Approximate String Matching over Ziv-Lempel Compressed Text

Juha Kärkkäinen[1], Gonzalo Navarro[2*], and Esko Ukkonen[1]

[1] Dept. of Computer Science, University of Helsinki, Finland. {tpkarkka,ukkonen}@cs.helsinki.fi
[2] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl

Abstract. We present a solution to the problem of performing approximate pattern matching on compressed text. The format we choose is the Ziv-Lempel family, specifically the LZ78 and LZW variants. Given a text of length u compressed into length n, and a pattern of length m, we report all the R occurrences of the pattern in the text allowing up to k insertions, deletions and substitutions, in $O(mkn + R)$ time. The existence problem needs $O(mkn)$ time. We also show that the algorithm can be adapted to run in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time, where σ is the alphabet size. The experimental results show a speedup over the basic approach for moderate m and small k.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_1 \ldots p_m$ and a text $T = t_1 \ldots t_u$, find all the occurrences of P in T, i.e. return the set $\{|x|,\ T = xPy\}$. The complexity of this problem is $O(u)$ in the worst case and $O(u \log_\sigma(m)/m)$ on average (where σ is the alphabet size), and there exist algorithms achieving both time complexities using $O(m)$ extra space [8, 3].

A generalization of the basic string matching problem is *approximate string matching*: an error threshold $k < m$ is also given as input, and we want to report all the ending positions of text substrings which match the pattern after performing up to k character insertions, deletions and replacements on them. Formally, we have to return the set $\{|xP'|,\ T = xP'y \text{ and } ed(P, P') \leq k\}$, where $ed(P, P')$ is the "edit distance" between both strings, i.e. the minimum number of character insertions, deletions and replacements needed to make them equal. The complexity of this problem is $O(u)$ in the worst case and $O(u(k + \log_\sigma(m))/m)$ on average. Both complexities have been achieved, despite that the space and preprocessing cost is exponential in m and k in the first case and polynomial in m in the second case. The best known worst case time complexity is $O(ku)$ if the space has to be polynomial in m (see [14] for a survey).

[*] Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland (grant (grant 44449) and Fundación Andes. Also supported by Fondecyt grant 1-990627.

A particularly interesting case of string matching is related to text compression. Text compression [5] tries to exploit the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv-Lempel family [23, 24] is one of the best in practice because of their good compression ratios combined with efficient compression and decompression time.

The *compressed matching problem* was first defined in the work of Amir and Benson [1] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \dots z_n$, and a pattern P, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches (note that it could be that $R = u > n$).

The compressed matching problem is important in practice. Today's textual databases are an excellent example of applications where both problems are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. Surprisingly, these two combined requirements are not easy to achieve together, as the only solution before the 90's was to process queries by uncompressing the texts and then searching into them. In particular, *approximate* searching on compressed text was advocated in [1] as an open problem.

This is the problem we solve in this paper: we present the first solution for compressed approximate string matching. The format we choose is the Ziv-Lempel family, focusing in the LZ78 and LZW variants. By modifying the basic dynamic programming algorithm, we achieve a time complexity of $O(mkn + R)$ and a space complexity of $O(n(mk + \log n))$ bits (i.e. $O(1 + mk/\log n)$ times the memory necessary to decompress). The existence problem needs $O(mkn)$ time and space. We show that the algorithm can be adapted to run in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time, where σ is the alphabet size.

Some experiments have been conducted to assess the practical interest of our approach. We have developed a variant of LZ78 which is faster to decompress in exchange for somewhat worse compression ratios. Using this compression format our technique can take less than 70% of the time needed by decompressing and searching on the fly with basic dynamic programming for moderate m and small k values. Dynamic programming is considered as the most flexible technique to cope with diverse variants of the problem. However, decompression followed by faster search algorithms specifically designed for the edit distance still outperforms our technique, albeit those algorithms are less flexible to cope with other variants of the problem.

2 Related Work

We consider in this work Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings previously appeared. LZ77 [23] is able to reference any substring of the text already processed, while LZ78 [24] and LZW [21] reference only a single previous reference plus a new letter that is added. The first algorithm for exact searching is from 1994 [2], which searches in LZ78 needing time and space $O(m^2 + n)$.

The only search technique for LZ77 [9] is a randomized algorithm to determine in time $O(m + n \log^2(u/n))$ whether a pattern is present or not in the text (it seems that with $O(R)$ extra time they could find all the pattern occurrences).

An extension of [2] to multipattern searching was presented in [11], together with the first experimental results in this area. They achieve $O(m^2 + n)$ time and space, although this time m is the total length of all the patterns.

New practical results appeared in [16], who presented a general scheme to search on Ziv-Lempel compressed texts (simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and a new variant proposed which was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented in [12]. In [17] a new, faster, algorithm was presented based on Boyer-Moore.

The aim of this paper is to present a general solution to the approximate string matching problem on compressed text in the LZ78 and LZW formats.

