Small-Space Dictionary Matching

Dissertation Proposal

Shoshana Neuburger

Department of Computer Science

The Graduate Center, CUNY

Abstract

The dictionary matching problem seeks all location 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 project focuses on the problem of performing dictionary matching in one and two dimensions within small space. In one-dimension, the theoretical ideas for performing dictionary matching in small space have mostly been developed. The major challenge is in the algorithmic engineering of combining the succinct data structures with an algorithm for the generalized suffix tree. In two-dimensions, we will develop new algorithms to perform dictionary matching in small space and linear time.

Contents

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 and we begin by presenting them in Section 3. The related problem of small-space dictionary matching in two-dimensional data has not been addressed until now. This project seeks to fill this void. We present relevant background and intuition for our techniques in Section 4. After our succinct 2D dictionary matching algorithm is complete, we plan to expand our focus to the setting in which the dictionary is *dynamic* and can change over time.

Small-space dictionary matching in one-dimensional data was recently closed with the development of an algorithm that meets optimal time and space bounds. Yet, there is a lag in the implementation of these theoretical contributions. This is likely due to their complexity and the novelty of their data structures. Thus, in this project we propose to develop a succinct dictionary matching program that is more intuitive and relies on more commonly used data structures. We describe our approach in Section 6 after describing the indexing data structures we rely upon in Section 5. We will then evaluate the time and space usage of our approach. Since dictionary matching is such a practical problem, we plan to use realistic data sets in our evaluation.

2 Research Plan

The objective of my thesis project is to develop algorithms and software for succinct dictionary matching. In particular, the three major goals of this project are to:

- Create the first succinct 2D dictionary matching algorithm. Our algorithm will run in linear time and small-space. We present background and intuition to our approach in Section 4.2.
- Modify our algorithm for succinct 2D dictionary matching to make it dynamic. It should be able to adapt to changes in the dictionary without reprocessing the entire dictionary. We discuss dynamic 2D dictionary matching in Section 4.3. None of these algorithms are suitable for the space-constrained environment.
- Develop software that efficiently solves one-dimensional dictionary matching in small space. Our technique is based on the compressed suffix tree, which forms a compressed self-index of the dictionary of patterns. We give an overview of our algorithmic approach and software-design considerations in Section 6.

3 1D Dictionary Matching

3.1 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 $D = \{P_1, P_2, \ldots, P_d\}$ is called the *dictionary.*

We can define dictionary matching by:

> INPUT: A set of patterns $P_1, P_2, \ldots, P_d$ of total length ℓ and a text $T = t_1t_2\ldots t_n$ 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 [37]. 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. However, hashing techniques can achieve linear time complexity in the Aho-Corasick algorithm.

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. [12]
$\ell H_k(D) + o(\ell \log \sigma) + O(d \log \ell)$	$O(n(\log^\epsilon \ell + \log d) + occ)$	Hon et al. [31]
$\ell(H_0(D) + O(1)) + O(d \log(\ell/d))$	$O(n + occ)$	Belazzougui [8]
$\ell H_k(D) + O(\ell)$	$O(n + occ)$	Hon et al. [30]

Table 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.

3.2 Small-Space

Linear-time *single* pattern matching algorithms in both one and two dimensions have achieved impressively small space complexities. For 1D data, we have pattern matching algorithms that require only constant extra space [23, 16, 46, 24]. The first time-space optimal pattern matching algorithm is from Galil and Seiferas [23]. Crochemore and Perrin developed "Two-Way String Matching" which blends the classical Knuth-Morris-Pratt and Boyer-Moore [10] algorithms but computes pattern shifts as needed [16]. Rytter presented a constant-space, yet linear-time version of the Knuth-Morris-Pratt algorithm [46]. 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 [24]. 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 1-dimensional dictionary matching in small space [12, 31, 8, 30]. We summarize the state of the art for small-space 1D dictionary matching in Table 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 1. Precise formulas for H_0 and H_k are included in Appendix 7.3.

