Regular Expression Searching on Compressed Text [*]

Gonzalo Navarro
Dept. of Computer Science, University of Chile.
Blanco Encalada 2120, Santiago, Chile.
gnavarro@dcc.uchile.cl

Abstract

We present a solution to the problem of regular expression searching on compressed text. The format we choose is the Ziv-Lempel family, specifically the LZ78 and LZW variants. Given a text of length u compressed into length n, and a pattern of length m, we report all the R occurrences of the pattern in the text in $O(2^m + mn + Rm \log m)$ worst case time. On average this drops to $O(m^2 + (n + Rm) \log m)$ or $O(m^2 + n + Ru/n)$ for most regular expressions. This is the first nontrivial result for this problem. The experimental results show that our compressed search algorithm needs half the time necessary for decompression plus searching, which is currently the only alternative.

1 Introduction

The need to search for regular expressions arises in many text-based applications, such as text retrieval, text editing and computational biology, to name a few. A *regular expression* is a generalized pattern composed of (i) basic strings, (ii) union, concatenation and Kleene closure of other regular expressions [1]. The problem of regular expression searching is quite old and has received continuous attention since the sixties.

A particularly interesting case of text searching arises when the text is compressed. Text compression [5] exploits the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv-Lempel family [35, 36] is one of the best in practice because of its good compression ratios combined with efficient compression and decompression times. The compressed matching problem consists of searching for a pattern on a compressed text without decompressing it. Its main goal is to search the compressed text faster than the trivial approach of decompressing it and then searching.

This problem is important in practice. Today's textual databases are an excellent example of applications where both problems are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. Surprisingly, these two combined requirements are not easy to achieve together: The only solution before the 90's was to process queries by decompressing the texts and then searching them.

[*]Partially supported by Fondecyt grant 1-020831.

Since then, a lot of research has been conducted on the problem. A wealth of solutions have been proposed to deal with simple, multiple and, very recently, approximate compressed pattern matching. Regular expression searching on compressed text seems to be the last goal that still defies the existence of any nontrivial solution.

This is the problem we solve in this paper: we present the first solution for compressed regular expression searching. The format we choose is the Ziv-Lempel family, focusing in the LZ78 and LZW variants [36, 33]. Given a text of length u compressed into length n, we are able to find the R occurrences of a regular expression of length m in $O(2^m + mn + Rm\log m)$ worst case time, needing $O(2^m + mn)$ space. We also propose two modifications that achieve $O(m^2 + (n + Rm)\log m)$ or $O(m^2 + n + Ru/n)$ average case time and, respectively, $O(m + n\log m)$ or $O(m + n)$ space, for "admissible" regular expressions, that is, those whose automaton runs out of active states after reading $O(1)$ text characters, on average. These results are achieved using bit-parallelism and are valid for short enough patterns, otherwise the search times have to be multiplied by $\lceil m/w \rceil$, where w is the number of bits in the computer word.

We have implemented our algorithm on LZW and compared it against the best existing algorithms on uncompressed text, showing that we can search the compressed text twice as fast as the naïve approach of decompressing and then searching.

A preliminary version of this paper appeared in [22].

2 Basic Concepts

2.1 Strings, Regular Expressions and Automata

We give a very basic introduction to the subject. For more details see, for example, [1].

Given an alphabet (finite set of symbols) Σ of size σ, a *string* is a sequence of elements of Σ, called *characters*. The length of a string S is denoted $|S|$, and the unique string of length zero is denoted ε. Given a string S we use $S_{i...j}$ to denote the string obtained by taking from the i-th to the j-th characters of S. The first position of a string is 1.

A *language* is a finite or infinite set of strings. In particular, the language Σ^* denotes the set of all the strings over alphabet Σ.

A *regular expression* is a string on the set of symbols $\Sigma \cup \{\ \varepsilon,\ |\ ,\ \cdot\ , *\ , (,\ )\ \}$, which is recursively defined as a simple string on Σ^*, $(\ E_1\ )$, $(E_1 \cdot E_2)$, $(E_1 \mid E_2)$, and $(E_1 *)$, where E_1 and E_2 are in turn regular expressions. By the *length* of a regular expression we mean the total number of elements of Σ it contains, disregarding the other operators.

A regular expression E denotes a language $L(E)$ as follows. Simple strings denote a singleton formed by that string; $L((E_1)) = L(E)$; $L(E_1 \cdot E_2) = L(E_1) \cdot L(E_2)$ (i.e., any string formed by concatenating a string in $L(E_1)$ with a string in $L(E_2)$); $L(E_1 \mid E_2) = L(E_1) \cup L(E_2)$; and $L(E_1 *) = \bigcup_{i \geq 0} L(E_1)^i$, where $L^0 = \{\varepsilon\}$ and $L^{i+1} = L \cdot L^i$. This last operation is called the Kleene closure.

An *automaton* is a graph where the arrows are labeled with elements of $\Sigma \cup \{\varepsilon\}$. The latter are called ε-transitions. One state is called *initial* and zero or more states are called *final*. An automaton *recognizes* a string $x \in \Sigma^*$ if there is a path from the initial to a final state such that the concatenation of the labels of the arrows traversed is x. The language recognized by an automaton is the set of strings it recognizes.

Given a string x, we say that a given state i of the automaton is *active* after reading x if x labels a path from the initial state to state i. An automaton is a *deterministic* finite automaton (DFA) if no more than one state can be active for a given string x. Otherwise it is a *nondeterministic* finite automaton (NFA).

There is a standard way to convert an NFA to a DFA that recognizes the same language. If the NFA has m states, the DFA may have up to 2^m states. Basically, every combination of active/inactive NFA states becomes a single DFA state.

Given a regular expression E, there are several techniques to produce an NFA that recognizes $L(E)$. The most classical is Thompson's [30]. Given an expression of length m, this method produces an NFA of at most $2m$ states and $4m$ edges. A less popular one is Glushkov's [9], which produces an NFA of exactly $m+1$ states but $O(m^2)$ edges. To fix ideas we will assume in this paper that we build NFAs using the version of Glushkov's algorithm popularized by Berry and Sethi [6].

The problem of searching for a regular expression E in a given text string T is that of finding all the text substrings that belong to $L(E)$. These are called *occurrences*. For simplicity, we report the text positions where the occurrences finish in the text, that is, $\{|xy|,\ T = xyz,\ y \in L(E)\}$. Those positions are called *matches*.

In order to use an automaton for text searching, we add a self-loop at the initial state, which can be followed by any character. Hence the initial state is always active. We feed the automaton with the characters of the text, and every time it reaches a final state we report a match. Since the initial state is always active, it detects occurrences starting anywhere in the text.

2.2 Bit-Parallelism and Bit Masks

Bit-parallelism is a technique to code many elements in the bits of a single computer word and manage to update all them in a single operation.

A bit mask is just a sequence of bits stored in one or several contiguous computer words. The number of bits of the computer word is w. Bit masks are used in this paper to represent sets of NFA states, so they will hold $m+1$ bits and hence will need $\lceil (m+1)/w \rceil$ computer words to be represented. Several set operations can be done via classical arithmetical and logical operations, in constant time when the bit masks fit in a single computer word (and in time $O(m/w)$ otherwise). Some of them are $A \cup B$, $A \cap B$, $\overline{A}$ (complement), $A = B$ (equality test), $A \leftarrow B$ (copy), $a \in A$. Another operation we will need to perform in constant time is to select any element of a set. This can be achieved by "bit magic", which means precomputing the table storing the position of, say, the highest bit for each possible bit mask of length $m+1$. This table needs $O(2^m)$ space.

