On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts[1]

Piotr Berman[a2], Marek Karpinski[b3], Lawrence L. Larmore[c 4],
Wojciech Plandowski[d5] and Wojciech Rytter[d5]

[a] Dept. of Computer Science & Eng., Pennsylvania State University, University Park, PA 16802.

[b] Dept. of Computer Science, University of Bonn, 53117 Bonn.

[c] Department of Computer Science, University of Nevada, Las Vegas, NV 89154-4019.

[d] Instytut Informatyki, Uniwersytet Warszawski, Banacha 2, 02-097 Warszawa.

[1]Part of this work was done while four of the authors were visiting Dept of Computer Science, University of Bonn. An extended abstract of this work was presented at CPM'97 of [5].

[2]Supported in part by NSF grant CCR-9700053. E-mail: berman@cse.psu.edu.

[3]Research was partially supported by the DFG Grant KA 673/4-1, ESPRIT BR Grants 7079, 21726, and EC-US 030, and the Max-Planck Research Prize. E-mail: marek@cs.uni-bonn.de.

[4]Research partially supported by National Science Foundation grants CCR-9503441, CCR-9821009. E-mail: larmore@cs.unlv.edu.

[5]Supported by the grant KBN 8T11C03915. E-mail: {wojtekpl,rytter}@mimuw.edu.pl.

We consider the complexity of problems related to 2-dimensional texts (2D-texts) described succinctly. In a succinct description, larger rectangular sub-texts are defined in terms of smaller parts in a way similar to that of Lempel-Ziv compression for 1-dimensional texts, or in strings with shortened descriptions as in [10], or in hierarchical graphs described by context-free graph grammars. A given 2D-text T with many internal repetitions can have a hierarchical description which may be exponentially smaller and which can be given as an input for a pattern-matching algorithm which gives information about T. Such a hierarchical description is given in terms of a straight-line program, or SLP (see [10]) or, equivalently, a 2-dimensional grammar.

We consider **compressed pattern-matching**, where the input consists of a 2D-pattern P and a hierarchical description of a 2D-text T, and **fully compressed pattern-matching**, where the input consists of hierarchical descriptions of both the pattern P and the text T. For 1-dimensional strings, there exist polynomial-time deterministic algorithms for these problems for similar types of succinct text descriptions [2, 7, 9, 10].

We show that the complexity dramatically increases in a 2-dimensional setting. For example, compressed 2D-matching is **NP**-complete, fully compressed 2D-matching is $\Sigma_2^{\mathbf{P}}$-complete, and testing a given occurrence of a two dimensional compressed pattern is co-**NP**-complete. On the other hand, we give efficient algorithms for the related problems of randomized equality testing and testing for a given occurrence of an uncompressed pattern.

We also show the surprising fact that the compressed size of a subrectangle of a compressed two dimensional array can grow exponentially, unlike the one-dimensional case.

1. INTRODUCTION

We consider a special form of highly compressed 2D-texts, *i.e.*, two dimensional arrays with entries from some finite alphabet. A 2D-text T is represented by a straight-line program (SLP), that uses constants from the alphabet and two types of assignment statements (see Fig. 1):

Horizontal concatenation:

$A \leftarrow B \oslash C$, which horizontally concatenates 2D-texts B and C of equal height.

Vertical concatenation:

$A \leftarrow B \ominus C$, which vertically concatenates 2D-texts B and C of equal width.

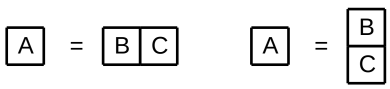


FIG. 1. Graphical illustration of horizontal concatenation $\oslash$ and vertical concatenation $\ominus$ of rectangles.

An SLP $\mathcal{P}$ of size n (we write $|\mathcal{P}| > n$) consists of n statements of the above form, where the result of the last statement is the compressed 2D-text. The derived 2D-text is denoted $P = \mathit{Decompress}(\mathcal{P})$, and we say that $\mathcal{P}$ is a *compressed form* of P. The *area* of P, denoted $|P|$, can be exponential in $|\mathcal{P}|$. For this reason it is possible for the text matching problems discussed in this paper to lack polynomial time algorithms.

The hardest computational problem discussed here is **fully compressed matching**:

Instance: two SLP's $\mathcal{P}$ and $\mathcal{T}$, representing the *pattern* and the *text*.

Question: does *Decompress*($\mathcal{P}$) occur in *Decompress*($\mathcal{T}$) as a contiguous subarray?

We can change the way we formulate a problem instance by representing the pattern directly, *i.e.*, as an array P, rather than as an SLP that is its compressed form. This defines the **compressed matching** problem. We may also add to the problem instance the coordinates of a location within the text, and ask whether the text contains an occurence of the pattern with this location as its lower left corner. By representing the pattern in the compressed and uncompressed form we define the **fully compressed pattern checking** and **compressed pattern checking** problems.

The complexity of basic string problems for one dimensional texts is polynomial (see [4, 7, 9, 11]). Surprisingly, the complexity jumps if we pass to two dimensions.

If $P = Decompress(\mathcal{P})$, write *width*($\mathcal{P}$) for the number of columns of P, *height*($\mathcal{P}$) for the number of rows of P, and $\mathcal{P}_{i,j}$ for $P_{i,j}$. We number rows and columns starting from 0.

In some of our proofs, we identify Boolean vectors with integers from $\{0,\ldots,2^n-1\}$, where n is the vector length (usually implied from context). In our notation, we have a correspondence between a vector $\overline{x} = (x_0, x_1, \ldots, x_n - 1) \in \{0,1\}^n$ and the number $x = \sum_{j=0}^{n-1} x_j 2^j$.

2. EXAMPLES OF COMPRESSION OF 2D-TEXTS

Example 1.

Hilbert's curve can be viewed as an image which is exponentially compressible in terms of SLP's. An SLP which describes the n^{th} Hilbert's curve, H_n, uses six (terminal) symbols , , , , , , and 12 variables $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, $_i$, for each $0 \le i \le n$. A variable with index i represents a text square of size $2^i \times 2^i$ containing part of a curve. The dots in the boxes show the places where the curve enters and leaves the box.

The 2D-text T = $_3$ describing the 3[rd] Hilbert's curve is shown in Figure 2. It is composed of four smaller square 2D-texts $_2$, $_2$, $_2$ and $_2$ according to one of the composition rules.

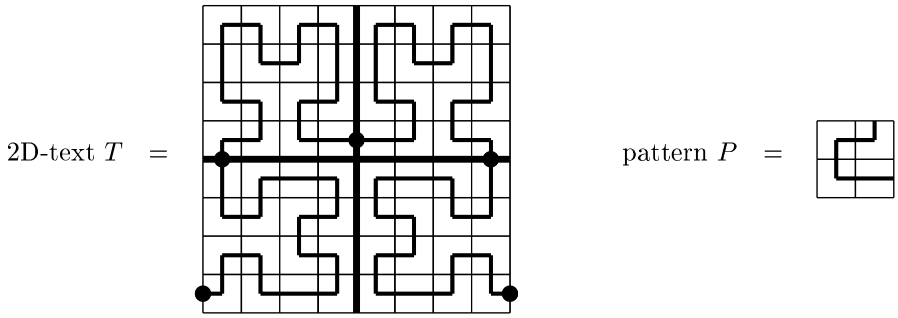
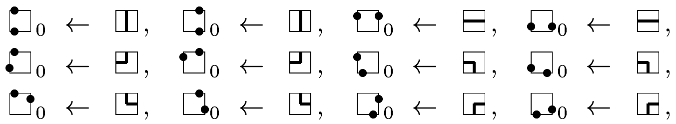
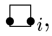
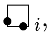
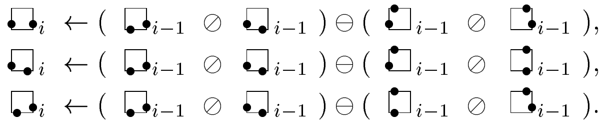


FIG. 2. An example of a 2D-text T and a pattern P. The pattern occurs twice in the text. The black dots indicate how T was defined with statement $_3 \leftarrow ($ $_2 \oslash$ $_2) \ominus ($ $_2 \oslash$ $_2)$.

