Algorithmica (2013) 65:662–684
DOI 10.1007/s00453-012-9615-9

Succinct 2D Dictionary Matching

Shoshana Neuburger · Dina Sokol

Received: 23 July 2010 / Accepted: 21 January 2012 / Published online: 2 February 2012

Abstract The dictionary matching problem seeks all locations in a given text that match any of the patterns in a given dictionary. Efficient algorithms for dictionary matching scan the text once, searching for all patterns simultaneously. Existing algorithms that solve the 2-dimensional dictionary matching problem all require working space proportional to the size of the dictionary.

This paper presents the first efficient 2-dimensional 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 finds all occurrences of P_i, $1 \leq i \leq d$, in T. The preprocessing of the dictionary forms a compressed self-index of the patterns, after which the original dictionary may be discarded. Our algorithm uses $O(dm \log dm)$ extra bits of space. The time complexity of our algorithm is close to linear, $O(dm^2 + n^2 \tau \log \sigma)$, where τ is the time it takes to access a character in the compressed self-index and σ is the size of the alphabet. Using recent results τ is at most sub-logarithmic.

Keywords Dictionary matching · Two-dimensional · Small space algorithm · Compressed self-index

S. Neuburger
Department of Computer Science, The Graduate Center of the City University of New York, 365 Fifth Avenue, New York, NY 10016, USA
e-mail: shoshana@sci.brooklyn.cuny.edu

D. Sokol (✉)
Department of Computer and Information Science, Brooklyn College of the City University of New York, 2900 Bedford Avenue, Brooklyn, NY 11210, USA
e-mail: sokol@sci.brooklyn.cuny.edu

1 Introduction

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, without a significant sacrifice in their query time. This research has extended to dynamic ordered trees, suffix trees, and suffix arrays, among other data structures. The *dictionary matching problem* is that of searching a text for all occurrences of any one of a set of patterns that occur in the text. Preferably, an algorithm scans the text once so that its running time depends only on the size of the text and not on the size of the patterns sought. 1-dimensional dictionary matching in small space has received a lot of attention in recent work [6, 9, 15, 19, 20]. This article addresses 2-dimensional dictionary matching in small space.

The focus of this paper is a problem of practical significance. Image identification software identifies smaller images in a large image based on a dictionary of previously identified images. This is a direct application of 2D dictionary matching. The time complexity of an efficient algorithm should not depend on the size of the database of known images. In many devices, such as mobile devices, additional storage space is limited. For this reason, we address small-space dictionary matching for 2D data.

Table 1 includes recent results for small-space 1D dictionary matching; in the following paragraphs we summarize these results. The empirical entropy of a string is the average number of bits per symbol needed to encode the string. 0th order or kth order empirical entropy (H_0 or H_k) are often used as a measure of space, as seen in Table 1. For completeness, we include precise formulas for H_0 and H_k in Appendix.

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 $O(n \log \sigma + occ)$ time [1]. Hashing techniques can achieve $O(n + occ)$ 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. [9]. They reduced the size of the dictionary index from $O(\ell \log \ell)$ 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 [19]. 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 [6]. His algorithm mimics the Aho-Corasick automaton within smaller space, requiring only $\ell(H_0(D) + O(1)) + O(d \log(\ell/d))$ bits. For simplicity we report the complexities in terms of the dictionary size ℓ, although several of the re-

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

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. [9]
$\ell H_k(D) + o(\ell \log \sigma) + O(d \log \ell)$	$O(n(\log^{\epsilon} \ell + \log d) + occ)$	Hon et al. [19]
$\ell(H_0(D) + O(1)) + O(d \log(\ell/d))$	$O(n + occ)$	Belazzougui [6]

sults can be stated in terms of s, the number of states in the Aho-Corasick automaton, $s \leq \ell$.

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 working with 2D patterns, we generally need space above the input space. Thus, when analyzing the space needed by the 2D algorithms, we distinguish between the space used by the data structures that *replace* the actual patterns, and the *extra space* that is needed above the input.

Since there are no known small-space 2D dictionary matching algorithms, we mention the existing results for small space 2D *single* pattern matching. Crochemore et al. [11] perform 2D pattern matching in linear time, using $O(\log m)$ extra space to preprocess a pattern of size m^2 and $O(1)$ extra space to scan the text. Such an algorithm can be trivially extended to perform dictionary matching but would require $O(dn)$ time to process the text, dependent on the number of patterns in the dictionary.

The linear-time 2D pattern matching algorithm developed independently by Bird [8] and Baker [5] (BB) extends easily to dictionary matching. They translate the 2D pattern matching problem into a 1D pattern matching problem. Rows of the characters are perceived as metacharacters and named with the help of an Aho-Corasick (AC) automaton [1]. The text is named in a similar fashion and a Knuth-Morris-Pratt (KMP) automaton [22] of the pattern rows is used on the text columns to identify occurrences of the pattern. Baker points out that the KMP automaton can be replaced by an AC automaton to solve dictionary matching. The AC automaton of the pattern rows, combined with the AC automaton of the 1D patterns of names, can replace the input. However, since this structure is larger than the original patterns, the 1D patterns of names must be considered extra space. In addition, the BB algorithm labels each position of the text, hence, the extra space is proportional to the size of the text.

The 2D dictionary matching algorithm of Amir and Farach [3] converts the patterns to a 1D representation by considering subrow/subcolumn pairs around the diagonals. Their method constructs a suffix tree of the text and the patterns, and an AC automaton of the 1D representation of patterns. Text scanning time is $O(n \log d)$, and the extra space used is again proportional to the size of the text plus the patterns of names. The algorithm of Amir and Farach works only with a dictionary of *square* patterns. Idury and Schaffer [21] developed an algorithm for dictionary matching in rectangular patterns, where the lengths and heights can be of different sizes. Their

algorithm requires working space proportional to the dictionary size, and has a slight slowdown in the time for text processing.

