Beating $\mathcal{O}(nm)$ in Approximate LZW-Compressed Pattern Matching*

Paweł Gawrychowski[1] and Damian Straszak[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
gawry@cs.uni.wroc.pl
[2] Institute of Computer Science, University of Wrocław, Poland
damian.straszak@gmail.com

Abstract. Given an LZW/LZ78 compressed text, we want to find an approximate occurrence of a given pattern of length m. The goal is to achieve time complexity depending on the size n of the compressed representation of the text instead of its length. We consider two specific definitions of approximate matching, namely the Hamming distance and the edit distance, and show how to achieve $\mathcal{O}(n\sqrt{m}k^2)$ and $\mathcal{O}(n\sqrt{m}k^3)$ running time, respectively, where k is the bound on the distance, both in linear space. Even for very small values of k, the best previously known solutions required $\Omega(nm)$ time. Our main contribution is applying a periodicity-based argument in a way that is computationally effective even if we operate on a compressed representation of a string, while the previous solutions were either based on a dynamic programming, or a black-box application of tools developed for uncompressed strings.

Keywords: approximate pattern matching, edit distance, Lempel-Ziv.

1 Introduction

Pattern matching, which is the question of locating an occurrence of a given pattern in a text, is the most natural task as far as processing text data is concerned. Virtually any programming language contains a more or less efficient procedure for solving this problem, and any text processing application, including the widely available `grep` utility, gives users the means of solving it. While exact pattern matching is well-understood, and in particular many linear time solutions are known [5], it seems that its approximate version is less understood. Two most natural versions of the question are pattern matching with errors, where one ask for a substring of the text with small edit distance to the pattern, and pattern matching with mismatches, where one is interested in a substring with small Hamming distance to the pattern. It is known that if N is the length of the text and k is the number of allowed errors or mismatches, both problems can be solved in $\mathcal{O}(Nk)$ time [10,11], and in fact the complexity for the latter version can be improved to $\mathcal{O}(N\sqrt{k\log k})$ [2]. Under the natural assumption that the value of k

* Supported by NCN grant 2011/01/D/ST6/07164, 2011–2014.

L. Cai, S.-W. Cheng, and T.-W. Lam (Eds.): ISAAC2013, LNCS 8283, pp. 78–88, 2013.
© Springer-Verlag Berlin Heidelberg 2013

is small, one can do even better, and solve the problems in $\mathcal{O}(N + \frac{Nk^4}{m})$ [4] and $\mathcal{O}((N + \frac{Nk^3}{m}) \log k)$ [2] time complexity, respectively, which might be linear in N if k is small enough. Unfortunately, in some cases even a linear time complexity might be not good enough. This is the case when we are talking about large collections of repetitive data stored in a compressed form. Then the length of the text N might be substantially larger than the size n of its actual representation, and the goal is to achieve a running time depending on n, not N. Whether achieving such goal is possible clearly depends on the power of the compression method. In this paper we focus on the LZW/LZ78 compression [12,13], which is not as powerful as the more general LZ77 method, but still has some nice theoretical properties, and is used in real-world applications. It is known that exact LZW-compressed pattern matching can be solved very efficiently [1,6], even in the fully compressed version, where both the text and the pattern are LZW-compressed [8]. The obvious question is how efficiently can we solve approximate LZW-compressed pattern matching?

The best previously known solution by Kärkkäinen, Navarro, and Ukkonen [9], locates all occ occurrences with up to k errors using $\mathcal{O}(nmk+occ)$ time and $\mathcal{O}(nmk)$ space. More precisely, it outputs all ending positions j such that there is i for which the edit distance between $t[i..j]$ and p is at most k. In some cases, this time bound can be decreased using the idea of Bille, Fagerberg, and Gørtz [3], who presented a way to translate all uncompressed pattern matching bounds into the compressed setting. Their approach works for both the edit and Hamming distance, and by plugging the best known uncompressed pattern matching solutions, we can get:

1. $\mathcal{O}(nmk + occ)$ time and $\mathcal{O}(\frac{n}{mk} + m + occ)$ space for the edit distance,
2. $\mathcal{O}(nk^4 + nm + occ)$ time and $\mathcal{O}(\frac{n}{k^4+m} + m + occ)$ space for the edit distance,
3. $\mathcal{O}(n(k^3 + m) \log k + occ)$ time and $\mathcal{O}(\frac{n}{(k^3+m) \log k} + m + occ)$ space for the Hamming distance.