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 $O(\ell \log \ell)$ 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. [12]. 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 [31]. 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 [8]. 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 trie, fail, and report functions separately. Hon et al. combine Belazzougui's work with the XBW transform to store an AC automaton in space that meets kth order empirical entropy bounds of the dictionary with no slowdown [30]. They follow the approach of Belazzougui [8] to compress the AC automaton and store the trie, fail, and report functions separately. However, they encode the forward transitions of the trie with the XBW transform [18]. With this new representation, the space meets optimal compression of the dictionary and runtime is linear.

The most recent result of Hon et al. [30] 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 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.

3.3 Dynamic Dictionary

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 circumvent the need to reprocess the entire dictionary and can adapt to changes as they occur.

Amir and Farach [2] introduced the use the suffix tree for dictionary matching. They delimit the dictionary patterns 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 prefix of another, the suffix tree contains all the information needed to perform dictionary matching. When one pattern can be a prefix 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. [4] 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 [2], 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 [33]. The fail function is updated efficiently 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. This is an improvement over [2, 4], which incur an $O(\log \ell)$ slowdown in preprocessing. However, the other phases of the algorithm incur a slight slowdown, as in [2, 4]. Text scanning runs in $O((n+occ)\log \ell)$ time and a pattern P, of length p, is added to or removed from the the dictionary in $O(p \log \ell)$ time.

Idury and Schaffer explore alternative representations of their dynamic AC automaton. They point out that other tradeoffs between search and update times are possible. Using a different data structure this algorithm achieves 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. [5] 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 [49]. 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/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 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. [12] use the compressed suffix tree for succinct dictionary matching. They build on the work of Amir and Farach [2] to use the suffix tree for dictionary matching. However, they use the earliest compressed suffix tree, developed by Sadakane [48], 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 / removing a pattern and of scanning text has a slowdown of $O(\log^2 \ell)$. The index they employ is Sadakane’s compressed suffix tree, which is stored in $O(\ell)$ bits.

An improved succinct dynamic dictionary matching algorithm was developed by Hon et al. [32]. 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.

4 2D Dictionary Matching

4.1 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 [9] and by Baker [7]. 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 and 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 the Bird and Baker algorithm was initially presented 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.

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 [3] 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 again proportional to the size of the text plus the patterns of names. This is considered a linear-time algorithm since the $\log d$ slowdown stems from branching in the AC automaton.

Giancarlo developed the first 2D suffix tree [25]. 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, text scanning simulates an automaton in $O(n^2 \log \overline{D} + occ)$ time.

Idury and Schaffer [34] 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 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.

4.2 Small-Space

An approach for small-space, yet linear-time *single* pattern matching in 2D was developed by Crochemore et al. [15]. Their algorithm preprocesses a pattern of length 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.We propose to develop an efficient algorithm for 2D dictionary matching whose space requirements meet the empirical entropy bounds of the dictionary.

This project will develop the first efficient 2D dictionary matching algorithm that operates in small space. Given 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 will find every occurrence of P_i, $1 \leq i \leq d$, in T. We will form a compressed self-index of the patterns, after which the original dictionary may be discarded. We want to limit the space usage of our algorithm to space proportional to the kth order empirical entropy of the dictionary, $H_k(D)$, with possibly an extra $O(dm \text{ log } dm)$ bits. The time complexity of our new algorithm will be close to linear.

Our algorithm will preprocess the dictionary of patterns before searching the text once for all patterns in the dictionary. The text scanning stage will initially filter the text to a limited number of *candidate* positions and then verify which of these positions are actual pattern occurrences. As is common in many space-constrained algorithms, we will divide the text into small blocks, and work with one block of text at a time. We will limit our algorithm to $O(dm \log dm)$ bits of working space to process the text and find all pattern occurrences. The text scanning stage will not depend on the size of the dictionary.