In this paper we present the first algorithm that solves the *Small-Space 2D Dictionary Matching Problem*. Given a dictionary of patterns, $P_1, P_2, \ldots, P_d$, each of size $m \times m$, and a text of size $n \times n$, we find all occurrences of patterns in the text. We discuss patterns that are all of size $m \times m$ for ease of exposition, but as with Bird/Baker, our algorithm generalizes to patterns that are the same size in only one dimension with the complexity dependent on the size of the largest dimension.

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

2 Overview

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

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

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. String S is *periodic* in u if $S = u'u^j$ where u' is a suffix of u, u is primitive, and $j \geq 2$. A periodic string p can be expressed as $u'u^j$ for one unique primitive u. We refer to u as "the period" of p. Depending on the context, u can refer to either the string u or the period size $|u|$.

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

A pattern can have only periodic rows with all periods $\leq m/4$ (Case I) or have at least one aperiodic row or a row with a period $> m/4$ (Case II). We began working on Case I in [26] and expand on those results in Sect. 3. Case II is addressed in Sect. 4.

In the case that $d \geq m$, i.e., when $dm = \Omega(m^2)$, we have more space to work with as the text is processed. We can store $O(m^2)$ information for a text block and present a different algorithm for that case in Sect. 4.3.

We assume the standard RAM with word-size $\Theta(\log \ell)$ bits as our computational model, where ℓ is the input size of our problem. In this model, standard arithmetic or bitwise boolean operations on word-sized operands, and reading or writing $O(\log \ell)$ consecutively stored bits, can each be performed in constant time.

3 Case I: Patterns with Rows of Period Size $\leq m/4$

We store the linearized dictionary in an entropy compressed form that allows constant time random access to any character in the original data, such as the compression scheme of Ferragina and Venturini [12] or of Fredriksson and Nikitin [16]. For Case I patterns we do not need additional functionality in the self-index, thus we do not construct a compressed suffix tree or suffix array. The space needed for storing the dictionary D in entropy-compressed form is $\ell H_k(D) + \gamma$ where γ is the low-order term,[1] and depends on the particular compression scheme that is employed.

We overcome the extra space requirement of traditional 2D dictionary matching algorithms with an innovative preprocessing scheme that names 2D patterns to represent them in 1D. The pattern rows are initially classified into groups, with each group having a single representative. We store a *witness*, or position of mismatch, between the group representatives. A 2D pattern is named by the group representative for each of its rows. This is a generalization of the naming technique used by Bird [8] and Baker [5] to name 2D data in 1D. The preprocessing is performed in a single pass over the patterns. Constant amount of information is stored per pattern row, occupying a total of $O(dm \log dm)$ bits of space. Details of the preprocessing stage can be found in Sect. 3.1.

In the text scanning phase, we name the rows of the text to form a 1D representation of the 2D text. Then, we use an Aho-Corasick (AC) automaton [1] to mark candidates of possible pattern occurrences in the 1D text in $O(n^2 \log \sigma)$ time. In this section, σ can be viewed as the size of the alphabet of names if it is smaller than the original alphabet; $\sigma \leq dm$. Since similar pattern rows are grouped together, we need a verification stage to determine if the candidates are actual pattern occurrences. With additional preprocessing of the 1D pattern representations, a single pass suffices to verify potential pattern occurrences in the text. The details of the text scanning stage are described in Sect. 3.2.

[1] For example, using Ferragina and Venturini [12], $\gamma = O(\frac{\ell}{\log_\sigma \ell}(k \log \sigma + \log\log \ell))$.

3.1 Pattern Preprocessing

Definition 1 ([10]) A 2D $m \times m$ pattern is *h-periodic*, or horizontally periodic, if two copies of the pattern can be aligned in the top row so that there is no mismatch in the region of overlap and the number of overlapping columns is $\geq m/2$.

Observation 1 *If a 2D pattern is h-periodic then each of its rows is periodic.*

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

We introduce two new data structures, the witness tree and the offset tree. The witness tree facilitates the linear-time preprocessing of pattern rows. It is described in Sect. 3.1.2. The offset tree allows the text scanning stage to achieve linear time complexity, independent of the number of patterns in the dictionary. It is described in Sect. 3.2.1.

3.1.1 Lyndon Word Naming

Definition 2 Two words x, y are *conjugate* if $x = uv$, $y = vu$ for some words u, v [23].

Definition 3 A *Lyndon word* is a primitive string which is lexicographically smaller than any of its conjugates [23].

Since conjugacy is an equivalence relation, we can partition the pattern rows into disjoint groups based on the conjugacy of their periods. We use the same name to represent all rows whose periods are conjugate. The smallest conjugate of a word, i.e. its Lyndon word, is the standard representation of its conjugacy class. *Canonization* is the process of computing a Lyndon word, and can be done in linear time and space [23]. We name one pattern row at a time by finding its period and canonizing. If a new Lyndon word or a new period *size* is encountered, the row is given a new name. Otherwise, the row adopts the name already given to another member of its conjugacy class. Each 2D pattern obtains a 1D representation of names in a similar manner to the BB algorithm, but using Lyndon word naming. The extra space needed to store the 1D patterns of names is $O(dm \log dm)$ bits.

