Practical and Flexible Pattern Matching over Ziv-Lempel Compressed Text [*]

Gonzalo Navarro[†] Mathieu Raffinot[‡]

Abstract

We address the problem of string matching on Ziv-Lempel compressed text. The goal is to search a pattern in a text without uncompressing it. This is a highly relevant issue to keep compressed text databases where efficient searching is still possible. We develop a general technique for string matching when the text comes as a sequence of blocks. This abstracts the essential features of Ziv-Lempel compression. We then apply the scheme to each particular type of compression. We present an algorithm to find all the matches of a pattern in a text compressed using LZ77. When we apply our scheme to LZ78, we obtain a much more efficient search algorithm, which is faster than uncompressing the text and then searching on it. Finally, we propose a new hybrid compression scheme which is between LZ77 and LZ78, being in practice as good to compress as LZ77 and as fast to search in as LZ78. We show also how to search some extended patterns on Ziv-Lempel compressed text, such as classes of characters and approximate string matching.

1 Introduction

String matching is one of the most pervasive problems in computer science, with applications in virtually every area. It is also one of the oldest and richest area of development. The *string matching problem* is: given a pattern $P = p_1 ... p_m$ and a text $T = t_1 ... t_u$, both sequences of symbols over a finite alphabet Σ of size σ, find all the occurrences of P in T. There are many algorithms to solve this problem, from classical to very recent [20, 9, 4, 14, 33, 10, 29]. The complexity of this problem is $O(u)$ in the worst case and $O(u \log(m)/m)$ on average, where $u = |T|$ and $m = |P|$, and there exist variants of [9, 10] which achieve this complexity. In practice, however, [33, 29] are the fastest algorithms in most cases.

Another old and rich area in computer science is text compression. Its aim is to exploit the redundancies of the text to reduce its space usage. There are many different compression schemes [6], among which the Ziv-Lempel family [38, 39] is one of the most commonly used in practice because of their good compression ratios (that is, the size of the compressed file as a percentage of that of the uncompressed file) combined with efficient compression and decompression times. Other compression schemes are Huffman coding [15] and arithmetic coding [35], among others.

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.

The *compressed matching problem* was first defined by 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 $Z = z_1 \ldots z_n$, 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 $O(u + m)$. An optimal algorithm takes worst-case time $O(n + m)$, where $n = |Z|$. In [2], a new criterion, called *extra space*, for evaluating compressed matching algorithms,

[*]Partially supported by ECOS/Conicyt Project C99E04.

[†]Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Partially supported by Fondecyt grant 1-020831.

[‡]Equipe Génome et Informatique, Tour Evry 2, 523, place des terrasses de l'Agora, 91034 Evry, France. raffinot@genopole.cnrs.fr.

was introduced. According to the extra space criterion, algorithms should use at most $O(n)$ extra space, optimally $O(m)$ in addition to the n-length compressed file.

We define now a variation where we are required to report all the matching positions. That is, given P and Z, report all the $|x|$ such that $T = xPy$. The optimal algorithm for this problem takes $O(m + n + R)$ time, where R is the number of matches.

Two different approaches have emerged in the last years to combine compression and searching in textual databases. A first one is strongly oriented to natural language texts, which are assumed to be composed of words which follow some statistical rules. The basic idea is to compress the text using Huffman, where the words instead of the characters are taken as the symbols [8, 25]. As Huffman assigns a fixed code to each symbol, searching a given string is a matter of compressing it and searching it in the compressed text using a classical string matching algorithm with minor modifications [27, 26]. Despite its simplicity, this approach is very effective on natural language text, with better compression ratios than those of the Ziv-Lempel family, and search time which is between 2 and 8 times faster than the fastest algorithms for standard string matching over the uncompressed text. They are also able to search for complex patterns (such as regular expressions) and allow errors in the matches, provided that words are matched against words. The average search time for a simple pattern is close to $O(m + n\log(u/n)/(u/n))$. The extra space is $O(\sqrt{u})$, which is the same space necessary to decompress the text. A weakness of this scheme is that it does not work well on small texts (say, less than 10 Mb), since in that case the vocabulary is almost as big as the text itself. Also, it can be applied only to natural language texts.

Another practical approach is an ad-hoc technique [21], which however obtains compression ratios of near 70% (against 30% to 40% of Ziv-Lempel algorithms) and relies on the ASCII encoding. A more elegant generalization [32] is based on byte-pair encoding and achieves similar search times and compression ratios close to those of classical Huffman.

The second line of research considers Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings previously appeared. LZ77 is able to reference any substring of the text already processed, while LZ78 references only a single previous reference plus a new letter that is added. In both cases, the referenced text to be found is normally limited by a *window* which precedes the current text position.

String matching in Ziv-Lempel compressed texts is much more complex, since the pattern can appear in different forms across the compressed text. In [2] a compressed matching algorithm for LZ78 is presented, which works in time and space $O(m^2 + n)$. For LZ77, the only result is [12], which is a randomized algorithm to determine in time $O(m + n\log^2(u/n))$ whether a pattern is present or not in an LZ77-compressed text, but they do not find all the pattern occurrences. More practical approaches to this problem have appeared in [30, 19] based on bit-parallelism and in [31] based on Boyer-Moore.

On the other hand, little has been done for searching flexible patterns on compressed text. Very recently, two solutions for approximate pattern matching have been proposed [16, 23], although their main value is theoretical. Simpler capabilities, such as permitting classes of characters and allowing replacements, have not yet received much attention.

In this paper, an extended version of [30], we aim at efficient algorithms for flexible string matching on Ziv-Lempel compressed texts. We present new theoretical developments but also give practical implementations and experiments on our algorithms.

Our approach is practical and relies on *bit-parallelism*. Bit-parallelism [3, 36] is a general technique to take advantage of the fact that the computer operates in parallel over all the bits of the machine word, so that if a process is so simple that it can be expressed with bit operations we can perform many of those steps in a single operation of the processor. If we call w the length in bits of the machine word (typically 32 or 64), then the possible speedups are up to $O(w)$.

Our main results are:

- We develop a general technique for string matching on a text which is given as a sequence of blocks. This abstracts the essential features of Ziv-Lempel compressed texts and is the basis for the algorithms which run over specific members of the family.

- We apply the technique to the LZ78 compression scheme. The result is an algorithm which turns out to be a practical implementation of the theoretical proposal of [2]. This algorithm is $O(n\lceil m/w\rceil + R)$ time in the worst and average case ($O(n+R)$ on short patterns), and is in practice twice as fast as decompressing and searching.

- We apply our technique to LZ77-compressed texts. The result is an algorithm to search under this compression scheme (recall that [12] cannot find all the occurrences of the pattern). The algorithm, however, is $O(u)$ time at best. In practice, the algorithm is slower than uncompressing the text and searching it with a classical algorithm.

- We propose LZ-Blocks, a hybrid compression scheme which is between LZ77 and LZ78, which keeps some of the good features of LZ77 and which can be searched in $O(\min(u, n\log m) + n\lceil m/w\rceil + R)$ time on average (and $O(\min(u, mn) + n\lceil m/w\rceil + R)$ in the worst case). In practice, the compression efficiency is similar to LZ77 and the search time is similar to LZ78.

- We show how to search some extended patterns in a sequence of blocks, such as how to allow classes of characters or approximate string matching, the last one being an open problem advocated in [2].

In all cases our preprocessing cost is $O((\sigma+m)\lceil m/w\rceil)$ and our extra space is $O(n\lceil m/w\rceil + R)$, almost the same necessary to decompress the text.

2 String Matching on Blocks

We describe now a general technique for string matching when the text is presented as a sequence of atomic strings (here called "blocks") instead of a sequence of characters. This technique is the basis for all the different searching algorithms on Ziv-Lempel compressed text, which are described in the next sections. To simplify the notation, we number pattern positions starting at zero.

Our general assumption is that the blocks either have just one letter (that we can access directly) or are formed by a concatenation of previously seen blocks. We describe an online algorithm where we process the text block by block. At any moment of the search we denote T' the text already processed (of $|T'|$ characters). When we finish the search, $T' = T$, i.e. the original text.

