Almost optimal fully LZW-compressed pattern matching

Leszek Gąsieniec [*] Wojciech Rytter [†]

Abstract

Given two strings: pattern $\mathcal{P}$ and text $\mathcal{T}$ of lengths $|\mathcal{P}| = M$ and $|\mathcal{T}| = N$. A *string matching* problem is to find all occurrences of pattern $\mathcal{P}$ in text $\mathcal{T}$. A *fully compressed string matching* problem is the string matching problem with input strings $\mathcal{P}$ and $\mathcal{T}$ given in compressed forms p and t respectively, where $|p| = m$ and $|t| = n$. We present first, almost optimal, string matching algorithms for LZW-compressed strings running in:

1. $O((n+m)\log(n+m))$-time on a single processor machine, and
2. $\widetilde{O}(n+m)$ work on a $(n+m)$-processor PRAM[1].

Techniques used in our paper can be used in design of efficient algorithms for a wide range of the most typical string problems, in the compressed LZW setting, including: computing a period of a word, finding repetitions, symmetries, counting subwords, and multi-pattern matching.

1 Introduction

In the recent years compressed string matching has been the one of the most intensively studied variant of the string matching problem. The ongoing research has been naturally developed in two directions: *compressed string matching* (CSM for short), i.e. when only text string is compressed and the pattern is given explicitly, see [2, 13], and *fully compressed string matching* (FCSM) when both pattern and text strings are compressed, see e.g. [14, 15]. Although it may seem that CSM stream has more practical motivation however fully compressed pattern matching appears to be much more interesting from the point of view of computation theory, e.g. see [14, 15, 4, 22]. Moreover, techniques developed in FCSM (including this paper) give firm foundations to algorithms dealing with other string problems including: computing periodicities, symmetries, repetitions, counting subwords and multi-pattern matching (for respective definitions check [8]), where all processed strings are available only in compressed form. In case where the techniques developed for CSM are designed to deal only with the string matching problem.

[*]Department of Computer Science, University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, United Kingdom, e-mail: `leszek@csc.liv.ac.uk`. Research supported in part by NUF-NAL (The Nuffield Foundation Awards to Newly Appointed Lecturers) award.

[†]Department of Computer Science, University of Liverpool, Chadwick Building, Peach Street, Liverpool L69 7ZF, United Kingdom and Institute of Informatics, Warsaw University, Banacha 2, Warszawa 02–097, Poland, e-mail: `rytter@csc.liv.ac.uk`, partially supported by the grant KBN 8T11C03915.

[1]Notation $\widetilde{O}(g(k))$ stands for $O(g(k)\log^c(k))$, for some constant $c \in \mathcal{R}^+$

There are two compression techniques for which string matching algorithms were developed, i.e. LZ-compression due to Lempel and Ziv [20, 25] and its derivate (simplification) called LZW-compression, introduced later by Welch [24]. The CSM problem has almost optimal solutions for both LZW- and LZ-compression, see [2], [13], also Figure 1. However problems in class FCSM appear to be much harder. The most dramatic change is due to the lack of processing time and space. All efficient string matching algorithms spend a great effort when the pattern is processed. The information obtained from the pattern processing (like pattern periodicities, failure function, witnesses, and naming, see [8]) allows later to reduce the work significantly when the actual text search is performed. Unlikely to CSM class all algorithms for FCSM problems have to deal with compressed patterns. Since *efficient string matching algorithm* is the one with a linear complexity in the size of its input strings thus in FCSM setting the algorithm must have a shorter processing time as well as a smaller space than its CSM counterpart. Some extensive work for LZ-compressed strings in FCSM class has been given in [14, 15, 4]. Unfortunately the highly polynomial time complexity of only known algorithms is still quite far from desired optimal solution (the only known lower bound is the size of the input of the compressed string). Although there are known algorithms for FCSM and LZ-compressed texts (they have application for highly compressed strings [14]) however they can not be used in case of LZW-compression, where the expected compression is less effective. In this paper we give the first and almost optimal algorithm for FCSM class with LZW-compressed strings. Our algorithm is superior to the one described in [2] since we do not decompress the pattern if such decompression would dominate the complexity of the whole algorithm. Moreover we deliver the first efficient parallel implementation of any CSM/FCSM algorithm. The parallel algorithm is the combination of matching techniques used earlier in CSM class with LZW-compressed strings [2] and in parallel pattern matching as *witnesses*, *duels* [23] and *pseudo-period*, see [5, 7].