The 1×1 text squares are described as follows.



The text squares for variables indexed by $i \geq 1$ are rotations of text squares for the variables , , . These variables are composed according to the rules:



Example 2.

In our constructions we will use, as a building block, rectangles filled with with 0's. Let $Z_{i,j}$ denote the all-0's rectangle of size $i \times j$. such an $i \times j$ 2D-text.

OBSERVATION 2.1. *For each $i, j > 0$ there exists an* SLP $\mathcal{Z}_{i,j}$ *with size below* $2(\log i + \log j) + 1$ *such that $Z_{i,j} = Decompress(\mathcal{Z}_{i,j})$.*

Example 3.

The compressed size of a subrectangle of a compressed two dimensional array A can be exponential with respect to the compressed size of A, though such a situation cannot occur in the 1-dimensional case. This phenomenon appears to be responsible for the increase in the time complexity.

THEOREM 2.2. *For each n there exists an* SLP *of size n describing a text image A_n and a subrectangle B_n of A_n such that the smallest* SLP *describing B_n has exponential size.*

Proof. We can generate, using an SLP of size $O(n)$, a $2^n \times (2^n + 1)$ bit pattern S_n, where the first column of S_n is all 1's, and the remaining square consists of 1's on the main diagonal and 0's elsewhere.

Using another $O(n)$ statements, we form a $(2^{n+1} - 1) \times (2^n + 1)2^n$ 2D-text T_n which contains 2^n copies of S_n, and where each copy of S_n intersects the central row of T_n at a different row. (See Fig. 3.) Each copy of S_n is shifted down by one with respect to the preceding copy. We claim that the central row of T_n cannot be described by an SLP of size less than 2^n.

Observe that the SLP that describes the central row is a context-free grammar in Chomsky Normal Form. For each integer $k \in \{0 \ldots 2^n - 1\}$, 10^k1 is a substring of that row. Let A_k be a minimal variable which derives a substring including 10^k1, *i.e.*, $A_k \leftarrow B_kC_k$, where B_k derives a string which has 10^p as a suffix and C_k derives a string which has $0^{k-p}1$ as a prefix, for some p. The A_k must be distinct. Thus, the grammar must have at least 2^n variables. ❐

3. EQUALITY TESTING IN RANDOMIZED POLYNOMIAL TIME

We reduce equality of two 2D-texts $\mathcal{A}$ and $\mathcal{B}$ to equality of two polynomials $Poly_{\mathcal{A}}(x, y)$ and $Poly_{\mathcal{B}}(x, y)$. Assume that the symbols are integers in some small range. For an $n \times m$ SLP $\mathcal{Z}$ define

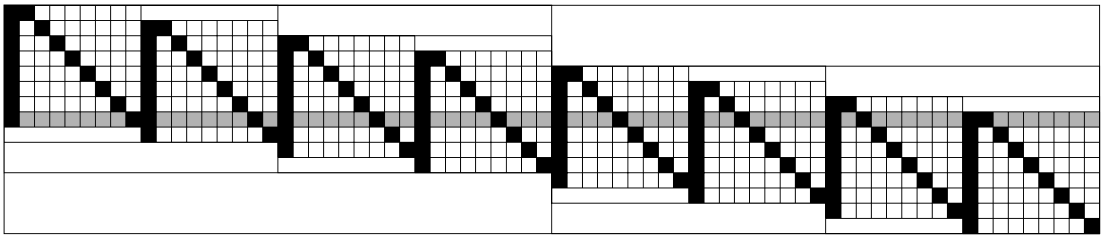


FIG. 3. Text T_3: black squares indicate the location of 1's, white squares and rectangles indicate 0's. White rectangles show how T_i is formed from T_{i-1}. The gray band shows the row where the numbers of 0's between consecutive 1's give all the numbers in the range from 0 to $2^n - 1$. There are 8 copies of S_3, each copy shifted down by one with respect to the previous one.

its corresponding polynomial

$$Poly_{\mathcal{Z}}(x,y) = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \mathcal{Z}_{i,j} x^i y^j.$$

OBSERVATION 3.1. *Let $\mathcal{A}$ and $\mathcal{B}$ be two 2D-texts. Then $\mathcal{A} = \mathcal{B} \Leftrightarrow Poly_{\mathcal{A}} \equiv Poly_{\mathcal{B}}$.*

LEMMA 3.1. *Given an* SLP $\mathcal{Z}$*, one can compute, in polynomial time, an* SLP *that computes $Poly_{\mathcal{Z}}$.*

Proof. We prove, by induction on $|\mathcal{Z}|$, that we can find SLP's that compute $Poly_{\mathcal{Z}}(x,y)$, $x^{height(\mathcal{Z})}$ and $y^{width(\mathcal{Z})}$

The basis is trivial. Assume that we can find corresponding polynomials for variables A and B where $|A|$, $|B| < |\mathcal{Z}|$ and the last statement of $\mathcal{Z}$ is $A \leftarrow B \oslash C$.

We append the SLP for the previous statements (corresponding to A and B) with

$$A \leftarrow B + y^{width(\mathcal{B})} C$$
$$y^{width(\mathcal{A})} \leftarrow y^{width(\mathcal{B})} \cdot y^{width(\mathcal{C})}$$
$$x^{height(\mathcal{A})} \leftarrow x^{height(\mathcal{B})}$$

where we treat expressions such as "$y^{width(\mathcal{A})}$" as variables of the new SLP. The case of $A \leftarrow B \ominus C$ is symmetric. ❐

The following results are versions of theorems given by Schwartz [18] and Zippel [15] (see also Ibarra and Moran [14]).

LEMMA 3.2. *Let $\mathcal{P}$ be a nonzero polynomial of degree at most d. Assume that we assign to each variable in $\mathcal{P}$ a random value from a set Ω of integers of cardinality R. Then*

$$Prob\{\mathcal{P}(\bar{x}) \neq 0 \} \geq 1 - \frac{d}{R}.$$

THEOREM 3.2. *There exists a polynomial time randomized algorithm for testing equality of two 2D-texts, given their* SLP*'s.*

Proof. Let n be the total size of compressed description of $\mathcal{A}$, $\mathcal{B}$. Denote

$$deg = \max\{degree(\mathcal{A}), degree(\mathcal{B})\}$$

The value of deg corresponds to the maximum size of the 2D-texts, and we have $deg \leq c^n$ for some constant c. By Lemma 3.2, we can test equality of $\mathcal{A}$ and $\mathcal{B}$ in a randomized way, by selecting random values x_0, y_0 of variables, say in the range $[1 \ldots 4 \cdot deg]$ and testing $y_1 = y_2$, where $y_1 = Poly_{\mathcal{A}}(x_0, y_0)$ and $y_2 = Poly_{\mathcal{B}}(x_0, y_0)$. This guarantees that the probability of test correctness is at least $\frac{1}{2}$. The test can be repeated a logarithmic number of times to guarantee that the probability of failure is at most $\frac{1}{n}$.

