Approximate String Matching on Ziv-Lempel Compressed Text

Juha Kärkkäinen* Gonzalo Navarro† Esko Ukkonen†

Abstract

We present the first nontrivial algorithm for approximate pattern matching on compressed text. The format we choose is the Ziv-Lempel family. 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. On LZ78/LZW we need $O(mkn + R)$ time in the worst case and $O(k^2 n + mk \min(n, (m\sigma)^k) + R)$ on average where σ is the alphabet size. The experimental results show a practical speedup over the basic approach of up to $2X$ for moderate m and small k. We extend the algorithms to more general compression formats and approximate matching models.

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 the characters are independent and uniformly distributed over an alphabet of size σ), and there exist algorithms achieving both time complexities using $O(m)$ extra space [CR94, AG97].

A generalization of the basic string matching problem is *approximate string matching*: an error threshold k is also given as input, and we want to report all the ending positions of text substrings which match the pattern after performing a number of operations on them whose total cost cannot exceed k. 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.

Different models for edit distance fit different applications. We deal in this paper with a rather general one: the operations permitted are character insertions, deletions and substitutions. A different nonnegative cost can be assigned to the operations depending on the involved characters. Two popular specializations of this model are the Levenshtein distance (where each insertion, deletion and substitution costs 1) and the Hamming distance (where each substitution costs 1 and insertions and deletions cost ∞, i.e. they are not allowed). In these two cases the problem makes sense for $k < m$.

A lot of study has been carried out on the Levenshtein distance. The complexity of the search problem is for this case $O(u)$ 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 or k in the first case and (high-degree) 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 [Nav01] for a survey).

A particularly interesting case of string matching is related to text compression. Text compression [BCW90] tries to exploit the redundancies of the text to represent it using less space. There

*Dept. of Computer Science, University of Helsinki. {tpkarkka,ukkonen}@cs.helsinki.fi.

†Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl. Supported in part by Fondecyt grant 1-020831.

are many different compression schemes, among which the Ziv-Lempel family [ZL77, ZL78] is one of the most popular in practice because of its good compression ratios combined with efficient compression and decompression time.

The *compressed matching problem* was first defined in [AB92] as the task of performing string matching in a compressed text without decompressing it. Given a text $T = t_1 \ldots t_u$, a corresponding compressed string $Z = z_1 \ldots z_n$, and a pattern $P = p_1 \ldots p_m$, 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, I/O and network time, and they should be efficiently searched. However, these two combined requirements are not easy to achieve together, as the only solution before the 90s was to process queries by uncompressing the texts and then searching them. In particular, *approximate* searching on compressed text was advocated in [AB92] as an open problem.

This is our focus in this paper. We present the first solution to the problem of compressed approximate string matching. The format we choose is the Ziv-Lempel family, first focusing on the LZ78 and LZW variants and on the Levenshtein distance, and later extending the results to more general scenarios, such as a more general Ziv-Lempel format proposed in [NR99] which we call "LZ-Blocks" in this paper, collage systems [KST$^+$99], and general edit distance with different costs. Table 1 summarizes our contribution on the different formats and cost models. The value k_{ids} is defined as k divided by the minimum cost of an edit operation (hence $k_{ids} = k$ on the Levenshtein and Hamming models), while k_{id} is defined as $1 + 2k/$(minimum cost of an insertion or a deletion) (hence $k_{id} = 1 + 2k$ on the Levenshtein distance and $k_{id} = 1$ for Hamming). The table gives the complexities for these two cases separately anyway. For more details about the meaning of the results we refer the reader to the body of the paper.

To assess the practical impact of our methods, we implemented the Levenshtein search over the LZ78 format. We wrote our own compressor, which in exchange of about 10% increase in the size of the compressed file permits faster searching. The experimental results show that this technique can take less than half of the time needed by the basic approach, for moderate m and small k values. This paper is an extended and updated version of [KNU00].

2 Related Work

Two classes of techniques exist to compress text. The first ones, called *static* (or semi-static) methods, choose a fixed mapping from symbols or sequences in T to symbols or sequences in Z, and apply the same mapping across all the compression process. The second ones, called *adaptive* methods, modify the mapping as the compression goes on.

Some compressed text search techniques focus on static methods. Efficient solutions based on Huffman coding [Huf52] on words have been presented in [MNZBY00], but they need that the text contains natural language and be large (say, 10 Mb or more). Moreover, they allow only searching for whole words and phrases. For general texts, diverse techniques related to byte-pair

Distance	LZ78/LZW	LZ-Blocks	Collage systems
General			
w.c.	$mk_{id}n$	$mk_{id}^2 n\alpha(n)$	$mk_{id}^2\|D\| + mk_{id}\|S\|$
a.c.	$k_{ids}k_{id}n + mk_{id}\min(n,(m\sigma)^{k_{ids}})$		
Levenshtein			
w.c.	mkn	$mk^2 n\alpha(n)$	$mk^2\|D\| + mk\|S\|$
a.c.	$k^2n + mk\min(n,(m\sigma)^k)$		
Hamming			
w.c.	mn	$mn\alpha(n)$	$m(\|D\| + \|S\|)$
a.c.	$kn + m\min(n,(m\sigma)^k)$		

Table 1: The search complexities (worst and average case) obtained for different models. We excluded "$+R$" from all the complexities and $\alpha(n)$ denotes the inverse of $A(2n,n)$ (Ackermann's function).

encoding (i.e. replacing frequent bigrams by unused characters) have been shown to be efficient [Man97, SMT$^+$00]. However, in general the compression ratios obtained are poor, i.e. inferior or similar to a classical Huffman coding of the text. Moreover, in all these compression formats $n = \Theta(u)$, so the speedups can only be measured in practical terms.

A second line of research considers adaptive schemes such as Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings previously appeared. LZ77 [ZL77] is able to reference any substring of the text already processed and has a best case of $n = O(\log u)$, while LZ78 [ZL78] and LZW [Wel84] reference only a single previous reference plus a new letter that is added, with a best case of $n = O(\sqrt{u})$. A hybrid among these is LZ-Blocks, which was proposed in [NR99] to achieve the search time of LZ78 and the compression ratio of LZ77.

The LZ family is extremely popular because of its general applicability, good compression ratios, and fast compression/decompression time. String matching in Ziv-Lempel compressed texts is, however, much more complex than on many static schemes, because the pattern can appear in different forms across the compressed text.

The first algorithm, from 1994 [ABF96], presents a compressed matching algorithm for LZ78 working in time and space $O(m^2 + n)$ for the existence problem (i.e. determine whether or not P appears in T). The only technique for LZ77 [FT98] is a randomized algorithm taking time $O(m + n\log^2(u/n))$ for the existence problem.

An extension of [ABF96] to multipattern searching was presented in [KTS$^+$98], together with the first experimental results in this area. They achieve $O(m^2 + n)$ time and space for the existence problem, although this time m is the total length of all the patterns.

New practical results appeared in [NR99], which presented a general scheme to search Ziv-Lempel compressed texts (for simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and LZ-Blocks, proposed there. A similar result, restricted to simple patterns and to the LZW format, was independently found and presented in [KTS$^+$99]. A Boyer-Moore type algorithm for LZ78/LZW was presented in [NT00], which is currently the fastest in practice for moderately long patterns.

An interesting abstraction of the existing algorithms over a general compression format called *collage systems* was presented in [KST$^+$99].

Approximate string matching on compressed text was advocated in [AB92]. It has been solved for Huffman coding of words [MNZBY00] by searching the uncompressed text vocabulary, but the solution is limited to search for a whole word and retrieve whole words that are similar, on natural language texts. The problem has also been solved for the simpler Hamming distance on LZ78 at $O(nmk^2 \log(k)/w + R)$ worst case time [NR98], where w is the length in bits of the machine word.

