of extra space are used to store the names given to text positions.

6.2 2D-DDM in Small-Space For Large Number of Patterns

The dynamic version of the Bird / Baker algorithm presented in Section 6.1 uses space proportional to the sizes of both the dictionary and the text. In this section we present a variation of Algorithm 4 that runs in small space for a dictionary in which $d \geq \overline{m}$. That is, when the number of patterns is larger than the width of a pattern.

We begin by modifying Algorithm 4 to work with small blocks of text and thereby relate the extra space to the size of the dictionary, not the size of the text. We use a known technique for minimizing space and process the text in small overlapping blocks of size $3m'/2 \times 3\overline{m}/2$, where $m' = max\{m_1, \ldots, m_d\}$. Since each text block is processed in time proportional to the size of the text block, the overall text scanning time remains linear.

By processing one text block at a time, we reduce the working space to $O(\ell \log \ell + dm' \log dm')$ bits of extra space to preprocess the patterns and $O(\overline{m} \log \overline{m} + \overline{m}m' \log dm')$ bits of extra space to search the text. This change does not affect the time complexity. We

seek to further reduce the working space by employing a smaller space mechanism to name the pattern rows and subsequently name the text positions.

Recent innovations in succinct full-text indexing provide us with the ability to compress a suffix tree, using no more space than the entropy of the original data it is built upon. These self-indexes can replace the original text, as they support retrieval of the original text, in addition to answering queries about the data, very quickly.

Several dynamic compressed suffix tree representations have been developed, each offering a different time/space trade-off. Chan et al. presented a dynamic suffix tree that occupies $O(\ell)$ bits of space [14]. Queries, such as edge label retrieval and insertion or removal of a substring, have an $O(\log^2 \ell)$ slowdown. Russo et al. developed a dynamic fully-compressed suffix tree requiring $\ell H_k(\ell) + o(\ell \log \sigma)$ bits of space, which is asymptotically optimal under kth order empirical entropy [56]. This compressed suffix tree representation uses a dynamic compressed suffix array and stores a sample of the suffix tree nodes. Although some operations can be executed more quickly, all operations have $O(\log^2 \ell)$ time complexity. This dynamic compressed suffix tree supports a larger set of suffix tree navigation operations than the compressed suffix tree proposed by Chan et al. [14]. It also reaches a better space complexity and can perform basic operations more quickly. We hereafter suppose that a dynamic compressed suffix tree is used to replace the dictionary of patterns and we refer to the slowdown of operations in the entropy-compressed self-index as τ.

For a succinct version of Algorithm 4, we use a dynamic compressed suffix tree to represent and index the pattern rows in entropy-compressed space. Traversing the dynamic compressed suffix tree introduces a slight sacrifice in runtime. We modify Algorithm 4 to use the compressed suffix tree. The changes are limited to steps 1a and 2.

During pattern preprocessing, the dynamic compressed suffix tree can be built incrementally, as one pattern row is named at a time. First, traversal of the suffix tree can be attempted by traversing a path from the root labeled by the characters in the pattern row. If a matching row is found, the new row is given the same name as the row that it matches. Otherwise, the new pattern row is inserted into the compressed suffix tree and given a new name.

The positions of a text block row are also named by traversing the suffix tree. Here the suffix tree is not modified by the text. Thus, an entire text block is named in linear time, with a τ slowdown. We use a technique similar to the one described by Gusfield in the computation of *matching statistics*, in [35] Section 7.8. Positions in a text block are named, row by row, according to the names of pattern rows. To name a new text block row, traversal begins at the root of the tree, with the edge whose label matches the first position of the text block row. When $\overline{m}$ consecutive characters trace a path from the root, traversal reaches a leaf, and the position is named with the matching pattern row. At a mismatch, suffix links quickly find the longest suffix of the already matched string that matches a prefix of some pattern row and the next text character is compared to that labeled edge of the tree.

All pattern rows have width $\overline{m}$. This ensures that each text position can be uniquely labeled. One pattern row cannot be a substring of another. Thus, we do not share the concern of Amir and Farach's suffix tree based approach to dictionary matching [3]. They use lowest marked ancestor queries to address the issue of possibly missing pattern occurrences when one pattern is a substring of another, and an occurrence may be skipped when a suffix link is traversed.

Theorem 6.2.1. *If $d \geq \overline{m}$, we can solve the dynamic 2D dictionary matching problem*

in almost linear $O((\ell + n_1 n_2)\tau)$ *time and* $O(\overline{m}\log\overline{m} + dm'\log dm')$ *bits of extra space, aside from the space used to represent the dictionary in a compressed self-index. Pattern* P *of size* $p \times \overline{m}$ *can be inserted to or removed from the dictionary in* $O(p\overline{m}\tau)$ *time and the updated index will occupy an additional* $O(p\log dm')$ *bits of space, where* m' *is updated to reflect the new maximum pattern height.*

Proof. With small blocks of text, and $d \geq \overline{m}$, Algorithm 4 uses $O(\ell\log\ell + dm'\log dm')$ bits of space for preprocessing when $O(\ell\log\ell)$ bits are used to prepare the pattern rows for dynamic dictionary matching. Replacing [60] with the compressed suffix tree traversal, the algorithm uses entropy-compressed space to represent and index the dictionary and an extra $O(dm'\log dm')$ bits of space to name the pattern rows. Since $d \geq \overline{m}$, $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits of extra space are used to label text positions and perform 1D dynamic dictionary matching in the columns of text. All operations (preprocess dictionary, update dictionary, search text) run in linear time, with an $O(\tau)$ slowdown to query the dynamic compressed suffix tree.

□

6.3 2D-DDM in Small-Space for Small Number of Patterns

This section deals with the case in which the number of patterns is smaller than the common dimension among all dictionary patterns, i.e., $d = o(\overline{m})$. For this case, we do not allow the space to label each text block location and therefore the dynamic version of the Bird and Baker algorithm cannot be applied trivially. We use several combinatorial tricks to preserve the spirit of Bird and Baker's algorithm without incurring the necessary storage overhead.

We use dynamic data structures that allow the dictionary to be updated efficiently. The dictionary is indexed by a dynamic compressed suffix tree, after which the patterns can be discarded. This can be done in space that meets kth order empirical entropy bounds of the input, as described in Section 6.2. Thus, the compressed self-index does not occupy *extra space*. Throughout this chapter, the extra space used by our algorithm is limited to $O(\overline{m}\log\overline{m}+dm'\log dm')$ bits of space. The running time of our algorithm is almost linear, with a slowdown to accommodate queries to the compressed suffix tree, referred to as τ.

