LZgrep: A Boyer-Moore String Matching Tool for Ziv-Lempel Compressed Text

Gonzalo Navarro[*] Jorma Tarhio[†]

Abstract

We present a Boyer-Moore approach to string matching over LZ78 and LZW compressed text. The idea is to search the text directly in compressed form instead of decompressing and then searching it. We modify the Boyer-Moore approach so as to skip text using the characters explicitly represented in the LZ78/LZW formats, modifying the basic technique where the algorithm can choose which characters to inspect. We present and compare several solutions for single and multipattern search. We show that our algorithms obtain speedups of up to 50% compared to the simple decompress-then-search approach. Finally, we present a public tool, *LZgrep*, which uses our algorithms to offer *grep*-like capabilities searching directly files compressed using Unix's *Compress*, a LZW compressor. *LZgrep* can also search files compressed with Unix *gzip*, using new decompress-then-search techniques we develop, which are faster than the current tools. This way, users can always keep their files in compressed form and still search them, uncompressing only when they want to see them.

1 Introduction

Perhaps one of the most recurrent subproblem appearing in every application is the need to find the occurrences of a pattern string inside a large text. The *string matching problem* lies at the kernel of applications such as information retrieval and management, computational biology, signal processing, databases, knowledge discovery and data mining, just to name a few. Text searching tools such as *grep* are extremely popular and routinely used in everyday's life.

Formally, the string matching problem is defined as, given a pattern $P = p_1 \ldots p_m$ and a text $T = t_1 \ldots t_u$, both sequences over an alphabet Σ, find all the occurrences of P in T, that is, return the set $\{|x|,\ T = xPy\}$. There are myriad of string matching algorithms [9, 33]. The most successful in practice are those algorithms able of skipping text characters without inspecting them all. This includes the Boyer-Moore [6] and the BDM [9] families.

In order to save space, it is usual to store the text in compressed form. *Text compression* [5] tries to exploit the redundancies of the text in order to represent it using less space. Compression is not only appealing for saving space, but also for saving disk and network transmission time. CPU speeds have been doubling every 18 months, while disk transfer times have stayed basically the

[*]Dept. of Computer Science, University of Chile. `gnavarro@dcc.uchile.cl`, `opti-mrm@sii.cl`. Supported in part by Fondecyt grant 1-020831.

[†]Dept. of Computer Science and Engineering, Helsinki University of Technology, Finland. `jorma.tarhio@hut.fi`. Supported in part by the Academy of Finland.

same for 10 years. This makes more and more appealing to save transmission time, even if it has to be paid with some CPU time for decompression.

There are many different compression schemes, among which the Ziv-Lempel family [43, 44, 39] is the most popular in practice because of its good compression ratios combined with efficient compression and decompression performance. As a matter of fact, most of the popular text and general-purpose compression packages in use are based on this family, for example *zip*, *pkzip*, *winzip*, *arj*, *gzip*, *compress*, and so on. Only for images, video and multimedia data are other compression formats used.

One problem that arises when searching a text document that is compressed is that one must decompress it first. This has been the usual approach for long time. Indeed, existing tools like Unix *zgrep* are shorthands for this decompress-then-search approach. However, in recent years, it has been shown that it is possible to speed up this process by searching the text directly in compressed form.

The *compressed matching problem* [3] is defined 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 consists of first decompressesing Z and then performing standard string matching. A smarter algorithm processes Z directly without decompressing it.

Many algorithms for compressed pattern matching have been proposed in the last decade. Many of them, however, work over compression formats that are not widely used, despite being convenient for efficient search. This reduces their possibility of becoming a tool of general use. There are, on the other hand, a few proposals about searching over Ziv-Lempel compressed text. Good worst-case complexities have been achieved, and there exist practical implementations able to search in less time than that needed for decompression plus searching.

However, Boyer-Moore techniques 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 worst-case complexity of the resulting algorithms is not competitive. However, in practice our algorithms are faster than all previous work, and beat the best decompress-then-search approach by up to 50%. We extend our techniques to search for multiple patterns simultaneously.

Using the algorithms developed in this article, we have built *LZgrep*, a compressed text searching tool that provides *grep*-like capabilities when directly searching over files compressed with Unix *compress* program, which is public. *LZgrep* also searches files compressed with *gzip*, a public LZ77 based compressor. *LZgrep* is faster than *zgrep* and resorts to it when the the pattern is more complex than simple string(s), so it can be safely used as a replacement of *zgrep*. *LZgrep* can be freely downloaded for noncommercial purposes from `www.dcc.uchile.cl/~gnavarro/software`.

2 Related Work

One of the most successful approaches to searching compressed text is oriented to natural language. Huffman coding [15] on words, that is, considering the text words instead of characters as the source symbols, has been shown to yield compression ratios[1] of 25%–30% [26]. Moreover, those compressed text can be searched extremely fast, sometimes several times faster than searching the uncompressed text [27]. This approach fits very well in the usual information retrieval scenarios and merges very well with inverted indices [40, 30]. It is, however, difficult to use this technology out of this scenario. On the one hand, the texts have to contain natural language, as the approach does not work for general texts such as DNA, proteins, music, oriental languages and even some agglutinating languages. On the other hand, the overhead posed by considering the set of words as the source symbols is alleviated only for very large files (10 MB or more). Hence, the approach is not well suited to compress individual files that can be independently stored, managed, and transferred, but to a well-organized text collection with a strict control that maintains a centralized vocabulary upon insertions, deletions and updates of the files in the collection. This is, for example, the model of *glimpse* [23]. In this work we aim at a more oblivious method where files can be managed independently.

Several other approaches have been proposed to search texts compressed under different formats, some existing and some specifically designed for searching. Some examples are: different variations of Byte-Pair encoding [22, 36], classical Huffman encoding [25], modified variants of Ziv-Lempel [31], and even general systems that abstract many formats [17]. A few of these approaches are good in practice, in particular a Boyer-Moore based strategy over Byte-Pair encoding [36]. These approaches are interesting. However, their main weakness is that in practice most people use Ziv-Lempel compression methods, and this makes up a barrier for the general adoption of these methods.

Searching Ziv-Lempel compressed texts is, however, rather more complex. The compression is based on finding repetitions in the text and replacing them with references to previous occurrences. The text is parsed as a sequence of "blocks", each of which is built by referencing previous blocks. Hence the pattern can appear in different forms across the compressed text, possibly split into two or more blocks. In LZ78/LZW the blocks can only be a previous block plus one character, while in LZ77 they can be any previous text substring.