The method works as follows. We process the blocks one by one. For each new block B, we compute a *description* for B which has all the information of the block which is relevant for the search. This description is denoted $D(B) = (L, O, S, P, M)$, where

- $L = |B|$, that is, the length of B in characters;

- $O = \text{Offs}(B) =$ the length in characters of the text we had processed when B appeared;

- $S = \text{Suff}(B) =$ all the pattern positions which either start a complete occurrence of B inside the pattern, or start a proper pattern suffix which matches with a prefix of B. Formally,

$$\text{Suff}(B) \;=\; \{|x|,\ \exists y,\ P = xBy\} \;\cup\; \{|x|,\ \exists y,z,\ |x|>0 \wedge |z|>0 \wedge P = xz \wedge B = zy\}\ ;$$

- $P = \text{Pref}(B) =$ all the pattern positions which either follow a complete occurrence of B inside the pattern, or follow a proper pattern prefix which matches with a suffix of B. Formally,

$$\begin{array}{rcl}\text{Pref}(B) & = & \{|xB|,\ \exists y,\ P = xBy\ \wedge\ |y|>0\}\ \cup \\ & & \{|z|,\ \exists y,z,\ |z|>0\ \wedge\ |y|>0\ \wedge\ P = zy\ \wedge\ B = xz\ \}\ ;\end{array}$$

- M = Matches(B) = all the block positions where the pattern occurs ($\emptyset$ if $|B| < |P|$). Formally,

$$\text{Matches}(B) \quad = \quad \{|x|,\ \exists y,\ B = xPy\}\ .$$

Figure 1 illustrates these concepts.

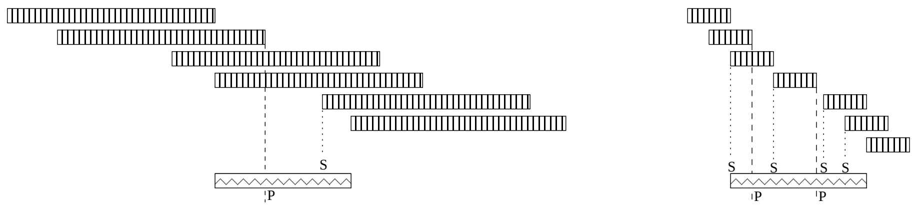

Figure 1: Prefixes (P) and suffixes (S) for a long and a short block. The pattern has the diagonal tiling and the possible blocks have a bar tiling. The suffixes (dotted lines) and prefixes (dashed lines) are pattern positions. Prefixes are marked after the position where they finish, suffixes are marked at the position they start.

The description $D(B)$ of a new block B is obtained in two forms: (a) the block is an explicit letter and then we obtain the description directly, or (b) the block is a concatenation of other blocks previously known, and we obtain its description by operating on the descriptions of the previous blocks.

Once the description of the new block is computed, we use that description to update the state of the search. This concludes the processing of the block and we move to the next one. The state of the search contains the matches that have already occurred and the potential matches in progress, that is,

- Res(T') = the text positions that matched up to now, formally

$$\text{Res}(T') \quad = \quad \{|x|,\ \exists y,\ T' = xPy\}\ ;$$

- Active(T') = the set of positions following the pattern prefixes which match a suffix of the current text. Formally,

$$\text{Active}(T') \quad = \quad \{|x|,\ \exists y, z\ |x| > 0\ \wedge\ |y| > 0\ \wedge\ P = xy\ \wedge\ T' = zx\}\ .$$

Hence, when we complete the text processing and T' is not a text prefix anymore but the whole text, Res(T) is our answer. The initial state of the search is $T' = \epsilon$, and Res(ϵ) = Active(ϵ) = $\emptyset$.

We have defined already the information we keep, and consider now how to compute that information. For the formulas that follow, we define some auxiliary functions, namely

- $\text{Left}_i(X) \ = \ \{x - i,\ x \in X\}\ \cup\ \{m - i, m - i + 1, \ldots, m - 1\}$, which receives a set of Suff() positions not smaller than i, subtracts i to all them and then adds new pattern positions filling the hole left by the shift.
- $\text{Right}_i(X) \ = \ \{x + i,\ x \in X\}\ \cup\ \{1, 2, \ldots, i\}$, which does the same for Pref() positions, in the other direction.
- $\text{Add}_i(X) \ = \ \{i + x,\ x \in X\}$, which adds i to all the elements of the set.
- $\text{Subtr}_i(X) \ = \ \{i - x,\ x \in X\}$, which subtracts all the elements of the set from i.

2.1 Description of a Letter

The base case of our scheme is to obtain the description of a block which is a letter a. The following is obtained by direct application of the general formulas.

- $|B| \;=\; 1$
- $\text{Offs}(B) \;=\; |T'|$
- $\text{Suff}(B) \;=\; \{|x|,\ \exists y,\ P = xay\}$
- $\text{Pref}(B) \;=\; \{|xa|,\ \exists y,\ P = xay\ \wedge\ |y| > 0\}$
- $\text{Matches}(B) \;=\; \text{if } P = a \text{ then } \{0\} \text{ else } \emptyset$

2.2 Concatenating Two Blocks

Assume that our block B is defined as the concatenation of one or more previous blocks. If only one previous block B' is referenced, we just copy its definition. We show now how to concatenate two blocks, since the case of more than two blocks is a simple iteration over this procedure. We are given two blocks B_1 and B_2, and we have to obtain the description for their concatenation $D(B) = D(B_1B_2) = D(B_1) \cdot D(B_2)$ (where we define $\cdot$ as the concatenation of block descriptions). The formulas are as follows

- $|B| \;=\; |B_1| + |B_2|$
- $\text{Offs}(B) \;=\; |T'|$
- $\text{Suff}(B) \;=\; \text{Suff}(B_1)\ \cap\ \text{Left}_{|B_1|}(\text{Suff}(B_2))$
- $\text{Pref}(B) \;=\; \text{Pref}(B_2)\ \cap\ \text{Right}_{|B_2|}(\text{Pref}(B_1))$
- $\text{Matches}(B) \;=\; \text{Matches}(B_1)\ \cup\ \text{Add}_{|B_1|}(\text{Matches}(B_2))$
  $\cup\ (\text{Subtr}_{|B_1|}(\text{Pref}(B_1)\ \cap\ \text{Suff}(B_2))\ \cap\ \{0, 1, 2, \ldots, |B| - m\})$

We explain now the rationale for the formulas (see Figure 2). The first two are immediate. For Suff(B), note that Suff(B_1B_2) considers that either a prefix of B_1 may be a suffix of P or B_1 may be completely inside P followed by a prefix of B_2 matching the a suffix of P. That is, if the number i belongs to Suff(B_1B_2) then either

- $i \geq m - |B_1|$, that is, a prefix of B_1B_2 is a suffix of P. Notice that in this case also a prefix of B_1 is a suffix of P. Since $\text{Left}_{|B_1|}$ will add all these positions, they will appear in the result if and only if they are present in Suff(B_1), which is correct.
- $i < m - |B_1|$, that is, B_1 appears inside P and is immediately followed by an occurrence of B_2 (which can be a complete occurrence or share a prefix with the pattern suffix). If we subtract $|B_1|$ to the elements in Suff(B_2), then we are interested in the positions which also appear in Suff(B_1) (which since $i < m - |B_1|$ can only correspond to complete occurrences of B_1).

The rationale for Pref() is analogous to Suff(). For Matches(B), there are three parts. The first one is the matches which are inside B_1, and the second one is the same for B_2 (displaced since now B_2 comes after B_1 in B). The third one accounts for matches that appear only when B_1 and B_2 are concatenated. If a prefix of the pattern is at the end of B_1, and the corresponding suffix is at the beginning of B_2, then we have the pattern in B_1B_2. The *Subtr* converts pattern to block positions and the final set which is intersected with the results ensures that we have really prefixes and suffixes instead of substrings of the blocks.

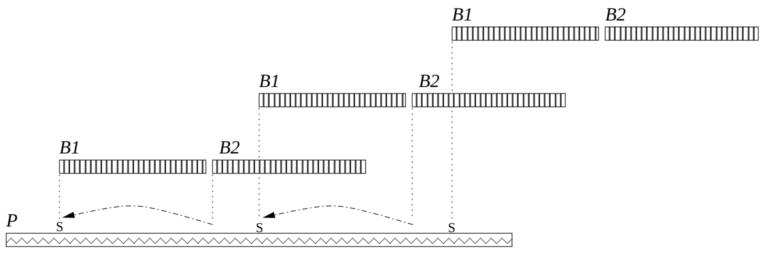

Figure 2: Suffixes of the concatenation of two blocks. It is possible that the result involves only B_1 (rightmost pair) or that it involves both. In this case B_1 is completely inside the pattern and B_2 may or may not be totally inside (leftmost and middle pairs, respectively).

2.3 Updating the Search State

We want now to update the state of our search by processing a new block B whose description has just been computed. The formulas to obtain the new $\text{Res}(T'B)$ and $\text{Active}(T'B)$ values from the old $\text{Res}(T')$ and $\text{Active}(T')$ ones are

- $\text{Active}(T'B) \;=\; \text{Right}_{|B|}(\text{Active}(T')) \;\cap\; \text{Pref}(B)$
- $\text{Res}(T'B) \;=\; \text{Res}(T') \;\cup\; \text{Add}_{|T'|}(\text{Matches}(B)) \;\cup$
  $\text{Subtr}_{|T'|}(\text{Active}(T') \;\cap\; \text{Suff}(B) \;\cap\; \{m-|B|, m-|B|+1, \ldots, m-1\})$

The new $\text{Active}(T'B)$ value considers that, since a new block B has been added to T', the pattern prefixes that are suffixes of $T'B$ are those that are already suffixes of B (i.e. $\text{Pref}(B)$), or those which are suffixes of T' and are followed by B in the pattern. As before, Right does the trick of considering both cases in a single formula.