While the space complexity of the resulting algorithms is small, even for constant values of k the time complexity is $\Omega(nm)$, and in fact this is an inherent shortcoming of the approach: the best we can hope for is $\mathcal{O}(nm)$ for sufficiently small values of k, say, $k = \mathcal{O}(m^{1/3})$.

In this paper we show that in fact this barrier can be broken. We prove that for the Hamming distance, running time of $\mathcal{O}(n\sqrt{m}k^2)$ is possible, which for $k = o(m^{1/4})$ is $o(nm)$. Then we show how to extend the algorithm by building on the ideas of Cole and Hariharan [4], and achieve $\mathcal{O}(n\sqrt{m}k^3)$ for the edit distance. Both algorithms use $O(n + m)$ space. For the sake of clarity, we concentrate on the question of detecting just one occurrence, but our algorithms generalize to generating all of them.

Some of our methods are based on the concepts first used by Cole and Hariharan [4], and later by Amir, Lewenstein and Porat [2]. Applying them in the compressed setting is not just a trivial exercise, and creates new challenges. For instance, verifying whether a given position corresponds to an occurrence with no more than k mismatches in $\mathcal{O}(k)$ time is straightforward in the uncompressed setting using the suffix tree, but in our case requires some additional ideas.

We start with some basic tools in Sections 2. Then we distinguish between two types of matches, called internal and crossing. Detecting the former is relatively straightforward in both versions. To detect the latter, we reduce the question to a problem that is easier to work with, which we call pattern matching in pc-strings, see Section 3. To solve pattern matching with mismatches in pc-strings, we distinguish between two cases depending on how periodic the pattern is. For this we apply the concept of z-breaks, heavily used in the previous papers on approximate pattern matching. If there are many such breaks, or in other words the pattern is not very repetitive, we can solve the problem by reducing to a generalization of (exact) compressed pattern matching with multiple patterns, see Section 4. Otherwise, the pattern is *highly periodic*, and the situation is more complicated. In Section 5 we show how to exploit the regular structure of such pattern to construct an efficient algorithm. Then in Section 6, which is the most technical part of the paper, we speed up the method using a new technique which considers all candidates in a more global manner. Finally, in Section 7 we generalize the solution to solve the version with errors. Because of the space limitation, we omit many details, which can be found in the full version.

2 Preliminaries

We are given a text $t[1..N]$ and a pattern $p[1..m]$, both are strings over an integer alphabet Σ. We assume that $m \leq N$ and $\Sigma = \{1, 2, \ldots, N\}$. The pattern is given explicitly, but the text is described implicitly using the LZW/LZ78 compression scheme. Such scheme is defined as follows: we partition the text into n disjoint fragments $t = z_1 z_2 \ldots z_n$, where each fragment z_i is either a single letter, i.e., $z_i = c$, or a word of the form $z_i = z_j c$, where $j < i$. The fragments z_i are usually called the *codewords*, and because their set is closed under taking prefixes, we may represent it as a trie, which will be further denoted by T. Depending on how we choose the partition and encode the codewords, we get different concrete compression methods, say LZW or LZ78. Our methods do not depend on such technicalities as long as we are given T and the text is described as a list of pointers to the nodes of T representing the successive fragments.

The Hamming distance between two strings of the same length is simply the number of positions where their corresponding characters differ. The edit distance $\text{ed}(s, t)$ is the minimal number of operations necessary to transform s into t, where an operation is an insertion, replacement, or removal of a character.

The first problem we consider is *compressed pattern matching with mismatches*, where we are given a compressed representation of a text t, a pattern p, and a positive integer k. We want to find i such that the Hamming distance between $t[i..i + m - 1]$ and the pattern is at most k. We also consider *compressed pattern matching with errors*, where the goal is to find i and j such that the edit distance between $t[i..j]$ and p is at most k.