We divide the dictionary patterns into two groups and search the text for patterns in each group separately. In the following sections, we describe first an algorithm for patterns in which the rows are highly periodic and then an algorithm for all other patterns. We begin by describing a dynamic data structure that is used by both parts of the algorithm.

6.3.1 Dynamic Witness Tree

In this section we show how to form a dynamic variant of the witness tree, whose static version was developed in Section 4.2.1. Recall that a witness tree can be constructed to name a set S of j strings, each of length m, in linear $O(jm)$ time and in $O(j)$ space so that identical strings receive the same name. An internal node in the witness tree denotes a position of mismatch, which is an integer $\in [1, m]$. Each edge of the tree is labeled with a single character. Sibling edges must have different labels. A leaf represents a name given to string(s) in S.

Query: For any two strings $s, s' \in S$, return a position of mismatch between s and s' if $s \neq s'$, otherwise return $m + 1$.

Preprocessing the witness tree for Lowest Common Ancestor (LCA) queries on its leaves allows us to answer the above witness query between any two named strings in

S in constant time. This preprocessing can be performed in linear time and space, with respect to the size of the dynamic tree [17].

Construction of the witness tree begins by choosing any two strings in S and comparing them sequentially. When a mismatch is found, comparison halts and an internal node is added to the witness tree to represent this witness of mismatch, with two children to represent the names of the two strings. If no mismatch is found, the two strings are given the same name. Each successive string is compared to the witnesses stored in the tree by traversing a path from the root to identify to which name, if any, the string belongs. Characters of a new string are examined in the order dictated by traversal of the witness tree, possibly out of sequence. If traversal halts at an internal node, the string receives a new name, and a new leaf is added as a child to the internal node. Otherwise, traversal halts at a leaf, and the new string is compared sequentially to the string represented by the leaf, as done with the first two strings.

Now we consider the scenario in which S is a dynamically changing set of strings.

Lemma 6.3.1. *A new string is added to the witness tree in $O(m)$ time.*

Proof. Including a new string in S and naming it with the witness tree follows the same procedure that the static witness tree uses to build the witness tree as each pattern is considered individually. By Lemma 4.2.4, this is done in $O(m)$ time and adds one or zero nodes to the witness tree.

□

Lemma 6.3.2. *A string is removed from the witness tree in $O(1)$ time.*

Proof. In removing a string s from S, there are two possibilities to consider. If s is the only string with its name, remove its leaf. In the event that the parent is an internal node

with only one other child, remove the hanging internal node as well. Then, the sibling of the deleted leaf becomes a child of its grandparent. The other possibility is that some other string(s) in S bear the same name as s. We do not want to remove a leaf while there is still a string in S that has its name. Thus, we augment each leaf with an integer field to store the number of strings in S that have its name. This counter is increased when a new string is named with an existing name. This counter is decreased when a row is deleted. When the counter is down to 0, the leaf is discarded, possibly along with its parent node, as described earlier.

□

Observation 5. *The dynamic witness tree of j strings, each of length m, occupies $O(j \log j)$ bits of space.*

6.3.2 Group I Patterns

As in Chapter 4, we consider two types of patterns, and each one presents its own difficulty. In the initial preprocessing step, we divide the patterns into two groups based on the 1D periodicity of their rows. In Group I, all pattern rows are periodic, with periods $\leq \overline{m}/4$. The difficulty in this case is that many overlapping occurrences can appear in the text in close proximity to each other, and we can easily have more candidates than the working space we allow. Patterns in Group II have at least one aperiodic row or one row whose period is larger than $\overline{m}/4$. Here, each pattern can occur only $O(1)$ times in a text block. Since several patterns can overlap each other in both directions, a difficulty arises in the text scanning stage. We do not allow the time to verify different candidates separately, nor do we allow space to keep track of the possible overlaps between different patterns.

Preprocessing Dictionary

We follow the succinct preprocessing scheme that we introduced in Chapter 4. We maintain the assumption of Chapter 4 that each dictionary pattern whose rows are highly periodic has an LCM that is $O(\overline{m})$. We use the dynamic witness tree and dynamic offset tree instead of their static counterparts. The following preprocessing steps are initially performed for each dictionary pattern in Group I and are later used upon arrival of a new pattern.

1. For each pattern row,
   (a) Compute period and canonize.
   (b) Lyndon word naming with dynamic witness tree, resulting in a 1D dictionary, D'.
   (c) Insert to dynamic compressed suffix tree.
2. Preprocess 1D dictionary:
   (a) Preprocess D' for dynamic dictionary matching.
   (b) Build LCM table for each 1D pattern.
   (c) Compute 2D Lyndon word of each 1D pattern and store shift.
   Add to compressed trie, if multiple patterns have the same 1D pattern of names.

Lemma 6.3.3. *Patterns in Group I are preprocessed in $O(\ell\tau)$ time and $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits of extra space.*

Proof. Step 1 processes a single pattern row in $O(\overline{m}\tau)$ time and $O(\overline{m}\log\overline{m})$ bits of extra space. Thus, the entire set of pattern rows are processed in $O(\ell)$ time to gather information and $O(\ell\tau)$ time to index the pattern rows in a dynamic compressed suffix tree. Since $O(1)$

information is stored per row, $O(dm' \log dm')$ bits of extra space are used to store information gathered about the pattern rows in the dictionary.

Step 2 preprocesses the 1D patterns in the dictionary of names. Using Sahinalp and Vishkin's algorithm, $O(dm')$ time and $O(dm' \log dm')$ bits of extra space are used to facilitate linear time dynamic dictionary matching in a 1D dictionary of size $O(dm')$ [60]. The LCM tables of the 1D patterns are computed in linear time and occupy $O(dm' \log m')$ bits of extra space. The 2D Lyndon word of each pattern can be computed in time proportional to its size. The 2D Lyndon words of the dictionary occupy $O(dm' \log dm')$ bits of extra space. Overall, Step 2 runs in $O(dm')$ time and $O(dm' \log dm')$ bits of extra space.