The new value $\text{Res}(T'B)$ adds to $\text{Res}(T')$ not only the matches which are completely inside B, but also those which appear when T' is concatenated to B. For this sake, we consider pattern prefixes which are suffixes of T' (i.e. $\text{Active}(T')$), and which are followed by the corresponding pattern suffix in B. The final intersection ensures that the complete pattern has appeared. Figure 3 illustrates.

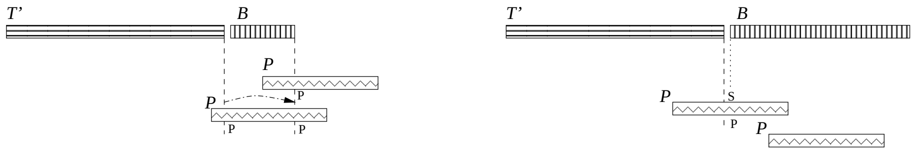

Figure 3: Updating the state of the search. In the first case we illustrate the updating of $\text{Active}(T')$ (a short block is added). In the second case we show how the matches are updated (when a long block is added). In general both updates are necessary.

2.4 Extended Patterns

We show now how to handle some extended patterns in this paradigm. A first alternative, which is easily implemented using bit-parallelism (see next section) is to allow *classes of characters*, i.e. the pattern at position i matches not just one letter but a set of letters. The pattern can then be seen as a sequence of sets of characters $P = c_1 \ldots c_m$, $c_i \subseteq \Sigma$. This is easily implemented by modifying our equations for a single letter a, so that instead of $P = xay$ we require $P = xAy \;\wedge\; a \in A$.

Approximate searching can also be performed in this scenario. In this case we allow that the pattern does not match exactly but up to k *errors*. An alternative definition is that we want all the text segments t such that $dist(t, P) \leq k$, where $dist(a, b)$ gives the minimum number of operations (errors) to perform on P or t to transform one into the other. There are many choices to define what is an error, but the most commonly used are: allow substitutions (Hamming distance) or allow substitutions, insertions, and deletions (Levenshtein distance). If we want to search all the pattern occurrences with up to $0 < k < m$ substitution errors, we keep for each block and each $i \in 0..k$ a description $D_i(B)$. It represents all the block information when the matches are allowed to contain up to i errors. There is a different search state for each i (i.e. $\text{Active}_i(T')$), representing that a pattern prefix matches a text suffix with up to i errors. $\text{Res}(T')$ is the same for any i, and keeps track of the matches allowing k errors.

We write $\text{Pref}_i(B)$ and $\text{Suff}_i(B)$ when we refer to $D_i(B)$, while the other components are independent on i. $\text{Matches}(B)$ refers to the matches allowing up to k errors which occurred completely inside the block.

Given $D(B) = (L, O, S, P, M)$ and $D(B') = (L, O, S', P', M)$ we define $D(B) \cup D(B') = (L, O, S \cup S', P \cup P', M)$ as their union. With this notation we can express the concatenation of two blocks $D_i(B) = D_i(B_1 B_2)$, allowing i errors:

$$D_i(B) = \bigcup_{j=0}^{i} D_j(B_1) \cdot D_{i-j}(B_2)$$

(where we recall that $\cdot$ represents the concatenation of descriptions). The reason for this formula is as follows: imagine that we search with $k = 2$ errors. Then, we can pair a prefix that matched with zero errors with a suffix that matched with two errors, or a prefix that matched with one error with a suffix that matched with one error, or a prefix that matched with two errors with a suffix that matched with zero errors. In general, the sum of errors between prefix and suffix must be k. This is easily generalized if we are interested in $i \leq k$ errors.

To update the $\text{Active}_i(T')$ values we use a similar idea, i.e.

$$\text{Active}_i(T'B) \quad = \quad \bigcup_{j=0}^{i} \text{Right}_{|B|}(\text{Active}_j(T')) \ \cap \ \text{Pref}_{i-j}(B)$$

where the rationale is the same as before: we match with i errors if we already matched a pattern prefix with j errors and the block starts with the corresponding pattern suffix matched with $i - j$ errors. Figure 4 illustrates.

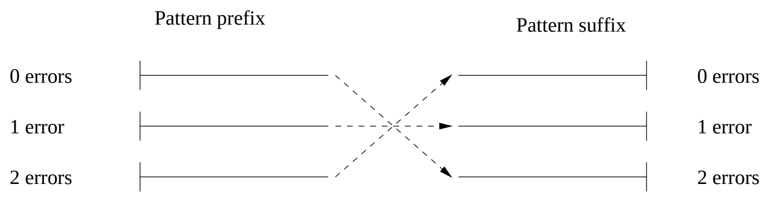

Figure 4: Updating $\text{Active}(T')$ when 2 errors are allowed.

The $\text{Res}(T')$ value is interested only in k errors:

$$\begin{aligned} \text{Res}(T'B) \quad = \quad & \text{Res}(T') \ \cup \ \text{Add}_{|T'|}(\text{Matches}(B)) \\ & \cup \ \text{Subtr}_{|T'|}(\text{Active}_k(T') \ \cap \ \text{Suff}_k(B) \ \cap \ \{m - |B|, m - |B| + 1, \ldots, m - 1\}) \end{aligned}$$

while however the other $\text{Active}_i(T')$ values are necessary to maintain $\text{Active}_k(T')$.

The values for an individual letter a is also modified:

- $\text{Suff}_i(B) \quad = \quad \{0, 1, 2, \ldots, m-1\}$
- $\text{Pref}_i(B) \quad = \quad \{1, 2, 3, \ldots, m-1\}$
- $\text{Matches}(B) \quad = \quad \text{if } (m = k+1 \ \wedge \ \exists x, y \ P = xay) \text{ then } \{0\} \text{ else } \emptyset$

Notice that if $k > 0$ (i.e. our case of interest), then a single letter matches at any pattern position. On the other hand, the pattern matches inside the letter only if we can delete all its letters and leave a single one which is equal to a (the case of deleting all the letters is not considered because it implies $m = k$, which is a trivial problem).

The case of the Levenshtein distance is more complicated, because the matches may not have the same length. In this case we need to store, instead of initial and final positions of matches ($\text{Pref}_i(B)$ and $\text{Suff}_i(B)$), the segments of the pattern that match with i errors or less. We define

$$\text{Segm}_i(B) \quad = \quad \{(i, j), \ dist(P_{i..j-1}, B) \leq k\}$$

instead of $\text{Pref}_i(B)$ and $\text{Suff}_i(B)$. For block concatenation, instead of intersecting prefixes of B_1 with suffixes of B_2, we consider the segments of B_1 immediately followed by segments of B_2 and take their concatenation. We do not work out the details because, as explained in the next section, we do not have an efficient implementation (the naive implementation is $O(m^2k^2n)$ if we perform $O(n)$ block concatenations).

3 A Bit-Parallel Implementation

Until now, we have defined our algorithms in terms of sets of pattern positions. We present now a very well-suited implementation paradigm which allows to convert the previous algorithms into efficient implementations.

We use the technique called *bit-parallelism* [3, 36]. This technique takes advantage of the fact that the processor works in parallel on all the bits of the computer word. We call w the number of bits of the computer word, which is 32 or 64 in current architectures. If one is able to map the elements of a set onto bits, and to express the operations to perform on them by using only the operators provided by the processor (which are rather limited, i.e. bit shifts, masking, etc.), then one can effectively parallelize the work on the set, obtaining speedups of up to $O(w)$ over the original algorithm.

Our aim is to represent a set of pattern positions (which is a subset of $\{0 \ldots m-1\}$) as a bit mask of m bits. The i-th bit of the bit mask will be 1 (and said to be "active") if and only if position $i-1$ belongs to the represented set. When writing down bit masks, the first bit position is the rightmost one, and exponentiation is used to denote bit repetition, e.g., $0^3 1$ means 0001, where only the first bit is active. We speak of bit masks of length m even if $m > w$, in which case we would actually need $\lceil m/w \rceil$ actual computer registers to represent the bit mask.

To write down the operations done on bit masks, we use a C-like syntax: "|" is the bitwise-or of two bit masks, and represents set union; "&" is the bitwise-and of two bit masks, and represents set intersection; "$<< \ell$" is a bit shift operation which assigns the i-th bit to the $(i+\ell)$-th, setting the first ℓ bits to zero; and "$>> \ell$" does the same in the other direction.

The sets $\text{Pref}(B)$, $\text{Suff}(B)$, and $\text{Active}(T')$ will be represented as bit masks. For blocks of one letter a we have $\text{Suff}(B) = S[a]$ and $\text{Pref}(B) = (S[a] << 1)$, where S is a precomputed bit mask table such that, for any $a \in \Sigma$, the i-th bit of $S[a]$ is active if and only if $P_i = a$.