The first algorithm to search Ziv-Lempel compressed text [4] is able of searching for single patterns in the LZ78 format. It was later extended to search for multiple patterns on LZW [18]. These algorithms have good worst-case complexities but are rather theoretical. Algorithms with a more practical flavor, based on bit parallelism, were proposed later for LZ78/LZW [31, 19]. Other algorithms for different specific search problems over LZ78/LZW have been presented [12, 16, 28].

Searching LZ77 compressed text has been even harder. The only search technique [10] is a randomized algorithm to determine whether a pattern is present or not in the text. Later studies [31] gave more evidence that LZ77 is difficult to handle.

Note that Boyer-Moore techniques, which have been successful in other formats, had not been applied to Ziv-Lempel compression. This was done for the first time in the earlier version of this work [34]. In that paper it was shown that the Boyer-Moore approach was superior to previous techniques. All those implementations were carried out over a simulated compression format, for simplicity. Our aim in this paper is to describe and extend those Boyer-Moore techniques, and show that they can be implemented over a real LZW compression format (Unix *compress*) to yield an efficient *grep*-like compressed text search tool that can be easily and widely used.

[1] The size of the compressed text as a percentage of the uncompressed text.

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 thereof. If the pointer takes less space than the string it is replacing, compression is obtained. Different variants over this type of compression exist [5]. We are particularly interested in the LZ78/LZW format, which we describe in depth.

The Ziv-Lempel compression algorithm of 1978 (usually named LZ78 [44]) 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 (that is, 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 $(1, s)$.

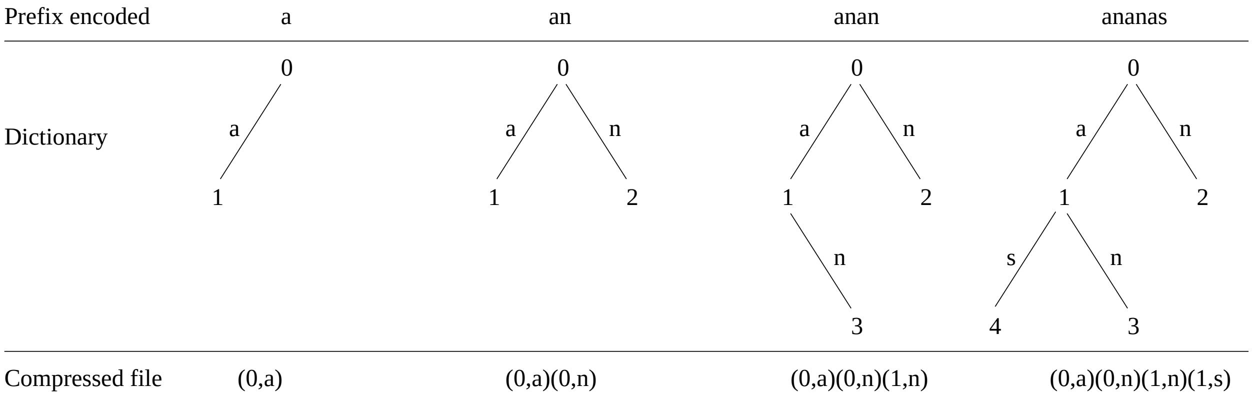


Figure 1: Compression of the word *ananas* with the algorithm LZ78.

The compression algorithm efficient in practice if the dictionary is stored as a trie data structure, 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 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.

Let us detail a bit the decompression process. We read block $b_r = (b, c)$, so we know that the last character of block b_r is c. Now we go to our stored block $b = (b', c')$ and then know that the next-to-last character of the block is c'. Now we go to the stored block $b' = (b'', c'')$ and know that the character preceding c' is c'', and so on until we reach block b_0 and we have found all the characters of the block. We will refer to the sequence $b_r, b, b', b'' \ldots$ as a *referencing chain.*

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 [24, 11]. A particularly interesting variant is from Welch, called LZW [39]. In this case, the extra character (second element of the pair) is not coded, but it is taken as the first character of the next block (the dictionary is started with one block per character). LZW is used by Unix's *Compress* program. Figure 2 shows the LZW compression of the word *ananas.*

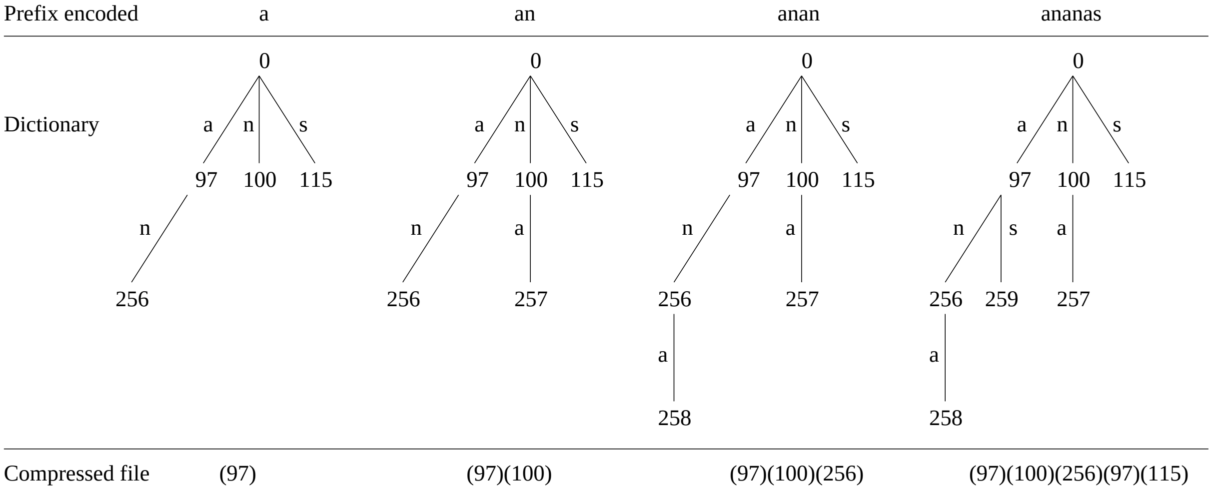


Figure 2: Compression of the word *ananas* with the algorithm LZW.

In this paper we focus on LZW. However, the techniques are easily translated from/to LZ78, as these are just coding variants. The final character of LZ78, which is implicit in LZW, can be readily obtained by keeping count of the first character of each block (which is copied directly from the referenced block) and then looking at the first character of the next block.

3.2 Character-Skipping String Matching Algorithms

There are several string matching algorithms able of skipping text positions without actually inspecting them. These are actually the fastest algorithms. In practice, the best algorithms come from two families: Boyer-Moore and Backward-DAWG-Matching algorithms.

