Optimal pattern matching in LZW compressed strings

PAWEŁ GAWRYCHOWSKI, University of Wrocław

We consider the following variant of the classical pattern matching problem: given an uncompressed pattern $p[1\,..\,m]$ and a compressed representation of a string $t[1\,..\,N]$, does p occur in t? When t is compressed using the LZW method, we are able to detect the occurrence in optimal linear time, thus answering a question of Amir, Benson, and Farach [Amir et al. 1994]. Previous results implied solutions with complexities $\mathcal{O}(n \log m + m)$ [Amir et al. 1994], $\mathcal{O}(n + m^{1+\epsilon})$ [Kosaraju 1995], or (randomized) $\mathcal{O}(n \log \frac{N}{n} + m)$ [Farach and Thorup 1995], where n is the size of the compressed representation of t. Our algorithm is simple and fully deterministic.

Categories and Subject Descriptors: F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*Pattern matching*

General Terms: Algorithms, Design, Theory

Additional Key Words and Phrases: Pattern matching, compression, Lempel-Ziv-Welch

ACM Reference Format:
Gawrychowski, P. 2011. Optimal pattern matching in LZW compressed strings. ACM Trans. Algor. 1, 1, Article 1 (January 2011), 16 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Even though the processing capabilities of a modern hardware are growing very rapidly, so is the amount of data we need to work with. While inventing efficient compressing methods to store this data has been a very active research area since decades, it seems that with more and more aspects of our lives being processed digitally, the necessity of developing very efficient compression methods is becoming even more important, where efficient should be understood as both fast and achieving a good compression ratio. But even the best compression scheme will not get us much gain if performing any operation on the stored information requires uncompressing it anyway. Thus we should aim to develop not only good compression methods but also efficient algorithms working on the compressed representation alone, preferably with the running times depending only (or mainly) on the size of this representation, not the original input.

The most ubiquitous problem concerning processing information is the *pattern matching* for strings, in which we are given a pattern $p[1\,..\,m]$ and a text $t[1\,..\,N]$, and the goal is to find if there is an occurrence of p in t. There are many efficient solutions to this problem, starting with the first linear time solution of [Morris and Pratt 1970], then optimized for the delay between reading two consecutive letter of the text in [Galil 1981; Knuth et al. 1977], the amount of additional memory used [Galil and

Supported by MNiSW grant number N N206 492638, 2010–2012 and START scholarship from FNP.
Author's addresses: P. Gawrychowski, Institute of Computer Science, University of Wrocław, ul. Joliot-Curie 15, 50–383 Wrocław, Poland, gawry@cs.uni.wroc.pl.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

Seiferas 1981], and the total number of comparisons [Breslauer 1993], to name just a few improvements, with the two most well-known algorithms being Knuth-Morris-Pratt [Knuth et al. 1977] and Boyer-Moore [Boyer and Moore 1977].

If the string is given in a compressed representation, we get the *compressed pattern matching problem*. Given a pattern p and a text t we need to check if there is an occurrence of p in t without decompressing t, i.e., the running time should depend on the size n of the compressed representation of T alone rather than on the original length N. According to Amir and Benson [Amir and Benson 1992], a solution to such problem is *efficient* if its time complexity is $o(N)$, *almost optimal* if its time complexity is $\mathcal{O}(n \log m + m)$, and *optimal* if the complexity is $\mathcal{O}(n + m)$.

Complexity of the compressed pattern matching problem clearly depends on the particular compression method chosen. If we use any nonadaptive compression, the problem becomes rather simple. On the other hand, adaptive compression methods based on the Lempel-Ziv method [Ziv and Lempel 1977] seems to be quite challenging to deal with efficiently, mostly because of the fact that the same fragment of a text might be encoded differently depending on its exact location. There has been a substantial amount of work devoted to developing fast algorithms for pattern matching in both Lempel-Ziv and its simplified version Lempel-Ziv-Welch [Welch 1984] compressed texts. In [Amir et al. 1994] the authors introduced two algorithms with time complexities $\mathcal{O}(n+m^2)$ and $\mathcal{O}(n \log m+m)$ for Lempel-Ziv-Welch compressed texts. The pattern preprocessing time was soon improved in [Kosaraju 1995] to get $\mathcal{O}(n+m^{1+\epsilon})$ time complexity. Then [Farach and Thorup 1995] developed algorithm working for any Lempel-Ziv compressed text in (randomized) time $\mathcal{O}(n \log^2 \frac{N}{n} + m)$, with a version simplified for Lempel-Ziv-Welch working in (also randomized) time $\mathcal{O}(n \log \frac{N}{n} + m)$. [Navarro and Raffinot 1999] developed a practical $\mathcal{O}(n\frac{m}{w})$ algorithm exploiting bit-parallelism. There has been also a substantial amount of research devoted to a more general problem of fully compressed pattern matching, where both the text and the pattern are compressed. For the case of LZW compression, Gąsieniec and Rytter [Gasieniec and Rytter 1999] developed a $\mathcal{O}((n+m)\log(n+m))$ algorithm, where n and m are the compressed sizes of the text and the pattern, respectively. Here we are concerned only with the case of uncompressed patterns, though.

In this paper we show that the compressed pattern matching problem can be solved in optimal linear time for Lempel-Ziv-Welch compressed texts. Variants of this compression method are used, for example, in Unix `compress` utility, in GIF images, and (optionally) in TIFF and PDF files so this result is not only of purely theoretical interest. Our algorithm is simple and fully deterministic.

2. PRELIMINARIES

Let Σ be a fixed finite alphabet (later we will also deal with the more general case of polynomial size integer alphabet). We consider strings over Σ given in a Lempel-Ziv-Welch compressed form, which is a simplified version of the Lempel-Ziv compression defined as follows: given an uncompressed string $t[1\,..\,N]$ we iteratively construct its representation by looking up and removing the longest prefix of the current suffix $t[k\,..\,N]$ which already occurs in $t[1..k-1]$. The output is a sequence of triples $(start_i, len_i, next_i)$ with $t[start_i\,..\,start_i + len_i - 1]$ being this longest prefix and $next_i$ being the next character. In Lempel-Ziv-Welch compression the string is represented as a sequence of *codewords*, where a codeword is either a single letter, or a previously occurring codeword concatenated with a single character. This additional character is not given explicitly: we define it as the first character of the next codeword, and initialize the set of codewords to contain all single characters in the very beginning. The resulting compression method enjoys a particularly simple encoding/decoding process,

but unfortunately requires outputting at least $\Omega(\sqrt{N})$ codewords. Still, its simplicity and good compression ratio achieved on real life instances make it an interesting model to work with. For the rest of the paper we will use LZW when referring to Lempel-Ziv-Welch compression.

We are interested in a variation of the classical pattern matching problem: given a pattern $p[1\,..\,m]$ and a text $t[1\,..\,N]$, does p occur in t? In our case t is given in a compressed form, and we wish to achieve a running time depending on the size n of this compressed representation, not the length of t itself. Additionally, we would like to keep the memory complexity as low as $\mathcal{O}(m)$, preferably with the algorithm reading the compressed representation of t just once from left to right. If the pattern does occur in the text, we would like to get the position of its first occurrence.