The formulas to concatenate blocks are directly translated by noticing that:

- $\text{Left}_\ell(X)$ is computed as $(X >> \ell) \mid 1^i 0^{m-i}$.
- $\text{Right}_\ell(X)$ is computed as $(X << \ell) \mid 0^{m-i-1} 1^i 0$.
- $X \cup Y$ is computed as $X \mid Y$.

- $X \cap Y$ is computed as $X \ \& \ Y$.

For example, the formula to update Pref(B) in Section 2.2 is computed as

$$\text{Pref}(B) \quad = \quad \text{Pref}(B_2) \ \& \ ((\text{Pref}(B_1) << \ell) \mid 0^{m-i-1}1^i0)$$

Hence, all those operations on sets are performed in $O(1)$ time if $m \leq w$, and $O(m/w)$ time in general. In practical text searching we can assume $m = O(w)$.

On the other hand, the sets Res(T') and Matches(B) are explicitly stored in an array. However, it is not difficult to see that the total amount of work to handle them is $O(R)$, where R is the number of occurrences of the pattern in the text. The cost cannot be $o(R)$ if we report all the occurrences.

Hence, if $f(n)$ concatenations are performed along all the process, our total search cost is $O(f(n) + R)$. The value of $f(n)$ depends on the compression algorithm. We have also to add a preprocessing cost to build the $S[\,]$ table, which is $O((\sigma + m)\lceil m/w \rceil)$.

The bit-parallel paradigm allows to seamlessly expand the type of patterns we are able to search. Since the T table is the only connection between the pattern and the search, we can for instance allow having classes of characters, that is, each pattern position matches with a set of characters instead of just one character. To achieve this, just set the i-th bit of $S[a]$ to "match" for any $a \in P_i$. Other extended patterns considered in [36], such as regular expressions, are not easily adapted to this scheme. It is also possible to handle errors in the matches, such as replacement errors [4] (at $O(m \log(k)/w)$ cost per character) or insertions, deletions and replacements at $O(mk/w)$ [36, 5] or even $O(m/w)$ [28] cost per character. The implementation of our technique to handle mismatches is $O(k^2 f(n) + R)$ cost, while the extension for Levenshtein distance is not easily implemented.

In all cases, the space complexity of our algorithms is $O(n\lceil m/w \rceil + R)$, since we need to store the descriptions of the blocks already seen and the matches found. Notice that this n refers in fact to the size of the compression window, and the R to the matches present in that window only.

Finally, we consider the practical problem of uncompressing a neighborhood of the occurrences. In practice it is undesirable that we just give the text positions matching the pattern. It is much better to uncompress and show a neighborhood of the match. This neighborhood can be defined as the line holding the occurrence, the record (delimited by some given pattern), a fixed number of characters, etc.

Assume that we find a pattern occurrence in the compressed text and want to show a neighborhood of the occurrence. Since we have searched up to that point, we have the information to decompress the surrounding blocks forward and backward, until from the plain text obtained we determine that the neighborhood has been decompressed. To decompress a block we have two cases: (a) the block is a letter, in which case we deliver the letter, (b) the block is a concatenation of other blocks, in which case we decompress each of those blocks in turn. This process takes $O(N)$ time at most (where N is the size of the decompressed neighborhood), since at each step we either obtain one character of N or split the final text to be obtained, and it is not possible to split it more than $O(N)$ times. This shows that it is practical to show a part of a Ziv-Lempel compressed file without necessarily uncompressing the whole file.

4 LZ78 Compression

4.1 Compression Algorithm

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [39]) 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 $t_1 \ldots t_i$ of T has been already compressed in a sequence of blocks $Z = b_1 \ldots b_c$, all them in the dictionary, then we look for the longest prefix of the rest of the text $t_{i+1} \ldots t_u$ which is a block of the dictionary. Once we found this block, say b_k of length l_k, we construct a new block $b_{c+1} = (k, t_{i+l_k+1})$, we write the pair at the end of the compressed file Z, i.e $Z = b_1 \dots b_c b_{c+1}$, 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 5. 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 $(1, s)$.

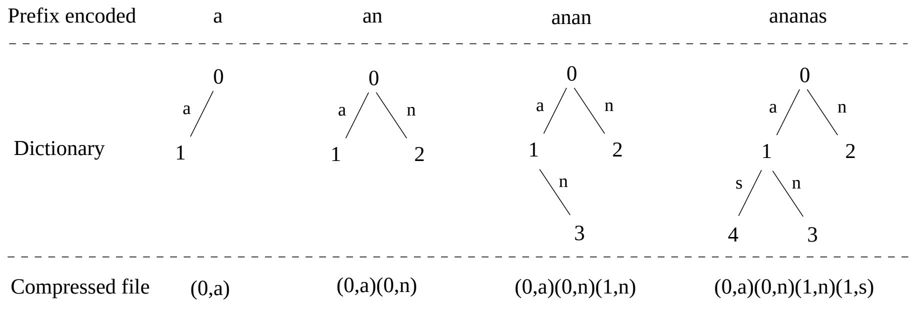

Figure 5: Compression of the word *ananas* with the algorithm LZ78.

The compression algorithm is $O(u)$ 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 c is read at the c-th step of the algorithm), although this time it is not convenient to have a trie, and an array implementation is preferable. Compared to LZ77, the compression is rather fast but decompression is slow. LZ78 is used by Unix's *Compress* program.

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 update the window. A particularly interesting variant is from Welch, called LZW [34]. 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). A variant over this is presented by Miller and Wegman [24] (which we call LZMW), 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.

4.2 Pattern Matching in LZ78 Compressed Files

Our general algorithm for searching in a sequence of blocks $Z = b_1 \dots b_n$ can be directly applied if we consider the new letter added after each block created by the LZ78 compression algorithm as a separate block. That is, each new pair (k, a) read at step c is taken as a reference to a previous block (b_k) followed by a literal block (a). Hence, we compute the description of the concatenation of b_k and a and add it as the new block b_c to our dictionary. At the same time, we update the state of the search using the description of b_c just computed. Of course, in practice we manage this one-letter block in a special way, to speed-up the block concatenation. We keep all the descriptions of the blocks b_k in an array which is directly accessed.

The algorithm we obtain is quite the same as in [2]. The main differences are that we obtain this algorithm as a particular case of a general string search algorithm for text that comes in blocks, that their algorithm is originally designed for LZW compression, and that we search all the occurrences of the pattern, not only the first one. Moreover, we present a practical implementation based on bit-parallelism, while [2] is a theoretical work that has not been implemented. To our knowledge ours is the first real implementation of this algorithm[1]. It is quite easy to adapt our algorithm to work on other variants of LZ78, such as LZW or LZMW. In particular we can easily adapt to different window management policies. The simplest one is that when the compressor memory is full, the dictionary is deleted and compression is restarted. Others try to remove the least interesting blocks from the dictionary, e.g. [13]. Our searcher can follow the same steps of the compressor along the search, using the same amount of memory.

4.3 Analysis

The theoretical complexity of the pattern matching algorithm is $O(n\lceil m/w\rceil + R)$, which becomes $O(n+R)$ on short patterns. If $n = o(u)$, this is faster than searching in the uncompressed text. In practical terms, the algorithm is rather efficient since no extra work apart from one block concatenation and one update of the search is performed per element of the compressed file.

Our experimental results, however (Section 7), show that the algorithm takes in practice twice the time of a Shift-Or run on the uncompressed text. This is because Shift-Or is very simple, and although we process many characters of the uncompressed text in one shot, in practice the cost of each step is big enough to amortize any possible gain due to compression. A specific problem is the locality of reference: the compressed matching algorithm reads random positions in the array of block definitions, while the uncompressed algorithm works basically in-place. The caching mechanism of the computer largely favors this last approach.

However, there is a positive result. Searching the compressed file with this algorithm is twice as fast as decompressing it and then searching the uncompressed file. For this comparison we are assuming that the file is compressed with LZ77 (which is much faster than LZ78 to decompress) and consider the time of *gunzip*, which is an optimized decompression software. Hence, if the text collection is kept compressed (which is definitely of interest) then it is much faster to search directly the compressed files.

We have tried to further improve our algorithm. For instance, we have created a variant called Mark-LZ78. In this compression algorithm, we mark with a bit flag for each block if the block is a leaf of the dictionary trie or not, to avoid storing the block description if this block is not used anymore. However, as we show in the experiments, the performance does not improve.