For clarity we will write the bit masks as sets of states instead of sequences of bits, and their operations as set operations.

2.3 The Ziv-Lempel Compression Formats LZ78 and LZW

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 [5]. We are particularly interested in the LZ78/LZW format, which we describe in depth.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [36]) is based on a dictionary of blocks, to 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 $T_{1\ldots j}$ of T has been already compressed in a sequence of blocks $Z = b_1 \ldots b_r$, all them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1\ldots u}$ that is a block of the dictionary. Once we have 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 $Z = b_1 \ldots b_r b_{r+1}$, 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. This is called the *Ziv-Lempel trie*. Every block corresponds to a node in this trie.

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

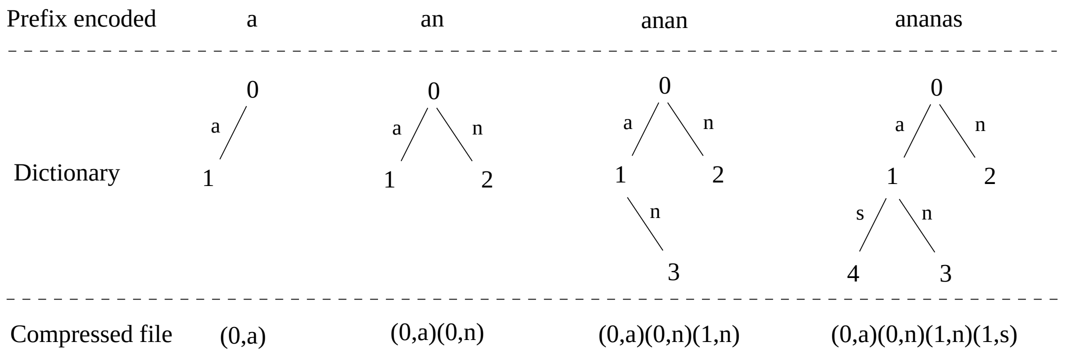


Figure 1: Compression of the string *ananas* with the algorithm LZ78.

So the compressed text is a sequence of block descriptions $Z = b_1 b_2 \ldots b_n$. Each block b_r represents a substring B_r of T, such that $B_1 \ldots B_n = T$. Moreover, each block $b_r = (b_s, a)$ is formed by the concatenation of a previously seen block b_s and an explicit letter a. We say that b_r *references* b_s, and will write $ref(b_r) = b_s$. The *referencing chain* that starts in b_r is defined as the sequence b_r, $ref(b_r)$, $ref(ref(b_r)$, $\ldots$, b_0. In this sequence, the lengths of the blocks decrease by 1 until they reach zero, which corresponds to the empty string represented by the block b_0.

In terms of the Ziv-Lempel trie, block b_0 corresponds to the root and every node references its parent. Following a referencing chain is equivalent to following a path towards the root.

The LZ78 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 for 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 block b_r is read at the r-th step of the algorithm), although this time an array implementation is preferable to the trie. Compared to LZ77, 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 cope with limited memory for compression. A particularly interesting variant is due to Welch, called LZW [33]. In this case, the extra letter (second element of the pair) is not coded, but implicitly taken as the first letter of the next block (the dictionary is initialized with one block per letter). LZW is used by the Unix program *Compress*.

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.

3 Related Work

3.1 Regular Expression Searching

The traditional technique [30] to search for a regular expression of length m in a text of length u is to convert the expression into an NFA with $O(m)$ nodes, add the self-loop at the initial state, and then search the text using the automaton at $O(mu)$ worst case time. The cost comes from the fact that $O(m)$ states of the NFA may be active at each step, and therefore all may need to be updated. Thompson [30] shows how to perform all the updates in $O(m)$ time.

A more efficient choice [1] is to convert the NFA into a DFA prior to searching. Since the DFA has only one active state at a time, it permits searching the text at $O(u)$ cost, which is worst-case optimal. The cost of this approach is that the DFA may have $O(2^m)$ states, which implies a preprocessing cost and extra space of $O(\sigma 2^m)$ for the table D that, given the current DFA state and text character, delivers the next DFA state.

An easy way to obtain a DFA from an NFA is via bit-parallelism: The vector of active and inactive states is stored as a bit mask. Instead of (ala Thompson) examining the active states one by one, the whole computer word is used to index a table that, given the current text character, provides the new set of active states (another computer word). This can be considered either as a bit-parallel simulation of an NFA, or as an implementation of a DFA (where the identifier of each deterministic state is the bit mask as a whole). This idea has been used several times, under Thompson's [34] and Glushkov's [27] constructions.

By using different properties of the constructions, both manage to implement the transition function D using $O(2^m)$ space (actually, the Thompson-based version [34] may need $O(2^{2m})$ states). In both cases, if the table is too big, it can be horizontally split into two or more tables [34]. For example, a table of size 2^m can be split into 2 subtables of size $2^{m/2}$. We need to access two tables for a transition but need only the square root of the space.

Some techniques have been proposed to obtain a tradeoff between NFAs and DFAs. In [19] a four-russians approach is presented that obtains $O(mu/\log u)$ worst-case time and extra space. The idea is to divide the syntax tree of the regular expression into "modules", which are subtrees of a reasonable size. Those subtrees are implemented as DFAs and are thereafter considered as leaf nodes in the syntax tree. The process continues with this reduced tree until a single final module is obtained, so the result is an NFA of DFAs.

The ideas presented up to now aim at a good implementation of the automaton, but they must inspect all the text characters. Other proposals try to skip some text characters, as it is usual for simple pattern matching. For example, in [32] they present an algorithm that determines the minimum length of a string matching the regular expression and forms a trie with all the prefixes of that length of strings matching the regular expression. A multipattern search algorithm like Commentz-Walter [7] is run over those prefixes as a filter to detect text areas where a complete occurrence may start. Those areas are then verified with a classical algorithm. Another technique of this kind is used in *Gnu Grep*, which extracts a set of strings that must appear in any occurrence. These strings are searched for and the areas where they appear are checked for complete occurrences using a lazy deterministic automaton (i.e., built on the fly).

The most recent development, also in this line, is [24]. They invert the arrows of the DFA and make all states initial and the initial state final. The result is an automaton that recognizes all the reverse prefixes of strings matching the regular expression. The idea is in this sense similar to that of [32], but takes less space. The search method is also different: instead of a Boyer-Moore like algorithm, it is based on BNDM [26].

3.2 Compressed Pattern Matching

The *compressed matching problem* was first defined in the work of Amir and Benson [2] 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 naïve 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 $R = u > n$).

Two different approaches exist to search compressed text. The first one is rather practical. Efficient solutions based on Huffman coding [10] on words have been presented by Moura et al. [18], but they need the text to contain natural language and be large (say, 10 Mb or more). Moreover, they allow only searching for whole words and phrases. There are also other practical ad-hoc methods [15], but the compression they obtain is poor. Moreover, in these compression formats $n = \Theta(u)$, so the speedups can only be measured in practical terms.