However, there is one technical difficulty. The numbers y_1, y_2 are exponential with respect to deg and could need an exponential number of bits. In that case, we are not able to compute them in polynomial time. Hence instead of computing and comparing the exact values of y_1, y_2 we choose a random prime number p from the interval $[1 \ldots n^2]$ and compute the values y_1, y_2 *modulo* p. We refer the reader to Theorem 7.5 and the discussion in Example 7.1 of [15], for details about randomized testing of the equality of two number using modular arithmetic and prime numbers with an exponentially smaller number of bits than the numbers to be tested.

If the computed values y_1 *mod* p and y_2 *mod* p are different, then the polynomials are different. Otherwise, by Lemma 3.2, the polynomials are identical with high probability. The computation of y_1 *mod* p and y_2 *mod* p, where p is a prime number with polynomially many bits, can be done in polynomial time with respect to n. This completes the proof. ❐

Open Problem:

We have given a fast randomized algorithm for the equality of two compressed 2D-texts. We conjecture that there is a polynomial time deterministic algorithm for this problem.

4. COMPRESSED TWO DIMENSIONAL PATTERN-MATCHING

First we consider the **Point test problem**: compute the symbol $\mathcal{T}_{i,j}$ for given $\mathcal{T}$, i and j.

LEMMA 4.1. *There exists an $O(|\mathcal{T}|)$-time algorithm for the point test problem.*

Proof. Assume that we are must compute $\mathcal{T}_{i,j}$ for given i, j and $\mathcal{T}$. Let $\mathcal{T}^k$ be the SLP $\mathcal{T}$ truncated to the first k statements. In linear time, we can find $height(\mathcal{T}^k)$ and $width(\mathcal{T}^k)$ for $k = 1, \ldots, |\mathcal{T}|$. Now, suppose that the last statement of $\mathcal{T}$ is $A \leftarrow B \oslash C$, the k^{th} statement of $\mathcal{T}$ is $B \leftarrow \ldots$, and the l^{th} statement is $C \leftarrow \ldots$ If $j < width(\mathcal{T}^k)$, then we must find $\mathcal{T}^k_{i,j}$, otherwise we must find $\mathcal{T}^l_{i,j-width(\mathcal{T}^k)}$. The case of $A \leftarrow B \ominus C$ is similar. ❐

THEOREM 4.1. *Compressed matching for 2D-texts is* **NP***-complete.*

Proof. To see that compressed matching is in **NP**, we express this problem as the following property of pattern P and SLP $\mathcal{R}$:

$$\exists (i, j)\{\forall k < height(P), l < width(P) : P_{k,l} = \mathcal{R}_{i+k,j+l}\}$$

The equality inside the braces can be tested in polynomial time (Lemma 4.1), hence we have expressed the problem in the normal form for **NP**.

To show **NP**-hardness, we will use a reduction from 3SAT.

Consider a set of clauses $C_0, \ldots, C_{k-1}$, where each clause is a disjunction of three literals with binary variables from the set $\{x_0, \ldots, x_{n-1}\}$. The 3SAT question is whether there exists x such that

$0 \le x \le 2^n - 1$ and $C_i(\overline{x}) = 1$ for $i = 0, \ldots, k-1$. By using the correspondence between an integer $x \in \{0, 1, \ldots, 2^n - 1\}$ and $\overline{x}$, its vector of binary digits, we will view a clause as a $0/1$ function with arguments in $\{0, 1, \ldots, 2^n - 1\}$. From this point of view, the 3SAT question is whether there exists $x < 2^n$ such that $C_i(\overline{x}) = 1$ for $i = 0, \ldots, k-1$.

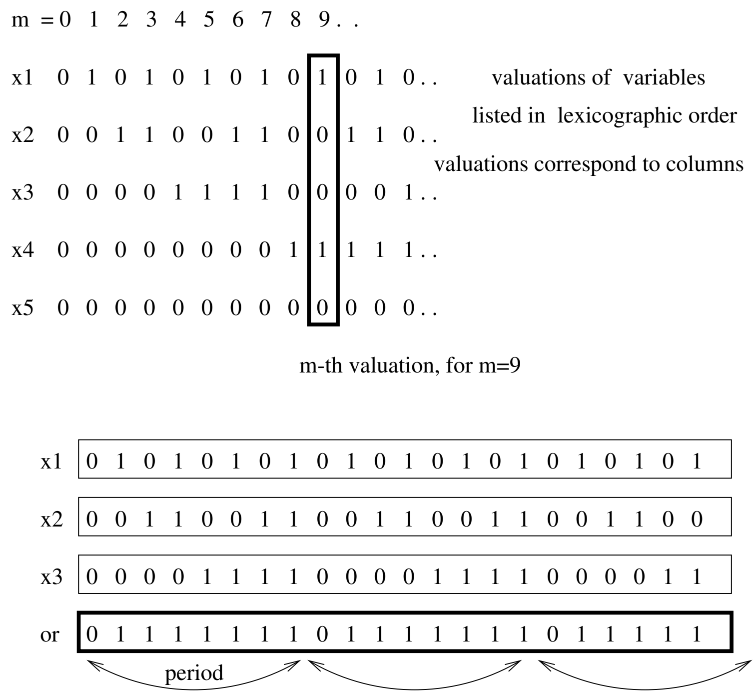


FIG. 4. The table of valuations of variables in lexicographic order and the construction of the k^{th} row of the array A, assuming that $C_k = (x_1 \vee x_2 \vee x_3)$. The 9^{th} valuation is 10010 since it is the binary representation of the number 9 read from the least significant digit. The sequence of valuations of $(x_1 \vee x_2 \vee x_3)$ has period 01111111 of size 8. Each row of A is well compressible (as highly periodic), so A is also well compressible as a (vertical) concatenation of only $O(n)$ rows.

To reduce this question to compressed pattern matching, we define a $k \times 2^n$ 2D-text A by $A_{i,x} = C_i(x)$. The above 3SAT question is now equivalent to the question of whether A contains a column consisting entirely of 1's. Because k, the column height, is the size of the 3SAT instance, we can use this column as the explicitly given text that will be searched for in the 2D-text A. To make it an instance of the compressed matching problem, we need to compute a small SLP $\mathcal{A}$ such that $A = \mathit{Decompress}(\mathcal{A})$.

We can find such an SLP with $O(kn)$ statements. For each clause C_i the i^{th} row of A, A_i, is defined solely by the clause C_i and the number of variables. We will show how to find an SLP $\mathcal{A}_i$ such that $A_i = \mathit{Decompress}(\mathcal{A}_i)$. Combining $\mathcal{A}_i$'s together and adding $k-1$ statements that stack the rows above each other yields the desired SLP.