The computational model we are going to use is the standard word RAM. In such model we can implement a very efficient dictionary storing a bounded number of elements which we will use in the integer alphabet case.

LEMMA 2.1 (ATOMIC HEAPS [FREDMAN AND WILLARD 1994]). *It is possible to maintain a collection of sets $S(i) \subseteq \{0, 2, \ldots, n-1\}$ so that inserting, removing and finding successor in each of those sets work in constant time (amortized for insert and remove, worst case for find) as long as $|S(i)| \leq \log n$ for all i, assuming a $\mathcal{O}(n)$ time and space preprocessing.*

The very first thing we do is preprocessing p. Using standard tools (suffix trees [Farach 1997; Ukkonen 1995] built for both the pattern and the reversed pattern, and lowest common ancestor queries [Bender and Farach-Colton 2000]) we get the following primitive.

LEMMA 2.2. *Pattern p can be preprocessed in linear time so that given any two fragments $p[i\,..\,i+k]$ and $p[j\,..\,j+k]$ we can find their longest common prefix (suffix) in constant time.*

LEMMA 2.3. *Pattern p can be preprocessed in linear time so that given any fragment $p[i\,..\,j]$ we can find its longest prefix which is a suffix of the whole pattern in constant time, assuming we know the (explicit or implicit) vertex corresponding to $p[i\,..\,j]$ in the suffix tree.*

PROOF. We assume that the suffix tree is built for p concatenated with a special terminating character, say \$. Each leaf in the suffix tree corresponds to some suffix of p, and is connected to its parent with an edge labeled with a single letter. If we mark all those parents, finding the longest prefix which is a suffix of the whole p reduces to finding the lowest marked vertex on a given root-to-vertex path, which can be precomputed for all vertices in linear time. □

At a very high level our procedures resemble the old and well-known Knuth-Morris-Pratt pattern matching algorithm: we compute the longest prefix of p which is a suffix of the current prefix of the text. Recall that if we already have such longest prefix for a given $t[1..i]$ and need to compute the longest match for $t[1..i+1]$, there are two possibilities. Either we extend the current longest prefix by $t[i+1]$, or we can safely replace it with the longest *border*, where border of a string is its proper fragment which is a suffix and a prefix at the same time (usually we identify such fragment with its length and say that $\mathrm{border}(w) = \{i_1, \ldots, i_k\}$ is the set of all borders of w). A closely related concept is the notion of *periods*: d is a period of w if $w[i] = w[i+d]$ whenever both $w[i]$ and $w[i+d]$ are defined. It is easy to see that d is a period of w if and only if $|w| - d$ is a border, and the following property holds.

LEMMA 2.4 (PERIODICITY LEMMA). *If both d and d' are periods of w, and $d + d' \leq |w| + \gcd(d, d')$, then $\gcd(d, d')$ is a period as well.*

This lemma helps us to avoid performing the computation for every possible prefix of the text. We will try to restrict attention to just $\mathcal{O}(n)$ of them without losing any potential occurrence, and to this aim we will need the following consequence of the periodicity lemma.

LEMMA 2.5. *If the longest border of w is of length $b \geq \frac{|w|}{2}$ then all borders of length at least $\frac{|w|}{2}$ create one arithmetic progression. More specifically, $\mathrm{border}(w) \cap \left\{\frac{|w|}{2}, \ldots, |w|\right\} = \left\{|w| - \alpha d : 1 \leq \alpha \leq \frac{|w|}{2d}\right\}$, where $d = |w| - b$ is the period of w. We call this set the long borders of w.*

PROOF. First note that $d = |w| - b$ is the period of $|w|$ so any αd is also a period, as long as $\alpha \leq \frac{|w|}{d}$. Thus all elements of the arithmetic progression in the lemma are borders. We must show that there are no other borders of length at least $\frac{|w|}{2}$. Let b' be any such border. Observe that $d' = |w| - b'$ is a period and $d + d' \leq |w|$ so from the periodicity lemma $\gcd(d, d')$ is a period as well. But d is the shortest period so d divides d' and b' belongs to the progression. □

Thanks to the above lemma instead of processing borders one-by-one we can split them into $\log |p|$ groups with all borders in a single group creating one arithmetic progression, and hope to process them all at once. We will need the borders of both prefixes and suffixes of p.

LEMMA 2.6. *Pattern s can be preprocessed in linear time so that we can find the border of each its prefix (suffix) in constant time.*

PROOF. Simply do the standard preprocessing from the Knuth-Morris-Pratt algorithm for both p and the reversed p. This preprocessing results in computing the border of each possible prefix. □

In the remaining part of this section we develop two efficient procedures operating on fragments of the pattern, which we call *snippets*.

Definition 2.7. A snippet is any substring $p[i\,..\,j]$ of the pattern p. If $i = 1$ we call it a prefix snippet, if $j = m$ a suffix snippet. A sequence of snippets of size k is a concatenation of k substrings of the pattern $p[i_1\,..\,j_1] \ldots p[i_k\,..\,j_k]$.

One could argue that snippets are simply substrings of the pattern and hence the name is redundant. We believe that repeating "substring of the pattern" multiple times makes the proofs more difficult to follow and prefer to call them snippet for the sake of brevity.

We identify snippets with the substrings they represent, and use $|s|$ to denote the length of the string represented by a snippet s. A snippet is stored as a pair of integers (i, j). Sometimes we will also require that a link to the corresponding node (either explicit or implicit) in the suffix tree and the longest suffix which is a prefix of the whole pattern (in case of a prefix snippet this is of course the whole fragment) are available. If this information is known, we call the snippet *extended*.

The first result shows how to detect an occurrence of the pattern in a concatenation of two snippets. We will perform a lot of such operations, and a constant time implementation is crucial.

LEMMA 2.8. *Given a prefix snippet and a suffix snippet we can detect an occurrence of the pattern in their concatenation in constant time.*

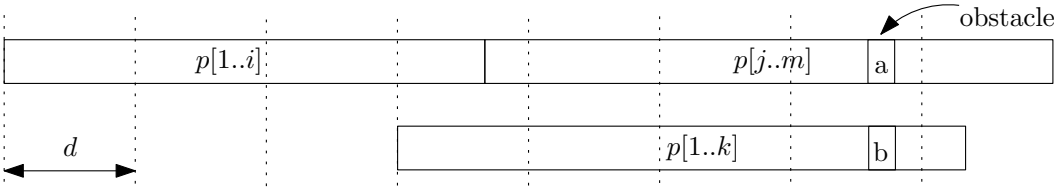

Fig. 1. Detecting an occurrence in a concatenation of two snippets.

PROOF. We need to answer the following question: does p occur in $p[1\,..\,i]p[j\,..\,m]$? Or, in other words, is there $x \in \text{border}(p[1\,..\,i])$ and $y \in \text{border}(p[j\,..\,m])$ such that $x+y=m$? Note that either $x \geq \frac{|p[1..i]|}{2}$ or $y \geq \frac{|p[j..m]|}{2}$, and without losing the generality assume the former. From Lemma 2.5 we know that all such possible values of x create one arithmetic progression. More specifically, $x = i - \alpha d$, where $d \leq \frac{i}{2}$ is the period of $p[1\,..\,i]$ extracted using Lemma 2.6. We need to check if there is an occurrence of p in $p[1\,..\,i]p[j\,..\,m]$ starting after the αd-th character, for some $0 \leq \alpha \leq \frac{i}{d}$. For any such possible interesting shift, there will be no mismatch in $p[1\,..\,i]$. There might be a mismatch in $p[j\,..\,m]$, though.

