Small-Space 2D Compressed Dictionary Matching

Shoshana Neuburger[1] * and Dina Sokol[2] **

[1] Department of Computer Science, The Graduate Center of the City University of New York, New York, NY, 10016
shoshana@sci.brooklyn.cuny.edu
[2] Department of Computer and Information Science, Brooklyn College of the City University of New York, Brooklyn, NY, 11210
sokol@sci.brooklyn.cuny.edu

Abstract. The dictionary matching problem seeks all locations in a text that match any of the patterns in a dictionary. In the *compressed dictionary matching* problem, the input is in compressed form. In this paper we introduce the 2-dimensional compressed dictionary matching problem in Lempel-Ziv compressed images, and present an efficient solution for patterns whose rows are all *periodic*. Given k patterns, each of (uncompressed) size $m \times m$, and a text of (uncompressed) size $n \times n$, all in 2D-LZ compressed form, our algorithm finds all occurrences of the patterns in the text. The algorithm is *strongly inplace*, i.e., the extra space it uses is proportional to the optimal compression of the dictionary, which is $O(km)$. The preprocessing time of the algorithm is $O(km^2)$, linear in the uncompressed dictionary size, and the time for performing the search is linear in the uncompressed text size, independent of the dictionary size. Our algorithm is general in the sense that it can be used for any 2D compression scheme which can be sequentially decompressed in small space.

1 Introduction

The compressed matching problem is the problem of finding all occurrences of a pattern in a compressed text. Various algorithms have been devised to solve the 2D compressed matching problem, e.g., [6, 3, 4]. The dictionary matching problem is that of locating all occurrences of a set of patterns in a given text. In this paper we introduce the compressed dictionary matching problem in 2-dimensions. Compressed dictionary matching can be trivially solved using any compressed pattern matching algorithm and searching for each pattern separately. Preferably, an algorithm should scan the text once so that its search time depends only on the size of the text and not on the size of the dictionary of

* This work has been supported in part by the National Science Foundation Grant DB&I 0542751.
** This work has been supported in part by the National Science Foundation Grant DB&I 0542751 and the PSC-CU Research Award 62280-0040.

patterns. Aho and Corasick achieved this goal for uncompressed patterns and text.

We address 2D compressed dictionary matching when the patterns and text are in LZ78 compressed form. Space is an important concern of a compressed pattern matching algorithm. An algorithm is *strongly inplace* if the amount of extra space it uses is proportional to the optimal compression of the data. The algorithm we present is both linear time and strongly inplace. The problem we are addressing is of practical significance. Many images are stored in Lempel-Ziv compressed form. Facial recognition is a direct application of 2D compressed dictionary matching. The goal of such software is to identify individuals in a larger image based on a dictionary of previously identified faces. An efficient algorithm does not depend on the size of the database of known images.

Pattern matching cannot be performed directly on compressed data since compression is context-sensitive. The same uncompressed string can be compressed differently in different files, depending on the data that precedes the matching content. The *key property* of LZ78 is the ability to perform decompression using constant space in time linear in the uncompressed string. We follow the assumption of Amir et. al. [3] and consider the row-by-row linearization of 2D data.

Existing algorithms for 2D dictionary matching are not sequential. Thus, they are not easily adapted to form strongly-inplace algorithms. Amir and Farach contributed a 2D dictionary matching algorithm that can be used for square patterns [2]. Its time complexity is linear in the size of the text with preprocessing time linear in the size of the dictionary. Their algorithm linearizes the patterns by considering subrow/subcolumn pairs around the diagonal, which is not conducive to row-by-row decompression. Idury and Schaffer discuss multiple pattern matching in two dimensions for rectangular patterns [9]. Although their algorithm is efficient, the data structures require more space than we allow.

We do not know of a small-space dictionary matching algorithm for even one-dimensional data. Multiple pattern matching in LZW compressed text is addressed by Kida et. al. [10]. They present an algorithm that simulates the Aho-Corasick search mechanism for compressed, one-dimensional texts and an uncompressed dictionary of patterns. Their approach uses space proportional to both the compressed text and uncompressed dictionary sizes, which is more space than we allow.

