A New Compression Method for Compressed Matching

EXTENDED ABSTRACT

Shmuel T. Klein
Dept. of Math. & CS
Bar Ilan University and
College of Judea and Samaria
Israel
tomi@cs.biu.ac.il

Dana Shapira
Dept. of Math. & CS
Bar Ilan University
Ramat-Gan 52900
Israel
shapird@cs.biu.ac.il

Abstract

A practical adaptive compression algorithm based on LZSS is presented, which is especially constructed to solve the compressed pattern matching problem, i.e., pattern matching directly in a compressed text without decompressing.

1. Introduction

The general approach for looking for a pattern in a file that is stored in its compressed form, is first decompressing and then applying one of the known pattern matching algorithms in the decoded file. In many cases, however, in particular on the Internet, files are stored in their original form, for if they were compressed, the host computer would have to provide memory space for each user in order to store the decoded file. This requirement is not reasonable, as many users can simultaneously quest the same information reservoir which will demand an astronomical quantity of free memory. Another possibility is transferring the compressed files to the personal computer of the user, and then decoding the files. However, we then assume that the user knows the exact location of the information he is looking for; if this is not the case, much unneeded information will be transferred.

There is therefore a need to develop methods for directly searching within a compressed file. For a given text T and pattern P and complementary encoding and decoding functions $\mathcal{E}$ and $\mathcal{D}$, our aim is to search for $\mathcal{E}(P)$ in $\mathcal{E}(T)$, rather than the usual approach which searches for the pattern P in the decompressed text $\mathcal{D}(\mathcal{E}(T))$. But this is not always straightforward, since an instance of $\mathcal{E}(P)$ in the compressed text is not necessarily the encoding of instance of P in the original text T. Consider for example a static Huffman code: if there are three characters x, y and z, such that a proper prefix of length k of $\mathcal{E}(x)$ is also a suffix of $\mathcal{E}(y)$, and the remaining suffix of length $|\mathcal{E}(x)| - k$ of $\mathcal{E}(x)$ is a prefix of $\mathcal{E}(z)$, then in the encoding of an occurrence of the string yz one will detect an occurrence of $\mathcal{E}(x)$, which is a wrong match. For arithmetic coding and for adaptive methods, the situation is even worse, as a given string is not always encoded the same way.

This so-called *compressed matching problem* has been introduced by Amir and Benson [1], and has recently gotten a lot of attention [2, 4, 8, 9, 10, 5, 7]. Some of the suggested solutions are theoretical in nature, some are more practical. We are following here the idea of Manber [7]: instead of trying to adapt some pattern matching algorithm to a given compression scheme, we suggest a new compression method, specially built to allow easy searching in the compressed file, even at the price of reduced compression efficiency.

In the next section we show the difficulty of applying a compressed pattern matching on a file compressed by LZSS [11]. In Section 3 we then present our new algorithm and bring some experimental results in Section 4.

2. Compressed pattern matching on LZSS compressed texts

In order to justify the need for our new method, we start by presenting a pattern matching scheme for files that were compressed by the LZSS algorithm, and then show the difficulty in it. We concentrate here on LZSS, because it is one of the most popular variants of LZ77, and many other schemes build on it, e.g., LZRW1 [13]. LZSS compressed files are also widespread, especially since it was also the basis of Microsoft's DoubleSpace procedure [12].

In LZSS, a text is encoded as a sequence of elements which are either single characters, or pointers to previously occurring strings, encoded as ordered pairs of numbers denoted (*off*, *len*), where *off* is the number of characters from the current location to the previous occurrence of a substring matching the one that starts at the current location, and *len* is the length of the matching string.

For example, if $T =$ `acdeabceabcdeaeab`, then $\mathcal{E}(T) =$ `acdeabc(4,4)(9,3)(7,3)`. If we now look for the pattern $P =$ `bcdeaea`, we see that P's characters appear in T in a different order than in P. The prefix `bc` of P appears in position 6 of $\mathcal{E}(T)$, the sub-pattern `dea` of P in position 3 and the suffix `ea` of P in position 4. Therefore, a compressed pattern matching algorithm has to save pointers to addresses that might be referred to later in the text, but we obviously do not want to keep references to all the scanned sub-string. The following procedure, the details of which will appear elsewhere, may, at first sight, solve the problem.

