Boyer-Moore String Matching over Ziv-Lempel Compressed Text

Gonzalo Navarro[1*] and Jorma Tarhio[2**]

[1] Dept. of Computer Science, University of Chile. gnavarro@dcc.uchile.cl
[2] Dept. of Computer Science, University of Joensuu, Finland. tarhio@cs.joensuu.fi

Abstract. We present a Boyer-Moore approach to string matching over LZ78 and LZW compressed text. The key idea is that, despite that we cannot exactly choose which text characters to inspect, we can still use the characters explicitly represented in those formats to shift the pattern in the text. We present a basic approach and more advanced ones. Despite that the theoretical average complexity does not improve because still all the symbols in the compressed text have to be scanned, we show experimentally that speedups of up to 30% over the fastest previous approaches are obtained. Moreover, we show that using an encoding method that sacrifices some compression ratio our method is twice as fast as decompressing plus searching using the best available algorithms.

1 Introduction

The *string matching problem* is defined as follows: given a pattern $P = p_1 \ldots p_m$ and a text $T = t_1 \ldots t_u$, find all the occurrences of P in T, i.e. return the set $\{|x|,\ T = xPy\}$. The complexity of this problem is $O(u)$ in the worst case and $O(u \log_\sigma(m)/m)$ on average (where σ is the size of the alphabet Σ), and there exist algorithms achieving both time complexities using $O(m)$ extra space [3, 6].

A particularly interesting case of string matching is related to text compression. Text compression [4] tries to exploit the redundancies of the text to represent it using less space. There are many different compression schemes, among which the Ziv-Lempel family [23, 24] is one of the best in practice because of their good compression ratios combined with efficient compression and decompression time.

The *compressed matching problem* was first defined in the work of Amir and Benson [1] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \ldots z_n$, and a pattern P, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching,

[*] Work developed during postdoctoral stay at the University of Helsinki, partially supported by the Academy of Finland and Fundación Andes. Also supported by Fondecyt grant 1-990627.

[**] Supported in part by the Academy of Finland.

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 $R = u > n$).

The compressed matching problem is important in practice. Today's textual databases are an excellent example of applications where both aspects of the problem are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. 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.

There exist a few works about searching on compressed text, which we cover in the next section. The most promising ones run over the LZ78/LZW variants of the LZ family. They have achieved a good $O(m^2 + n + R)$ worst case search time, and there exist practical implementations able to search in less time than that needed for decompression plus searching. All those works have concentrated in the worst case.

However, Boyer-Moore type techniques, which are able to skip some characters in the text, have never been explored for searching compressed text. Our work points in this direction. We present an application of Boyer-Moore techniques for string matching over LZ78/LZW compressed texts. The resulting algorithms are $\Omega(n)$ time on average, $O(mu)$ time the worst case, and $O(n)$ extra space. This does not improve the existing complexities, but they are faster in practice than all previous work for $m \geq 15$, taking up to 30% less time than the fastest existing implementation. We also present experiments showing that, using an LZ78 encoder that sacrifices some compression ratio for decompression speed, our algorithms are twice as fast as a decompression followed by a search using the best algorithms for both tasks.

2 Related Work

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 in [16], 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 [23] is able to reference any substring of the text already processed, while LZ78 [24] and LZW [20] 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 [2], which searches in LZ78 needing time and space $O(m^2 + n)$ for the existence problem.

The only search technique for LZ77 [7] is a randomized algorithm to determine in time $O(m + n\log^2(u/n))$ whether a pattern is present or not in the text. It seems that with $O(R)$ extra time both [2] and [7] could find all the R occurrences of the pattern.

An extension of [2] to multipattern searching was presented in [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. Other algorithms for different specific search problems have been presented in [8, 11].

New practical results appeared in [17], who presented 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 in [14]. Finally, [12] generalized the existing algorithms and nicely unified the concepts in a general framework.

3 Basic Concepts

3.1 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 [4]. We are particularly interested in the LZ78/LZW format, which we describe in depth (this is taken from [17]).