The aim of this paper is to present the first general solution to this problem for the Ziv-Lempel family and the so-called *regular collage systems*. The specialization of this solution to LZ78/LZW and Levenshtein distance first appeared in [KNU00], of which this work is an extended version. Shortly after [KNU00], an alternative solution was presented in [MKT$^+$00]. This alternative solution, based on bit parallelism, is restricted to solve the existence problem for the Levenshtein distance.

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, $S_i = s_i$. P and T, the pattern and the text, are strings of length m and u, respectively.

We recall that $ed(A, B)$, the edit distance between strings A and B, is the minimum total cost of the operations necessary to convert A into B or vice versa (the costs are usually symmetric). The basic algorithm to compute the edit distance between two strings A and B was discovered many times in the past, e.g. [NW70]. This was converted into a search algorithm much later [Sel80]. We first show how to compute the edit distance between two strings A and B. Later, we extend that algorithm to search for the approximate occurrences of a pattern in a text.

3.1 Computing the Edit Distance

Let us call $c(\varepsilon \rightarrow a)$ the cost to insert a character a, $c(a \rightarrow \varepsilon)$ that to delete a and $c(a \rightarrow b)$ that to replace a by b. It is assumed that no character of the strings to convert is operated upon more than once, so for consistency a triangular inequality has to hold: $c(a \rightarrow c) \leq c(a \rightarrow b) + c(b \rightarrow c)$. It has also to hold that $c(a \rightarrow a) = 0$.

The algorithm to compute edit distance is based on dynamic programming. To compute $ed(A, B)$, a matrix $C_{0 \ldots |A|, 0 \ldots |B|}$ is filled, where $C_{i,j}$ represents the minimum cost of the operations needed to convert $A_{1 \ldots i}$ into $B_{1 \ldots j}$. This is computed as follows

$$\begin{aligned} C_{0,0} &= 0 \\ C_{i,j} &= \min(C_{i-1,j-1} + c(a_i \rightarrow b_j), C_{i-1,j} + c(a_i \rightarrow \varepsilon), C_{i,j-1} + c(\varepsilon \rightarrow b_j)) \end{aligned}$$

where at the end $C_{|A|,|B|} = ed(A, B)$. It is assumed that C has the value ∞ when accessed outside bounds.

The rationale of the above formula is as follows. First, $C_{0,0}$ represents the edit distance between two empty strings. For two non-empty strings of length i and j, we assume inductively that all the edit distances between shorter strings have already been computed, and try to convert $A_{1\ldots i}$ into $B_{1\ldots j}$. Three choices exist, according to the three edit operations we are considering. We can substitute a_i by b_j and then proceed in the best possible way to convert $A_{1\ldots i-1}$ into $B_{1\ldots j-1}$. We can also delete a_i and convert, in the best way, $A_{1\ldots i-1}$ into $B_{1\ldots j}$. Finally, we can insert b_j at the end of $A_{1\ldots i}$ and convert, in the best way, $A_{1\ldots i}$ into $B_{1\ldots j-1}$. In all cases, the cost to convert the rest is already computed.

The above formula is simplified when we use the Levenshtein distance:

$$\begin{aligned} C_{i,0} &= i, \quad C_{0,j} \;=\; j \\ C_{i,j} &= \text{if } (a_i = b_j) \text{ then } C_{i-1,j-1} \text{ else } 1 + \min(C_{i-1,j-1}, C_{i-1,j}, C_{i,j-1}) \end{aligned}$$

The dynamic programming algorithm must fill the matrix in such a way that the upper, left, and upper-left neighbors of a cell are computed prior to computing that cell. This is easily achieved by either a row-wise left-to-right traversal or a column-wise top-to-bottom traversal. Figure 1 (left) illustrates this algorithm to compute ed(`"survey"`, `"surgery"`) under the Levenshtein distance.

		s	u	r	g	e	r	y
	0	1	2	3	4	5	6	7
s	1	**0**	1	2	3	4	5	6
u	2	1	**0**	1	2	3	4	5
r	3	2	1	**0**	1	2	3	4
v	4	3	2	1	**1**	2	3	4
e	5	4	3	2	2	**1**	**2**	3
y	6	5	4	3	3	2	2	**2**

		s	u	r	g	e	r	y
	0	**0**	**0**	**0**	**0**	**0**	**0**	**0**
s	1	**0**	**1**	**1**	**1**	**1**	**1**	**1**
u	2	**1**	**0**	**1**	**2**	**2**	2	2
r	3	2	**1**	**0**	**1**	**2**	2	3
v	4	3	2	**1**	**1**	**2**	3	3
e	5	4	3	2	2	**1**	2	3
y	6	5	4	3	3	2	2	2

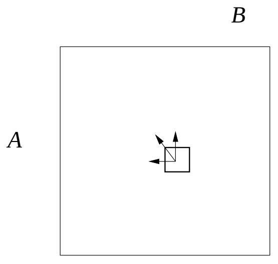

Figure 1: On the left, the dynamic programming algorithm to compute the Levenshtein distance between `"survey"` and `"surgery"`. The bold entries show the path to the final result. At the center, the variation to search for `"survey"` the text `"surgery"`. All the entries up to the final active cells for $k = 1$ are in boldface. On the right, the dependency scheme between cells.

Therefore, the algorithm is $O(|A||B|)$ time in the worst and average case. However, the space required is only $O(\min(|A|, |B|))$. This is because, in the case of a column-wise processing, only the previous column must be stored in order to compute the new one, and therefore we just keep one column and update it. We can process the matrix row-wise or column-wise so that the space requirement is minimized.

On the other hand, the sequences of operations performed to transform A into B can be easily recovered from the matrix, simply by proceeding from the cell $C_{|A|,|B|}$ to the cell $C_{0,0}$ following the path (i.e. sequence of operations) that matches the update formula (multiple paths may exist). In this case, however, we need to store the complete matrix or at least an area around the main diagonal. Therefore, for each alignmnent there exists at least one optimal path of edit steps from cell $(0, 0)$ to cell $(|A|, |B|)$.

3.2 Approximate Text Searching

We show now how to adapt this algorithm to search for a short pattern P a long text T. We recall that the problem is: given a pattern P of length m, a text T of length u, and an error level k, find all the text positions j such that $ed(P, T_{j'\ldots j}) \leq k$ for some j'. We will call "matches" the ending positions of the occurrences (i.e. the j values).

The algorithm is basically the same, with $A = P$ and $B = T$ (proceeding 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

$$\mathcal{C}_0 \;=\; 0 \;\;,\;\; \mathcal{C}_i \;=\; \mathcal{C}_{i-1} + c(p_i \rightarrow \varepsilon)$$

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 for $i > 0$ is

$$\mathcal{C}'_i \;=\; \min(\mathcal{C}_{i-1} + c(p_i \rightarrow t_j), \mathcal{C}'_{i-1} + c(p_i \rightarrow \varepsilon), \mathcal{C}_i + c(\varepsilon \rightarrow t_j))$$

which for the Levenshtein distance reduces to

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

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 matches (ending points of occurrences).

The search time of this algorithm is $O(mu)$ and its space requirement is $O(m)$. Figure 1 (center) exemplifies.

3.3 Some Properties and Definitions

We make a few definitions that are useful to analyze the efficiency of the algorithms and to relate different error models.

Definition 1 *Let k_{ids} be the maximum number of operations that can be carried out to convert A into B with error threshold k. That is*

$$k_{ids} \;=\; \left\lfloor \frac{k}{\min\{c(x \rightarrow y),\; x \neq y \in \Sigma \cup \{\varepsilon\}\}} \right\rfloor$$

The definition is useful when there are no zero costs, otherwise we could use $k_{ids} = |A| + |B|$. Note that $k_{ids} = k$ for Hamming and Levenshtein distances.