To illustrate how $\mathcal{A}_i$ can be defined, we give the details of one example. Assume that, for some $p < q < r$, the clause C_i is $x_p \vee \neg x_q \vee x_r$. Suppose that an SLP $\mathcal{B}$ decompresses to a string of length 2^{r-1} that represents clause $x_p \vee \neg x_q$ viewed as a function of the first $r-1$ variables, or, equivalently, with arguments from $\{0, \ldots, 2^{r-1} - 1\}$, and that B is the variable defined by the last statement of $\mathcal{B}$. Let $\mathcal{E}_{r-1}$ be an SLP that decompresses to $1^{2^{r-1}}$. Then we can obtain an SLP A'_i that represents

$x_p \vee \neg x_q \vee x_r$ as a function of the first r variables by appending $\mathcal{B}$ with $A_i \leftarrow E_{r-1}B$. To obtain $\mathcal{A}_i$, we add $n-r$ statements of the form $A_i \leftarrow A_i \oslash A_I$.

The SLP $\mathcal{B}$ can be obtain similarly. Suppose that an SLP $\mathcal{D}$ decompresses to a string of length 2^{q-1} that represents the clause x_p as a function of the first $q-1$ variables. We obtain $\mathcal{B}$ by appending $\mathcal{D}$ with $B \leftarrow D \oslash E_{q-1}$ and $r-q-1$ statements $B \leftarrow B \oslash B$. One can see that this approach can be inductively extended to clauses of arbitrary length. We start our construction with statements $E_0 \leftarrow 1$, $E_1 \leftarrow E_0 \oslash E_0, \ldots, E_{r-1} \leftarrow E_{r-2} \oslash E_{r-2}$ and $Z \leftarrow 0$. Here $\mathcal{Z} = \{Z \leftarrow 0\}$ corresponds to the empty clause as a function of 0 variables, the basis of our inductive construction. ❐

5. FULLY COMPRESSED PATTERN CHECKING

The problem of fully compressed pattern checking at a given location is to check, given SLP's $\mathcal{P}$ and $\mathcal{R}$ and a pair (k,l), whether the pattern $P = \mathit{Decompress}(\mathcal{P})$ occurs within the text $R = \mathit{Decompress}(\mathcal{R})$ with its lower left corner at position (k,l).

We write SSP for the **Subset Sum Problem** (problem SP13 of [8]). An instance of co-SSP is a vector of positive integer weights $w = (w_0, \ldots, w_{n-1})$ and a target value t; the question is whether $w\overline{x} \neq t$ for all $x < 2^n$, where $w\overline{x}$ is an "inner product," that is, $\sum_{i=0}^{n-1} w_i \overline{x}_i$, where the product of an integer and a boolean is computed by first casting the boolean as the integer 0 or 1.

For a given vector w we define the *sum checking* 2D-text T^w that has $1 + \sum_{i=0}^{n-1} w_i$ rows and 2^n columns; the symbol $T^w_{x,i}$ is the truth value of "$w\overline{x} = i$".

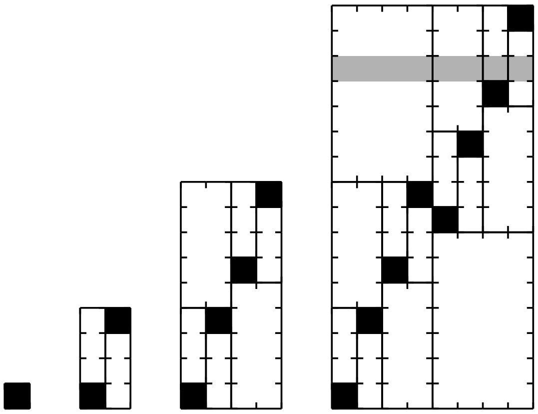


FIG. 5. The table T^w for the vectors $w = ()$, $w = (3)$, $w = (3,5)$ and $w = (3,5,7)$. The black squares indicate 1's. The gray band shows that $Z_{1,8}$ occurs in T^w at position $(0,13)$ and thus no subset sum of $\{3,5,7\}$ equals 13.

LEMMA 5.1. *There is an* SLP $\mathcal{T}$ *that describes* T^w.

Proof. Let $w(i) = (w_0, \ldots, w_{i-1})$. We recursively construct an SLP $\mathcal{T}_i$ that describes $T^{w(i)}$ (see Figure 5 for an example). $\mathcal{T} = \mathcal{T}_n$ is then the desired SLP.

Define $\mathcal{T}_0$ as $\{T_0 \leftarrow 1\}$ (this is the sum checking text for the empty vector). To show how to define $\mathcal{T}_i$ from $\mathcal{T}_{i-1}$ we examine how $T^{w(i)}$ can be constructed from $T^{w(i-1)}$. The left half of $T^{w(i)}$ corresponds to the subset sums of $\{w_0, \ldots, w_i\}$ in which w_i does not participate (because the respective binary digit of the column number is 0), so it is identical to $T^{w(i-1)}$ except that each column should be extended by w_i many 0's. Let $\mathcal{Z}_i$ be an SLP such that $\mathit{Decompress}(\mathcal{Z}_i)$ is a text consisting of w_i rows and 2^{i-1} columns, all filled with 0's. According to our observsation, the left half of $T^{w(i)}$

can be described by adding the statement $L_i \leftarrow Z_i \ominus T_i$. By similar reasoning, the right half can be described by adding $R_i \leftarrow T_i \ominus Z_i$. To obtain $\mathcal{T}_i$, we add $T_i \leftarrow L_i \oslash R_i$. ❐

THEOREM 5.1. *Fully compressed pattern checking for 2D-texts is co-**NP**-complete.*

Proof. To prove co-**NP**-hardness, we reduce co-SSP to the pattern-checking problem. Our co-SSP question is equivalent to the question of whether the t^{th} row of the sum checking text T^w consists entirely of 0's.

The SLP $\mathcal{Z}_{1,2^n}$ generates the pattern consisting of one row of 2^n 0's. Then $(t, 0)$ is the position where we should check for this pattern in T^w. By Lemma 5.1, we can compute an SLP $\mathcal{T}$ that describes T^w and has polynomial size. ❐

6. FULLY COMPRESSED TWO DIMENSIONAL PATTERN-MATCHING

Recall that the fully compressed matching problem is to determine, given two SLP's which are the compressed forms of a pattern and a text, whether the pattern occurs within the text. We prove that this problem is $\Sigma_2^{\mathbf{P}}$-complete. (See [16] for an explanation of the class $\Sigma_2^{\mathbf{P}}$.)

THEOREM 6.1. Fully compressed matching for 2D-texts is $\Sigma_2^{\mathbf{P}}$-complete.

Given a compressed pattern $\mathcal{P}$ and a compressed 2D-text $\mathcal{T}$, a positive answer to the fully compressed two dimensional pattern matching question is equivalent to the following:

$$(\exists i, j)(\forall k < \mathit{width}(\mathcal{P}))(\forall l < \mathit{height}(\mathcal{P}))\{\mathcal{P}_{k,l} = \mathcal{T}_{i+k,j+l}\}$$

By Lemma 4.1, the equality in this formula can be checked in polynomial time, hence the problem can be formulated in the normal form of $\Sigma_2^{\mathbf{P}}$ problems.

The proof of $\Sigma_2^{\mathbf{P}}$-hardness requires two lemmas.

LEMMA 6.1. *There exists a log-*SPACE *function f with $f(F) = (u, v, t)$ for any 3CNF formula F, where u and v are vectors of non-negative integers, t is an integer, and*

$$(\forall x < 2^n)\{F(\overline{x}) \equiv (\exists y < 2^m)\{u\overline{x} + v\overline{y} = t\}\}$$

where n is the number of variables of F, and vectors u and v have n and m coefficients, respectively.