3 Approximate String Matching by Dynamic Programming

We introduce some notation for the rest of the paper. A string S is a sequence of characters over an alphabet Σ. If the alphabet is finite we call σ its size. The length of S is denoted as $|S|$, therefore $S = s_1 \ldots s_{|S|}$ where $s_i \in \Sigma$. A substring of S is denoted as $S_{i \ldots j} = s_i s_{i+1} \ldots s_j$, and if $i > j$, $S_{i \ldots j} = \varepsilon$, the empty string of length zero. In particular, $S_i = s_i$. The pattern and the text, P and T, are strings of length m and u, respectively.

We recall that $ed(A, B)$, the edit distance between A and B, is the minimum number of characters insertions, deletions and replacements needed to convert A into B or vice versa. The basic algorithm to compute the edit distance between two strings A and B was discovered many times in the past, e.g. [18]. This was converted into a search algorithm much later [19]. We first show how to compute the edit distance between two strings A and B. Later, we extend that algorithm to search a pattern in a text allowing errors.

To compute $ed(A, B)$, a matrix $C_{0 \ldots |A|, 0 \ldots |B|}$ is filled, where $C_{i,j}$ represents the minimum number of operations needed to convert $A_{1 \ldots i}$ to $B_{1 \ldots j}$. This is computed as $C_{i,0} = i$, $C_{0,j} = j$, and

$$C_{i,j} = \text{if } (A_i = B_j) \text{ then } C_{i-1,j-1} \text{ else } 1 + \min(C_{i-1,j}, C_{i,j-1}, C_{i-1,j-1})$$

where at the end $C_{|A|,|B|} = ed(A, B)$.

We consider the text searching problem now. The algorithm is basically the same, with $A = P$ and $B = T$ (computing C column-wise so that $O(m)$ space is required). The only difference is that we must allow that any text position is the potential start of a match. This is achieved by setting $C_{0,j} = 0$ for all $j \in 0 \ldots u$. That is, the empty pattern matches with zero errors at any text position (because it matches with a text substring of length zero).

The algorithm then initializes its column $\mathcal{C}_{0 \ldots m}$ with the values $\mathcal{C}_i = i$, and processes the text character by character. At each new text character T_j, its column vector is updated to $\mathcal{C}'_{0 \ldots m}$. The update formula is

$$\mathcal{C}'_i \;=\; \text{if } (P_i = T_j) \text{ then } \mathcal{C}_{i-1} \text{ else } 1 + \min(\mathcal{C}'_{i-1}, \mathcal{C}_i, \mathcal{C}_{i-1})$$

With this formula the invariant that holds after processing text position j is $\mathcal{C}_i = led(P_{1 \ldots i}, T_{1 \ldots j})$, where

$$led(A,B) \;=\; \min_{i \in 1 \ldots |B|} \; ed(A, B_{i \ldots |B|})$$

that is, $\mathcal{C}_i$ is the minimum edit distance between $P_{1 \ldots i}$ and a suffix of the text already seen. Hence, all the text positions where $\mathcal{C}_m \leq k$ are reported as ending points of occurrences.

The search time of this algorithm is $O(mu)$ and it needs $O(m)$ space.

The dynamic programming matrix has a number of properties that have been used to derive better algorithms. We are interested in two of them.

Property 1 *Let A and B be two strings such that $A = A_1 A_2$. Then there exist strings B_1 and B_2 such that $B = B_1 B_2$ and $ed(A,B) = ed(A_1,B_1) + ed(A_2,B_2)$.*

That is, there must be some point inside B where its optimal comparison against A can be divided at any arbitrary point in A. This is easily seen by considering an optimal path on the C matrix that converts A into B. The path must have at least one node in each row (and column), and therefore it can be split in a path leading to the cell $(|A_1|, r)$, for some r, and a path leading from that cell to $(|A|, |B|)$. Thus, $r = |B_1|$, which determines B_1. For example $ed(\text{"}\mathtt{survey}\text{"}, \text{"}\mathtt{surgery}\text{"}) = ed(\text{"}\mathtt{surv}\text{"}, \text{"}\mathtt{surg}\text{"}) + ed(\text{"}\mathtt{ey}\text{"}, \text{"}\mathtt{ery}\text{"})$.

The second property refers to the so-called *active* cells of the $\mathcal{C}$ vector when searching P allowing k errors. All the cells before and including the last one with value $\leq k$ are called "active". As noted in [20]:

Property 2 *The output of the search depends only on the active cells, and the rest can be assumed to be $k+1$.*

Between consecutive iterations of the dynamic programming algorithm, the last active cell can be incremented at most in 1 (because neighboring cells of the C matrix differ at most in 1). Hence the last active cell can be maintained at $O(1)$ amortized time per iteration. The search algorithm needs to work only on the active cells. As conjectured in [20] and proved in [6,4], there are $O(k)$ active cells per column on average and therefore the dynamic programming takes $O(ku)$ time on average.

Considering Property 2, we use a modified version of ed in this paper. When we use $ed(A, B)$ we mean the exact edit distance between A and B if it is $\leq k$, otherwise any number larger than k can be returned. It is clear that the output of an algorithm using this definition is the same as with the original one.