Definition 2 *Let k_{id} be one plus two times the maximum difference in lengths between two strings A and B which are at distance k. That is*

$$k_{id} \quad = \quad 1 \; + \; 2 \left\lfloor \frac{k}{\min\{c(x \to \varepsilon), c(\varepsilon \to x), \;\; x \in \Sigma\}} \right\rfloor$$

Again, $k_{id} = \infty$ if there are zero costs for insertion or deletion. Note that $k_{id} = 1 + 2k$ for the Levenshtein distance and $k_{id} = 1$ for Hamming. That is, the occurrences of a pattern of length m have length between $m - k$ and $m + k$ under the Levenshtein model, and exactly m under Hamming. This property is important for our complexity results.

On the other hand, 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 that converts A into B. The path must have at least one node in each row (and column), and therefore it can be split into 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 $ed($`"survey"`,`"surgery"`$) = ed($`"surv"`,`"surg"`$) + ed($`"ey"`,`"ery"`$)$.

Note that this property depends on our choice of operations. For example, it is not true anymore if we introduce the *transposition*, which allows us to switch adjacent characters in just one step. If transpositions cost 1, then $ed($`"survey"`,`"suvrey"`$) = 1$, yet we cannot split the first string into `"sur"` and `"vey"` and obtain the same result as before.

The second property refers to the so-called *active* cells of the $\mathcal{C}$ vector when searching for P allowing k errors. All the cells with value $\leq k$ are called "active". As noted in [Ukk85]:

Property 2 *The output of the search depends only on the active cells, and the rest can be assumed to have any value larger than k.*

Under the Levenshtein distance it holds that, from an iteration of the dynamic programming algorithm to the next, the *last* active cell can be incremented at most by 1, because neighboring cells of the matrix differ at most by 1. Hence the position of the last active cell can be maintained at $O(1)$ amortized time per iteration. That is, for each new column computed we have to check whether it has grown by 1 or whether it has decreased arbitrarily.

The search algorithm needs to work only on the active cells. As conjectured in [Ukk85] and proved in [CL92, BYN99], there are $O(k)$ active cells on average and therefore the dynamic programming takes $O(ku)$ time on average. Figure 1 (center) illustrates. This can be generalized to arbitrary costs, to obtain $O(k_{ids} u)$ search 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 The Ziv-Lempel Compression Format

The general idea of Ziv-Lempel compression is to replace substrings in the text by a pointer to a previous occurrence of them. If the pointer takes less space than the string it is replacing, compression is obtained. Different variants over this type of compression exist, see for example [BCW90]. We are particularly interested in two formats, which we describe more in depth.

4.1 LZ78 and LZW Compression

The Ziv-Lempel compression algorithm named LZ78 [ZL78] is based on a dictionary of blocks, in which we add every new block computed. At the beginning of the compression, the dictionary contains a single block b_0 of length 0. The current step of the compression is as follows: if we assume that a prefix $T_{1...j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \dots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1...u}$ which is a block of the dictionary. Once we found this block, say b_s of length ℓ_s, we construct a new block $b_{r+1} = (s, T_{j+\ell_s+1})$, we write the pair at the end of the compressed file Z, i.e $Z = b_1 \dots b_r b_{r+1}$, and we add the block to the dictionary. It is easy to see that this dictionary is prefix-closed (i.e. any prefix of an element is also an element of the dictionary) and a natural way to represent it is a trie.

We give as an example the compression of the word *ananas* in Figure 2. The first block is $(0, a)$, and next $(0, n)$. When we read the next a, a is already the block 1 in the dictionary, but an is not in the dictionary. So we create a third block $(1, n)$. We then read the next a, a is already the block 1 in the dictionary, but as do not appear. So we create a new block $(1, s)$.

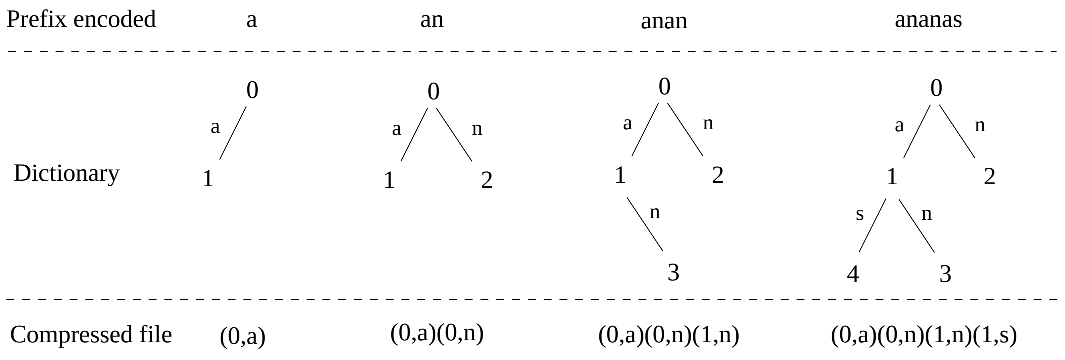

Figure 2: Compression of the word *ananas* with the algorithm LZ78.

The compression algorithm is $O(u)$ time in the worst case and efficient in practice if the dictionary is stored as a trie, which allows rapid searching of the new text prefix (for each character of T we move once in the trie). The decompression needs to build the same dictionary (the pair that defines the block r is read at the r-th step of the algorithm), although this time an array implementation is preferable over a trie based one. Compared to LZ77, the compression is rather fast but decompression is slow.

Many variations on LZ78 exist, which deal basically with the best way to code the pairs in the compressed file, or with the best way to compress using limited memory. A particularly interesting variant is from Welch, called LZW [Wel84]. In this case, the extra letter (second element of the pair) is not coded, but it is taken as the first letter of the next block (the dictionary is started with one block per letter). LZW is used by Unix's *Compress* program.

In this paper we do not consider LZW separately but just as a coding variant of LZ78. This is because the final letter of LZ78 can be readily obtained by keeping count of the first letter of each block (this is copied directly from the referenced block) and then looking at the first letter of the next block.

4.2 LZ-Blocks Format

LZ78 does not achieve as good performance as the LZ77 compression format. As noted in [NR99, KST$^+$99], searching in LZ77 compressed text is very difficult. In [NR99] they propose a format which is a hybrid, achieving a compression ratio better than LZ78 and keeping the same search efficiency. We describe that format now.

Assume that a prefix $T_{1...j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \dots b_r$. We look now for the longest prefix v of $T_{j+1...u}$ which is represented by a sequence $b_s \dots b_{s+h}$ already present in the compressed file. If there are many alternative choices for the same v, the one with the minimum of blocks is used (to reduce the cost of concatenations). And if still several possibilities occur, the first occurrence is selected, to code smaller numbers. This new block is coded as (s,h). If v is empty (i.e the letter t_{j+1} is new), a special block $(0, t_{j+1})$ is coded. With the same example *ananas*, we obtain: $(0,a)$ *nanas*; $(0,a)(0,n)$ *anas*; $(0,a)(0,n)(1,1)$ *as*; $(0,a)(0,n)(1,1)(1,0)$ *s*; $(0,a)(0,n)(1,1)(1,0)(0,s)$.

The compression can still be performed in $O(u)$ time by using a sparse suffix tree [KU96] where only the block beginnings are inserted and when we fall out of the trie we take the last node visited which corresponds to a block ending. Decompression needs to keep track of the blocks already seen to be able to retrieve the appropriate text. The compression ratio is between those of LZ77 and LZ78.