In this paper we present an algorithm that solves the *2D LZ-Compressed Dictionary Matching Problem* where all pattern rows are periodic and the periods are no greater than $m/4$. Given a dictionary of 2D LZ-compressed patterns, $P_1, P_2, \ldots, P_k$, each of uncompressed size $m \times m$, and a compressed text of uncompressed size $n \times n$, we find all occurrences of patterns in the text. Our algorithm is strongly inplace since it uses $O(km)$ space. The best compression that LZ78 can achieve on the dictionary is $O(km)$ [14]. The time complexity of our algorithm is $O(km^2 + n^2 \log \sigma)$, where $\sigma = min(km, |\Sigma|)$ and Σ is the alphabet. After preprocessing the dictionary, the time complexity is independent of the dictionary size.

Amir et. al. present an algorithm for strongly-inplace single pattern matching in 2D LZ78-compressed data [3]. Their algorithm requires $O(m^3)$ time to preprocess the pattern of uncompressed size $m \times m$ and search time proportional to the uncompressed text size. Our preprocessing scheme can be used to reduce the preprocessing time of their algorithm to $O(m^2)$, linear in the size of the uncompressed pattern, resulting in an overall time complexity of $O(m^2 + n^2)$.

2 Overview

We overcome the space requirement of traditional 2D dictionary matching algorithms with an innovative preprocessing scheme that converts 2D patterns to a linear representation. 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 [7] and Baker [5] to linearize 2D data. The preprocessing is performed in a single pass over the patterns with no need to decompress more than two pattern rows at a time. $O(1)$ information is stored per pattern row, resulting in a total of $O(km)$ information. Details of the preprocessing stage can be found in Section 3.

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. Since similar pattern rows were grouped together, we need a verification stage to determine if the candidates are actual pattern occurrences. With additional preprocessing of the 1D pattern representations, a single pass suffices to verify potential pattern occurrences in the text. The details of the text scanning stage are described in Section 4.

The algorithm of Amir et. al. [3] is divided into two cases. A pattern can (i) have only periodic rows with all periods $\leq m/4$ or (ii) have at least one aperiodic row or a row with a period $> m/4$. We focus on the more difficult case, (i). In such an instance, the number of pattern occurrences is potentially larger than the amount of working space we allow. Our algorithm performs linear-time strongly-inplace 2D LZ-compressed dictionary matching of patterns in which all rows are periodic with periods $\leq m/4$.

A known technique for minimizing space is to work with small overlapping text blocks of uncompressed size $3m/2 \times 3m/2$. The potential starts all lie in the upper-left $m/2 \times m/2$ square. If $O(km^2)$ space were allowed, then the 2D-LZ dictionary matching problem would easily be solved by decompressing small text blocks and using any known 2D dictionary matching algorithm within each text block. However, a strongly-inplace algorithm, such as ours, uses only $O(km)$ extra space.

We follow the framework of [3] to sequentially decompress small blocks of 2D-LZ data in time linear in the uncompressed text and in constant space. $O(m)$ pointers are used to keep track of the current location in the compressed text.

3 Pattern Preprocessing

Definition 1. *A string p is* primitive *if it cannot be expressed in the form $p = u^k$, for $k > 1$ and a prefix u of p.*

Definition 2. *A string p is* periodic *in u if $p = u'u^k$ where u' is a suffix of u, u is primitive, and $k \geq 2$.*

A periodic string p can be expressed as $u'u^k$ 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 $|u|$.

Definition 3. *[8] 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 length of overlap in each row is $\geq m/2$.*

Observation 1 *A 2D pattern is h-periodic iff each of its rows is periodic.*

A dictionary of h-periodic patterns can occur $\Omega(km)$ 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 $O(km)$ space suffices to store all occurrences of h-periodic patterns.

We introduce two new data structures that allow our algorithm to achieve a small space yet linear time complexity. They are the witness tree and the offset tree. The witness tree facilitates the linear-time preprocessing of pattern rows. The offset tree allows the text scanning stage to achieve linear time complexity, independent of the number of patterns in the dictionary.