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 usually 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

[14] uses the m-th window character and Sunday [37] 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 [6] 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 with a previous pattern area. This requires a table of size m that for each pattern position gives that last occurrence of $P_{i \ldots m}$ in $P_{1 \ldots m-1}$. This is used in the original Boyer and Moore method [6].

The case of multiple patterns is handled by building d tables that permit the minimum jump over the set of all the patterns. This table is usually built over more than one character to enable larger shifts [41]. An alternative is to extend the original Boyer-Moore method [7]. A trie is built over the set of reversed patterns, and instead of comparing right-to-left the text window and the pattern, the window characters are used to enter the trie of patterns. The trie nodes have the precomputed shifts.

The Backward-DAWG-Matching (BDM) family gives better algorithms than the BM family when the pattern is long or the alphabet is small. Some prominent members of this family are BDM itself [8], BNDM [32] and BOM [2]. Multipattern versions of these algorithms include MultiBDM [9] and SBOM [33].

Currently, the fastest single-pattern matching algorithms are Horspool and BOM. In the case of multipattern matching, the best in practice are the method of Wu and Manber (WM) [41] and SBOM.

4 Decompressing and Searching

Before getting into the direct search algorithms developed, let us study in some depth which would be the best option if we decided to decompress the text and then search it. This would be our main competitor.

Our experiments, in the whole paper, measure user plus system time over an Intel PIV 1.6 GHz, with 256 MB RAM and local disk, running Linux RedHat 8.0, compiling using gcc 3.2 and full optimization. We have used two 10 MB texts: WSJ is English text obtained from 1987 Wall Street Journal articles (from TREC-3 [13], while DNA is Homo Sapiens DNA obtained from Genbank (`www.ncbi.nlm.nih.gov`). Patterns were randomly chosen from the text, averaging over 100 patterns.

WSJ was compressed to 38.75% of its original size using *compress* and 33.57% using *gzip*. DNA, on the other hand, was compressed to 27.91% of its original size using *compress* and 30.43% with *gzip*.

Note that, if we are willing to apply a decompress-then-search approach, then there is no reason to use an LZ78/LZW format. Rather, LZ77 is faster to decompress (although for some types of text LZW compresses better). Since our goal is to provide a free tool, we have chosen *gzip/gunzip* as our LZ77 compressor (*gzip* produces files with `.gz` extension). Likewise, we have chosen *compress/uncompress* as our LZW compressor (*compress* produces files with `.Z` extension). The source code of these two programs are freely available, and they are the most popular in the Unix world. Our aim is to modify the decompressor so that it performs pattern matching instead of decompression.

There exist several versions of *uncompress*, all of which handle the same `.Z` format. Moreover, *gunzip* is able of decompressing this format as well. Interestingly, among all the variants we found, *gunzip* was the fastest. The reason is that *uncompress* obtains the characters of a block in reverse order, and then has to output them reversed again so as to get the correct order. On the other hand, *gunzip* obtains them in reverse order and stores them in reverse order, so the output can be done directly with a machine instruction.

In order to uncompress LZ77, on the other hand, *gunzip* stores the text already uncompressed and, given a new block, copies the referenced text substring at the end of the uncompressed text. This is faster than decompressing LZW format.

We have modified the decompression code of *gunzip*, both for LZW and for LZ77. These are called DW and D77 in our experiments. Over each format, we have implemented different plain text search algorithms over the uncompressed buffer. Unlike a usual decompression work, we do not write the buffer to the output, but rather use it for searching.

4.1 Single Patterns

We have implemented the best two search algorithms we are aware of: BM-Horspool [14] and BOM [2], so as to obtain techniques DW-BM, D77-BM, DW-BOM and D77-BOM.

Besides implementing different search algorithms, we have also included some alternatives that evaluate how good can these schemes possibly be: DW-decode (just decoding the LZW compressed file and following the referencing chains), DW-nosearch (just uncompressing the LZW file in memory, without searching), and D77-nosearch (just uncompressing the LZ77 file in memory, without searching). Note that DW-decode is a lower bound to any decompress-then-search algorithm on LZW compressed text, DW-nosearch is a lower bound to any such algorithm that writes the uncompressed text before searching it, and D77-nosearch is a lower bound to any algorithm that searches LZ77 compressed text.

Figure 3 shows how the Boyer-Moore approach performs. On English text, D77-BM is a good choice, which gets in fact close to its lower bound D77-nosearch. It does not perform so well on DNA, as expected given the small alphabet size.

Figure 4 measures the performance of the BOM approach. As it can be seen, D77-BOM performs very well in either type of text, getting very close to its lower bound D77-nosearch. DW-BOM is also close to DW-nosearch, but both are not competitive against their LZ77-based counterparts.

We have also tried KMP algorithm [20] over LZW, obtaining DW-KMP. Although KMP algorithm is far from being competitive, it examines text characters in a forward-only fashion, always advancing. This is interesting because there is no need to actually write the uncompressed characters in the buffer. On the other hand, LZ77 needs to write the buffer for uncompressing, so there was no advantage in combining it with KMP. Figure 5 evaluates this alternative. Even when on WSJ we surpass the DW-nosearch barrier (and hence all its LZW-based competitors), it is clear that this alternative will not be competitive against its LZ77-based counterparts.

Finally, in order to simulate the behavior of *zgrep*, we also implemented DW-grep and D77-grep,

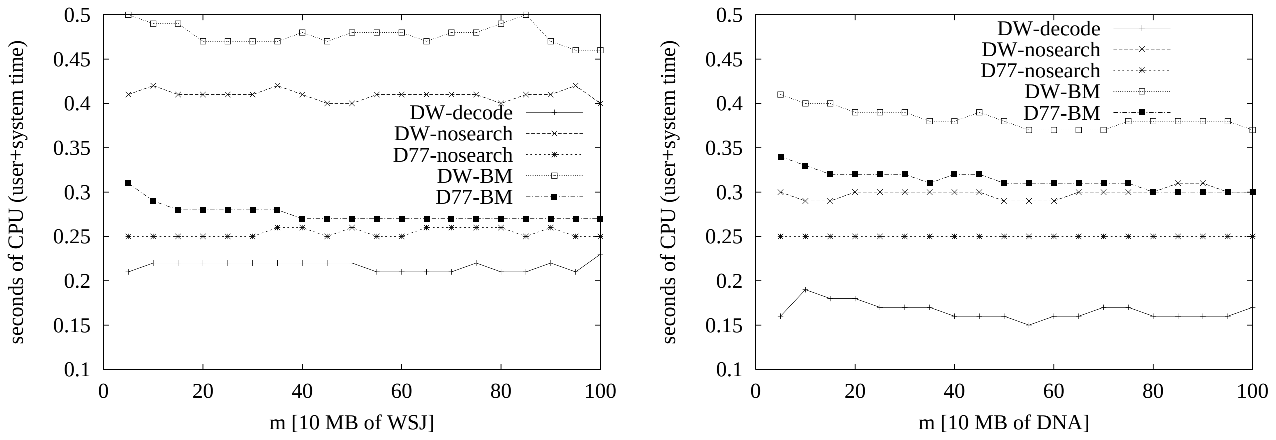


Figure 3: Decompress-then-search approaches based on Boyer-Moore.