4 A General Search Approach

We present now a general approach for approximate pattern matching over a text $Z = b_1 \ldots b_n$, that is expressed as a sequence of n *blocks*. Each block b_r represents a substring B_r of T, such that $B_1 \ldots B_n = T$. Moreover, each block B_r is formed by a concatenation of a previously seen blocks and an explicit letter. This comprises the LZ78 and LZW formats. Our goal is to find the positions in T where occurrences of P end with at most k errors, using Z.

Our approach is to adapt an algorithm designed to process T character by character so that it processes T block by block, using the fact that blocks are built from previous blocks and explicit letters. In this section we show how have we adapted the classical dynamic programming algorithm of Section 3. We show later that the $O(ku)$ algorithm based on active cells can be adapted as well.

We need a little more notation before explaining the algorithm. Each match is defined as either *overlapping* or *internal.* A match j is internal if there is an occurrence of P ending at j totally contained in some block B_r (i.e. if the block repeats the occurrence surely repeats). Otherwise it is an overlapping match. We also define $b^{(r)}$, for a block b and a natural number r, as follows: $b^{(0)} = b$ and $b^{(r+1)} = (b')^{(r)}$, where b' is the block referenced by b. That is, $b^{(r)}$ is the block obtained by going r steps in the backward referencing chain of b.

The general mechanism of the search is as follows: we read the blocks b_r one by one. For each new block b read, representing a string B, and where we have already processed $T_{1 \ldots j}$, we update the state of the search so that after working on the block we have processed $T_{1 \ldots j+|B|} = T_{1 \ldots j} B$. To process each block, three steps are carried out: (1) its *description* (to be specified shortly) is computed, (2) the occurrences ending inside the block B are reported, and (3) the state of the search is updated. The state of the search consists of two elements

- The last text position considered, j (initially 0).
- A vector $\mathcal{C}_i$, for $i \in 0 \ldots m$, where $\mathcal{C}_i = led(P_{1 \ldots i}, T_{1 \ldots j})$. Initially, $\mathcal{C}_i = i$. This vector is the same as for plain dynamic programming.

The description of all the blocks already seen is maintained. Say that block b represents the text substring B. Then the description of b is formed by the length $len(b) = |B|$, the referenced block $ref(b) = b^{(1)}$ and some vectors indexed by $i \in 1 \ldots m$ (their values are assumed to be $k+1$ if accessed outside bounds).

- $\mathcal{I}_{i,i'}(b) = ed(P_{i \ldots i'}, B)$, for $i \in 1 \ldots m$, $i' \in \max(i+|B|-k-1, i-1) \ldots \min(i+|B|+k-1, m)$, which at each point gives the edit distance between B and $P_{i \ldots i'}$. Note that $\mathcal{I}$ has $O(mk)$ entries per block. In particular, the set of possible i' values is empty if $i > m+k+1-|B|$, in which case $\mathcal{I}_{i,i'}(b) = k+1$.

- $\mathcal{P}_i(b) = led(P_{1\ldots i}, B)$, for $i \in 1\ldots m$, gives the edit distance between the prefix of length i of P and a suffix of B. $\mathcal{P}$ has $O(m)$ entries per block.
- $\mathcal{M}(b) = b^{(r)}$, where $r = \min\{r' \geq 0,\ \mathcal{P}_m(b^{(r')}) \leq k\}$. That is, $\mathcal{M}(b)$ is the last block in the referencing chain for b that finishes with an internal match of P. Its value is -1 if no such block exists.

Figure 1 illustrates the $\mathcal{I}$ matrix and how is it filled under different situations.

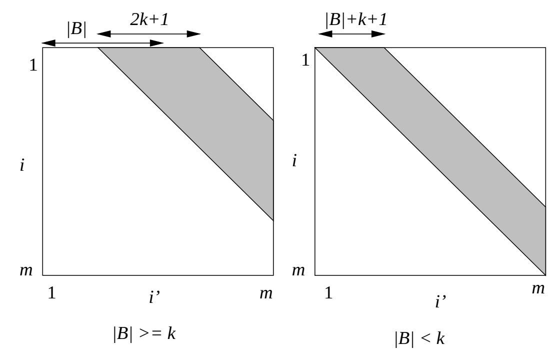

Fig. 1. The $\mathcal{I}$ matrix for a block b representing a string B.

5 Computing Block Descriptions

We show how to compute the description of a new block b' that represents $B' = Ba$, where B is the string represented by a previous block b and a is an explicit letter. The procedure is almost the same as for LZW so we omit it here and concentrate on LZ78 only. An initial block b_0 represents the string ε, and its description is: $len(b_0) = 0$; $\mathcal{I}_{i,i'}(b_0) = i' - i + 1$, $i \in 1\ldots m, i' \in i-1\ldots\min(i+k-1, m)$; $\mathcal{P}_i(b_0) = i$, $i \in 1\ldots m$; and $\mathcal{M}(b_0) = -1$.