A particular case of this format is presented by Miller and Wegman [MW85], where the new block is not the previous one plus the first letter of the new one, but simply the concatenation of the previous and the new one.

4.3 Collage Systems

Many compression formats have been unified in [KST$^+$99] under the concept of *collage system*. This model divides the compression format into two parts: a *dictionary* D which stores the set of symbols that can be used in the compressed text, and the compressed text S itself, which is a sequence of elements in D. The Ziv-Lempel format interleaves the representations of D and S, since a new element of D is created after each symbol of S is output. In simpler formats, such as Huffman, the dictionary is the set of bit streams that represent each text character.

Collage systems are classified according to the type of operations that can be applied to build D. Atomic elements, concatenation of other elements in D, repetition and truncation of an element in D are the operations considered in [KST$^+$99]. In particular, atomic elements and concatenation (which are the allowed operations in the so-called *regular collage systems*) are enough to encompass LZ78/LZW and LZ-Blocks, while LZ77 requires truncation and this complicates the work of compressed pattern matching algorithms.

5 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 previously seen blocks and/or explicit letters. Our goal is to find the positions in T where occurrences of P with at most k errors end, using Z.

For simplicity of the exposition we concentrate on the Levenshtein model. Later we show how the algorithm can be extended.

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 we have adapted the classical dynamic programming algorithm. Part of the algorithm depends on the specific compression format used, and this is covered in the following sections. We also 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 either as *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.

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, except that all cells whose value is larger than k can have any value larger than k (recall Property 2).

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 $|B|$ 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 minimum edit distance between the prefix of length i of P and a suffix of B. Note that $\mathcal{P}$ has $O(m)$ entries per block.

- $\mathcal{S}_{i,i'}(b) = ed(P_{i\ldots m}, B_{1\ldots i'})$, for $i \in 1\ldots m,\ i' \in \max(m-i+1-k, 0)\ldots\min(m-i+1+k, |B|)$, gives the edit distance between the suffix starting at i in P and the prefix ending at i' in B. Note that $\mathcal{S}$ has $O(mk)$ entries per block. Again, the set of possible i' values is empty (and $\mathcal{S}_{i,i'}(B) = k+1$) if $i < m+1-k-|B|$.
- $\mathcal{M}(B)$, which is an array storing the internal matches of B (its size ranges from zero to $|B|$). The offsets with respect to the beginning of B are stored, in increasing order. The total number of entries over all the blocks is the number of internal matches found, which cannot exceed R.

Figure 3 illustrates the matrices $\mathcal{I}$ and $\mathcal{S}$ and how are they filled under different situations. The $\mathcal{P}$ and $\mathcal{M}$ vectors are simpler and hence excluded from the figure.

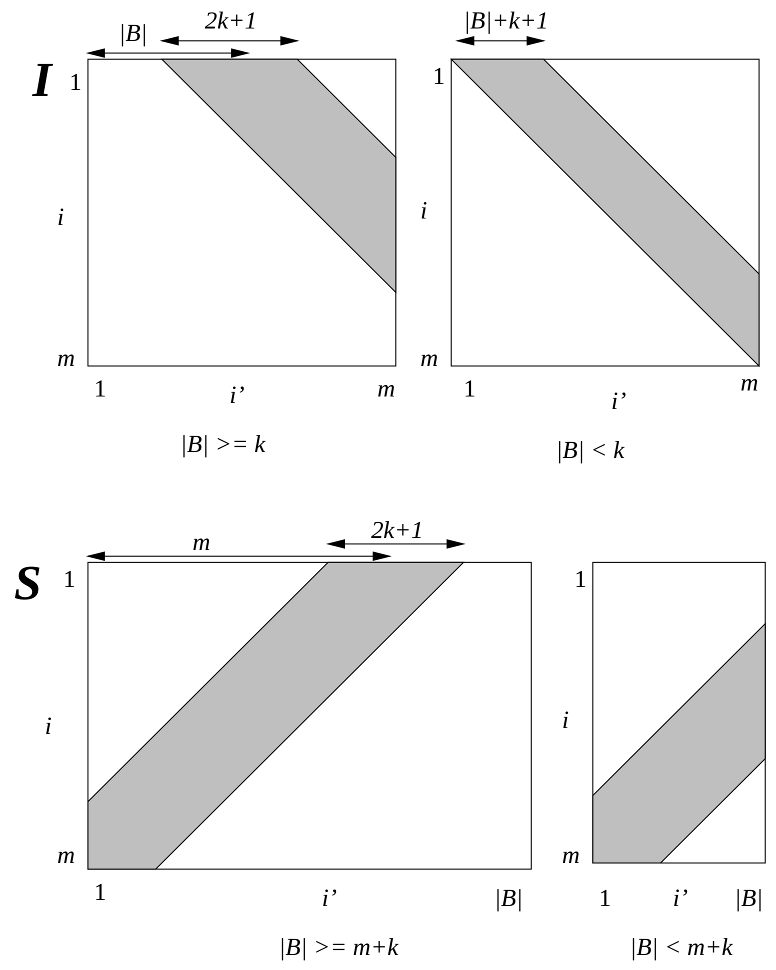

Figure 3: The $\mathcal{I}$ and $\mathcal{S}$ matrices that comprise the description of a block b representing a string B.

The way to compute the description of the blocks is format-dependent and is covered later. 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 into $P_{1\ldots i}$ 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 4 illustrates.

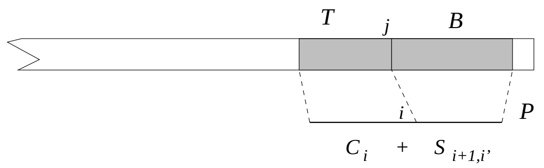

Figure 4: Splitting of an overlapping match (grayed).

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 $\mathcal{S}$. Finally, we require that $k_1+k_2 \le k$.

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

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

and we also have to report the positions $j+i'$ such that $\mathcal{C}_m + i' \le k$ (for $i' \in 1\ldots k$). This corresponds to $\mathcal{S}_{m+1,i'}(b) = i'$, which is not stored in that matrix.

The total cost for this check is $O(mk)$. The occurrences are not immediately reported but stored in increasing order 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 have already been stored in $\mathcal{M}(b)$ when the description of b was computed. 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. 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 5 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 $P_{1\ldots i'}$ against some suffix of $T_{1\ldots j}$ and $P_{i'+1\ldots i}$ against the whole B. This can be solved by combining $\mathcal{C}$ and $\mathcal{I}$.

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

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

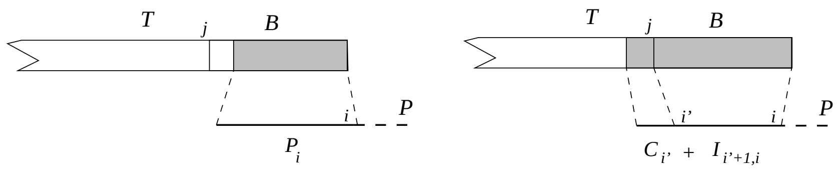

Figure 5: Two choices to update the $\mathcal{C}$ vector.

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.

Complexity. All the processes described up to now take $O(mkn)$ time for the existence problem and $O(mkn+R)$ time to report the R matches of P. We have to add the time to compute the block descriptions, a process that is detailed in the next sections.

The space requirement for this algorithm is basically that to store the block descriptions: the lengths, matrices and matches. The lengths can be stored using $n\log(u)$ bits[1]. For the matrices, we observe that each element of those arrays differs from the previous one by at most 1, that is $\mathcal{I}_{i,i'+1}(b) = \mathcal{I}_{i,i'}(b) \pm 1$, $\mathcal{P}_{i+1}(b) = \mathcal{P}_i(b) \pm 1$, and $\mathcal{S}_{i,i'+1}(b) = \mathcal{S}_{i,i'}(b) \pm 1$. 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 $(8mk+2m)n$ bits at most.