Three 2D patterns and their 1D representations are shown in Fig. 1. To understand the naming process we will look at Pattern 1. The period of the first row is *aabb*, which is four characters long. It is given the name *1*. When the second row is examined, its period is found to be *aabc*, which is also four characters long. *aabb* and *aabc* are both Lyndon words of size four, but they are different, so the second row

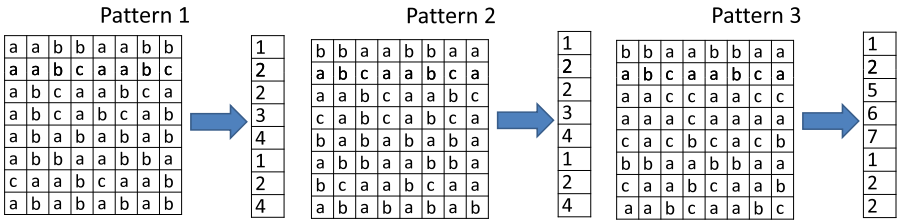


Fig. 1 Three 2D patterns with their 1D representations. We use these patterns to illustrate the concepts, although their periods are larger than $m/4$. Patterns 1 and 2 are different, yet their 1D representations are the same

is named *2*. The period of the third row is *abca*, which is represented by the Lyndon word *aabc*. Thus, the second and third rows are given the same name even though they are not identical.

When naming a pattern row, its period is identified using known techniques in linear time and space, i.e., using a KMP automaton [22] of the string. Then, we compute and store several discrete pieces of information per row: period size (in $\log m/4$ bits), name (in $\log dm$ bits), and position of the first Lyndon word occurrence in the period, which we call *LYpos* (in $\log m/4$ bits).

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

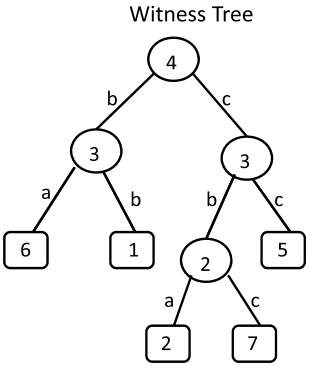
3.1.2 Witness Tree

Components of witness tree:

Internal node: position of a character mismatch. The position is an integer $\in [1, m]$.
Edge: labeled with a character in the alphabet. Two edges emanating from a node must have different labels.
Leaf: an equivalence class representing one or more pattern rows.

When a new row is examined, we need to determine if the Lyndon word of its period has already been named. The witness tree allows us to identify the only named string of the same size that has no recorded position of mismatch with the new string. Then, the found string is compared sequentially to the new row. A witness tree for Lyndon words of length four is depicted in Fig. 2.

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



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

Fig. 2 A witness tree for the Lyndon words of length 4 that are in the table of names

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

Lemma 1 *Of the named strings that are the same size as a new string, i, there is at most one equivalence class, j, that has no recorded mismatch against i.*

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

Lemma 2 *The witness trees for the rows of d patterns, each of size $m \times m$, occupies $O(dm \log dm)$ bits of space.*

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

(i) A leaf is replaced with an internal node to represent the position of mismatch. The new internal node has two leaves as its children. One leaf represents the new

name, and the other represents the string to which it was compared. The new edges are labeled with the characters that mismatched.

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

Corollary 1 *The witness tree for Lyndon words of length u has depth $\leq u$.*

Lemma 3 *A pattern row of size $O(m)$ is named in $O(m)$ time using the appropriate witness tree.*

Proof By Lemma 1, a new string is compared to at most one other string, j. A witness tree is traversed from the root to identify j. Traversal of a witness tree ceases either at an internal node or at a leaf. The time spent traversing a tree is bounded by its depth. By Corollary 1, the tree-depth is $O(m)$, so the tree is traversed in $O(m)$ comparisons. Thus, a new string is classified with $O(m)$ comparisons. □

3.1.3 Preprocessing the 1D Patterns

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

The next preprocessing step computes the Least Common Multiple (LCM) of each distinct 1D pattern. This can be done incrementally, one row at a time, in time proportional to the number of pattern rows. The LCM of an h-periodic pattern reveals the horizontal distance between its potential occurrences in a text block. This conserves space as there are fewer candidates to maintain. In addition, we use this to conserve verification time. The LCM of the 1D patterns can be stored in $d \log m$ bits of space since we are only interested in an LCM that is $\leq m$, i.e., the LCM of a pattern that can overlap itself in a text block.

If several patterns share a 1D name, an *offset tree* is constructed of the Lyndon word positions in these patterns. We defer the description of the offset tree to Sect. 3.2.1 where it is used in the verification phase.
In summary, pattern preprocessing in $O(dm^2)$ time and $O(dm \log dm)$ bits of space:

1. For each pattern row,
   (a) compute period and canonize
   (b) store period size, name, first Lyndon word occurrence (*LYpos*).
2. Construct AC automaton of 1D patterns.
3. Find LCM of each 1D pattern.
4. For multiple patterns of same 1D name, build an offset tree.
5. Compress dictionary. Can discard original dictionary.

3.2 Text Scanning

The text scanning stage has three steps.

1. Name rows of text.
2. Identify candidates with a 1D dictionary matching algorithm, e.g. AC.
3. Verify candidates separately for each text row using the offset tree of the 1D patterns.

Step 1 *Name Text Rows*

We search a 2D text for a 1D dictionary of patterns using a 1D Aho-Corasick (AC) automaton. A 1D pattern can begin at any of the first $m/2$ positions of a text block row. The AC automaton can branch to one of several characters; we can't afford the time or space to search for each of them in the text block row. Thus, we name the rows of a text block before searching for patterns. The divide-and-conquer algorithm of Main and Lorentz finds all maximal repetitions in linear time, searching for repetitions to the right and to the left of the midpoint of a string [24]. Repetitions of length $\geq m$ that cross the midpoint and have a period size $\leq m/4$ are the only ones that are of interest to our algorithm.