We give now the update formulas for the case when a new letter a is added to B in order to form B'. These can be seen as special cases of dynamic programming matrices between B and parts of P.

- $len(b') = len(b) + 1$.
- $ref(b') = b$.
- $\mathcal{I}_{i,i'}(b') = \mathcal{I}_{i,i'-1}(b)$ if $a = P_{i'}$, and $1 + \min(\mathcal{I}_{i,i'}(b), \mathcal{I}_{i,i'-1}(b'), \mathcal{I}_{i,i'-1}(b))$ otherwise. We start with[1] $\mathcal{I}_{i,\max(i-1,i+|B'|-k-2)}(b') = \min(|B'|, k+1)$, and compute the values for increasing i'. This corresponds to filling a dynamic programming matrix where the characters of $P_{i\ldots}$ are the columns and the characters of B are the rows. Adding a to B is equivalent to adding a new

[1] Note that it may be that this initial value cannot be placed in the matrix because its position would be outside bounds.

row to the matrix, and we store at each block only the row of the matrix corresponding to its last letter (the rest could be retrieved by going back in the references). For each i, there are $2k+1$ such columns stored at each block B, corresponding to the interesting i' values. Figure 2 illustrates. To relate this to the matrix of $\mathcal{I}$ in Figure 1 one needs to consider that there is a three dimensional matrix indexed by i, i' and $|B|$. Figure 1 shows the plane stored at each block B, corresponding to its last letter. Figure 2 shows a plane obtained by fixing i.

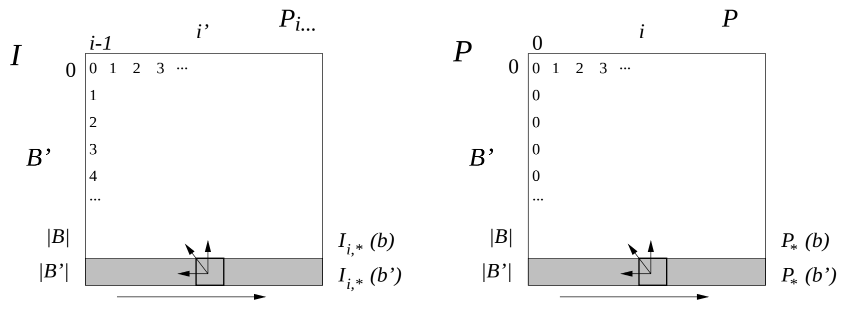

Fig. 2. The virtual dynamic programming matrices. On the left, between B and $P_{i\ldots}$, to compute $\mathcal{I}$. On the right, between B and P, to compute $\mathcal{P}$.

- $\mathcal{P}_i(b') = \mathcal{P}_{i-1}(b)$ if $a = P_i$ and $1 + \min(\mathcal{P}_i(b), \mathcal{P}_{i-1}(b'), \mathcal{P}_{i-1}(b))$ otherwise. We assume that $\mathcal{P}_0(b') = 0$ and compute the values for increasing i. This corresponds again to filling a dynamic programming matrix where the characters of P are the columns, while the characters of B are the rows. The (virtual) matrix has i at the i-th column of the first row and zeros in the first column. Figure 2 illustrates.
- $\mathcal{M}(b') = \mathcal{M}(b)$ if $\mathcal{P}_m(b') > k$, and b' otherwise. That is, if there is a new internal match ending at $|B'|$ then b' is added to the list. This takes constant time.

6 Updating the Search State

We specify now how to report the matches and update the state of the search once the description of a new block b has been computed. Three actions are carried out, in this order.

Reporting the overlapping matches. An overlapping match ending inside the new block B corresponds to an occurrence that spans a suffix of the text already seen $T_{1\ldots j}$ and a prefix of B. From Property 1, we know that if such an occurrence matches P with k errors (or less) then it must be possible to split P in $P_{1\ldots i}$ and $P_{i+1\ldots m}$, such that the text suffix matches the first half and the prefix of B matches the second half. Figure 3 illustrates.

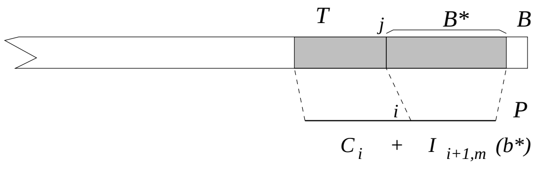

Fig. 3. Splitting of an overlapping match (grayed), where $b^* = b^{(|B|-i')}$.

Therefore, all the possible overlapping matches are found by considering all the possible positions i in the pattern. The check for a match ending at text position $j + i'$ is then split into two parts. A first condition states that $P_{1 \ldots i}$ matches a suffix of $T_{1 \ldots j}$ with k_1 errors, which can be checked using the $\mathcal{C}$ vector. A second condition states that $P_{i+1 \ldots m}$ matches $B_{1 \ldots i'}$ with k_2 errors, which can be checked using the $\mathcal{I}$ matrix of previous referenced blocks. Finally, $k_1 + k_2$ must be $\leq k$.

