Inplace 2D Matching in Compressed Images

Amihood Amir* Gad M. Landau † Dina Sokol‡

Abstract

The compressed matching problem, defined in [1] is the problem of finding all occurrences of a pattern in a compressed text. In this paper we discuss the 2-dimensional compressed matching problem in Lempel-Ziv compressed images. Given a pattern of (uncompressed) size $m \times m$, and a text of (uncompressed) size $n \times n$, *both* in 2D-LZ compressed form, our algorithm finds all occurrences of P in T. The algorithm is *strongly inplace*, that is, the amount of extra space used is proportional to the best possible compression of a pattern of size m^2. The best compression that the 2D-LZ technique can obtain for a file of size m^2 is $O(m)$. The time for performing the search is $O(n^2)$ and the preprocessing time is $O(m^3)$. Our algorithm is general in the sense that it can be used for any 2D compression which can be sequentially decompressed in small space.

1 Introduction.

The compressed matching problem, motivated by the vast increase of stored compressed data, is the problem of finding all occurrences of a pattern in a compressed text. Since its definition in [1], many results have been achieved in the area of compressed matching. Most of the works have been in 1D texts (e.g. [9, 12, 11]) with a few addressing the 2D version. The 2D algorithms work with run-length compression [6, 7] and hierarchical compression [8]. In this paper we pioneer 2D compressed matching in Lempel-Ziv[1] (LZ78) compressed images [18, 14]. This problem is of both theoretical and practical interest. From a theoretical viewpoint, it has been an open challenge to extend compressed matching to Lempel-Ziv in 2-dimensions. In practice, many images are compressed using variations of the Lempel-Ziv compression technique. For example, CompuServe's GIF standard, widely used on the World-Wide Web, uses LZW (a variation of LZ78) on the image linearized row-by-row.

The goal of compressed matching has generally been to perform pattern matching in time $O(|compress(T)|)$ where $compress(T)$ refers to the compressed text. Using a character-based compression such as Lempel-Ziv, this goal often has little practical value. For a typical file, the compression ratio is a small constant, yielding $O(|compress(T)|) = O(|T|)$. However, as discussed previously in [3, 7, 17], when dealing with compression the criterion of minimizing the extra space is perhaps more important because of various considerations. These include, among others, the efficiency of doing work in main memory – where the file fits in its entirety but auxilliary data structures proportional to the file size will not fit, applications requiring search in local processors with small memory (e.g. in wireless phones) streaming large amounts of data, and local searches on the net. This has led the pattern matching community to consider a paradigm of compressed search with a small amount of additional auxilliary memory. A compressed matching with inputs $compress(P)$ and $compress(T)$, has been defined to be *inplace* if the extra space used, besides the input pattern and text, is $O(|compress(P)|)$. Since we encounter the same difficulty with the constant size compression ratio, we define a stronger space constraint.

DEFINITION 1.1. *Let P be a pattern of size m, and let m' be the optimal compression over all strings of length m using the given compression technique. A compressed matching algorithm with input pattern P is*

*AT&T Labs – Research, Shannon Laboratory, E278, 180 Park Avenue, Florham Park, NJ 07932-0971, (973)360-8120; aamihood@research.att.com; Partially supported by ISF grant 282/01. Part of this work was done when the author was at Georgia Tech, College of Computing and supported by NSF grant CCR-01-04494.

†Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: landau@poly.edu; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grants 173/98 and 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

‡Bar-Ilan University, 52900 Ramat-Gan, Israel, (972-3)531-8407; sokold@cs.biu.ac.il; Partially supported by an Israel Ministry of Industry and Commerce Magnet grant (KITE) and an *AT&T* travel grant.

[1]The 2D compression defined by Lempel and Ziv in [14] uses the Hilbert Curve to linearize the image, and then applies the 1D LZ78 algorithm. We use the row-by-row linearization, since for practical purposes the Hilbert Curve does not achieve a better compression.

called strongly inplace if the amount of extra space used is proportional to m'.

In the current model, our goal is to develop a strongly inplace compressed matching algorithm, while maintaining search time $O(|T|)$. The idea of minimizing the extra space used while relaxing the time constraint originates in [3]. The point is that the Lempel-Ziv compression works by compressing repeated substrings in the text. Pattern matching in LZ compressed texts can exploit this same repetition by storing known information about previous occurrences of a substring. However, if we disallow the storage space, then it seems impossible to obtain the time bound of $O(|compress(T)|)$.

In this paper we present a strongly inplace algorithm for pattern matching in 2-dimensional compressed texts where the compression technique allows sequential decompression is small space. Specifically, our algorithm uses $O(m)$ space for a pattern of size m^2. Using e.g., the 2D-LZ compression, the best possible compression for a string of length m^2 is m. The time complexity of the algorithm is $O(m^3 + n^2)$.