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 [35] is able to reference any substring of the text already processed, while LZ78 [36] and LZW [33] reference only a single previous reference plus a new letter that is added.

String matching in Ziv-Lempel compressed texts is much more complex, since the pattern can appear in different forms across the compressed text. The first algorithm for exact searching is from 1994, by Amir, Benson and Farach [3], who search LZ78 compressed texts needing time and space $O(m^2 + n)$.

The only search technique for LZ77 is by Farach and Thorup [8], a randomized algorithm to determine in time $O(m + n \log^2(u/n))$ whether a pattern is present or not in the text.

An extension of the first work [3] to multipattern searching was presented by Kida et al. [13], together with the first experimental results in this area. They achieve $O(m^2 + n)$ time and space, although this time m is the total length of all the patterns.

New practical results were presented by Navarro and Raffinot [25], who proposed a general scheme to search Ziv-Lempel compressed texts (simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and a new variant proposed that was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented by Kida et al. [14]. The same group generalized the existing algorithms and nicely unified the concepts in a general framework [12]. Recently, Navarro and Tarhio [28] presented a new, faster, algorithm based on Boyer-Moore.

Approximate string matching on compressed text aims at finding the pattern where a limited number of differences between the pattern and its occurrences are permitted. The problem, advocated in 1992 [2], was solved for Huffman coding of words [18], but the solution is limited to search for a whole word and retrieve whole words that are similar. The first true solutions appeared very recently, by Kärkkäinen et al. [11], Matsumoto et al. [16] and Navarro et al. [23].

4 A Search Algorithm

We present now our approach for regular expression searching on a text $Z = b_1 \ldots b_n$, which is expressed by the LZ78 algorithm as a sequence of n *blocks*. Our goal is to find the last positions in T of the regular expression occurrences, using Z instead of T.

Our approach is to modify the DFA algorithm based on bit-parallelism [27], which is 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 characters. Since we assume that Glushkov's construction is used, the NFA has $m+1$ states. So we start by building the DFA in $O(2^m)$ time and space. Recall that the state identifiers of the DFA are exactly the bit masks that represent the corresponding sets of active NFA states.

We assume that the states of our automaton are numbered $0 \ldots m$, being 0 the initial state. We call F the bit mask of final states and the transition function is $D : \mathit{bitmasks} \times \Sigma \rightarrow \mathit{bitmasks}$ (which, as explained, it is implemented using $O(2^m)$ space).

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* is computed and stored, (2) the occurrences ending inside the block B are reported, and (3) the state of the search is updated.

Say that block b represents the text substring B. Then the *description* of b is formed by

- a number $len(b) = |B|$, its length;
- a block number $ref(b)$, the referenced block;
- a vector $tr_{0\ldots m}(b)$ of bit masks, where tr_i gives the states of the NFA that become active after reading B if only the i-th state of the NFA is active at the beginning;
- a vector $mat_{0\ldots m}(b)$ of block numbers, where $mat_i(b)$ gives the first (i.e., longest) block b' in the referencing chain of b such that $tr_i(b') \cap F \neq \emptyset$, or $\perp$ if there is no such block. This is, the first block in the referencing chain where state i produces a match at the end of the block.

The state of the search, in turn, consists of two elements:

- the last text position considered, j (initially 0);
- a bit mask S of $m+1$ bits, that indicates which states are active after processing $T_{1\ldots j}$. Initially, S has only its initial state active, $S = \{0\}$.

As we show next, the total cost to find all the matches with this scheme is $O(2^m+mn+Rm\log m)$ in the worst case. The first term corresponds to building the DFA from the NFA, the second to computing block descriptions and updating the search state, and the last to report the occurrences. The existence problem is solved in time $O(2^m+mn)$. The space requirement is $O(2^m+mn)$. For expressions longer than w, the time is $O((2^m+mn)\lceil m/w\rceil + Rm\log m)$.

4.1 Computing Block Descriptions

We show how to compute the description of a new block $b' = (b, a)$ that represents $B' = Ba$, where B is the string represented by the referenced block b and a is an explicit character. An initial block b_0 represents the string ε, and its description is: $len(b_0) = 0$; $tr_i(b_0) = \{i\}$; $mat_i(b_0) = \perp$. We give now the update formulas for $B' = Ba$.

- $len(b') \leftarrow len(b) + 1$.
- $ref(b') \leftarrow b$.
- $tr_i(b') \leftarrow D(tr_i(b), a)$.
- $mat_i(b') \leftarrow$ if $tr_i(b') \cap F \neq \emptyset$ then b' else $mat_i(b)$.

For each block, we have to update all the cells of tr and mat, so we pay $O(mn)$ time (recall that bit-parallelism permits performing set operations in constant time). The space required for the block descriptions is $O(mn)$ as well.

4.2 Reporting Matches and Updating the Search State

If $mat_i(b) \neq 0$ for some $i \in S$, then there are matches to report inside the new block B'. In fact, there may be more than one match, and $mat_i(b)$ gives us only the last match in the block. The previous one can be obtained by considering $mat_i(ref(mat_i(b')))$, and so on. All the matches produced by state i in B' are obtained in reverse order by considering the sequence $mat_i(b)$, $mat_i(ref(mat_i(b')))$, $\ldots$ until it gives $\perp$. If B' starts at text position j, then we have to report the text positions $j + len(mat_i(b')) - 1$, $j + len(mat_i(ref(mat_i(b')))) - 1$, $\ldots$

However, there may be more than one state in S that produces matches inside B'. For each such i we can retrieve the matches in reverse order, but we have to merge the positions reported by the different states in S. Moreover, the same match may be reported by several states in S. Even $O(m)$ states can participate. A priority queue can be used to obtain each position in $O(\log m)$ time. If there are R occurrences overall, then in the worst case each occurrence can be reported m times (reached from each state), which gives a total cost of $O(Rm\log m)$.

Finally, we update S in $O(m)$ time per block using $S \leftarrow \cup_{i\in S}\, tr_i(b')$.

5 A Faster Algorithm on Average

Although we have presented the best worst-case algorithm we could devise, several improvements can be made to its average case performance.

5.1 Active States

Let us define $act(b)$ as the set of NFA states that, if active at the beginning of block b, yield an active state after processing b. This set will turn out to be helpful in reducing unnecessary work.

To the block description of Section 4 we add a new bit mask:

- a bit mask $act(b) = \cup \{i,\ tr_i(b) \neq \emptyset\}$, which indicates the states of the NFA that may yield any active state after processing b.

For the initial block, this is defined as $act(b_0) = \{0 \ldots m\}$. For subsequent blocks $b' = (b, a)$, $act(b')$ is computed as follows:

- $act(b') \ \leftarrow\ \{i \in act(b),\ tr_i(b') \neq \emptyset\}$.

Note that we need to consider only the states already in $act(b)$ in order to define $act(b')$. This effect extends over other elements that require updating: (i) $tr_i(b')$ needs to be computed only for those $i \in act(b')$, since otherwise we know it is empty; and the set of active states S can be updated using the formula $S \ \leftarrow\ \cup_{i \in S \cap act(b')} tr_i(b')$ since, again, the other $tr_i(b')$ values are empty sets.

Except for mat, all the computation of the block description is proportional to the size of act: $tr_i(b')$, $act(b')$ need only work on the states of $act(b)$, and S only on the states of $act(b')$. In order to extract the active bits of act in constant time we resort to bit magic.