5 LZ77 Compression

5.1 Compression Algorithm

The Ziv-Lempel compression algorithm of 1977 (usually named LZ77 [38]) is, in some sense, simpler than LZ78, since the basic idea is just to recognize two repeated segments of the text and to mark the second as a reference (position in the text and length of the repeated part) to the first one. More formally, assume that a prefix $t_1 \ldots t_i$ of T has been already compressed in a sequence of blocks $Z = b_1 \ldots b_c$. We look for the longest prefix v of $t_{i+1} \ldots t_u$ which appears already in $t_1 \ldots t_i t_{i+1} \ldots t_{i+|v|-1}$. Once we have it, say that we find it starting at position $j \leq i$, we add a new block $(j, |v|)$ to the compressed file Z. A special case occurs if v is empty, in which case t_{i+1} is a new letter and we code it with a special block $(0, t_{i+1})$. With the same example *ananas*, we obtained: $(0,a)$ *nanas*; $(0,a)(0,n)$ *anas*; $(0,a)(0,n)(1,3)$ *s*; $(0,a)(0,n)(1,3)(0,s)$.

Notice that the above definition allows that the referenced block overlaps the one which is being compressed. Another variant avoids this for simplicity, i.e. v must be found in $t_1 \ldots t_i$. In this case the compression of *ananas* becomes: $(0,a)$ *nanas*; $(0,a)(0,n)$ *anas*; $(0,a)(0,n)(1,2)$ *as*; $(0,a)(0,n)(1,2)(1,1)$ *s*; $(0,a)(0,n)(1,2)(1,1)(0,s)$.

Yet another variant codes the repeated block and then the letter which follows it in the still uncompressed text. There are many other variants as well, mainly related to how to represent the pairs in the compressed

[1]See, however, [19], in this very same conference.

file and how to compress fast. In general, the position j is coded as the difference $i+1-j$, since the last occurrence of the block is used and v is normally restricted to not appear too far away from t_i.

LZ77 compresses more than LZ78, both in theory and in practice. From a theoretical point of view, LZ77 can reference any text substring seen before, while LZ78 can only reference a subset of those strings. In particular, the LZ77 variant that allows overlaps can obtain a compressed file of $O(1)$ blocks in the best case, while the one not allowing overlaps obtains at most $O(\log u)$. LZ78, on the other case, cannot obtain less than $O(\sqrt{u})$. This is easily seen by considering the best-case file $T = a^u$. In practice it is also true that LZ77 compresses more than LZ78. LZ77 is implemented in the Gnu *gzip* program.

Compression is rather slow with LZ77. It is expensive in time and space to find the longest prefix of the uncompressed part of the file that appears already in the compressed part. In theory, the compression is $O(u)$ in time and space by the use of a suffix tree or a DAWG automaton [38, 37]. In practice, the search in done in a buffer window and an large hash table is normally used, as in *gzip*. An experimental comparison of different techniques to find the prefix can be found in [7]. The decompression algorithm, on the other hand, is very fast (faster than for LZ78) because to decompress a block is it just necessary to copy a part of the text and no dictionary has to be kept.

5.2 Pattern Matching in LZ77 Compressed Files

Our algorithm for LZ77 is an adaptation of the general algorithm on blocks, with a main difference. On LZ77 compressed files, when we want to process a new block, the situation shown in Figure 6 generally occurs: the new block references a sequence of r contiguous previously processed blocks, but it overlaps with the first and last one (u and v in the Figure). That is, the new block does not exactly correspond to previously processed blocks. Therefore, we do not have all the information on the blocks u and v that we need to concatenate the blocks.

We solve this by computing recursively the descriptions of the two blocks u and v with the same method. That is, we simulate that we are back in the text, where those blocks appeared, and compute their description (this may trigger more recursive invocations with the same purpose). When we finally obtain the descriptions of u and v, we concatenate all the referenced blocks to obtain the description of the new block. Another possibility is that the new block is completely inside another block already processed, in which case we have to recursively consider the blocks that define the referenced block.

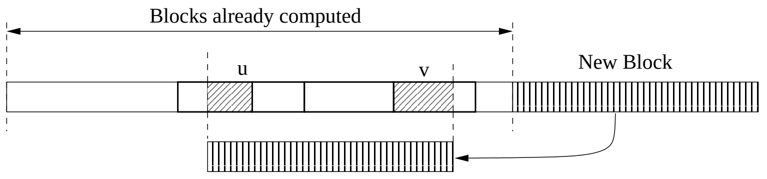

Figure 6: Recursive computation of the description of a block in LZ77 compressed files.

We explain now a technique to concatenate the r blocks in low average time. Instead of computing Pref(B) and Suff(B) of the first block, then concatenating with the second, then to the third, until the r blocks are concatenated, we compute Suff(B) from the first block to the r-th and Pref(B) from the r-th block to the first one. We analyze this shortly.

5.3 Analysis and Improvements

We analyze now the many aspects of our algorithm and propose some improvements.

Block concatenation. If we use the proposed block concatenation technique, we have that in the worst case only the first m blocks can affect Suff(B) and only the last m blocks can affect Pref(B), so the worst case time for concatenating the blocks becomes $O(\min(u, mn))$.

We show now that on average only $O(\log m)$ blocks are processed until Suff(B) becomes stable. Each new block character we process will either extend the current suffixes of the set Suff(B) or make them disappear from the set. Each suffix is removed from the set with probability $1 - 1/\sigma$ (i.e. if the new character block cannot extend it). Before we read the block characters all the m pattern positions are in Suff(B), and therefore on average no pattern positions remain in the set after $O(\log m)$ block characters are read (after the i-th character is read, the pattern positions $m - i$ to $m - 1$ cannot be removed from the set, but their situation cannot change anyway).

Even if we consider all blocks of length 1 (the worst), we work on average $O(n \log m)$ because of concatenations. The same reasoning holds for Pref(B).

The only part of the block concatenation which cannot skip blocks is the computation of Matches(B). However, this adds up $O(R)$ time along all the search. Therefore, the total time for block concatenation is $O(\min(u, n \log m) + R)$ on average.

Finding the blocks. We consider now how to find the indices of the block that define a text position j. We keep an array with the blocks already seen. Binary searching the text position among these blocks adds $O(n \log n)$ to the cost. Instead, we keep a table of $O(n)$ entries where the element i points to the block where the text position $\lfloor iu/n \rfloor$ is defined. By accessing this table we directly arrive at the correct block with an average inaccuracy of $O(u/n)$, and a final binary search finds the correct position, for a total cost of $O(n \log(u/n))$ (in practice a linear search is faster for the final part). This gives good results in practice. Another alternative is that the compressor does not store the text position and length of the repeated part, but instead it gives the block numbers involved and the offsets inside u and v. Since a text position needs $O(\log u)$ bits and a block number plus an offset inside the block needs on average $\lceil \log_2 n \rceil + \lceil \log_2(u/n) \rceil = O(\log u)$ bits, the *order* of compression ratio should not worsen. We show in the experiments that this version of the algorithm (called Block-LZ77) is faster than the plain version, since no searching of the text position is necessary. However, compression ratios worsen significantly in practice due to round-offs.

Computing partial blocks. However, the really costly part of the algorithm is not here, but in the recursive computation of the partial blocks u and v. If we consider that each time we perform a recursive call we "split" the original block B at a new position, then it is clear that at most $|B|$ recursive calls can be done until we have split it in single characters and therefore we have found the definition of each one. This shows that the total cost of the recursive calls is $O(u)$ in the worst case. Our experiments suggest that this is also the average case, but we were not able to prove it.

Consider now the cost of the recursive invocations in the case where the new block B is strictly inside its referencing block. For instance, a letter which repeats inside a large block could trigger a long chain of recursive invocations until its real definition is found. In the worst case, we could have a block of size s which references one of size $s - 1$, and this one references another of size $s - 2$, and so on. We would work $O(s)$, but the size of the text at that point is $O(s^2)$. Hence, at text position i we cannot work more than $\sqrt{i}$, which gives a total worst-case cost of $O(n\sqrt{u})$, which is too high. This problem does not disappear if the compressor always stores the first occurrence of the repeated block instead of the last one, because we may not point to the first occurrence when we consider partial blocks.

Hence the total amount of work is $\omega(u)$ in the worst case whenever $n = \omega(\sqrt{u})$, and we conjecture that this is also the average case. See the left plot of Figure 7, where we have experimented with the *English* text described in Section 7. Least squares fitting shows that a good model for the number of recursive invocations per text character is $0.177{+}0.1 \ln u$ (with less than 0.5% error in the approximation). The experiment suggests that the algorithm is $O(u \log u)$ on average. This is, unfortunately, worse than uncompressing and searching. We present now some techniques to improve this situation.

Improvements. A first improvement we tried consisted in storing more information than simply one description per block. For instance, when we compute the description for the partial blocks u and v (which are not part of the original sequence of blocks), we could store instead of discarding them. If later another block needs the description of u and v, we have already computed them. Figure 7 (right plot) shows that the total amount of recursive calls is reduced using this technique, and we conjecture that in this case we work $O(u)$ (least squares fitting yields a complexity of $O(u^{0.99927})$). These blocks, however, cannot be easily stored in the array of blocks since they do not belong to the sequence. A hashing implementation gave bad results in practice, that is, the cost to add the new blocks outweighted the gains of having them already computed. This could change for longer texts, if the orders of the two algorithms are different.

