Small-Space Dictionary Matching

Shoshana Neuburger

Abstract

Dictionary matching is the problem of finding any pattern in a given set of patterns in the given text. It has applications in diverse areas, among them molecular biology, WWW search engines, online databases, and multimedia. In this project we address succinct dictionary matching in one and two dimensions since many devices have limited storage capacity. Although several algorithms have been developed for 1-dimensional dictionary matching that meet optimal space bounds, they have not been implemented. We propose a different approach that yields a more implementable algorithm and will deploy the method with the use of algorithmic engineering. This project is the first to address 2-dimensional dictionary matching in a space-constrained environment. The objectives of this project are to develop efficient algorithms and software for dictionary matching in one and two dimensions whose execution requires very little space.

1 Introduction

The classical textual pattern matching problem consists of finding all occurrences of a given pattern string P, in a given text T. The solution to this problem has found a variety of applications, including querying a database, searching the web, and searching a DNA sequence for a motif. The *dictionary matching problem* is an extension of this and attempts to identify a set of patterns, $D = \{P_1, \ldots, P_k\}$, which is called a dictionary, within a given text T. A search for specific phrases in a book, scanning for virus signatures, and network intrusion detection, are all applications of dictionary matching. Biological applications include searching a DNA sequence for a set of motifs, or for a regular expression. Image identification software, which identifies smaller images in a large image based on a set of known images, is a direct application of 2-dimensional dictionary matching. An efficient dictionary matching algorithm should search for all patterns in one pass through the text, rather than searching for each pattern individually.

In this work we are concerned with efficiently solving the dictionary matching problem in *small space.* The motivation for this is that there are many scenarios, such as on mobile and satellite devices, where storage capacity is limited. The challenge in developing search algorithms for space-constrained applications takes on a new dimension. In addition to reducing the running time of the algorithm, the algorithm must operate using very limited extra space.

A lot of effort has been devoted recently to solving 1-dimensional dictionary matching in small space [7, 14, 15, 4, 13]. Although the final result achieves linear-time dictionary matching in optimal space, meeting kth order empirical entropy bounds, none of these results have been implemented. Furthermore, the algorithms are complex and difficult to implement. The space-efficient implementations that exist are based upon different algorithms that have been tailored toward specific applications. For example, [21] was designed for use in network intrusion detection systems as an improvement over traditional Aho-Corasick. Fredriksson's software [10] for succinct dictionary matching has optimal

average time, and is only efficient for certain types of data (e.g. large alphabets, texts that are available in reverse order).

The first goal of this project is to present a simple, implementable algorithm for 1D dictionary matching in small space. Our algorithm is general in the sense that it will work well for any alphabet size, and for large inputs. We will begin by implementing a dictionary matching program that relies on the generalized suffix tree , then improving on this method by merging the algorithm with a compressed suffix tree representation [6]. The suffix tree is a versatile tool often needed in many applications. Our tool will augment the suffix tree to allow the additional functionality of fast dictionary matching, using very little additional space.

The second component of this project works with 2-dimensional patterns and texts. To date, 2-dimensional dictionary matching in small space has been neglected, although algorithms for 2D pattern matching in small space have been achieved [8]. We will develop the first efficient 2D dictionary matching algorithm to operate in small space. For highly periodic patterns, our algorithm will use a newly developed preprocessing scheme [18] that is based upon succinctly representing periods by their Lyndon words [17]. For patterns that are not periodic, we will extend the ideas for 1D dictionary matching with compressed suffix trees, using the methodology that converts a 2D pattern to a linear representation [3, 5]. The ideas once again are clear and simple, yielding an algorithm that is practical to implement.

2 Background

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 $T = t_1 t_2 \ldots t_n$ all over an alphabet Σ, with alphabet-size $|\Sigma| = \sigma$.
- OUTPUT: All ordered pairs (i, j) such that pattern P_j matches the segment of text beginning at location t_i.