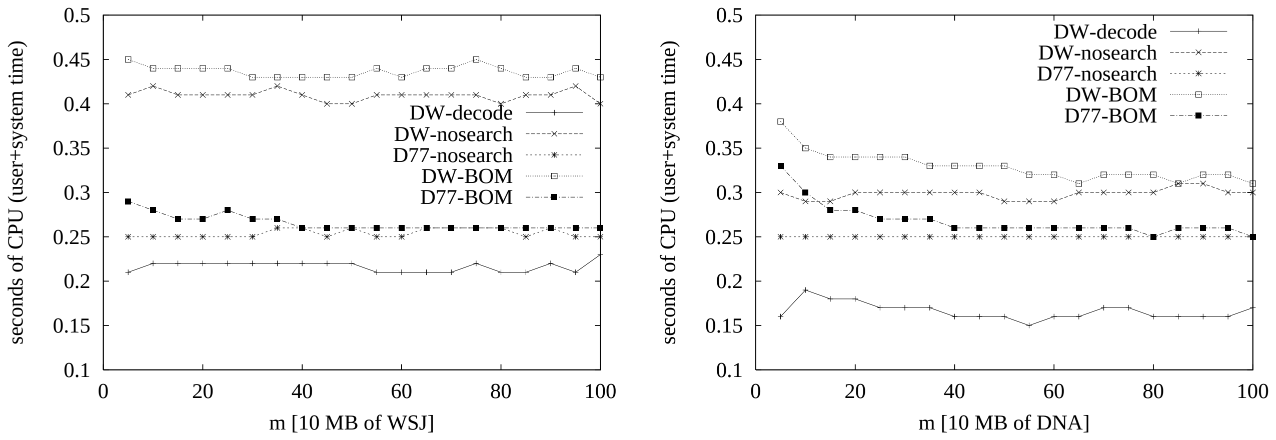


Figure 4: Decompress-then-search approaches based on BOM.

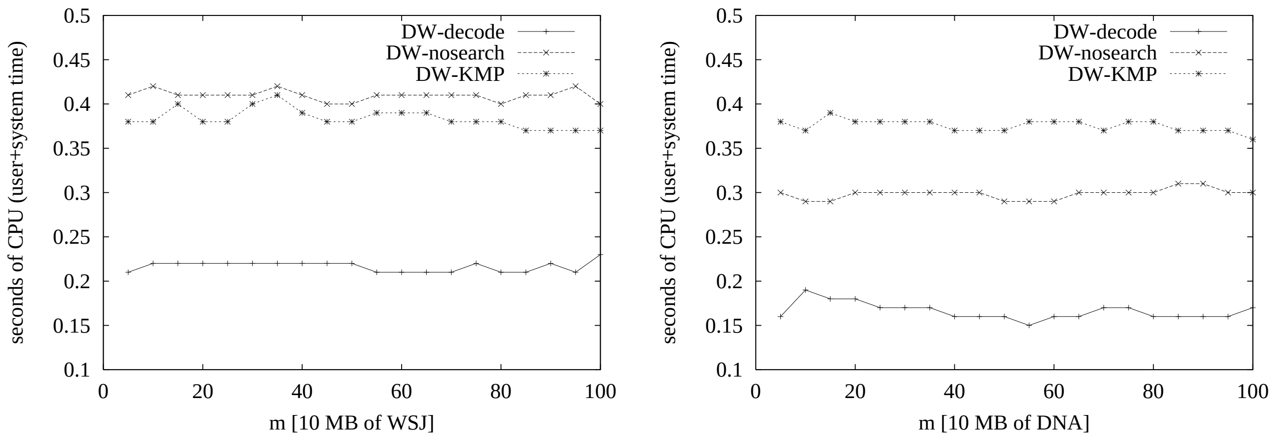


Figure 5: Decompress-then-search approaches based on KMP, on LZW.

which output the buffer and use it as an input to *grep*. We used, however, *agrep* [42] rather than *GNU grep*, as it was faster.

Figure 6 shows that this alternative, despite being popular because it is easily implemented, is far from competitive against the choices we have just reviewed.

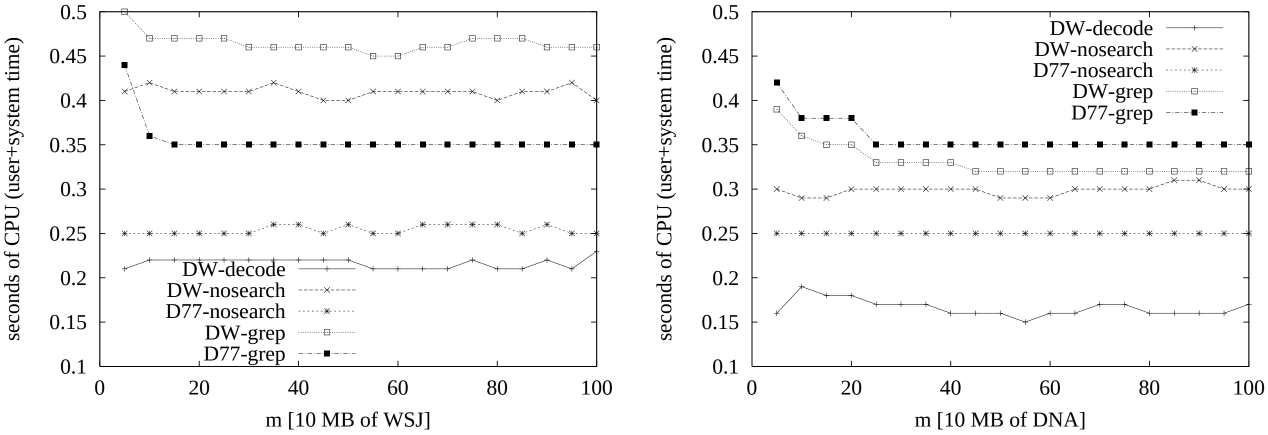


Figure 6: Decompress-then-search approaches based on *grep*.