We keep a table which records, for each character x of the pattern P, the address in T of the *last* occurrence, as well as that of the *last longest* occurrence, of the sub-pattern of P starting with x. For the up to $2|P|-1$ prefixes and suffixes of P, we keep pointers to *all* of their previous occurrences. Referring to the above example, suppose x is the character `c` in position 2 of P, and that our current position in $\mathcal{E}(T)$ is just after the pair (4,4), i.e., the part of T known to us so far consists of the eleven first characters `acdeabceabc`; then the *last* occurrence we are looking for is the single character `c` in position 11 of T, whereas the *last longest* occurrence is the string `cd` in position 2. These tables are updated constantly as we process the encoded string $\mathcal{E}(T)$, and after each step we check whether P has been detected.

Unfortunately, this algorithm does not always find *all* the occurrences of P as wanted. For example, if T = `cabeabecabdfabd`, then $\mathcal{E}(T)$ = `cabe(3,3)(7,3)df (4,3)`. Let us assume that the pattern P is `fabd`. The problem is that when recording the occurrences of the sub-pattern starting with `a` at position 2 of P, the reference to `ab` in position 2 of T will be erased by its re-occurrence in position 5 which is found by means of the pair (3,3); but the pair (7,3) refers to the string starting in position 1 of T and including `ab`, thus we cannot compute the occurrence of `ab` in position 9, nor that in position 13 using the pair (4,3), so we miss the occurrence of P in address 12. Note that both of the prefix and suffix lists do not contain the needed information.

Instead of trying to rectify the situation, we suggest in the next section a solution tackling the problem from another angle: rather than sticking to one of the known compression techniques, we devise a new one, specially adapted to our pattern matching problem.

3. A new compression algorithm

For a given text T, let n be the length of the compressed text $\mathcal{E}(T)$. The compressed matching algorithm which searches for a pattern P of length m runs in time $O(m \cdot n)$ in the worst case, and requires $O(\mathit{Maxoffset})+O(m)+O(|\Sigma|)$ memory storage, where Σ is the alphabet of T, and *Maxoffset* is a constant which limits the offset size in the compressed file. Note that it does not require space for the decompressed text, as the searching process is done on the compressed text itself, which is the great advantage of our algorithm.

3.1 Design of the method

As has been illustrated in the previous section, the problem, when searching directly in an LZSS compressed text, is that when a character string is processed, we have no knowledge of whether it will be referenced later by means of a back-pointer of the form (*off, len*). This leads to the idea of changing the *position* of these pointers, so as to store them close to the strings they replace. Thus contrarily to the LZSS algorithm, where pointers point backwards into the text that has already been processed, the new algorithm uses its pointers to point *forward*, to the text that will be encoded later.

The compressed text is represented by a sequence of items, each of which being either a single character, or a pointer to a string that appears later, encoded as a pair (*off, len*) like in LZSS. Only now *off* is the number of characters from the current location to the *following* occurrence of the longest substring matching the one that starts at the current location, and *len* is the length of the matching string.

In a first attempt, it seems as if this idea can be implemented by simply swapping the pointer with the string it replaces. For example, if the text is `dabcdeabcb`, LZSS would compress it as `dabcde(5,3)b` and the new algorithm as `d(5,3)abcdeb`, where the pair (5,3) means that the following 3 characters occur again 5 characters after the current position. But such an implementation would not be convenient: since our aim is building a compression algorithm adjusted to the pattern matching problem, we want to know immediately what characters are referred to when an (*off, len*) pair is met. We therefore move the pair to immediately *after* the characters it refers to, adjusting the value of *off* to *off* $-$ *len*. This gives, for the above example, `dabc(2,3)deb`, where the pair (2,3) now means that the last 3 characters occur again 2 characters later.