The paper is organized as follows. In the following section we define the problem of both 1D and 2D LZ78 compressed matching, and extract the feature of LZ78 that is exploited by our algorithm. §3 contains some preliminary definitions and an outline of the algorithm's framework. The algorithm is presented in §4 and §5.

2 Problem Definition.

We describe the LZ78 compression technique in 1 and 2-dimensions. The **key property** of LZ78 is the ability to perform *decompression using constant space in time linear in the uncompressed string.* We describe this operation in this section, and in the remainder of the paper we do not use any additional properties of the LZ78 technique.

2.1 LZ78 in 1-Dimension.

The LZ78 [18] compression technique is an adaptive dictionary based compression scheme. Given a string $T = s_1, \ldots, s_n$ over an alphabet Σ, the algorithm parses the string from left to right, creating a sequence of pairs $(i_1, c_1), \ldots, (i_z, c_z)$ called the *compressed string.* At the start, the compressed string, as well as the dictionary is empty. At each step a pair (i, c) is appended to the compressed string and a new codeword is added to the dictionary. i represents the longest dictionary codeword that matches a prefix of the still uncompressed string (0 if no such word exists). c is the character which follows that prefix. The new codeword that is added to the dictionary is the concatenation of word number i and the character c. We illustrate the compression in the following example.

Example. We show how to compress the string *abbababbb*.

Step number:	1	2	3	4	5
Compressed String:	$(0, a)$	$(0, b)$	$(2, a)$	$(3, b)$	$(2, b)$
Codewords:	a	b	ba	bab	bb

Observation 2.1. *The ith element in the compressed string creates the ith codeword in the dictionary.*

Key Operation: Decompression is done in *constant space* and linear time. Using Observation 2.1, we can perform the key operation on the LZ78 technique. Suppose we are uncompressing a string T of length n and the compressed string has length z. The decompression works from right to left. We begin with the last element in the compressed string (i_z, c_z). We set $T[n] = c_z$. To begin the decompression of codeword i_z we jump to location i_z of the compressed string and set $T[n-1] = c_{i_z}$. This continues until we reach an element whose codeword is 0. We then return to location $z-1$ of the compressed string. Referring back to the example above, the decompression of the string *abbababbb* works as follows. We set $T[9] = b$ since the character in position 5 of the compressed string is (2,b). Then, we decompress codeword number 2 by accessing position 2 of the compressed string and setting $T[8] = b$. We return to compressed position number 4 since the codeword at position 2 is a 0, and we set $T[7] = b$. This continues until all the codewords have been decompressed.

Complexity: The space used is $O(1)$ since only two pointers into the compressed string are needed, one to the compressed character that is being decompressed and one to follow the jumps. Since for each jump a single character is uncompressed the time for the decompression is $O(n)$.

Remark: The algorithm of Gasieniec et al [15] called *Sequential Sampling* is a constant space pattern matching algorithm that is sequential. Using our decompression algorithm combined with the *Sequential Sampling Algorithm* we get a compressed matching algorithm for the 1D LZ78 problem which works in constant space and linear time (in the uncompressed text).

2.2 LZ78 in 2 Dimensions.

Definition 2.1. *The 2D-LZ Compression is defined as follows. Given an image $T[1 \ldots n, 1 \ldots n]$, we create a string $T_{lin}[1 \ldots n^2]$ by concatenating all rows of T. Compressing T_{lin} with 1D-LZ78 yields the 2D-LZ compression of the image T.*

Note that the 2D-LZ definition slightly differs from the definition in [14]. In [14] Lempel and Ziv defined their 2D compression technique using the Hilbert Curve to linearize the image followed by the 1D-LZ78. Although they theoretically prove that the Hilbert Curve is optimal, it seems that this is often not the case in practice. We have tested numerous images and the results show that there is no significant difference between the row-by-row order and the Hilbert Curve. Furthermore, in [16] it is proven that in general, for lossless compression of real images, the row-by-row order will yield a better compression than the Hilbert Curve. It should be noted that the GIF standard also used a row-by-row linearization followed by LZW compression, which is very similar to the LZ78.

We also point out that our definition differs slightly from the definition of a 2D run-length compressed text [6]. There, the 2D compression is defined as the concatenation of the compressed rows, while we first concatenate the rows and compress the concatenation of the rows. This distinction is particularly important for LZ compressed images, where it is crucial to be able to reference recurring sequences among different rows.

We define the **Two-Dimensional LZ Compressed Matching Problem** as follows:
Input: Text array T of size $n \times n$, and pattern array P of size $m \times m$ *both* in 2D-LZ compressed form.
Output: All locations in T of occurrences of P. Formally, the output is the set of locations (i, j) such that $T[i+k, j+l] = P[k+1, l+1] \quad k, l = 0 \ldots m-1$.

In this paper we present an algorithm that solves the 2D-LZ Compressed Matching Problem. Our algorithm is strongly inplace since it uses $O(m)$ space. The best compression that LZ78 can achieve on a pattern of length m^2 is $O(m)$ [18]. The time complexity of the algorithm is $O(m^3 + n^2)$. We note that Gasieniec et al [15] have a constant space algorithm for 2-dimensional matching. However, this algorithm is not sequential, and it is very complicated. As such, it cannot be extended to work with a compressed text.