3.1 Lyndon Word Naming

Definition 4. *Two words x, y are* conjugate *if $x = uv$, $y = vu$ for some words u, v [12].*

Definition 5. *A* Lyndon word *is a primitive string which is lexicographically smaller than any of its conjugates [12].*

We partition the pattern rows into disjoint groups. Each group is given a different name and a representative is chosen for each group. Pattern rows whose periods are conjugates of each other are grouped together. Conjugacy is an equivalence relation. Every primitive word has a conjugate which is a Lyndon word; namely, its least conjugate. Computing the smallest conjugate of a word is a practical way of obtaining a standard representation of a word's conjugacy class. This process is called *canonization* and can be done in linear time and space [12]. We

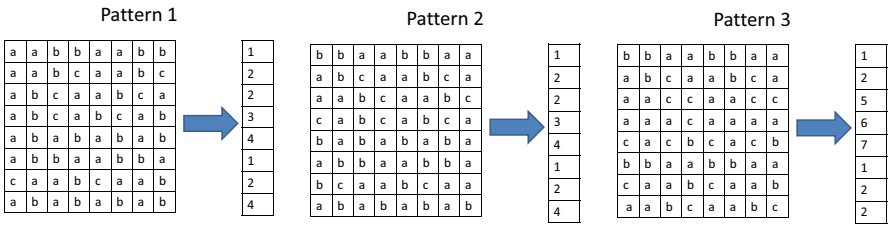


Fig. 1. Three 2D patterns with their 1D representations. Patterns 1 and 2 are not the same, yet their 1D representations are the same.

will use the same 1D name to represent all patterns whose periods are conjugates of each other. This enables us to linearize the 2D patterns in a meaningful manner.

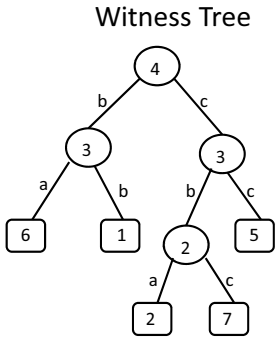
We decompress and name the pattern rows, one at a time. After decompressing a row, its period is found and canonized. If a new Lyndon word or a new period size is encountered, the row is given a new name. Otherwise, the row adopts the name already given to another member of its conjugacy class. A 2D pattern is transformed to a 1D representation by naming its rows. Thus, an $m \times m$ pattern can be represented in $O(m)$ space and a 2D dictionary can be represented in $O(km)$ space.

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

Pattern preprocessing is performed on one row at a time to conserve space. We decompress one row at a time and gather the necessary information. An LZ78 compressed string can be decompressed in time and space linear to the size of the uncompressed string [3]. After decompressing a pattern row, its period is identified using known techniques in linear time and space, i.e., using a KMP automaton [11] of the string. Then, we compute and store $O(1)$ information per row [3]: period size, name, and position of the first Lyndon word occurrence in the row ($LYpos$).

We use the *witness tree* to name the pattern rows. Since we know which Lyndon word represents a row, the same name is given to pattern rows whose periods are conjugates of each other. Rows that have already been named are stored in a witness tree. We only compare a new string to previously named strings of the same size. The witness tree keeps track of failures in Lyndon word

[3] This is under the assumption that the word size is large enough to store log m bits in one word.



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.

character comparisons. With the witness tree, we compare at most one named row to the new row.

3.2 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 Σ. 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. An Aho-Corasick [1] automaton completes this task in $O(km^2)$ time and space, but we allow only $O(km)$ space. 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, if there is one. A witness tree for Lyndon words of length four is depicted in Figure 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. Depending on whether comparison completes successfully, the new string receives either the name of the leaf or a new name.

As an example, we explain how the name 7 becomes a leaf in the witness tree of Figure 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 leaves labeled *2* and *7* as its children, 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, l and j. Both l 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 l and j. This means that l and j should have received the same name. This contradicts the assumption that l and j are different classes. □

Lemma 2. *The witness trees for the rows of k patterns, each of size $m \times m$, is $O(km)$ in size.*

Proof. The proof is by induction. The first time a string of size u is encountered, initialize the tree for strings of size u to a single leaf. Subsequent examination of a string of size u contributes 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. □

The patterns are named in $O(km^2)$ time using only $O(km)$ extra space. This time complexity is optimal since each pattern row must be decompressed and examined at least once. Since we require only $O(1)$ rows to be decompressed at a time, naming is done within $O(m)$ extra space.

3.3 Preprocessing the 1D Patterns

Once the pattern rows are named, an Aho-Corasick (AC) automaton is constructed for the 1D patterns of names. (See Figure 1 for the 1D names of three patterns.) Several different patterns have the same 1D name if their rows belong to the same equivalence classes. 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 candidates in a text block. This conserves space as there are fewer candidates to maintain. In effect, this will also conserve verification time.