The rest of the paper is organized as follows. In section 2 we give definitions and in section 3 we introduce techniques and data structures used later in the paper. Section 4 contains description of both: sequential and parallel FCSM algorithms. Finally, in section 5, we conclude with a brief summary of known CSM/FCSM results.

2 Definitions

Let Σ be a set of ordered symbols, called later an *alphabet*. A *string* (word, text) is a finite sequence of consecutive symbols drawn from alphabet Σ. The set of all strings is denoted by Σ^*. Let $w \in \Sigma^*$. Notation $|w|$ stands for the length of string w, and w_i denotes ith symbol of word w. Enumeration in strings start from position 0, i.e. $w = w_0 w_1 w_2 .. w_{k-1}$, where $k = |w|$. Any initial segment of the word is called its *prefix* and final segment its *suffix*. We say that word w has a period r iff $w_i = w_{i+r}$, for $i = 0, .., |w| - r$. In case $r < |w|$ is not a period of word w, there is a position $0 < j \leq |w| - 1$ such that $w_{j-r} \neq w_j$. Position j is called a *witness* against periodicity r in word w. Word w is called *periodic* if it has a period r such that $r \leq \frac{|w|}{2}$, otherwise w is called *non-periodic*. Word w is called *free* if it is given in uncompressed form.

Perhaps the most studied text compression scheme based on subword repetition was introduced in early 70-ties by Lempel and Ziv, for details see [20]. A very interesting and successful modification of LZ-compression was proposed by Welch few years later in [24]. The main difference between both techniques is in choice of *code words*. Unlikely to LZ-compression, where a code word can be potentially any subword of already processed prefix, in the LZW setting all code words of input string w can be stored in a trie T_w [6], search tree, in which all edges are labelled with alphabet symbols. In the trie each codeword is represented by a path from the root to some other node of a trie. All code words are distinct (but possibly the last one) and every node (internal as well as a leaf) identifies one code word of the compression code. In general LZW-compression is less efficient (gives weaker compression) but it is much easier to implement than its LZ counterpart, e.g. see compression/decompression code in [11]. On the other hand LZW-compression is non-trivial in parallel setting since computing LZ-compression belongs to NC, see [8], while the problem of computing LZW-compression is P-complete, see [12].

3 Tools

Our FCSM algorithm needs data structures supporting efficient comparisons of single symbols as well as subwords of compressed pattern and text.

Naming technique and factor comparisons. Recall that compressed forms p and t are the sequences of *code words*, where each single code word is represented by a path in the *trie* $T_{\mathcal{P}}$ and $T_{\mathcal{T}}$ respectively. The crucial idea in pattern matching for LZW compressed texts is to treat full and partial code words as a single comparison elements. By a *factor* we mean any subword of a code word. Our algorithm needs a data structure which compares equality of any two factors efficiently. For this purpose we use *naming technique*, e.g. see [8], chapter 9. Given set of code words $S = \{s_1, s_2, s_3, ..\}$. A *basic factor* in S is a factor of a length being a power of 2. The main purpose of the naming technique is to give consistent names to all basic factors in S. Having all basic factors named, each equality test of arbitrary two factors can be checked by decomposing these factors into two (possibly overlapping) basic factors. The advantage of this technique follows from the fact that there is only logarithmic number of powers of two, and one gets handy data structure for string comparison. with only logarithmic time/space overhead. We create the *dictionary of basic factors* of pattern and text code words (dbf $\mathcal{D}$, in short) similarly as in [8], chapter 9.