3 Preliminaries.

3.1 Definitions.

DEFINITION 3.1. *A string p is* primitive *if there is no prefix u of p for which $\exists k \geq 2$ such that $p = u^k$.*

DEFINITION 3.2. *A string p is* periodic in u *if $p = u'u^k$ where u' is a suffix of u, u is primitive, and $k \geq 2$.*

LEMMA 3.1. *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, and u can refer to both the string u and the period size $|u|$.

DEFINITION 3.3. *A string u is a* cyclic rotation *of a string v if $u = xy$ and $v = yx$ for some prefix x and suffix y of u. The* phase *of v in relation to u is $|y|$.*

3.2 Framework. A known technique used for minimizing space is to work with small overlapping text blocks of size $2m \times 2m$. If $O(m^2)$ extra space were allowed, then the 2D-LZ problem would be easily solved by decompressing small text blocks, and performing any known 2D pattern matching within each text block. However, a strongly inplace algorithm allows only $O(m)$ extra space. The advantage of the algorithm presented in this paper is that it uses only $O(m)$ extra space.

Our algorithm works with overlapping squares of the text each having uncompressed size $3m/2 \times 3m/2$. The potential starts all lie in the upper-left $m/2 \times m/2$ square. We mark the current block using $3m$ pointers, $3m/2$ to the left edge of the block and $3m/2$ to the right edge. We begin with the bottom-right portion of the text. By sequentially decompressing the bottom $3m/2$ rows, without storing any information, we can find the $3m$ pointers that mark the first block. To move from one text block to another, we decompress sequentially from right to left. When we reach the left edge of the text we return to the right and move upward by decompressing additional rows. Each row of the text is decompressed at most three times. The sequential decompression is done in time linear in the uncompressed text and in constant space as described in §2.1.

Algorithm 1: Marking text blocks

1. **Mark 1st block**
   The pointer to the end of the compressed string is the bottom-right corner of the 2D text. Decompress $3m/2$ characters from right to left to mark the left edge of the bottom row in the text block. Then, sequentially decompress the bottom $3m/2$ rows in the text to get a pointer to the left and right ends of all rows in the bottom-right block.

2. **Move to next block on the left**
   Move the pointers $m/2$ logical columns to the left by sequentially decompressing the $3m/2$ rows from the point of the current block.

3. **Moving upwards**
   We always save the $2m$ pointers to the top m rows of the rightmost block. To move up, we sequentially decompress the next $m/2$ rows above the rightmost block, finding the two pointers per row as in Step 1.

The algorithm for finding the pattern in the small text block is presented in the following two sections. We differentiate between two types of patterns. The first class of patterns are patterns in which all rows are periodic with period $\leq m/4$. The second class of patterns consists of patterns that have at least one row that is either non-periodic, or is periodic with period $> m/4$. In the following section we describe the algorithm for the first type of pattern, and in §5 we describe the algorithm for the second type of pattern.

4 Patterns with All Rows Periodic.

The search for periodic patterns is difficult due to the ability of many pattern occurrences to appear in a small text block. The output itself may be larger than the amount of extra space that we allow ourselves. In previous works periodicity properties were used to succinctly represent the pattern. In [2] three types of 2D periodicity were defined: line, radiant and lattice periodicity. However, in order to ascertain the periodicity type, a witness table of size $O(m^2)$ must be constructed [10]. Our algorithm deals only with the 1D periodicity of each row, and thus can search for periodic patterns using $O(m)$ space.

4.1 Pattern Preprocessing. We use a novel variation of the naming technique to succinctly represent a pattern whose rows are all periodic. We define an equivalence relation R_ℓ over the rows of P. Given two rows, i, j, $iR_\ell j$ iff the period of row i is a cyclic rotation of the period of row j. The "name" of the equivalence class, ℓ, is the lowest row in P whose period is a cyclic rotation of the periods of i and j.

We precompute the following information for each row of P.

1. period size.
2. name.
3. phase (in relation to name).

To compute the period size of each row, we decompress one row at a time and use known techniques to find the string period. One such technique would be to construct the KMP automaton [13] of the string. The failure link of the final state points to the pattern's period.

The naming of the rows of P is performed in $O(m^3)$ time as follows. After decompressing row 1 and determining its period, we search for row 1 in every other row of P, separately. Every row i which has a suffix of length $> 3m/4$ that matches a prefix of row 1 receives the name 1. We mark the position at which row 1 occurs as the phase of row i. The preprocessing of all rows with name 1 is now complete. We continue with the next row following row 1 that has not yet received a name. Upon completion, each row has been named according to the equivalence class of its period. We create a 1D pattern consisting of a single column, using the name of a row to represent the row. We construct the KMP automaton for the new pattern in $O(m)$ time and space.