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

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

After finding the only maximal periodic substring of length $\geq m$ with period $\leq m/4$, the text block rows are named in much the same way as the pattern rows are named. The period of this maximal run is found and canonized. Then, the appropriate witness tree is used to name the text block row. We use the witness tree constructed during pattern preprocessing since we are only interested in identifying text block rows that correspond to Lyndon words found in the pattern rows. At most one pattern row will be examined to classify the conjugacy class of a text block row. In addition to the name, period size, and *LYpos*, we maintain a *left* and a *right* pointer for each row of a text block. *left* and *right* mark the endpoints of the periodic substring in the text. The *LYpos* (position of first Lyndon word occurrence) is computed relative to the *left* pointer of the row. This process is repeated for each row, and $O(m)$ information is obtained for the text block.

Complexity of Step 1 The largest periodic substring of a row of width $3m/2$, if it exists, can be found in $O(m)$ time and space [24]. Its period can be found and canonized in linear time and space [23]. The row is named in $O(m)$ time and space using the appropriate witness tree (Lemma 3). Overall, $O(m^2)$ time and $O(m \log dm)$ bits of space are needed to name the rows of a text block.

Step 2 *Identify Candidates*

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

Complexity of Step 2 1D dictionary matching in a string of size m can be done in $O(m \log \sigma)$ time with $O(dm \log dm)$ bits of space for the AC automaton. The AC automaton can be represented in even smaller space for some types of data using the approach of [6].

Step 3 *Verify Candidates*

The occurrence of a 1D pattern is not sufficient evidence that a 2D pattern actually occurs. The verification process considers each candidate row of the text separately. Several patterns can share a 1D representation. We need to verify the overall width of the 1D names, as well as the alignment of the periods among rows.

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

The verification stage must also ascertain that the Lyndon word positions in the text align with the Lyndon word positions in the pattern rows. Naively, this can be done in $O(m^3)$ time. We verify a candidate row in $O(m)$ time using the offset tree of a 1D pattern.

We say that two distinct patterns are *horizontally consistent* if it is possible for the patterns to occur in the same text block row. Note that simply having the same 1D representation does not render candidates consistent, although it is a necessary condition. Horizontally consistent patterns can be obtained from one another by performing a horizontal cyclic permutation of the characters, i.e., by moving several columns to the opposite end of the matrix. Figure 3 depicts a pair of horizontally consistent patterns. Horizontal consistency is an equivalence relation just as conjugacy is. We can form equivalence classes of patterns with the same 1D name and then classify the text as belonging to at most one group. We choose a representative for each equivalence class. The class representative is the shift in which the Lyndon word of the first row begins at the first position.

Each row of a 2D pattern array is represented by the 1D array containing its 1D row names and the 1D array of *LYpos* entries. To convert a pattern to one that is horizontally consistent with it, its rows are shifted by the same constant, but the *LYpos* of its rows may not be. However, the shift is the same across the rows, relative to the period size of each row. Figure 3 shows an example of horizontally consistent patterns and the relative shifts of their rows. Notice that (c) can be obtained from (b)

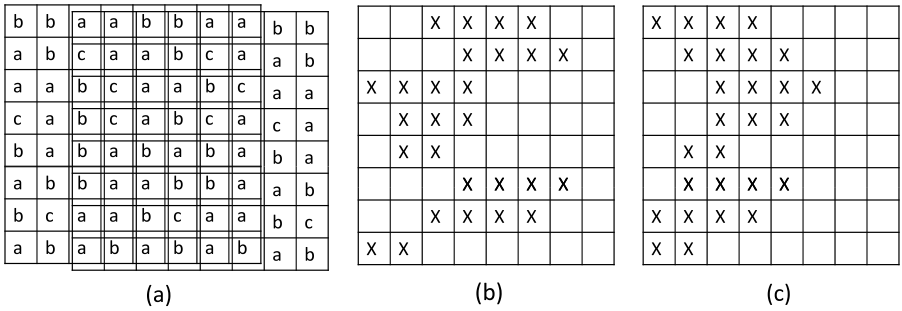


Fig. 3 (**a**) Horizontally consistent patterns have overlapping columns: one is a horizontal cyclic shift of the other. The first is Pattern 2 of Fig. 1, the other is new. (**b**) and (**c**) are horizontally consistent patterns. The first Lyndon word occurrence on each row is represented by X. (**c**) Is the representative of this consistency class

by shifting two columns towards the left. The first occurrence of the Lyndon word of the first row is at position 3 in (b) and at position 1 in (c). This shift seems to reverse in the third row, since the Lyndon word first occurs at position 1 in (b) and at position 3 in (c). However, the relative shift remains the same, since the shift is cyclic. We summarize this relationship in the following lemma.

Lemma 5 *Two patterns with the same* 1*D representation are* horizontally consistent *iff the* LYPos *of all their rows are shifted by* C mod period size of the row, *where C is a constant*.

