Regular Expression Searching over Ziv-Lempel Compressed Text

Gonzalo Navarro*

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 $O(m^2 + (n + R) \log m)$ 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 until our days (see Section 2.1).

A particularly interesting case of text searching arises when the text is compressed. Text compression [6] exploits the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv-Lempel family [31, 32] 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 a pattern on a compressed text without uncompressing 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, as the only solution before the 90's was to process queries by uncompressing the texts and then searching into them.

Since then, a lot of research has been conducted on the problem. A wealth of solutions have been proposed (see Section 2.2) to deal with simple, multiple and, very recently, approximate compressed pattern matching. Regular expression searching on compressed text seems to be the last goal which 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 [32, 28]. 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

*Dept. of Computer Science, University of Chile. Blanco Encalada 2120, Santiago, Chile. gnavarro@dcc.uchile.cl. Supported in part by Fondecyt grant 1-990627.

$O(2^m + mn)$ space. We also propose two modifications which achieve $O(m^2 + (n+R)\log m)$ or $O(m^2 + n + Ru/n)$ average case time and, respectively, $O(m + n\log m)$ or $O(m+n)$ space, for "reasonable" regular expressions, i.e. those whose automaton runs out of active states after reading $O(1)$ text characters. 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 naive approach of uncompressing and then searching.

2 Related Work

2.1 Regular Expression Searching

The traditional technique [25] to search a regular expression of length m (which means m letters, not counting the special operators such as "*", "|", etc.) in a text of length u is to convert the expression into a nondeterministic finite automaton (NFA) with $O(m)$ nodes. Then, it is possible to search the text using the automaton at $O(mu)$ worst case time. The cost comes from the fact that more than one state of the NFA may be active at each step, and therefore all may need to be updated.

On top of the basic algorithm for converting a regular expression into an NFA, we have to add a self-loop at the initial state which guarantees that it keeps always active, so it is able to detect a match starting anywhere in the text. At each text position where a final state gets active we signal the end point of an occurrence.

A more efficient choice [1] is to convert the NFA into a deterministic finite automaton (DFA), which has only one active state at a time and therefore allows 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 exponential in m.

An easy way to obtain a DFA from an NFA is via *bit-parallelism*, which 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. In this case, the vector of active and inactive states is stored as the bits of a computer word. Instead of (ala Thompson [25]) examining the active states one by one, the whole computer word is used to index a table which, 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 was first proposed by Wu and Manber [30, 29].

Later, Navarro and Raffinot [22] used a similar procedure, this time using Glushkov's [7] construction of the NFA. This construction has the advantage of producing an automaton of exactly $m+1$ states, while Thompson's may reach $2m$ states. A drawback is that the structure is not so regular and therefore a table $D : 2^{m+1} \times (\sigma+1) \rightarrow 2^{m+1}$ is required, where σ is the size of the pattern alphabet Σ. Thompson's construction, on the other hand, is more regular and only needs a table $D : 2^{2m} \rightarrow 2^{2m}$ for the ε-transitions. It has been shown [22] that Glushkov's construction normally yields faster search time. In any case, if the table is too big it can be split horizontally in

two or more tables [30]. 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 1992, Myers [19] presented a four-russians approach which 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. These 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.

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, Watson [27, chapter 5] presented 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 [8] is run over those prefixes as a filter to detect text areas where a complete occurrence may start. Another technique of this kind is used in *Gnu Grep 2.0*, which extracts a set of strings which must appear in any match. This string is searched for and the neighborhoods of its occurrences are checked for complete matches using a lazy deterministic automaton.

The most recent development, also in this line, is from Navarro and Raffinot [22]. 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 Watson, but takes less space. The search method is also different: instead of a Boyer-Moore like algorithm, it is based on BNDM [22].

2.2 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 naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches (note that it could be that $R = u > n$).

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 that the text contains natural language and is 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 [31] is able to reference any substring of the text already processed, while LZ78 [32] and LZW [28] 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 in LZ78 needing time and space $O(m^2 + n)$.