There is one more step in the pattern preprocessing stage. The text scanning algorithm uses the periodicity of the rows of P to indicate a partial 2d periodicity of P. Specifically, as shown in Lemma 4.2, the distance between any two overlapping pattern occurrences within the same row will be a multiple of the least common multiple (LCM) of the periods of all rows of P. To find these pattern occurrences it is necessary to incrementally compute the LCM of the periods of the rows of P. We precompute this information by building a table $LCM\_table[1 \ldots m]$, where $LCM\_table[i]$ has the value of the LCM of the periods of rows $1 \ldots i$. The entry $LCM\_table[i]$ is computed by taking the LCM of $LCM\_table[i-1]$ and the period of row i. The LCM of two numbers x and y can be found by multiplying $x \times y$ and dividing the product by the greatest common divisor of x and y, which can be found in $O(min(x, y))$ time. Each value takes $O(m)$ time to compute, thus the overall time is $O(m^2)$ and the extra space is $O(m)$.

4.2 Text Scanning. The text scanning stage has two steps.

1. Run the KMP algorithm downwards in the text and mark rows that contain potential starts.
2. Verify candidates separately for each row.

We can run the KMP algorithm downwards in the text if we view the text as a 1-column string in a similar way that we have done with the pattern. When testing whether a row in the text equals a name u, we uncompress the entire row and search for a maximal chain $u^k u'$ with length $\geq m - |u|$. If such a chain exists, we search for a suffix of u to the left of the chain to ensure maximality in both directions. If the length of the resulting chain is $\geq m$ then we mark the location as a match. As in the pattern preprocessing, we label the text row with the name, the phase, and the two ends of its maximal chain. If no such chain exists, the location is said to mismatch, and the KMP algorithm continues accordingly. It remains to show that at most one name can match a given text row.

LEMMA 4.1. *At most 1 name can match a given text row.*

Proof. The proof is by contradiction. Assume that two different names, u and v, match a given text row. Since we are looking at pattern starts within an $m/2 \times m/2$ square, the two chains overlap with at least m/2 characters. The period sizes of u and v are both smaller than $m/4$ and thus at least two adjacent copies of *both* u and v occur in the overlap. By Lemma 3.1 this contradicts the fact that both u and v are primitive.

Complexity of Step 1: The KMP algorithm for a string of length m runs in $O(m)$ time and space. Each "character comparison" costs $O(m)$ time, but only one is done at a time, and therefore $O(m)$ space is used. The total time is $O(m^2)$.

After step 1 completes, a 1D text remains, each row labeled with a name, a phase, and left/right boundaries. All *candidates*, or possible pattern starts, are in the rows that are marked with occurrences of the 1D pattern. Let u be the period of the first row of P. The candidates in a given row begin at the leftmost occurrence of u, and then at jumps of multiples of $|u|$. We call this the *candidate list* of row i and $|u|$ is said to be the period of the candidate list. Note that a candidate list is stored in constant space. We verify each candidate list separately in $O(m)$ time.

To verify the candidate list in row i, we check all rows j in the text, $i \leq j \leq i+m$. For each row j, we do the following three things. After all rows are checked, the remaining candidates in the list are all pattern occurrences.

1. Find the leftmost candidate in the list which has the proper phase in row j. This is done by naively checking each candidate in the list until one is satisfied with the phase of row j.
2. Find the new period of the candidate list. The new period is the least common multiple (LCM) of the old period of the candidate list and the period of row j. (This is proven in Lemma 4.2.)
3. Trim the right edge of the candidate list according to the length of row j.

Complexity of Step 2: Finding the leftmost candidate in a list (step 1) can cost $O(m)$ for a given candidate list and row. However, for a given candidate list in row i, over all rows $i \leq j \leq i+m$ no more than $O(m)$ work will be done. This is because each candidate that is discarded is never checked again. There are at most $O(m)$ discarded candidates. And, for each row j at most one candidate that remains alive is checked. The new period of the list is found by retrieving the LCM from the *LCM_table* in constant time. The trimming of the right edge of the list is also done in constant time per row of P. Thus, the overall time complexity for the verification of a candidate list is $O(m)$. Since there are at most $O(m)$ different candidate lists, the time complexity of the verification phase is $O(m^2)$. A constant amount of extra space is needed for the verification.

LEMMA 4.2. *Consider a 2D pattern P in which all rows of P are periodic and let u equal the LCM of the periods of all rows of P. If a copy of the pattern were to be placed upon itself at location $(1, j)$ the overlap will not conflict if and only if j is a multiple of u.*

Proof. The proof is by induction on the rows of P. For a 1D periodic string, it is trivially true that the string can overlap itself at all multiples of its period. We assume that for a pattern with k rows the LCM of the rows gives the consistent locations. Given a pattern with $k+1$ rows, we let u be the LCM of the first k rows. By the induction hypothesis, multiples of u are the only consistent locations. Let v be the period of the $k+1$st row. The locations at which the pattern can overlap itself will be the intersection between the sets $\{u, 2u, 3u, \ldots\}$ and $\{v, 2v, 3v, \ldots\}$ (see example below). The intersection equals the set of all multiples of LCM(u, v).