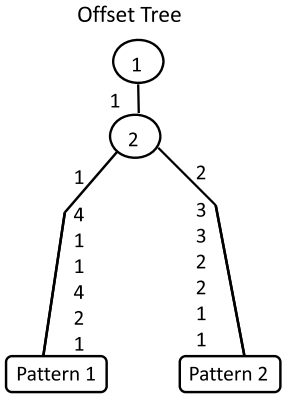
Proof Let patterns P_i and P_j be horizontally consistent. Then, their corresponding rows are cyclic permutations. Matrix P_i is obtained from P_j by shifting C columns from the beginning to the end of P_j. The *LYpos* of a row is between 1 and the period size of a row. On a row with period size u, a shift of C columns translates to a shift of C mod u. Similarly, if we know that the shift of each row is C mod u, the 2D patterns must be horizontally consistent. □

3.2.1 Offset Tree

In the preprocessing stage, we construct an offset tree to align the shifted *LYpos* arrays of patterns with the same 1D name so that the text can be classified, and ultimately verified, in $O(m)$ time. This allows the text scanning stage to complete in time proportional to the text size, independent of the dictionary size. An offset tree is shown in Fig. 4.
Components of offset tree:

Root: represents the first row of a pattern.
Internal node: represents a row index from 1 to m, strictly larger than its parent's.
Edge: labeled by shifted *LYpos* entries.
Leaf: represents a consistency class of dictionary patterns.

We construct an offset tree for each set of patterns that were named with the same 1D representation. One pattern at a time, we traverse the tree and compare the shifted



Name	Period size	Pattern1 LYpos	Pattern2 LYpos
1	4	1	3
2	4	1	4
2	4	4	1
3	3	1	2
4	2	1	2
1	4	4	4
2	4	2	3
4	2	1	1

Fig. 4 Offset tree for Pattern 1 and Pattern 2 (the first two patterns of Fig. 1) which have the same 1D name. The *LYpos* entries of Pattern 1 are not shifted since its first entry is a 1, while the *Lypos* entries of Pattern 2 are shifted by 2 mod period size of row, to match the *LYpos* array of its group representative shown in Fig. 3(c)

LYpos arrays in sequential order until either a mismatch is found or we reach a leaf. If a mismatch occurs at an edge leading to a leaf, a new internal node with a leaf are created, to represent the position of mismatch and the new consistency class, respectively. If a mismatch occurs at an edge leading to an internal node, a new branch is created with a new leaf to represent the new consistency class.

Lemma 6 *The consistency class of a string of length m is found in $O(m)$ time.*

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

We modify the *LYpos* array of the text to reflect the first Lyndon word occurrence in each text block row after *maxLeft*. Each modified *LYpos* entry is $\geq$ *maxLeft* and can be computed in $O(1)$ time with basic arithmetic.

We shift the *LYpos* values of the text so that the Lyndon word of the first row occurs at the first position. We traverse the offset tree to determine which pattern(s), if any, are horizontally consistent with the text. If traversal ceases at a leaf, then its pattern(s) can occur in the text, provided the text is sufficiently wide.

At this point, we know which patterns are horizontally consistent with the text block row. The last step is to locate the positions at which a pattern begins, within the row. We need to reverse the shift of the horizontally consistent patterns. This is done for each pattern that is horizontally consistent with the text block by looking up the *LYpos* of the pattern's first row. Then, we verify that the periodic substrings of the text are sufficiently wide. That is, we announce position i as a pattern occurrence

when $minRight - i \geq m$. Subsequent pattern occurrences in the same row are at LCM multiples of the pattern.

Observation 2 *The offset trees for d 1D patterns, each of size m, have $O(d)$ nodes and thus can be stored in $O(d \log d)$ bits of space.*

Complexity of Step 3 $O(m)$ rows in a text block can contain candidates. For each candidate row, *maxLeft* and *minRight* are computed, and the *LYpos* array is shifted. This is all done in $O(m)$ time for the m rows that a pattern can span. Then, the offset tree is traversed with $O(m)$ comparisons. Finally, the actual occurrences of a pattern are determined in $O(m)$ time. Overall, a text block is verified in $O(m^2)$ time, proportional to the size of a text block. The verification process requires $O(m \log dm)$ extra bits of space.

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

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

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

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

We allow $O(dm \log dm)$ bits of space to process a block of text. In the event that $d < m$, Case IIa, this limit on space is a significant constraint. We address this case in Sect. 4.2. When $d \geq m$, Case IIb, the number of candidates for pattern occurrences can exceed the size of a text block. It is difficult to verify such a large number of candidates in time proportional to the size of a text block. Because we allow working space larger than the size of a text block, there is no need to begin by filtering the text and identifying a limited set of candidate positions. We present a different algorithm to handle this case in Sect. 4.3.

4.1 Compressed Suffix Trees

For Case II patterns, we again *linearize* the dictionary by concatenating the rows of all patterns, inserting a delimiter at the end of each row. We then replace the original dictionary by storing an entropy-compressed self-index of the linearized dictionary. For Case IIa, a compressed suffix array (CSA) and compressed LCP array encapsulate

sufficient information for our dictionary matching algorithm. However, in Case IIb, we need the ability to traverse the compressed suffix tree. For consistency, we discuss the usage of a compressed suffix tree in both cases.

Russo et al. [27] achieved fully-compressed suffix trees requiring $\ell H_k + o(\ell \log \sigma)$ 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, the time complexities of all operations are $O(\log \ell)$.

The fully-compressed suffix tree presented by Fischer et al. needs $2H_k(2 \log \frac{1}{H_k} + \frac{1}{\epsilon} + O(1)) + o(\ell)$ bits of space [14]. It accommodates almost all navigational and retrieval operations in sub-logarithmic time. It is based on a compressed suffix array [17], a compressed LCP array, and data structures for range minimum and previous/next smaller value queries. 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 $O(\log^{\epsilon} \ell)$, $0 < \epsilon \leq 1$.