Lemma 1 *The dictionary $\mathcal{D}$ of basic factors can be computed sequentially in time $O((n+m)\log(n+m))$ or with $\widetilde{O}(n+m)$ work on $n+m$ processor PRAM. Having dictionary $\mathcal{D}$ ready, the equality of arbitrary factors can be tested in logarithmic time.*

Instant access to symbols. The parallel algorithm needs an access to single pattern/text symbols, without actual decompression. We process tries $T_{\mathcal{T}}$ and $T_{\mathcal{P}}$ such that all symbol queries are served in logarithmic time. Assume that each node in a trie has a *reverse link* to its parent. The work is done by creating logarithmic number of additional links by doubling reverse links in logarithmic number of rounds. A separate tables $A_{\mathcal{P}}$ and $A_{\mathcal{T}}$ holding ends of code words in pattern $\mathcal{P}$ and text $\mathcal{T}$ respectively are computed as well. The query for the ith symbol of the pattern/text is performed as follows. During the first step a code word containing desired symbol is found in logarithmic time applying binary search in table $A_{\mathcal{P}}/A_{\mathcal{T}}$. Then the proper symbol is found on the way from the node representing the end of a code word towards the root in logarithmic time with a help of additional reverse links created while doubling.

Queries involving uncompressed subpatterns. Another data structure is a border tree used previously in [2]. We use it to answer the queries Q (defined below) for short strings or for strings with short periods. Let α be a prefix and B an arbitrary factor of pattern $\mathcal{P}$. The *query* (function) $Q(\alpha, B)$ gives the length of the longest prefix of pattern $\mathcal{P}$ which is a suffix of αB, see [2]. We use the following result.

Lemma 2 *[2] For a free pattern query Q can be implemented in logarithmic time.*

Lemma 3 *Let α be a free prefix of pattern $\mathcal{P}$ whose period is smaller than n, and B be an arbitrary factor of $\mathcal{P}$. We can preprocess the pattern and the text in time $O((n+m)\cdot\log(n+m))$ such that query $Q(\alpha, B)$ can be answered in logarithmic time.*

Proof: If α is short (has length at most $2n$) then we can decompress it and use Lemma 3. Otherwise $\alpha = v^k u$, where $k \geq 2$, v is a period of α and u is a prefix of v. Let $P' = v^2 u$. We can decompress P' and then implement all queries Q with respect to P' in logarithmic time, due to Lemma 2. In the preprocessing we compute which factors have period $|v|$, and the position of the first mismatch in each non-periodic factor. Then query $Q(\alpha, B)$ is implemented as follows. If B has period $|v|$ then check whether the period of B is the continuation of the period of α. If it is, then the output consists of αB. Otherwise we look for the first mismatch in B with respect to the periodicity $|v|$. The pattern is shifted in such a way that the first mismatch breaking periodicity in $\mathcal{P}$ matches the first mismatch in B. We use similar method to one used in [18]. If there is no (long) match of the type above then we know that the output is small and we return the value $Q(P', B)$ which can be computed in logarithmic time due to Lemma 3, since P' is short. $\square$

4 Algorithm

In this section we present two efficient FCSM algorithms. The first algorithm is sequential and it is based on the concept of classical Knuth-Morris-Pratt algorithm [19]. The second algorithm is parallel and it combines known parallel matching techniques with efficient methods used for matching compressed strings. Both algorithms have two phases: *preprocessing* – preparation of compressed pattern and text, and the actual search, called later the *searching* phase.

4.1 Sequential approach

In the algorithm we use comparisons of factors (subwords of code words) against arbitrary but the same length segments (possibly consisting of many code-words) of pattern $\mathcal{P}$ and text $\mathcal{T}$. We call such comparisons as *atomic comparisons*. Our sequential algorithm uses only linear number of atomic comparisons. The amortized work per single atomic comparison is logarithmic due to Lemma 4.