Let $p[1\,..\,k]$ be the longest prefix of p for which d is a period (so $k \geq i$). Such k can be calculated efficiently by looking up the longest common prefix of $p[d+1\,..\,m]$ and the whole p. We shift $p[1\,..\,k]$ by $\left\lfloor \frac{\min(i, i+|p[j..m]|-m)}{d} \right\rfloor d$ characters. Note this is the maximal shift of the form αd which, after extending $p[1\,..\,k]$ to the whole p, does not result in sticking out of the right end of $p[j\,..\,m]$. Then compute the leftmost mismatch of the shifted $p[1\,..\,k]$ with $p[j\,..\,m]$, see Figure 1. Position of the first mismatch, or its nonexistence, allows us to eliminate all but one interesting shift. More precisely, we have two cases to consider.

(1) There is no mismatch. If $k = m$ we are done, otherwise $p[k+1] \neq p[k+1-d]$, meaning that choosing any smaller interesting shift results in a mismatch.
(2) There is a mismatch. Let the conflicting characters be a and b and call the position at which a occurs in the concatenation the *obstacle*. Observe that we must choose α so that $p[1\,..\,k]$ shifted by αd is completely on the left of the obstacle. If there is enough space for that and $k = m$ we are done. If $k < m$ and $p[1\,..\,k]$ shifted by both αd and $(\alpha+1)d$ is completely on the left of the obstacle, shift by αd results in a mismatch after extending to the whole p because $p[k+1] \neq p[k+1-d]$ and $p[k+1-d]$ matches with the corresponding character in $p[j\,..\,m]$. Thus we may restrict our attention to the largest shift for which $p[1\,..\,k]$ is on the left of the obstacle.

Having identified the only interesting shift, we verify if there is a match using one longest common prefix query on p. More precisely, if the shift is αd, we check if the common prefix of $p[i-\alpha d\,..\,m]$ and $p[j\,..\,m]$ is of length $|p[i-\alpha d\,..\,m]|$. Overall, the whole procedure takes constant time. □

The second result shows how to compute the longest prefix of the pattern which is a suffix of a concatenation of two snippets. Unfortunately, a constant time implementation seems rather unlikely here. In the above proof we had the simple observation that an occurrence corresponds to a long border of either the left or the right part, and here we do not know that.

LEMMA 2.9. *Given a prefix snippet $p[1\,..\,i]$ and a snippet $p[j\,..\,k]$ we can find the longest long border b of $p[1\,..\,i]$ such that $p[1\,..\,b]p[j\,..\,k]$ is a prefix of the whole p in constant time.*

PROOF. As in the previous proof, we begin with computing $p[1\,..\,i']$, the longest prefix of p for which $d \leq \frac{i}{2}$ is a period, where d is the period of $p[1\,..\,i]$ (so $i' \geq i$). We need to choose the smallest α so that shifting p by αd characters results in no mismatches with $p[1\,..\,i]p[j\,..\,k]$, and the shifted p does not end before the right end of $p[1\,..\,i]p[j\,..\,k]$. We begin with shifting $p[1\,..\,i']$ by $\left\lceil \frac{i+|p[j..k]|-m}{d} \right\rceil d$ characters, which is the smallest possible shift, and compute its leftmost mismatch with $p[1\,..\,i]p[j\,..\,k]$, which must be on the right of $p[1\,..\,i]$. We have two cases to consider.

(1) There is no mismatch. If $i' = m$ or $p[i'+1]$ is on the right of $p[j\,..\,k]$, we have the longest long border. Otherwise compute $p[j\,..\,j']$, the longest prefix of $p[j\,..\,k]$ such that d is a period of $p[1\,..\,i]p[j\,..\,j']$. if $j' = k$, there is no such long border. Otherwise we must align $p[1\,..\,i']$ so that $p[i'+1]$ corresponds to $p[j'+1]$. Any other shift results in aligning either $p[i'+1]$ or $p[j'+1]$ with a character continuing the period. Hence we either get that there is no such long border or there is exactly one candidate which can be verified in constant time using one longest common prefix query.
(2) There is a mismatch. Again, call the position in $p[1\,..\,i]p[j\,..\,k]$ at which the mismatch occurs the *obstacle*. Even if we choose a larger shift, $p[1\,..\,i']$ will expect the same character at the obstacle, so we will get a mismatch as well. Thus there is no such long border.

□

3. OVERVIEW OF THE ALGORITHM

Our goal is to detect an occurrence of p in a given Lempel-Ziv-Welch compressed text $t[1\,..\,N]$. At a very high level our idea is to simulate the Knuth-Morris-Pratt algorithm on t. Clearly we cannot afford to process the characters one-by-one and hence try to skip whole codewords. It turns out that working with the codewords directly is somehow rather inconvenient, as they do not have to be anyhow similar to the pattern and hence we cannot hope to preprocess any information which would help us to process whole fragments in constant time. Hence we start with reducing the original problem to a more uniform variant, where we are given a pattern and a collection of strings constructed by concatenating a number of substrings of the pattern one after another. In the next section we show that such problem, which we call pattern matching in a sequence of snippets, can be solved in optimal linear time. Then we observe that pattern matching in LZW compressed text reduces in linear time to such restricted problem. If the alphabet is of constant size, the reduction is rather straightforward. If it does not, we need some additional ideas to keep the running time linear.

4. PATTERN MATCHING IN A SEQUENCE OF SNIPPETS

In this section we develop an optimal linear time algorithm which detects an occurrence of the pattern in a string constructed by concatenating a number of extended snippets. Given such sequence, we would like to simulate the Knuth-Morris-Pratt algorithm efficiently. To this aim we need Lemma 2.8 and Lemma 2.9 presented in the preliminaries. Using those two procedures as basic building blocks, we can present the first algorithm NAIVE-PATTERN-MATCHING. It simulates the Knuth-Morris-Pratt algorithm, adding just two optimizations: we extend the current match by whole snippets, not single letters, and process all long borders of the current match at once. This is not enough to achieve a linear complexity yet: it might happen that we need to

Algorithm 1 NAIVE-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

```
1: ℓ ← longest prefix of p ending s_1                                  ▷ Lemma 2.3
2: k ← 2
3: while k ≤ n and ℓ + Σ_{i=k}^{n} |s_i| ≥ m do
4:     check for an occurrence of p in p[1..ℓ]s_k                      ▷ Lemma 2.8
5:     if p[1..ℓ]s_k is a prefix of p then
6:         ℓ ← ℓ + |s_k|
7:         k ← k + 1
8:         continue
9:     end if
10:    b ← longest long border of p[1..ℓ] s.t. p[1..b]s_k is a prefix of p   ▷ Lemma 2.9
11:    if b is undefined then
12:        ℓ ← longest prefix of p ending p[⌈ℓ/2⌉..ℓ]                   ▷ Lemma 4.1
13:        continue
14:    end if
15:    ℓ ← b + |s_k|
16:    k ← k + 1
17: end while
```

spend $\Omega(\log m)$ time at each snippet. Nevertheless, such method will serve as a basis for developing the optimal solution.