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 Section 4.1 where it is used in the verification phase.

In summary, pattern preprocessing in $O(km^2)$ time and $O(m)$ space:

1. For each pattern row, (i) decompress (ii) compute period and canonize (iii) 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 offset tree.

4 Text Scanning

Our algorithm processes the text once and searches for all patterns simultaneously. The text is broken into overlapping blocks of uncompressed size $3m/2 \times 3m/2$. Each text row is decompressed $O(1)$ times with 1 or 2 pointers to mark the end of each row in the block of text. One pointer indicates the position in the compressed text. When the endpoint of a row in the text block occurs in middle of a compressed character, a second pointer indicates its position within the compressed character. In total, $O(m)$ pointers are used to keep track of the current location in the compressed text.

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 pattern.

Step 1. Name Text Rows

We search a 2D text for a 1D dictionary 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 row. Thus, we name the rows of a text block before searching for patterns. The divide-and-conquer algorithm of Main and Lorentz [13] finds all maximal repetitions that

cross a given point in linear time. 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 rows are named in much the same way as the pattern rows are named. The period of the maximal run is found and canonized. Then, the appropriate witness tree is used to name the text row. We use the witness tree constructed during pattern preprocessing since we are only interested in identifying text rows that correspond to Lyndon words found in the pattern rows. At most one pattern row will be decompressed to classify the conjugacy class of a text row. In addition to the name, period size, and *LYpos*, we maintain a *left* and *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 [13]. Its period can be found and canonized in linear time and space [12]. The row is named in $O(m)$ time and space using the appropriate witness tree (Lemma 3). Overall, $O(m^2)$ time and $O(m)$ space is 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 the 1D patterns of names. The occurrence of a 1D pattern indicates the potential occurrence of 2D pattern(s) since several 2D dictionary patterns can have the same 1D name. All *candidates*, or possible pattern starts, are in rows marked with occurrences of the 1D pattern. The occurrence of a 1D pattern is not sufficient evidence that a 2D pattern actually occurs. Thus, a separate verification step is necessary. The *left* pointer with the LYpos identify the first occurrence of a pattern row. Since the patterns are h-periodic, pattern occurrences are at multiples of the first row's period size that leave enough space for the pattern width before *right*.

Complexity of Step 2: 1D dictionary matching in a string of size m can be done in $O(m \log \sigma)$ time and $O(mk)$ space using an AC automaton [1].

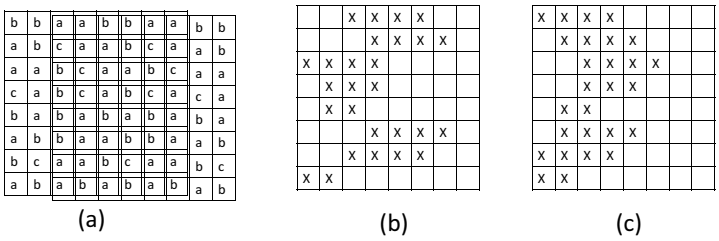


Fig. 3. (a) Two consistent patterns are shown. Each pattern is a horizontal cyclic shift of the other. (b) The first Lyndon word occurrence on each row of the pattern is represented by a sequence of Xs.
(c) The representative of this consistency class. The class representative is the shift in which the Lyndon word of the first row begins at the first position.

Step 3. Verify Candidates

The verification process considers each row of text that contains candidates separately. Recall that a text row contains candidates iff a 1D pattern begins there. 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.

After identifying a text row as containing candidates for a pattern occurrence, we need to ensure 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., $minRight - maxLeft < m$, candidates in the row are 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.

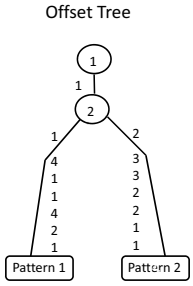
Several different patterns that have the same 1D representation can occur at overlapping positions on the same text row. We call such a set of patterns *consistent*. Consistent patterns can be obtained from one another by performing a horizontal cyclic permutation of the characters, i.e., by moving several columns to the opposite end of the matrix. Figure 3 depicts a pair of consistent patterns. Pattern consistency is an equivalence relation. 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 the 2D array is represented by its 1D arrays of names and *LYpos*. To convert a pattern to one that is 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 consistent patterns and the relative shifts of their rows. Notice that (b) can be obtained from (c) 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 are consistent iff the* LYpos *of all their rows are shifted by* C mod period size of the row*, where C is a constant.*

The proof is omitted due to lack of space and will be included in the journal version.