Finally, Figure 7 compares the best approach of each kind. As it can be seen, D77-BOM is always the best decompress-then-search choice. It is only slightly over its lower bound, D77-nosearch (and usually below DW-nosearch). It has also beaten DW-KMP, which on WSJ improves upon DW-nosearch. However, it is clearly slower than DW-decode, which means that there is hope for improving upon it with a direct search algorithm.

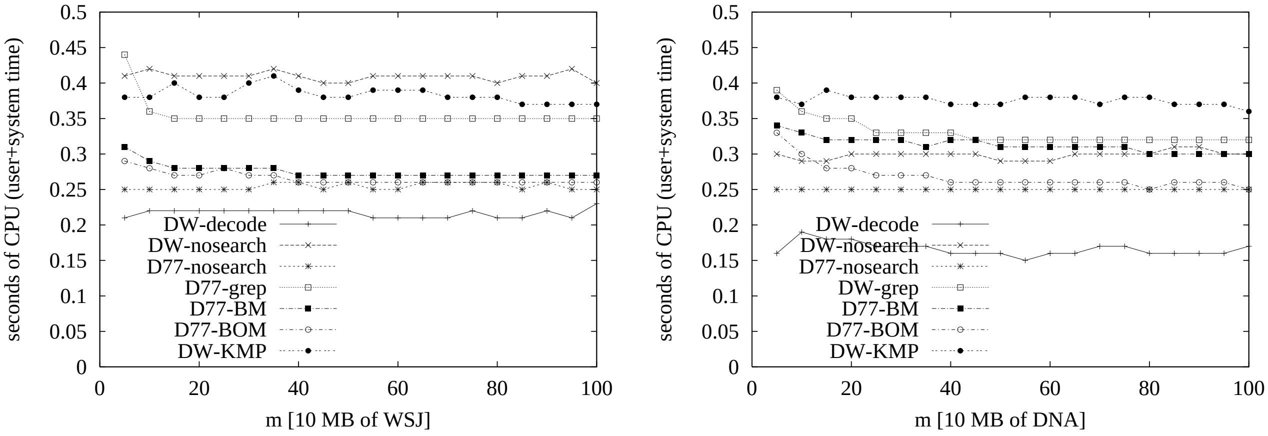


Figure 7: Comparison among the best decompress-then-search approaches.

When our *LZgrep* search tool has to search LZ77-compressed text, it always chooses D77-BOM.

4.2 Multiple Patterns

We have considered the cases of searching for $r = 10$, 100 and 1000 patterns simultaneously. As this time the search times are much higher, the overhead posed by the decompression of different formats is less important than in the case of a single pattern.

In this case the best search algorithms are WM [41] and SBOM [33]. So we have created decompress-then-search variants called DW-WM, D77-WM, DW-SBOM and D77-SBOM. Figures 8 and 9 show their performance, where it can be seen that the LZ77-based versions are better.

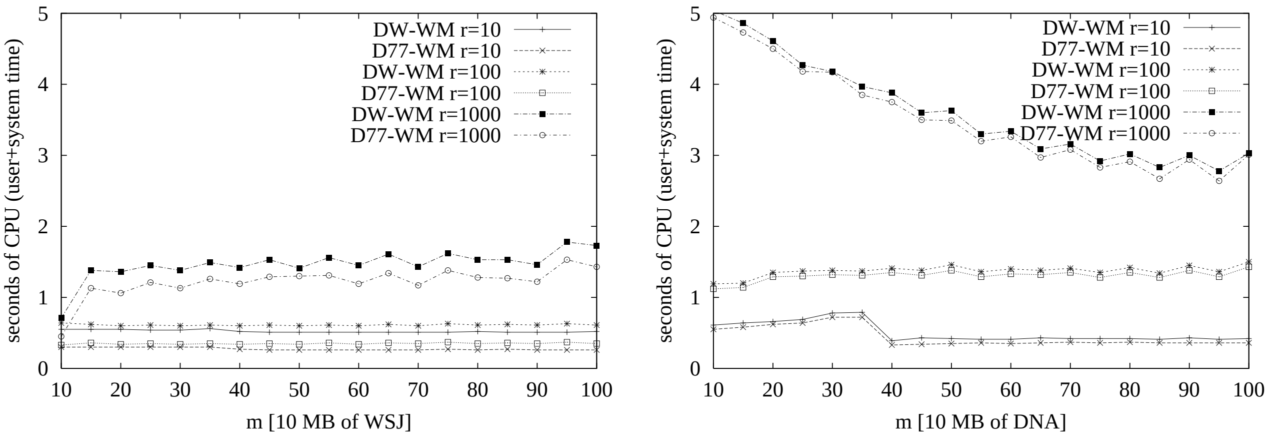


Figure 8: Decompress-then-search based on WM.

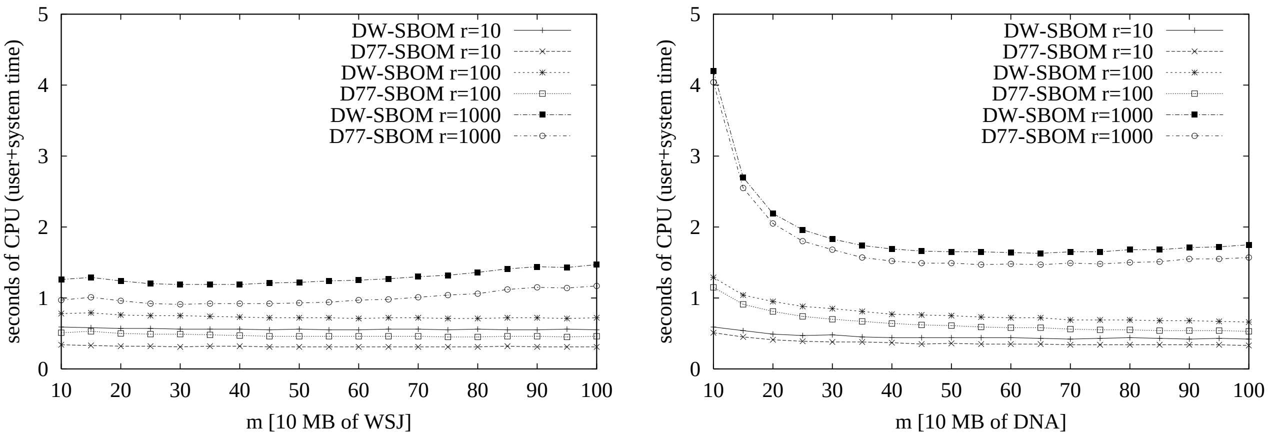


Figure 9: Decompress-then-search based on SBOM.