Lemma 4 *A sequence of $O(n+m)$ atomic comparisons can be executed in $O((n+m)\log(n+m))$-time.*

Proof: Each time a factor B is compared successfully against a segment S (possibly composed of several code-words) of text or pattern we record the longest subsegment S' of S consisting of full code-words as a subword of B. It is possible since dictionary $\mathcal{D}$ stores the name for S' as well.

Let $S = b_0, b_1, \ldots b_k$, where $b_1, \ldots b_{k-1}$ are consecutive code-words in $\mathcal{P}$ or $\mathcal{T}$ (called *internal* with respect to a considered atomic comparison) and b_0, b_k are factors then for each factor b_i, $1 \leq i < k$ we store three items: the name of factor B, the initial position of B where B is matched against $b_i \ldots b_k$ and the length of the match $r = |S| - |b_0 \ldots b_{i-1}|$.

The code-words which are internal with respect to some atomic comparison are not involved later *actively* in any atomic comparison as a part of a segment S, since we refer directly to the recorded part of B and use the dictionary of basic factors for B to omit explicitly scanning of these internal code-words again. $\square$

The algorithm uses a restricted version of the *failure table* F, see [8].

$$F(i) = \max\{-1 \leq t < i \ : \ \mathcal{P}_0 \ldots \mathcal{P}_t \text{ is a suffix of } \mathcal{P}_0 \ldots \mathcal{P}_i\}.$$

We call end-positions of code words *special positions*. The restricted failure table *Restricted_F* stores only the values of F for special positions i and for positions $i \leq n$. Observe that the size of *Restricted_F* is $O(n)$ while the size of F would be quadratic in terms of input size.

4.1.1 preprocessing

Assume that the whole pattern has no period smaller than n. The preprocessing phase consists in computation of table *Restricted_F*. Let k be the size of the longest prefix of $\mathcal{P}$ whose period is smaller than n.

By an *elementary operation* we mean an atomic comparison or a single query Q.

Lemma 5 *Assume we have a recursive function $\mathcal{F}$, such that in each instance at a single level of recursion we make $O(s+1)$ elementary operations, where s is the number of recursive calls at this level. Then the total number of elementary operations when computing $\mathcal{F}(t)$ is $O(r+1)$, where r is the total number of recursive calls (including calls at deeper levels).*

Proof: We can assume without loss of generality that we make at most $s+1$ elementary operations at a given level. We show that the total number of elementary operations is bounded by $2r+1$.
Assume in these s recursive calls we make $r_1, r_2, \ldots, r_s$ recursive calls (including calls at deeper levels). By an inductive argument the total number of elementary operations in these s calls is bounded by $\Sigma_{i=1}^{s}(2r_i+1)$. Observe that $\Sigma_{i=1}^{s} r_i = r-s$, hence the total number (including $s+1$ operations at a given level) is bounded by:

$$\Sigma_{i=1}^{s}(2r_i+1) \;+\; s+1 \;=\; 2(r-s)+2s+1 \;=\; 2r+1.$$

This completes the proof. □

We construct the following function *Recursively_Compute_F* which computes F for positions (not necessarily special ones) if the values of *Restricted_F* for preceding special positions and positions smaller than n are known. It is easier to describe the function in a recursive way.

function *Recursively_Compute_F*(i)
$\{i$ is not necessarily a special position in $\mathcal{P}$, the values of *Restricted_F* for preceding *special positions* and positions smaller than n are known$\}$