2.1 Dictionary Matching in One-Dimensional Data

Knuth, Morris, and Pratt developed a well-known linear-time algorithm for pattern matching [16]. 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 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 and does not depend on the dictionary size.

Since applications arise in which additional storage space is extremely limited, effort has been invested in developing pattern matching algorithms that are linear in time but require only constant extra space. The first time-space optimal pattern matching algorithm is from Galil and Seiferas [11]. Rytter [19] presented a constant-space, yet linear-time version of the Knuth-Morris-Pratt algorithm . Space-efficient real-time searching is discussed by Gasieniec and Kolpakov [12].

Concurrently searching for a set of patterns within limited working space presents a greater challenge than searching for a single pattern in small space. Recent work has begun to address 1-dimensional dictionary matching in small space [7, 14, 15, 4]. H_k, kth order empirical entropy of a string, describes the minimum number of bits that are needed to encode the string within context. Empirical entropy reveals that storage space meets the information-theoretic lower bounds of data. In the analysis of algorithms it is standard to use $O(n)$ to denote a function whose rate of growth is proportional to n; we follow this convention.

The first linear-time algorithm for dictionary matching, devised by Aho and Corasick, requires an automaton that occupies $O(\ell)$ words, or $O(\ell \log \ell)$ bits. The first solution to beat this space complexity was presented by Chan et al. [7]. They reduce the size of the index by a $\log$ factor to $O(\ell)$ bits and slow the linear time complexity by $log^2 \ell$.

The first succinct dictionary matching algorithm with no slowdown was introduced by Belazzougui [4]. His algorithm mimics the Aho-Corasick automaton within smaller space. This new approach encodes the three kinds of transitions of the AC trie separately and in different ways. Hon et al. extend this approach to encode the forward transitions of the AC automaton more efficiently without slowing down the procedure [13]. With this new representation, the space meets optimal compression of the dictionary (kth order empirical entropy) and runtime is linear.

The space-efficient algorithms described thus far have not been implemented, and their implementation will be a complex task. Several software packages have been created to perform efficient 1D dictionary matching in small space. Zha and Sahni recently described and implemented a compressed version of the Aho-Corasick automaton [21]. This implementation does not meet optimal space bounds, but offers a practical space saving over the regular Aho-Corasick automaton. Fredriksson [10] designed and implemented dictionary matching in small space with optimal *average time* complexity. However, this approach is not efficient for all types of data. In particular, when the alphabet is small, as in DNA sequences, regular Aho-Corasick is preferred. Furthermore, the algorithm is not online in the sense that it cannot process a text that arrives character by character in real-time.

This work will close the gap that exists between the theoretical algorithms and the practical implementations. We have designed and almost fully coded a practical and implementable algorithm that meets information-theoretic space bounds, is almost linear in time time, and is truly online.

2.2 Dictionary Matching in Two-Dimensional Data

This work is the first to address 2-dimensional dictionary matching in small space. In this section we review recent results in 2D single pattern matching and 2D dictionary matching.

The first linear-time 2D pattern matching algorithm was developed independently by Bird [5] and by Baker [3]. 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. Baker points out that this algorithm solves dictionary matching if 1D dictionary matching is performed on the linearized data. That is, the KMP automaton [16] can be replaced by an AC automaton [1]. Since an AC automaton is formed of the pattern rows, this method requires working space proportional to the total size of the dictionary. We summarize the Bird/Baker algorithm since our 2D algorithm is partially based upon these ideas.

Bird / Baker algorithm

1. Preprocess Pattern: Form Aho-Corasick automaton of pattern rows.

   Name pattern rows and construct Knuth-Morris-Pratt automaton of 1D pattern.