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 patterns 1 and 2 which have the same 1D name. The *LYpos* entries are not shifted for the first pattern since its first entry is 1, while the *LYpos* entries of the second pattern are shifted by 2 mod period size of row.

4.1 Offset Tree

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 Figure 4.
Components of offset tree:

- *Root:* represents the first row of a pattern.
- *Internal node:* represents a row from 1 to m, strictly larger than its parent.
- *Edge:* labeled by shifted *LYpos* entries. Two edges that leave a node must have different labels.
- *Leaf:* represents a consistency class of dictionary patterns.

We construct an offset tree for each 1D pattern of names. One pattern at a time, we traverse the tree and compare the shifted *LYpos* arrays in sequential order until a mismatch is found or we reach a leaf. If a mismatch occurs at an edge leading to a leaf, a new internal node and 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 (and possibly a new internal node) 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.* The proof is omitted due to lack of space and will be included in the journal version.

Observation 2 *The offset trees for k 1D patterns, each of size m, is of size $O(km)$.*

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

We shift the text's *LYpos* values 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 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 consistent with the window of m rows beginning in a given text row. The last step is to locate the actual positions at which a pattern begins, within the given text row. We need to reverse the shift of the consistent patterns by looking up the first *LYpos* of each pattern that is consistent with the text block. Then we verify that the periodic substrings of the text are sufficiently wide. That is, we announce position i as a pattern occurrence iff $minRight - i \geq m$. Subsequent pattern occurrences in the same row are at LCM multiples of the pattern.

Complexity of Step 3: There can be $O(m)$ rows in a text block that contain candidates. *maxLeft* and *minRight* are computed in $O(m)$ time for the m rows that a pattern can span. The *LYpos* array is modified and shifted in $O(m)$ time. Then, the offset tree is traversed with $O(m \log \sigma)$ comparisons. Determining the actual occurrences of a pattern requires $O(m)$ time, proportional to the width of a pattern row.

Verification of a candidate row is done in $O(m \log \sigma)$ time. Overall, verification of a text block is done in time proportional to the uncompressed text block size, $O(m^2 \log \sigma)$. The verification process requires $O(m)$ space in addition to the $O(km)$ preprocessing 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 $O(m)$ space are required to process text of uncompressed size $n \times n$.

5 Conclusion

We have developed the first strongly-inplace dictionary matching algorithm for 2D LZ78-compressed data. Our algorithm is for h-periodic patterns in which the

period of each row is $\leq m/4$. The preprocessing time-complexity of our algorithm is optimal, as it is proportional to the uncompressed dictionary size. The text scanning stage searches for multiple patterns simultaneously, allowing the text block to be decompressed and processed one row at a time. After information is gathered about the rows of a text block, potential pattern occurrences are identified and then verified in a single pass. Overall, our algorithm requires only $O(km)$ working space.

We would like to extend the algorithm to patterns with an aperiodic row or with a row whose period period $> m/4$. With such a row, many pattern rows with different 1D names can overlap in a text block row. Pattern preprocessing can focus on the first such row of each pattern and form an AC automaton of those rows. However, verification of the candidates requires a small-space 1D dictionary matching algorithm, which seems to be an open problem.

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

[4] A. Amir, G. M. Landau, and D. Sokol. Inplace run-length 2d compressed search. *Theor. Comput. Sci.*, 290(3):1361–1383, 2003.

[5] 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.

[6] P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. Rytter. On the complexity of pattern matching for highly compressed two-dimensional texts. *J. Comput. Syst. Sci.*, 65(2):332–350, 2002.

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

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

[9] R. M. Idury and A. A. Schäffer. Multiple matching of rectangular patterns. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 81–90, New York, NY, USA, 1993. ACM.

[10] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in lzw compressed text. In *DCC '98: Proceedings of the Conference on Data Compression*, page 103, Washington, DC, USA, 1998. IEEE Computer Society.

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

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

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

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