The internal matches, on the other hand, are at most R numbers that need $R\log u$ bits. We also need $n\log u$ bits to point directly into the array of internal matches. Therefore the total space requirement in bits is $2m(4k+1)n \;+\; 2n\log u \;+\; R\log u$.

6 Computing Block Descriptions for the LZ78 and LZW Formats

We show now how to do the rest of the updates in the LZ78 format, where each block b' 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 as follows.

- $|\varepsilon| = 0$.
- $\mathcal{I}_{i,i'}(b_0) = i' - i + 1$, $i \in 1 \ldots m$, $i' \in i-1 \ldots \min(i+k-1, m)$.
- $\mathcal{P}_i(b_0) = i$, $i \in 1 \ldots m$.
- $\mathcal{S}_{i,0}(b_0) = m - i + 1$, $i \in m-k+1 \ldots m$.

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 visualized as special cases of dynamic programming matrices between B and parts of P.

[1]We give all the space requirements in exact number of bits, disregarding lower order terms. The logarithms are base 2 unless otherwise indicated.

- $|B'| = |B| + 1$.

- $\mathcal{I}_{i,i'}(b') = \mathcal{I}_{i,i'-1}(b)$ 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[2] $\mathcal{I}_{i,\max(i-1,i+|B'|-k-2)}(b') = \min(|B'|, k+1)$, 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 row to the matrix, and we store at each block only the row of the matrix corresponding to its last letter (the rest can 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 6 illustrates. To relate this to the matrix of $\mathcal{I}$ in Figure 3 one needs to consider that there is a three dimensional matrix indexed by i, i' and $|B|$. Figure 3 shows the plane stored at each block B, corresponding to its last letter. Figure 6 shows a plane obtained by fixing i.

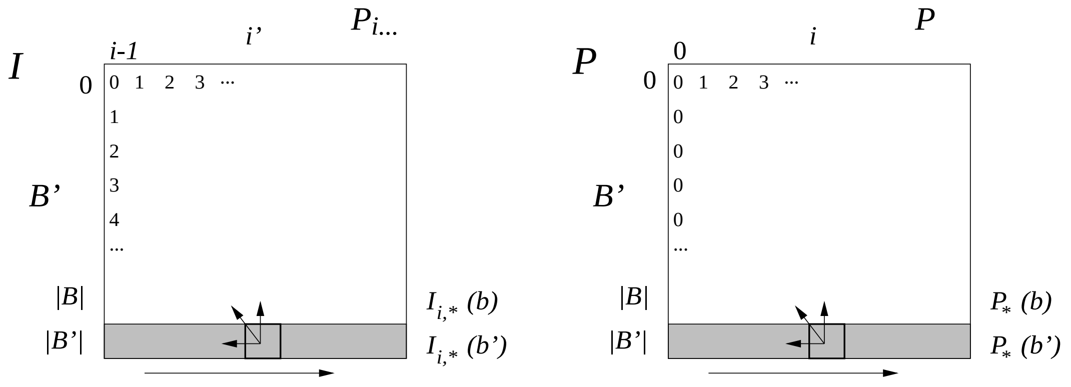

Figure 6: The virtual dynamic programming matrices. On the left, between B and $P_{i\ldots}$, to compute $\mathcal{I}$. On the right, between B and P, to compute $\mathcal{P}$.

- $\mathcal{P}_i(b') = \mathcal{P}_{i-1}(b)$ 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 6 illustrates.

- $\mathcal{S}_{i,i'}(b') = \mathcal{S}_{i,i'}(b)$ if $i' \le |B|$, and $\mathcal{I}_{i,m}(b')$ otherwise. This is a simpler formula because if we have a prefix of B matching a suffix of P the fact stays true after adding more characters at the end of B. Only the matches comprising the whole B' are new, and those are easily retrieved using $\mathcal{I}$. That is, $\mathcal{S}_{i,i'}(b) = \mathcal{I}_{i,m}(b^{(|B|-i'+1)})$, where $b^{(r)}$ denotes the block reached after following r times the backward chain of referenced blocks. Formally, $b^{(0)} = b$ and $b^{(r+1)}$ is the block referenced by $b^{(r)}$.

  This shows that we do not need in practice to store $\mathcal{S}$, since we can retrieve it by following the back chain of pointers. Moreover, $\mathcal{S}$ is used only to report the overlapping matches and

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

it is not hard to use the values in the same order given by the backward chain. Instead of $\mathcal{S}$ we need to store

$$ref(b) \quad = \quad b^{(\max(1,|B|-(m+k-1)))}$$

which allows us to recover the values $\mathcal{S}_{i,m+k-1},\ \mathcal{S}_{i,m+k-2},\ \ldots,\mathcal{S}_{i,1}$, in that order. This does not alter the time complexity.

- $\mathcal{M}(b') = \mathcal{M}(b)$, where the position $|B'|$ is added if $\mathcal{P}_m(b') \leq k$. That is, all the matches internal to B are also internal to B'. Then, a new internal match at the last position of the block may be added if an occurrence of P inside B' ends there.

  This is so simple in LZ78 that we can even not store $\mathcal{M}$ explicitly. Instead, each block can store the number of the last block in the referencing chain which holds a match in its last position, let us call it $Match(b)$. Hence, $Match(b') = Match(b)$ if $\mathcal{P}_m(b) > k$, and b otherwise. The original value of $\mathcal{M}(b)$ can be obtained by following the chain, in reverse order, in $O(|\mathcal{M}(b)|)$ time. Again, this does not alter the complexity. Moreover, it can be easily combined to the removal of $\mathcal{S}$ since both sets of matches that have to be merged (internal and external) will be obtained in reverse order.

Complexity. As can be seen, the updates of $\mathcal{P}$ cost $O(m)$ per block, but those of $\mathcal{I}$ and $\mathcal{S}$ take $O(mk)$. The updates to $\mathcal{M}$ add up $O(R)$ along the total process. In any case, the general complexity $O(mkn + R)$ is maintained.

This complexity is the same if we replace $\mathcal{S}$ and $\mathcal{M}$ by ref and $Match$. Each of the new vectors needs $n \log n$ bits. Therefore the number of bits required in this case becomes $2m(3k+1)n + 3n \log u$.

7 Computing Block Descriptions for More General Formats

In the format proposed in Section 4.2, each block is a concatenation of many previous blocks. In this case the search cost rises to $O(mk^2 n\alpha(n) + R)$, where $\alpha(n)$ is the inverse of Ackermann's $A(2n, n)$. We describe the case of concatenating two previous blocks in $O(mk^2)$ time, and later consider how to generalize for several blocks.

7.1 Concatenating Two Variable-Size Blocks

Assume now that b' is formed by concatenating b_1 and b_2, i.e. $B' = B_1B_2$. The formulas to compute the block description make heavy use of Property 1. The descriptions are computed as follows (see Figure 7).

- $\mathcal{I}_{i,i'}(b') = \min_{i'' \in i+|B_1|-k \ldots i+|B_1|+k}(\mathcal{I}_{i,i''}(b_1) + \mathcal{I}_{i''+1,i'}(b_2))$. This accounts for the fact that pattern substrings matching B_1B_2 are formed by a substring matching B_1 followed by a substring matching B_2. In this case filling each of the $O(mk)$ cells costs $O(k)$, for a total update cost of $O(mk^2)$ per block.