Proof. Assume that F has a clauses with three literals, b clauses with two literals and c clauses with one literal. In our construction, the number of coefficients in v is $m = 7a + 3b$. We will describe all coefficients of u and v, as well as the threshold t, as vectors of $6a + 4b + 2c$ bits. Moreover, while describing a number d, we will view $\overline{d}$ as a concatenation $\overline{d}_1\overline{d}_2 \ldots \overline{d}_{a+b+c}$, where $\overline{d}_k$ is *the fragment of $\overline{d}$ corresponding to clause C_k.* The fragments corresponding to a clause with l literals will have length $2l$. We describe only the case of a clause with three literals in detail, as the other cases are similar, only simpler.

Assume that clause C_k contains three variables, x_h, x_i, x_j. Coefficients of u correspond to the variables of F. The fragments of $\overline{u}_h$, $\overline{u}_i$, and $\overline{u}_j$ corresponding to C_k are 000100, 000010 and 000001 respectively. The vector v contains seven coefficients corresponding to those truth assignments for x_h, x_i and x_j that satisfy clause C_k. There are seven such assignments; if v_g corresponds to the assignment $b_0b_1b_2$, then the fragment of $\overline{v}_g$ that corresponds to C_k is $000(1-b_0)(1-b_1)(1-b_2)$. The fragments of the coefficients of u and v that are not specified by these rules consists only of 0's. Finally, the fragment of $\overline{t}$ that corresponds to C_k is 100111.

Consider now $x < 2^n$ such that $F(\overline{v})$ is true. Then the fragment of $\overline{u\overline{x}}$ corresponding to a clause C_k is $000b_0b_1b_2$, where $b_0b_1b_2$ is a truth assignment satisfying C_k. (Note that x satisfies all the clauses of F.) Let v_l be the coefficient of v corresponding to this truth assignment. We set y_l to 1, and for each other l' that corresponds to another truth assignment for C_k, we set $y_{l'}$ to 0. The fragment of $\overline{v\overline{y}}$ that corresponds to C_k is the same as the respective fragment of $\overline{v_l}$. The fragment of $\overline{u\overline{x}+v\overline{y}}$ that corresponds to C_k is 100111, the same as the respective fragment of $\overline{t}$. Since this is true for every fragment of t, we have $u\overline{x}+v\overline{y}=t$.

Now suppose that there exists y such that $u\overline{x}+v\overline{y}=t$. If, for every clause C_k, exactly one of the entries corresponding to the truth assignments that satisfy C_k has coefficient 1 in the vector y, and if the addition is performed without carries, then each C_k is satisfied. It can be proved by induction that this is indeed the case. (Note that in our string representations of numbers we write the least significant bit first.) We leave the details to the reader.

Finally, the method of creating (u, v, t) is sufficiently regular to be done by a deterministic log-SPACE Turing machine. ❐

Define the Σ_2SSP problem as follows:

Instance: Vectors of positive integers u and v, of length m and n respectively, and integer t.
Question: Is $(\exists x < 2^m)(\forall y < 2^n)\{u\overline{x}+v\overline{y} \neq t\}$ true?

We will reduce Σ_2SSP to fully compressed pattern matching. This reduction will finish our proof because of the following lemma.

LEMMA 6.2. *The Σ_2SSP problem is $\Sigma_2^{\mathbf{P}}$-complete.*

Proof. Consider now an arbitrary property L of binary strings that belongs to $\Sigma_2^{\mathbf{P}}$. In its normal form, L is represented as

$$L(x) \quad \equiv \quad \exists y_1 \forall y_2 \; P(x, y_1, y_2)$$

where P is a polynomial time predicate. Because $\mathbf{P} \subseteq \mathbf{NP} \cap$ co-$\mathbf{NP}$, the predicate P can be represented as

$$P(x, y_1, y_2) \quad \equiv \quad \neg(\exists y_3 F(x, y_1, y_2, y_3))$$

where F is a 3CNF formula (computed using space which is logarithmic in the size of x in unary). Let "$\cdot$" denote concatenation of vectors. By the previous lemma,

$$F(x, y_1, y_2, y_3) \equiv \exists y_4 \; u(x \cdot y_1 \cdot y_2 \cdot y_3) + v y_4 = t$$

where (u, v, t) can be computed in logarithmic space from F. Define $\bar{u}, \bar{v}, \bar{w}$ and $\bar{t}$ so that $u(x \cdot y_1 \cdot y_2 \cdot y_3) + v y_3 = \bar{w}x + \bar{u}y_1\bar{v}(y_2 \cdot y_3 \cdot y_4)$ and $\bar{t} = t - \bar{w}x$.

By substitution and De Morgan's laws, we have

$$\begin{aligned} L(x) &\equiv \exists y_1 \forall y_2 \neg(\exists y_3 \exists y_4 \; u(x \cdot y_1 \cdot y_2 \cdot y_3) + v y_4 = t) \\ &\equiv \exists y_1 \forall y_2 \forall y_3 \forall y_4 \; u(x \cdot y_1 \cdot y_2 \cdot y_3) + v y_4 \neq t \\ &\equiv \exists y_1 \forall y_2 \forall y_3 \forall y_4 \; \bar{w}x + \bar{u}y_1\bar{v}(y_2 \cdot y_3 \cdot y_4) \neq t \end{aligned}$$

$$\equiv \quad \exists y_1 \forall (y_2 \cdot y_3 \cdot y_4)\ \bar{u} y_1 \bar{v} (y_2 \cdot y_3 \cdot y_4) \neq \bar{t}$$

In the last statement we can represent the 0/1 vectors y_1 and $y_2 \cdot y_3 \cdot y_4$ as some $\overline{x}$ and $\overline{y}$. In this way it becomes an instance of Σ_2SSP. Thus we have shown how L can be reduced to Σ_2SSP. ❐

To prove that fully compressed two dimensional pattern matching is $\Sigma_2^{\mathbf{P}}$-complete, it suffices to show how to translate an instance of Σ_2SSP into an instance of the fully compressed matching problem. Consider an instance of Σ_2SSP given by (u, v, t). Let $r = \sum_{i=0}^{m} u_i$ and $s = \sum_{i=0}^{n} v_i$. If $r > t$, then this instance can be answered in the positive, because we can use $x = 2^m - 1$: for every $y < 2^n$, $u\overline{x} + v\overline{y} = r + v\overline{y} > t$. The situation is similar if $s < t$, because then we can use $x = 0$: for every $y < 2^n$, $u\overline{x} + v\overline{y} = v\overline{y} \leq s < t$. Therefore, we may assume that $r \leq t \leq s$.

Recall the definition of the sum checking text T^w from our proof of Theorem 5.1: $T^w_{x,i} = 1$ if and only if $w\overline{x} = i$. We translate the formula

$$(\exists x < 2^m)(\forall y < 2^n)\{u\overline{x} + v\overline{y} \neq t\}$$

into an instance of the fully compressed matching problem.

Let U be the 2D-text T^u and let V be the 2D-text T^v with all columns reversed. Recall that $Z_{a,b}$ is the $a \times b$ two dimensionsal text consisting entirely of 0's. Recall, also, that the dimensions of U and V are $(r+1) \times 2^m$ and $(s+1) \times 2^n$, respectively.