□

Corollary 6.3.4. *A new pattern of size $p \times \overline{m}$ is added to Group I in $O(p\overline{m}\tau)$ time and $O(\overline{m} \log \overline{m} + p \log dm')$ bits of extra space.*

Lemma 6.3.5. *A pattern in Group I of size $p \times \overline{m}$ is removed from the dictionary in $O(p\overline{m}\tau)$ time and eliminates $O(\overline{m} \log \overline{m} + p \log dm')$ bits of extra space the algorithm allocated for it.*

Proof. The following steps meet the indicated time and space bounds and remove a pattern from Group I. Each pattern row is removed from the dynamic witness tree, in $O(1)$ time (by Lemma 6.3.2), and from the dynamic compressed suffix tree, in $O(\overline{m}\tau)$ time. This takes $O(p\overline{m}\tau)$ time in total. If this is the only pattern with its 1D representation, its LCM table is deleted and the 1D pattern is removed from the dictionary of names that has been preprocessed for dynamic dictionary matching. If this is one of several patterns with the same 1D representation, and the sole member of its consistency class, the 2D Lyndon word is removed from the compressed trie.

□

Text Scanning

The text is searched for occurrences of patterns in Group I in a three step process. First, the text block rows are named by the Lyndon words of their periods using the dynamic witness tree of the dictionary (Section 6.3.1). We store the Lyndon word name, period size, *LYpos*, *right*, and *left* of each pattern row. Then, the linearized text, T', is searched for candidate positions that match a pattern in the 1D dictionary using 1D dynamic dictionary matching, since the patterns can be of varying heights. Finally, the verification step finds the actual pattern occurrences among the candidates. This requires consideration of the alignment of periods among rows and of the overall width of the 1D names. Since the first two steps have been described, the remainder of this section discusses the verification stage.

If $m' = O(\overline{m})$, we can use a verification procedure almost identical to the procedure that appears in Chapter 4. However, if the uniform width, $\overline{m}$, is asymptotically smaller than the height of the tallest pattern, m', then this algorithm does not yield a linear time text scanning. This is due to the fact that the algorithm costs $O(m')$ time to process each candidate row, resulting in $O(m' * m')$ time if $\overline{m} = o(m')$. For this situation, new ideas are needed and we introduce a new verification process that verifies a single pattern in $O(m')$ time. Since the dictionary has d patterns, and $d < \overline{m}$, the entire text block is verified in $O(\overline{m}m')$ time.

We verify candidates for each pattern, P_i, separately. Verification of each candidate consists of two tasks:

1. **Verify shifts:** The verification of shifts is summarized in Algorithm 5 and described in the following paragraph.

   Let P_i' be the 1D pattern of names for P_i. If P_i' is not periodic, there is no need to

Algorithm 5 Verify Shifts for Pattern P_i

Input: P_i': 1D pattern of names for P_i,
$LW[1...m]$: 2D Lyndon word representing P_i
compressed trie of $LW[1...m]$ subsequence at each period
KMP automaton of P_i': longest border horizontally consistent with P_i''s prefixes

if P_i' is not periodic **then**
  for all candidate rows r for pattern P_i **do**
    compute $T_LW[1...m]$ = 2D Lyndon word for $T[r...r+m_i]$
    compare $LW[1...m]$ and $T_LW[1...m]$
  end for
  eliminate candidates that mismatch P_i
else
  $\{P_i'$ is periodic$\}$
  for all p_blocks j in P_i' **do**
    $\{p_block\ j$ is a period in $P_i'\}$
    compute T_LW_j: 2D Lyndon word to represent rows of p_block_j in T
    bucket sort T_LW_j with other shifts for same text rows
    match T_LW_j against compressed trie
  end for
  KMP on shifts among p_blocks
  eliminate candidates that mismatch P_i
end if

worry about overlapping candidates, and we verify each candidate row for P_i separately. Verification of the *LYpos* shifts for a candidate of P_i consists of matching P_i's 2D Lyndon word with the 2D Lyndon word of the corresponding rows of the text. If P_i' is periodic, the idea is similar. We call each period in P_i' a p_block. We compute the shifts of each p_block separately in the text. In order to remain linear, we use bucket sort on the shifts of the rows as the 2D Lyndon word for each period is computed. Since the different p_blocks can have different 2D Lyndon words, we construct a compressed trie of 2D Lyndon words and match each text representative against the compressed trie. Once the shifts within each p_block are verified, it remains to verify the shifts among the p_blocks. Since each p_block has the same horizontal period (i.e., the LCM of the periods of the rows of a p_block), it is possible to use a Knuth-Morris-Pratt automaton [44] on the shifts to complete the verification. For this, the preprocessing of P_i' stores for each prefix of P_i', the longest border of the prefix that is horizontally consistent with itself. To compare p_blocks when constructing the KMP automaton, it suffices to check whether the p_blocks point to the same block in the compressed trie of 2D Lyndon words.

2. **Check width:** Use range minimum and maximum queries to calculate *minRight* and *maxLeft* for each candidate of P_i. Then, reverse the shift and make sure that there is room for the pattern between *minRight* and *maxLeft*, i.e., that the candidate spans at least $\overline{m}$ columns.

Time and Space Complexity of Text Scanning: The linear representation of the text is computed in $O(\overline{m}m')$ time and occupies $O(m' \log dm')$ bits of space, as shown in Section 4.2.2. Candidates can be identified using Sahinalp and Vishkin's algorithm [60] in time linear in the 1D representations. Verification as done in Chapter 4 is linear. It remains to

show that the new verification, when $\overline{m} = o(m')$, is linear time. Since we have d patterns and $d < \overline{m}$, a linear time search in the 1D text of size $O(m')$ for each pattern, will yield overall $O(\overline{m}m')$ time, linear in the text block. The challenge that arose in Step 1 is the $O(\overline{m})$ time complexity for computing each 2D Lyndon word for the text. For each pattern, P_i, we have to compute the 2D Lyndon word for every candidate row. However, we can show that the $O(\overline{m})$ work is over all rows of the text when verifying P_i (and not for each candidate of P_i). Thus, the total complexity for verifying P_i is $O(\overline{m}+m')$. Linear time and space preprocessing schemes allow us to answer range minimum and maximum queries in $O(1)$ time [28]. Check-width (Step 2) consists of constant-time RMQ per candidate, which totals $O(m')$ time overall for P_i, and for all P_i, $1 \leq i \leq d$, text scanning completes in $O(\overline{m}m')$ time.

Lemma 6.3.6. *A text of size $n_1 \times n_2$ is searched for patterns in Group I in $O(n_1 n_2 \tau)$ time and $O(\overline{m} \log \overline{m} + m' \log dm')$ bits of extra space.*

Proof. Each block of text is searched in $O(\overline{m}m'\tau)$ time and $O(\overline{m} \log \overline{m} + m' \log dm')$ bits of extra space. Thus, the entire text is searched for patterns in Group I in $O(n_1 n_2 \tau)$ time and $O(\overline{m} \log \overline{m} + m' \log dm')$ bits of extra space.

