Approximate Searching on Compressed Text

Carlos Avendaño Pérez, Claudia Feregrino Uribe
Instituto Nacional de Astrofísica Óptica y Electrónica
carlosap@inaoep.mx, cferegrino@inaoep.mx

Gonzalo Navarro[1]
Universidad de Chile
gnavarro@dcc.uchile.cl

Abstract

The approximate searching problem on compressed text tries to find all the matches of a pattern in a compressed text, without decompressing it and considering that the match of the pattern with the text can have a limited number of differences. This problem has diverse applications in information retrieval, computational biology and signal processing, among others. One of the best solutions to this problem is to execute a multipattern search of a set of pieces of the pattern, followed by a local decompression and a direct verification in the decompressed areas. In this work an improvement to this solution concerning verification is presented, where instead of executing a decompression process and searching for the pattern, bit-parallel automata are constructed that recognize the pattern. In this way, we perform the entire searching process without decompressing the text and obtain competitive times, compared to decompressing text and searching it with the best existing algorithms.

1. Introduction

The considerable increase of available textual information in line drives to the necessity of compressing the collections to reduce storage requirements. The problem is to perform efficient searches in compressed text, considering that it is common to find documents that contain words badly written and that it would be impossible to recover these documents unless the same error in the consultation is made.

One of the solutions to this problem is the approximate search of patterns on compressed text, which is defined as follow: Given a text T of length u whose compressed text corresponds to Z of length n, a P pattern of length m, and a maximum number of allowed errors k, find all the matches of P in T using only Z and whose maximum distance of edition to the pattern is k. The distance between two strings x and y is the minimum number of edition operations necessary to transform x into y. The allowed edition operations are insertion, elimination or replacements of a character. This solution has diverse applications in information retrieval, computational biology and signal processing, among others.

Diverse methods of compression exist, among them the family Ziv-Lempel stands out for its good compression ratios combined with suitable times for compression and decompression, particularly LZ78 [1] and LZW [2]. One of the best approaches to the problem of approximate searching on LZW compressed text was proposed in [3], and consists on splitting the pattern in k+1 subpatterns and performing an exact multipattern search, with the set of resulting subpatterns using Boyer-Moore's algorithm [4], whenever a subpattern is found, the existence of the complete subpattern is verified by decompressing the text around the subpattern and executing an approximate searching classic algorithm on the decompressed text.

[1] Funded in part by Fondecyt Grant 1-020831

In this work an improvement to the verification is presented, instead of executing decompression and searching processes on the pattern, two bit-parallel automata are constructed that recognize the complete pattern, one recognizes to the left of the pattern found and the other one to the right, such that the sum of the errors found in each of the automata is smaller or equal to k. In this way, we perform the entire search process without decompressing the text and obtain competitive times, compared to decompressing text and searching it with the best existing algorithms.

2. Related Work

Approximate searching on compressed text was proposed as an open problem in [5]. In [6] this problem has been solved for the LZ78/LZW formats in $O(mkn + R)$ time for the worst case (where R is the number of matches to find) and in $O(k^2n+R)$ time for the average case using dynamic programming techniques. Bit-parallelism is other technique used for approximate searching. With this technique it is possible to pack many values in the bits of a computer word of w bits and to update all of them in parallel. In [7] this technique was used to obtain a time $O(nmk^3/w)$ in the worst case. Nevertheless, the solutions presented as in [6] as in [7] are very slow.

In [3] the first practical solution to the approximate searching problem was presented using filtering algorithms. This solution works fine for text compressed with LZ78/LZW algorithms and it is based on splitting the pattern in k+1 subpatterns and executing a multipattern search of them, followed by a local decompression and a direct verification in the candidate text areas. In spite of the good performance of this approach, the verification can be improved using techniques of bit-parallelism and filtering. In this work, we demonstrate it.

3 The LZW Compression Format

The Ziv-Lempel family of algorithms replaces substrings in the text by a pointer to a previous occurrence of them. A particularly interesting variant is LZW, which is used for Unix's *Compress* program. The LZW method starts by initializing the dictionary to all the symbols in the alphabet. In the common case of 8-bits symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input.

The principle of LZW is the following: the encoder inputs symbols one by one and accumulates them in a string *I*. After each symbol is input and is concatenated to *I*, the dictionary is searched for string *I*. As long as *I* is found in the dictionary, the process continues. At a certain moment, adding the next symbol '*a*' causes the search to fail; string *I* is in the dictionary but string *Ia* is not. At this moment the encoder outputs the dictionary pointer that points to string *I*, saves string *Ia* in the next available dictionary entry and initializes string *I* to symbol *a*.