While we assume that for all input snippets the information about corresponding vertex in the suffix tree and longest suffix being a prefix of p is already known, during the execution we might create some new snippets. Fortunately, they are either prefix snippets, or snippets of the form $p[\lceil \frac{\ell}{2} \rceil \,..\, \ell]$ which will be called the *half snippets*. Before we analyze the running time of NAIVE-PATTERN-MATCHING we need a small technical lemma concerning those fresh snippets. Note that it would be possible to modify the algorithm a little bit so that we create only prefix snippets (or so that not every snippet has to be extended), but we prefer to keep its description more modular and show how to make the fresh snippets extended instead.

LEMMA 4.1. *Information for all prefix and half snippets can be computed in linear time.*

PROOF. In case of prefix snippets the computation is trivial. The case of all half snippets is not completely trivial, though. To compute the suffixes we loop through the possible values of ℓ. Assuming we have the longest suffix of $p[\lceil \frac{\ell}{2} \rceil \,..\, \ell]$ which is a prefix $p[1\,..\,i]$, we try to compute such longest suffix of $p[\lceil \frac{\ell+1}{2} \rceil \,..\, \ell+1]$. For that we consider the borders of $p[1\,..\,i]$ looking for the one which can be extended with $p[\ell+1]$. Then we check if the extended border is longer than $p[\lceil \frac{\ell+1}{2} \rceil \,..\, \ell+1]$ and if so, take its border. The total complexity of this procedure is linear because at each step we increase the length of the current prefix by at most 1. Now consider locating the vertices in the suffix tree. Assume that we know the corresponding vertex for a given value of ℓ and need to update the information after increasing ℓ by one. Extending the current word by one letter requires traversing at most one edge in the suffix tree, then we might need to remove the first letter. If the current vertex is explicit, we can use its suffix link. Otherwise we use the suffix link of its deepest explicit ancestor, and then traverse edges down from the vertex found. This traversing might require more than constant time, but can be amortized by noting that the number of explicit ancestors of the current vertex cannot exceed n, and decreases by at most one at for each ℓ. To

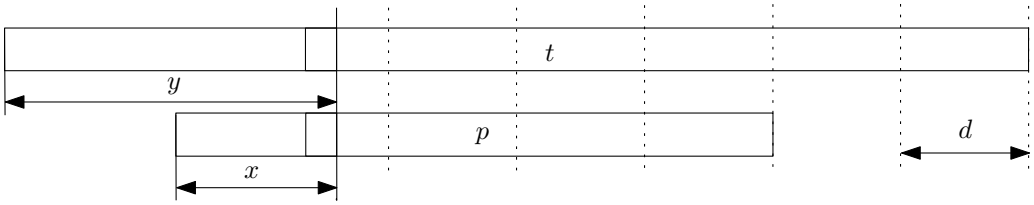

Fig. 2. Shifting p to get an occurrence in t.

finish the proof note that looking up the edge can be performed very quickly even when the alphabet is not constant: there are at most two half snippets of a given length and so we can afford to simply iterate through all outgoing edges, then the total complexity of all lookups will be just linear in $|p|$. □

THEOREM 4.2. NAIVE-PATTERN-MATCHING *works in time* $\mathcal{O}(n \log m)$.

PROOF. Observe that each iteration of the **while** loop results in either increasing k by 1 or decreasing $\ell \leq m$ at least twice. Thus the total number of iterations is $\mathcal{O}(n \log m)$. In each iteration we spend just a constant time due to the above lemmas. Note that we require that for any snippet we know not only its start and end in p but also the corresponding vertex in the suffix tree (this will be very important for the improved method). While we assume that we get such information for all the input snippets, we might create some new snippets during the execution of the algorithm. Fortunately, the only possible non-input snippets we create are prefix and half snippets, and so we can use Lemma 4.1 to preprocess them. □

To accelerate the above basic method we use the concept of *levers*:

Definition 4.3. We call s_i a lever of a sequence of snippets $s_1 s_2 \ldots s_k$ if $\sum_{j=1}^{i-1} |s_j| \leq \frac{|s_i|}{2}$.

Given such a lever we can eliminate many potential occurrences at once. This is formalized in Lemma 4.4 and Lemma 4.5.

LEMMA 4.4. *Given a sequence of extended snippets* $s_1 s_2 \ldots s_i$ *in which* s_i *is a lever, we can detect an occurrence of* p *in* $s_1 s_2 \ldots s_i$ *in time* $\mathcal{O}(i)$.

PROOF. Let $L = |s_i|$ and $S = \sum_{j=1}^{i-1} |s_j|$. Because $S \leq \frac{L}{2}$ and $L \leq m$, S is at most $\frac{m}{2}$, meaning that any occurrence of p must end in s_i. Note that we can safely replace s_i with the longest suffix $p[k\,..\,m]$ of the pattern which is its prefix, by Lemma 2.3 this requires just constant time. First we check if p is a suffix of the whole sequence in time $\mathcal{O}(i)$. Now, any possible occurrence of p cannot end earlier than after the $m - S \geq \frac{L}{2}$-th character and so in fact we are looking for a long border b of $p[k\,..\,m]$ such that $p[1\,..\,m-b]$ ends $s_1 s_2 \ldots s_{i-1}$. From Lemma 2.5 we know that any such b must be of the form $|p[k\,..\,m]| - \alpha d$, where $d \leq \frac{|p[k..m]|}{2}$ is the period of $p[k\,..\,m]$ calculated in constant time by Lemma 2.6. It turns out that we can restrict the set of possible values of α to just one, which can be then checked naively in time $\mathcal{O}(i)$.

First we compute how far the period of $p[k\,..\,m]$ continues to the left starting from the right end of both $p[1\,..\,k-1]$ and $s_1 s_2 \ldots s_{i-1}$. In order to perform both those computations efficiently we only have to develop a constant time procedure which, given two snippets $p[i\,..\,j]$ and $p[k\,..\,\ell]$, finds the longest suffix of the former which is also a suffix of some power of the latter. Such procedure works in two steps:

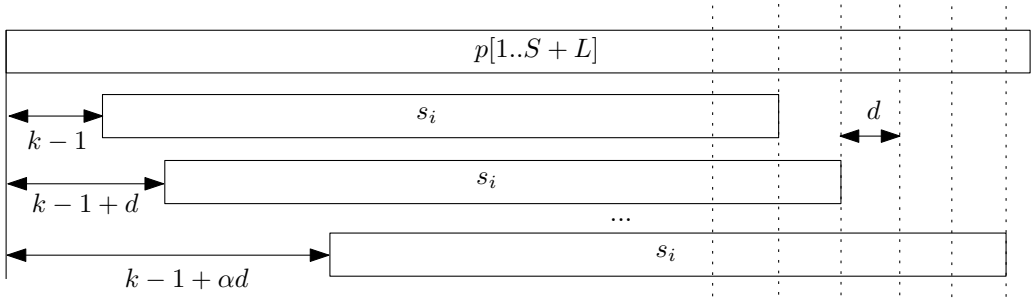

Fig. 3. All occurrences of s_i in $p[1\,..\,S+L]$.