We propose to divide patterns into two groups based on 1D periodicity. Our algorithm will consider each of these cases separately. A pattern can consist of rows that are periodic with period $\leq m/4$ (Case I). Alternatively, a pattern can have one or more possibly aperiodic rows whose periods are larger than $m/4$ (Case II). We are faced with different bottlenecks in each of these cases. In the case of a pattern whose rows are highly periodic, one pattern can overlap itself with several occurrences in close proximity to each other. Thus, we can easily have more candidates than the space we allow. In the case of an aperiodic pattern row, we have a smaller number of potential pattern occurrences, but several patterns can overlap in both directions. For this reason, we realize that the dictionary must be represented in a way that permits random access to any pattern character.

For Case I, we developed a scheme that generalizes the naming technique used by Bird [9] and Baker [7] (Algorithm 1) to represent 2D data in 1D. We overcome the space requirement of traditional 2D dictionary matching algorithms with an innovative preprocessing scheme that groups pattern rows, 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.

For Case II, we use the aperiodic row of each pattern to filter the text and limit the number of candidate positions in the text. Verifying the candidates poses a challenge. We propose to use recent developments in compressed self-indexing to rapidly answer queries about the dictionary within limited working space.

Our next goal is to concentrate on square patterns of different sizes. We would like to adapt Amir and Farach's 2D dictionary matching algorithm for square patterns [3] to a small-space version. By replacing the underlying data structures in this algorithm with a dynamic compressed suffix tree we hope to meet this goal. Compressed suffix trees are discussed in Section 5.3.2.

4.3 Dynamic Dictionary

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 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 4.1. 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 $n \times n$. 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 $\overline{D} = \sum_{i=1}^{d} p_i$.

Dictionary Update Time	**Text Searching Time**	**Reference**
$O(p^2 \log \|D\|)$	$O((n^2 + occ) \log \|D\|)$	Amir et al. [5]
$O(p^2 \log^2 \|D\|)$	$O((n^2 \log \overline{D} + occ) \log \|D\|)$	Giancarlo [25]
$O(p^2 + p \log \overline{D})$	$O((n^2 + occ) \log \overline{D})$	Choi and Lam [13]

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

Amir et al. [5] 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 [33] 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 a $\log |D|$ slowdown is incurred.

Giancarlo's 2D suffix tree can be used for dictionary matching in the case that the dictionary is static and in the case that the dictionary is dynamic [25]. 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. [5] achieved slightly better time complexity.

Choi and Lam [13] set out to demonstrate that Giancarlo's suffix tree based approach to dictionary matching is just as good as Amir's 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 [34]. 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.

5 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. Suffix trees and suffix arrays can be generalized to two or more dimensions. 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 may even attain better

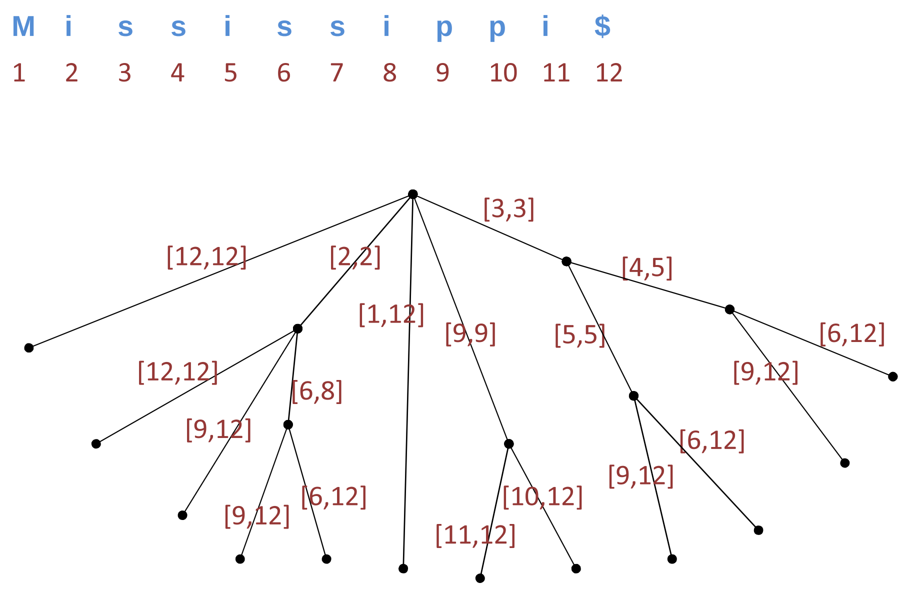

Figure 1: Suffix tree for the string Mississippi.

space complexity than the original text.