4. Improved approximate searching on compressed text algorithm.

The algorithm presented in [3] is divided in three phases: 1) splitting the pattern into k+1 subpatterns, 2) executing multipattern search, and 3) verifying the match of complete pattern. In this work, phase 1 and 2 are identical to that ones proposed in [3], however we present an improvement to phase 3, the verification. This phase is explained next in detail.

4.1 Verification

We begin supposing that the pattern has been splitted in $k + 1$ subpatterns and one of them has been found by means of an exact multipattern search. If $P = P_1 F P_2$ and F is the subpattern found, a bit mask B for P_1 (inverted) is precalculated and other for P_2 for each of the $k + 1$ subpatterns. Next, using the bit mask corresponding to the subpattern found, the bit mask $B[c]$ has the j^{th} bit active if $p_j = c$ (where p_j represents the jth character of P_1 or P_2).

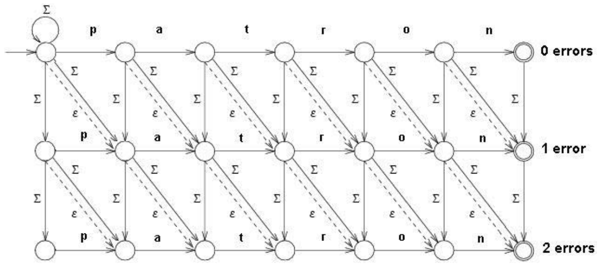


Fig. 1. No-deterministic automaton for approximate searching for the string 'patron' allowing 2 errors.

For the verification of P_1 y P_2 we simulate a non-deterministic automaton (NFA) similar to the one in figure 1. Every row denotes the number of errors seen, the first row denote zero errors, the second row one, etc. Each time a character of the text is read the NFA change its state. The horizontal, vertical, diagonal and dashed diagonal arrows represent matching, inserting, replacing or deleting of one character respectively. The NFA indicates a match when some final state is active, the row in which is that state will indicate the number of errors found.

The automaton simulation was made according to the algorithms proposed in [8], using *k+1* bit masks for representing the NFA rows. The formula for updating the bit masks according to the character reading is:

$$R'_0 \longleftarrow ((R_0 << 1) \mid 0^{m-1}\,1)\ \&\ B[t_j] \qquad \textbf{(1)}$$
$$R'_i \longleftarrow ((R_i << 1)\ \&\ B[t_j] \mid R_{i-1} \mid (R_{i-1} << 1) \mid (R'_{i-1} << 1)$$

The bit masks R_i are initialized to $0^{m-i}\,1^i$. In (1), R'_i denotes the horizontal, vertical, diagonal and dashed diagonal arrows of figure 1 respectively.

The figure 2 shows how to obtain the characters to feed the automaton. We showed hypothetical windows of a text compressed with LZW. The dark boxes represent the first character of the next block, and the lines represent the referenced block. For each block the final character, length, the block it refers, and any other data dependent of the algorithm is kept.

Next, the recognition with P_1 begins; it is less time consuming to verify the blocks towards the left because the characters have been obtained in the adequate order to feed the automaton.

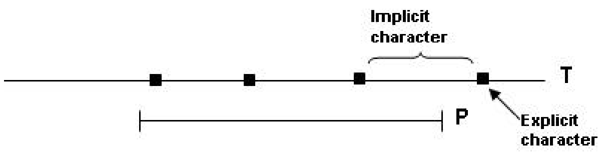


Fig. 2. Hypothetical windows of a text compressed with LZW.

All initiates with the block where the subpattern *F* starts. In figure 3, the block *j* represents the block where the subpattern initiates, which is within block *jj*. From the block *j*, we obtain all explicit characters to what *j* refers, until we obtain a block of length minor or equal to 1. Then, all the referenced characters by the block *jj*-1 are obtained, by the block *jj*-2 and so on, feeding the automaton with the characters that are obtained. The process finishes when the automaton dies, or when the number of errors found is greater that *k*.

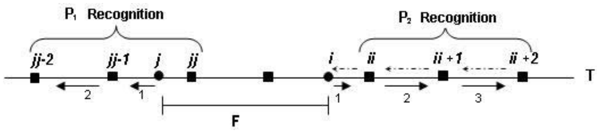


Fig. 3. Order in which the characters are obtained to feed the automaton in the verification process.