Summarizing, the text position $j + i'$ (for $i' \in 1 \ldots \min(m + k - 1, |B|)$) is reported if

$$\min_{i=\min(1,m-i'-k)}^{\max(m-1,m-i'+k)} (\mathcal{C}_i + \mathcal{I}_{i+1,m}(b^{(|B|-i')})) \quad \leq \quad k \tag{1}$$

where we need $ref(\cdot)$ to compute $b^{(|B|-i')}$. Additionally, we have to report the positions $j + i'$ such that $\mathcal{C}_m + i' \leq k$ (for $i' \in 1 \ldots k$). This corresponds to $\mathcal{I}_{m+1,m}(b^{(|B|-i')}) = i'$, which is not stored in that matrix.

Note that if we check the i' positions in decreasing order then the backward reference chain has to be traversed only once. So the total cost for this check is $O(mk)$. The occurrences are not immediately reported but stored in decreasing order of i' in an auxiliary array (of size at most $m + k$), because they can mix and collide with internal matches.

Reporting the internal matches. These are matches totally contained inside B. Their offsets can be retrieved by following the $\mathcal{M}(b)$ list until reaching the value -1. That is, we start with $b' \leftarrow \mathcal{M}(b)$ and while $b' \neq -1$ report the positions $j + len(b')$ and update $b' \leftarrow \mathcal{M}(b')$. This retrieves all the internal matches in time proportional to their amount, in reverse order. These matches may collide and intermingle with the overlapping matches. We merge both chains of matches and report them in increasing order and without repetitions. All this can be done in time proportional to the number of matches reported (which adds up $O(R)$ across all the search).

Updating the $\mathcal{C}$ vector and j. To update $\mathcal{C}$ we need to determine the best edit distance between $P_{1 \ldots i}$ and a suffix of the new text $T_{1 \ldots j+|B|} = T_{1 \ldots j}B$. Two choices exist for such a suffix: either it is totally inside B or it spans a suffix of $T_{1 \ldots j}$ and the whole B. Figure 4 illustrates the two alternatives. The first case corresponds to a match of $P_{1 \ldots i}$ against a suffix of B, which is computed in $\mathcal{P}$. For the second case we can use Property 1 again to see that such an occurrence is formed by matching $P_{1\dots i'}$ against some suffix of $T_{1\dots j}$ and $P_{i'+1\dots i}$ against the whole B. This can be solved by combining $\mathcal{C}$ and $\mathcal{I}$.

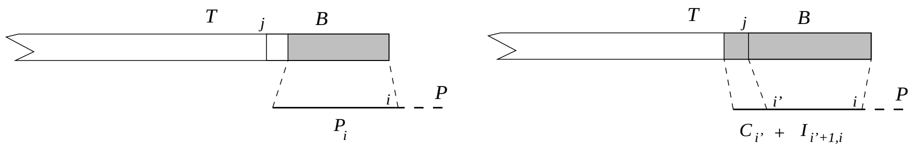

Fig. 4. Two choices to update the $\mathcal{C}$ vectors.

The formula to update $\mathcal{C}$ to a new $\mathcal{C}'$ is therefore

$$\mathcal{C}'_i \quad \leftarrow \quad \min(\mathcal{P}_i(b), \min_{i'=\max(1,i-|B|-k)}^{\min(i-1,i-|B|+k)} (\mathcal{C}_{i'} + \mathcal{I}_{i'+1,i}(b))) \tag{2}$$

which finds the correct value if it is not larger than k, and gives something larger than k otherwise (this is in accordance to our modified definition of ed). Since there are m cells to compute and each one searches over at most $2k+1$ values, the total cost to update $\mathcal{C}$ is $O(mk)$.

Finally, j is easily updated by adding $|B|$ to it.

7 Complexity

The space requirement for the algorithm is basically that to store the block descriptions. The lengths $len(\cdot)$ add up u, so in the worst case $n \log(u/n) = O(n \log n)$ bits are necessary to store them. The references to the previous blocks $ref(\cdot)$ and $\mathcal{M}(\cdot)$ need $O(\log n)$ bits per entry, for a total of $O(n \log n)$ bits. For the matrices, we observe that each element of those arrays differs from the previous one in at most 1, that is $|\mathcal{I}_{i,i'+1}(b) - \mathcal{I}_{i,i'}(b)| \leq 1$ and $|\mathcal{P}_{i+1}(b) - \mathcal{P}_i(b)| \leq 1$. Their first value is trivial and does not need to be stored. Therefore, each such cell can be represented only with 2 bits, for a total space requirement of $O(mkn)$ bits.

With respect to time complexity, all the processes described take $O(mkn)$ time for the existence problem and $O(mkn + R)$ time to report the R matches of P. The update of $\mathcal{P}$ costs $O(m)$ per block, but that of $\mathcal{I}$ takes $O(mk)$. The same happens to finding the overlapping matches and the update of $\mathcal{C}$. The handling of internal matches and merging with overlapping matches add up $O(R)$ along the total process.