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [24]) 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 $O(u)$ 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

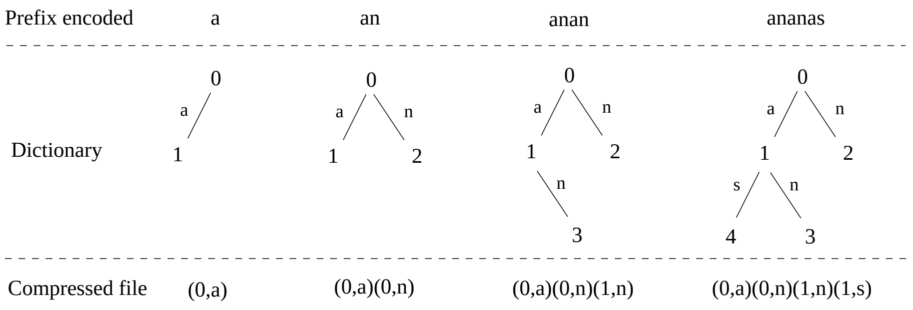

Fig. 1. Compression of the word *ananas* with the algorithm LZ78.

needs to build the same dictionary (the pair that 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 [20]. 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.

3.2 Boyer-Moore String Matching

The Boyer-Moore (BM) family of text searching algorithms proceed by sliding a *window* of length m over the text. The window is a potential occurrence of the pattern in the text. The text inside the window is checked against the pattern normally from right to left (although not always). If the whole window matches then an occurrence is reported. To shift the window, a number of criteria are used, which try to balance between the cost to compute the shift and the amount of shifting obtained. Two main techniques are used:

Occurrence heuristic: pick a character in the window and shift the window forward the minimum necessary to align the selected text character with the same character in the pattern. Horspool [9] uses the m-th window character and Sunday [19] the $(m+1)$-th (actually outside the window). These methods need a table d that for each character gives its last occurrence in the pattern (the details depend on the versions). The Simplified BM (SBM) method [5] uses the character at the position that failed while checking the window, which needs a larger table indexed by window position and character.

Match heuristic: if the pattern was compared from right to left, some part of it has matched the text in the window, so we precompute the minimum shift necessary to align the part that matched again with the pattern. This requires a table of size m that for each pattern position gives that last occurrence of $P_{i \dots m}$ in $P_{1 \dots m-1}$. This is used in the original Boyer and Moore method [5].

4 A Simple Boyer-Moore Technique

Consider Figure 2, where we have plotted a hypothetical window approach to a text compressed using LZ78. Each LZ78 block is formed by a line and a final box. The box represents the final *explicit* character c of the block $b = (s, c)$, while the line represents the *implicit* characters, i.e. a text that has to be obtained by resorting to previous referenced blocks (s, then the block referenced by s, and so on).

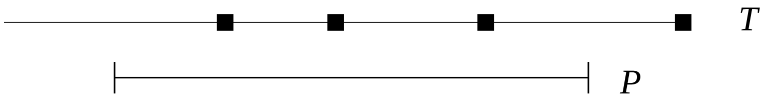

Fig. 2. A window approach over LZ78 compressed text. Black boxes are the explicit characters at the end of each block, while the lines are the implicit text that is represented by a reference.

Trying to apply a pure BM in this case may be costly, because we need to access the characters "inside" the blocks (the implicit ones). A character at distance i to the last character of a block needs going i blocks backward in the referencing chain, as each new LZ78 block consists of a previous one concatenated with a new letter.

Therefore we prefer to start by considering the explicit characters in the window. To maximize the shifts, we go from the rightmost to the leftmost. We precompute a table

$$B(i, c) = \min(\{i\} \cup \{i - j,\ 1 \leq j \leq i\ \wedge\ P_j = c\})$$

which gives the maximum safe shift given that at window position i the text character is c (this is similar to the SBM table, and can be easily computed in $O(m^2 + m\sigma)$ time). Note that the shift is zero if the pattern matches that window position.