(1) find the longest common suffix of $p[i\,..\,j]$ and $p[k\,..\,\ell]$, return if it is shorter than $|p[k\,..\,\ell]|$,
(2) find and return the longest common suffix of $p[i\,..\,j]$ and $p[i\,..\,j-|p[k\,..\,\ell]|]$.

By repeating this procedure at most i times we can assume that we know how far the period continues in both strings. If d is a period of the whole p, recall that p is not a suffix of the whole $s_1 s_2\,..\,s_i$, and shifting it to the left by any multiple of d cannot result in a match. Otherwise we have found the rightmost position x such that $p[x] \neq p[x+d]$. If d is a period of the whole $t = s_1 s_2\,..\,s_i$, p does not occur there. Otherwise we have found the rightmost position y such that $t[y] \neq t[y+d]$. Now we claim that in order to get a match, we must shift p so that its x-th character is aligned with the y-th character of t, see Figure 2 (recall that we shift p by a multiple of d). Indeed, choosing any smaller shift creates a mismatch concerning $p[x]$ and choosing any bigger shift creates a mismatch concerning $t[y]$. This gives us a simple arithmetic condition on the only possible value of α: if d does not divide $|t|-|p|+x-y$ there is no occurrence, otherwise we check $\alpha = \frac{|t|-|p|+x-y}{d}$. □

LEMMA 4.5. *Given a sequence of extended snippets $s_1 s_2 \ldots s_i$ in which s_i is a lever, we can compute the longest prefix of p which is a suffix of $s_1 s_2 \ldots s_i$ in time $\mathcal{O}(i)$.*

PROOF. As in the previous proof, let $L = |s_i|$ and $S = \sum_{j=1}^{i-1} |s_j|$. s_i is an extended snippet so we already know the longest prefix of p which is its suffix. We should check if there is any longer prefix. If so, it corresponds to an occurrence of s_i in $p[1\,..\,S+L]$. First we locate the node corresponding to s_i in the suffix tree built for p. The tree can be preprocessed (in linear time) so that having this node we can compute the first and second occurrence of s_i in $p[1\,..\,S+L]$ (more precisely, we compute the first and second occurrences in the whole p, and check if they are inside $p[1\,..\,S+L]$). If there is none, we terminate. If there is just one, we check naively in time $\mathcal{O}(i)$ the corresponding prefix. If there are two, the situation is more complicated, as in fact there can be many more of them, and we cannot afford to iterate through all possibilities.

Because s_i is a lever, $L \geq 2|S|$ and $L \geq \frac{2}{3}(S+L)$. Thus the overlap of any two occurrences of s_i in $p[1\,..\,S+L]$ is of length at least $\frac{L}{2}$ and so any non-leftmost occurrence corresponds to a long border of s_i. Let d be the period of s_i (note that we do not know d yet), the first occurrence starts at the k-th character of p, and the last occurrence starts at the $k+\alpha d$-th character of p. Then it is clear that there are occurrences starting at the $k+\beta d$-th characters, for any $0 \leq \beta \leq \alpha$, see Figure 3. In particular, the second occurrence starts at the $k+d$-th character, so knowing the positions of the first and second occurrence allows us to compute d. Having the value of d and k, we find the longest common prefix of $p[k\,..\,S+L]$ and $p[k+d\,..\,S+L]$, which gives us the value of

Algorithm 2 LEVERED-PATTERN-MATCHING$(s_1, s_2, \ldots, s_n)$

```
1: ℓ ← longest prefix of p ending s_1                          ▷ Lemma 2.3
2: k ← 2
3: while k ≤ n and ℓ + Σ_{i=k}^{n} |s_i| ≥ m do
4:     choose t ≥ k minimizing |s_k| + |s_{k+1}| + ... + |s_{t-1}| − |s_t|/2
5:     if ℓ + |s_k| + |s_{k+1}| + ... + |s_{t-1}| ≤ |s_t|/2 then
6:         check for occurrence of p in p[1..ℓ]s_k s_{k+1} ... s_t   ▷ Lemma 4.4
7:         ℓ ← longest prefix of p ending p[1..ℓ]s_k s_{k+1} ... s_t ▷ Lemma 4.5
8:         k ← t + 1
9:     else
10:        execute lines 4–16 of NAIVE-PATTERN-MATCHING
11:    end if
12: end while
```

α (more precisely, if the length is ℓ, $\alpha = \lfloor \frac{\ell+d-L}{d} \rfloor$). Now we can use the same method as in the previous proof. First compute how far the period of s_i continues to the left starting from the right end of both $p[1\,..\,k-1]$ and $s_1 s_2 \ldots s_{i-1}$. Then consider the following three cases.

(1) If d is a period of $s_1 s_2 \ldots s_i$, for a suffix of length at least $|s_i|$ which is a prefix of p to exists d must be a period of $p[1\,..\,k-1+d]$ as well. Then the longest such suffix corresponds to the biggest $\beta \leq \alpha$ for which $S \geq k-1+\beta d$.
(2) If d is not a period of the whole $s_1 s_2 \ldots s_i$ but is a period of $p[1\,..\,k-1+d]$, any prefix of p of length at least $|s_i|$ which is a suffix of $s_1 s_2 \ldots s_i$ must be completely contained in the periodic suffix of $s_1 s_2 \ldots s_i$. We can select the longest such suffix in constant time.
(3) If d is neither a period of the whole $p[1\,..\,k-1+d]$ nor $s_1 s_2 \ldots s_i$, we get a simple arithmetic condition on the only possible value of α as in the previous lemma. Then we verify this value using $\mathcal{O}(i)$ longest common prefix computations.

□

We are ready to present the improved (and final) algorithm. The idea is that whenever the sequence contains a lever, we can use Lemma 4.4 and Lemma 4.5 to quickly process a bunch of snippets. Otherwise we stick to the same method as in NAIVE-PATTERN-MATCHING. We call the resulting method LEVERED-PATTERN-MATCHING.

While the idea is rather simple, it turns out that the notion of levers captures all inputs on which NAIVE-PATTERN-MATCHING runs in superlinear time. To formalize this claim we will need a few technical lemmas, but before that we show how to execute line 4 efficiently.