- $\mathcal{P}_i(b') = \min(\mathcal{P}_i(b_2), \min_{i' \in i-|B_2|-k \ldots i-|B_2|+k}(\mathcal{P}_{i'}(b_1) + \mathcal{I}_{i'+1,i}(b_2))$. This accounts for the fact that pattern prefixes matching a suffix of B_1B_2 are either those matching a suffix of B_2 or those matching a suffix of B_1 followed by an occurrence of B_2. This costs $O(mk)$.

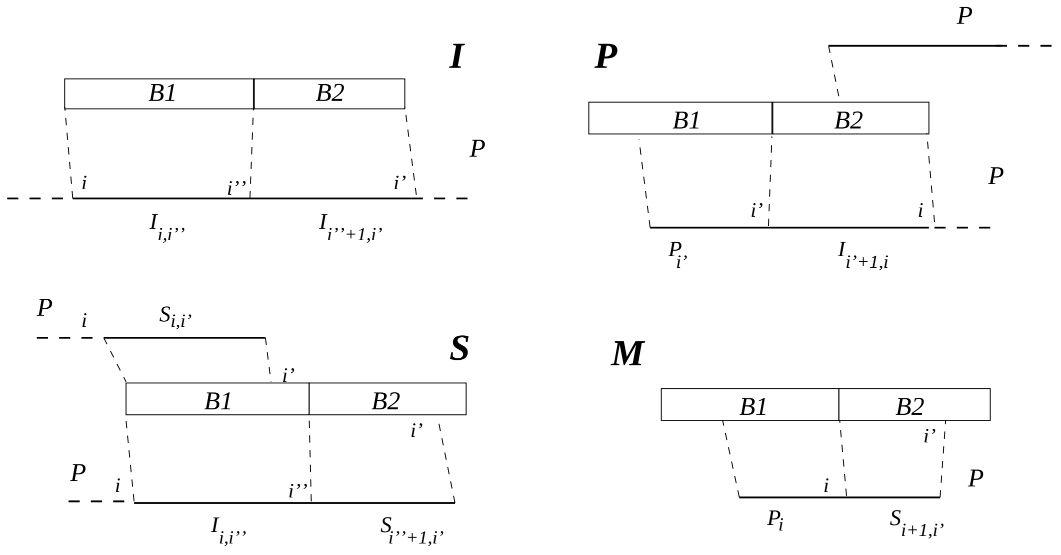

Figure 7: Computing block descriptions under the block format.

- $\mathcal{S}_{i,i'}(b') = \mathcal{S}_{i,i'}(b_1)$ if $i' \leq |B_1|$, else $\min_{i'' \in i+|B_1|-k \ldots i+|B_1|+k}(\mathcal{I}_{i,i''}(b_1) + \mathcal{S}_{i''+1,i'}(b_2))$. This is because suffixes of P matching a prefix of B_1B_2 match either a prefix of B_1 (if the prefix is shorter than B_1) or are formed by a pattern substring matching the whole B_1 followed by the rest of P matching a prefix of B_2. The cost for this is also $O(mk^2)$.

- For $\mathcal{M}(b')$ we copy $\mathcal{M}(b_1)$, then add the matches that appear when concatenating B_1 and B_2 and then copy $\mathcal{M}(b_2)$. The matches that appear in the concatenation ending at $i' \in 1 \ldots m+k-1$ in B_2 satisfy $\min_{i \in m-i'-k \ldots m-i'+k}(\mathcal{P}_i(b_1) + \mathcal{S}_{i+1,i'}(b_2)) \leq k$. For each such i', $|B_1| + i'$ is added as a match. Now, those matches must be merged with those internal to B_2. The complexity is $O(mk)$ per block plus $O(R)$ in total.

7.2 Concatenating Many Variable-Size Blocks

The method above can be trivially extended by composition to concatenate h blocks in $O(h)$ concatenation operations. This gives a total time complexity of $O(mk^2H)$ for compressed pattern matching in the LZ-Blocks format, where H is the total number of blocks in the concatenations. In the worst case, $H = \Omega(n^2)$. We next show how the time is lowered to $O(mk^2 n\alpha(n))$, where $\alpha(n)$ is the inverse of $A(2n, n)$ (the Ackermann function).

We will use an algorithm by Tarjan [Tar79] based on the path compression technique. The algorithm operates on a semigroup $(D, \odot)$ with associative operation $\odot$. In our case, D is the set of possible block descriptions and $\odot$ is the concatenation operation. The algorithm carries out a sequence of instructions that build and manipulate a forest with vertices labeled by elements of D. There are three kinds of instructions:

- LABEL(r, x): Label the root r with x.

- LINK(v, w): Combine the trees with roots v and w into a single tree by making v the parent of w.
- EVAL(v): Find the root of the tree currently containing v, say r, and return the product of all labels on the path from v to r.

A LABEL-instruction takes the time needed to make a copy of the label x and a LINK-instruction executes in constant time. For N EVAL-instructions in a forest of n nodes Tarjan gives an amortized time complexity of $O((N+n)\alpha(N+n,n)T_{\odot})$, where $T_{\odot}$ is the time taken by a $\odot$-operation and $\alpha(N,n)$ is the inverse Ackermann's function. The algorithm requires that the sequence of instructions is given off-line except for the labels in the LABEL-instructions.

The block descriptions for the LZ-Blocks format can be produced with a suitable sequence of instructions. The sequence of instructions builds a linear tree by adding one vertex at a time as the root of the tree. Let $v_1, v_2, \ldots$ be the sequence of vertices added to the tree. After v_i is added to the tree, it is labeled by the description of the basic block b_i and then all block concatenations (requested by later blocks) that end at b_i are computed and saved for later. The description of a block concatenation $b_j \cdots b_i$ is computed by executing EVAL(v_j). Thus the sequence of instructions consists of n instructions of each kind. The sequence can be computed with linear time scan of the compressed text satisfying the off-line requirement. This gives the total time complexity of $O(mk^2 n\alpha(n))$, where $\alpha(n)$ is short for $\alpha(2n, n)$.

7.3 Collage Systems

Under the model of a collage system, a compressed pattern matching algorithm has conceptually two parts: first precompute any required information on the dictionary D, and second process the symbols of S.

The algorithm we have presented in this section can be directly applied to any regular collage system (i.e. the dictionary is formed by basic symbols and concatenations of previously formed symbols). The description of every block in the dictionary D is computed first, and then we go over the elements of S updating the search state and reporting the matches.

The worst case time complexity of this scheme is $O(mk^2|D| + mk|S| + R)$.

8 A Faster Algorithm on Average for LZ78/LZW

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 are interested in showing that 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 Ziv-Lempel blocks in the text. Moreover, we would like to consider more general random models, since uniform distribution of characters does not marry well with text compression.

First assume that the text is arbitrary but fixed and that the pattern P is generated by a Markov process such that no string has probability 1 (i.e. the Markov process is not degenerated). The number of active cells at the end of the blocks depends on the probability of common characters between P and the last $m+k$ text characters at the end of each block. Even if we consider that all such text substrings match with the most probable patterns, still the combinatorial argument used in [BYN99] to prove the $O(k)$ bound stays valid. Just the constant $1/\sigma$, which represents probability of matching between two random characters, has to be replaced by the highest probability of a pattern substring in the Markov process. This only changes the constant of the $O(k)$ bound.

The same happens if we consider a text generated by a Markovian source and a fixed arbitrary pattern: despite that the distribution of the text at the end of the blocks needs not be the same as for random text positions, still each substring has a probability of occurring smaller than 1, and therefore even if the pattern is formed by the most common text substrings the argument leading to the $O(k)$ result stays valid. Finally, this also holds obviously if both pattern and text are generated by a Markovian source.

Our conjecture is in fact that the suffixes of Ziv-Lempel blocks have a distribution which is more uniform than that of the text, but we cannot prove it and it is not necessary for our results.