2. Row Matching: Run Aho-Corasick automaton on each text row to label positions at which a pattern row ends.

3. Column Matching: Run Knuth-Morris-Pratt algorithm on named text columns. Output pattern occurrences.

An approach for small-space, yet linear-time *single* pattern matching in 2D was developed by Crochemore et al. [8]. Their algorithm preprocesses a pattern in time proportional to its length within logarithmic space. Then, a text is searched in time proportional to its length and only constant extra space. Such an algorithm can be trivially extended to perform dictionary matching but its runtime would depend on the number of patterns in the dictionary.

The 2D dictionary matching algorithm of Amir and Farach converts the patterns to a 1D representation by considering subrow/subcolumn pairs around the diagonals [2]. Text scanning time is proportional to $n \log d$ and the data structures occupy space proportional to the total dictionary size.

None of the existing approaches to 2D dictionary matching are easily adapted to a space-constrained environment. We propose to develop an efficient algorithm for 2D dictionary matching whose space requirements meet the information-theoretic bounds of the dictionary.

3 Project Design

In this project we plan to develop and implement time and space efficient dictionary matching algorithms. Four steps will be taken, each of which is described in the ensuing subsections.

1. Write a dictionary matching program based on the generalized suffix tree.

2. Combine the implementation with the compressed suffix tree to perform 1D dictionary matching in small space.

3. Develop the first succinct 2D dictionary matching algorithm.

4. Evaluation: compare time and space usage of our approach to the state of the art methods.

3.1 Dictionary Matching on Generalized Suffix Tree

Our first goal is to obtain a fast, space-efficient program for dictionary matching in 1-dimension. 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 currently being improved and maintained by the Compact Data Structures Library development team at the Libcds website.

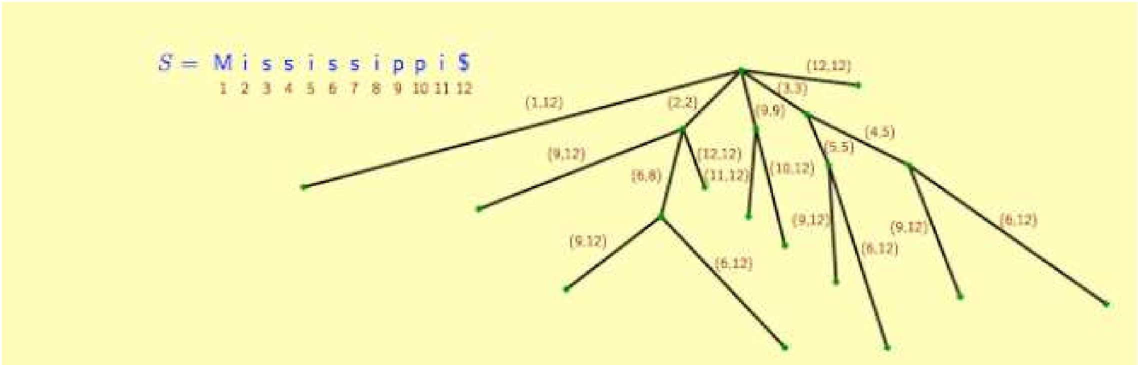

Figure 1: Suffix Tree

The suffix tree is a compact trie that represents all suffixes of an input string. The suffix tree for $S = s_1 s_2 \cdots s_n$ 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 $S[i \dots n]$. As an example, the suffix tree for Mississippi is shown in Figure 1. 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.

Ukkonen gives an algorithm for suffix tree construction that is both linear time and online [20]. 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. 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.

In our code, we follow the specific conventions that ensure linear time complexity and correctness. To guarantee that each suffix ends at a leaf and cannot be a substring of another suffix, a special character, $\$ \notin \Sigma$, is appended to the string before construction of the suffix tree. An edge is not labeled by characters as the size of this substring is not predetermined. Instead, an edge is labeled by the start and end positions of the substring it represents.

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. That approach wastes a significant amount of space 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. It is built incrementally, in an online fashion, inserting one string at a time.