5.1 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 S and no two edges out of a node begin with the same character. The path from the root to leaf i spells out suffix $S[i \ldots n]$. 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 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 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 [29], 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 was first introduced by Weiner in 1973 as a position tree [53]. The construction was greatly simplified by McCreight in 1976 [42]. 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 construction of suffix trees in linear time [51]. 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 in [26].

Suffix links are an implementation trick necessary to achieve linear time and space bounds 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.

Kosaraju improved the suffix tree of Weiner to an alphabet independent data structure. Amir and Nor [6] combined Kosaraju's quasi-real time algorithm [39] with Ukkoknen's algorithm in 2008, to produce a real-time linear suffix tree construction algorithm. That is, they deamortized an existing online algorithm. Their main contribution is in changing the order in which suffixes are inserted into the tree.

5.1.1 Dynamic Suffix Tree

Instances arise in which the text indexed by the suffix tree changes over time. Substrings can be added to or removed from the text. We call such a suffix tree a dynamic suffix tree. Furthermore, the suffix tree can be generalized to represent a set of strings in such a way that strings can be added and removed efficiently. An efficient implementation for the dynamic suffix tree is described by Choi and Lam [14]. 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. Their method requires a two-way pointer for each edge, which can be stored in space proportional to the size of the tree.

5.2 Suffix Array

A suffix array stores the lexicographic order of the suffixes of the text under consideration. The suffix array achieves greater space efficiency than the suffix tree. Augmented with an LCP array to store the longest common prefix between adjacent suffixes, a binary search on the suffix array can locate all instances of a pattern in the text.

The naive approach to suffix array construction is to sort the suffixes using a string sorting algorithm. This approach ignores the fact that the suffixes are related. The worst case complexity of such an algorithm is proportional to the sum of the lengths of all suffixes, $O(n^2)$. 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. This indirect construction does not achieve better space complexity than the suffix tree since the suffix tree is constructed along the way.

Manber and Myers introduced the suffix array in 1993 [40]. Their algorithm employs a sort and search paradigm. Simply sorting the suffixes is not enough to produce an efficient pattern search

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: Suffix Array of the string MISSISSIPPI

mechanism. The precomputation and utilization of the longest common prefix (LCP) between the suffixes make the technique very efficient. In the worst case, preprocessing a text of length n requires $O(n \log n)$ time to sort the suffixes and $O(n)$ time to compute the LCPs, all within $O(n)$ space. Their algorithm achieves expected running time of $O(n)$, independent of the alphabet size. Searching for a pattern of length m takes $O(m \log n)$ time using a binary search on the suffix array and the text. This is improved to $O(m + \log n)$ time using the suffix array with the LCP array. The enhanced suffix array searches for a pattern of length m in $O(m|\Sigma|)$ time. This requires an additional array, the *child* table.

The efficient suffix sort is based on the *doubling technique* of Karp, Miller, and Rosenberg [36]. The idea is to assign a *rank* to all substrings whose length is a power of two. The rank tells the lexicographic order of the substring among substrings of the same length. The method iteratively applies a linear sorting mechanism, such as bucket sort or radix sort. The algorithm sorts each suffix by its first k letters, then doubles k to use the results of the previous round. There are $\log n$ phases of sorting. In total, $O(n \log n)$ work is done to sort the suffixes. Since the method is iterative, the space is bounded by the text size. Once the suffix array has been constructed, the longest common prefix (LCP) array can be created in linear time and space.