The pattern P has a single row consisting of 1 followed by $2^m + 2^n$ zeros. Assume that we combine the SLP's that describe U, V and the necessary Z's. Then we can create the SLP $\mathcal{T}$ for our text by adding several statements that implement the following assignment:

$$T \leftarrow (Z_{t-r,2^m} \ominus U \ominus Z_{s-t,2^m}) \oslash V \oslash Z_{s+1,2^m}$$

Figure 6 shows the structure of the resulting text. Observe that we have three groups of colums in T: the first 2^m columns are the columns of U, padded on the top and the bottom with zeros. The next group consists of 2^n columns of V, and the third group consists of 2^m zero-columns. Because the pattern length is $2^m + 2^n + 1$, if it occurs in T then it must start in the first group, and end in the third group, hence it contains an entire row of V.

Suppose that a copy of the pattern occurs in T in row i. Because this pattern contains a 1 from row $i - (s - t)$ of $U = T^u$, we know that $i + t - s = u\overline{x}$ for some $x < 2^m$. Because this pattern contains entire i^{th} row of V, we know that in row number $s - i$ of T^v there are only 0's. Thus for every $y < 2^n$, $v\overline{y} \neq s - i$ and consequently $u\overline{x} + v\overline{y} \neq i + t - s + s - i = t$. Thus, our Σ_2SSP instance has a positive answer.

On the other hand, if we assume that the answer is positive, then there exists x that makes it true. By calculations simialr to those above, one can show that the pattern starts in position $(x, u\overline{x} + s - t)$. This concludes the proof of Theorem 6.1.

7. COMPRESSED PATTERN CHECKING

Recall that the compressed pattern checking problem is to check whether an uncompressed pattern P occurs at a position (x, y) of a 2D-text T given by an SLP $\mathcal{T}$. Let n be the size of $\mathcal{T}$ (the number

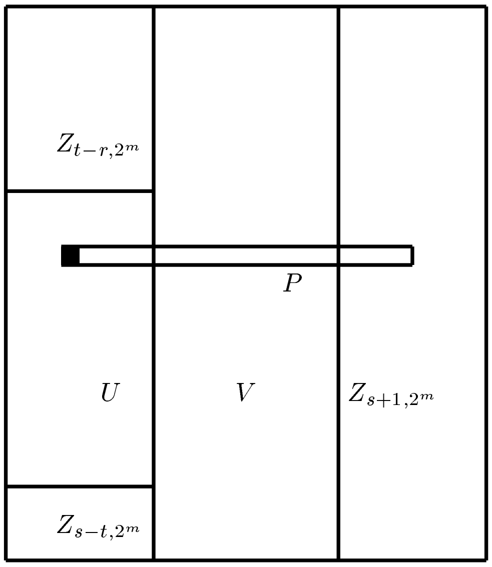


FIG. 6. Text T and a possible location of the pattern P.

of statements) and N be the size of T (the number of symbols). Denote by m the length of a longer side of P. The compressed pattern checking problem can be solved easily in polynomial time.

THEOREM 7.1. [naive algorithm]
There is an algorithm for the compressed pattern checking problem which works in $O(n|P|)$ time.

Proof. We use an algorithm for the point test problem $|P|$ times, for each symbol of P separately. Using this approach, we obtain, by Lemma 4.1, an algorithm for the compressed pattern checking problem which works in $O(n|P|)$ time. ❒

We improve this, first to $O(|P| + mn)$ which is better for not extremely elongated (for instance square) patterns, and then to $O(|P|+n+m\log N\log m)$, which is better if the text is not extremely well compressed, *i.e.*, if $\log N \log m = o(n)$.

In practical situations, if it happens that $N = O(n^c)$, where c is a constant, then $\log N = O(\log n)$ (clearly $\log m < \log N$ since pattern must be smaller than the text).

Combining these two algorithms we obtain an algorithm for compressed pattern checking problem which works in $O(|P| + n + m \times \min\{\log N \log m, n\})$ time.

THEOREM 7.2. [two improvements]
Let $m = \max\{width(P), height(P)\}$. Then the compreesed checking problem can be solved in:
(1) *$O(|P| + mn)$ time, which is $O(|P| + n\sqrt{|P|})$ for square patterns, or*
(2) *$O(|P| + n + (m\log N)(\log m))$ time, which is $O(|P| + n + m \cdot \log n \cdot \log m)$ for $N = n^{O(1)}$).*

We prove each point separately. The idea behind the algorithms is to consider point tests in groups, called *queries*. We identify V with the 2D-text which is generated by a variable V in the SLP $\mathcal{T}$. A *query* is a triple (V, p, R) where V is a variable in the SLP $\mathcal{T}$, p is a position inside V, and R is a subrectangle of the pattern P. This query is answered positively if V contains a subrectangle R' with lower left corner p that has the same content as R.

The rectangle R is represented by a pair (s, t) of positions of the lower left and upper right corners of R in P. Obviously, if $p + t - s$ is not a position in the text V, the query is answered negatively.

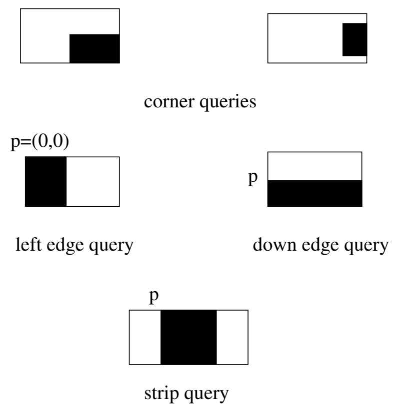


FIG. 7. Queries (V, p, R). Big rectangles represent rectangles $\mathcal{V}$. Black subrectangles represent rectangles R'. p is a position of a left upper corner of R' in $\mathcal{V}$.

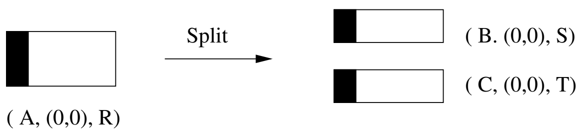


FIG. 8. Division and nondivision splits of a query. Here $A \leftarrow B \ominus C$. Note $R_1 = S \ominus T_1$ and $R_2 = T_2$.

A variable V is a *base* if it is defined in $\mathcal{T}$ with a statement of the form $V \leftarrow a$. In this case the answer to the query (V, p, R) is positive if and only if $p = (0, 0)$ and $R = [a]$. We can answer such a query in $O(1)$ time. If V is not a base, then it is defined with a statement of the form $V \leftarrow A \oslash B$ or $V \leftarrow A \ominus B$. In this case we reduce the answer to queries that involve V to the answers to queries that involve A and/or B. Observe that, in this case, A and B are defined in $\mathcal{T}$ before V.