□

6.3.3 Group II Patterns

Patterns in Group II have at least one aperiodic row or one row whose period is larger than $\overline{m}/4$. We assume that each pattern in this group has at least one aperiodic row. The case of a pattern having a row that is periodic with period size between $\overline{m}/4$ and $\overline{m}/2$ is handled similarly, since each pattern can occur only $O(1)$ times per text block row.

For patterns in Group II, many different pattern rows can overlap in a text block row. As a result, it is difficult to employ a succinct naming scheme to linearize the text block and find all occurrences of patterns in the text. Instead, we use the aperiodic row of each pattern to filter the text block and identify a limited set of candidates for pattern occurrences. We use dynamic dueling to eliminate inconsistent candidates within each text column. Then, a single pass over the text suffices to verify all remaining candidates for pattern occurrences.

Preprocessing Patterns

The following preprocessing steps are initially performed for each dictionary pattern in Group II and are later used upon arrival of a new pattern.

1. Locate first aperiodic row in each pattern and preprocess for dynamic dictionary matching.

2. Name pattern rows using a single witness tree and store 1D patterns of names.

3. Insert pattern rows to dynamic compressed suffix tree.

4. Construct dynamic suffix tree of 1D patterns.

5. Preprocess witness tree and suffix tree for dynamic LCA.

Lemma 6.3.7. *Patterns in Group II are preprocessed in* $O(\ell\tau)$ *time and* $O(d\overline{m}\log d\overline{m} + dm'\log dm')$ *bits of extra space.*

Proof. 1. The period of a pattern row is computed in $O(\overline{m})$ time and $O(\overline{m}\log\overline{m})$ bits of extra space [46]. At most, all pattern rows are examined, in $O(\ell)$ time and $O(\overline{m}\log\overline{m})$ bits of extra space. Sahinalp and Vishkin's preprocesses these rows in $O(d\overline{m})$ time and stores information in $O(d\overline{m}\log d\overline{m})$ bits [60].

2. Pattern rows are named by the witness tree in $O(\ell)$ time. By Observation 5, the dynamic witness tree of pattern rows occupies $O(dm' \log dm')$ bits of space. A single witness tree suffices since all pattern rows are the same size.

3. The set of pattern rows is indexed by the dynamic compressed suffix tree in $O(\ell\tau)$ time.

4. The 1D dictionary of names is stored in $O(dm' \log dm')$ bits of space and its dynamic generalized suffix tree is constructed in $O(dm')$ time and occupies $O(dm' \log dm')$ bits of space [16].

5. The dynamic suffix and witness trees are preprocessed in linear time to answer LCA queries in $O(1)$ time [17].

□

Corollary 6.3.8. *The dictionary is updated to add a new pattern of size $p \times \overline{m}$ to Group II in $O(p\overline{m}\tau)$ time and $O(p\overline{m} \log d\overline{m} + p \log dm')$ bits of extra space.*

Lemma 6.3.9. *A pattern in Group II of size $p \times \overline{m}$ is removed from the dictionary in $O(p\overline{m}\tau)$ time and eliminates $O(\overline{m} \log \overline{m} + p \log dm')$ bits of extra space the algorithm allocated for it.*

Proof. The following steps are performed to remove a pattern from Group II:

The first aperiodic row of the pattern is removed from the 1D dictionary that has been preprocessed for dynamic dictionary matching in $O(\overline{m})$ time and deallocates $O(\overline{m} \log \overline{m})$ bits of space [60].

The 1D representation of the pattern is deleted and it is removed from the suffix tree of 1D patterns in $O(p)$ time and deallocates $O(p \log dm')$ bits of space [16].

Each row of the pattern is removed from the compressed suffix tree in $O(p\overline{m}\tau)$ time.

□

Text Scanning

The text is searched for patterns in Group II in almost the same way as in the static algorithm of Chapter 4. The only difference between the text scanning stage of the static algorithm and that of the dynamic algorithm lies in the method used to identify 1D pattern occurrences in the linearized text. The Aho-Corasick automaton is not suitable for a dynamic dictionary since it is not updated efficiently. Rather, we use Sahinalp and Vishkin's method for dynamic dictionary matching since it completes all preprocessing and searching tasks, including updating the dictionary, in linear time and space. We summarize the text scanning and the complexity analysis in the following.

Summary of Text Scanning

1. **Identify candidates:** Sahinalp and Vishkin's 1D dynamic dictionary matching algorithm finds occurrences of the first aperiodic row of the patterns. It searches the text block, one row at a time, in $O(\overline{m}m')$ time and $O(\overline{m}\log\overline{m})$ bits of extra space. $O(dm')$ candidates are stored in $O(dm'\log dm')$ bits of extra space.

2. **Duel vertically:**

   (a) An LCP query between suffixes of the 1D patterns finds the number of rows that match in overlapping candidates. An LCA query in the generalized suffix tree of 1D patterns is performed in $O(1)$ time and space to find a row of mismatch.

   (b) We use the witness tree to compare row names in $O(1)$ time. An LCA query in the witness tree of the pattern rows is performed in $O(1)$ time and space. Then a character in each pattern row is retrieved in $O(\tau)$ time.

   Time: each duel takes $O(\tau)$ time. Due to transitivity, the number of duels is limited

by the number of candidates. There are $O(dm')$ candidate positions, with $d < \overline{m}$ so this step completes in $O(\overline{m}m'\tau)$ time.

3. **Verify candidates:** Duels eliminate horizontally inconsistent candidates. A duel consists of an LCP query on the dynamic compressed suffix tree in $O(\tau)$ time. By transitivity, the number of candidates limits the number of duels. With $O(dm')$ candidates, and $d < \overline{m}$, dueling is completed in $O(\overline{m}m'\tau)$ time. We verify one text block row at a time and mark positions at which a pattern row (1D name) is expected to begin. The surviving labels are carried to the next row. This uses space proportional to the labels for one text row plus the number of candidates, $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits. Each text character within an anticipated pattern occurrence is only compared to one pattern character, in $O(\tau)$ time, which is $O(\overline{m}m'\tau)$ time overall.

Lemma 6.3.10. *A text of size $n_1 \times n_2$ is searched for patterns in Group II in $O(n_1n_2\tau)$ time and $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits of extra space.*

Proof. Each block of text is searched in $O(\overline{m}m'\tau)$ time and $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits of extra space. Thus, the entire text is searched for patterns in Group II in $O(n_1n_2\tau)$ time and $O(\overline{m}\log\overline{m} + dm'\log dm')$ bits of extra space.