The only search technique for LZ77 is by Farach and Thorup [9], 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 [23], who proposed a general scheme to search on Ziv-Lempel compressed texts (simple and extended patterns) and specialized it for the particular cases of LZ77, LZ78 and a new variant proposed which was competitive and convenient for search purposes. A similar result, restricted to the LZW format, was independently found and presented 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 [24] 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], had been solved for Huffman coding of words [18], but the solution is limited to search a whole word and retrieve whole words that are similar. The first true solutions appeared very recently, by Kärkkäinen et al. [11] and by Matsumoto et al. [16].

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

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [32]) 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 $T_{1\ldots j}$ 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 $T_{j+1\ldots u}$ which is a block of the dictionary. Once we found this block, say b_s of length ℓ_s, we construct a new block $b_{r+1} = (s, T_{j+\ell_s+1})$, we write the pair at the end of the compressed file Z, i.e $Z = b_1 \ldots b_r b_{r+1}$, 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.

We give as an example the compression of the word *ananas* in Figure 1. 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 $(1, s)$.

The compression algorithm is $O(u)$ time in the worst case and efficient in practice if the dictionary is stored as a trie, which allows rapid searching of the new text prefix (for each character of T we move once in the trie). The decompression needs to build the same dictionary (the pair that

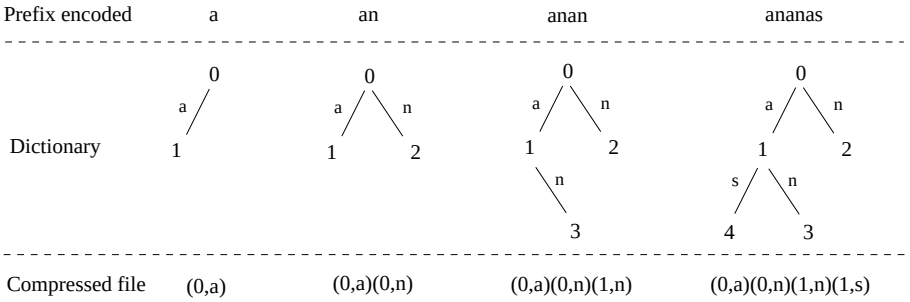

Figure 1: Compression of the word *ananas* with the algorithm LZ78.

defines the block r is read at the r-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.

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 from Welch, called LZW [28]. In this case, the extra letter (second element of the pair) is not coded, but it is taken as the first letter of the next block (the dictionary is started with one block per letter). LZW is used by Unix's *Compress* program.

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

4 A Search Algorithm

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

Our approach is to modify the DFA algorithm based on bit-parallelism, 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 letters. We assume that Glushkov's construction [7] is used, so the NFA has $m+1$ states. So we start by building the DFA in $O(2^m)$ time and space.

Our bit masks will denote sets of NFA states, so they will be of width $m+1$. For clarity we will write the sets of states, keeping in mind that we can compute $A \cup B$, $A \cap B$, A^c, $A = B$, $A \leftarrow B$, $a \in A$ in constant time (or, for long patterns, in $O(\lceil m/w \rceil)$ time, where w is the number of bits in the computer word). Another operation we will need to perform in constant time is to select any element of a set. This can be achieved with "bit magic", which means precomputing the table storing the position of, say, the highest bit for each possible bit mask of length $m+1$, which is not much given that we already store σ such tables.

About our automaton, we assume that the states are numbered $0 \ldots m$, being 0 the initial state. We call F the bit mask of final states and the transition function is $D : bitmasks \times \Sigma \rightarrow bitmasks$.

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 $tr_{0\ldots m}$ of bit masks, where tr_i gives the states of the NFA that remain active after reading B if only the i-th state of the NFA is active at the beginning;
- a bit mask $act = \cup \{i,\ tr_i \neq \emptyset\}$, which indicates which states of the NFA may yield any surviving state after processing B;
- a bit mask fin, which indicates which states, if active before processing B, produce an occurrence inside B (after processing at least one character of B); and
- a vector $mat_{0\ldots m}$ of block numbers, where mat_i gives the most recent (i.e. longest) block b' in the referencing chain $b, ref(b), ref(ref(b)), \ldots$ such that $i \in fin(b')$, or a null value if there is no such block.