As soon as one of the explicit characters permits a non-zero shift, we shift the window. Otherwise, we have to consider the implicit characters. Figure 3 shows the order in which we consider them. The last block is left for the end, since despite it can give good shifts, it is costly to reach the relevant characters (the block can be unfolded only from right to left). The other blocks are unfolded in right to left order, block by block. When unfolding a block, we obtain a new text character (right to left) for each step backward in the referencing chain. For each such character, if we obtain a non-zero shift we immediately advance the window and restart the whole process with a new window.

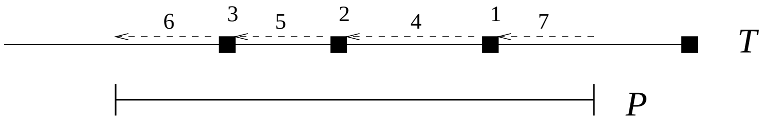

Fig. 3. Evaluation order for the simple algorithm. First the explicit characters right to left, then the implicit ones right to left (save the last one), and finally the last block.

If, after having considered all the characters we have not obtained a non-zero shift, we can report an occurrence of the pattern at the current window position. The window can then be advanced by one.

The algorithm can be applied on-line, that is, reading the compressed file block by block from disk. We read zero or more blocks until the last block read finishes ahead the window, then apply the previous procedure until we can shift the window, and start again. For each block read we store its last character, the block it references, its position in the uncompressed text and its length (these last two are not stored in the compressed file but computed on the fly).

Note that it is possible that the pattern is totally contained in a block, in which case the above algorithm will unfold the block to compare its internal characters against the pattern. It is clear that the method is efficient only if the pattern is not too short compared to the block length.

A slight improvement over this scheme is to add a kind of "skip-loop": instead of delaying the shifting until we read enough blocks, try to shift with the explicit character of each new block read. This is in practice like considering the explicit characters in left to right order. It needs more and shorter shifts but resorts less to previously stored characters. We call "BM-simple" our original version and "BM-simple-opt" this improvement.

Note that even in the best case we have a complexity of $\Omega(n)$ because all the text blocks have to be scanned. However, the method is faster in practice than previous algorithms, as shown later. Appendix A depicts the complete algorithm.

5 Multicharacter Boyer-Moore

Although the simple method is fast enough for reasonably large alphabets, it fails to produce good shifts when the alphabet is small (e.g. DNA). Multicharacter techniques, consisting in shifting by q-tuples of characters instead of one character, have been successfully applied to search uncompressed DNA [18]. Those techniques effectively increase the alphabet size and produce longer shifts in exchange for slightly more costly comparisons.

We have attempted such an approach for our problem. We select a number q and build the shift tables considering q-grams. For instance, for the pattern "abcdefg", the 3-gram "cde" considered at the last position yields a shift of 2, while "xxx" yields a shift of 5. Once the pattern is preprocessed we can shift using text q-grams instead of text characters. That is, if the text window is $x_1 x_2 \ldots x_m$ we try to shift using the q-grams $x_{m-q+1} \ldots x_m$, then $x_{m-q} \ldots x_{m-1}$, etc. until $x_1 \ldots x_q$. If none of these q-grams produces as positive shift, then the pattern matches the window. The preprocessing takes $O(m^2 + m\sigma^q)$ time.

The method is applied to the same LZ78 encoding as follows. At search time, we do not store anymore the last character of each block but its last q-gram. This last q-gram is computed on the fly, the format of the compressed file is the same as before. To compute it, we take the referenced block, strip the first character of its final q-gram and append the extra character of the new block. Then, the basic method is used except because we shift using the whole q-grams.

One complication appears when the block is shorter than q. In this case the best choice is to pad its q-gram with the last characters of the block that appears before it (if this is done all the time then the previous block does have a complete q-gram, except for the first blocks of the text). However, we must be careful when this short block is referenced, since only the characters that really belong to it must be taken from its last q-gram.

Finally, if q is not very small, the shift tables can be very large ($O(\sigma^q)$ size). We have used hashing from the q-grams to an integer range $0 \ldots N-1$ to reduce the size of the tables and to lower the preprocessing time to $O(m^2 + mN)$. This makes it necessary an explicit character-wise checking of possible matches, which is anyway required because we cannot efficiently check the first $q-1$ characters of the pattern.