The main point is that, on average, $|act(b)| = O(1)$, that is, the number of states of the automaton that can survive after processing a block is constant. We prove in the Appendix that this holds under very general assumptions and for "admissible" regular expressions (i.e., those whose automata run out of active states after processing $O(1)$ text characters, on average). Admissible regular expressions are those of most interest for searching, as unadmissible ones report too many matches.

Therefore, the average time to compute the block descriptions is $O(n)$ for admissible regular expressions. The exception is the mat vector, which we consider in the next sections.

5.2 Updating the *mat* Vector

We need a mechanism to update mat fast. Note that the number of blocks where $mat_i(b) \neq \perp$ is not necessarily $o(n)$, since once a block b has $mat_i(b) \neq \perp$, the same is true for all its descendants in the Ziv-Lempel trie.

However, it is still true that just $O(1)$ values of $mat(b)$ change in $mat(b')$, where $ref(b') = b$, since mat changes only on those $\{i,\ tr_i(b') \cap F \neq \emptyset\} \subseteq act(b')$, and $|act(b')| = O(1)$.

Hence, we do not represent a new mat vector for each block, but only its differences with respect to the referenced block. This must be done such that (i) the mat vector of the referenced block is not altered, as it may have to be used for other trie descendants; and (ii) we are able to find mat_i fast for any i.

A solution is to represent mat as a complete tree (i.e., perfectly balanced) that will always have $m+1$ nodes and associate the keys $\{0 \ldots m\}$ to their value mat_i. This permits obtaining in $O(\log m)$ time the value mat_i. We start with a complete tree, and later need only to modify the values associated to tree keys, but never add or remove keys (otherwise an AVL would have been a good choice). When a new value has to be associated to a key in the tree of the referenced block in order to obtain the tree of the referencing block, we find the key in the old tree and create of copy of the path from the root to the key. Then we change the value associated to the new node holding the key. Except when the new nodes are involved, the created path points to the same nodes where the old paths points, hence sharing part of the tree. The new root corresponds to the modified tree of the new block. The cost of each such modification is $O(\log m)$. We have to perform this operation $O(1)$ times on average per block, yielding $O(n \log m)$ time.

Figure 2 illustrates the idea. This kind of technique is usual when implementing the logical structure of WORM (write once read many) devices, in order to reflect the modifications of the user on a medium that does not permit alterations.

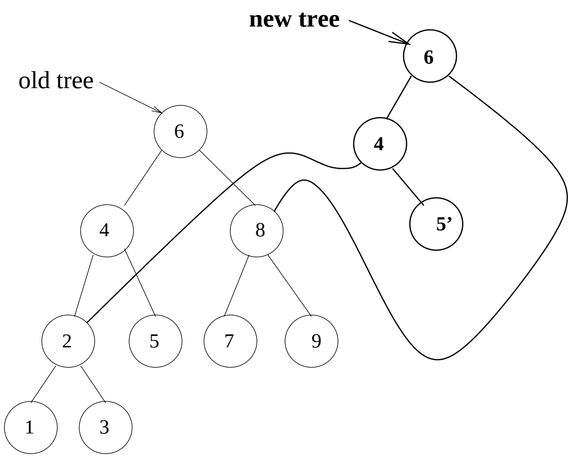


Figure 2: Changing node 5 to 5' in a read-only tree.

5.3 An Alternative for Small R

In most interesting cases the size R of the result is extremely small. This enables us to use a much lighter mechanism to keep track of the matches in exchange for a more costly match reporting procedure.

Instead of representing mat, we store for each block two bit masks $ffin$ and fin. The first indicates which states, if active before processing b, produce a match *exactly* at the end of the block. The second does the same but permits the match anywhere inside the block. In both cases it is necessary to have read at least one character of the block before looking for matches.

In the beginning we have $ffin(b_0) = fin(b_0) = \emptyset$. Given $b' = (b, a)$, we compute $ffin(b')$ and $fin(b')$ as follows:

- $ffin(b') \ \leftarrow \ \{i \in act(b'),\ tr_i(b') \cap F \neq \emptyset\}$.
- $fin(b') \ \leftarrow \ fin(b) \cup ffin(b')$.

Reporting the occurrences is now done as follows. The mask $fin(b')$ tells us whether there are any occurrences to report depending on the active states at the beginning of the block. Therefore, our first action is to compute $S \cap fin(b')$, which tells us which of the currently active states will produce occurrences inside B'. If this set turns out to be empty, we can skip the process of reporting matches.

If $S \cap fin(b') \neq \emptyset$, then we will have to report matches inside the block. We consider the blocks b in the referencing chain of b'. As long as $fin(b) \cap S \neq \emptyset$, we report block the final position of block b if $ffin(b) \cap S \neq \emptyset$ and go to the referenced block, $b \leftarrow ref(b)$. The final position of block b is $j + len(b) - 1$ if b' starts at text position j.

The mechanism is similar to the one used with mat, but this time we do not have a direct link to the previous block that has a match at the end. We just know that, if $fin(b) \cap S \neq \emptyset$, then there are still more matches to report, and that, in particular, if $ffin(b) \cap S \neq \emptyset$, we have to report b. We have to traverse the referencing chain one block by one.

On average, we can consider that every match reported makes us traverse character by character a constant fraction of its block. In exchange, we need $O(1)$ time per block to compute $ffin$. This gives $O(n + Ru/n)$ search time, instead of the previous $O((n + Rm)\log m)$ (preprocessing costs excluded).

5.4 Lowering Space and Preprocessing Costs

In the Appendix we also show that $|tr_i(b)| = O(1)$ on average for admissible regular expressions. This shows another possible improvement.

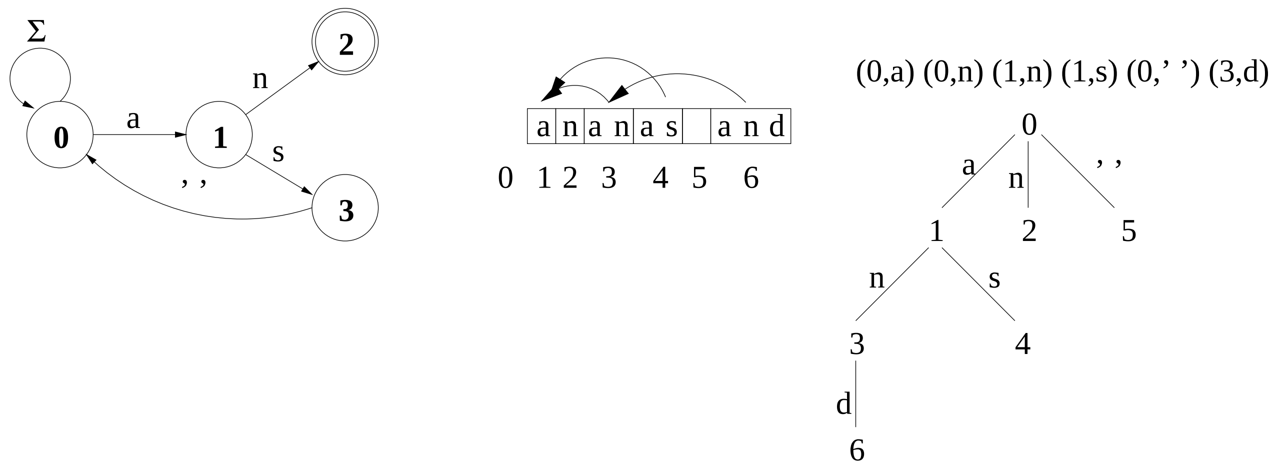
We have chosen a DFA representation of our automaton that needs $O(2^m)$ space and preprocessing time. Instead, an NFA representation would require $O(m^2)$. The problem with the NFA is that, in order to build $tr_i(b')$ for $b' = (b, a)$, we need to make the union of the NFA states reachable via character a from each state in $tr(b)$. This has a worst case of $O(m)$, yielding $O(m^2)$ worst case search time to update a block. However, on average this drops to $O(1)$ since only $O(1)$ states i have $tr_i(b) \neq \emptyset$ (because $|act(b)| = O(1)$) and each such $tr_i(b)$ has constant size.