If the number of errors found in the process *k*' is greater that *k*, we search for other subpattern, else if *k*' = *k* the automaton of P_2 is reconfigured to allow only *k*'' = *k* – *k*' errors.

Now, we begin with the block where *F* finalizes. In the figure 3 this block is represented by *i*, which is within block *ii*. To obtain the characters is more difficult in this moment, because it is necessary to go backwards in the referenced chain from block *ii* until we find block *i*, storing in the array the explicit characters of each visited blocks. When finding the block *i*, we feed the automaton with the last characters stored in the array and so on.

If the obtained characters are not enough for arriving at a final state of the automaton, a new block *ii*= *ii*+1 is read and processed in the same way. Obtaining so the characters of the blocks to which the block *ii* makes reference, storing again the characters in an array and feeding the automaton as in the previous case. The process continues until the automaton dies or a match of the patterns is found.

When the verification is finalized, if the total number of errors found is smaller or equal to *k*, a match of the complete pattern is reported.

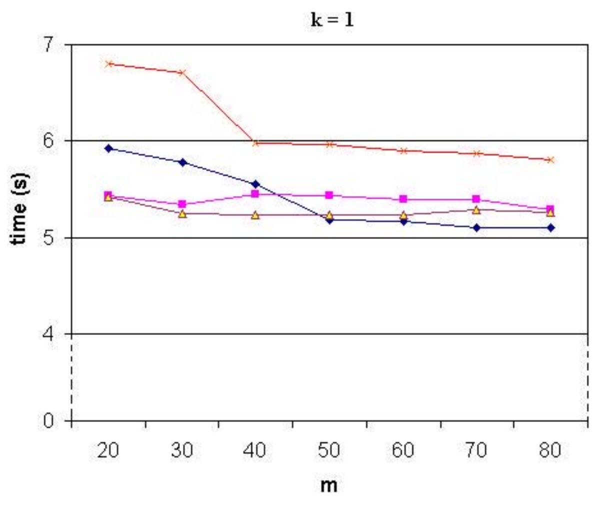
5. Experimental Results

The characteristics of the computer used for carrying out the experiments are: Intel Pentium IV processor of 1.3 GHz with 256 MB RAM running Linux Red Hat 8.0.

The experiments were carried out over 80 MB of English text extracted from a set of the titles and abstract of 270 medical magazines. The text file was compressed with Unix`s *Compress* program obtaining a compression of 58%. The patterns were chosen randomly with length of 20 to 80 and $k = 1$.

In order to carry out the experiments, a tool denominated *alzgrep* (*lzgrep* extended with *a*pproximate searching) was implemented and later will be publicly available. The results were compared with *agrep* [8] and *nrgrep* [9] that are the best approximate searching classic algorithms available and with the original algorithm denominated *PP-BM*.

The figure 4 shows the times achieved for each of the algorithms. The graphs show how our algorithm achieves better times for long patterns (greater than 40) and its efficiency decrements when the pattern is short (shorter than or equal to 40). This happens because the subpatterns have a short length and the algorithm finds matches of them and must verify frequently.



m	20	30	40	50	60	70	80
◆ alzgrep	5.92	5.77	5.556	5.18	5.166	5.105	5.1
■ nrgrep	5.43	5.34	5.44	5.437	5.386	5.395	5.29
△ agrep	5.42	5.25	5.236	5.232	5.23	5.285	5.26
✕ PP-BM	6.8	6.7	5.97	5.96	5.9	5.87	5.8

Fig. 4. Searching time in seconds allowing different pattern lengths and k=1.

6. Conclusions

We have presented an improvement of a solution for approximate searching on compressed text. From the obtained results of the experiments carried out, we conclude that the proposed algorithm has the capability of performing approximate searching on compressed text with competitive times compared with the best approximate searching algorithms available nowadays, obtaining better results than them for low error rates.

In addition, it is not necessary to decompress the text to perform the search, which allows searching on the compressed text without any additional decompression tool.

References

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

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

[3] 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.

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

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

[6] 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.

[7] 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.

[8] S. Wu and U. Manber. agrep- a fast approximate pattern-matching tool. In Proc. USENIX Technical Conference, pages 153-162, 1992.

[9] G. Navarro, NR-grep: a fast and flexible pattern-matching tool. Software-Practice & Experience, 31(13). p. 1265-1312, 2001.