1. j = largest special position less than i
2. t := *Restricted_F*$[j]$ (in constant time, since it is already computed).
3. B := $\mathcal{P}[j \ldots i]$ {Observe that B is a factor (a part of a code-word).}
4. **repeat**
   begin
   (a) If $t < n$ **then return** *Restricted_F*$[t]$
   (b) If $t \leq k$ **then return** $Q(\mathcal{P}[0 \ldots t], B)$
   {it can be done in logarithmic time due to Lemma 3 and Lemma 2}
   (c) If $t > k$ **and** the prefix $\mathcal{P}[0 \ldots t]$ extends in $\mathcal{P}$ by the factor B
   {use a single *atomic comparison* to check it} **then**
   return $t+i-j$
   (d) Otherwise compute
   { **recursive call**: } t := *Recursively_Compute_F*(t)
   { t decreases at least by n}
   end

Lemma 6 *[preprocessing] The failure table for special positions in pattern $\mathcal{P}$ can be computed sequentially in time $O((n+m)\log(n+m))$.*

Proof: It is enough to show that the table *Restricted_F* can be computed using linear number of elementary operations (atomic comparisons or queries of type Q).

Let $i_1, i_2, \ldots, i_h$ be the consecutive special positions in P, we compute the table *Restricted_F* by the following algorithm:

compute $Restricted\_F[i_1]$ using the algorithm from Lemma 3;
for s:=2 **to** h **do**
$Restricted\_F[i_s] := Recursively\_Compute\_F(i_s)$;

Due to Lemma 5 the total number of elementary operations needed to compute the value $Restricted\_F[i_s]$ is proportional to the number of recursive calls, and in each recursive call the difference between $Restricted\_F[i_{s-1}]$ and $Restricted\_F[i_s]$ increases by at least n. Observe that $i > k \Rightarrow i - F(i) > n$.
In each recursive call (step (d) in the recursive description) we have

$$t - Recursively\_Compute\_F(t) \geq n$$

since $\mathcal{P}[0..t]$ has no period shorter than n (otherwise we terminate in one of the steps (a), (b) or (c). We have (observe that $h \leq m$)

$$\sum_{s=2}^{h}\left(\frac{Restricted\_F[i_{s-1}] - Restricted\_F[i_s]}{n} + 1\right) = O(m).$$

Hence the total number of elementary operations is linear. □

Corollary 7 *The smallest period of any string with LZW-compressed representation of size m can be computed in time $O(m \log m)$.*

Proof: The value of restricted failure table for the last position i_h (which is a special position also) gives the required result, since the period of the text $\mathcal{P}$ equals $m - Restricted\_F[i_h]$. □

4.1.2 searching

The searching algorithm behaves like its KMP counterpart applied at special positions in text $\mathcal{T}$. Recall that KMP algorithm iterates the failure table look-ups, checking if currently computed prefix of $\mathcal{P}$ extends by a single symbol in $\mathcal{T}$. We can concatenate the pattern $\mathcal{P}$ and the text $\mathcal{T}$ getting a new text $P' = \mathcal{P}\#\mathcal{T}$. Then we compute the table *Restricted_F* for P'.

The algorithm checks if prefix can be extended by a single segment-factor using the table *Restricted_F*. $\mathcal{P}$ occurs in $\mathcal{T}$ iff there is a special position i in P' lying to the right of '#' such that $\mathcal{P}$ occurs in the text $\mathcal{P}[0 \ldots t] \cdot B$ where $t = Restricted\_F[i]$ and B is the code-word in P' following the position i. If the prefix of size n of $\mathcal{P}$ is nonperiodic then a single atomic comparison is sufficient, otherwise we proceed similarly as in Lemma 3. Altogether $O(n+m)$ atomic comparisons are done, since $|T|/n = O(n)$ and the costs are amortised due to Lemma 6. This proves our first main result.

Theorem 8 *[sequential complexity]*
The fully compressed string-matching for LZW-compressed strings can be solved in sequential time $O((n+m)\log(n+m))$.

4.2 Parallel approach

In this section we present efficient parallel FCSM algorithm for LZW-compressed strings. Unlikely to standard parallel string matching algorithms every approach in parallel CSM/FCSM has to struggle with two crucial obstacles, i.e.