□

Theorem 6.3.11. *Our algorithm for dynamic 2D dictionary matching when $d < \overline{m}$ completes in $O((\ell + n_1n_2)\tau)$ time and $O(d\overline{m}\log d\overline{m} + dm'\log dm')$ bits of extra space. Pattern P of size $p \times \overline{m}$ can be inserted to or removed from the dictionary in $O(p\overline{m}\tau)$ time and the index will occupy an additional $O(p\log dm')$ bits of space, where m' is updated to reflect the new maximum pattern height.*

Proof. We separate the patterns into two groups and search for patterns in each group separately. Classifying a pattern entails finding the period of each pattern row. This is done in $O(\overline{m})$ time and $O(\overline{m} \log \overline{m})$ bits of extra space per row [46]. Overall, the dictionary is separated into two groups in $O(\ell)$ time and $O(\overline{m} \log \overline{m})$ bits of extra space.

For patterns in Group I, this complexity is demonstrated by Lemmas 6.3.3, 6.3.5, 6.3.6 and Corollary 6.3.4.

For patterns in Group II, this complexity is demonstrated by Lemmas 6.3.7, 6.3.9, 6.3.10 and Corollary 6.3.8.

□

Chapter 7

Implementation

The *succinct 1D dictionary matching problem* was recently closed with the development of an algorithm that meets optimal time and space bounds. There is a lag in the implementation of the theoretical contributions to solve this problem. This is likely due to their complexity and the novelty of the data structures which they rely upon. Thus, as part of this thesis we have developed a succinct dictionary matching program that is more intuitive and relies on more commonly used data structures. Our algorithms for succinct 2D dictionary matching reduce the two-dimensional problem to its one-dimensional counterpart in different ways. Hence, we see the development of software for succinct 1D dictionary matching as a first step towards developing a program that solves 2D dictionary matching in small space.

We took the following steps to create our succinct 1D dictionary matching program.

1. Coded Ukkonen's suffix tree construction algorithm.

2. Modified suffix tree to form generalized suffix tree.

3. Wrote program to perform dictionary matching over generalized suffix tree.

4. Merged dictionary matching software with compressed suffix tree.

7.1 Software Development

We created a time and space efficient program for dictionary matching on one-dimensional data. We chose to use the suffix tree as the data structure for this implementation, since there are compressed suffix tree representations that reach empirical entropy bounds of the input string. Furthermore, these data structures have been implemented and their code is readily available. These implementations have been proven to be very space efficient in practice.

We began by distilling the precise implementation details necessary to convert a regular suffix tree to a generalized suffix tree for dictionary matching. Ukkonen's suffix tree construction algorithm [61] extends quite naturally to the construction of a generalized suffix tree for several strings [35], which can be used in a straightforward manner for dictionary matching. We coded Ukkonen's suffix tree construction algorithm and modified it to index a set of strings. Since it is an online algorithm, it can insert one string at a time to the index.

We give a brief outline of Ukknonen's algorithm in the following paragraphs, and our specifications of the algorithm's flow are depicted in pseudocode in Algorithm 6.

Ukkonen's algorithm is linear time and online. The elegance of Ukkonen's algorithm is evident in its key property. The algorithm admits the arrival of the string during construction. Yet, each suffix is inserted exactly once, and never updated after its insertion. An extra variable is incremented as characters arrive, eliminating the need to update each of the leaves representing suffixes already indexed by the tree. The end index of each leaf is demarcated by this special variable. Thus, a leaf is never updated after its creation.

As a new character is appended to the string, Ukkonens's algorithm makes sure that all

Algorithm 6 Ukkonen's suffix tree construction algorithm

```
j = -1;
{j is last suffix inserted}
for i = 0 to n - 1 do
   {phase i: i is current end of string}
   while j < i do
      {let j catch up to i}
      if singleExtensionAlgorithm(i, j) then
         break; {implicit suffix so proceed to next phase}
      end if
      if lastNodeInserted ≠ root then
         lastNodeInserted.SuffixLink ← root;
      end if
      lastNodeInserted ← root;
   end while
end for
```

suffixes of the input string are indexed by the tree. As soon as a suffix is implicitly found in the tree, modification of the tree ceases until the next new character is examined. The next phase begins by extending the implicit suffix with the new character. A suffix link is a pointer from the internal node at the end of the path labeled xS to the internal node at the end of the path labeled S, where x is an arbitrary character and S is a possibly empty substring. A suffix tree with several suffix links is shown in Figure 7.1. Using suffix links and a pointer to the last suffix inserted, a suffix is added to the tree in amortized constant time. The combination of one-time insertion of each suffix and rapid suffix insertion results in linear-time suffix tree construction.

The *generalized suffix tree* is a suffix tree for a set of strings. A suffix tree is often used to index several strings by concatenating the strings with unique delimiters between them. With that approach, a significant amount of space is wasted by indexing artificial suffixes that span several strings. Ukkonen's algorithm lends itself to a more space efficient construction of the generalized suffix tree in which only actual suffixes are stored. $O(1)$

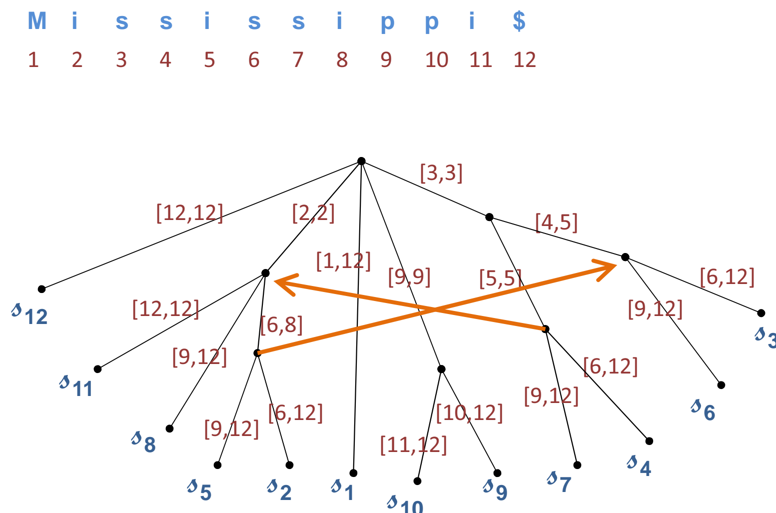


Figure 7.1: Suffix tree for the string Mississippi with several suffix links.

extra information is stored at each node, representing the string number of the node. The generalized suffix tree then consists of the suffix trees of the individual patterns merged together. It is built incrementally, in an *online* fashion, inserting one string at a time.