In particular, we do not need to represent act as a bit mask but can simply enumerate its states. This removes the need of the exponential space for the bit magic.

Therefore, we have obtained average complexity $O(m^2 + (n + Rm)\log m)$ or $O(m^2 + n + Ru/n)$, depending on whether we use mat or $ffin/fin$ to handle the matches. The space requirements are lowered as well. The NFA requires only $O(m)$ space. The block descriptions take $O(n)$ space because there are only $O(1)$ nonempty tr_i masks. With respect to the mat trees, we have that there are on average $O(1)$ modifications per block and each creates $O(\log m)$ new nodes, so the space required for mat is on average $O(n \log m)$. Hence the total space is $O(m + n\log m)$ if we use mat, and $O(m + n)$ if we use $ffin/fin$.

6 An Example

In this section we show a toy example to illustrate how the algorithm works. We search for a regular expression in the text `"ananas and"`. Figure 3 shows the NFA with the corresponding state numbers, the LZ78 blocks of the text, the LZ78 trie, and the evolution of the variables along the search process. We show all the variables mentioned, although in no version of the algorithm all them are used simultaneously.



$0 = \varepsilon$	$1 = (0,a) = a$	$2 = (0,n) = n$	$3 = (1,n) = an$	$4 = (1,s) = as$	$5 = (0,' ') =' '$	$6 = (3,d) = and$
$len(\varepsilon) = 0$	$len(a) = 1$	$len(n) = 1$	$len(an) = 2$	$len(as) = 2$	$len(' ') = 0$	$len(and) = 3$
$ref(\varepsilon) =\perp$	$ref(a) = 0$	$ref(n) = 0$	$ref(an) = 1$	$ref(as) = 1$	$ref(' ') = 1$	$ref(and) = 3$
$tr_0(\varepsilon) = \{0\}$ $tr_1(\varepsilon) = \{1\}$ $tr_2(\varepsilon) = \{2\}$ $tr_3(\varepsilon) = \{3\}$	$tr_0(a) = \{0,1\}$ $tr_1(a) = \emptyset$ $tr_2(a) = \emptyset$ $tr_3(a) = \emptyset$	$tr_0(n) = \{0\}$ $tr_1(n) = \{2\}$ $tr_2(n) = \emptyset$ $tr_3(n) = \emptyset$	$tr_0(an) = \{0,2\}$ $tr_1(an) = \emptyset$ $tr_2(an) = \emptyset$ $tr_3(an) = \emptyset$	$tr_0(as) = \{0,3\}$ $tr_1(as) = \emptyset$ $tr_2(as) = \emptyset$ $tr_3(as) = \emptyset$	$tr_0(' ') = \{0\}$ $tr_1(' ') = \emptyset$ $tr_2(' ') = \emptyset$ $tr_3(' ') = \{0\}$	$tr_0(and) = \{0\}$ $tr_1(and) = \emptyset$ $tr_2(and) = \emptyset$ $tr_3(and) = \emptyset$
$mat_0(\varepsilon) =\perp$ $mat_1(\varepsilon) =\perp$ $mat_2(\varepsilon) =\perp$ $mat_3(\varepsilon) =\perp$	$mat_0(a) =\perp$ $mat_1(a) =\perp$ $mat_2(a) =\perp$ $mat_3(a) =\perp$	$mat_0(n) =\perp$ $mat_1(n) = 2$ $mat_2(n) =\perp$ $mat_3(n) =\perp$	$mat_0(an) = 3$ $mat_1(an) =\perp$ $mat_2(an) =\perp$ $mat_3(an) =\perp$	$mat_0(as) =\perp$ $mat_1(as) =\perp$ $mat_2(as) =\perp$ $mat_3(as) =\perp$	$mat_0(' ') =\perp$ $mat_1(' ') =\perp$ $mat_2(' ') =\perp$ $mat_3(' ') =\perp$	$mat_0(and) = 3$ $mat_1(and) =\perp$ $mat_2(and) =\perp$ $mat_3(and) =\perp$
$act(\varepsilon) = \{0,1,2,3\}$	$act(a) = \{0\}$	$act(n) = \{0,1\}$	$act(an) = \{0\}$	$act(as) = \{0\}$	$act(' ') = \{0,3\}$	$act(and) = \{0\}$
$fin(\varepsilon) = \emptyset$	$fin(a) = \emptyset$	$fin(n) = \{1\}$	$fin(an) = \{0\}$	$fin(as) = \emptyset$	$fin(' ') = \emptyset$	$fin(and) = \{0\}$
$ffin(\varepsilon) = \emptyset$	$ffin(a) = \emptyset$	$ffin(n) = \{1\}$	$ffin(an) = \{0\}$	$ffin(as) = \emptyset$	$ffin(' ') = \emptyset$	$ffin(and) = \emptyset$
$j = 0$	$j = 1$	$j = 2$	$j = 4$	$j = 6$	$j = 7$	$j = 10$
$S = \{0\}$	$S = \{0,1\}$	$S = \{0,2\}$	$S = \{0,2\}$	$S = \{0,3\}$	$S = \{0\}$	$S = \{0\}$
		Report at 1	Report at 3			Report at 9

Figure 3: An example of the search process.

7 Approximate String Matching

Approximate string matching on compressed text aims at finding a pattern string of length m in the text where a limited number, $0 < k < m$, of differences between the pattern and its occurrences are permitted. A "difference" is a character insertion, deletion, or substitution.

The first true solutions to compressed approximate string matching appeared very recently [11, 16, 23]. We disregard the latter because it is an engineering solution without a complexity analysis. The first solution [11] gives worst-case time $O(mkn + R)$ and average case time $O(k^2n + \min(mkn, m^2(m\sigma)^k) + R)$, where σ is the alphabet size. The second solution [16] gives $O(mk^3n/w)$ worst case time for the existence problem.

It is interesting to notice that any solution for compressed regular expression searching implies a solution for compressed approximate string matching, as the latter can be expressed as the output of an automaton [20]. Consider the NFA for $k = 2$ differences shown in Figure 4. Every row denotes the number of differences seen (the first row zero, the second row one, etc.). Every column represents matching a pattern prefix. Horizontal arrows represent matching a character (i.e., if the pattern and text characters match, we advance in the pattern and in the text). All the others increment the number of differences (move to the next row): vertical arrows insert a character in the pattern (we advance in the text but not in the pattern), solid diagonal arrows substitute a character (we advance in the text and pattern), and dashed diagonal arrows delete a character of the pattern (they are ε-transitions, since we advance in the pattern without advancing in the text). The initial self-loop allows a match to start anywhere in the text. The automaton signals (the end of) a match whenever a rightmost state is active.