The queries are divided into three classes: *strip queries*, *edge queries*, and *corner queries* (see Fig. 7). Let (V, p, R) be a query. Let R' be the rectangle of the same shape as R which is positioned at p in V. (V, p, R) is a *corner* query if R contains at least one corner of the pattern. (V, p, R) is an *edge* query if R' contains one side of V. There are four types of edge queries depending on which side of V is contained in R'. We call these *down*, *left*, *right* and *up* queries. A special case of an edge query is a *full* query. It is an edge query (V, p, R) such that $p = (0, 0)$ and R is of the same shape as V. (V, p, R) is a *strip* query if R is a strip (*i.e.*, a set of consecutive rows or columns) of the pattern, and R' is a strip of V.

```
Algorithm CHECKING
{ input: an SLP T, a pattern P and a position p
output: true iff P occurs at p in a text described by T }
begin
   V_1, V_2, ..., V_n ← consecutive variables in the SLP T;
   Q ← {(V_n, p, P)};
   for i ← n downto 1 do
      Q_0 ← {all V_i-queries from Q}, Q ← Q − Q_0;
      if V_i is base then
         for each q ∈ Q_0
            if answer to q is false then return false
      else
         for each q ∈ Q_0 do (*)
            Q ← Q ∪ Split(q)
   return true
end
```

Assume that $A \leftarrow B \oslash C$ or $A \leftarrow B \ominus C$ is an assignment for A in the SLP $\mathcal{T}$. The algorithm CHECKING for the checking problem uses a procedure $Split(q)$, where q is a query. The procedure $Split((A,p,R))$ replaces the A-query (A,p,R) with equivalent B-queries and C-queries, as illustrated in Figure 8. Let R' be a rectangle of the same shape as R positioned at p in $\mathcal{A}$. Then division of A into B and C, according to the assignment for A, causes either R' to be wholly contained in B or C, or to be divided into two smaller rectangles, one of which is in B and the other in C. In the latter case the split of a query is called a *division* of the query. Observe that a *Split* of a full query is a division.

The time complexity of the algorithm CHECKING is determined by the number of *Split* operations. In turn, they can be classified into those that lead to divisions, and the rest. The number of divisions is bounded as follows.

LEMMA 7.1. *The total number of all divisions of queries during the work of the algorithm is at most* $|P| - 1$.

Proof. The proof is a consequence of a well known fact that a full binary tree (in the sense each internal node is of degree 2) with s leaves has $s-1$ internal nodes. The structure of divisions is a tree. Each division corresponds to an internal node and there are $|P|$ leaves. Hence there are at most $|P|-1$ divisions. ❐

The number of nondivision splits depends heavily on the data structure which is used to represent the variable Q. If Q is represented by lists of V-queries, then $Split(q)$ and $Q \leftarrow Q \cup Split(q)$ can be implemented to work in constant time. Then the number of nondivision executions of *Split* can be bounded by $O(m \times n)$.

Proof of the first point of Theorem 7.2.

Proof. We show that the number of nondivision splits is $O(m \times n)$. Consider the set Q in some step of the algorithm. The pattern subrectangles which are in queries in Q are disjoint. It is sufficient to prove that the number of nonfull queries in Q is $O(m)$. The ones which are not full cover the boundary of P. Since the size of the boundary of P is $O(m)$, the number of nonfull queries is $O(m)$.

If Q is represented by lists of V-queries, the time complexity of the algorithm CHECKING is $O(|P| + mn)$. This completes the proof of the first point of Theorem 7.2. ❒

Now we pass to the proof of the second point of Theorem 7.2. We describe a data structure for Q which allows a time complexity of $O(|P| + n + m \log N \log m)$, which is better for texts which are not very highly compressed. Before we present the algorithm we need an auxiliary lemma.

LEMMA 7.2. *In each step of algorithm* CHECKING *the set* Q *contains at most four corner queries and* m *strip queries.*

Proof. At each stage of the algorithm, pattern subrectangles from queries cover some disjoint fragments of the pattern. Since there are four corners of the pattern, the number of corner queries is at most four. Strip queries contains either whole rows or columns of the pattern. Since the pattern subrectangles in queries of Q are disjoint, if there are strip queries in Q containing columns of the pattern, there are no strip queries containing rows of the pattern. Since there are at most m rows and columns in the pattern the result follows. ❒

For each type of query in Q we have a different data structure. All corner queries are stored in a list. For each type of edge query, and for each variable in the SLP $\mathcal{T}$, we have at most one edge query of this type which we call the *champion*. Strip queries are stored in 2-3 trees (see [1]), one for each variable of $\mathcal{T}$. The queries in these 2-3 trees are ordered according to their second parameter (the position p).

Initially, $Q = \{(V_n, p, P)\}$, so there is only one query, a corner query in Q. Hence, all lists for edge queries and all 2-3-trees are empty.

To explain our new approach for edge queries, it suffices to explain how to implement the line $Q \leftarrow Q \cup \mathit{Split}(q)$ *i.e.*, how we add edge queries to Q. Assume we want to add a V-query q to Q. If q is a left-edge V-query and there is no left-edge V-query champion in Q, then q becomes the left-edge V-query champion. If there is a left-edge V-query champion q' in Q, we use the following property.

OBSERVATION 7.3. *Let* R *and* R' *be pattern subrectangles contained in queries* q *and* q'*, respectively, where* R *is smaller than* R'*. Then, the answer for both* q *and* q' *is true if and only if the answer for* q' *is true and* R *occurs in* R' *at position* $(0,0)$.

The algorithm checks whether R occurs at $(0,0)$ in R'. If not, the algorithm stops and returns false. Otherwise, the new left-edge query champion is q', and q is removed.

Corner queries are handled in the same manner.

The algorithm deals with strip queries in groups *i.e.*, the loop (*) is performed for all strip V_i-queries in one step. The strip V_i-queries are stored in a 2-3 tree. Assume the assignment for V_i is $V_i \leftarrow B \ominus C$ (the approach for $V_i \leftarrow B \oslash C$ is similar). Then either the horizontal line dividing $\mathcal{B}$ and $\mathcal{C}$ in V_i divides all strip queries into strip B-queries and strip C-queries or it divides strip queries into two groups: a group of strip B-queries and a group of C-queries. In the first case we first split (which is a division) the V_i queries into B-queries and C-queries and then merge the resulting 2-3 trees with the 2-3 trees for B and for C. In the latter case we split the 2-3 tree for V_i into two parts. The first one is merged with the 2-3 tree for B and the second one with the 2-3 tree for C (**).

To perform this split correctly we modify the positions in C-queries while moving them from the V_i tree to the C tree. Instead of modifying the information in each element of the tree, we store, in each vertex, an offset which is the difference between a position in the query and the corresponding actual position. Initially, for each tree, there is only one offset. However since the trees will be split and joined, we must allow offsets at each vertex of the tree.

Recall that 2-3 trees provide operations *split* and *join* in $O(\log s)$ time, where s is the number of elements in the input trees. Suppose the strip queries in the algorithm contain rows. Merging two 2-3 trees T_1 and T_2 is done in the following recursive way. Let $q_1 = (V, (p_1, 0), R_1)$ and $q_2 = (V, (p_2, 0), R_2)$ be the first queries in T_1 and T_2, respectively. Assume $p_1 < p_2$, the other case being similar. First, we split T_1 into two 2-3 trees T' and T'', where T' contains queries $(V, (p, 0), R)$ with $p \leq p_2$ and T'' contains queries with $p > p_2$. Next, we recursively merge T'' with T_2, obtaining a tree T. Finally, we join the tree T' with T.

Observe that merging 2-3 trees may result in a large number of splits of 2-3 trees. Fortunately, the number of splits which are used in merges is $O(m \log N)$, as we can prove by arguments similar to those of [7].

LEMMA 7.3. *The number of splits of 2-3 trees in algorithm* CHECKING *is* $O(m \log N)$.