LEMMA 4.6. *Line 4 of* LEVERED-PATTERN-MATCHING *can be executed in amortized constant time, and $\mathcal{O}(m)$ additional memory.*

PROOF. If we are allowed to use as much as $\Theta(n)$ additional memory, we can preprocess all minima going right to left: the best possible choice of t for a given value of k is either the same as for $k+1$, or it is equal k. If we would like to optimize the amount of additional memory used, and avoid the necessity of reading all input snippets when there is a match somewhere near the very beginning, we can use a slightly more complicated method. First note that we are interested only in $t \leq k+m$. For each current value of k we keep an increasing list of *candidates* $k \leq t_1 < t_2 < \ldots < t_c \leq k+m$. Let

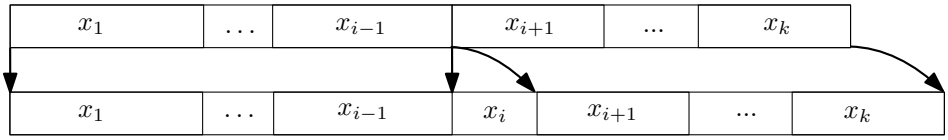

Fig. 4. Inserting x_i between x_{i-1} and x_{i+1}.

$f(i) = |s_1| + |s_2| + \ldots + |s_{i-1}| - \frac{|s_i|}{2}$, then t_1 is the position with the minimum value of f in the interval $[k, k+m]$, t_2 is the position with the minimum value of f in the interval $[t_1+1, k+m]$, and so on. Before increasing k by one we need to remove t_1 from the list, if $t_1 = k$, and consider a new candidate $k+1+m$: remove all t_i with $f(t_i) > f(k+1+m)$, and add $k+1+m$ to the list. The amortized cost of this update is clearly constant, and the maximum number of stored candidates $\mathcal{O}(m)$. To find t, simply take the first candidate t_1. □

With each sequence of snippets $s_1 s_2 \ldots s_k$ we associate its potential $\Phi(|s_1|, |s_2|, \ldots, |s_k|)$, which roughly corresponds to the amount of work we still need to perform if we use the concept of levers. Before we use it to bound the running time of LEVERED-PATTERN-MATCHING, we need a few observations.

Definition 4.7. Let $x_1, x_2, \ldots, x_k \leq m$ be a sequence of natural numbers. Consider a segment of length $\sum_{i=1}^{k} x_i$ split into k blocks of length x_i, for $i = 1, 2, \ldots, k$. First mark its suffix of length m. Then, for each i, mark a fragment of length $\frac{x_i}{2}$ ending just before the part corresponding to x_i. Let y_i be the length of the marked suffix in the block corresponding to x_i, and define the potential $\Phi(x_1, x_2, \ldots, x_k)$ as $\sum_{i=1}^{k} 2 + \log \frac{x_i}{y_i}$.

Note than all y_i are strictly positive so the potential is well-defined for any sequence of snippets.

LEMMA 4.8. $\Phi(x_1, x_2, \ldots, x_k) \leq 7k$.

PROOF. We apply induction on k. If $k = 1$ the whole segment is marked so the potential is 2 and the claim trivially holds. Let $k > 1$, choose $x_i = \min\{x_1, x_2, \ldots, x_k\}$ and assume that the claim holds for $\Phi(x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_k)$. We compute the maximum possible increase in the potential after inserting a block of length x_i between x_{i-1} and x_{i+1}, see Figure 4. There are two reasons the potential might increase.

(1) We create a new block of x_i cells. Note that either $i = k$ and they are all marked, or $i < k$ and $x_{i+1} \geq x_i$ so at least $\frac{x_i}{2}$ of them are marked. In either case, the resulting increase in the potential is at most $2 + \log \frac{x_i}{y_i} \leq 2 + \log 2 = 3$.

(2) We move all blocks corresponding to $x_{i+1}, \ldots, x_k$ further to the right. It might result in unmarking some cells in the blocks corresponding to $x_1, x_2, \ldots, x_{i-1}$. Because we shift all those blocks by x_i to the right, the unmarked blocks are contained in a segment of such length. Because x_i is smaller than any x_j with $j < i$, any segment of length x_i intersects at most two blocks on the left of x_i. Consider the situation in one such block: the old potential was $2 + \log \frac{x_j}{y_j}$, the new potential is $2 + \log \frac{x_j}{y'_j}$ with $y'_j \geq \max(y_j - x_i, \frac{x_{j+1}}{2}) \geq \max(y_j - x_i, \frac{x_i}{2})$. The increase is at most:

$$\log \frac{x_j}{y'_j} - \log \frac{x_j}{y_j} = \log \frac{y_j}{y'_j} \leq \log \frac{y_j}{\max(y_j - x_i, \frac{x_i}{2})}$$
$$= \begin{cases} \log \frac{y_j}{y_j - x_i} \leq \log \frac{y_j}{y_j - \frac{2}{3} y_j} = \log 3 & \text{if } x_i \leq \frac{2}{3} y_j \\ \log \frac{2y_j}{x_i} \leq \log \frac{2\frac{3}{2} x_i}{x_i} = \log 3 & \text{if } x_i > \frac{2}{3} y_j \end{cases}$$

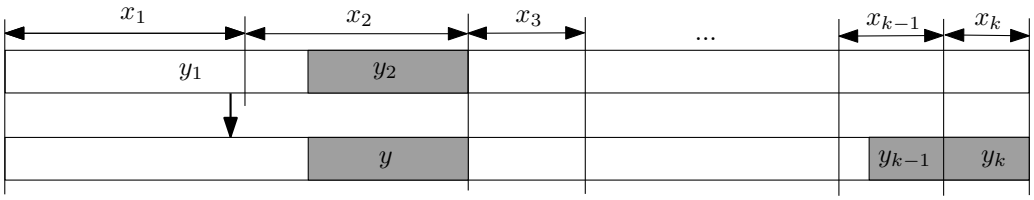

Fig. 5. Merging blocks corresponding to x_1 and x_2.

because there might be two such blocks, the maximum increase is $2 \log 3 \leq 4$.

By summing the above cases $\Phi(x_1, x_2, \ldots, x_k) \leq \Phi(x_1, \ldots, x_{i-1}, x_i, \ldots, x_k) + 7 \leq 7k$. □

LEMMA 4.9. $\Phi(x_1, x_2, \ldots, x_k) \geq 1 + \Phi(x_1 + x_2, \ldots, x_k)$.

PROOF. Let $y \geq y_2$ be the number of marked cells in the block corresponding to $x_1 + x_2$ in $\Phi(x_1 + x_2, x_3, \ldots, x_k)$, see Figure 5. The only change in the potential concerns the first $x_1 + x_2$ cells:

$$\begin{aligned} \Delta &= \Phi(x_1, x_2, x_3, \ldots, x_k) - \Phi(x_1 + x_2, x_3, \ldots, x_k) \\ &= 2 + \log \frac{x_1}{y_1} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y} \end{aligned}$$

We have a few cases to consider.