With Fischer's new compressed representation of the LCP array [13], the compressed suffix tree of Fischer et al. [14] can be stored in even smaller space. That is, the suffix tree can be stored in $(1 + \frac{1}{\epsilon})\ell H_k + o(\ell)$ bits of space with all operations computed in sub-logarithmic time. The time for character retrieval, locate, string depth, LCA queries, and suffix link traversal ranges from $\log^{\epsilon} \ell$ time to $\log^{\epsilon+\epsilon'} \ell \log^2 \log \ell$ time, for any constant $0 < \epsilon, \epsilon' \leq 1$.

In this paper we simply use τ to refer to the time complexity of operations in the compressed suffix tree, and we use the term *entropy-compressed* to refer to storage space that is close to $\ell H_k(D)$. The reader can either refer back to this section to see the time-space tradeoffs, or apply other results to the storage of the linearized patterns.

4.2 Case IIa: $d < m$

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

4.2.1 Pattern Preprocessing

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

The pattern rows are named to form a 1D dictionary of patterns. Distinct rows are given different names, much the same way that Bird and Baker convert a 2D pattern

to a 1D representation. However, Bird and Baker form an AC automaton of all pattern rows. We do not allow that much space. Instead, we use a witness tree, Sect. 3.1.2, to store distinctions between the pattern rows, which are all strings of length m. The witness tree of the row names is preprocessed for Lowest Common Ancestor (LCA) to provide a witness between any pair of distinct pattern rows.

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

1. Construct AC automaton of first aperiodic row of each pattern. Store row number of each of these aperiodic rows.
2. Name pattern rows using a single witness tree. Store 1D patterns of names.
3. Preprocess witness tree for LCA.
4. Construct generalized suffix tree of 1D patterns. Preprocess for LCA.

Lemma 7 *The pattern preprocessing stage completes in* $O(dm^2)$ *time and* $O(dm \log dm)$ *extra bits of space.*

Proof 1. The AC automaton of the first non-periodic row of each pattern is constructed in $O(dm)$ time and is stored in $O(dm \log dm)$ bits. (For some types of data, this can be done in less space with the new result of [6].)

2. By Lemma 2, the witness tree occupies $O(dm \log dm)$ bits of space. By Lemma 3, pattern rows are named with the help of the witness tree in $O(dm^2)$ time.

3. The suffix and witness trees are preprocessed in linear time to answer LCA queries in $O(1)$ time [7, 18].

4. The 1D dictionary of names is stored in $O(dm \log dm)$ bits of space and its generalized suffix tree is constructed and stored in time and space proportional to this 1D representation. □

4.2.2 Text Scanning

The text scanning stage has three steps.

1. Identify candidates in text block with 1D dictionary matching of a non-periodic row of each pattern.
2. Duel to eliminate vertically inconsistent candidates.
3. Verify pattern occurrences at surviving candidate positions.

Step 1 *Identify Candidates*

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

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

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

Step 2 *Eliminate Vertically Inconsistent Candidates*

We call a pair of patterns *consistent* if they can overlap in a single text block. Overlapping segments of consistent candidates can be verified simultaneously. In this stage we eliminate inconsistent candidates with a dueling technique inspired by the 2D *single* pattern matching algorithm of Amir et al. [2]. In the single pattern matching algorithm, a witness table is computed in advance, and duels are performed between candidates of the same pattern. In dictionary matching, we want to perform duels between candidates for *different* patterns. It would be inefficient both in terms of time and space to store witnesses between all locations in all patterns.

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

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

Text block columns are scanned top-down, one at a time, to determine vertical consistency of candidates. We confirm consistency pairwise over the candidates within a column, since consistency is a transitive relation. A duel eliminates at least one element of a pair of inconsistent candidates. If only the lower candidate is killed, this does not affect the consistent candidates above it in the same column, as they are still consistent with the text character. However, if the lower candidate survives, it triggers the elimination of all candidates within m rows above it. Pointers link consecutive candidates in each column. This way, a duel eliminates the set of consistent

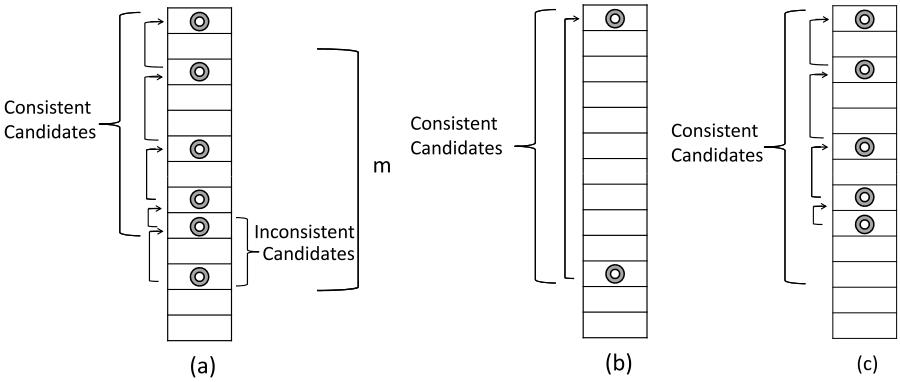


Fig. 5 (**a**) Duel between vertically inconsistent candidates in a column. (**b**) Surviving candidates if the lower candidate wins the duel. (**c**) Surviving candidates if the upper candidate wins the duel

candidates that are within range of the mismatch. This is shown in Fig. 5. Distinct patterns have different 1D representations. Thus, the same method can be used when two (or more) candidates occur at a single text position.