Proof. Suppose that only strip queries which contain rows of the pattern arise during execution of the algorithm. For a set of strip V-queries $\mathcal{S} = (V, (p_1, 0), R_1)$, $(V, (p_2, 0), R_2)$,..., $(V, (p_k, 0), R_k)$ with $p_1 \leq \ldots \leq p_k$, define a potential function:

$$F(\mathcal{S}) = 2 \log p_1 + 2 \log(width(V) - p_k) + 2 \textstyle\sum_{i=1}^{k-1} \log(p_{i+1} - p_i).$$

The potential is allocated to queries and left and right boundaries of V in the following way. The potential for the left boundary is $\log p_1$, the potential for the right boundary is $\log(width(V) - p_k)$, and the potential for a query $q = (V, (p_i, 0), R_i)$ is $\log(p_i - p_{i-1}) + \log(p_{i+1} - p_i)$, where $p_0 = 0$ and $p_{k+1} = width(V)$.

If we put a query $(V, (p, 0), R)$ between queries $q' = (V, (p_i, 0), R_i)$ and $q'' = (V, (p_{i+1}, 0), R_{i+1})$, then the interval $[p_i, p_{i+1}]$ is divided by p into two pieces, one of them smaller than $\frac{1}{2}(p_{i+1} - p_i)$. This means that the potential of either q' or q'' decreases by at least one. Similarly, the potential of either a boundary of V or of some query decreases by at least one if $p < p_1$ or $p > p_k$. This means, in turn, that merging two 2-3 trees decreases the potential function by at least the number of splits which occur during merging.

The potential function starts from 0 since there are no strip queries at the beginning of the algorithm. It increases when new strip queries are created by a division of a corner query or a division of a strip query.

Assume that an assignment for V in $\mathcal{T}$ is $V \leftarrow B \ominus C$. If a strip V-query is split into a strip B-query and a strip C-query, we assume that the B-query inherits the potential of the V-query and the C-query gets a new potential, which is the same as the potential of B, and which is smaller than $2 \log N$. Boundaries of $\mathcal{B}$ inherit the potential from boundaries of V and, if there are no strip C-queries, the boundaries of $\mathcal{C}$ get a potential which is at most $\log N$.

Assume that an assignment for V in the SLP $\mathcal{T}$ is $V \leftarrow B \oslash C$. In this case the B-queries and C-queries which are obtained by splitting strip V-queries inherit their potential from the corresponding V-query.

Since, during the work of the algorithm, at most m strip queries get a new potential, and the others inherit it, the potential function F can increase up to at most $4m \log N$. Hence, the total number of splits in merges of 2-3 trees is $4m \log N$. This completes the proof. ❒

The splits of 2-3 trees which are not used in merging 2-3 trees are the ones which occur in (*). The total number of those splits is bounded by n, so the total cost of them is $O(n \log m)$. We show now how to avoid this component in the time complexity function of CHECKING. Consider one such split. The boundary between $\mathcal{B}$ and $\mathcal{C}$ in V divides the interval between the last B=query and the first C-query into two parts causing the potential of one of the queries to decrease by 1, so this split is one of the $O(m \log N)$ splits from last lemma. This reasoning does not work if all V_i-queries are moved either to the tree for B or the tree for C. However, such a split can be done in constant time by modifying the offset in the root of the 2-3 tree for V_i.

Proof. (of the second point of Theorem 7.2) By Lemma 7.1, the total cost of all divisions of queries is $O(|P|)$. We compute total cost of splits which deals with edge queries. A new edge query is obtained by splitting either an edge query, a corner query, or a strip query. In the main loop of the algorithm, we deal only with V_i-queries. Since, by Lemma 7.2, there are at most four corner queries in Q and at most four edge V_i-queries which are champions, after splitting them we obtain at most 8 edge queries. Over all executions of the main loop, we obtain at most $8n$ edge queries from edge queries and corner queries. From all strip queries we may obtain m edge queries, so the total cost of dealing with edge queries is $O(n + m)$. The total cost of dealing with strip queries is determined by the total number of splits of 2-3 trees, which, by Lemma 7.3, is $O(m \log N \log m)$. This completes the proof. ❒

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.
2. A. Amir, G. Benson and M. Farach, Let sleeping files lie: pattern-matching in Z-compressed files. *Journal of Computer and System Sciences*, 1996, Vol. 52, No. 2, pp. 299-307
3. A. Amir, G. Benson, Efficient two dimensional compressed matching, *Proceedings of the* 2^{nd} *IEEE Data Compression Conference*, pp. 279–288 (1992).
4. A. Amir, G. Benson and M. Farach, Optimal two-dimensional compressed matching, *Journal of Algorithms*, 24(2), pp. 354-379, August 1997
5. P. Berman, M. Karpinski, L. L. Larmore, W. Plandowski, and W. W. Rytter, On the Complexity of Pattern Matching for Highly Compressed Two-Dimensional Texts, *Proceedings of the* 8^{th} *Annual Symposium on Combinatorial Pattern Matching* (CPM'97), Aarhus, Denmark, June 30 – July 2, 1997, LNCS **1264**, Edited by A. Apostolico and J. Hein, Springer-Verlag (1997), pp. 40–51.
6. M. Crochemore and W. Rytter, *Text Algorithms,* Oxford University Press, New York (1994).
7. M. Farach and M. Thorup, String matching in Lempel-Ziv compressed strings, *Proceedings of the* 27^{th} *Annual Symposium on the Theory of Computing* (1995), pp. 703–712.
8. M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness,* Freeman, New York (1979).
9. L. Gąsieniec, M. Karpinski, W. Plandowski and W. Rytter, Efficient algorithms for Lempel-Ziv encoding, *Proceedings of the* 5^{th} *Scandinavian Workshop on Algorithm Theory*. Springer-Verlag (1996).
10. M. Karpinski, W. Rytter and A. Shinohara, Pattern-matching for strings with short description, *Nordic Journal of Computing*, 4(2):172-186, 1997
11. D. Knuth, *The Art of Computing, Vol. II: Seminumerical Algorithms. Second edition.* Addison-Wesley (1981).
12. A. Lempel and J. Ziv, On the Complexity of finite sequences, *IEEE Trans. on Inf. Theory* **22** (1976) pp. 75–81.
13. A. Lempel and J. Ziv, Compression of two-dimensional images sequences, *Combinatorial Algorithms on Words* (ed. A. Apostolico, Z. Galil), Springer-Verlag (1985) pp. 141–156.
14. O. H. Ibarra and S. Moran, Probabilistic algorithms for deciding equivalence of straight-line programs, *JACM* **30** (1983), pp. 217–228.

15. R. Motwani, P. Raghavan, *Randomized Algorithms*, Cambridge University Press (1995).
16. Ch. H. Papadimitriou, *Computational Complexity*, Addison Wesley (1994).
17. W. Plandowski, Testing equivalence of morphisms on context-free languages, *Proceedings of the* 2nd *Annual European Symposium on Algorithms* (ESA'94), LNCS 855, Springer-Verlag (1994), pp. 460–470.
18. J. Schwartz, Fast probabilistic algorithms for verification of polynomial identities, *J. ACM* **27** (1980) pp. 701–717.
19. J. Storer, *Data Compression: Methods and Theory,* Computer Science Press (1988).
20. R. E. Zippel, Probabilistic algorithms for sparse polynomials, *Proceedings of the International Symposium on Symbolic and Algebraic Manipulation* (EUROSAM '79) LNCS 72, Springer-Verlag (1979), pp. 216–226.
21. J. Ziv and A. Lempel, A Universal algorithm for sequential data compression, *IEEE Transactions on Information Theory* **IT–23** (1977), pp. 337–343.