8 A Faster Algorithm on Average

A simple form to speed up the dynamic programming algorithm on compressed text is based on Property 2. That is, we try to work only on the active cells of $\mathcal{C}$, $\mathcal{I}$ and $\mathcal{P}$.

We update only the active cells of $\mathcal{C}$ and $\mathcal{P}$. If we assume a random pattern P, then the property that says that there are on average $O(k)$ active cells in $\mathcal{C}$ at a random text position holds also when those text positions are the endpoints of blocks in the text. The same happens to the $\mathcal{P}$ values, since $\mathcal{P}_i(b) \geq \mathcal{C}_i$ after processing block b.

We recall the minimization formula (2) to update $\mathcal{C}$, and note that the $\mathcal{C}_{i'}$ are on average active only for $i' = O(k)$. Therefore only the values $i \in |B| \pm O(k)$ have a chance of being $\leq k$. The minimization with $\mathcal{P}_i$ does not change things because this vector has also $O(k)$ active values on average.

Therefore, updating $\mathcal{C}$ costs $O(k^2)$ per block on average. Computing $\mathcal{P}$ takes $O(k)$ time since only the active part of the vector needs to be traversed.

A more challenging problem appears when trying to apply the technique to $\mathcal{I}_{i,i'}(b)$. They key idea in this case comes from considering that $ed(P_{i\ldots i'}, B) > k$ if $|B| - (i' - i + 1) > k$, and therefore any block B such that $|B| > m + k$ cannot have any active value in $\mathcal{I}$. Since there are at most $O(\sigma^{m+k})$ different blocks of length at most $m + k$ (recall that σ is the alphabet size of the text), we can work $O(mk\sigma^{m+k})$ in total in computing $\mathcal{I}$ values. This is obtained by marking the blocks that do not have any active value in their $\mathcal{I}$ matrix, so that the $\mathcal{I}$ matrix of the blocks referencing them do not need to be computed either (moreover, the computation of $\mathcal{C}$ and overlapping matches can be simplified).

However, this bound can be improved. The set of different strings matching a pattern P with at most k errors, called $U_k(P) = \{P',\ ed(P, P') \leq k\}$, is finite. More specifically, it is shown in [20] that if $|P| = m$, then $|U_k(P)| = O((m\sigma)^k)$ at most. This limits the total number of different blocks B that can be preprocessed for a given pattern substring $P_{i\ldots i'}$. Summing over all the possible substrings $P_{i\ldots i'}$, considering that computing each such entry for each block takes $O(1)$ time, we have a total cost of $O(m^2(m\sigma)^k)$. Note that this is a worst case result, not only average case. Another limit for the total amount of work is still $O(mkn)$, so the cost is $O(\min(mkn, m^2(m\sigma)^k))$.

Finally, we have to consider the cost of processing the matches. This is $O(R)$ plus the cost to search the overlapping matches. We recall the formula (1) to find them, which can be seen to cost $O(k^2)$ only, since there are $O(k)$ active values in $\mathcal{C}$ on average and therefore $i' \in m \pm O(k)$ is also limited to $O(k)$ different values.

Summarizing, we can solve the problem on LZ78 and LZW in $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + R)$ average time. Note in particular that the middle term is asymptotically independent on n. Moreover, the space required is $O(k^2 n + \min(mkn, m^2(m\sigma)^k) + n \log n)$ bits because only the relevant parts of the matrices need to be stored.

9 Significance of the Results

9.1 Memory Requirements

First consider the space requirements. In the worst case we need $O(n(mk + \log n))$ bits. Despite that this may seem impractical, this is not so. A first consideration is that normal compressors use only a suffix ("window") of the text already seen, in order to use bounded memory independent of n. The normal mechanism is that when the number of nodes in the LZ78 trie reaches a given amount N, the trie is deleted and the compression starts again from scratch for the rest of the file. A special mark is left in the compressed file to let the decompressor know of this fact.

Our search mechanism can use the same mark to start reusing its allocated memory from zero as well, since no node seen in the past will be referenced again. This technique can be adapted to more complex ways of reusing the memory under various LZ78-like compression schemes [5].

If a compressor is limited to use N nodes, the decompression needs at the very least $O(N \log N)$ bits of memory. Since the search algorithm can be restarted after reading N blocks, it requires only $O(N(mk + \log N))$ bits. Hence the amount of memory required to search is $O(1 + mk/\log N) \times$ memory for decompression, and we recall that this can be lowered in the average case. Moreover, reasonably fast decompression needs to keep the decompressed text in memory, which increases its space requirements.

9.2 Time Complexity

Despite that ours is the first algorithm for approximate searching on compressed text, there exist also alternative approaches, some of them trivial and others not specifically designed for approximate searching.

The first alternative approach is DS, a trivial decompress-then-search algorithm. This yields, for the worst case, $O(ku)$ [10] or $O(m|U_k(P)| + u)$ [20] time, where we recall that $|U_k(P)|$ is $O((m\sigma)^k)$. For the average case, the best result in theory is $O(u + (k + \log_\sigma m)u/m) = O(u)$ [7]. This is competitive when u/n is not large, and it needs much memory for fast decompression.