In 2003, three different algorithms were introduced to directly construct a suffix array in linear time. They were published by Kärkkäinen and Sanders [35], Ko and Aluru [38], and Kim et al. [17]. Kärkkäinen and Sanders apply a divide and conquer approach to the algorithm of Manber and Myers. First the suffix array of the suffixes starting at positions $i \, mod \, 3 \neq 0$ is constructed, then a suffix array of the other third of the suffixes is constructed. The two suffix arrays are merged by comparison.

A suffix array with an LCP array can simulate the bottom-up traversal of its suffix tree and can answer lowest common ancestor (LCA) queries. With an additional table, an enhanced suffix array can simulate top-down traversal of a suffix tree for a full range of suffix tree functionality. This can all be done in time and space linear in the size of the text. The LCP array can be computed by efficient range minimum queries.

Suffix sorting algorithms can be used to perform the Burrows-Wheeler transform (BWT). The Burrows-Wheeler transformation permutes the order of the characters and is easily reversed to obtain the original string. The BWT requires sorting cyclic permutations of a string, not suffixes. We can append to the string a special character which is lexicographically smaller than every other character. Then, sorting cyclic permutations is then equivalent to sorting suffixes. The BWT is more readily compressed than the original string, with either run-length encoding or move-to-front encoding. The BWT array can be computed from the suffix array using the following equations.

$$\begin{aligned} &BWT[i] = T[SA[i] - 1], && \text{when } SA[i] \neq 0 \\ &BWT[i] = T[n - 1], && \text{when } SA[i] = 0 \end{aligned}$$

5.2.1 Dynamic Suffix Array

Recent work by Salson et al. [50] addresses the scenario in which text is edited after its suffix array has been constructed. They developed an algorithm for dynamically updating the suffix array, which is more efficient than rebuilding the suffix array from scratch. Even though the theoretical worst-case time complexity is $O(n\, log\, n)$, it performs much better in practice. The online version has yet to be developed.

5.3 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 uncompressed data structures 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 k^{th}-order compressor, thus relating the index size to the compressibility of the text. Empirical entropy is defined in Section 7.3.

5.3.1 Compressed Suffix Array

Since the suffix array is an array of n indices, it can be stored in $n \log n$ bits. The compressed suffix array was introduced by Grossi and Vitter [28] 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 [47]. 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 $O(n \log H_0(n))$ 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 $(1 + \frac{1}{\epsilon})nH_k + o(n)$ bits, with character lookup time of $O(\log^{\epsilon} n)$, assuming an alphabet size $\sigma = O(polylog(n))$ [27].

Ferragina and Manzini developed the first compressed suffix array to encode the index size with respect to the high-order empirical entropy [19]. 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 [43]. 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 [43]. There are also different options for compressing the LCP array.

5.3.2 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 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-offs. We plan to use one of the compressed suffix tree representations in our succinct dictionary matching software which we describe in Section 6.

1. Sadakane introduced the first linear representation of suffix trees that supports all navigation operations efficiently [48]. 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. [52].

   This compressed suffix tree has been adapted for a dynamically changing set of patterns by Chan et al. [12]. They represent the compressed suffix tree as the combination of (1) balanced parentheses (2) CSA and FM index (3) LCP array. A substring of size p is inserted or removed from the index in $O(p \log^2 n)$ time, where n is the total size of the indexed text. A dictionary matching query that finds all pattern occurrences in a text of size t runs in $O((t + occ) \log^2 n)$ time. An edge label can be retrieved in $O(\log^2 n)$ time.

2. Russo et al. [45] achieved a fully-compressed suffix tree requiring $nH_k + o(n \log \sigma)$ 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 have $O(\log n)$ 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 [11]. The dynamic fully-compressed suffix tree has not yet been implemented. This dynamic compressed suffix tree supports a larger set of suffix tree navigation operations than the compressed suffix tree proposed by Chan et al. [12]. 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 [21]. 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 $2H_k(2 \log \frac{1}{H_k} + \frac{1}{\epsilon} + O(1)) + o(n)$ bits of space [21]. This data structure has been implemented and evaluated by Cánovas and Navarro [11].