To simulate the behavior of *zgrep*, we use again *agrep* instead of *GNU grep*. However, since *agrep* also uses WM algorithm, the difference between D-grep and D-WM is just the use of a pipe in the first case versus a direct memory buffer in the second. Figure 10 shows its behavior, showing that the LZ77-based versions are better. Since the underlying algorithms of D77-grep and D77-WM are the same, we prefer to choose one right now. Figure 11 shows that D77-WM is always superior, as expected.

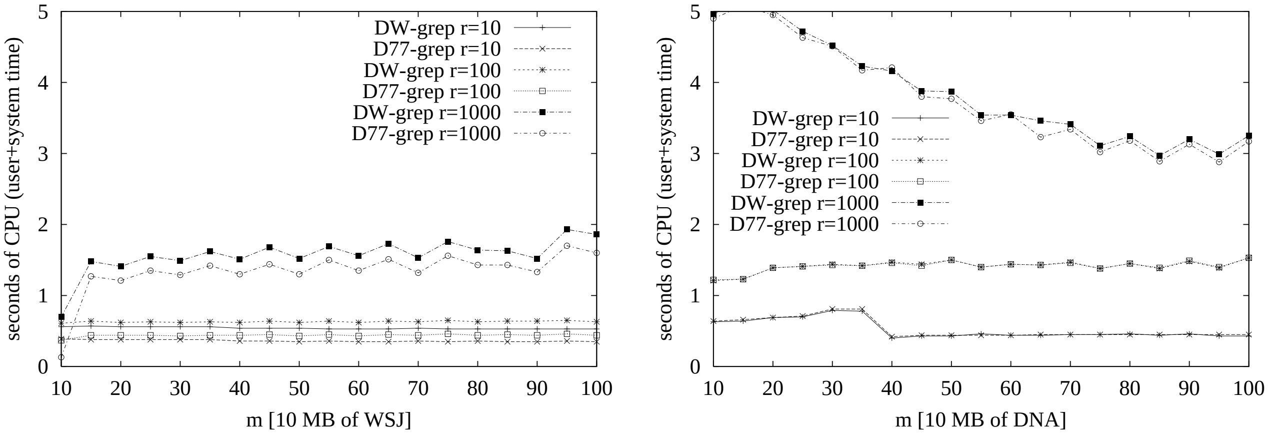


Figure 10: Decompress-then-search based on *grep*.

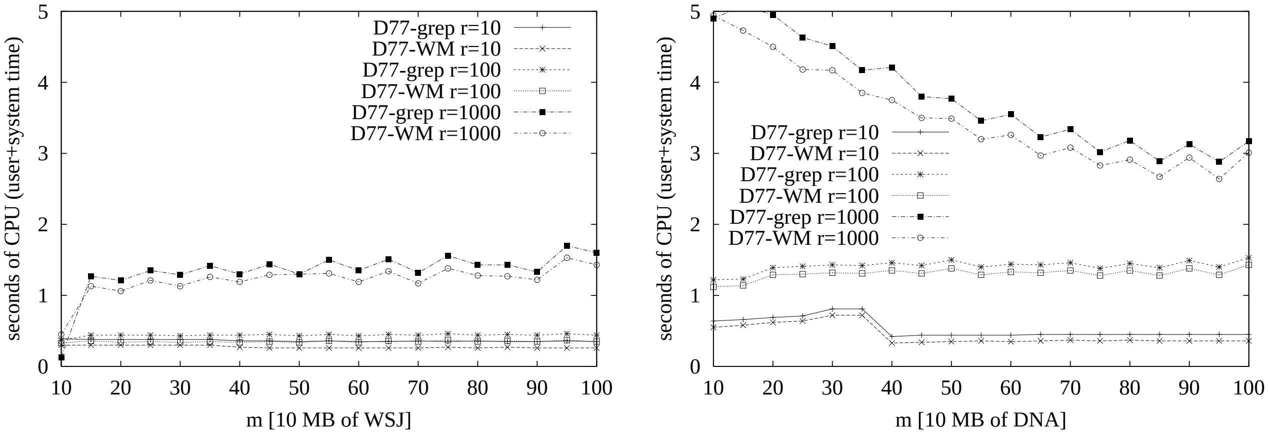


Figure 11: Decompress-then-search based on WM versus *grep*.

For the same reason we tried KMP for single patterns, we have considered Aho-Corasick (AC) [1] as an interesting option to apply over LZW decompression, as it does not require writing the uncompressed text in memory. The algorithm is called DW-AC.

Figure 12 compares all the search algorithms. D77-WM is the best on WSJ for $r = 10$ and 100, but for $r = 1000$ D77-BOM is almost always the best. On DNA, on the other hand, the best performace is almost always disputed between D77-BOM and DW-AC.

As it can be seen, there is not a clear simple winner as in the case of single patterns. When our search tool *LZgrep* has to search LZ77-compressed text for multiple patterns, it uses D77-WM for up to 100 patterns, and D77-SBOM for more.

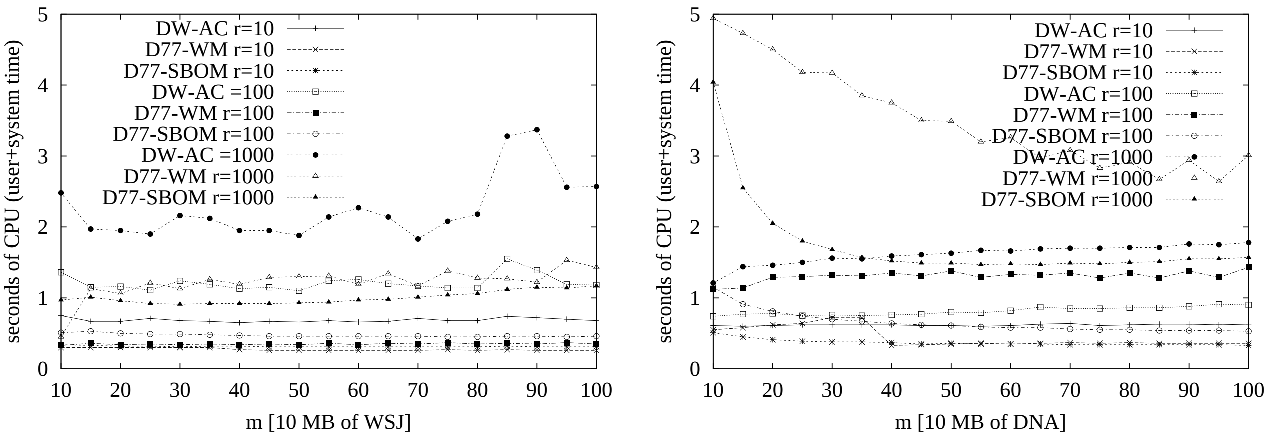


Figure 12: Comparison among the best multipattern decompress-then-search approaches.