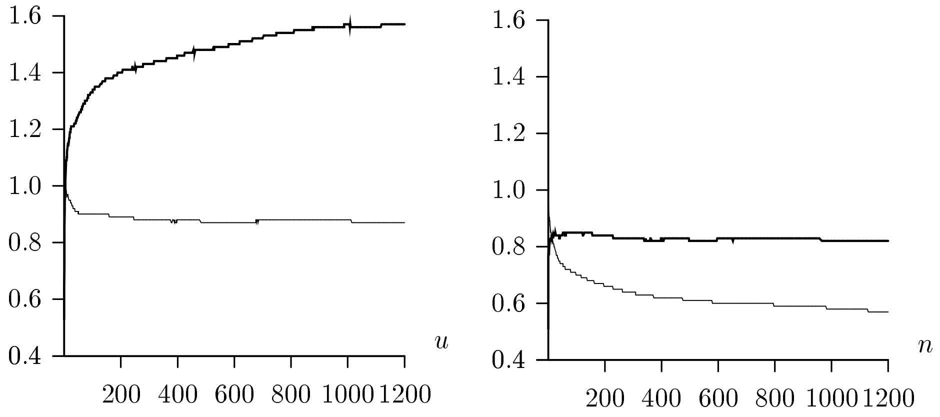

Figure 7: Number of recursive invocations (thick line) and block concatenations (thin line) per text character, for natural language text. The left plot shows the basic algorithm and the right plot shows the improvement of adding the computed blocks.

Another improvement, which gave good practical results, was to try to compute *less* (instead of more) information. Our aim was to avoid the recursive computation of u and v. Hence, instead of computing their descriptions recursively, we pessimistically assume that they match all the pattern positions. If they are short enough we will not have a match even assuming this, and we could process them without actually obtaining their descriptions. Only when we find a (possible) match we backtrack to the point where it could have been started and compute correctly the involved blocks. For each block, we store whether it has been correctly or pessimistically computed. As we show in the experiments, this improves search time for patterns of length 15 or more in practice. However, the method is limited since we cannot skip more than m characters of T without having at least one character correctly computed, hence in the very best case we pay $O(u/m)$ with this speedup. We call this algorithm Skip-LZ77 (and combined with Block-LZ77 it yields Skip-Block-LZ77).

Final remarks. Even with all these improvements, the experiments show that this algorithm is much slower than decompressing (with *gunzip*) and searching (with Shift-Or). We believe that it is not possible in practice to beat a decompress-then-search approach. The root of this limitation lies in the need to recursively compute u and v. Another consequence of the existence of partial blocks is that, even if the compressor uses a window of fixed size to select the strings to repeat, we need to keep in memory all the previous blocks, since even if they are not directly referenced anymore, we may need to resort to them in case of partial blocks. We propose in the next section a slightly different compression scheme which gets rid of all the aspects of LZ77 compression that degrade the searching performance.

We finish this section with a couple of comments. First, as it is clear from the algorithm, we do not handle the case of overlapping compression, i.e. when the referenced block can overlap with the new block B. Although we could handle it, the result is the same in cost as if the compressor avoided such overlapping (i.e. performing many steps, where a step ends when an overlap occurs). Second, other variants of LZ77 are easily accommodated. Finally, we notice that a neighborhood of size N around the occurrences can be obtained using the general mechanism at $O(N\sqrt{u})$ cost (or, according to the empirical results, $O(N \log u)$ cost). This is because of the cost to find the definitions of the incomplete blocks.

6 LZ-Blocks: A New Hybrid Compression Algorithm

It became clear in the previous section that the worst part of the cost of the LZ77 search algorithm was due to the cost of recursively computing partial blocks, and of finding the block corresponding to a text position. We design a new compression algorithm between LZ78 and LZ77, to have multiple-block compression (not just one block like in LZ78), but also to avoid the recursive situation which appears in searching LZ77-compressed files (Figure 6).

We propose the following algorithm. Assume that a prefix $t_1 \ldots t_i$ of T has been already compressed in a sequence of block $Z = b_1 \ldots b_c$. We look now for the longest prefix v of $t_{i+1} \ldots t_u$ which is represented by a sequence $b_r \ldots b_{r+h}$ already present in the compressed file. If there are many alternative choices for the same v, we take the one with the minimum of blocks (to reduce the cost of concatenations). And if still several possibilities occur, we take the first occurrence (the minimum in the number of the first block). We code this new block by (r, h). As in LZ77, if v is empty (i.e the letter t_{i+1} is new), we code a special block $(0, t_{i+1})$. 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 main advantage of this compression scheme is that it avoids the recursive case in the LZ77 pattern matching (Figure 6), because we know already that the new block corresponds directly to a concatenation of already processed blocks. Moreover, we do not need to search the text position in the blocks, since we can directly access the relevant blocks.

The compression can still be performed in $O(u)$ time by using a sparse suffix tree [17] 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 is slower than for LZ77, since we need to keep track of the blocks already seen to be able to retrieve the appropriate text. Finally, the compression ratio is in principle worse than for LZ77 since we are limited in the text segments that we can use. On the other hand, the numbers to code are smaller since we code block positions in $O(\log n)$ bits instead of text positions in $O(\log u)$ bits. Moreover, if we use a simple trick, the compression is in general better than for LZ78 since we are not limited to using just one block. The trick is to represent the pairs $(r, 0)$ as $(2r)$, and the pairs $(r, h+1)$ as $(2r+1, h)$. This pays off because the second element of the pair is frequently zero.

The searching algorithm is like that of LZ77 except because we do not need to search for the blocks and we do not have to recursively find the partial blocks u and v (they simply do not exist now). From the analysis of the LZ77 pattern matching algorithm we have that we work $O(\min(u, n \log m) + n\lceil m/w \rceil + R)$ on average and $O(\min(u, mn) + n\lceil m/w \rceil + R)$ in the worst case (thanks to the improved algorithm to concatenate blocks). In practice, this algorithm performance is very close to LZ78 pattern matching. We also tried a marked version (called Mark-LZBlocks) where for each block a bit is stored which tells whether or not the block will be used again, but as for LZ78, the search time does not improve in practice.

Unlike LZ77, we can use less memory if the compressor restricts the references to a window of the text. Since there are no recursive references, those blocks which are far away in the past need not be stored since they will not be referenced anymore. Hence, as in LZ78, we need the same memory as the compressor. A window of size N can be displayed in $O(N)$ time.

7 Experimental Results

We show in this section our empirical results on the behavior or our search and compression schemes. We first study the compression techniques and later the search performance.

We use mainly two files for the experiments. One is an *English* literary text (from B. Franklin) of 1.29 Mb, filtered to lower-case and with separators normalized. The other is the *DNA* chain of "h.influenzae", of 1.36 Mb. For comparative purposes, we also show the results on some files of the the Calgary Corpus[2]: two books (book*), six troff-formatted scientific articles (paper*) and three source program codes (prog*).

7.1 Compression Performance

It is interesting to study the compression performance of the algorithms for two reasons: first, we propose LZ-Blocks, a hybrid compression scheme which we have to evaluate in terms of compression ratios. Second, our search algorithms use a technique to code the pairs which speeds up search time but which is suboptimal: the numbers are stored in as many bytes as needed (using the highest bit to denote if there are more bytes or not).

We first compare the number of bits needed to code a file with LZ-Blocks against the same number for LZ77 and LZ78. We call this approach "bit-coding". This is aimed to give and idea of the expected compression performance when the file is compressed with a real technique (such as Elias [11] or Huffman codes). Many other improvements are possible. A deeper study of the best techniques for LZ-Blocks is deferred for future study.

Table 1 shows the results. The "Ideal" column counts exactly the bits used by each number stored in the compressed file, while both "Elias" columns count the number of bits needed to represent the numbers using these codes[3] [11]. The letters, on the other hand, are Huffman coded. For *English* and *DNA* we show in a second line the percentages for different variants of the compressors: Block-LZ77, Mark-LZ78 and Mark-LZBlocks, respectively. With LZ-Blocks we obtain estimated compression ratios comparable to LZ77. The LZ-Blocks and LZ77 compression are better than LZ78 except for *DNA*, where only two bits are necessary to code a letter. Block-LZ77, on the other hand, compresses quite badly.

We now perform a practical comparison using our byte-coding techniques against good LZ77 and LZ78 compressors, namely *gzip* and *Compress* respectively. This is to show how much compression are we loosing in order to ease the searching process.