(1) $x_1 \leq \frac{x_2}{2}$. Then $y_1 = x_1$ and the change in the potential is:

$$\begin{aligned} \Delta &\geq 2 + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &= 1 + \log \frac{2x_2}{x_1 + x_2} \geq 1 + \log \frac{4}{3} \geq 1 \end{aligned}$$

(2) $x_1 \geq \frac{x_2}{2}$ and $y_1 = \frac{x_2}{2}$. The change in the potential is:

$$\begin{aligned} \Delta &\geq 2 + \log \frac{x_1}{y_1} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &\geq 3 + \log \frac{x_1}{x_2} + \log \frac{x_2}{y_2} - \log \frac{x_1 + x_2}{y_2} \\ &= 3 + \log \frac{x_1}{y_2} - \log \frac{x_1 + x_2}{y_2} = 3 + \log \frac{x_1}{x_1 + x_2} \\ &\geq 3 + \log \frac{1}{3} = 1 + \log \frac{4}{3} \geq 1 \end{aligned}$$

(3) $y_1 > \frac{x_2}{2}$. Then all marked cells in the block corresponding to x_1 are marked either because of some long x_i with $i > 2$ or because they are among the m rightmost cells, so $y_2 = x_2$ and merging two first blocks does not change the number of marked cells there, $y = y_1 + y_2$. We can bound Δ as follows:

$$\begin{aligned} \Delta &= 2 + \log \frac{x_1}{y_1} - \log \frac{x_1 + x_2}{y_1 + y_2} \\ &= 2 + \log \frac{x_1}{y_1} - \log \frac{x_1 + x_2}{y_1 + x_2} \geq 2 \end{aligned}$$

where the last inequality follows from the fact that if $p \geq q$ then $\frac{p}{q} \geq \frac{p+r}{q+r}$.

□

LEMMA 4.10. $\Phi(x_1, x_2, \ldots x_k) \geq \Phi(x'_1, x_2, \ldots, x_k)$ *if* $x_1 \geq x'_1$.

PROOF. Decreasing x_1 cannot change any y_i with $i > 1$. Let y'_1 be the number of marked cells in the block corresponding to x'_1. Then either $y_1 = y'_1$, and from the definition of the potential we get the claim, or $y'_1 = x'_1$, and $\log \frac{x'_1}{y'_1} = 0 \leq \log \frac{x_1}{y_1}$. □

THEOREM 4.11. LEVERED-PATTERN-MATCHING *can be implemented to work in time* $\mathcal{O}(n)$ *and use* $\mathcal{O}(m)$ *additional memory.*

PROOF. Consider any execution of the algorithm. In the very beginning we allocate $\Phi(|s_1|, |s_2|, \ldots, |s_n|)$ credits which we then use to pay for all executions of lines 4–10 of LEVERED-PATTERN-MATCHING. We keep the following invariant during the whole execution: we have $\Phi(\ell, |s_k|, |s_{k+1}|, \ldots, |s_n|)$ credits available. Consider a single pass through the body of the **while** loop. From Lemma 4.6 the amortized cost of executing line 4 is constant. Consider the remaining lines.

(1) There is a lever s_t in $p[1\,..\,\ell]s_k s_{k+1} \ldots s_t$. We need as much as $\mathcal{O}(t-k+1)$ time to process it, but then by $t-k+1$ applications of Lemma 4.9 the required potential is at most:
$$\begin{aligned}&\Phi(\ell + |s_k| + |s_{k+1}| + \ldots + |s_t|, |s_{t+1}|, \ldots, |s_n|)\\ &\quad\leq \Phi(\ell, |s_k|, |s_{k+1}|, \ldots, |s_n|) - (t-k+1)\end{aligned}$$
which leaves us with $t-k+1$ free credits which we can use to pay for the processing time.

(2) There is no lever. Then we execute the corresponding lines from NAIVE-PATTERN-MATCHING which takes just constant time and results in either increasing k by 1 and ℓ by at most $|s_k|$ or decreasing ℓ at least twice. In the first case the required potential is by Lemma 4.9 and Lemma 4.10 at most:
$$\Phi(\ell + |s_k|, |s_{k+1}|, \ldots, |s_n|) \leq \Phi(\ell, |s_k|, \ldots, |s_n|) - 1$$
The second case is slightly more involved: if decreasing ℓ creates a lever, we pay for the pass in the next round. Otherwise we replace $\log \frac{x_1}{y_1}$ with $\log \frac{x_1}{2y_1}$ in the potential so the required amount of credits left is just:
$$\Phi(\frac{\ell}{2}, |s_k|, \ldots, |s_n|) \leq \Phi(\ell, |s_k|, \ldots, |s_n|) - 1$$
which leaves us with one spare credit.

The total amount of allocated credits is by Lemma 4.8 just $\mathcal{O}(n)$ and so is the time complexity. □

5. LZW COMPRESSION WITH CONSTANT ALPHABET

Recall that a LZW compressed string is a sequence of codewords $t_1 t_2 \ldots t_n$, each codeword being either a single letter, or a previously occurring codeword concatenated with a single letter. First we check if the pattern p occurs inside one of the strings represented by the codewords. Then for each codeword we need to know if the string it represents occurs in the pattern p, and if it does, we need to find the corresponding vertex in the suffix tree. We also need to know its longest prefix which is a suffix of p, and the longest suffix which is a prefix of p (actually, knowing the prefix is necessary only when the string does not occur inside p). All those informations can be found efficiently (in constant time for each codeword) assuming that the alphabet is of constant size.

LEMMA 5.1. *If the alphabet is of constant size, we can perform the preprocessing for all codewords in total linear time.*

PROOF. The whole set of codewords should be viewed as a trie. For each vertex of this trie we are required to compute:

(1) the longest prefix of the corresponding word which is a suffix of p,
(2) if the corresponding word occurs in p, locate its (implicit or explicit) vertex in the suffix tree,
(3) the longest suffix of the corresponding word which is a prefix of p.

We build an automaton $\mathcal{A}$ recognizing all prefixes of p, i.e., its states set is $\{0, 1, \ldots, |p|\}$ and after reading a word w we are in the state corresponding to the longest prefix of p ending w. Such automaton can be easily constructed in linear time if the alphabet is of constant size. Using the automaton we can compute the longest suffix of each codeword in constant time per codeword.

Then we build the suffix tree for p in linear time [Ukkonen 1995]. For each codeword we will find the corresponding vertex in the suffix tree, if any. Note that such information is actually enough to compute the longest prefix which is a suffix of p for each codeword: first apply Lemma 2.3 to all codewords with corresponding vertex in the suffix tree. Then for all other codewords, simply take the answer computed for its lowest ancestor with a known result. To locate all corresponding vertices in the suffix tree we traverse the trie in a top-bottom fashion. To find the vertex for v, take the vertex found for its parent and traverse at most one edge. This gives a total linear time. □

After applying the above lemma to the set of all codewords, for each maximal sequence of codewords representing substrings of p ($p[i_1 \,..\, j_1]p[i_2 \,..\, j_2] \ldots p[i_k \,..\, j_k]$) we take the longest suffix of the preceeding codeword which is a prefix of p ($p[1 \,..\, i]$) and the longest prefix of the succeeding codeword which is a suffix of p ($p[j \,..\, m]$), and run the algorithm from the previous section on the sequence of snippets $p[1 \,..\, i]p[i_1 \,..\, j_1] \ldots p[i_k \,..\, j_k]p[j \,..\, m]$. Because each such run takes time $\mathcal{O}(k + 2)$, and all those values of k sum up to at most n, the total complexity including the preprocessing of the pattern will be $\mathcal{O}(n + m)$. Note that in fact we do not have to process all codewords in the very beginning: we can process the input in an online fashion codword-by-codeword. Computing the information for each single codeword takes constant time, and if there is an occurrence starting at the i-th codeword, running LEVERED-PATTERN-MATCHING requires accessing no more than $i+m$ first codewords. Hence if the first occurrence starts at the n'-th codeword, we can detect it in $\mathcal{O}(n' + m)$ time.