We have implemented this technique using $q = 4$ (which is appropriate to store the q-gram in a word of our machine), and it is called "BM-multichar" in the experiments, where we show that it improves over BM-simple on DNA text.

6 Shifting by Complete Blocks

Despite that we have obtained good results with BM-multichar, we present now a more sophisticated technique that gave better results.

The idea is that, upon reading a new block, we could shift using the whole block. However, we cannot have an $B(i,b)$ table with one entry for each possible block b. Instead, we precompute

$$\begin{aligned} J(i,\ell) = \max(& \{j,\ \ell \le j < i \ \wedge\ P_{j-\ell+1\ldots j} = P_{i-\ell+1\ldots i}\} \\ & \cup\ \ \{j,\ 0 \le j < \ell \ \wedge\ P_{1\ldots j} = P_{i-j+1\ldots i}\}) \end{aligned}$$

that tells, for a given pattern substring of length ℓ ending at i, the ending point of its closest previous occurrence in P (a partial occurrence trimmed at the beginning of the pattern is also valid). The J table can be computed in $O(m^2)$ time by the simple trick of going from $\ell = 0$ to $\ell = m$ and using $J(*, \ell - 1)$

to compute $J(*, \ell)$, so that for all the cells of the form $J(i, *)$ there is only one backward traversal over the pattern.

Now, for each new block read $b_r = (s, c)$, we compute its last occurrence in P, $last(r)$. This is acomplished as follows. We start considering $last(s)$, i.e. the last position where the referenced block appears in P. We check if $P_{last(s)+1} = c$, in which case $last(r) = last(s) + 1$. If this is not the case, we need to obtain the previous occurrence of b_s in P, but this is also the previous occurrence of a pattern substring ending at $last(s)$. So we can use the J table to obtain all the following occurrences of b_s inside P, until we find one that is followed by c (and then this is the last occurrence of $b_r = b_s c$ in P) or we conclude that $last(r) = 0$. This process takes $O(\min(mn, \sigma^m))$ across all the search and is cheap in practice.

Once we have computed the last occurrence of each block inside P, we can use the information to shift the window. However, it is possible that the last occurrence of a block b_r inside P is indeed after the current position of b_r inside the window. In the simple approach (Section 4) this is solved by computing $B(i, c)$, i.e. the last occurrence of c inside P *before position* i. This would require too much effort in our case. We prefer to use J again in order to find previous occurrences of b_r inside P until we find one that is at the same position of b_r in the window or before. If it is at the same position we cannot shift, otherwise we displace the window. Figure 4 illustrates.

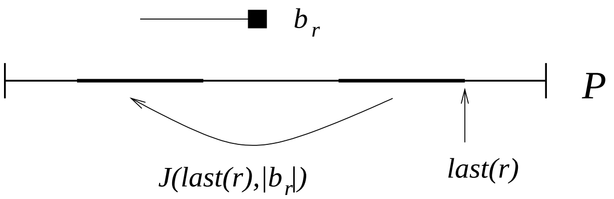

Fig. 4. Using the whole LZ78 block to shift the window. If its last occurrence in P is ahead, we use J until finding the adequate occurrence.

The blocks covered by the window are checked one by one, from right to left (excluding the last one whose endpoint is not inside the window). As soon as one allows a shift the window is advanced and the process restarted. If no shift is possible, the last block is unfolded until we obtain the contained block that corresponds exactly to the end of the window and make a last attempt with it. If all the shifting attempts fail, the window position is reported as a match and shifted in one.

As before, we read blocks until surpass the window and then try to shift. This method is called “BM-blocks” in this paper. The alternative method of trying to shift with each new block read is called “BM-blocks-opt”. The algorithm is depicted in Appendix B.