5 A Simple Boyer-Moore Technique

Consider Figure 13, where we have plotted a hypothetical window approach to a text compressed using LZ78/LZW. Each LZ78/LZW 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, that is, a text that has to be obtained by resorting to previous referenced blocks (s, then the block referenced by s, and so on).

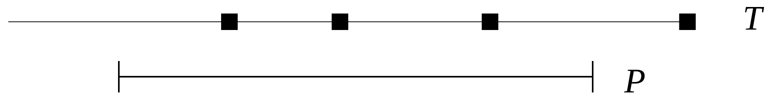


Figure 13: A window approach over LZ78/LZW compressed text. Black boxes are the explicit characters at the end of each block, while the lines are the implicit text that is represented by a reference.

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/LZW 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. 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.

The order in which blocks should be unfolded is not immediate, in particular with respect to the last block. On the one hand, the last block can yield good shifts. On the other hand, it is costly to reach its relevant characters, as it can only be unfolded from right to left. We consider two choices: BM-simple unfolds the blocks right to left but leaves the last block for the end, while BM-simple-lf starts with the last block and then unfolds the others right to left shows the order in which we consider them. Figure 14 illustrates.

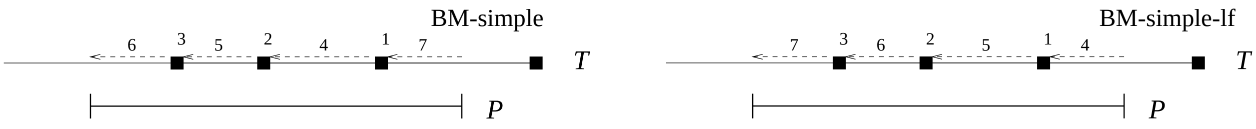


Figure 14: Evaluation orders for the simple algorithm.

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, and its length (the latter is not available in the compressed file but computed on the fly). We also keep the current position in the uncompressed text.

On the other hand, the LZW format of *compress* specifies the maximum number of bits, x, used for a backward reference. Once 2^x blocks have been processed, it still continues generating blocks but these cannot be referenced later. For the same reason, once we surpass the 2^x blocks, we do not store their information during the search until a mark is found in the compressed file indicating the start of a new buffer of blocks.

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-basic" our original version and "BM-simple" this improvement. Also, as we soon show that the use of the skip-loop improves performance, let us assume that BM-simple-lf does use the skip-loop technique.

Figure 15 compares the alternatives based on Boyer-Moore. The skip-loop alternative is always superior. BM-simple-lf is slightly better for intermediate pattern lengths over WSJ, otherwise BM-simple is better. So, for the rest of the paper, we retain BM-simple and BM-simple-lf on WSJ, and just BM-simple on DNA.

Note that, even in the best case, our algorithms have to scan all the text blocks. However, they

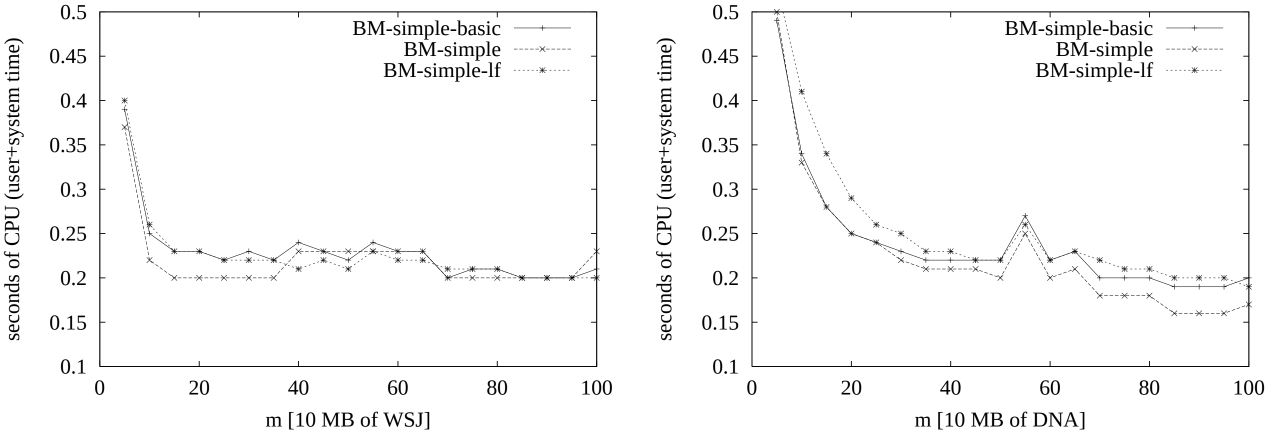


Figure 15: Direct searching with simple Boyer-Moore approaches.

are faster in practice than previous algorithms and, as shown in Figure 16, competitive against the best decompress-then-search approaches. Moreover, note that in most cases we have surpassed the D77-nosearch and even the DW-decode barriers, which shows that no decompress-then-search algorithm can be faster than this direct search approach. This shows that direct searching can be faster than the naive approach.