Moreover, we cannot use the LZSS compressed file and simply move each pointer backwards adjusting the offsets, because of overlapping strings. Consider the following example: let T be `abcdeabcddea`, which would be compressed by LZSS as `abcde(5,4)(6,3)`. Using the idea above, the first few items created by the new algorithm would be `abcd(1,4)e`. When we reach the string `dea` in T, we would like to store the pair (3,3) just after the second `a` which appears in position 6 of T, and point to the last three characters `dea`. But that `a` is part of the string `abcd` which occurred earlier in the text and therefore was encoded by means of the pair (1,4). It is thus not clear, where the pair (3,3) should be stored.

One possibility to correct this, is by avoiding overlapping references. Some (*off, len*) pairs will be omitted altogether, others can be redefined by pointing to strings which are not necessarily the longest possible. For the last example, this could yield a compressed string of the form `abcd(2,3)ea(3,3)`. But the loss in compression efficiency could be significant.

As alternative, we suggest replacing, when necessary, the pair (*off, len*) by a triple (*off, len, slide*), where *slide* is the number of characters by which the original (*off, len*) pair should be shifted forwards. Returning to the last example, we would get `abcd(1,4)e(3,3,1)`. The triple (3,3,1) means that the original pair (3,3) should have been placed after the character which follows the character `e` in the reconstructed text T. Therefore, the triple refers to the last 3 characters, counted from the shifted position, i.e., to the characters `dea`; these should be copied 3 characters later, which are also counted from the current shifted position. The deterioration of the compression ratio will in this case be due only to the encoding overhead incurred by the triples, not because of the omission of copy items.

Note, however, that the definition of the triples does not extend to self-references, which should thus be avoided, but removing a self-reference can be done at a penalty which is only logarithmic in its size. For example, if T is a string of 16 `a`'s, it could be encoded by LZSS as `a(1,15)`; when self-reference is prohibited, it could be encoded by LZSS as `aa(2,2)(4,4)(8,8)`, and in our case as `aa(0,2)(0,4)(0,8)`,

Encoding is done as for LZSS, but using a temporary buffer so as to enable the encoder to insert pairs or triples at earlier positions. For decoding, each individual character in the encoded file is transmitted immediately; when a pair (*off, len*) is encountered, the last *len* characters are temporarily copied into a buffer, *off* characters ahead from the current position; in case of a triple (*off, len, slide*), the current position moves temporarily *slide* characters forward, and proceeds then as for the pair. We shall assume that some bounds *Maxoffset* and *Maxlength* exist for the permissible *off* and *len* values, which is true for most practical implementations of LZ algorithms. This allows us to use a circular buffer of size *Maxoffset* + *Maxlength.*

3.2 The compressed matching algorithm

The new compression algorithm ensures that whenever we meet a pointer item, we immediately know if we must remember the characters of the string it refers to. If there is at least one sub-pattern of P among the last *len* characters, the pattern may occur in the later location. This obligates us to remember the characters of the relevant sub-pattern and their locations. In practice we use a circular buffer, named buf, of size *Maxoffset*, which functions as a storage buffer for the relevant subpatterns.

The main loop processes sequentially the items of the compressed file and scans in parallel also the buffer, using a pointer *current* to the current location. The input is assumed to be a compressed file $\mathcal{E}(T) = \mathcal{E}_1\mathcal{E}_2\cdots$, where each $\mathcal{E}_k$ is either an individual character, or a pair or a triple.

A procedure named copy_relevant_char() is used when we meet a pointer item in order to copy only the relevant characters, i.e., characters of P, to their correct position *offset* characters ahead into the buffer. For simplicity, we refer to pairs as triples with *slide* $= 0$, so that the procedure deals only with triples; it copies characters of P within the last *len* characters preceding position *current* + *slide* to position *current* + *slide* + *off*. Even though characters that are not in P are not relevant, we must know their location for the correct calculation of the *offset* component. In order to do this in a useful way, the procedure links the locations of successive relevant characters, which allows us to skip over the non relevant ones. These links are kept in vector *next*. By default, if *next(current)* is empty, no shortcut is available, and *current* can be increased only by 1.

We use a procedure named find_pat(x) for checking whether the pattern P occurs in the text starting at position x. Since this information is needed during a left to right scan of the text, the KMP pattern matching algorithm [6] seems appropriate. To reduce the number of unneeded checkings, the variable *relevant* indicates the index in T of the last character which was a character of the pattern P.