If the last active cell is at position $O(k)$ in $\mathcal{C}$, then the same happens to the $\mathcal{P}$ values, since $\mathcal{P}_i(b) \geq \mathcal{C}_i$ after processing block b.

We update $\mathcal{C}$ and $\mathcal{P}$ up to their last active cells only. 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 $\mathcal{I}_{i,i'}(b)$. The key idea in this case comes from considering that $ed(P_{i...i'}, B) > k$ 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 [Ukk85] that if $|P| = m$, then $|U_k(P)| = O((m\sigma)^k)$. This limits the total number of different blocks B that can be preprocessed for a given pattern substring $P_{i...i'}$. Summing over all the possible substrings $P_{i...i'}$, considering that computing each such entry for each block takes $O(1)$ time, we have a total cost of $O(mk(m\sigma)^k)$. 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 $O(mk \min(n, (m\sigma)^k))$.

Finally, we have to consider the cost of reporting the matches. This is $O(R)$ plus the cost to search for the overlapping matches. We use 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 $O(k^2 n + mk \min(n, (m\sigma)^k) + R)$ average time. Note in particular that the middle term is asymptotically independent on n, leading to $O(k^2n + R)$ for large enough n. Moreover, the space required is reduced as well, because only the relevant parts of the matrices need to be stored.

9 Searching Under General Cost Models

We consider now how to modify our previous algorithms in order to adapt them to varying search costs. The general mechanism is very similar and the information we store does not change. The only difference at this respect is that it is not true anymore that adjacent cells differ by at most one, so we cannot store the $\mathcal{I}$, $\mathcal{P}$ and $\mathcal{S}$ matrices anymore using 2 bits per cell. Instead, we need $\log(k + 2)$ bits.

The main operational change corresponds to the places where k plays a role in the algorithm. As explained in Section 3.3, k_{id} is the width of the range of differences in length between two strings that match with error threshold k. The two dimensional matrices $\mathcal{I}$ and $\mathcal{S}$ will store only $O(mk_{id})$ entries. Moreover, all the places where $O(mk)$ or $O(mk^2)$ complexities have appeared to deal with overlapping matches, compute block descriptions or update the search state, the k (actually $1 + 2k$) comes in fact from the maximum difference in length between two matching strings, so it has to be replaced by k_{id}. Another operational change appears in the LZ78/LZW format, where we explicitly fill cells of dynamic programming matrices when we compute new block descriptions. The generalized formulas of edit distance have to be used in this case.

This means that in the worst case we can search in $O(mk_{id}n+R)$ on LZ78/LZW, $O(mk_{id}^2 n\alpha(n)+R)$ on LZ-Blocks and $O(mk_{id}^2|D| + mk_{id}|S| + R)$ on regular collage systems. For example, for Hamming distance this reduces to $O(mn + R)$ in LZ87/LZW, $O(mn\alpha(n) + R)$ on LZ-Blocks and $O(m(|D|+|S|+R)$ on regular collage systems. This side result becomes the best existing complexity for this problem (the only one was from [NR98], $O(k^2n\lceil m\log(k)/w\rceil)$ on LZ78/LZW).

When it comes to the average case improvement for LZ78/LZW, the result of [Ukk85] that $|U_k(P)| = O((m\sigma)^k)$ remains valid if we replace k by k_{ids}. The rationale is that in the worst case we can perform k_{ids} operations on P before surpassing the error threshold k, and each such operation can alter one character in P (hence the m choices) and replace/insert any other character (hence the σ choices). On the other hand, it is not hard to show that the number of active cells per column is on average $O(k_{ids})$. Therefore the average case time can be made $O(k_{ids}k_{id}n + mk_{id}\min(n,(\sigma m)^{k_{ids}}) + R)$. For Hamming, this is $O(kn + m\min(n,(\sigma m)^k) + R)$.

Despite the generality of this cost function, we have left aside some operations that violate Property 1, such as transposition of adjacent characters. This has been mainly for technical convenience, as in fact many extensions of this kind can be included in our approach by laboriously considering and patching the exceptions they trigger. For example, transpositions could be arranged for by explicitly trying a transposition in the limit of two blocks each time we apply Property 1. General substring replacement (i.e. arbitrary substrings can be replaced at arbitrary costs) is also possible at the expense of checking the limits between blocks for the presence of those strings, which certainly complicates the approach.

10 Significance of the Results

We consider now the theoretical and practical significance of the results obtained. We concentrate on the results for LZ78/LZW, where more competing options exist.

10.1 Memory Requirements

First consider the space requirements. In the worst case, when we remove the $\mathcal{S}$ and $\mathcal{M}$ tables, we need $2m(3k+1)n + 3n\log n$ 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.

Our search mechanism can use the same mark to start reusing its allocated memory from scratch as well, since no node seen in the past will be referenced again (only the state of the search $(\mathcal{C}, j)$ has to be remembered). This technique can be adapted to more complex ways of reusing the memory under various LZ compression schemes [BCW90].

If a compressor is limited to use N nodes, then the decompression needs at the very least $O(N\log(N\sigma))$ bits of memory[3]. Since the search algorithm can be restarted after reading N blocks, it requires only $2m(3k+1)N + 3N\log N$ bits. Hence the amount of memory required to search is never more than

$$\left(3 + \frac{2m(3k+1)}{\log N}\right) \quad \times \quad \text{memory for decompression}$$

and we recall that this can be lowered in the average case.

10.2 Time Complexity

Despite that ours is the first algorithm for approximate searching allowing errors, there exist also alternative approaches, some of them trivial and others not specifically designed for approximate searching. Moreover, an alternative algorithm truly designed for approximate searching appeared in the while [MKT$^+$00].

The first alternative approach is DS, a trivial decompress-then-search algorithm. This yields, for the worst case, $O(ku)$ [GP90] or $O(m|U_k(P)| + u)$ [Ukk85] time, where we recall that $|U_k(P)|$ is $O((m\sigma)^k)$. For the average case, the best result in theory is $O(u + (k + \log_\sigma m)u/m) = O(u)$ [CM94]. 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 $O(m|U_k(P)| + mn + R)$, and $O((k + \log_\sigma m)n + mn + R) = O(mn + R)$

[3]In fact, reasonably fast decompression needs to store the text already decompressed, which requires $U\log\sigma + N\log U$ bits, where U is the text size that was compressed to N symbols.

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	$m^2\|U_k(P)\|^2 + n + R$	$m^2\|U_k(P)\|^2 + n + R$
BP	$mk^3/w\ n$	$mk^3/w\ n$
Ours	$mkn + R$	$k^2n + mk\min(n, \|U_k(P)\|) + R$

Table 2: Worst and average case time for different approaches on LZ78/LZW.

on average. Except for large u/n, 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 [KTS+98], a set of strings of *total* length M can be searched for 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 number of patterns to search for.

Table 2 compares the complexities for LZ78/LZW. As can be seen, our algorithm yields the best average case complexity for

$$k \;=\; O(\sqrt{u/n}) \quad\wedge\quad k \;=\; O(\sqrt{m}) \quad\wedge$$

$$\frac{\log_\sigma n}{2(1+\log_\sigma m)} \;\leq\; k + \frac{1}{2} + O\left(\frac{1}{\log_\sigma m}\right) \;\leq\; \frac{\log_\sigma n}{1+\log_\sigma m}$$

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 and OM). This means in practice that our approach is the fastest for short and medium patterns and low error levels.

We consider now the alternative approach BP [MKT+00], which can solve only the existence problem in $O(mk^3n/w)$ worst case time, where w is the number of bits in the computer word. We are better in the worst case for $k = \Omega(\sqrt{w})$. On average, we are also better when $k + O(1) \leq \log_\sigma(n/w)/(1+\log_\sigma m)$.