1. how to reduce the number of pattern occurrences from N to $O(\frac{N}{M})$, when one must perform reduction with only $\widetilde{O}(n+m) << N$ work, and
2. how to check remaining $O(\frac{N}{M})$ candidates efficiently, again with $\widetilde{O}(n+m) << N$ work.

The answer to the first question is given in Stage 1 of a *searching* phase and the answer to the second question is given in the proof of Lemma 12. We assume here that (the length of the pattern) $M > n$ as well as all pattern prefixes of length $2^k < n$ are non-periodic. The algorithm for shorter patterns as well as the periodic case will appear in full version of the paper. In the rest of this chapter we say that computation is efficient when it is performed on $(n+m)$-processor PRAM in polylogarithmic time.

4.2.1 preprocessing

Preprocessing of pattern $\mathcal{P}$ and text $\mathcal{T}$ is performed in 5 consecutive steps.

1. Construction of dictionary D (naming technique) extended later by additional reverse links (instant symbol access), see section 3.
2. Decompression of pattern prefix π of length n, the ith symbol of π is recovered by ith processor by instant access to pattern/text symbols, section 3
3. Find all full occurrences $o_0, o_1, .., o_k$ of prefix π in pattern $\mathcal{P}$ using searching being a simpler version (prefix π is uncompressed and $M - reduction$ is based only on duels by witnesses of prefix π) of the algorithm described later in section 4.2.2.
4. Form pseudo-pattern q^p such that $q^p_i = o_{i+1} - o_i$, for $i = 0, .., k-1$. Process word q^p as a pattern by any optimal parallel string matching algorithm, e.g. use [23]. The period q of word q^p is called a *pseudo-period of pattern* $\mathcal{P}$.
5. Compute witnesses in pattern $\mathcal{P}$ against multiples of pseudo-period q. Use Vishkin's parallel witnesses computation [23] supported by efficient factor testing (naming technique, section 3).

Theorem 9 *Each step of the preprocessing can be performed efficiently.*

Proof: Step 1 and 2 are done in logarithmic time on $n + m$ processors, see 3. Step 3 is performed in $O(\log^2 m)$-time on m processors, see section 4.2.2. Step 4 is performed optimally in logarithmic time, see [5, 7]. Step 5 is done in $O(\log^2 m)$-time on m processors. □

4.2.2 searching

The searching phase is divided into two stages, each dealing with one of the obstacles mentioned in the beginning of section 4. Operation of reduction of number of candidates such that only one candidate survives in factor of length c is called c-reduction. The purpose of the first stage is n-reduction.

Stage 1: *n-reduction*

Lemma 10 *The number of occurrences of prefix π starting in each code word w is initially reduced to $O(\log m)$ by application of a naming technique.*

Proof: Divide word w into logarithmic disjoint factors $b_0, b_1, .., b_k$ such that k is the largest power of two not exceeding $|w|$. Let b_0 be a prefix of w of length $|w| - 2^k$. Let b_1 stands for the first half of the suffix of length 2^k, b_2 for the first half of the suffix of length 2^{k-1}, etc. In each factor b_i can start at most 2 (full in terms of code word w) occurrences π prefix of length 2^{k-i}, otherwise such a prefix would be periodic, which contradicts our assumption, see section 7. □

The number of candidates is further reduced one by one by duels to only one candidate in each code word. All duels are based on witnesses computed for the free prefix π. Instant access to single text symbols, required in each duel, is provided by data structure described in section 3.

Lemma 11 *Stage 1 can be performed efficiently and afterwards there is at most one candidate left in every code word of text $\mathcal{T}$.*

In the proof of the following Lemma we introduce new type of duels which are based on rather comparison of respective text segments then comparison of pattern and text symbols used earlier in parallel string matching [23, 7].