Example. LCM$(u, v) = 6$. The pattern can overlap itself without conflict at locations 6 and 12.

	a a a a a a	a a a a a a	a a a a a a
$u = 3$	a b c a b c	a b c a b c	a b c a b c
$v = 2$	a b a b a b	a b a b a b	a b a b a b

Algorithm 2: Patterns will all rows periodic

Begin Algorithm

Pattern Preprocessing:

1. Perform naming on the rows of P.
2. Build the KMP automaton for the new 1-D pattern.
3. Build the *LCM_table* for P.

Text Scanning:

4. Run KMP downwards in the text viewing each text row as a meta-character. Name each text row that matches a pattern row with the same name as the pattern row.
5. Verify each candidate list separately. Verification for a candidate list is done by verifying the phase and the length of every row in the pattern. For each row in

the pattern, the verification of the row modifies the candidate list as follows:
(a) Find the leftmost candidate in the list that has the proper phase, and discard all candidates to the left of it.
(b) Find the new period of the list by taking the LCM of the old period of the candidate list and the period of the additional row.
(c) Trim the right end of the list according to the length of the new row.
After all m rows are verified, all candidates that remain in the candidate list are pattern occurrences.

End Algorithm

Complexity (patterns with all rows periodic): The pattern preprocessing stage has time complexity $O(m^3)$ and space complexity $O(m)$. The text scanning stage has time complexity $O(m^2)$ and space complexity $O(m)$.

5 Patterns with 1 Non-periodic Row.

In this section we discuss patterns that have at least 1 non-periodic row, or a row with period $> m/4$. Henceforth we assume that we have a non-periodic row. The second case will add at most a constant factor to our results. The idea that we will use when searching the text is to begin the search with finding all occurrences of the non-periodic row in the text block of size $3m/2 \times 3m/2$. There will be at most $O(m)$ occurrences of this row. This will narrow the number of potential candidates to $O(m)$. We process these candidates using a modified version of the 2D pattern matching algorithm of [4]. We have used this algorithm in a similar way for the 2D run-length length compressed matching problem [7]. In the following subsection we describe the 2D matching algorithm of [4].

5.1 Algorithm Overview.

The algorithm of [4] performs 2D pattern matching in an uncompressed text in linear time and space. We first give a brief description of the original algorithm, and then describe the necessary modifications to the algorithm in order to reduce the space requirement to $O(m)$.

5.1.1 Pattern Preprocessing. Two witness tables are constructed for the 2D pattern, one for each direction of a duel. Given two overlapping copies of a pattern, a *witness* is a location at which a conflict occurs. The witness table, $Witness[1 \ldots m, 1 \ldots m]$ contains a position of a witness for every possible position of overlap, i.e. $Witness[i,j]$ is the position of a conflict when the pattern is placed upon itself at position (i,j). If all overlapping elements are the same, then we set $Witness[i,j] = *$.

5.1.2 Text Scanning. Part 1 (Candidate Consistency): A *candidate* is a location in the text where the pattern may occur. We say that two candidates are *consistent* if they expect the same text characters in their region of overlap (*i.e.* the value of the witness is a $*$). At the start of the candidate consistency step, every text location is a potential candidate. The goal of the candidate consistency step is to eliminate candidates by performing duels, until all remaining candidates are mutually consistent. A duel is performed between two candidates as follows. We first compute the distance between the candidates, and access the witness table at that position. If the value of the witness is a *, then both candidates remain alive. Otherwise, the value of the text location at the witness position is compared with the witness, and the appropriate candidate is killed.

Part 2 (Verification): In the verification phase the text elements are compared to the pattern elements to discover actual occurrences of the pattern. Although a given text location may be contained in several candidates, since all of the candidates are mutually consistent, each text element must only be compared to a single pattern element. The text elements are compared to the appropriate pattern elements, and all candidates that have a mismatch within their domain are discarded. All remaining candidates are pattern occurrences.

5.1.3 The Compressed Version. Since we want our space to be $O(m)$ we cannot construct the entire witness table. However, the number of candidates at the start of the text processing phase is $O(m)$ and therefore we can allow $O(m)$ time per duel, instead of performing the usual constant time duel. Thus, we construct a table of size $O(m)$ which allows us to locate a witness in $O(m)$ time. The outline of the algorithm is similar to the description of [4]. The implementation of the stages differ as described in the ensuing sections.

Algorithm 3: Patterns with a non-periodic row

Begin Algorithm

Pattern Preprocessing:
Modified witness table construction in $O(m)$ space.

Text Scanning:

1. Search for the non-periodic row in the text to mark $O(m)$ potential candidates.
2. Duel between the $O(m)$ candidates.