The state of the search consists of two elements

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

As we show next, the total cost to search for all the occurrences 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 $O(2^m + mn)$. We recall that patterns longer than the computer word w get their search cost multiplied by $\lceil m/w \rceil$.

4.1 Computing Block Descriptions

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

- $len(b') \leftarrow len(b) + 1$.
- $ref(b') \leftarrow b$.
- $tr_i(b') \leftarrow D(tr_i(b), a)$ (we only need to do this for $i \in act(b)$).

- $act(b') \leftarrow \{i \in act(b),\ tr_i(b') \neq \emptyset\}$.
- $fin(b') \leftarrow fin(b) \cup \{i \in act(b),\ tr_i(b') \cap F \neq \emptyset\}$.
- $mat_i(b') \leftarrow mat_i(b)$ if $tr_i(b') \cap F = \emptyset$, and b' otherwise.

In the worst case 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

The $fin(b')$ mask 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 mask turns out to be null, we can skip the process of reporting matches.

If there are states in the intersection then we will have matches to report inside B'. Now, each state i in the intersection produces a list of positions which can be retrieved in decreasing order using $mat_i(b')$, $mat_i(ref(mat_i(b')))$, ... If B' starts at text position j, then we have to report the text positions $j + len(mat_i(b')) - 1$, $j + len(mat_i(ref(mat_i(b')))) - 1$, ... These positions appear in decreasing order, but we have to merge the decreasing lists of all the states in $S \cap fin(b')$. 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 with $S \leftarrow \cup_{i \in S \cap act(b')} tr_i(b')$.

5 A Faster Algorithm on Average

An average case analysis of our algorithm reveals that, except for mat, all the other operations can be carried out in linear time. This leads to a variation of the algorithm that is linear time on average.

The main point is that, on average, $|act(b)| = |tr_i(b)| = O(1)$, that is, the number of states of the automaton which 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). Note that, thanks to the self loop in the initial state 0, this state is always in $act(b)$ and in $tr_0(b)$.

Except for mat, all the computation of the block description is proportional to the size of act and hence it takes $O(n)$ time: $tr_i(b')$ needs to be computed only for those $i \in act(b)$; and $act(b')$ and $fin(b')$ can also be computed in time proportional to $|act(b)|$. The update to S needs only to consider the states in $act(b')$. Each active bit in act is obtained in constant time by bit magic.

What we need is a mechanism to update mat in constant time per block. Note that it may not be true that $|fin| = O(1)$ on average, because as soon as a state belongs to $fin(b)$, it belongs to all its descendants in the LZ78 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 descendants; and (ii) we are able to quickly find the last block (in the referenced chain) where a given state i produced an occurrence.

A solution is to represent *mat* as a complete tree (i.e. perfectly balanced), which will always have $m+1$ nodes and associates 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 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.

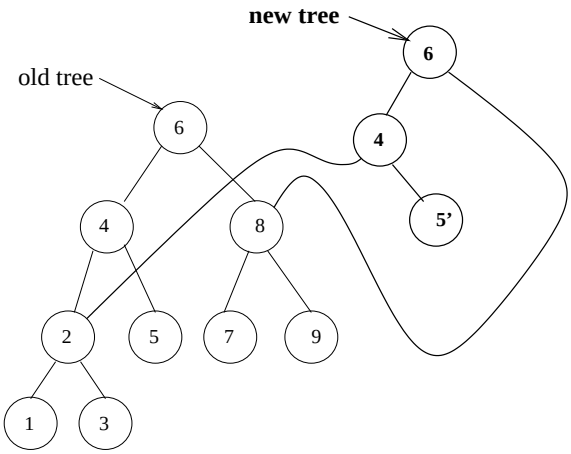

Figure 2: Changing node 5 to 5' in a read-only tree.

We have to add now the cost to report the R matches. Since $|tr_i(b)| = O(1)$ on average, there are only $O(1)$ states able to trigger an occurrence at the end of a block, and hence each occurrence is triggered by $O(1)$ states on average. The priority queue gives us those positions in $O(\log m)$ per position, so the total cost to trigger occurrences is on average $O(R \log m)$.