Lemma 12 *Following c-reduction, for any $c \geq n$, all occurrences of any pattern prefix of length $\leq 2c$ can be found efficiently.*

Proof: All prefix occurrences are found in logarithmic number of rounds. In each round candidates are grouped in pairs, and in each pair c_1, c_2 a duel, based on the text segements B_1, B_2 (sequences of code words) corresponding to paired candidates c_1, c_2, is performed. Equality of segments B_1, B_2 is verified by comparison of respective (full or partial) text code words efficiently as follows. Lists of ends of code words in B_1 and B_2 are merged and then corresponding smaller text factors are compared with a help of basic factors kept in dictionary $\mathcal{D}$. Due to c-reduction, total work in all duelling pairs is linear in n. When text segments B_1, B_2 are consistent candidates c_1, c_2 fall into the same *equivalence class*, and one of them is chosen (arbitrarily) as a *leader* of the new class. The leader represents the whole class in the next round duel. Otherwise, when segments B_1, B_2 differ, respective two text code words and relative position l on which they differ is found by the test of logarithmic number of pairs of basic factors in dictionary D. When the relative position l is found the lth symbol of pattern prefix is compared with positions $c_1 + l$ and $c_2 + l$ in the text. The result of at least one of these tests must be negative (duels property) so at least one of the classes is withdrawn.

After logarithmic number of rounds at most one class survives and its leader is compared (code word by code word) with the pattern prefix of corresponding size. If prefix fits, the class defines all occurrences of desired prefix. Otherwise there is no pattern prefix occurrence in the text. □

Further $M/2$-reduction is performed in 3 steps as follows.

Stage 2: *$M/2$-reduction*

1. Find all occurrences of prefix π in text $\mathcal{T}$ applying Lemma 12.
2. Let q^t be the string which contains distances between occurrences of prefix π in text $\mathcal{T}$. String q^t is called *pseudo-text*. Find all occurrences of pseudo-pattern q^p (see preprocessing, step 4) in pseudo-text q^t. Use any optimal pattern matching algorithm, e.g. [10].
3. If pseudo-pattern q_p is periodic reduce the number of candidates by duels using witnesses against multiples of pseudo-period (preprocessing, step 5).

All 3 steps are performed efficiently and afterwards $M/2$-reduction is completed.

Theorem 13 *All occurrences of patter $\mathcal{P}$ in text $\mathcal{T}$ can be found efficiently.*

Proof: After $M/2$-reduction is completed, due to Lemma 12 remaining candidates can be verified efficiently. □

The main result follows from Theorem 8 and Theorem 13.

Theorem 14 *The fully compressed string matching problem, in which both pattern $\mathcal{P}$ and text $\mathcal{T}$ are given in a LZW-compressed forms p and t respectively, can be solved in time:*

1. *$O((n+m)\log(n+m))$ on a single processor machine, and with*
2. *$\widetilde{O}(n+m)$ work on a $(n+m)$-processor PRAM,*

where $n = |t|$ and $m = |p|$.

Compression\Class	CSM	FCSM
LZW-compression	$O(n \log M + M)$ [2]	$\tilde{O}(n+m)$ Here
LZ-compression	$O(n \log N + M)$ [13]	$O((n+m)^5)$ [15]

Figure 1: Results in CSM/FCSM class

5 Conclusion

In this paper we contribute to the *fully compressed string matching problem*, delivering first, *almost optimal* sequential as well as parallel algorithm for the case when both the pattern and the text are given in LZW-compressed forms (see table 1). Techniques used in our paper are not only limited to compressed pattern matching but they can be applied also in efficient algorithms for a wide range of other string problems, including: computing a period of an LZW-compressed text, finding repetitions, symmetries, counting subwords, and multi-pattern matching.

References

[1] A. Amir and G. Benson, Efficient two dimensional compressed matching, *Proc. of 2nd IEEE Data Compression Conference*, pp. 279–288, March 1992.

[2] A. Amir, G. Benson, and M. Farach, Let sleeping files lie: Pattern matching in Z-compressed files, *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1994.

[3] A. Amir, G. Benson, and M. Farach, Optimal two-dimensional compressed matching, *Proc. of 21st International Colloquium on Automata, Languages, and Programming*, July 1994.