Table 2 shows the compression ratios achieved. The percentages in the second row of *English* and *DNA* have the same meaning as before. Interestingly, *Compress* is better than *gzip* on *DNA*, which rarely happens on natural language texts. Our compression ratios show a penalty with respect to those of *gzip*. Our byte compression method is very simple, and these results show in which proportion our compression ratios could be improved by engineering techniques, keeping in mind that complicating the encoding of the numbers risks slowing down the pattern matching process.

7.2 Search Algorithms

We compare now the search time for our algorithms against the decompressing and searching approach. The experiments were run on a Sun UltraSparc-1 of 167 MHz, with 64 Mb of RAM, running Solaris 2.5.1. We consider user time, which is within 2% of accuracy with 95% confidence. Time is expressed in seconds everywhere in this section.

In general, searching a compressed text has the additional advantage over the uncompressed text that it performs less I/O. However, this is relevant if we compare compressed versus uncompressed searching. This is not what we compare here: we consider that the text is always compressed. Hence, we measure the cost

[2]`ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus/`

[3]Recall that Elias-γ precedes the number x by its length in unary, while Elias-δ uses Elias-γ to code that length that precedes the number.

File	Size	Ideal			Elias-γ			Elias-δ		
	(Kb)	LZ77	LZ78	LZBlocks	LZ77	LZ78	LZBlocks	LZ77	LZ78	LZBlocks
English	1,324	29.67%	36.15%	29.28%	59.34%	64.01%	58.57%	48.96%	52.04%	46.17%
		52.45%	*38.01%*	*31.24%*	*104.9%*	*82.31%*	*62.48%*	*74.25%*	*54.71%*	*48.75%*
DNA	1,390	28.03%	25.30%	29.08%	56.06%	47.33%	58.18%	45.77%	37.71%	46.40%
		47.21%	*26.77%*	*31.15%*	*94.43%*	*67.62%*	*62.30%*	*73.14%*	*39.91%*	*49.03%*
book1	751	34.10%	40.70%	35.62%	68.20%	70.83%	71.25%	41.26%	44.96%	41.50%
book2	597	29.33%	40.21%	30.44%	58.66%	69.46%	60.89%	35.51%	44.41%	35.72%
paper1	52	32.33%	46.20%	34.29%	64.53%	77.01%	68.59%	41.05%	51.92%	41.91%
paper2	80	32.68%	43.00%	34.80%	65.27%	72.84%	69.60%	41.08%	48.28%	42.01%
paper3	45	35.10%	45.50%	38.12%	70.07%	76.23%	76.24%	44.84%	51.36%	46.55%
paper4	13	37.60%	47.95%	41.07%	74.74%	78.30%	82.15%	49.92%	54.81%	51.55%
paper5	12	39.85%	50.79%	41.74%	79.13%	82.42%	83.49%	52.63%	57.92%	52.39%
paper6	37	33.60%	47.72%	35.69%	67.03%	79.08%	71.38%	42.91%	53.72%	43.81%
progc	39	32.21%	47.99%	34.16%	64.24%	79.14%	68.32%	41.24%	53.96%	41.95%
progl	70	22.45%	39.10%	23.30%	44.82%	65.83%	44.92%	28.04%	43.85%	27.65%
progp	48	21.34%	40.36%	22.46%	42.54%	66.95%	46.60%	27.16%	45.33%	28.46%

Table 1: Estimated compression ratios with three different methods. For each number in the compressed file, if we note n the bits needed to code it, then *Ideal* counts only n, *Elias-γ* counts $2n$ and *Elias-δ* counts $n + 2\lceil \log_2 n \rceil$. The second line (in italics) of *English* and *DNA* correspond to Block-LZ77, Mark-LZ78 and Mark-LZBlocks, respectively.

File	*gzip*	*Compress*	Byte-LZ77	Byte-LZ78	Byte-LZBlocks
English	35.58%	38.90%	44.49%	54.41%	43.29%
			79.32%	*56.20%*	*45.24%*
DNA	30.44%	27.96%	41.07%	43.17%	42.23%
			75.24%	*44.90%*	*44.22%*
book1	40.76%	43.19%	53.21%	59.92%	53.30%
book2	33.83%	41.05%	45.60%	58.55%	46.53%
paper1	34.94%	47.17%	54.70%	66.17%	52.67%
paper2	36.19%	43.99%	54.65%	62.02%	52.10%
paper3	38.89%	47.63%	60.19%	67.92%	58.75%
paper4	41.66%	52.36%	69.20%	75.71%	68.24%
paper5	41.78%	55.04%	72.27%	79.47%	68.16%
paper6	34.72%	49.06%	56.84%	69.33 %	54.76%
progc	33.51%	48.32%	54.97%	67.99%	51.95%
progl	22.71%	37.89%	37.82%	55.30%	35.47%
progp	22.77%	38.90%	35.97%	57.20%	34.20%

Table 2: Compression ratios for classical compressors and our byte versions. The second (italics) lines of *English* and *DNA* correspond to Block-LZ77, Mark-LZ78 and Mark-LZBlocks, respectively.

of searching it without decompressing versus the cost of decompressing it and then searching. Clearly the last task can be done using an intermediate buffer in main memory, and therefore the I/O is the same in both cases. Therefore, we will measure user time, which excludes I/O time.

When we compare our algorithms against decompressing plus searching, we have to bear in mind that, in this alternative, one can use any compression format (not necessarily LZ78, which happens to be the best for direct searching). Therefore, we have opted for *gzip/gunzip*, an LZ77-based optimized compression software that gives better compression and faster decompression when compared to other Ziv-Lempel based compressors. When considering the overall decompress-plus-search time, we add the user time of *gunzip* plus that of a search program run over the plain file. In our experience, the user time is almost the same as that of a specialized implementation using an internal buffer.

Figure 8 compares the marked and unmarked versions of LZ78 and LZ-Blocks. As it can be seen, there is no advantage in practice by the use of marking. Therefore, we do not further consider the marked versions. Another conclusion we take from the figure is that the searcher for LZ-Blocks is slightly faster than for LZ78 on *English* but slower for *DNA*. This may be related to the good performance of the LZ78 compressor on *DNA*.

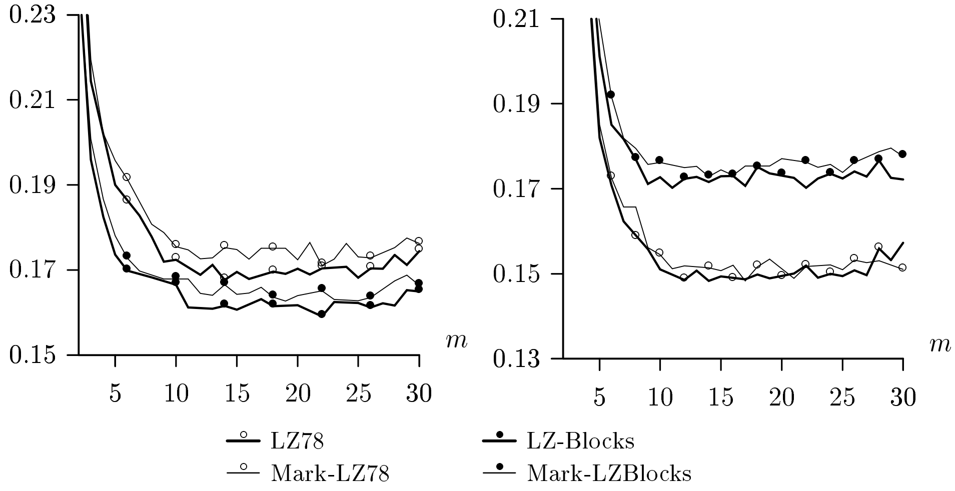

Figure 8: Comparison between the marked and unmarked versions of LZ78 and LZ-Blocks compressors. The left plot is for *English* text and the right one for *DNA*. The y axis is the user time in seconds for the whole files.

Figure 9 compares all the search algorithms together, as well as decompression (with *gunzip*) plus search time (with Shift-Or and BNDM [29], a bit-parallel searcher which is the fastest in practice together with [33]). It can be seen that Block-LZ77 improves significantly over LZ77, and that the Skip-LZ77 versions improve as the pattern length grows. However, all the LZ77 search algorithms are not competitive against decompressing and searching, especially on *DNA*. On the other hand, both the LZ-Blocks and LZ78 search algorithms are twice as fast as decompressing and searching.

Table 3 compares the time to search a random 10-letter pattern on *English*, *DNA* and the selected files of the Calgary Corpus. We consider the time to decompress with *gunzip* and to search with Shift-Or (as seen, for $m = 10$ the time is very close to BNDM). We show the results for LZ78 and LZ-Blocks only, as LZ77 has been shown to be much inferior.