Dictionary matching over the generalized suffix tree of patterns mimics Ukkonen's process for inserting a new string into a generalized suffix tree (as shown in Algorithm 6), pretending to index the text, without modifying the actual tree. The text is processed in an online fashion, traversing the tree of patterns as each successive character is read. A pattern occurrence is announced each time a labeled leaf is encountered. A leaf is labeled when it represents the entire pattern, i.e., its first suffix, to indicate a pattern occurrence when traversal reaches the leaf. At a position of mismatch or a pattern occurrence, suffix links are used to navigate to successively smaller suffixes of the matching string. When a suffix link is used within the label of a node, the corresponding number of characters can be skipped, obviating redundant character comparisons. This ensures that the text is scanned

Algorithm 7 Dictionary matching over the generalized suffix tree

```
1:  curNode ← root
2:  textIndex ← 0
3:  curNodeIndex ← 0
4:  skipcount ← 0
5:  usedSkipcount ← false
6:  repeat
7:    lastNode ← curNode
8:    if usedSkipCount ≠ true then
9:      textIndex+ =curNodeIndex
10:     curNodeIndex ← 0
11:     curNode ← curNode.child(text[textIndex])
12:     if curNode.length> 0 then
13:       curNodeIndex++ {already compared the first character on the edge}
14:     end if
15:   else
16:     usedSkipCount ← false
17:   end if
18:   {compare text}
19:   while curNodeIndex<curNode.length AND curNodeIndex+textIndex<text.length
      do
20:     if text[textIndex+curNodeIndex] ≠ pat[curNode.stringNum][curNode.beg+
        curNodeIndex] then
21:       break {mismatch}
22:     end if
23:     curNodeIndex++
24:   end while
25:   if curNodeIndex=curNode.length AND curNode.firstLeaf() then
26:     announce pattern occurrence
27:   end if
28:   if curNodeIndex = curNode.length AND curNode.length >
      0 AND text[textIndex + curNodeIndex − 1] =
      pat[curNode.stringNum][curNode.beg + curNodeIndex − 1] then
29:     continue {branch and continue comparing text to patterns}
30:   end if
31:   handleMismatch
32: until textIndex+curNodeIndex ≥ text.length {scan entire text}
```

Algorithm 8 Handling a Mismatch

```
if curNode.depth ≠ 0 OR lastNode.depth ≠ 0 then
  if curNode.suffixLink = root AND lastNode.suffixLink≠ root then
    curNode ← lastNode
    curNodeIndex ← curNode.length {mismatched when trying to branch}
    textIndex − = curNode.length
  end if
  if curNode.parent = root AND curNodeIndex = 1 then
    textIndex++
    curNodeIndex = 0
    curNode = curNode.parent
    continue {when traverse suffix link: will be at mismatch, so skip 1 char}
  end if
  useSkipcountTrick(skipcount, curNode)
else
  {mismatch at root}
  textIndex++
end if
```

in linear time. We modified the peudocode in Algorithm 6 to perform dictionary matching. The pseudocode is in Algorithm 7, with its submodules extracted to Algorithms 8 and 9.

We now provide the intuition behind the skip-count trick. It is based on the property of suffix trees summarized in the following lemma. Recall that a suffix link is a directed edge from the internal node at the end of the path labeled xS to another internal node at the end of the path labeled S, where x is an arbitrary character and S is a possibly empty substring. We can similarly define suffix links for leaves in the tree. The suffix link of the leaf representing suffix i points to the leaf representing suffix $i + 1$.

Lemma 7.1.1. *In any suffix tree, the number of nodes along the path labeled α is at least as large as the number of nodes along the path from the root labeled $x\alpha$.*

Proof. The proof is by contradiction. Suppose α's path has fewer nodes than $x\alpha$'s. This means that some suffix of $x\alpha$ is not indexed by the suffix tree. This implies that the suffix

Algorithm 9 Skip-Count trick

```
repeat
  curNode ← curNode.suffixLink
  usedSkipCount ← true
  textPos = curNodeIndex+textIndex
  skipcount ← curNode.length − curNodeIndex
  if skipcount ≥ curNode.length then
    if curNode.length = 0 then
      usedSkipCount ← false {branch at next iteration of outer loop, look for next text
      char}
      curNodeIndex ← 0
      skipcount ← 0
    else
      if skipcount = curNode.length then
        curNodeIndex−−
        usedSkipCount ← false {branch at next iteration of outer loop}
      end if
      skipcount − = curNode.length
      curNode ← curNode.parent
    end if
  else
    curNodeIndex ← curNode.length − skipcount
    skipcount ← 0
  end if
until skipcount ≤ 0
textIndex = textPos − curNodeIndex
```

Algorithm 10 Announcing Pattern Occurrence in CST

if *getCharAtNodePos(curNode, curNodeIndex)* = END_OF_STRING_MARKER **then**
  pos $\leftarrow csa[lb(curNode)] - 1$
  $\{lb(v)$ returns the left bound of node v in the suffix array$\}$
  $\{$*pos* is dictionary index immediately preceding this leaf's ancestor emanating from *root*$\}$
  if *pos*< 0 **then**
    occ $\leftarrow$ *true* $\{$beginning of first pattern$\}$
  else
    $c \leftarrow$ *getCharAtPatternPos(pos)*
    if $c =$ END_OF_STRING_MARKER **then**
      occ $\leftarrow$ *true* $\{$beginning of some pattern after first$\}$
    end if
  end if
end if

tree is not fully constructed. Hence, a contradiction. □

Corollary 7.1.2. *If the suffix link of the root points to itself, every node of the suffix tree has a suffix link.*

The skip-count trick that we use is fashioned after Ukkonen's. In his suffix tree construction algorithm, he uses suffix links to navigate across and then skip down the appropriate number of characters. We navigate across the tree and then jump up to the appropriate position at which the mismatch occurred. We make this modification since it is more efficient to navigate up a tree than down. That is, every node has a single parent but when navigating to a child, several branches may have to be considered. We are able to make this improvement since our algorithm uses a fully constructed suffix tree. The suffix link of every node must already exist. On the partially constructed tree, that Ukkonen uses, this is not guaranteed. However, Ukkonen's algorithm uses the fact that the parent node must already have a suffix link. When a suffix link is traversed, we know the number of characters to skip going up the tree as this is the number of characters that remain along the

edge after the position of mismatch. Yet, we do not know at which node we will end up. The shorter label may be split over more edges than the longer label spans. This is a result of Lemma 7.1.1.