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

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

Step 3 *Verify Surviving Candidates*

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

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

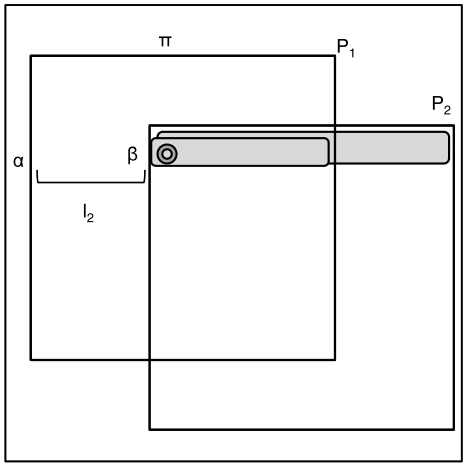


Fig. 6 Consistency is determined by LCP queries and duels

that is expected to begin. If the distance between the candidates is smaller than the LCP, a duel resolves the inconsistency among candidates.

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

Consider Fig. 6. Suppose we are at position π in row α of pattern P_1 and we approach the expected beginning of row β in pattern P_2. An LCP query on suffix π of α and the entire β determines if they can overlap at this distance. Let the LCP of these substrings be l_1 and the distance between α and β in the text be l_2. Suppose $l_1 < m - l_2$. That is, the mismatch is within the expected overlap. Then we can duel between the candidates using the text character at $\pi + l_1$ to eliminate one or both candidates. However, if $l_1 = m - l_2$, the text can be compared to a substring of either pattern row since these substrings are identical.

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

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

Lemma 8 *The algorithm for 2D dictionary matching in Case IIa, when $d < m$, completes in $O(n^2\tau \log \sigma)$ time and $O(dm \log dm)$ bits of space, in addition to the entropy compressed self-index of the linearized dictionary.*

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

4.3 Case IIb: $d \geq m$

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

We can name the text positions using the compressed suffix tree of pattern rows in much the same way as an AC automaton. With suffix links, we name the positions of the text block, row by row, according to the names of pattern rows. Beginning at the root of the tree, traverse the edge whose label matches the first character of the text block row. When m consecutive characters trace a path from the root, and traversal reaches a leaf, the position is named with the appropriate pattern row. At a mismatch, we traverse suffix links to find the longest suffix of the already matched string that matches a prefix of a pattern row and compare the next text character to that labeled edge of the tree. With suffix links, this is done in time proportional to the number of characters that have already matched a path from the root of the tree. This is done in the spirit of Ukkonen's online suffix tree construction algorithm which is linear time [28].

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

Lemma 9 *The algorithm for 2D dictionary matching in Case IIb, when $d \geq m$, completes in $O(n^2\tau \log \sigma)$ time and $O(dm \log dm)$ bits of space, in addition to the entropy compressed self-index of the linearized dictionary.*

Proof It suffices to show that the procedure completes in $O(m^2\tau \log \sigma)$ time for a text block of size $3m/2 \times 3m/2$. The algorithm names the text positions by traversing the compressed suffix tree of the dictionary in $O(m^2\tau \log \sigma)$ time and then locates occurrences of the 1D patterns of names with 1D dictionary matching in $O(m^2)$ time. Our algorithm uses an AC automaton of the dictionary of 1D pattern names and a compressed suffix tree of the linearized dictionary. $O(dm \log dm)$ bits of space suffice to store an AC automaton of the 1D patterns of names. A compressed self-index and compressed suffix tree can be stored in entropy compressed space [13]. After forming the two data structures, $O(m^2 \log dm) = O(dm \log dm)$ bits of space are used to name a text block. □

Theorem 1 *Our algorithm for 2D dictionary matching completes in* $O(dm^2 + n^2\tau \log \sigma)$ *time and* $O(dm \log dm)$ *bits of extra space.*

Proof Our algorithm is divided into several cases.

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

The complexity of the pattern preprocessing stage is summarized in Sect. 3.1 and the complexity of the text scanning stage is summarized in Sect. 3.2. Both of them meet the bounds specified by this theorem.

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

Case IIa: $d < m$. The complexity is summarized in Lemma 8.

Case IIb: $d \geq m$. The complexity is summarized in Lemma 9. □

5 Data Compression

The *compressed pattern matching problem* seeks all occurrences of a pattern in text, and works with pattern and text that are stored in compressed form. Amir et. al. presented an algorithm for strongly-inplace single pattern matching in 2D LZ78-compressed data [4]. They define an algorithm as *strongly inplace* if the extra space it uses is proportional to the optimal compression of the data. Their algorithm preprocesses the pattern of uncompressed size $m \times m$ in $O(m^3)$ time and searches a text of uncompressed size $n \times n$ in $O(n^2)$ time. Our preprocessing scheme can be applied to their algorithm to achieve an optimal $O(m^2)$ preprocessing time, resulting in an overall time complexity of $O(m^2 + n^2)$.

In the *compressed dictionary matching* problem, the input is in compressed form and one would like to search the text for all occurrences of any element of a *set* of patterns. Case I of our algorithm, for patterns with rows of periods $\leq m/4$, is both linear time and strongly inplace. It can be used for 2D compressed dictionary matching when the patterns and text are compressed by a scheme that can be sequentially decompressed in small space. For example, LZ78 [29] has this property.

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

6 Conclusion

We have developed the first small-space 2D dictionary matching algorithm. We work with a dictionary of d patterns, each of size $m \times m$. After preprocessing the dictionary in small space, and storing the dictionary in a compressed self-index, our algorithm processes the text in linear time, with a sub-logarithmic slowdown. That is, it uses $O(n^2\tau \log \sigma)$ time to search a 2D text that is $O(n^2)$ in size, where $\tau = O(\log^{\epsilon} n)$ is the slowdown introduced by the compressed self-index of the dictionary, $0 < \epsilon \leq 1$. Yet, our algorithm requires only $O(dm \log dm)$ bits of extra space.