3. Verify the consistent candidates.

End Algorithm

To search for the non-periodic row, we uncompress one text row at a time and perform the search. At most $O(1)$ occurrences will be found in each text row, resulting in $O(m)$ possible pattern starts. We can then apply the two phases of the ABF algorithm, dueling and verification. Both the dueling order §5.3, and the verification, §5.4, are similar in [7] where the authors applied the ABF algorithm to inplace run-length compressed search. The section on performing the actual duel (which immediately follows) is new to this algorithm.

5.2 Witness Computation and Dueling. In this section we describe the preprocessing necessary for performing a duel in the compressed text in $O(m)$ time. We then describe how the duel is performed. When constructing a witness table for a 2-dimensional pattern it is necessary to construct two independent witness tables, one for each direction of a duel. This applies as well to the witness table that we will describe in this section. Specifically, we number the quadrants of P $1, 2, 3, 4$ counterclockwise. We discuss certain properties of quadrants 1 and 3 for computing witnesses in the bottom/right direction. For the other direction, the same holds for quadrants 2 and 4.

Definition 5.1. *A $\frac{1}{2}$-row is either the first or last $m/2$ characters in a row.*

We divide the patterns into two subclasses. The first group of patterns has a $\frac{1}{2}$-row in quadrant 1 or 3 that is non-periodic. In the second group, all $\frac{1}{2}$-rows in quadrants 1 and 3 are periodic.

5.2.1 Case 1: Some $\frac{1}{2}$-row (in quadrant 1 or 3) is non-periodic. Pattern Preprocessing: We assume wlog that the non-periodic $\frac{1}{2}$-row is row r in quadrant 1. We search the pattern for complete occurrences of the $\frac{1}{2}$-row r. There are at most $O(m)$ occurrences of r in P since r is a non-periodic string. For each occurrence at location (i, j) in P we compute a witness for location $(i-r, j)$. The witness is computed in the naive way in $O(m^2)$ time and stored in a linked list of length $O(m)$. Note that for every location (i, j) of P for which r does *not* occur at $(i+r, j)$, no preprocessing is necessary since there will be a witness for (i, j) in the row beginning at position $(i + r, j)$. The total time complexity is therefore $O(m^3)$.

Dueling: We define two candidates in the text $UL = (x, y)$ (for upper-left) and $BR = (x + i, y + j)$ (for bottom right). There is an occurrence of r in the text at both $(x + r, y)$ and at $(x + i + r, y + j)$ since the candidates were marked according to occurrences of r in the text. We search the witness list to find whether we have stored a witness for the location (i, j). If no such witness exists, then we kill UL since it has an occurrence of r at the wrong location. If a witness has been recorded then we uncompress the row of the witness, and in $O(m)$ time we can perform the duel.

5.2.2 Case 2: All $\frac{1}{2}$-rows in quadrants 1 and 3 are periodic. Pattern Preprocessing: Every row has a prefix and/or suffix of length $\geq m/2$ that is periodic. For each row, we compute the period of its prefix and/or suffix and perform the naming over the prefixes and suffixes using the equivalence relation defined in §4.1. We mark the length of the maximal periodic prefix and suffix for each row, that is, we check how far to the right (left) the periodic prefix (suffix) extends. Next, we compute a constant amount of information per row of P. This information will allow a duel to be performed in $O(m)$ time.

We describe the preprocessing for row i of P. Assume UL and BR are two overlapping copies of P such that BR begins in row i of UL. The overlap between the two patterns consists of prefixes of rows $1 \ldots m - i$, and suffixes of rows $i \ldots m$. We consider the set of maximal periodic prefixes of rows $1 \ldots m - i$ and maximal periodic suffixes of rows $i \ldots m$, and we search for the *shortest* string in the set. Let ℓ be the length of the *shortest* string in the set, and wlog we assume that the shortest string is a periodic prefix in row $1 \leq r \leq m - i$. The values (ℓ, r) for row i will be used to calculate witnesses for all (i, j), $1 \leq j \leq m/2$. For values of j such that $m - j > \ell$ (i.e. the width of the overlap is larger than ℓ) row r will have a witness for all but at most one location. This is true since the overlap in row r is a non-periodic string. A non-periodic string cannot match another string at more than one location prior to position $m/2$, since it would match itself before its $1/2$-point contradicting its non-periodicity.

Thus, the preprocessing for row i proceeds as follows. After finding the values (ℓ, r) we search for row r in row $r + i$ and mark the occurrence of the longest prefix of row r that matches a suffix of row $r + i$, if one exists. Then, we search for a witness for that one location in the naive way in $O(m^2)$ time.

In summary, the preprocessing for Case 2 has 4 steps.

1. For each row of P, find the maximal periodic prefix and suffix.
2. For each row i of P,
   (a) find the shortest string in the set of periodic

prefixes of rows $1 \ldots m-i$ and periodic suffixes of rows $i \ldots m$. Let ℓ, r be the length/row of the shortest string.