In addition to these implementation details, a key challenge in implementing dictionary matching, as pointed out in [3], is when one pattern string is a proper substring of another. In the straightforward traversal, these pattern occurrences can be passed unnoticed in the text. A solution to this problem is to augment each node of the suffix tree with an extra field that points to its lowest ancestor that is a pattern occurrence. This information can be obtained in linear time by performing a depth-first traversal of the suffix tree.

The suffix tree is a versatile tool in string algorithms, and is already needed in many applications to facilitate other queries. Thus, in practice, our dictionary matching program requires very little additional space. This tool is itself a contribution, allowing efficient dictionary matching in small space, however, we improved this application by using a compressed suffix tree as the underlying data structure.

We ported our dictionary matching code to run over a compressed suffix tree. We would have liked to create a dynamic dictionary matching program that runs in small space. However, neither of the dynamic compressed suffix tree representations have been implemented yet. None of the existing compressed suffix suffix tree representations have online construction algorithms so we cannot build the compressed suffix tree incrementally, one pattern at a time. Instead, we concatenate the dictionary with a unique delimiter, the end of string marker, between each pattern and index this single string. We used the Succinct Data Structures Library (SDSL)[1] since it provides a C++ implementation of several compressed suffix tree representations.

[1] `http://www.uni-ulm.de/en/in/institute-of-theoretical-computer-science/research/sdsl.html`

Although the ultimate capability of the compressed suffix tree is modeled after the functionality of its uncompressed counterpart, many operations that are straightforward in the uncompressed suffix tree require creativity in the compressed suffix tree. Understanding how the suffix tree components are represented in the compressed variation is a necessary prerequisite to finding these workarounds to seemingly straightforward navigational tasks. Furthermore, the compressed suffix tree is a self-index and allows us to discard the original set of patterns. Thus, we had to figure out which component data structure to query in order to randomly access a single pattern character. For instance, announcing a pattern occurrence (Algorithm 7, line 25) is not simply a question of checking whether traversal has reached the end of a leaf representing the first suffix of a pattern. A simple *if* statement is replaced by the segment of pseudocode delineated in Algorithm 10 and described in the following paragraph.

Instead of an *if* statement that checks properties of a leaf, we perform the following computation, involving several function calls, to determine if we have found a pattern in the text. When traversing the compressed suffix tree according to the text, a mismatch along an edge leading into a leaf may in fact be a pattern occurrence. Thus, we first check if the mismatch is at an end of string marker, which mismatches every text character. Then, we have to determine if this leaf represents the first suffix of some pattern. This is done by finding out which character precedes the beginning of this leaf's path from the root. If the path begins at the beginning of the dictionary, this leaf represents the first suffix of the first pattern, and we announce a pattern occurrence. Otherwise, if the character at that position is a pattern delimiter, we know this suffix is an entire dictionary pattern, and also announce a pattern occurrence.

7.2 Evaluation

We plan to assess the effectiveness of our software against space efficient 1D dictionary matching software. Specifically, we will compare our approach to that of Fredriksson [26] who achieves dictionary matching in small space and *expected* linear time using compressed self-indexes and backward DAWG matching. The space used by his algorithm is close to the information theoretic bounds of the patterns. However, the algorithm is not online in the sense that it cannot process a text as it arrives.

Since we are primarily interested in large sets of data on which dictionary matching is performed, we will use realistic sets of biological, security and virus detection data. For biological sequences, we obtained fly sequences from FlyBase[2], and flu sequences from the Influenza Virus Sequence Database[3]. Network intrusion detection system signatures are readily available at ClamAV[4], and virus signatures at Snort[5].

[2]http://flybase.org/static_pages/downloads/bulkdata7.html
[3]http://www.ncbi.nlm.nih.gov/genomes/FLU/Database/nph-select.cgi?go=database
[4]http://www.clamav.net
[5]http://www.snort.org/

Chapter 8

Conclusion

This thesis has contributed several algorithms for efficient yet succinct 2D dictionary matching. We have developed the first linear-time small-space 2D dictionary matching algorithm. We have extended our focus to the dynamic setting and developed an algorithm whose running time is linear, besides the slowdown to query the compressed suffix tree of the dictionary. Yet its extra working space is quite limited.

The algorithms developed in this thesis for 2D data are suitable for rectangular patterns that are all the same size in at least one dimension. We would like to expand our focus to consider dictionary matching for other kinds of 2D dictionaries in the space-constrained environment. Succinct 2D dictionary matching among square patterns of different sizes has not yet been addressed and requires a different set of techniques.

When the dictionary contains rectangular patterns of different height, width and aspect ratios, a method that labels text positions is not appropriate. Idury and Schaffer developed both static and dynamic dictionary matching algorithms for such patterns [41]. They use techniques for multidimensional range searching as well as several applications of the Bird / Baker algorithm, after splitting each pattern into overlapping pieces and handling these

segments in groups of uniform height. Idury and Schaffer's algorithms require working space proportional to the dictionary size. The static version has an $O(|D|\log(d+\sigma))$ slowdown to preprocess a dictionary D of d patterns and an $O(\log^2|D|\log(B+d+\sigma))$ slowdown to scan the text, where B is the largest size of any pattern. The dynamic version has an $O(\log^4|D|)$ slowdown to preprocess the dictionary, insert, or remove a pattern, and an $O(\log^4|D|\log B)$ slowdown to scan the text, where B is the largest size of any pattern. We see our succinct version of the Bird / Baker algorithm as a first step towards addressing the more general problem of succinct static and dynamic 2D dictionary matching among all rectangular patterns.

Other interesting variations of small-space 2D dictionary matching include the approximate versions of the problem in which one or more changes occur either in the patterns or in the text. The approximate matches may accommodate character mismatches, insertions, deletions, "don't care" characters, or swaps. Beyond their theoretical intrigue, these problems all have many practical applications. We hope to explore these problems in future research.

We have developed software for succinct 1D dictionary matching. We would like to develop software for *dynamic* succinct dictionary matching, in which a pattern can be inserted to or removed from the dictionary without reprocessing the entire dictionary. Our program relies on the compressed suffix tree to index the dictionary. A natural way of generalizing this program to the dynamic setting would use a dynamic compressed suffix tree as the underlying index. However, the existing dynamic compressed suffix tree representations [14, 56] have not yet been implemented.