To efficiently operate on the compressed text and the pattern, we need a number of data structures. Given two subwords of the pattern s_1 and s_2 we can calculate their longest common prefix, denoted $\text{LCPref}(s_1, s_2)$, and longest common suffix,

denoted $\text{LCSuf}(s_1, s_2)$, in constant time. Given i and j, we can retrieve $z_i[j]$ in constant time. Given a chunk s_1, where a chunk is a subword of some root-to-leaf path in T, and a subword of the pattern s_2, we can calculate $\text{LCSuf}(s_1, s_2)$ in constant time and $\text{LCPref}(s_1, s_2)$ in $\mathcal{O}(\log m)$. The total preprocessing time is $\mathcal{O}(n + m)$.

We need also some basic concepts from combinatorics on words. α is a period of a string s if $s[i] = s[i+\alpha]$ holds for every $i = 1, 2, \ldots, |s| - \alpha$, or in other words we can write $s = w^i u$, where $|w| = \alpha$ and $u \neq w$ is a prefix of w. The smallest such α is called **the** period of s. If the period of s is at most $\frac{|s|}{2}$, s is periodic, and otherwise we call it a break, or $|s|$-break. A word is primitive if it cannot be represented as a nontrivial power of some other word. For every word s, there exists its unique cyclic shift s' which is lexicographically smallest, and we call s' the cyclic representative of s. For a periodic s, the cyclic representative of w corresponding to the period of s is called the canonical period of s. One of the basic results concerning periods is the periodicity lemma, which says that if q and q' are both periods of s, and $q + q' \leq |s|$, so is $\gcd(q, q')$.

3 Further Preprocessing

From now on we fix k to be the number of allowed mismatches (errors) in our problem. We will say in short that the pattern matches at some position in the text if the Hamming distance (or the edit distance) between the pattern and the fragment of the text starting at this position is at most k. It is natural to distinguish between two types of matches: *internal matches* (the pattern lies fully within a single codeword) and *crossing matches* (the pattern crosses some boundary between two codewords). The internal matches can be efficiently generated using standard tools, and we focus on detecting the crossing matches, where the situation is much more complicated. In this case the pattern crosses at least one boundary between two codewords, and it may cross a lot of them, which seems hard to deal with. Anyway, it suffices to iterate over all $n - 1$ boundaries and for each of them find all matches that cross it. After fixing such a boundary, we may concentrate only on a window of length $2m$ containing m characters to the left and m to the right. Problems arise when there are many very short codewords in some fragment of the text, because in such a case all boundaries in this fragment will create windows containing lots of codewords. This is one of the obstacles we need to tackle to construct an efficient algorithm.

We want to make now one technical assumption, which simplifies significantly some definitions and the description of the algorithm. Namely, we will assume that each letter appearing in text, appears also in the pattern. Our algorithms work in the general case after minor modifications.

The notion of a pc-string will play the main role in the rest of the paper. Note that the definition changes slightly when we want to move from mismatches to errors. Nevertheless, the change is very small, so we prefer to have just one common definition, and keep in mind that its meaning depends on the variant.

Definition 1. *Let p be a pattern and f be a string. We say that $f = v_1v_2...v_l$ is a pattern-compressed-string, in short pc-string, if:*

1. *$|f| \leq 2m$ ($|f| \leq 2m + 2k$ when we are dealing with errors) and $l \leq 4k + 5$,*
2. *v_i is a factor of p, for $i = 1, 2, \ldots, l$,*
3. *v_iv_{i+1} is not a factor of p, for $i = 1, 2, \ldots, l - 1$.*

We represent such string as a list $(a_1, b_1), (a_2, b_2), ..., (a_l, b_l)$, where $v_i = p[a_i..b_i]$.

Pc-strings are very convenient to deal with. Because no v_iv_{i+1} appears in p as a substring, we can answer any LCPref and LCSuf query between a subword of f and a subword of the pattern in constant time, as each result of such a query overlaps at most 3 v_i's, so we need at most 3 queries between factors of p.

Proposition 1. *Given a position in a pc-string f, we can verify whether the alignment of the pattern at this position results in a match in $\mathcal{O}(k)$ time.*

It turns out that finding matches crossing a fixed boundary can be reduced to one instance of pattern matching with mismatches or errors in a pc-string.