Our algorithm is suitable for patterns that are the same size in at least one dimension. Situations arise in which the dictionary contains patterns that are different sizes in both dimensions. Idury and Schaffer's [21] dictionary matching algorithm is for rectangular patterns that can differ in height, width and aspect ratio. A small-space 2D dictionary matching algorithm for rectangular patterns remains an open problem.

Acknowledgements The authors wish to thank S. Muthukrishnan for fruitful discussions.

This work has been supported in part by the National Science Foundation Grant BD&I 0542751 and the Professional Staff Congress—City University of New York Research Award 63343-0041.

Appendix: 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 $H_k(S)$ provides a lower bound to the compression we can achieve using for each symbol 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 symbols following the occurrences of w in S, taken from left to right. The kth order empirical entropy of S is defined as

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

The value $|S|H_k(S)$ 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 $k \geq 0$, $H_{k+1}(S) \leq H_k(S)$ [25].

References

1. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Commun. ACM **18**(6), 333–340 (1975)
2. Amir, A., Benson, G., Farach, M.: An alphabet independent approach to two-dimensional pattern matching. SIAM J. Comput. **23**, 313–323 (1994)
3. Amir, A., Farach, M.: Two-dimensional dictionary matching. Inf. Process. Lett. **44**(5), 233–239 (1992)

4. Amir, A., Landau, G.M., Sokol, D.: Inplace 2d matching in compressed images. J. Algorithms **49**(2), 240–261 (2003)
5. Baker, T.J.: A technique for extending rapid exact-match string matching to arrays of more than one dimension. SIAM J. Comput. **7**, 533–541 (1978)
6. Belazzougui, D.: Succinct dictionary matching with no slowdown. In: Symposium on Combinatorial Pattern Matching (CPM), pp. 88–100 (2010)
7. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Latin American Theoretical Informatics Symposium (LATIN), pp. 88–94 (2000)
8. Bird, R.S.: Two dimensional pattern matching. Inf. Process. Lett. **6**(5), 168–170 (1977)
9. Chan, H.-L., Hon, W.-K., Lam, T.W., Sadakane, K.: Compressed indexes for dynamic text collections. ACM Trans. Algorithms **3**(2), 21 (2007). doi:10.1145/1240233.1240244
10. Crochemore, M., Gasieniec, L., Hariharan, R., Muthukrishnan, S., Rytter, W.: A constant time optimal parallel algorithm for two-dimensional pattern matching. SIAM J. Comput. **27**(3), 668–681 (1998)
11. Crochemore, M., Gasieniec, L., Plandowski, W., Rytter, W.: Two-dimensional pattern matching in linear time and small space. In: Annual Symposium on Theoretical Aspects of Computer Science (STACS), pp. 117–129 (1995)
12. Ferragina, P., Venturini, R.: A simple storage scheme for strings achieving entropy bounds. Theor. Comput. Sci. **372**(1), 115–121 (2007)
13. Fischer, J.: Wee lcp. Inf. Process. Lett. **110**(8–9), 317–320 (2010)
14. Fischer, J., Mäkinen, V., Navarro, G.: Faster entropy-bounded compressed suffix trees. Theor. Comput. Sci. **410**(51), 5354–5364 (2009)
15. Fredriksson, K.: Succinct backward-DAWG-matching. ACM J. Exp. Algorithmics **13**(8), 1.8–1.26 (2009)
16. Fredriksson, K., Nikitin, F.: Simple random access compression. Fundam. Inform. **92**(1–2), 63–81 (2009)
17. Grossi, R., Gupta, A., Vitter, J.S.: High-order entropy-compressed text indexes. In: ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 841–850 (2003)
18. Harel, D., Tarjan, R.E.: Fast algorithms for finding nearest common ancestors. SIAM J. Comput. **13**(2), 338–355 (1984)
19. Hon, W.-K., Lam, T.W., Shah, R., Tam, S.-L., Vitter, J.S.: Compressed index for dictionary matching. In: Data Compression Conference (DCC), pp. 23–32 (2008)
20. Hon, W.-K., Lam, T.W., Shah, R., Tam, S.-L., Vitter, J.S.: Succinct index for dynamic dictionary matching. In: International Symposium on Symbolic and Algebraic Computation (ISAAC), pp. 1034–1043 (2009)
21. Idury, R.M., Schäffer, A.A.: Multiple matching of rectangular patterns. Inf. Comput. **117**(1), 78–90 (1995)
22. Knuth, D.E., Morris, J.H., Pratt, V.R.: Fast pattern matching in strings. SIAM J. Comput. **6**(2), 323–350 (1977)
23. Lothaire, M.: Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications). Cambridge University Press, New York (2005)
24. Main, M.G., Lorentz, R.J.: An O(n log n) algorithm for finding all repetitions in a string. J. Algorithms **5**(3), 422–432 (1984)
25. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM **48**(3), 407–430 (2001)
26. Neuburger, S., Sokol, D.: Small-space 2d compressed dictionary matching. In: Symposium on Combinatorial Pattern Matching (CPM), pp. 27–39 (2010)
27. Russo, L.M.S., Navarro, G., Oliveira, A.L.: Fully compressed suffix trees. ACM Trans. Algorithms **7**(4), 53:1–53:34 (2011)
28. Ukkonen, E.: On-line construction of suffix trees. Algorithmica **14**(3), 249–260 (1995)
29. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theory **24**(5), 530–536 (1978)