In the same spirit of shifting using a variable number of characters, we have also adapted the match heuristic of BM. In this case, however, we cannot guarantee that the pattern will be matched right-to-left as on uncompressed text. Therefore, we compute a table $C(i, j)$ that gives the maximum shift if we have matched $P_{i...j}$ and P_{i-1} has mismatched. The definition of C is very similar to that of J. The C table is used when we compare the internal characters, since in that case a contiguous portion of P has been compared (the situation when comparing the explicit characters is much more complex since an arbitrary subset of the pattern positions have been compared). This method, called "BM-complete" in the experiments, did not yield good results.

7 Experimental Results

We tested our algorithms against the fastest existing implementation of previous work [17][1], using the same LZ78 compression format. The format uses a version that loses some compression in exchange for better decompression/search time. It stores the pair (s, c) as follows: s is stored as a sequence of bytes where the last bit is used to signal the end of the code; and c is coded as a whole byte.

The experiments were run on an Intel Pentium III machine running Linux. We have averaged user times over two different files of 10 Mb each. Patterns of lengths 10 to 100 were randomly selected from the texts (1,000 patterns of each length) and the same patterns were used for all the algorithms. The first text, WSJ, is a set of articles from The Wall Street Journal 1987 (natural language), while the second one is DNA with lines cut every 60 characters. We show user times in all the experiments.

Table 1 shows results related to compression efficiency for our compression format and two widely used compressors. As can be seen, our compression ratios are worse than those of classical compressors. On the other hand, decompression time is faster for our format, which improves search time.

Method	Compression ratio	Compression time	Decompression time
Ours (LZ78)	WSJ: 45.02% DNA: 39.69%	WSJ: 5.09 sec DNA: 4.31 sec	WSJ: 0.79 sec DNA: 0.72 sec
Unix *Compress* (LZW)	WSJ: 38.75% DNA: 27.91%	WSJ: 2.52 sec DNA: 2.43 sec	WSJ: 0.92 sec DNA: 0.75 sec
Gnu *gzip* (LZ77)	WSJ: 33.57% DNA: 30.43%	WSJ: 10.63 sec DNA: 25.10 sec	WSJ: 0.81 sec DNA: 0.78 sec

Table 1. Results on compression and decompression using different formats.

Figure 5 shows a comparison of the diverse search methods we have proposed along the paper. As can be seen, BM-simple-opt is the best choice for natural language, while BM-blocks (without the "optimization") is the best on DNA. BM-multichar works better than BM-simple on DNA, but BM-blocks is superior.

[1] The bit-parallel algorithm of [14] should be similar, but it is implemented over Unix *Compress* and it is slower.

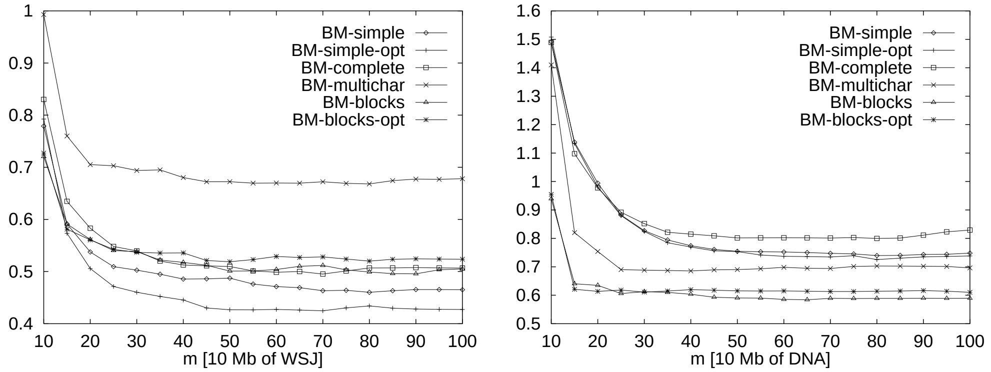

Fig. 5. Search time (in seconds of user time) for our different algorithms.

Figure 6 compares our best algorithms against previous work. The previous algorithm [17] is called "Bit-parallel" and our implementation of it works only until $m = 32$ (it would be slower, not faster, for longer patterns). We have also considered the "naive" approach of decompressing-then-searching. Two choices are shown: DS uses our LZ78 format and decompresses the file in memory while applying a Sunday [19] search algorithm over it; "D+Agrep" first decompresses the text and then then runs *agrep* over it. *Agrep* [21, 22] is considered the fastest text searching tool, and we recall that the decompression time of our format is the fastest.