Theorem 1. *Suppose we have an algorithm solving pattern matching with k mismatches (errors) in pc-strings in $T_{PC}(m)$ time. Then we can solve pattern matching with k mismatches (errors) in LZW-compressed text in $\mathcal{O}(nk \log^2 m + m + n \cdot T_{PC}(m))$ ($\mathcal{O}(nk^2 + nk \log^2 m + m + n \cdot T_{PC}(m))$) time.*

4 Detecting Matches in Pc-Strings

In this section we concentrate on the version with mismatches and present an efficient algorithm for detecting matches in a pc-string. It will use a certain preprocessing of the pattern, which takes $\mathcal{O}(m)$ time and is performed just once in the whole solution, not every time we get a new pc-string.

We distinguish between two cases depending on the "level of periodicity" of the pattern. Let $z \geq 3$ be a parameter to be fixed later. We find in p as many disjoint z-breaks as possible, which can be done in $\mathcal{O}(m)$ time [4]. If there are just a few such breaks, the pattern can be seen as *highly periodic.* First we consider the opposite case when p contains at least $2k$ disjoint z-breaks. Then we can discard most of the starting positions, and verify all the remaining ones separately.

Lemma 1 (see [2]). *Let f be a text of length $2m$. Assume that the pattern p contains at least $2k$ disjoint z-breaks. Then there are at most $\mathcal{O}(\frac{m}{z})$ matches (with k mismatches) of p in f.*

Proof. Choose $2k$ disjoint occurrences of breaks in the pattern. Let $b_1, b_2, ..., b_r$ be all pairwise different breaks among them, with b_i occurring x_i times, so $\sum_{i=1}^{r} x_i = 2k$. Consider one break b_i, and denote the positions of the disjoint occurrences of b_i in p by $o_1, o_2, ..., o_{x_i}$. For each occurrence of b_i in the text, say at position q, we add a mark to all positions $q - o_1 + 1, q - o_2 + 1, \ldots, q - o_{x_i} + 1$ within the text. Since the distance between two different occurrences of b_i in the text is at least $\frac{z}{2}$

there will be at most $\sum_{i=1}^{r} x_i \frac{2m}{z} = \frac{4km}{z}$ marks. Consider now a position in the text where p matches with at most k mismatches. At least k of the $2k$ breaks have to match exactly, so we have at least k marks there. But there are only at most $\frac{4m}{z}$ positions with at least k marks. □

This lemma is very useful, but it does not give a method to find all these $\mathcal{O}(\frac{m}{z})$ positions. For this we need to locate all occurrences in f of up to $2k$ pattern breaks. We cannot simply use the usual multiple pattern matching algorithm, because it would cost $\Omega(m)$ time, which is too much. However, we know that there are at most $\mathcal{O}(\frac{km}{z})$ occurrences of these breaks in f. This fact, combined with an efficient algorithm for multiple pattern matching in a pc-string, which is an adaptation of the method of Gawrychowski [7], gives a solution.

Lemma 2. *We can preprocess the pattern and a collection of its disjoint z-breaks in $\mathcal{O}(m)$ time, so that later given any pc-string $f = v_1 v_2 ... v_l$ we can find all occ occurrences of the breaks in f in $\mathcal{O}(l \log m + occ)$ time.*

Theorem 2. *Suppose the pattern contains at least $2k$ disjoint z-breaks. Then pattern matching with k mismatches in pc-strings can be solved in $\mathcal{O}(k \log m + \frac{km}{z})$ time.*

Proof. First we find $2k$ disjoint z-breaks in the pattern. We want now to detect the at most $\mathcal{O}(\frac{m}{z})$ positions in f where p can potentially match. Proceeding as in the proof of Lemma 1, first choose some $2k$ disjoint z-breaks and find all their matches in f using the algorithm from Lemma 2. This costs us $\mathcal{O}(l \log m + occ) = \mathcal{O}(k \log m + \frac{km}{z})$ time. The marking phase can be done in $\mathcal{O}(\frac{km}{z})$ time. Now for each of the $\mathcal{O}(\frac{m}{z})$ positions verify whether p matches there in $\mathcal{O}(k)$ time. So we can find all matches of p in f in $\mathcal{O}(k \log m + \frac{km}{z})$ time. □

Choosing big z makes our algorithm really fast. However, the larger is z, the harder is for the pattern to contain many z-breaks. Furthermore, we cannot expect each pattern to have many z-breaks, even for small z. Therefore, we need a different algorithm for the case when p has few breaks, or is *highly periodic*. The algorithm has to take advantage of the regular structure of the pattern.