(b) Find the leftmost occurrence of row r in row $r+i$.

(c) For the position found in the previous step, find a witness in $O(m^2)$ time.

Dueling: We define two candidates in the text, $UL = (x, y)$ (for upper-left) and $BR = (x+i, y+j)$ (for bottom right). To duel between UL and BR we retrieve the values ℓ and r for row i and check whether $m-j > \ell$.

If $m-j > \ell$ then the overlap between UL and BR is longer than ℓ, and the overlap in row r is a non-periodic string. Thus, there exists a witness in row r for all but at most one possible value of j. We check whether j is the matching location computed in the preprocessing. If yes, the precomputed witness is retrieved, the text row is uncompressed, and the appropriate candidate is killed. Otherwise, row $x+i+r$ (row r of BR) in the text contains a witness. The row is uncompressed, and matched against both rows r and $r+i$ of P. Only one candidate will survive.

If $m-j \leq \ell$ then all overlapping strings are periodic. We use a technique similar to the verification described in §4.2 for patterns with all rows periodic. For each row in the overlap we check whether the rows in UL and BR are consistent. The overlap is equal in a given row if the prefix and suffix have the same name, and the phase is correct according the position of BR. This checking can be done in constant time per row. If all overlapping rows are equal then the candidates are consistent. Otherwise, we uncompress a non-equal row in $O(m)$ time and perform the duel.

5.3 Dueling Order. Dueling between all candidates is necessary to ensure that the candidates are pairwise consistent. Dueling between all pairs in the naive way may result in $O(m^2)$ duels. We would like the number of duels to be proportional to m since we spend $O(m)$ per duel. In this subsection we show how to order the duels to result in $O(m)$ duels.

We sort the candidate list, first by column and then by row. If any column has more than one candidate, we duel within the column first. We then move from right to left, adding one column at a time. We consider the four periodicity classes defined in [2]. We show separately for each periodicity type that using the order that we described, no more than $O(m)$ duels will be necessary. Following we describe, for each periodicity class, the way the pattern starts may appear in an $m/2 \times m/2$ text block B.

1. Non-periodic: There is at most one pattern occurrence in B.
2. Line Periodic: The pattern starts in B all fall on one line.
3. Radiant Periodic: The pattern starts in B are *ordered monotonically.* We say that candidates of a pattern in a text are *ordered monotonically* if they are non-decreasing in both row and column indices or non-increasing in row index and non-decreasing in column index.
4. Lattice Periodic: The pattern starts in B fall on the nodes of a lattice. The lattice is defined by the basis vectors of the pattern (see [2, 10]).

(1) Non-periodic: If the pattern is non-periodic, then one candidate is killed in every duel. Since at the start there are at most $O(m)$ candidates, the number of duels is no more than $O(m)$.

(2,3) Line and Radiant Periodic: As in [4], we perform the duels within each logical column. Due to the transitivity lemma (lemma 3.1 in [4]) the number of duels within a logical column is no more than the number of candidates within the column. If the pattern is line or radiant periodic then at most one candidate remains alive in each column, resulting in at most m candidates. We move from right to left, adding one candidate at a time. Note that the transitivity lemma holds within the group of consistent candidates since they are ordered monotonically. Thus, the number of duels performed between columns is $O(m)$.

(4) Lattice Periodic: Similarly, in the lattice periodic case we first remove conflicts within each logical column. Moving from right to left, we add one candidate at time to the group of consistent candidates. A given column can contain several candidates, however, each new candidate can duel with **any** candidate to its right. If a candidate dies, then the operation is charged to the dead candidate. No more than $O(m)$ such duels will take place. If both candidates remain alive, then the new candidate is consistent with all of the candidates to its right. This has been proven in [7].

5.4 Verification. The standard verification entails comparing each text element with a given pattern element and marking the positions of each mismatch. Since all candidates are consistent, each text element must be compared to only one pattern element. All candidates that contain a mismatch within their domain are discarded. We have shown in [7] how a set of consistent candidates can be verified using $O(m)$ space. We use a similar technique here, but in this case the verification is simpler since there are at most $O(m)$ candidates.

All candidates within an $m/2 \times m/2$ text square overlap, and thus it is sufficient to verify a given text row for two candidates: the rightmost and leftmost. In addition, it is not necessary to mark all of the mismatches in a given text row. We call column $\frac{3}{4}m$ the *center* of the text. We find the two mismatches in each row that are the closest to the center, one to its left, and one to its right. In [7] we have proven that any candidate that contains a mismatch will include one of the mismatches that is closest to the center.

Thus, the verification proceeds as follows. For each text row i, we find the two mismatches that are the closest to the center. First, in $O(m)$ time we search the candidates to find the leftmost and rightmost candidates that include row i. Since these two candidates overlap, the entire row is covered by the two candidates, and we verify the row, marking only the one mismatch that is closest to the center on the left and one on the right. After all text rows have been verified, we check each candidate individually. Each candidate checks whether any of the marked mismatches is in its domain. All candidates that contain a mismatch are discarded, and all remaining candidates are reported as pattern occurrences.