Our algorithms for succinct 2D dictionary matching employ techniques for succinct 1D dictionary matching. Thus, we see our 1D dictionary matching program as a first step in

developing software for succinct 2D dictionary matching. We hope to expand our dictionary matching software to employ our new techniques and to perform dictionary matching among 2D data.

Bibliography

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. *SIAM Journal on Computing*, 23(2):313–323, 1994.

[3] A. Amir and M. Farach. Adaptive dictionary matching. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 760–766, 1991.

[4] A. Amir and M. Farach. Two-dimensional dictionary matching. *Information Processing Letters*, 44(5):233–239, 1992.

[5] A. Amir, M. Farach, Z. Galil, R. Giancarlo, and K. Park. Dynamic dictionary matching. *Journal of Computer and System Sciences*, 49(2):208–222, 1994.

[6] A. Amir, M. Farach, R. M. Idury, J. A. L. Poutré, and A. A. Schäffer. Improved dynamic dictionary matching. *Information and Computation*, 119(2):258–282, 1995.

[7] A. Amir, G. M. Landau, and D. Sokol. Inplace 2d matching in compressed images. *Journal of Algorithms*, 49(2):240–261, 2003.

[8] T. J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM Journal on Computing*, (7):533–541, 1978.

[9] D. Belazzougui. Succinct dictionary matching with no slowdown. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 88–100, 2010.

[10] M. A. Bender and M. Farach-Colton. The lca problem revisited. In *Latin American Theoretical Informatics Symposium (LATIN)*, pages 88–94, 2000.

[11] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[12] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commununications of the ACM*, 20(10):762–772, 1977.

[13] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *Symposium on Experimental Algorithms (SEA)*, pages 94–105, 2010.

[14] H.-L. Chan, W.-K. Hon, T.-W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms*, 3(2), 2007. Article 21.

[15] Y. Choi and T.-W. Lam. Two-dimensional dynamic dictionary matching. In *International Symposium on Symbolic and Algebraic Computation (ISAAC)*, pages 85–94, 1996.

[16] Y. Choi and T. W. Lam. Dynamic suffix tree and two-dimensional texts management. *Information Processing Letters*, 61(4):213–220, 1997.

[17] R. Cole and R. Hariharan. Dynamic LCA queries on trees. *SIAM Journal on Computing*, 34(4):894–923, 2005.

[18] M. Crochemore, L. Gasieniec, R. Hariharan, S. Muthukrishnan, and W. Rytter. A constant time optimal parallel algorithm for two-dimensional pattern matching. *SIAM Journal on Computing*, 27(3):668–681, 1998.

[19] M. Crochemore, L. Gasieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 181–192, 1995.

[20] M. Crochemore and D. Perrin. Two-way string-matching. *Journal of the ACM*, 38(3):650–674, 1991.

[21] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 184–196, 2005.

[22] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[23] P. Ferragina and R. Venturini. A simple storage scheme for strings achieving entropy bounds. *Theoretical Computer Science*, 372(1):115–121, 2007.

[24] J. Fischer. Wee LCP. *Information Processing Letters*, 110(8-9):317–320, 2010.

[25] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theoretical Computer Science*, 410(51):5354–5364, 2009.

[26] K. Fredriksson. Succinct backward-DAWG-matching. *ACM Journal of Experimental Algorithmics*, 13, 2009. Article 8.

[27] K. Fredriksson and F. Nikitin. Simple random access compression. *Fundamenta Informatica*, 92(1-2):63–81, 2009.

[28] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *ACM Symposium on Theory of Computing (STOC)*, pages 135–143, 1984.

[29] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *ACM Symposium on Theory of Computing (STOC)*, pages 106–113, 1981.

[30] L. Gasieniec and R. M. Kolpakov. Real-time string matching in sublinear space. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 117–129, 2004.

[31] R. Giancarlo. A generalization of the suffix tree to square matrices, with applications. *SIAM Journal on Computing*, 24(3):520–562, 1995.

[32] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[33] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[34] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *ACM Symposium on Theory of Computing (STOC)*, pages 397–406, 2000.

[35] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

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

[37] W.-K. Hon, T.-H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster compressed dictionary matching. In *Symposium on String Processing and Information Retrieval (SPIRE)*, pages 191–200, 2010.

[38] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Compressed index for dictionary matching. In *Data Compression Conference (DCC)*, pages 23–32, 2008.

[39] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Succinct index for dynamic dictionary matching. In *International Symposium on Symbolic and Algebraic Computation (ISAAC)*, pages 1034–1043, 2009.

[40] R. M. Idury and A. A. Schäffer. Dynamic dictionary matching with failure functions. *Theoretical Computer Science*, 131(2):295–310, 1994.

[41] R. M. Idury and A. A. Schäffer. Multiple matching of rectangular patterns. *Information and Computation*, 117(1):78–90, 1995.

[42] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 943–955, 2003.

[43] D. K. Kim, J. S. Sim, H. Park, , and K. Park. Linear-time construction of suffix arrays. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 186–199, 2003.

[44] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[45] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 200–210, 2003.

[46] M. Lothaire. *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, New York, NY, USA, 2005.

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

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

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

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

[51] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1), 2007. Article 2.

[52] S. Neuburger and D. Sokol. Succinct 2d dictionary matching. *Algorithmica*, pages 1–23. 10.1007/s00453-012-9615-9.

[53] S. Neuburger and D. Sokol. Small-space 2d compressed dictionary matching. In *Symposium on Combinatorial Pattern Matching (CPM)*, pages 27–39, 2010.

[54] S. Neuburger and D. Sokol. Succinct 2d dictionary matching with no slowdown. In *Algorithms and Data Structures Symposium (WADS)*, pages 619–630, 2011.

[55] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *Symposium on String Processing and Information Retrieval (SPIRE)*, pages 322–333, 2010.

[56] L. M. S. Russo, G. Navarro, and A. L. Oliveira. Fully compressed suffix trees. *ACM Transactions on Algorithms*, 7(4):53:1–53:34, 2011.

[57] W. Rytter. On maximal suffixes and constant-space linear-time versions of kmp algorithm. *Theoretical Computer Science*, 299(1-3):763–774, 2003.

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

[59] K. Sadakane. Compressed suffix trees with full functionality. *Theory of Computing Systems*, 41(4):589–607, 2007.

[60] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 320–328, 1996.

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

[62] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM Journal of Experimental Algorithmics*, 14, 2009. Article 2.

[63] P. Weiner. Linear pattern matching algorithm. In *IEEE Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[64] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.