THEOREM 5.2. *Pattern matching for LZW compressed strings over a constant size alphabet can be solved in optimal linear time.*

6. LZW COMPRESSION WITH INTEGER ALPHABET

The assumption of constant size alphabet is not necessary to achieve the claimed linear running time. If the alphabet is of non-constant size but consists of integers which can be sorted in linear time, we can create all necessary snippets sequences in linear time as well.

LEMMA 6.1. *If the alphabet consists of integers which can be sorted in linear time, we can perform the preprocessing for all codewords in total linear time, assuming the RAM model of computation.*

PROOF. We begin with constructing the suffix tree in linear time using the assumption of integer alphabet and applying the result of [Farach 1997]. Then we are left with answering the following three questions for each different codeword:

(1) find its longest prefix which is a suffix of p,

(2) check if it occurs in p, and if it does, find the corresponding vertex,
(3) find its longest suffix which is a prefix of p.

We answer those questions for all codewords at once, which requires reading the whole input even if there is an occurrence in the very beginning. Questions of each type are processed separately.

(1) We use Lemma 2.3 and the information found for the second type questions.
(2) We build a trie T containing all the codewords. Then answering the questions reduces to computing the intersection of this trie and the suffix tree S (which is a compressed representation of a trie containing all subwords of p). This can be done in time $\mathcal{O}(|S| + |T|)$ assuming the edges outgoing from each vertex (both in S and T) are sorted according to their labels. We process the codewords in order of their lengths. Assuming that we know the corresponding vertices for all codewords of length ℓ, we consider all codewords wx of length $\ell + 1$, and group them according to the vertex corresponding to w (if there is none, wx clearly does not occur in p). In each group the codewords are sorted using x as the key, which can be assured by sorting all codewords in the very beginning. Then we find the corresponding vertices for all codewords in a single group at once, using a left-to-right scan.
(3) Finding such suffix for a single codeword can be performed by running the Knuth-Morris-Pratt algorithm. While the amortized complexity of processing a single letter is constant, we have a lot of different letters that can extend a given codeword, and the amortization argument does not give a linear bound in such case.
Consider the automaton recognizing all prefixes of p from Lemma 5.1, i.e., with the states set $\{0, 1, \ldots, m\}$ and the transitions $\delta(i, a) = \max\{j : p[1 \ldots j] = p[1 \ldots i]a\}$. If the alphabet is of unbounded size the simple linear time construction is no longer possible. Fortunately, the number of nonzero transitions outgoing from a single vertex is at most $2 \log m$, which follows from a well known fact [Crochemore et al. 2007, Lemma 2.32] that $\pi'^{(2)}(k) < \frac{k}{2}$, where π' is the so-called strong failure function, defined as follows: $\pi'(k)$ is the longest border of $p[1 \ldots k]$ such that $p[k+1] \neq p[\pi'(k)+1]$. In fact much more is known: due to a result of [Simon 1994], the total number of nontrivial transitions is linear in m, regardless of the size of the alphabet. By Lemma 2.1, we can maintain a collections of sets logarithmic size, with amortized constant time update and worst-case constant time look-up, which is enough to construct the automaton in linear time. We create one set for each state. Assuming that we have the sets for all $i' < i$, we consider the i-th state. Its outgoing transitions are exactly the transitions outgoing from the border of $p[1 \ldots i]$ with one exception: $\delta(i, p[i + 1]) = i + 1$. Hence we can copy the set computed for the border and add (or replace) one element. Then we are able to compute the transition function in constant time per character, which allows us to process all codewords in linear time. Introducing the atomic heaps results in a rather complicated method, and in this particular case it is possible to significantly simplify this structure because the universe is of just polynomial (in m) size. It does not result in changing the claimed time bounds, though.

□

We believe the method of storing the automaton used in the above proof, while simple, might find other applications.

This gives us the final result. Note that in case of integer alphabets, it is not clear how to avoid reading the whole input even in the case when there is an occurrence somewhere in the very beginning.

THEOREM 6.2. *Pattern matching for LZW compressed strings over a polynomial size integer alphabet can be solved in optimal linear time assuming the word RAM model.*

REFERENCES

AMIR, A. AND BENSON, G. 1992. Efficient two-dimensional compressed matching. In *DCC '92: Proceedings of the Conference on Data Compression*. 279–288.

AMIR, A., BENSON, G., AND FARACH, M. 1994. Let sleeping files lie: pattern matching in z-compressed files. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 705–714.

BENDER, M. A. AND FARACH-COLTON, M. 2000. The LCA problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*. Springer-Verlag, London, UK, 88–94.

BOYER, R. S. AND MOORE, J. S. 1977. A fast string searching algorithm. *Commun. ACM 20,* 10, 762–772.

BRESLAUER, D. 1993. Saving comparisons in the Crochemore-Perrin string matching algorithm. In *ESA '93: European Symposium on Algorithms*. Lecture Notes in Computer Science Series, vol. 726. Springer, 61–72.

CROCHEMORE, M., HANCART, C., AND LECROQ, T. 2007. *Algorithms on Strings*. Cambridge University Press.

FARACH, M. 1997. Optimal suffix tree construction with large alphabets. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, Washington, DC, USA, 137.

FARACH, M. AND THORUP, M. 1995. String matching in Lempel-Ziv compressed strings. In *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 703–712.

FREDMAN, M. L. AND WILLARD, D. E. 1994. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci. 48,* 3, 533–551.

GALIL, Z. 1981. String matching in real time. *J. ACM 28,* 1, 134–149.

GALIL, Z. AND SEIFERAS, J. 1981. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 106–113.

GASIENIEC, L. AND RYTTER, W. 1999. Almost optimal fully LZW-compressed pattern matching. In *DCC '99: Proceedings of the Conference on Data Compression*. IEEE Computer Society, Washington, DC, USA, 316.

KNUTH, D. E., MORRIS, JR., J. H., AND PRATT, V. R. 1977. Fast pattern matching in strings. *SIAM J. Comput. 6,* 2, 323–350.

KOSARAJU, S. R. 1995. Pattern matching in compressed texts. In *Proceedings of the 15th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag, London, UK, 349–362.

MORRIS, JR., J. H. AND PRATT, V. R. 1970. A linear pattern-matching algorithm. Tech. Rep. 40, University of California, Berkeley.

NAVARRO, G. AND RAFFINOT, M. 1999. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *CPM '99: Proceedings of the 10th Annual Symposium on Combinatorial Pattern Matching*. Springer-Verlag, London, UK, 14–36.

SIMON, I. 1994. String matching algorithms and automata. In *Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science*. Springer-Verlag, London, UK, 386–395.

UKKONEN, E. 1995. On-line construction of suffix trees. *Algorithmica 14,* 3, 249–260.

WELCH, T. A. 1984. A technique for high-performance data compression. *Computer 17,* 6, 8–19.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory 23*, 337–343.

Received January 1999; revised January 1999; accepted January 1999