Figure 1 presents the formal pattern matching algorithm. In the main loop, the item $\mathcal{E}_k$ is first checked whether it is a triple or pair; in this case, if relevant characters are referenced, they have to be copied ahead into the buffer. In the next step (2.2), characters which have been previously copied into the current position are processed. Step 2.3 skips over positions in the buffer that are known not to contain relevant characters. At step 2.4, $\mathcal{E}_k$ is known to be an individual character, which is processed only if it appears in P.

Figure 2 shows the buffer which is constructed while running the Compressed Matching Algorithm on the text T = `xbcyxbcabcdeabcdddea`, when the pattern is P = `ddea`. The compressed text is `xbc(1,3)yabcd(1,4)e(3,3,1)`. Note how the empty positions allow to skip forward without processing every character of the decompressed file.

4. Experimental results

For our experiments, we have chosen four files of different nature from the Calgary corpus, to which we have added the file *dna*, representing the tobacco chloroplast genome.

Table 1 gives the compression results. The second column gives the size (in bytes) of the uncompressed files. For the following columns, the figures are in percent, giving the savings for each case, i.e., 100 (1 − size of compressed file/size of original file). First are the compression results for LZSS, then for the new algorithm suggested

```
1         $relevant \longleftarrow 0$    $current \longleftarrow 1$    $k \longleftarrow 1$
2         while  next symbol $\mathcal{E}_k$ of $\mathcal{E}(T)$ is not EOF
          or  $buf(current)$ not empty      // check $buf$ after finishing with $\mathcal{E}(T)$
          {
2.1           while $\mathcal{E}_k$ is a triple $($off, len, slide$)$
              {                 // possibly several pointers at same position
2.1.1             if  $relevant \geq current - len$
2.1.1.1               copy_relevant_char()
2.1.2             else
2.1.2.1               $next(current + off) \longleftarrow current + off + len$
2.1.3             $k \longleftarrow k + 1$
              }
2.2           if  $buf(current)$ is not empty
              {
2.2.1             find_pat$(current)$
2.2.2             $relevant \longleftarrow current$
              }
2.3           if  $next(current)$ is not empty
2.3.1             $current \longleftarrow next(current)$
2.4           if  $\mathcal{E}_k$ is a character of $P$
              {
2.4.1             $buf(current) \longleftarrow \mathcal{E}_k$
2.4.2             find_pat$(current)$
2.4.3             $next(relevant) \longleftarrow current$
2.4.4             $relevant \longleftarrow current$
              }
2.5           $current \longleftarrow current + 1$
          }
```

FIGURE 1: The compressed matching algorithm

current	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
buf								a			d	e	a			d	d	e	a
next					8								16						

FIGURE 2: Example of buffer during compressed matching

here, finally for Manber's method [7], which is based on the substitution of common pairs of characters by special symbols that are still encoded in a single byte. As this method requires the basic alphabet to consist of up to 127 characters, it could not be applied to file *obj1*.

File	size	LZSS	new	Manber
obj1	21645	45.6	41.8	–
progc	38368	57.5	47.9	20.8
paper1.txt	52860	55.1	43.7	24.3
book1	768769	45.4	30.2	26.7
dna	155847	64.3	45.6	24.5

TABLE 1: *Comparitive chart of compression performance*

For LZSS and the new algorithm, the characters were encoded by 9 bits. For pairs and triples, we used variable length encodings based on the ideas of [3], which take advantage of the fact that the smaller values of *offset*, *length* and *slide* appear more frequently. As can be seen, there is a certain loss in compression efficiency when using the new algorithm instead of LZSS, which is mainly due to the encoding of the triples. Nevertheless, the new method performs much better than that of [7].

To empirically compare the processing times, the patterns were chosen in the following way. For each file, we considered sub-strings of lengths 5 to 25, selected their starting positions randomly within the file and repeated this process 5 times. We thus considered 105 patterns of different lengths, each of which occurring at least once, looked for all their occurrences, and averaged the search times. The results, in seconds, are given in Table 2.