5 Basic Algorithm for Highly Periodic Patterns

In this section we assume the pattern is highly periodic. This means we can write it in the form $p = s_1 b_1 s_2 b_2 ... s_r b_r s_{r+1}$, where $r < 2k$, each b_i is a z-break and each s_i is a (possibly empty) string with period at most $\frac{z}{2}$. The fragments $s_1, s_2, ..., s_{r+1}$ are called *periodic stretches*. As in the previous section we are interested in finding a match (with at most k mismatches) of p in a pc-string f.

Below we describe how to reduce the general case to the one where the number of breaks in the text is small. A very similar reasoning can be also used in matching with errors, the only change being increasing some constants.

Lemma 3. *Suppose f is a string of length at most $2m$ and p is a pattern containing at most $2k$ disjoint z-breaks. There exists a subword f' of f having at most $6k+1$ disjoint z-breaks such that each match of p in f lies fully within f'. Moreover, such f' can be found in $\mathcal{O}(kz)$ time.*

By the discussion above we can restrict ourselves to pc-strings having at most $\mathcal{O}(k)$ disjoint z-breaks. We will give now an algorithm achieving $\mathcal{O}(zk^4)$ running time for pattern matching with k mismatches in such pc-strings. While this is not the best algorithm we have obtained, it serves well as an introduction to the more complicated $\mathcal{O}(zk^3)$ algorithm presented in the next section.

Let us summarize the situation. We are given a pattern of the form $p = s_1 b_1 ... s_r b_r s_{r+1}$ and a pc-string $f = s'_1 b'_1 ... s'_q b'_q s'_{q+1}$, where $r, q = \mathcal{O}(k)$, b's denote z-breaks and the periods of all s's are at most $\frac{z}{2}$. We will soon see that alignments of the pattern, where the pattern breaks and text breaks are not too close from each other, are nice to work with, so we handle the remaining ones separately.

Proposition 2. *There are at most $\mathcal{O}(zk^3)$ alignments of the pattern in the text such that some text break (or text endpoint) is within a distance of $z(k+1)$ from some pattern break (or pattern endpoint).*

In this (simple) version of the algorithm we just verify all these $\mathcal{O}(zk^3)$ positions in $\mathcal{O}(k)$ time per one. This results in $\mathcal{O}(zk^4)$ complexity and leaves us with the convenient case, where all distances between pattern and text breaks (or endpoints) are at least $z(k+1)$. We call such alignments *fine*, and we will soon see that a fine alignment resulting in a match has a very special structure.

Starting from now we assume that the distances between consecutive breaks in the text (and in the pattern) are at least $z(k+1)$, and otherwise group some breaks together. Our argument works also for such groups but we describe it just for breaks. Similarly, we want to assume that s_1 and s_{r+1}) are either empty or of length at least $z(k+1)$, so we extend the boundary breaks if needed.

One can easily see that there are at most $\mathcal{O}(k^2)$ intervals of consecutive fine alignments in the text. Within such an interval the order of appearance of the breaks does not change. Fix one interval and suppose we have at least one match there. We want to argue that in such a case all periodic stretches involved in this match are compatible, meaning that their canonical periods are identical, and moreover start with the same offset modulo the period.

Proposition 3. *Suppose w_1, w_2 are periodic strings with periods not exceeding $\frac{z}{2}$. If $w_1 \neq w_2$ and $|w_1| = |w_2| \geq z(k+1)$ then there are at least $k+1$ mismatches between these two words.*

Suppose there is a match at some fine alignment. Between two consecutive breaks (we consider here all pattern and text breaks) there is always a periodic portion of length at least $z(k+1)$. By Proposition 3, there must be a perfect match between the corresponding fragments. So in particular, the periods of the corresponding pattern periodic stretch and text periodic stretch agree. Considering the

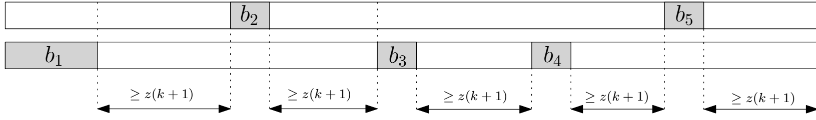

Fig. 1. Long overlaps between stretches imply their canonical periods are the same

way how the stretches overlap each other, see Figure 1, by transitivity all periodic stretches involved in the match have the same canonical period.