We have implemented *dictionary matching* on a text over the generalized suffix tree of patterns. This algorithm mimics Ukkonen's process for inserting a new string into a generalized suffix tree, 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 at each encounter of a labeled leaf. The suffix link of a node allows rapid navigation to a distant part of the tree. At a position of mismatch, suffix links are used to navigate the tree, eliminating redundant character comparisons. Since the tree is not modified, there are several details that need attention. For example, at each level, null links must be bypassed. Also, we may attempt to examine nodes that do not exist, since we do not index the text.

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 will require very little additional space. This tool is itself a contribution, allowing efficient dictionary matching in small space, however, in the next section we discuss how we will further improve this application.

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 they are 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. As our data sets

increase in size, it becomes increasingly advantageous to discover more applications of compressed self-indexes since they occupy roughly the same space as the optimal compression of the underlying data.

We will combine our efficient dictionary matching algorithm with a compressed suffix tree representation to obtain an even more space-efficient dictionary matching program. The suffix tree for a string of length ℓ, described in Section 3.1, occupies $O(\ell)$ words or $O(\ell \log \ell)$ bits of space. Several compressed suffix tree representations have been designed and implemented. Fischer et al. [9] achieve kth order empirical entropy in their compressed representation of the suffix tree, with sub-logarithmic slowdown in traversal operations. This data structure has been implemented and evaluated by Cánovas and Navarro [6] and their code is readily available.

3.3 Succinct 2D Dictionary Matching

We have developed 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 form a compressed self-index of the patterns, after which the original dictionary may be discarded. We 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 $O(dm \log dm)$ bits. The time complexity of our new algorithm is linear.

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. As is common in many space-constrained algorithms, we divide the text into small blocks, and work with one block of text at a time. We limit our algorithm to $O(dm \log dm)$ bits of working space to process the text and search for occurrences of all patterns simultaneously.

We 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.

In [18] we outlined an algorithm that accepts 2D Lempel-Ziv compressed input and can handle patterns that fit Case I. We have now extended our focus to include small-space 2D dictionary matching of all 2D data. We generalize our approach in Case I and deal with patterns of the second type, Case II.

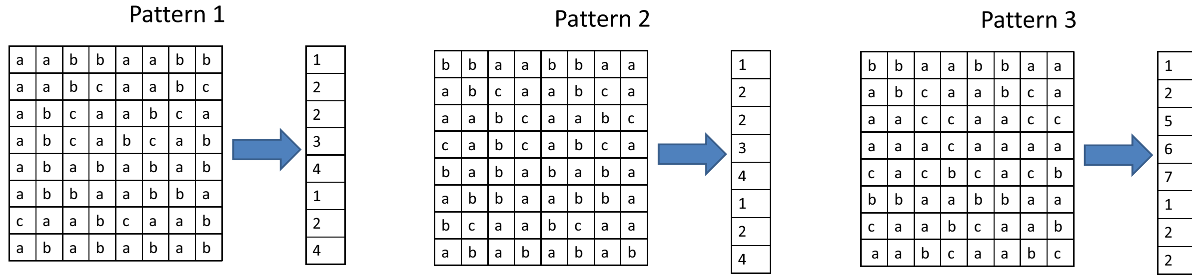

Figure 2: Three 2D patterns converted to 1D representations with the naming scheme we introduce. Patterns 1 and 2 are different, yet their 1D representations are the same.

For Case I, we generalize the naming technique used by Bird [5] and Baker [3] (Section 2.2) 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. Three 2D patterns and their 1D representations are shown in Figure 2.

In the text scanning phase, we name the rows of the text to form a 1D representation of the 2D text. Then, we use 1D dictionary matching, e.g., [1], to mark each instance of a 1D pattern as a candidate for a 2D pattern occurrence. Since similar pattern rows are grouped together, we need a verification stage to determine if the candidates are actual pattern occurrences. We allow only one pass over the text to preserve the independence between text processing time and the dictionary size. To this end, we have identified characteristics of *consistent* patterns, i.e., patterns that can overlap in the same text row. Figure 3 depicts a pair of consistent patterns.