The fact that $|tr_i(b)| = O(1)$ on average shows another possible improvement. We have chosen a DFA representation of our automaton which 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 the letter 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, 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.

Therefore, we have obtained average complexity $O(m^2 + (n+R)\log m)$. 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 R is really small we may prefer an alternative implementation. Instead of representing *mat*, we store for each block a bit mask $ffin$, which tells whether there is a match exactly at the end of the block. While fin is active we go backward in the referencing chain of the block reporting all those blocks whose $ffin$ mask is active in a state of S. This yields $O(m^2 + n + Ru/n)$ time on average instead of $O(m^2 + (n + R)\log m)$. The space becomes $O(m + n)$.

6 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, with a single table (no horizontal partitioning) and map (at search time) the character set to an integer range representing the different pattern characters, to reduce space. Finally, we have chosen to use the $ffin$ 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 times are negligible. Each data point has been obtained by repeating the experiment 10 times.

In the absence of other algorithms for compressed regular expression searching, we have compared our algorithm against the naive approach of decompressing and searching. The WSJ file 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. 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 [22], which is much faster. In any case, the time to uncompress 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 "random" regular expression, so it is not possible to search, say, 1,000 random patterns. Lacking such good choice, we fixed a set of 7 patterns which were selected 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.

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

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

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

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 compressed 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.

7 Conclusions

We have presented the first solution to the open problem of regular expression searching over 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, $O(m^2 + (n + R) \log m)$ or $O(m^2 + n + Ru/n)$ average case time. We have shown that this is also of practical interest, as we are able to search on compressed text twice as fast as decompressing plus searching.

An interesting question is whether we can improve the search time using character skipping techniques [27, 22]. The first would have to be combined with multipattern search techniques on LZ78/LZW [13]. For the second type of search (BNDM [22]), there is no existing algorithm on compressed text yet. We are also pursuing on extending these ideas to other compression formats, e.g. 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, 1985.

[2] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. 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 Comp. and Sys. Sciences*, 52(2):299–307, 1996. Earlier version in *Proc. SODA'94*.

[4] R. Baeza-Yates. *Efficient Text Searching*. PhD thesis, Dept. of Computer Science, Univ. of Waterloo, May 1989. Also as Research Report CS-89-17.

[5] 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.

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

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

[8] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. ICALP'79*, LNCS v. 6, pages 118–132, 1979.

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

[10] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the 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. 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. 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. 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. 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. 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*, 2000. To appear.

[21] G. Navarro. Nr-grep: A fast and flexible pattern matching tool. Technical Report TR/DCC-2000-3, Dept. of Computer Science, Univ. of Chile, August 2000.

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

[23] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. CPM'99*, LNCS 1645, pages 14–36, 1999.

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

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

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

[27] B. Watson. *Taxonomies and Toolkits of Regular Language Algorithms.* Phd. dissertation, Eindhoven University of Technology, The Netherlands, 1995.

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

[29] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. of USENIX Technical Conference*, pages 153–162, 1992.

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

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

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

Appendix: Average Number of Active Bits

The goal of this Appendix is to show that, on average, $|act(b)| = |tr_i(b)| = O(1)$. In this section σ denotes the size of the text alphabet.

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 initializing our NFA with just state i active. Now, we backtrack on the LZ78 trie, entering into all possible branches and feeding the automaton with the corresponding letter. We stop when the automaton runs out of active states.

The total amount of trie nodes touched in this process is exactly the amount of text blocks b whose i-th bit in $act(b)$ is active, i.e. 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 touched when starting the NFA initialized with each possible state i.