   With Fischer's new compressed representation of the LCP array [20], the compressed suffix tree of Fischer et al. [21] can be stored in even smaller space. That is, the suffix tree can be stored in $(1 + \frac{1}{\epsilon})nH_k + o(n \log \sigma)$ 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 [44]. They point out that the CST 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. [45] 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 [44].

6 Implementation

6.1 Software Development

We are developing 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 are available as part of the Succinct Data Structures Library [1]. 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 [51] suffix tree construction algorithm extends quite naturally to the construction of a generalized suffix tree for several strings [29], 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. We proceeded by writing a program that performs dictionary matching over a text by traversing the generalized suffix tree without modifying 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 2.

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

Algorithm 2 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 $lastNodeInserted \neq root$ then
         $lastNodeInserted.SuffixLink \leftarrow root$;
      end if
      $lastNodeInserted \leftarrow root$;
   end while
end for
```

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 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 an internal node labeled xS to another internal node labeled S, where x is an arbitrary character and S is a possibly empty substring. 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. $O(1)$ 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 2), 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. At a position of mismatch, suffix links are used to navigate to

Algorithm 3 Dictionary matching over the generalized suffix tree

```
$curNode \leftarrow root$
$textIndex \leftarrow 0$
$curNodeIndex \leftarrow 0$
$skipcount \leftarrow 0$
$usedSkipcount \leftarrow false$
repeat
  $lastNode \leftarrow curNode$
  if $usedSkipCount \neq true$ then
    $textIndex+ = curNodeIndex$
    $curNodeIndex \leftarrow 0$
    $curNode \leftarrow curNode.child(text[textIndex])$
    if $curNode.length > 0$ then
      $curNodeIndex{+}{+}$ {already compared the first character on the edge}
    end if
  else
    $usedSkipCount \leftarrow false$
  end if
  {compare text}
  while $curNodeIndex < curNode.length$ AND $curNodeIndex + textIndex < text.length$ do
    if $text[textIndex + curNodeIndex] \neq pat[curNode.stringNum][curNode.beg + curNodeIndex]$ then
      break {mismatch}
    end if
    $curNodeIndex{+}{+}$
  end while
  if $curNodeIndex = curNode.length$ AND $curNode.firstLeaf()$ then
    announce pattern occurrence
  end if
  if $curNodeIndex = curNode.length$ AND $curNode.length > 0$ AND $text[textIndex + curNodeIndex - 1] = pat[curNode.stringNum][curNode.beg + curNodeIndex - 1]$ then
    continue {branch and continue comparing text to patterns}
  end if
  if $curNode.depth \neq 0$ OR $lastNode.depth \neq 0$ then
    if $curNode.suffixLink = root$ AND $lastNode.suffixLink \neq root$ then
      $curNode \leftarrow lastNode$
      $curNodeIndex \leftarrow curNode.length$ {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
until $textIndex + curNodeIndex \geq text.length$ {scan entire text}
```

Algorithm 4 Skip-Count trick