Suppose now all the periodic stretches in the pattern have the same canonical period u. We consider an interval of consecutive fine alignments. Assume there is a match somewhere in this interval. One can see that each two alignments i and $i+|u|$ from the interval have the same number of mismatches, because each break is aligned with a u-periodic stretch, so the fragment we compare it to is the same. So in order to find all matches within one interval, we only need to verify at most $|u| \leq \frac{z}{2}$ alignments. Each verification takes $\mathcal{O}(k)$ time, so the time taken over all intervals is $\mathcal{O}(k^2 \cdot \frac{z}{2} \cdot k) = \mathcal{O}(zk^3)$.

Theorem 3. *For highly periodic patterns, pattern matching with k mismatches in pc-strings can be solved in $\mathcal{O}(zk^4)$ time.*

6 Faster Algorithm for Highly Periodic Patterns

The purpose of this section is to show a faster algorithm for pattern matching with k mismatches in pc-strings, assuming the pattern is highly periodic. We will improve the time complexity from $\mathcal{O}(zk^4)$ to $\mathcal{O}(zk^3)$. We will make sure that the additional space required by the improved algorithm is just $\mathcal{O}(zk^2)$, which will be crucial in achieving linear space usage of the whole solution.

In the previous section we showed that one can assume that the text has at most $\mathcal{O}(k)$ disjoint z-breaks. The idea of the basic algorithm was to first work with the "bad" alignments. An alignment was considered "bad" if there was a text break and a pattern break close to each other (within a distance of $z(k+1)$). We took all such alignments and verified them in $\mathcal{O}(k)$ time each. The fine alignments (meaning not "bad") were analyzed in total time $\mathcal{O}(zk^3)$. This approach, although simple, seems to be very naive. Each time there is a single pair of close breaks, we waste $\Omega(k)$ time to deal with such an alignment. It turns out that we can verify a "bad" position in time proportional to the number of "bad" breaks. In the following definitions and lemmas we make the idea formal.

Definition 2. *In a fixed alignment of the pattern in the text, we call a pattern break black if there is some text break or text endpoint within distance $23zk$ from it. Similarly, we call a text break black if there is some pattern break or pattern endpoint within distance $23zk$ from it. Non-black breaks are called white.*

Note that one extreme case when a break is black is when it overlaps with some other break. It is convenient to deal with such situations separately. There are only

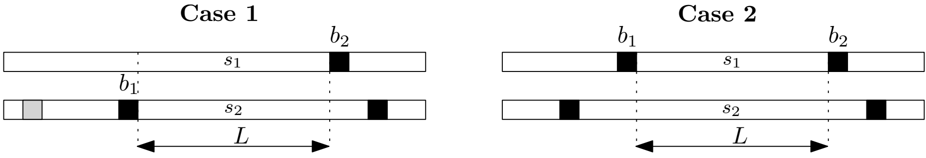

Fig. 2. Two consecutive black breaks

$\mathcal{O}(zk^2)$ such alignments, so they can be all verified in $\mathcal{O}(zk^3)$ time, and from now on we assume that no two breaks overlap. Moreover, we assume that there is at least one black break, as otherwise the alignment is fine.

Lemma 4. *After $\mathcal{O}(zk^3)$ time preprocessing, given an alignment with $B \geq 1$ black breaks we can test whether it corresponds to a match in $\mathcal{O}(B)$ time.*

We will prove the above lemma in the remaining part of this section. Suppose for a moment it holds, and consider all alignments with some black breaks. Call the number of black breaks in these alignments $B_1, B_2, ..., B_g$. Then by the above lemma, each single alignment can be processed in $\mathcal{O}(B_i)$ time, so the total time is $\mathcal{O}(\sum_{i=1}^{g} B_i)$. Every specific break is black at most $\mathcal{O}(k \cdot (46z(k+1)+2z)) = \mathcal{O}(zk^2)$ times, so $\mathcal{O}(\sum_{i=1}^{g} B_i) = \mathcal{O}(zk^3)$. So if we use this method to process the alignments, we will obtain an algorithm with $\mathcal{O}(zk^3)$ running time.