A second alternative approach, OM, considers that all the overlapping matches can be obtained by decompressing the first and last $m + k$ characters of each block, and using any search algorithm on that decompressed text. The internal matches are obtained by copying previous results. The total amount of text to process is $O(mn)$. Using the previous algorithms, this yields worst case times of $O(kmn + R)$ and $O(m|U_k(P)| + mn + R)$ in the worst case, and $O((k + \log_\sigma m)n + mn + R) = O(mn + R)$ on average. Except for large u/m, it is normally impractical to decompress the first and last $m + k$ characters of each block.

Yet a third alternative, MP, is to reduce the problem to multipattern searching of all the strings in $U_k(P)$. As shown in [11], a set of strings of *total* length M can be searched in $O(M^2 + n)$ time and space in LZ78 and LZW compressed text. This yields an $O(m^2|U_k(P)|^2 + n + R)$ worst case time algorithm, which for our case is normally impractical due to the huge preprocessing cost.

Table 1 compares the complexities. As can be seen, our algorithm yields the best average case complexity for

$$k = O(\sqrt{u/n}) \quad \wedge \quad k = O(\sqrt{m}) \quad \wedge \quad \frac{\log_\sigma n}{2(1 + \log_\sigma m)} \le k + O(1) \le \frac{\log_\sigma n}{1 + \log_\sigma m}$$

where essentially the first condition states that the compressed text should be reasonably small compared to the uncompressed text (this excludes DS), the second condition states that the number of errors should be small compared to the pattern length (this excludes OM) and the third condition states that n should be large enough to make $|U_k(P)|$ not significant but small enough to make $|U_k(P)|^2$ significant (this excludes MP). This means in practice that our approach is the fastest for short and medium patterns and low error levels.

Algorithm	Worst case time	Average case time
DS	ku $m\|U_k(P)\| + u$	u
OM	$kmn + R$ $m\|U_k(P)\| + mn + R$	$mn + R$
MP	$m^2\|U_k(P)\|^2 + n$	$m^2\|U_k(P)\|^2 + n + R$
Ours	$kmn + R$	$k^2 n + \min(mkn, m^2\|U_k(P)\|) + R$

Table 1. Worst and average case time for different approaches.

9.3 Experimental Results

We have implemented our algorithm in order to determine its practical value. Our implementation does not store the matrix values using 2 bit deltas, but their full values are stored in whole bytes (this works for $k < 255$). The space is further reduced by not storing the information on blocks that are not to be referenced later. In LZ78 this discards all the leaves of the trie. Of course a second compression pass is necessary to add this bit to each compressed code. Now, if this is done then we can even not assign a number to those nodes (i.e. the original nodes are renumbered) and thus reduce the number of bits of the backward pointers. This can reduce the effect of the extra bit and reduces the memory necessary for decompression as well.

We ran our experiments on a Sun UltraSparc-1 of 167 MHz and 64 Mb of RAM. We have compressed two texts: WSJ (10 Mb of Wall Street Journal articles) and DNA (10 Mb of DNA text with lines cut every 60 characters). We use an ad-hoc LZ78 compressor which stores the pair (s, a) corresponding to the backward reference and new character in the following form: s is stored as a sequence of bytes where the last bit is used to signal the end of the code; and a is coded as a whole byte. Compression could be further reduced by better coding but this would require more time to read the compressed file. The extra bit indicating whether each node is going to be used again or not is added to s, i.e. we code $2s$ or $2s + 1$ to distinguish among the two possibilities.

Using the plain LZ78 format, WSJ was reduced to 45.02% of its original size, while adding the extra bit to signal not referenced blocks raised this percentage to 45.46%, i.e. less than 1% of increment. The figures for DNA were 39.69% and

40.02%. As a comparison, Unix *Compress* program, an LZW compressor that uses bit coding, obtained 38.75% and 27.91%, respectively.

We have compared our algorithm against a more practical version of DS, which decompresses the text on the fly and searches over it, instead of writing it to a new decompressed file and then reading it again to search. The search algorithm used is that based on active columns (the one we adapted). This gives us a measure of the improvement obtained over the algorithm we are transforming.

It is also interesting to compare our technique against decompression plus searching using the best available algorithm. For this alternative (which we call "Best" in the experiments) we still use our compression format, because it decompresses faster than Gnu *gzip* and Unix *compress*. Our decompression times are 2.09 seconds for WSJ and 1.80 for DNA. The search algorithms used are those of [15, 4, 13], which were the fastest for different m and k values in our texts.

On the other hand, the OM-type algorithms are unpractical for typical compression ratios (i.e. u/n at most 10) because of their need to keep count of the $m + k$ first and last characters of each block. The MP approach does not seem practical either, since for $m = 10$ and $k = 1$ it has to generate an automaton of more than one million states at the very least. We tested the code of [11] on our text and it took 5.50 seconds for just one pattern of $m = 10$, which outrules it in our cases of interest.