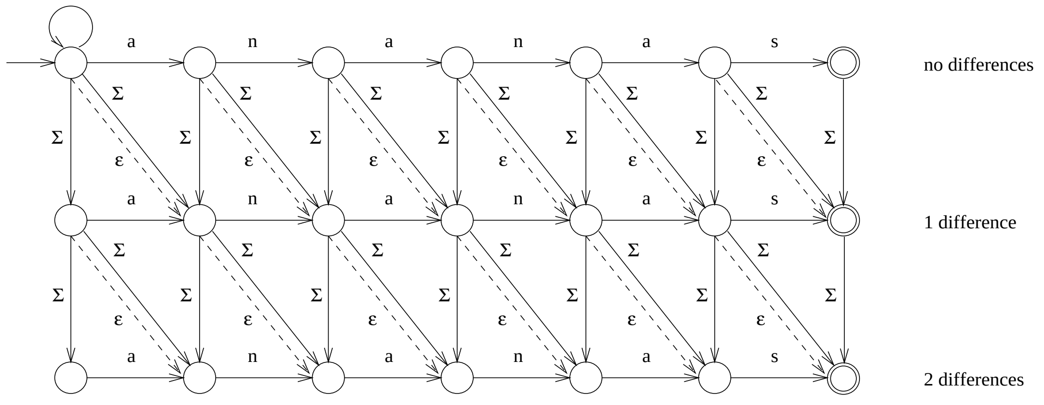


Figure 4: An NFA for approximate string matching of the pattern "ananas" with two differences.

We now consider which would be the behavior of our present algorithm over the automaton for approximate string matching.

Let us forget for a second that the top left state has a self-loop. Except for this arrow, the automaton is cycle-free. Take any state (i, j), at row $i \in 0 \dots k$ and column $j \in 0 \dots m$. It cannot activate any state after processing a string longer than $(k - i) + (m - j)$. Let us pessimistically assume that it will activate states after processing every string of up to that length. This means that state (i, j) will be part of *act* for $O(\sigma^{(k-i)+(m-j)})$ different blocks. Adding up over every (i, j)

gives us $O(\sigma^{m+k})$, which is the total size of *act* over all the possible blocks.

State $(0,0)$, on the other hand, is part of *act* in every block. Therefore, the total size of the set *act* over all the n blocks is upper bounded by $O(\sigma^{m+k}+n)$.

To determine the cost of an update operation we must consider how we implement the $D()$ function. A naïve deterministic table implementation requires $O(2^{mk})$ space. We prefer a bit-parallel technique that computes $D()$, in $O(mk/w)$ time [34]. No exponential preprocessing or storage are involved.

Therefore the existence problem can be solved using *act* in worst case time $O((\sigma^{m+k}+n)(mk/w))$. This compares favorably against the $O(mk^3n/w)$ complexity of [16] for $m+k \leq \log_\sigma n$, that is, when the text is long enough compared to the pattern.

Let us now focus on how to report all the matches. Every occurrence can be reported mk times. However, many of those are redundant: It is not hard to show that, if a given state (i,j) is active, then all the states $(i+r,j)$ are active too, and any occurrence produced by state $(i+r,j)$ is also produced by state (i,j), for all $r>0$. So we can consider only the highest active states of each column and we will have all the relevant matches. This gives us only $O(m)$ states to consider, and hence the worst case time to report the occurrences is $O(Rm\log m)$.

The problem, however, is how to maintain mat. Using the same analysis as for *act*, the total number of mat cells that change is $O(\sigma^{m+k}+n)$. On blocks shorter than $m+k$ we use the balanced tree mechanism of Section 5.2 and obtain $O(\sigma^{m+k}\log m)$ time. On longer blocks, only state $(0,0)$ can change in mat, what can be tracked in $O(1)$ time and space.

Therefore the total search time is in the worst case $O(\sigma^{m+k}(mk/w+\log m)+nmk/w+Rm\log m)$ if we want to report all the matches. This compares favorably against the $O(mkn+R)$ time of [11]: we are better for $m+k \leq \log_\sigma n$. (We are disregarding the uninteresting case of very large $R=\Omega(kn/\log m)$).

Hence, we are always better when $m+k \leq \log_\sigma n$ in the worst case. Let us now consider the average case.

It is well known [20], that if we activate state (i,j), the effect will last on average for $O(k-i)$ text positions, instead of the worst case of $(k-i)+(m-j)$. If we add up over every (i,j), we get that the total size of *act* is $O(m\sigma^{O(k)}+n)$. Using $ffin$, this gives an average search time of $O((m\sigma^{ck}+n)(mk/w)+Ru/n)$ for some $c>1$, which compares favorably against the $O(k^2n+\min(mkn, m^2(m\sigma)^k)+R)$ time of [11] roughly for the case $ck \leq \log_\sigma n$ and $k/m > 1/w$. This means large enough n (although much less than that required for the worst case) and not very small k/m ratio.

8 Experimental Results

We have implemented our algorithm in order to determine its practical value. We chose to use the LZW format by modifying the code of Unix's *uncompress*, so our code is able to search files compressed with *compress* (.Z). This implies some small changes in the design, but the algorithm is essentially the same. We have used bit-parallelism [27] with a single table (no horizontal partitioning). Finally, we have chosen to use the $ffin/fin$ masks instead of representing mat.

We ran our experiments on an Intel Pentium III machine of 550 MHz and 64 Mb of RAM. We have compressed 10 Mb of Wall Street Journal articles, which gets compressed to 42% of its original size with *compress*. We measure user time, as system time was negligible. Each data point has been obtained by repeating the experiment 10 times, which yielded a relative error below 2% with 95% confidence.

In the absence of other algorithms for compressed regular expression searching, we have compared our algorithm against the naïve approach of decompressing and searching. The text needed 3.58 seconds to be decompressed with *uncompress*. After decompression, we run two different search algorithms. A first one, *DFA*, uses a bit-parallel DFA to process the text [27]. This is interesting because it is the algorithm we are modifying to work on compressed text. A second one, the software *nrgrep* [21], uses a character skipping technique for searching [24, 27], which is much faster. In any case, the time to decompress is an order of magnitude higher than that to search the uncompressed text, so the search algorithm used does not significantly affect the results.

A major problem when presenting experiments on regular expressions is that there is not a concept of a "random" regular expression, so it is not possible to search for, say, 1,000 random patterns. Lacking such a good choice, we selected a set of 7 patterns to illustrate different interesting cases. The patterns are given in Table 1, together with some parameters and the obtained search times. We use the normal operators to denote regular expressions plus some extensions, such as `"[a-z]"` = $(a|b|c|...|z)$ and `"."` = all the characters. Note that the 7th pattern is not "admissible" and the search time gets affected (we show the average number of active bits to stress that fact).