```
repeat
  $curNode \leftarrow curNode.suffixLink$
  $usedSkipCount \leftarrow true$
  $textPos = curNodeIndex{+}textIndex$
  $skipcount \leftarrow curNode.length - curNodeIndex$
  if $skipcount \geq curNode.length$ then
    if $curNode.length = 0$ then
      $usedSkipCount \leftarrow false$ {branch at next iteration of outer loop, look for next text char}
      $curNodeIndex \leftarrow 0$
      $skipcount \leftarrow 0$
    else
      if $skipcount = curNode.length$ then
        $curNodeIndex{-}{-}$
        $usedSkipCount \leftarrow false$ {branch at next iteration of outer loop}
      end if
      $skipcount - = curNode.length$
      $curNode \leftarrow curNode.parent$
    end if
  else
    $curNodeIndex \leftarrow curNode.length - skipcount$
    $skipcount \leftarrow 0$
  end if
until $skipcount \leq 0$
$textIndex = textPos - curNodeIndex$
```

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. We modified the peudocode in Algorithm 2 to perform dictionary matching. The pseudocode is in Algorithms 3 and 4.

In addition to these implementation details, a key challenge in implementing dictionary matching, as pointed out in [2], 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 have begun to improve this application by using a compressed suffix tree as the underlying data structure.

If we use an existing compressed suffix tree implementation as a black-box, then we may confront a difficulty in augmenting the nodes with the additional information on nearest marked ancestors. However, a straightforward solution to this problem may be obtained by storing this information in a separate compressed data structure. We can use ideas similar to those of Amir and Farach [5], where this information is stored separately. They reduced the problem of *nearest marked ancestor* to the *parenthesis maintenance* problem.

6.2 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 [22] 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/`

7 Useful Terminology

7.1 Periodicity

Computing the period of a string is a crucial task in the preprocessing stage of many pattern matching algorithms. 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 longest border of a string 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 $|u| < \frac{|S|}{2}$.

7.2 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 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 $uv < vu$. A Lyndon word is primitive.

Any primitive 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*.

7.3 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 0-th 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: [41]

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

References

[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 and M. Farach. Adaptive dictionary matching. In *FOCS*, pages 760–766, 1991.

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

[4] 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.

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

[6] A. Amir and I. Nor. Real-time indexing over fixed finite alphabets. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1086–1095, 2008.

[7] 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.

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

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

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

[11] R. Cánovas and G. Navarro. Practical compressed suffix trees. In *SEA*, pages 94–105, 2010.

[12] Ho-Leung Chan, Wing-Kai Hon, Tak-Wah Lam, and Kunihiko Sadakane. Compressed indexes for dynamic text collections. *ACM Trans. Algorithms*, 3(2), May 2007. Article 21.

[13] Y. Choi and T.-W. Lam. Two-dimensional dynamic dictionary matching. In *ISAAC*, pages 85–94, 1996.

[14] Y. Choi and T. W. Lam. Dynamic suffix tree and two-dimensional texts management. *Information Processing Letters*, pages 213–220, 1997.

[15] M. Crochemore, L. Gasieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 1995.

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

[17] H. Park D. K. Kim, J. S. Sim and K. Park. Linear-time construction of suffix arrays. In *Proc. Annual Symp. on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 186–199. Springer Verlag, 2003.

[18] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. In *FOCS*, pages 184–196, 2005.

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

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

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

[22] K. Fredriksson. Succinct backward-DAWG-matching. *ACM Journal of Experimental Algorithmics*, 13:8:1.8–8:1.26, 2009.

[23] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 106–113, New York, NY, USA, 1981. ACM.

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

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

[26] 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.

[27] 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.

[28] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC*, pages 397–406, 2000.

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

[30] W.-K. Hon, T.-H. Ku, R. Shah, S. V. Thankachan, and J. S. Vitter. Faster compressed dictionary matching. In *SPIRE*, pages 191–200, 2010.

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

[32] W.-K. Hon, T. W. Lam, R. Shah, S.-L. Tam, and J. S. Vitter. Succinct index for dynamic dictionary matching. In *ISAAC*, pages 1034–1043, 2009.

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

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

[35] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[36] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC*, pages 125–136, 1972.

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

[38] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 200–210. Springer Verlag, 2003.

[39] S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix tree. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 310–316, Montréal, Canada, 1994. ACM Press.

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

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

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

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

[44] E. Ohlebusch, J. Fischer, and S. Gog. CST++. In *SPIRE*, pages 322–333, 2010.

[45] 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.

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

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

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

[49] S. C. Sahinalp and U. Vishkin. Efficient approximate and dynamic matching of patterns using a labeling paradigm (extended abstract). In *FOCS*, pages 320–328, 1996.

[50] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *J. Discrete Algorithms*, 8(2):241–257, 2010.

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

[52] N. Välimäki, V. Mäkinen, W. Gerlach, and K. Dixit. Engineering a compressed suffix tree implementation. *ACM Journal of Experimental Algorithmics*, 14:2:4.2–2:4.23, 2009.

[53] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.