10.3 Experimental Results

We have implemented our algorithm on LZ78 in order to determine its practical value. Our implementation is based on the version that does not store $\mathcal{S}$ and $\mathcal{M}$. It 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 run our experiments on a Sun UltraSparc-1 of 167 MHz and 64 Mb of RAM. We have compressed 10 Mb of Wall Street Journal articles (WSJ) and 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 mark the leaves of the trie raised this percentage to 45.46%, i.e. less than 1% of increment. The figures for DNA were 39.69% with plain compression and 40.02% adding the extra bit. As a comparison, Unix *Compress* program, an LZW compressor that uses bit coding, obtained 38.75% for WSJ and 27.91% for DNA. Compression took about 20 seconds of user time, while decompression took 2.09 seconds for WSJ and 1.80 for DNA. In our complexity analysis n is measured in blocks and u in bytes. In this case we have $u/n = 8.6$ for WSJ and 9.9 for DNA.

We have compared our algorithm against a more practical version of DS, which decompresses the text on the fly and searches it, instead of writing it to a new decompressed file and then reading it again to search. The search algorithm is the $O(ku)$ average time algorithm described in [Ukk85], that is, a modified dynamic programming that works on the active cells only. We use this algorithm because it is the one that we adapted to obtain our new algorithm (this gives a measure of the improvement obtained) and because it is the only one able to cope with the general version of the problem with arbitrary edition costs. It would also be possible to use the same DS approach with a faster algorithm, but this would work only on specific instances of the problem. The $O(ku)$ algorithm is unbeaten in flexibility to cope with all the variants of the approximate search problem, and our algorithms share this flexibility.

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 [KTS$^+$98] on our texts and it took 5.50 seconds for just one pattern of $m = 10$. This outrules it in our cases of interest.

We have tested $m = 10$, 20 and 30, and $k = 1$, 2 and 3. These are the most interesting values in text searching applications. For each pattern length, we selected 100 random patterns from the text and used the same patterns for both algorithms. Table 3 shows the results.

As the table shows, we can actually improve upon the decompression of the text and the application of the same search algorithm. In practical terms, we can search the original file at about 2.5 . . . 3.2 Mb/sec when $k = 1$, while the time stays reasonable and competitive for $k = 2$ as

WSJ						
k	Ours $m = 10$	DS $m = 10$	Ours $m = 20$	DS $m = 20$	Ours $m = 30$	DS $m = 30$
1	3.77	4.72	3.28	4.64	3.13	4.62
2	5.63	5.62	4.77	5.46	6.10	5.42
3	11.60	6.43	9.22	6.29	13.61	6.22
DNA						
k	Ours $m = 10$	DS $m = 10$	Ours $m = 20$	DS $m = 20$	Ours $m = 30$	DS $m = 30$
1	3.91	5.21	2.49	5.08	2.57	5.06
2	6.98	6.49	3.81	6.31	5.02	6.28
3	11.51	8.91	9.28	7.51	15.35	7.50

Table 3: CPU times to search compressed files WSJ and DNA of $u = 10$ Mb.

k	$\mathcal{I}$	$\mathcal{P}$	Others	Total
1	0.05 Mb	1.23 Mb	5.59 Mb	6.87 Mb
2	0.78 Mb	1.91 Mb	5.59 Mb	8.28 Mb
3	4.50 Mb	2.60 Mb	5.59 Mb	12.69 Mb

Table 4: Space requirement to search for a pattern with $m = 10$ over WSJ, of $u = 10$ Mb.

well.

Our implementation is not memory-efficient. However, it may be interesting to measure the number of bytes required by the $\mathcal{I}$ and $\mathcal{P}$ vectors as k grows. Table 4 shows the results for $m = 10$ and $k = 1$ to 3. It can be seen how $\mathcal{I}$ grows sharply as k increases, while $\mathcal{P}$ grows linearly. In our implementation there is also an overhead of 10 bytes per block, which adds about 5.6 Mb to the space. As a comparison, a fast DS approach needs at least 12.23 Mb.

11 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 $O(k^2n + R)$ average case time. We have shown that this is of theoretical and practical interest for small k and moderate m values. We can also deal with more general LZ formats at $O(mk^2n\alpha(n) + R)$ worst case time and with regular collage systems in $O(mk^2|D| + mk|S| + R)$ time. We have also shown that more complex distances can be dealt with, as well as simpler cases such as Hamming distance, where $O(mn + R)$ worst case time and $O(kn + R)$ average time can be obtained. Our experiments show that in LZ78/LZW we can search at twice the speed of decompressing and searching with the basic technique.

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 [GP90] 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, for the Levenshtein distance we can store 2 bits per cell and adapt [Mye99] to update $\mathcal{P}$ and $\mathcal{I}$ using bit-parallelism. 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, but perhaps some clues can be found in [MKT$^+$00] (which however reports very bad experimental results). 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 more work and higher space requirement.

A third question is whether 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 [WM92] could be applied by using the multipattern search algorithm of [KTS$^+$98]. In theory the complexity is $O(m^2+n+ukm^2/\sigma^{\lfloor m/(k+1)\rfloor})$, which is competitive for $k < m/\Theta(\log_\sigma(u/n)+\log_\sigma m)$. Some progress in this respect has already been made [NKT$^+$01].

Finally, it would be interesting to consider other more general compression models, such as LZ77. LZ77 is more popular than LZ78 because of its generally higher compression ratio and faster decompression, but searching on it seems to be extremely difficult.

References

[AB92] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.

[ABF96] 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*.

[AG97] A. Apostolico and Z. Galil. *Pattern Matching Algorithms.* Oxford University Press, Oxford, UK, 1997.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.

[BYN99] R. Baeza-Yates and G. Navarro. Faster approximate string matching. *Algorithmica*, 23(2):127–158, 1999.

[CL92] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, LNCS 644, pages 172–181, 1992.

[CM94] W. Chang and T. Marr. Approximate string matching and local similarity. In *Proc. CPM'94*, LNCS 807, pages 259–273, 1994.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, Oxford, UK, 1994.

[FT98] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998. Previous version in *STOC'95*.

[GP90] Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM J. on Computing*, 19(6):989–999, 1990.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.

[KNU00] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, pages 195–209, 2000.

[KST$^+$99] T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th Intl. Symp. on String Processing and Information Retrieval (SPIRE'99)*, pages 89–96. IEEE CS Press, 1999.

[KTS$^+$98] 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.

[KTS$^+$99] 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.

[KU96] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Conference on Computing and Combinatorics (COCOON'96)*, LNCS 1090, 1996.

[Man97] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM Trans. on Information Systems*, 15(2):124–136, 1997.

[MKT$^+$00] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In *Proc. SPIRE'2000*, 2000. To appear.

[MNZBY00] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems (TOIS)*, 18(2):113–139, 2000.

[MW85] V. Miller and M. Wegman. Variations on a theme by Ziv and Lempel. In *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 131–140. Springer-Verlag, 1985.

[Mye99] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic progamming. *Journal of the ACM*, 46(3):395–415, 1999.

[Nav01] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[NKT$^+$01] Gonzalo Navarro, Takuya Kida, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. Faster approximate string matching over compressed text. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 459–468, 2001.

[NR98] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. Technical Report TR/DCC-98-12, Dept. of Computer Science, Univ. of Chile, 1998.

[NR99] 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.

[NT00] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, pages 166–180, 2000.

[NW70] 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.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[SMT$^+$00] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. CPM'2000*, LNCS 1848, pages 181–194, 2000.

[Tar79] R. E. Tarjan. Applications of path compression on balanced trees. *Journal of the ACM*, 26(4):690–715, 1979.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[Wel84] T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.