No.	Pattern	m	R	$\|act\|$	Ours	Uncompress + Nrgrep	Uncompress + DFA
1	`American|Canadian`	17	1801	1.011	1.81	3.75	3.85
2	`Amer[a-z]*can`	9	1500	1.612	1.79	3.67	3.74
3	`Amer[a-z]*can|Can[a-z]*ian`	16	1801	2.213	2.23	3.73	3.87
4	`Ame(i|(r|i)*)can`	10	1500	1.010	1.62	3.70	3.72
5	`Am[a-z]*ri[a-z]*an`	9	1504	2.196	1.88	3.68	3.72
6	`(Am|Ca)(er|na)(ic|di)an`	15	1801	1.014	1.70	3.70	3.75
7	`Am.*er.*ic.*an`	12	92945	7.091	2.74	3.68	3.74

Table 1: The patterns used on Wall Street Journal articles and the search times in seconds.

As the table shows, we can actually improve over the decompression of the text followed by the application of any search algorithm (indeed, just the decompression takes much more time). In practical terms, we can search the original file at about 4–5 Mb/sec. This is about half the time necessary for decompression plus searching with the best algorithm.

We have used *compress* because it is the format we are dealing with. In some scenarios, LZW is the preferred format because it maximizes compression (e.g., it compresses DNA better than LZ77). However, we may prefer a decompress plus search approach under the LZ77 format, which decompresses faster. For example, Gnu *gzip* needs 2.07 seconds for decompression in our machine. If we compare our search algorithm on LZW against decompressing on LZ77 plus searching, we are still 20% faster.

9 Conclusions

We have presented the first solution to the open problem of regular expression searching on Ziv-Lempel compressed text. Our algorithm can find the R occurrences of a regular expression of length m over a text of size u compressed by LZ78 or LZW into size n in $O(2^m + mn + Rm \log m)$ worst-case time and, for most regular expressions, $O(m^2 + (n + Rm) \log m)$ or $O(m^2 + n + Ru/n)$ average case time. This gives also a new competitive algorithm for compressed approximate string matching. We have shown that the algorithm is of practical interest, as we are able to search compressed text twice as fast as decompressing plus searching.

An interesting question is whether we can improve the search time using character skipping techniques [32, 24]. The first would have to be combined with multipattern search techniques on LZ78/LZW [13]. For the second type of search (BNDM [24]), there is no existing algorithm on compressed text yet. We are also pursuing on extending these ideas to other compression formats, such as a Ziv-Lempel variant where the new block is the concatenation of the previous and the current one [17]. The existence problem seems to require $O(m^2 n)$ time for this format.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, 1986.

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

[3] A. Amir, G. Benson, and M. Farach. Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Computer and Systems Sciences*, 52(2):299–307, 1996. Earlier version in *Proc. SODA'94*.

[4] R. Baeza-Yates and G. Gonnet. Fast text searching for regular expressions or automaton searching on a trie. *J. of the ACM*, 43(6):915–936, 1996.

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

[6] G. Berry and R. Sethi. From regular expression to deterministic automata. *Theoretical Computer Science*, 48(1):117–126, 1986.

[7] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. Int. Colloquium on Automata, Languages and Programming (ICALP'79)*, LNCS 6, pages 118–132, 1979.

[8] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.

[9] V-M. Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16:1–53, 1961.

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

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

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

[13] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. 9th Data Compression Conference (DCC'98)*, pages 103–112, 1998.

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

[15] 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.

[16] 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. Symposium on String Processing and Information Retrieval (SPIRE'2000)*, pages 221–228. IEEE CS Press, 2000.

[17] 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.

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

[19] G. Myers. A Four-Russian algorithm for regular expression pattern matching. *J. of the ACM*, 39(2):430–448, 1992.

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

[21] G. Navarro. NR-grep: a fast and flexible pattern matching tool. *Software Practice and Experience (SPE)*, 31:1265–1312, 2001.

[22] G. Navarro. Regular expression searching over Ziv-Lempel compressed text. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching (CPM'2001)*, LNCS 2089, pages 1–17, 2001.

[23] G. Navarro, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Faster approximate string matching over compressed text. In *Proc. 11th IEEE Data Compression Conference (DCC'01)*, pages 459–468, 2001.

[24] G. Navarro and M. Raffinot. Fast regular expression search. In *Proc. 3rd Workshop on Algorithm Engineering (WAE'99)*, LNCS 1668, pages 198–212, 1999.

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

[26] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000. `http://www.jea.acm.org/2000/NavarroString`.

[27] G. Navarro and M. Raffinot. Compact DFA representation for fast regular expression search. In *Proc. 5th Workshop on Algorithm Engineering (WAE'01)*, LNCS 2141, pages 1–12, 2001.

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

[29] R. Sedgewick and P. Flajolet. *Analysis of Algorithms.* Addison-Wesley, 1996.

[30] K. Thompson. Regular expression search algorithm. *Comm. of the ACM*, 11(6):419–422, 1968.

[31] J. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In *Handbook of Theoretical Computer Science*, chapter 9. Elsevier Science, 1990.

[32] B. Watson. A new regular grammar pattern matching algorithm. In *Proc. 4th Annual European Symposium on Algorithms (ESA'96)*, pages 364–377, 1996.

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

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

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

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

Appendix: Average Number of Active Bits

The goal of this Appendix is to show that, on average and under certain conditions, $|act(b)|$ and $|tr_i(b)|$ are $O(1)$.

Let us consider the process of generating the LZ78/LZW trie. A string from the text is read and the current trie is followed, until the new string read "falls out" of the trie. At that point we add a new node to the trie and restart reading the text. It is clear that, at least for Bernoulli sources, the resulting trie is the same as the result of inserting n random strings of infinite length.

Let us now consider that we initialize our NFA with just state i active and that we backtrack on the LZ78 trie, entering into all possible branches and feeding the automaton with the corresponding character. We stop when the automaton runs out of active states.

The total amount of trie nodes reached in this process is exactly the amount of text blocks b whose i-th bit in $act(b)$ is active, that is, the blocks such that, if we start with state i active, we finish the block with some active state. Hence the total amount of states in *act* over all the blocks of the text corresponds to the sum of trie nodes reached when starting the NFA initialized with every possible state i.

As shown by Baeza-Yates and Gonnet [4], the cost of backtracking on a trie of n nodes with a regular expression is $O(\text{polylog}(n)n^\lambda)$, where $0 \leq \lambda < 1$ depends on the structure of the regular expression. This result applies only to random tries over a uniformly distributed alphabet and for an arbitrary regular expression that has no outgoing edges from final states. We remark that the character probabilities on the LZ78 trie are more uniform than on the text, so even on biased text the uniform model is not so bad an approximation. In any case the result can probably be extended to biased cases.