The main idea in the proof of the lemma is to partition the alignment into disjoint parts, such that in each of these parts we can count the number of mismatches easily. More precisely, if there are B black breaks in the considered alignment, we distinguish $\mathcal{O}(B)$ intervals where the Hamming distance can be determined in $\mathcal{O}(1)$ time, assuming some precalculation. We will now give the details by analyzing the relative arrangement of black and white breaks. Recall we have already reduced the situation to the case where no two breaks overlap.

Consider a periodic stretch s between two breaks in the pattern (text). It can be written in the form $s = u_1 u^i u_2$ where u is its canonical period (of length at most $\frac{z}{2}$), $i \geq 0$, u_1 is some suffix of u and u_2 is some prefix of u. Note also that the word u is primitive in such a case. It is easier to imagine the whole picture (and also to describe it) if $u_1 = u_2 = \varepsilon$, in other words when s is a power of its canonical period. We can achieve it by merging u_1 (u_2 respectively) to the neighboring break on the left (on the right). After this operation the breaks have lengths between z and $2z$ and all periodic stretches, maybe except these at the start and at the end of the word, are powers of primitive words.

Let us fix an alignment with at least one black break, and take any black pattern break (the reasoning for text breaks is the same). We want to count the number of mismatches between it and the corresponding periodic stretch from the text. To answer such a query in constant time, for each pattern break and periodic stretch u^i from the text we count mismatches between the break and the stretch for every possible shift smaller than $|u| \leq \frac{z}{2}$. Each such count can be performed in $\mathcal{O}(k)$ time, which results in $\mathcal{O}(zk^3)$ time preprocessing.

Now take two consecutive black breaks b_1, b_2. Consider the case, when there are no more breaks between them (of course there are no black ones, because we chose b_1, b_2 to be consecutive, but some white breaks might be there). Two possible situations are depicted in Figure 2. Our aim is now to count the number of mismatches between s_1 and s_2, which are length-L subwords of periodic stretches from the text and pattern, respectively. If $L \geq z(k+1)$ then by Proposition 3 either there are no mismatches between s_1 and s_2, or there are at least $k+1$ of them. It is easy to detect which case occurs: the strings agree if and only if their canonical periods are the same and they start with the same period offset, which can be determined in $\mathcal{O}(1)$ time after some straightforward preprocessing. So we can assume $L < z(k+1)$. We consider the cases from Figure 2 separately.

Case 1. In this case s_1 is length-L suffix of some text periodic stretch, s_2 is length-L prefix of some pattern periodic stretch. We want to precalculate all possible $\mathcal{O}(zk^3)$ results of such queries. Fix one pair of periodic stretches. We will calculate all the $\mathcal{O}(zk)$ required numbers in $\mathcal{O}(zk)$ total time. Let w be the canonical period of s_1, $d = |w|$ and let u be the canonical period of s_2. First calculate the answer for all overlaps of length at most d in $\mathcal{O}(dk) = \mathcal{O}(zk)$ time. Now to process an overlap of length $D > d$, we use the result for $D - d$, and add the number of mismatches between w and some factor of an infinite word u^∞, which can be previously precomputed in $\mathcal{O}(|u|k) = \mathcal{O}(zk)$ total time. Hence we can precalculate all values in $\mathcal{O}(zk^3)$ time, but space usage of $\mathcal{O}(zk^3)$ is too high to achieve linear total space complexity. It can be reduced to $\mathcal{O}(z)$ per a pair of stretches by carefully arranging some partial results so that the final answer can be computed as a difference of their prefix sums.

Case 2. In this case s_1 is a complete periodic stretch, and s_2 is a factor of a periodic stretch. Note that if s_2 has period d then there are only d essentially different alignments of such form. Overall there are only $\mathcal{O}(zk^2)$ possible queries, so we precalculate all of them in $\mathcal{O}(zk^3)$ time.

Then we need to consider the general situation when there are some white breaks between two consecutive black breaks b_1, b_2. Using a similar (although more complex) reasoning it can be solved in constant time after $\mathcal{O}(zk^2)$ space and $\mathcal{O}(zk^3)$ time preprocessing. Hence whenever we have an alignment with B black breaks, we may partition it into $\mathcal{O}(B)$ regions and either count the mismatches in each of them, or report that it exceeds k, in constant time, thus the theorem.