As shown by Baeza-Yates and Gonnet [4, 5], 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 which has no outgoing edges from final states. We remark that the letter probabilities on the trie are more uniform than on the text, so even on biased text the uniform model is not so bad 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 (the previous result is relevant because as soon as a final state is reached they report the whole subtrie). On the other hand, we cannot afford an arbitrary text and pattern simultaneously because it will always be possible to design a text tailored to the pattern that yields a low efficiency. Hence, we consider the most general scenario which is reasonable to face:

Definition: 1 *Our* arbitrariness assumption *states that text and pattern are arbitrary but independent, in the sense that there is zero correlation between text substrings and strings generated by the regular expression.*

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 that are accepted by an automaton is [26]

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

where the sum is finitary and π_j and ω_j are constants. The result is simple to obtain with generating functions: 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, i.e. a quotient between polynomials $P(z)/Q(z)$, which corresponds to a sequence of the form $\sum_j \pi_j \omega_j^{\ell}$. We are ready now to establish our condition over the admissible regular expressions.

Definition: 2 *A regular expression is* admissible *if the number of strings of length ℓ that it generates is at most c^ℓ, where $c < \sigma$, for any $\ell = \omega(1)$.*

Unadmissible regular expressions are those which basically match all the strings of every length, e.g. $a(a|b)^*a$ over the alphabet $\{a, b\}$, which matches $2^\ell/4 = \Theta(2^\ell)$ strings of length ℓ. However, there are other cases. For example, pattern matching allowing k errors can be modeled as a regular expression which matches every string for $\ell = O(k)$ [20]. As we see shortly, we can handle some unadmissible regular expressions anyway.

If a regular expression is admissible and the arbitrariness assumption holds, then if we feed it with characters from a random text position the automaton runs out of active states after $O(1)$ iterations. 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. Hence its height is $h = \log_\sigma n$. If we start an automaton at the root of the trie, it will touch $O(c^\ell)$ nodes at the trie level ℓ. This means that the total number of nodes traversed is

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

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

$$\sum_{\ell=0}^{h} f(\ell) = n \quad \wedge \quad f(0) = 1,\ f(\ell-1) \le f(\ell) \le \sigma^\ell$$

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

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

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, 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 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 admissible regular expression is $O(n^\lambda)$ for some $\lambda < 1$. We use now this result for our analysis.

It is clear that if we take our NFA and make state i the initial state, the result corresponds to a regular expression because any NFA can be converted into a regular expression. So the total amount of states in *act* is

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

where λ_i corresponds to taking i as the initial state. We say that a state is admissible if, when that state is considered as the initial state, the regular expression becomes admissible.

Note that, given the self-loop we added at state 0, we have $\alpha_0 = 1$, i.e. state 0 is unadmissible. However, all the other states must be admissible because otherwise the original regular expression would not be admissible. That is, there is a fixed probability p of reaching the unadmissible state and from there the automaton recognizes all the σ^ℓ strings, which gives at least $p\sigma^\ell = \Theta(\sigma^\ell)$.

Hence, calling

$$\lambda \;=\; \max(\alpha_1, \ldots, \alpha_m) \;<\; 1$$

we have that the total number of active states in all the *act* bit masks is

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

where we made the last simplification considering that $m = O(\text{polylog}(n))$, 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 $O(1)$.

Note that we can afford even that the unadmissible states are reachable only from $O(1)$ other states, and the result still holds. For example, if our regular expression is $a(a|b)^*a^m$ we have only $O(1)$ initial states that yield unadmissible expressions, and our result holds. On the other hand, if we have $a^m(a|b)^*a$ then the unadmissible state can be reached from $\Theta(m)$ other states and our result does not hold.

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 initial state, which is always active. How many states can get activated from the initial state? At each step, the initial state may activate $O(\sigma)$ admissible states, but given the arbitrariness assumption, the probability of each such state being active ℓ steps later is $O(\alpha^\ell)$. While processing $B_{1..k}$, the initial state is always active, so at the end of the processing we have $\sum_{\ell=0}^{k} \sigma\alpha^\ell = O(1)$ active states (the term α^ℓ corresponds to the point where we were processing $B_{k-\ell}$).

We consider now the other m admissible states, whose activation vanishes after examining $O(1)$ text positions. In their case the probability of yielding an active state after processing B is $O(\alpha^k)$. Hence they totalize $O(m\alpha^k)$ active states. As before, the worst trie is the most balanced one, in which case there are σ^k blocks of lengths 0 to $h = \log_\sigma n$. The total number of active states totalizes

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

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 initial state and the mn^λ from the other states.