As can be seen, our algorithms are significantly faster than Bit-parallel (up to 30%) and than both decompress-then-search approaches (up to 50%), even for short patterns ($m \geq 15$).

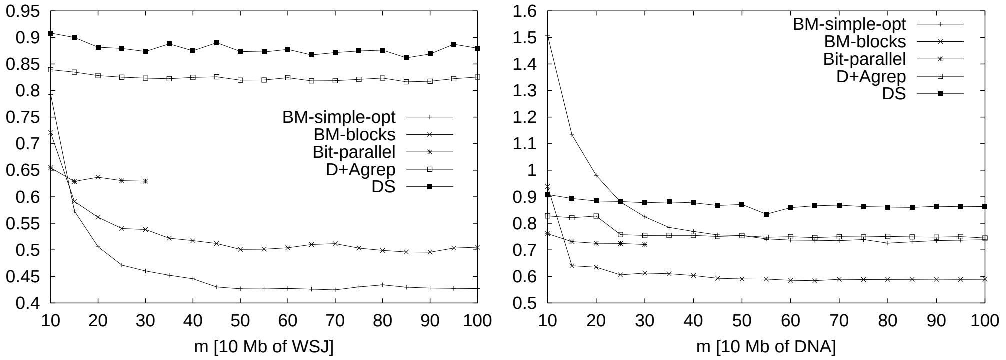

Fig. 6. Search time (in seconds of user time) of the best previous algorithms and our new ones.

It is also interesting that the methods reach soon a limit on m from where they do not improve anymore. This is due to the need for reading all the text blocks, regardless of how long is the pattern. This is unavoidable in principle to know the text position we are on.

If we compute the scanning efficiency of our best algorithms, we have that they are able to search at up to 16.6 Mb/sec on DNA and 22.2 Mb/sec on natural language text (computed on the uncompressed text). Bit-parallel obtains 14 Mb/sec on DNA and 16 Mb/sec on WSJ, while decompress-then-search achieves 13 Mb/sec on DNA and 12 Mb/sec on WSJ. If we have the text already decompressed, then *agrep* alone scans the text at about 300 Mb/sec times faster. This shows that, despite that we offer an interesting alternative to decompressing and searching of texts that have to be compressed by some other reason, we are far from giving an extra reason to compress the text (i.e. achieving less time in searching the compressed text than for searching the uncompressed text).

On the other hand, we would like to point out that these results have a strong dependence with respect to the type of machine used. The same experiments run on a Sun UltraSparc-1 of 167 MHz gave, for $m = 30$ on WSJ, 1.4 seconds for BM-simple-opt as well as for the bit-parallel algorithm of [17], while *agrep* took about 0.45 seconds. We developed another version of BM-simple-opt based on a different coding that, in exchange for 2% to 4% extra space, permits to know the length of the new block without accessing the referenced one. This new algorithm takes about 0.85 seconds under the above conditions, which is 60% of the time of the bit-parallel algorithm and about twice the time of pure *agrep*. This version, however, is slower than the original one on the Intel machine. This shows that locality of reference is much more important on the Sun machine.

8 Conclusions

We have presented the first Boyer-Moore approaches to string matching over Ziv-Lempel compressed text (specifically, the LZ78 format). We first presented a simple approach close to the Simplified-Boyer-Moore algorithm that is the best for all but very small alphabets (e.g. it is suitable for natural language). Then we presented stronger approaches, the most successful one based on shifting on complete LZ78 blocks. This one is the fastest for small alphabets (e.g. DNA text). Our experimental results show that the new algorithms are faster than the best previous approaches even from patterns of length 15, achieving up to 30% reductions in the search time. The new algorithms hold the characteristic feature of all the search algorithms of Boyer-Moore type: the algorithms run faster when the pattern gets longer (up to a certain limit).