Theorem 4. *For highly periodic patterns, pattern matching with k mismatches in pc-strings can be solved in $\mathcal{O}(zk^3)$ time using $\mathcal{O}(zk^2)$ additional space.*

It is now a good moment to specify z. Let $z = \frac{\sqrt{m}}{k}$. Using Theorem 2 and Theorem 4 we see that such a choice of z gives us a running time $\mathcal{O}(k \log m + \sqrt{m}k^2) = \mathcal{O}(\sqrt{m}k^2)$ for pattern matching with k mismatches in pc-strings. The additional space needed is $\mathcal{O}(zk^2) = \mathcal{O}(\sqrt{m}k)$. By Theorem 1 we then obtain that pattern matching with k mismatches in LZW-compressed text can be solved in $\mathcal{O}(nk \log^2 m + m + n\sqrt{m}k^2)$, which is $\mathcal{O}(n\sqrt{m}k^2)$ because $n \geq \sqrt{m}$. The space complexity is

$\mathcal{O}(n+m+\sqrt{m}k)$. This is bounded by $\mathcal{O}(n+m)$ whenever $k=\mathcal{O}(\sqrt{m})$. In the opposite case we use the $\mathcal{O}(mk)$ algorithm [10] to process each pc-string using $\mathcal{O}(n+m)$ space and $\mathcal{O}(nmk)=\mathcal{O}(n\sqrt{m}k^2)$ total time.

Theorem 5. *Pattern matching with k mismatches in LZW-compressed strings can be solved in $\mathcal{O}(n\sqrt{m}k^2)$ time and $\mathcal{O}(n+m)$ space.*

7 Algorithm for Pattern Matching with Errors

In this section we discuss the algorithm for pattern matching with k errors in pc-strings. It is obtained by combining our methods for compressed strings (applied for pattern matching with mismatches) with the ideas used by Cole and Hariharan [4]. We need $\mathcal{O}(\frac{mk^2}{z}+k\log m)$ time for the case when p has at least $2k$ disjoint z-breaks and $\mathcal{O}(zk^4)$ for the case when p has less than $2k$ disjoint z-breaks. Choosing z to be $\frac{\sqrt{m}}{k}$ we obtain the following result.

Theorem 6. *Pattern matching with k errors in LZW-compressed strings can be solved in $\mathcal{O}(n\sqrt{m}k^3)$ time and $\mathcal{O}(n+m)$ space.*

References

1. Amir, A., Benson, G., Farach, M.: Let sleeping files lie: Pattern matching in Z-compressed files. J. Comput. Syst. Sci. 52(2), 299–307 (1996)
2. Amir, A., Lewenstein, M., Porat, E.: Faster algorithms for string matching with k mismatches. J. Algorithms 50(2), 257–275 (2004)
3. Bille, P., Fagerberg, R., Gørtz, I.L.: Improved approximate string matching and regular expression matching on Ziv-Lempel compressed texts. ACM Transactions on Algorithms 6(1) (2009)
4. Cole, R., Hariharan, R.: Approximate string matching: A simpler faster algorithm. SIAM J. Comput. 31(6), 1761–1782 (2002)
5. Crochemore, M., Rytter, W.: Jewels of stringology. World Scientific (2002)
6. Gawrychowski, P.: Optimal pattern matching in LZW compressed strings. In: Proceedings of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2011, pp. 362–372. SIAM (2011)
7. Gawrychowski, P.: Simple and efficient LZW-compressed multiple pattern matching. In: Kärkkäinen, J., Stoye, J. (eds.) CPM 2012. LNCS, vol. 7354, pp. 232–242. Springer, Heidelberg (2012)
8. Gawrychowski, P.: Tying up the loose ends in fully LZW-compressed pattern matching. In: Dürr, C., Wilke, T. (eds.) STACS. LIPIcs, vol. 14, pp. 624–635. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2012)
9. Kärkkäinen, J., Navarro, G., Ukkonen, E.: Approximate string matching on Ziv-Lempel compressed text. J. Discrete Algorithms 1(3-4), 313–338 (2003)
10. Landau, G.M., Vishkin, U.: Efficient string matching with k mismatches. Theor. Comput. Sci. 43, 239–249 (1986)
11. Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. J. Algorithms 10(2), 157–169 (1989)
12. Welch, T.A.: A technique for high-performance data compression. Computer 17(6), 8–19 (1984)
13. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Transactions on Information Theory 24(5), 530–536 (1978)