Despite being suggestive, the previous result cannot be immediately applied to our case. First, it is not meaningful to consider such a random text in a compression scenario, since in this case compression would be impossible. Even a scenario where the text follows a biased Bernoulli or Markov model can be restrictive. Second, our DFAs can perfectly have outgoing transitions from the final states. On the other hand, we cannot afford an arbitrary text and pattern simultaneously because it will be always possible to design a text tailored to the pattern that reaches the worst case. Hence, we consider the most general scenario that we consider reasonable to face:

Definition. Our *arbitrariness assumption* states that text and pattern are arbitrary but independent, in the sense that there is zero correlation between text substrings and substrings of strings generated by the regular expression. Formally, if T is the text and E the regular expression, then for any string x,

$$Pr(x = T_{i...i+|x|-1} \;/\; \exists y, z,\; yxz \in L(E)) \quad = \quad Pr(x = T_{i...i+|x|-1}) \qquad \square$$

The arbitrariness assumption permits us extending our analysis to any text and pattern, under the condition that the text cannot be especially designed for the pattern. Our second step is to set a reasonable condition over the pattern. The number of strings of length ℓ accepted by an automaton is [31]

$$N(\ell) \quad = \quad \sum_j \pi_j \omega_j^\ell \quad = \quad O(c^\ell)$$

where the sum is finitary and π_j and ω_j are constants. The result is simple to obtain with generating functions [29]: For each state i the function $f_i(z)$ counts the number of strings of each length that can be generated from state i of the DFA, so if edges labeled $a_1 \ldots a_k$ reach states $i_1 \ldots i_k$ from i we have $f_i(z) = z(f_{i_1}(z) + \ldots + f_{i_k}(z) + 1 \cdot [i \text{ final}])$, which leads to a system of equations formed by polynomials and possibly fractions of the form $1/(1-z)$. The solution to the system is a rational function, that is, a quotient between polynomials $P(z)/Q(z)$, which corresponds to a sequence of the form $\sum_j \pi_j \omega_j^\ell$. This gives us a tool to define what are admissible states.

Definition. An NFA state i is *admissible* if the number of strings of length ℓ recognized from state i is at most c^ℓ, for some $c < \sigma$, for any $\ell \geq 1$. $\square$

If a state i is admissible and the arbitrariness assumption holds then, if we initialize the NFA with only state i active and feed the NFA with characters from a random text substring, then the automaton runs out of active states after reading $O(1)$ characters. The reason is that the automaton recognizes c^ℓ strings of length ℓ, out of the σ^ℓ possibilities. Since text and pattern are uncorrelated, the probability that the automaton recognizes the selected text substring after ℓ iterations is $O((c/\sigma)^\ell) = O(\alpha^\ell)$, where we have defined $\alpha = c/\sigma < 1$. Hence the expected amount of steps until the automaton runs out of active states is $\sum_{\ell >= 0} \alpha^\ell = 1/(1-\alpha) = O(1)$.

Let us consider a perfectly balanced trie of n nodes obtained from the text, of height $h = \log_\sigma n$. If we start an automaton at the root of the trie, it will reach $O(c^\ell)$ nodes at the trie level ℓ. This means that the total number of nodes traversed is

$$O\left(c^h\right) \quad = \quad O\left(c^{\log_\sigma n}\right) \quad = \quad O\left(n^{\log_\sigma c}\right) \quad = \quad O\left(n^\lambda\right)$$

for $\lambda < 1$. So in this particular case we repeat the result that exists for random tries, which is not surprising. Let us now consider the LZ78 trie of an *arbitrary* text, which has $f(\ell)$ nodes at depth ℓ, where

$$\sum_{\ell=0}^{h} f(\ell) = n \qquad \text{and} \qquad f(0) = 1,\ f(\ell-1) \leq f(\ell) \leq \sigma^\ell$$

By the arbitrariness assumption, those $f(\ell)$ strings cannot have correlation with the pattern, so the traversal of the trie touches $\alpha^\ell f(\ell)$ of those nodes at level ℓ. Therefore the total number of nodes traversed is

$$C \quad = \quad \sum_{\ell=0}^{h} \alpha^\ell f(\ell)$$

Let us now start with an arbitrary trie and try to modify it in order to increase the number of traversed nodes while keeping the same total number of nodes n. Let us move a node from level i to level j. The new cost is $C' = C - \alpha^i + \alpha^j$. Clearly we increase the cost by moving nodes upward. This means that the worst possible trie is the perfectly balanced one, where all nodes are as close to the root as possible. On the other hand, LZ78 tries obtained from texts tend to be quite balanced, so the worst and average case are quite close anyway. As an example of the other extreme, consider a LZ78 trie with maximum unbalancing (e.g., for the text a^u). In this case the total number of nodes traversed is $O(1)$.

So we have that, under the arbitrariness assumption, the total number of trie nodes traversed by an NFA initialized at an admissible state i is $O(n_i^\lambda)$ for some $\lambda_i < 1$. Unadmissible states, on the other hand, reach all the $O(n)$ nodes.

The total number of active states in *act* is the sum, over all the states i of the NFA, of the trie nodes reached when backtracking with the NFA initialized with state i active. This is

$$O\left(n^{\lambda_0} \ + \ n^{\lambda_1} \ + \ \ldots \ + \ n^{\lambda_m}\right)$$

Note that, given the self-loop at state 0, we have $\alpha_0 = 1$, that is, state 0 is unadmissible. Only now we are in position to define which are the regular expressions to which our result applies.

Definition: A regular expression is *admissible* if its Glushkov's NFA has $O(1)$ unadmissible states. $\Box$

If a regular expression is admissible, then only $O(1)$ NFA nodes reach all the trie nodes, while the rest reach only $O(n^\lambda)$, where

$$\lambda \quad = \quad \max\{\lambda_i,\ \lambda_i < 1\}$$

Therefore, in an admissible regular expression the total number of elements in *act* across all the blocks is

$$O\left(n \,+\, mn^\lambda\right) \quad = \quad O(n)$$

where we made the last simplification considering that $m = O(n^{1-\lambda})$, which is weaker than usual assumptions and true in practice. Therefore, we have proved that, under mild restrictions (much more general than the usual randomness assumption), the amortized number of active states in the *act* masks is $O(1)$.

Unadmissible regular expressions are those that basically match all the strings of every length, for example, $a(a|b)^*a$ over the alphabet $\{a,b\}$ matches $2^\ell/4 = \Theta(2^\ell)$ strings of length ℓ. Although we have used Glushkov construction to fix ideas when defining what an admissible regular expression is, being admissible is likely to be independent on how the NFA is produced.

We focus now on the size of the $tr_i(b)$ sets for admissible regular expressions. Let us consider the text substring B corresponding to a block b.

We first consider the $O(1)$ unadmissible states, which are always active, and compute how many admissible states can they activate. At each step, those states may activate $O(\sigma)$ admissible states, but given the arbitrariness assumption, the probability of each such admissible state being active ℓ steps later is $O(\alpha^\ell)$. While processing $B_{1..|B|}$, the unadmissible states are always active, so at the end of the processing we have $\sum_{\ell=0}^{|B|} \sigma\alpha^\ell = O(1)$ active states overall (the term α^ℓ corresponds to the point where we were processing $B_{k-\ell}$).

We consider now that the $O(m)$ admissible states could have been active in the beginning. In this case the probability of yielding an active state after processing B is $O(\alpha^{|B|})$. Hence they totalize $O(m\alpha^{|B|})$ active states. As before, the worst trie is the most balanced one, in which case there are $\sigma^{|B|}$ blocks of lengths 0 to $h = \log_\sigma n$. The total number of active states adds up

$$\sum_{\ell=0}^{h} \sigma^\ell m \alpha^\ell \quad = \quad O(mc^h) \quad = \quad O\left(mn^\lambda\right)$$

Hence, we have in total $O(n + mn^\lambda) = O(n)$ active bits in the tr_i sets, where the n comes from the $O(1)$ states activated from the unadmissible state and the mn^λ from the admissible states.