[4] P. Berman, M. Karpiński, L. Larmore, W. Plandowski, and W. Rytter, The complexity of pattern matching of highly compressed two-dimensional texts, Proc. of *8th Annual Symposium on Combinatorial Pattern Matching*, CPM'97, July 1997.

[5] B.S. Chlebus and L. Gąsieniec, Optimal pattern matching on meshes, *Proc. 11th Symposium on Theoretical Aspects of Computer Science*, 1994, pp. 213–224.

[6] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Introduction to Algorithms, *MIT Press*, Cambridge MA and McGraw-Hill, New York, 1990.

[7] M. Crochemore, Z. Galil, L. Gąsieniec, K. Park, and W. Rytter, Constant-time randomised parallel string matching, *SIAM Journal on Computing*, 26(4), pp. 950–960, 1997.

[8] M. Crochemore, W. Rytter, Text algorithms, Oxford University Press, 1994.

[9] M. Crochemore, W. Rytter, Usefulness of the Karp-Miller-Rosenberg algorithm in parallel computations on strings and arrays, Theor. Comp. Science 88 (1991) 59-82

[10] A. Czumaj, Z. Galil, L. Gasieniec, W. Plandowski, and K. Park, Optimal Parallel Pattern Matching, *STOC*, May 1995.

[11] A. Czumaj, P. Ferragina, L. Gąsieniec, S. Muthukrishnan, and J. Träff, The Architecture of a Software Library for String Processing, *Workshop on Algorithm Engineering*, Venice, September 1997.

[12] S. De Agostino, P-complete problems in data compression, TCS 127, 181-186, 1994

[13] M. Farach and M. Thorup, String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th ACM Symposium on Theory of Computing*, pp. 703–713, 1994.

[14] L. Gąsieniec, M. Karpiński, W. Plandowski, and W. Rytter, Randomised efficient algorithms for compressed strings: the finger-print approach, *Proc. 7th Combinatorial Pattern Matching*, pp. 39–49, LNCS 1075, June 1996.

[15] L. Gąsieniec, M. Karpiński, W. Plandowski, and W. Rytter, Efficient algorithms for Lempel-Ziv encoding, *Proc. 5th Scandinavian Workshop on Algorithms Theory*, pp. 392–403, LNCS 1097, July 1996.

[16] M. Gu, M. Farach, and R. Beigel, An efficient algorithm for dynamic text indexing, *Proc. of 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1994.

[17] R. Karp, R. Miller, and A. Rosenberg, Rapid identification of repeated patterns in strings, arrays, and trees. *4th ACM Symposium on Theory of Computing* (1972), pp. 125–136.

[18] M. Karpiński, W. Rytter, and A. Shinohara, A pattern matching for strings with short description, *Proc. 6th Combinatorial Pattern Matching*, June 1995.

[19] D. Knuth, J. Morris, and V. Pratt, Fast pattern matching in strings, *SIAM J. on Computing*, 6 (1977), pp. 323–360.

[20] A. Lempel and J. Ziv On the complexity of finite sequences, *IEEE Transactions on Information Theory*, pp. 22:75–81, 1976.

[21] M. Miyazaki, A. Shinohara, and M. Takeda, An Improved Pattern Matching for Strings in Terms of Straight-Line Programs, *8th Combinatorial Pattern Matching*, July 1997, LNCS 1264, pp. 1–11.

[22] W. Plandowski and W. Rytter, Application of Lempel-Ziv encodings to the solution of words equations, *Proc. of 25th International Colloquium on Automata, Languages, and Programming*, to appear.

[23] U. Vishkin, Optimal parallel pattern matching in strings, *Information and Control*, 67 (1985), pp. 371–380.

[24] T.A. Welch, A technique for high performance data compression, *IEEE transactions on Computers*, pp. 17:8–19, 1984.

[25] J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, pp. IT-23(3):337–343, 1977.