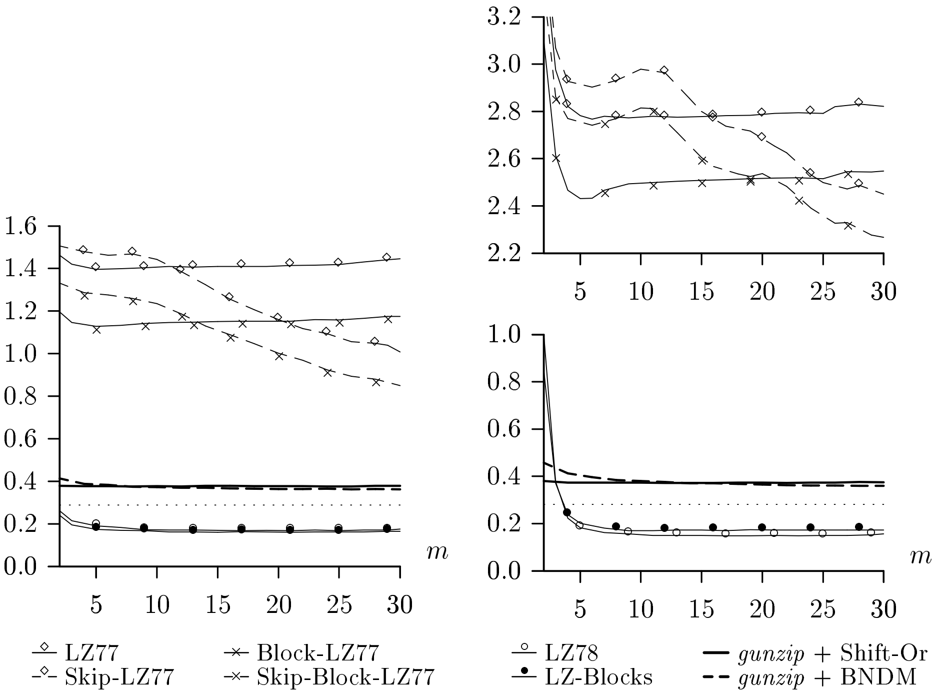

Figure 9: Comparison of the search algorithms. The dotted line is the time taken by *gunzip* alone. The left plot is for *English* text and the right one for *DNA*. The y axis is the user time in seconds for the whole files.

8 Conclusions

We have focused in the problem of string matching on Ziv-Lempel compressed text. This is an important practical problem, as it is of interest keep the texts compressed and at the same time being able to efficiently search on them.

We presented a general paradigm to search in a text that is expressed as a sequence of blocks, which abstracts the main features of Ziv-Lempel compression. Then, we applied the technique to the different variants, i.e. LZ77 and LZ78. For LZ78, we are able to search in half the time of uncompressing and searching, while for LZ77 our algorithm, is much slower. This motivated us to present LZ-Blocks, a new hybrid compression technique which allows to search as fast as in LZ78 but which keeps many of the features of LZ77 compression, being in practice similar in compression ratios.

Therefore, we are able to search in a compressed text faster than uncompressing and then searching. In general, on the other hand, searching on compressed text at the same speed of on uncompressed text seems difficult to achieve in practice because of a basic problem of locality of reference.

It is interesting to note that our algorithms are general enough to work on general *collage systems* (which encompass LZ77), and have good performance on *regular* collage systems (which encompass LZ78, LZW and LZ-Blocks) [18]. This model divides the compression format in 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. A regular collage system builds D using atomic elements and concatenation of other elements in D. As we have described our algorithms in terms of concatenations of blocks, the techniques immediately generalize to regular collage systems. General collage systems also permit repetition and truncation of other elements in D. More insights are given in [18] about the difficulty of searching on general collage systems.

Later work reported in [31] presents fast searching on LZ78/LZW by using Boyer-Moore techniques. Still, according to the experiments presented there, our approach is the fastest one for moderate length patterns ($m \leq 15$), which is a very common case in practice. Moreover, their approach has not yet been extended to

File	*gunzip*	Shift-Or	LZ78	LZ-Blocks
English	28.80	8.90	17.24 (45.7%)	16.65 (44.2%)
DNA	28.10	9.21	15.10 (40.5%)	17.27 (46.3%)
book1	18.40	4.92	10.91 (46.8%)	11.42 (49.0%)
book2	12.40	4.14	8.01 (48.4%)	7.78 (47.0%)
paper1	1.80	1.67	1.88 (54.2%)	1.92 (55.3%)
paper2	2.40	1.76	2.07 (49.8%)	2.18 (52.4%)
paper3	1.80	1.60	1.73 (50.9%)	1.88 (55.3%)
paper4	1.20	1.48	1.50 (56.0%)	1.59 (59.3%)
paper5	0.80	1.42	1.52 (68.5%)	1.54 (69.4%)
paper6	1.90	1.53	1.69 (49.3%)	1.78 (51.9%)
progc	1.50	1.55	1.73 (56.7%)	1.75 (57.4%)
progl	1.90	1.72	1.88 (51.9%)	1.84 (50.8%)
progp	1.20	1.62	1.74 (61.7%)	1.70 (60.3%)

Table 3: Search times for different files, in 1/100-th of seconds. The percentages indicate the time of the compressed searching as a fraction of uncompressing plus Shift-Or searching.

LZ-Blocks, although this seems possible as well.

Some open questions left involve studying better the performance of LZ-Blocks, both in theory and in practice (especially on finding better methods to encode the numbers while keeping the good search times). In particular, it would be interesting to compare it against other compression formats that seem to lie between the simplicity of LZ78 and the compression efficiency of LZ77 [24, 13, 22].

References

[1] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 2nd Data Compression Conference (DCC'92)*, pages 279–288, March 1992.

[2] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *Journal of Computer and System Sciences*, 52(2):299–307, 1996.

[3] R. Baeza-Yates. Text retrieval: Theory and practice. In *12th IFIP World Computer Congress*, volume I, pages 465–476. Elsevier Science, September 1992.

[4] R. Baeza-Yates and G. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, October 1992.

[5] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. 7th Annual Symp. on Combinatorial Pattern Matching (CPM'96)*, LNCS 1075, pages 1–23. Springer-Verlag, 1996.

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

[7] T. Bell and D. Kulp. Longest-match string searching for Ziv-Lempel compression. *Software– Practice and Experience*, 23(7):757–771, July 1993.

[8] J. Bentley, D. Sleator, R. Tarjan, and V. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29:320–330, 1986.

[9] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[10] A. Czumaj, Maxime Crochemore, L. Gasieniec, S. Jarominek, Thierry Lecroq, W. Plandowski, and W. Rytter. Speeding up two string-matching algorithms. *Algorithmica*, 12:247–267, 1994.

[11] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[12] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. In *27th ACM Annual Symposium on the Theory of Computing (STOC'95)*, pages 703–712, 1995.

[13] E. Fiala and D. Greene. Data compression with finite windows. *Communications of the ACM*, 32(4):490–505, 4 1989.

[14] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

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

[16] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. 11th Annual Symp. on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 195–209. Springer-Verlag, 2000.

[17] J. Kärkkäinen and E. Ukkonen. Sparse suffix trees. In *Proc. 2nd Annual International Computing and Combinatorics Conferenc (COCOON'96)*, pages 219–230, 1996. LNCS 1090.

[18] 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.

[19] T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Shift-and approach to pattern matching in LZW compressed text. In *Proc. 10th Annual Symp. on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 1–13. Springer-Verlag, 1999.

[20] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

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

[22] Y. Matias, N. Rajpoot, and S.C. Sahinalp. The effect of flexible parsing for dynamic dictionary based data compression. In *Proc. 9th Data Compression Conference (DCC'99)*, pages 238–246, 1999.

[23] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In *Proc. 7th Intl. Symp. on String Processing and Information Retrieval (SPIRE 2000)*, pages 221–228. IEEE CS Press, 2000.

[24] 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.

[25] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

[26] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. 5th Intl. Symp. on String Processing and Information Retrieval (SPIRE'98)*, pages 90–95. IEEE CS Press, 1998.

[27] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. 21st ACM Intl. Conf. on Research and Development in Information Retrieval (SIGIR'98)*, pages 298–306. York Press, 1998.

[28] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. 9th Annual Symp. on Combinatorial Pattern Matching (CPM'98)*, LNCS 1448, pages 1–13. Springer-Verlag, 1998.

[29] G. Navarro and M. Raffinot. A bit-parallel approach to suffix automata: Fast extended string matching. In *Proc. 9th Annual Symp. on Combinatorial Pattern Matching (CPM'98)*, LNCS 1448, pages 14–33. Springer-Verlag, 1998.

[30] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Annual Symp. on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 14–36. Springer-Verlag, 1999.

[31] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th Annual Symp. on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 166–180. Springer-Verlag, 2000.

[32] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Annual Symp. on Combinatorial Pattern Matching (CPM 2000)*, LNCS 1848, pages 181–194. Springer-Verlag, 2000.

[33] D. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, August 1990.

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

[35] I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–541, 1987.

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

[37] M. Zipstein. Data compression with factor automata. *Theor. Comput. Sci.*, 92(1):213–221, 1992.

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

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