We could theoretically strenghten the algorithms in the following way: if the window contains a border between blocks then we apply our shifting machinery, but when it is inside a block we simply copy the matches already found in the text area that the containing block references, and shift the window to the end of the block. However, in practice the blocks are not so long for this to make a real difference.

We are currently working in improving the current techniques and exploring new ones, as there are many open options. A very interesting question is whether it is possible to avoid reading all the text blocks, as this is the major bottleneck for further improvement.

References

1. A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.
2. 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.
3. A. Apostolico and Z. Galil. *Pattern Matching Algorithms.* Oxford University Press, Oxford, UK, 1997.
4. T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.
5. R. S. Boyer and J. S. Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, 1977.
6. M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.
7. M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.
8. L. Gasieniec, M. Karpinksi, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encodings. In *Proc. SWAT'96*, 1996.
9. R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.
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, 2000. In this same volume.
12. T. Kida, Y. Shibata, M. Takeda, A. Shinohara, and S. Arikawa. A unifying framework for compressed pattern matching. In *Proc. 6th Intl. Symp. on String Processing and Information Retrieval (SPIRE'99)*, pages 89–96. IEEE CS Press, 1999.
13. T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, 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. E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Trans. on Information Systems*, 2000. To appear. Previous versions in *SIGIR'98* and *SPIRE'98.*
17. 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.
18. H. Peltola and J. Tarhio. String matching in the DNA alphabet. *Software Practice and Experience*, 27(7):851–861, 1997.
19. D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, 1990.
20. T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

21. S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, October 1992.
22. S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX Technical Conference*, pages 153–162, Berkeley, CA, USA, Winter 1992.
23. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.
24. J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.

A The Simple Optimized Algorithm in Detail

```
Search ($P$,$m$,$Z$,$n$)

        /* Preprocessing */
  for ($i \in 1 \ldots m$, $c \in \Sigma$) $B(i,c) \leftarrow i$
  for ($i \in 1 \ldots m$)
      for ($j \in i \ldots m$) $B(j,P_i) \leftarrow j - i$
        /* Searching */
  $tpos \leftarrow 0$    /* window initial position */
  $rpos \leftarrow 0$    /* amount of text read */
  $length(0) \leftarrow 0$    /* length of the blocks */
  $from(0) \leftarrow 0$    /* text position of the blocks */
  $i \leftarrow 0$    /* number of current block */
  while ($true$)
      readBlocks:
      if ($rpos - tpos \leq m$)
          while ($true$)
              $i \leftarrow i + 1$
              if ($i > n$) finish the search
              obtain $b_i = (j,c)$ from $Z$
              $ref(i) \leftarrow j$    /* referenced block */
              $lastchar(i) \leftarrow c$    /* char at the end */
              $from(i) \leftarrow rpos$
              $length(i) \leftarrow length(j) + 1$
              $rpos \leftarrow rpos + length(i)$
              if ($rpos - tpos > m$) break loop
              $tpos \leftarrow tpos + B(rpos - tpos, c)$
          /* try to shift with explicit characters */
      $j \leftarrow i$
      while ($true$)
          $offset \leftarrow from(j) - tpos$
          if ($offset < 1$) break loop
          $shift \leftarrow B(offset, char(j-1))$
          if ($shift > 0$)
              $tpos \leftarrow tpos + shift$
              goto label readBlocks
          $j \leftarrow j - 1$
          /* unable to shift by explicit characters, unfold */
      $j \leftarrow i - 1$
```

```
while ($true$)
    $length \leftarrow length(j) - 1$
    $k \leftarrow j$
    $offset \leftarrow from(j) - tpos$
    if ($offset + length \leq 0$) goto label expandLast
    while ($length > 0$)
        $k \leftarrow ref(k)$
        $shift \leftarrow B(offset + length, char(k))$
        if ($shift > 0$)
            $tpos \leftarrow tpos + shift$
            goto label readBlocks
        $length \leftarrow length - 1$
        if ($offset + length \leq 0$) goto label expandLast
    $j \leftarrow j - 1$
    /* only the last ($i$-th) block rests to be tested */
expandLast:
$length \leftarrow length(i) - 1$
$k \leftarrow i$
$offset \leftarrow from(i) - tpos$
while ($offset + length > m$)
    $k \leftarrow ref(k)$
    $length \leftarrow length - 1$
while ($length > 0 \ \wedge \ offset + length > 0$)
    $k \leftarrow ref(k)$
    $shift \leftarrow B(offset + length, char(k))$
    if ($shift > 0$)
        $tpos \leftarrow tpos + shift$
        goto label readBlocks
    $length \leftarrow length - 1$
    /* it passed all the tests, report the match */
report a match beginning at $tpos$
$tpos \leftarrow tpos + 1$
```