The processing time is typically reduced by about a third, since parts of the decompressed file are not scanned. The savings are more important for the shorter patterns.

5. Conclusion and future work

The new compression algorithm is not meant to compete with others on the grounds of compression ratio or processing speed. It is an adaptation of standard LZ methods, but because of the forward pointers, both encoding and decoding are more involved and therefore much slower. We should however remember that for the application at

File	LZSS	new
obj1	0.44	0.28
progc	0.91	0.65
paper1	1.48	1.02
book1	40.89	25.70
dna	4.83	3.88

TABLE 2: *Empirical comparison of processing time*

hand here, compression is done only once, usually off-line, and if indeed we can search within the compressed text, decompression may never be necessary. The compression performance of the new method is also inferior to those of comparable LZ methods, but it is still superior to that of simple methods like Huffman coding, and even a method specially built for the compressed matching. The main motivation was to facilitate the direct search in the compressed file, which is done faster with the new method.

We are now working on refining the matching algorithm. In the description above, we have to deal with every occurrence in $\mathcal{E}(T)$ of every character on P. If P is very long, or if most of the characters or the most frequent ones are in P, we might be forced to process almost every character on the input. It therefore seems useful to record only the appearances of sub-patterns of P of length 2 or more. For instance, if the pattern is $P =$ `adcba` and the compressed text includes ...`abcd(3,3)`..., then the sub-string `bcd` referred to by the pair (3,3) seems not relevant, despite the fact that all its characters appear in P, but it is not a sub-string of P, nor is a suffix of `bcd` matching a prefix of P, or a prefix of `bcd` matching a suffix of P. Nevertheless, one can build examples for which such a strategy may miss certain occurrences of P. Empirical tests have shown that this occurs rarely. We are thus working both of extending the exact matching technique to include tests for longer sub-patterns, and on turning the algorithm to a heuristic, which may occasionally miss a match, but works generally much faster.

References

[1] AMIR A., BENSON G., Efficient two-dimensional compressed matching. *Proc. Second IEEE Data Compression Conference* 279–288 (1992).

[2] Amir A., Benson G., Farach M., Let Sleeping Files Lie: Pattern Matching in Z-compressed Files *Journal of Computer and System Sciences* **52** (1996) 299-307.

[3] Choueka Y., Fraenkel A.S., Klein S.T., Compression of Concordances in Full-Text Retrieval Systems, *Proc. 11-th ACM-SIGIR Conf.,* Grenoble (1988) 597–612.

[4] Farach M., Thorup M., String Matching in Lempel-Ziv Compressed Strings, *Proc. 27th Annual ACM Symposium on the Theory of Computing*, (1995) 703–712.

[5] Kida T., Takeda M., Shinohara A., Arikawa S., Shift-And approach to pattern matching in LZW compressed text, *Proc. 10th Annual Symposium on Combinatorial Pattern Matching (CPM 99)*, *LNCS* **1645**, Springer Verlag, (1999) 1–13.

[6] Knuth D.E., Morris J., Pratt V., Fast Pattern matching in strings, *SIAM J. on Computing* **6** (1977) 323–350.

[7] Manber U., A Text Compression Scheme That allows Fast Searching Directly in the compressed File, *ACM Trans. on Inf. Sys.* **15** (1997) 124–136.

[8] Moura E., Navarro G., Ziviani N., Baeza-Yates R., Direct pattern matching on compressed text. *Proc SPIRE'98*, *IEEE CS Press* (1998) 90–95.

[9] Moura E., Navarro G., Ziviani N., Baeza-Yates R., Fast searching on compressed text allowing errors. *Proc. 21-st ACM-SIGIR Conf.,* Melbourne (1998) 597–612.

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

[11] Storer J.A., Szymanski T.G., Data Compression via Textual Substitution, *Journal of the ACM* **29** (1982) 928–951.

[12] Whiting D.L., George G.A., Ivey G.E., Data compression apparatus and method, U.S. Patent 5,016,009, May 14, (1991).

[13] Williams R.N., An extremely fast Ziv-Lempel data compression algorithm, *Proc. Data Compression Conference DCC–91*, Snowbird, Utah (1991) 362–371.