Complexity (patterns with 1 non-periodic row): Pattern Preprocessing: For case 1, the search for the non-periodic row in P is done in $O(m^2)$ time, and the computation of $O(m)$ witnesses is done in $O(m^3)$ time. For case 2, both the naming of the prefixes/suffixes, and the search for $O(m)$ witnesses take $O(m^3)$ time. Thus, the pattern preprocessing stage takes $O(m^3)$ time and uses $O(m)$ space.

Text Scanning: The search for the non-periodic row in the text is done in $O(m^2)$ time. In the dueling stage, $O(m)$ duels are performed, each taking $O(m)$ time. The verification also takes $O(m^2)$ time since $O(m)$ work is done per text row and per candidate (of which there are $O(m)$). The extra space used for the candidate list in the dueling stage, and the list of mismatches in the verification stage is $O(m)$. Overall, the time complexity of the text scanning phase is $O(m^2)$ and the space complexity is $O(m)$.

6 Conclusion

In this paper, we have presented the first compressed matching algorithm for 2D Lempel-Ziv Compressed Matching. The main advantage of our algorithm is its space efficiency. The algorithm is strongly inplace, i.e. the amount of extra space used is proportional to the optimal compression obtainable for a pattern of size $|P|$. In addition, the time complexity of the algorithm is linear in the uncompressed text. Solving the 2D Lempel-Ziv Compressed Matching Problem in time proportional to the compressed text remains an open problem. Reducing the pattern preprocessing time of our algorithm from $O(m^3)$ is another desired improvement.

Another direction for further research is to solve the 2D Lempel-Ziv Matching Problem for other variations of the Lempel-Ziv technique (e.g. LZW, LZ77). The algorithm presented in this paper requires decompression to be performed in linear time and constant space. There is no known algorithm for LZW or LZ77 that performs this key operation. In [5], a decompression algorithm for LZW using constant space is described. However, the time complexity of the algorithm is $O(n^{1.5})$ for decompressing a string of length n. Thus, even in the 1-dimensional case, the problem of finding a strongly inplace/linear time algorithm for LZW and LZ77 Compressed Matching remains open.

Acknowledgements

The authors thank Gadi Horev, chairman of the Department of radiology at Schneider Children's Medical Center in Petach-Tikva, Israel, and Ken Church, for helpful discussions.

References

[1] A. Amir and G. Benson. Efficient two dimensional compressed matching. *Proc. of Data Compression Conference, Snow Bird, Utah*, pages 279–288, Mar 1992.

[2] A. Amir and G. Benson. Two-dimensional periodicity and its application. *SIAM J. Comp.*, 27(1):90–106, February 1998.

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. Technical Report GIT-CC-93/42, Georgia Institute of Technology, June 1993.

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

[5] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[6] A. Amir, G. Benson, and M. Farach. Optimal two-dimensional compressed matching. *Journal of Algorithms*, 24(2):354–379, August 1997.

[7] A. Amir, G.M. Landau, and D. Sokol. Inplace run-length 2-dimensional compressed search. *Theoretical Computer Science*, 2002. to appear.

[8] P. Berman, M. Karpinski, L. Larmore, W. Plandowski, and W. Rytter. On the complexity of pattern matching for highly compressed two dimensional texts. In *Proc. 8th Annual Symposium on Combinatorial Pattern Matching (CPM 97)*, pages 40–51. LNCS 1264, Springer, 1997.

[9] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Proc. 27th Annual ACM Symposium on the Theory of Computing*, pages 703–712, 1995.

[10] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Computing*, 25(5):907–935, October 1996.

[11] L. Gasieniec, M. Karpinski, W. Plandowski, and W. Rytter. Randomized efficient algorithms for compressed strings: The finger-print approach. In *Proc. 7th Annual Symposium on Combinatorial Pattern Matching (CPM 96)*, pages 39–49. LNCS 1075, Springer, 1996.

[12] L. Gasieniec and W. Rytter. Almost optimal fully compresssed pattern matching. In *Data Compression Conference (DCC), Snow Bird, Utah*, pages 434–443, 1999.

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

[14] A. Lempel and J. Ziv. Compression of two-dimensional images. In Z. Galil A. Apostolico, editor, *Combinatorial Algorithms on Words*, volume 12, pages 141–154. NATO ASI Series F, 1985.

[15] W.Plandowski L.Gasieniec and W.Rytter. Constant-space string matching with smaller number of comparisons: sequential sampling. In *Proc. 6th Annual Symposium on Combinatorial Pattern Matching (CPM 95).* LNCS, Springer, 1995.

[16] N. Memon, D. Neuhoff, and S. Shende. An analysis of some common scanning techniques for lossless image coding. `http://citeseer.nj.nec.com/23495.html`.

[17] J. Ziv. personal communication. 1995.

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