We tested $m = 10$, 20 and 30, and $k = 1$, 2 and 3. For each pattern length, we selected 100 random patterns from the text and used the same patterns for all the algorithms. Table 2 shows the results.

	WSJ								
k	Ours	DS	Best	Ours	DS	Best	Ours	DS	Best
	$m = 10$	$m = 10$	$m = 10$	$m = 20$	$m = 20$	$m = 20$	$m = 30$	$m = 30$	$m = 30$
1	3.77	4.72	2.40	3.23	4.64	2.28	3.08	4.62	2.27
2	5.63	5.62	2.74	4.72	5.46	2.42	6.05	5.42	2.33
3	11.60	6.43	3.64	9.17	6.29	2.75	13.56	6.22	2.44

	DNA								
k	Ours	DS	Best	Ours	DS	Best	Ours	DS	Best
	$m = 10$	$m = 10$	$m = 10$	$m = 20$	$m = 20$	$m = 20$	$m = 30$	$m = 30$	$m = 30$
1	3.91	5.21	2.66	2.49	5.08	2.46	2.57	5.06	2.51
2	6.98	6.49	2.88	3.81	6.31	2.91	5.02	6.28	2.71
3	11.51	8.91	3.08	9.28	7.51	3.24	15.35	7.50	3.18

Table 2. CPU times to search over the WSJ and DNA files.

As the table shows, we can actually improve over the decompression of the text and the application of the *same* search algorithm. In practical terms, we can search the original file at about 2.6 . . . 4.0 Mb/sec when $k = 1$, while the time keeps reasonable and competitive for $k = 2$ as well. Moreover, DS needs for fast decompression to store the uncompressed file in main memory, which could pose a problem in practice.

On the othe hand, the "Best" option is faster than our algorithm, but we recall that this is an algorithm specialized for edit distance. Dynamic programming is unbeaten in its flexibility to accommodate other variants of the approximate string matching problem.

10 Conclusions

We have presented the first solution to the open problem of approximate pattern matching over Ziv-Lempel compressed text. Our algorithm can find the R occurrences of a pattern of length m allowing k errors over a text compressed by LZ78 or LZW into n blocks in $O(kmn + R)$ worst-case time and $O(k^2n + \min(mkn, m^2(m\sigma)^k) + R)$ average case time. We have shown that this is of theoretical and practical interest for small k and moderate m values.

Many theoretical and practical questions remain open. A first one is whether we can adapt an $O(ku)$ worst case time algorithm (where u is the size of the uncompressed text) instead of the dynamic programming algorithm we have selected, which is $O(mu)$ time. This could yield an $O(k^2n + R)$ worst-case time algorithm. Our efforts to adapt one of these algorithms [10] yielded the same $O(mkn + R)$ time we already have.

A second open question is how can we improve the search time in practice. For instance, we have not implemented the version that stores 2 bits per number, which could reduce the space. The updates to $\mathcal{P}$ and $\mathcal{I}$ could be done using bit-parallelism by adapting [13]. We believe that this could yield improvements for larger k values. On the other hand, we have not devised a bit-parallel technique to update $\mathcal{C}$ and to detect overlapping matches. Another idea is to map all the characters not belonging to the pattern to a unique symbol at search time, to avoid recomputing similar states. This, however, requires a finer tracking of the trie of blocks to detect also descendants of similar states. This yields a higher space requirement.

A third question is if faster filtration algorithms can be adapted to this problem without decompressing all the text. For example, the filter based in splitting the pattern in $k + 1$ pieces, searching the pieces without errors and running dynamic programming on the text surrounding the occurrences [22] could be applied by using the multipattern search algorithm of [11]. In theory the complexity is $O(m^2 + n + ukm^2/\sigma^{\lfloor m/(k+1) \rfloor})$, which is competitive for $k < m/\Theta(\log_\sigma(u/n) + \log_\sigma m)$.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, 52(2):299–307, 1996. Earlier version in *Proc. SODA'94*.

3. A. Apostolico and Z. Galil. *Pattern Matching Algorithms.* Oxford University Press, Oxford, UK, 1997.
4. R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.
5. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.
6. W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.
7. W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.
8. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.
9. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998. Previous version in *STOC'95.*
10. Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.
11. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, pages 103–112, 1998.
12. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Shift-And approach to pattern matching in LZW compressed text. In *Proc. CPM'99*, LNCS 1645, pages 1–13, 1999.
13. G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13, 1998.
14. G. Navarro. A guided tour to approximate string matching. Technical Report TR/DCC-99-5, Dept. of Computer Science, Univ. of Chile, 1999. To appear in *ACM Computing Surveys.* `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/-survasm.ps.gz`.
15. G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.
16. G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. CPM'99*, LNCS 1645, pages 14–36, 1999.
17. G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS, 2000. In this same volume.
18. S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. of Molecular Biology*, 48:444–453, 1970.
19. P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.
20. E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.
21. T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.
22. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.
23. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
24. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.