B The Algorithm that Shifts by Blocks in Detail

```
Search ($P$,$m$,$Z$,$n$)

        /* Preprocessing ($O(m^2)$, not $O(m^3)$) */
  for ($\ell \in 0 \ldots m$) $J(0,\ell) \leftarrow 0$
  for ($i \in 1 \ldots m$) $J(i,0) \leftarrow i$
  for ($\ell \in 1 \ldots m$)
      for ($i \in 2 \ldots m$)
          $j \leftarrow J(i-1,\ell-1)$
          while ($j > 0 \ \wedge \ P_{j+1} \neq P_i$) $j \leftarrow J(j,\ell-1)$
          if ($j = 0 \ \wedge \ P_{j+1} \neq P_i$) $j \leftarrow j - 1$
          $J(i,\ell) \leftarrow j + 1$
        /* Searching */
  $tpos \leftarrow 0$    /* window initial position */
  $rpos \leftarrow 0$    /* amount of text read */
```

```
$length(0) \leftarrow 0$    /* length of the blocks */
$from(0) \leftarrow 0$    /* text position of the blocks */
$last(0) \leftarrow m$
$i \leftarrow 0$    /* number of current block */
while ($true$)
   readBlocks:
   while ($rpos - tpos \leq m$)
      $i \leftarrow i+1$
      if ($i > n$) finish the search
      obtain $b_i = (j,c)$ from $Z$
      $ref(i) \leftarrow j$     /* referenced block */
      $lastchar(i) \leftarrow c$    /* char at the end */
      $from(i) \leftarrow rpos$
      $length(i) \leftarrow length(j)+1$
      $pos \leftarrow last(j)$
      if ($length(j) > m$) $\ell \leftarrow m$ else $\ell \leftarrow length(j)$
      while ($pos > 0 \ \wedge \ (pos = m \ \vee \ P_{pos+1} \neq c)$)
         $pos \leftarrow J(pos,\ell)$
      if ($pos = 0 \ \wedge \ (P_1 \neq c)$) $pos \leftarrow pos - 1$
      $last(i) \leftarrow pos+1$
      $rpos \leftarrow rpos + length(i)$
      /* try to shift with complete blocks (exclude last one) */
   $j \leftarrow i-1$
   while ($true$)
      $offset \leftarrow from(j+1) - tpos$
      if ($offset < 1$) break loop
      $pos \leftarrow last(j)$
      if ($pos > offset$)
         $\ell \leftarrow length(j)$
         if ($\ell > m$) $\ell \leftarrow m$
         $pos \leftarrow J(pos,\ell)$
         while ($pos > offset$) $pos \leftarrow J(pos,\ell)$
      if ($pos < offset$)
         $tpos \leftarrow tpos + offset - pos$
         goto label readBlocks
      $j \leftarrow j-1$
      /* only the last ($i$-th) block rests to be tested */
   $j \leftarrow i$
   $offset \leftarrow from(j) - tpos - m$
   $length \leftarrow length(i)$
   while ($offset + length > 0$)
      $j \leftarrow ref(j)$
      $length \leftarrow length - 1$
   $pos \leftarrow last(j)$
   if ($pos < m$)
      $tpos \leftarrow tpos + m - pos$
      goto label readBlocks
      /* it passed all the tests, report the match */
   report a match beginning at $tpos$
   $tpos \leftarrow tpos + 1$
```