In Case II, when some pattern row is not highly periodic, many completely different patterns can overlap closely in the text. As a result, it is difficult to employ a naming scheme to find all occurrences of patterns. However, we can filter the text to identify a limited number of potential pattern occurrences. This preliminary step is straightforward as we can employ the first aperiodic row (or row with period $> m/4$) of each pattern for this purpose. However, verification of these candidates in one pass over the text presents a hurdle. We propose two methods to overcome this difficulty and verify the candidates in one pass. One method uses the compressed suffix tree (Section 3.2) that occupies space proportional to the optimal compression of the dictionary. While this method employs a data structures that has many other uses, it slows down verification by a sublogarithmic factor. Alternatively, we can use the novel small-space 1D dictionary matching algorithm with no slowdown [13] in the verification stage. Employing this new idea as a black box gives us the first small-space 2D dictionary matching algorithm with no slowdown.

b b a a b b a a b b
a b c a a b c a a b
a a b c a a b c a a
c a b c a b c a c a
b a b a b a b a b a
a b b a a b b a a b
b c a a b c a a b c
a b a b a b a b a b

Figure 3: Horizontally consistent patterns have overlapping columns: one is a horizontal cyclic shift of the other. The first is Pattern 2 of Figure 2, the other is new.

3.4 Evaluation

We plan to assess the effectiveness of our software against space-efficient 1D dictionary matching software. Specifically, we will compare our approach to those of Zha and Sahni [21] and Fredriksson [10]. Zha and Sahni describe and provide an efficient implementation of the Aho-Corasick automaton. Fredriksson achieves dictionary matching in small space and expected linear time.

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

3.5 Summary

In summary, this project proposes to tackle the problem of performing dictionary matching in one and two dimensions with **small space**. In one-dimension, the theoretical ideas for performing dictionary matching in small space have mostly been developed. The major challenge lies in the algorithmic engineering of combining the succinct data structures with the generalized suffix tree. In two-dimensions, we are developing a new algorithm to perform dictionary matching in small space. The work is almost complete and is currently being prepared for publication. These results have potential benefits for a wide audience including biologists, computer scientists, and the software industry. It is anticipated that the PI will complete her dissertation work in the next academic year.

4 Timeline

Spring 2011	Finalize and publish our succinct 2D dictionary algorithm.
Summer 2011	Complete implementation of 1D dictionary matching over the compressed suffix tree.
Fall 2011	Publish experimental results on 1D data and make them accessible on the World Wide Web.
Spring 2012	Write and defend dissertation.

References

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

[2] A. Amir and M. Farach. Two-dimensional dictionary matching. *Inf. Process. Lett.*, 44(5):233–239, 1992.

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

[4] D. Belazzougui. Succinct dictionary matching with no slowdown. In *CPM*, pages 88–100, 2010.

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

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

[7] H. L. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[8] 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.

[9] J. Fischer, V. Mäkinen, and G. Navarro. Faster entropy-bounded compressed suffix trees. *Theor. Comput. Sci.*, 410(51):5354–5364, 2009.

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

[11] 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.

[12] L. Gasieniec and R. M. Kolpakov. Real-time string matching in sublinear space. In *CPM*, pages 117–129, 2004.

[13] 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.

[14] 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.

[15] 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.

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

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

[18] S. Neuburger and D. Sokol. Small-space 2d compressed dictionary matching. In *CPM: 21st Symposium on Combinatorial Pattern Matching*, pages 27–39, 2010.

[19] W. Rytter. On maximal suffixes and constant-space linear-time versions of kmp algorithm. *Theor. Comput. Sci.*, 299(1-3):763–774, 2003.

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

[21] X. Zha and S. Sahni. Highly compressed aho-corasick automata for efficient intrusion detection. In *ISCC*, pages 298–303, 2008.