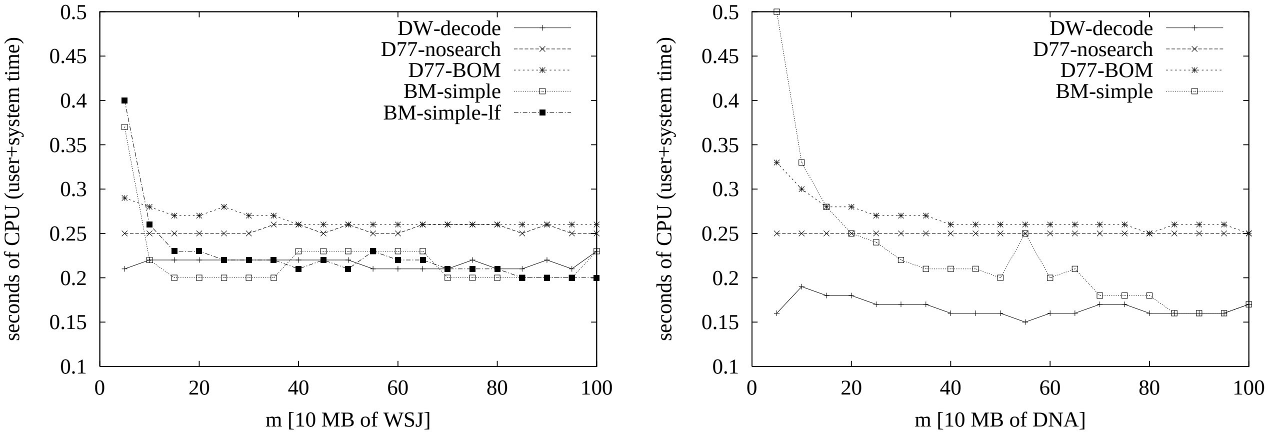


Figure 16: Comparison between decompress-then-search and direct searching with simple Boyer-Moore approaches.

6 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 (for example, DNA). Multicharacter techniques, consisting in shifting by q-tuples of characters instead of one character, have been successfully applied to search uncompressed DNA [35]. 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 $x_1 x_2 \dots x_m$ we try to shift using the q-grams $x_{m-q+1} \dots x_m$, then $x_{m-q} \dots x_{m-1}$, etc. until $x_1 \dots x_q$. 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/LZW 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 $0 \dots N-1$ 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, called “BM-multichar”, using $q = 4$ (which is appropriate to store the q-gram in a word of our machine), and $N = 1,017$, which was experimentally found to be appropriate. We use the skip-loop improvement. Figure 17 compares it against the best decompress-then-search approaches. As it can be seen, BM-multichar is attractive on DNA text, where it is not only faster by far than any naive approach (except for $m = 5$), but also surpasses the DW-decode barrier at times (and the D77-nosearch barrier for $m \geq 15$). So it will not be beaten by any decompress-then-search approach.

7 Shifting by Complete Blocks

Despite that we have obtained good results with BM-multichar, we present now a more elegant technique that is especially suited to the LZW compression format.

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

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

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

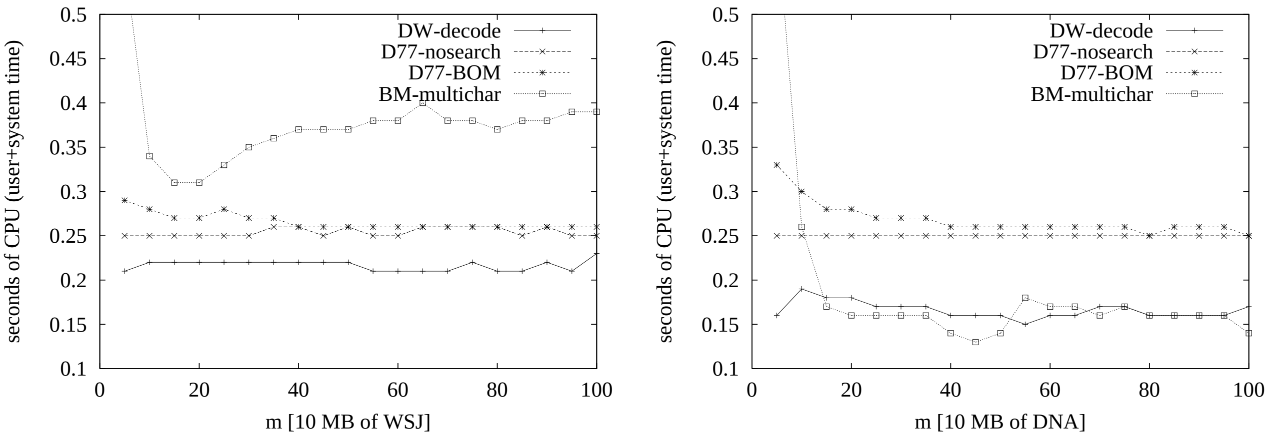


Figure 17: Comparison between decompress-then-search and direct searching with the multicharacter Boyer-Moore approach.

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 by considering $last(s)$, that is, the last position where the referenced block appears in P. We check whether $P_{last(s)+1} = c$, 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 just $O(m^2\sigma)$ extra work across all the search, since text blocks are all different and there are only $O(m^2)$ different pattern substrings. Hence, there are only $O(m^2)$ possible values for $last(r)$ in the above process, and hence only $O(m^2\sigma)$ possible block contents for which there will be some $last(r)$ value. For each such value, we can work $O(m)$ using the referencing chain, as long as the shorter and shorter strings we find occur in the pattern. But each such occurrence is a pattern substring that cannot be a $last(r)$ answer, as there is a longer substring later that contains it. Therefore any pattern substring can either (i) be a last occurrence and hence a possible $last(r)$ value, or (ii) have another repetition later and hence be a possible intermediate step for at most σ subsequent repetitions, but not both. Hence adding over all the substrings we have $O(m^2\sigma)$ time overall.

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 5) this is solved by computing $B(i, c)$, that is, the last occurrence of c inside P *before position* i. This may require too much effort in our case. We note that we can 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 18 illustrates. For this part we can work more than $O(m^2\sigma)$ overall in some unlikely cases.

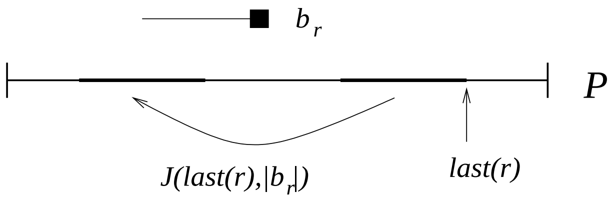


Figure 18: Using the whole LZ78/LZW block to shift the window. If its last occurrence in P is ahead, we use J until finding the adequate occurrence.

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. We do not attempt the alternative of unfolding the latter block first, because in this case the internal blocks are much cheaper to process than the final block. Moreover, this idea was not good on DNA in BM-simple (Section 5), and we basically aim at DNA searching in this section.

As before, we read blocks until surpass the window and then try to shift. This method is called "BM-blocks-basic". The alternative of using a skip-loop is called "BM-blocks".

Let us now review two possible improvements that one can imagine over the above scheme. Both turn out to be impractical, but it is of some value to know that these ideas do not yield good results.

The first adapts the match heuristic of Boyer-Moore in the spirit of shifting using a variable number of characters. In this case 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 $P_{i\ldots j}$ and P_{i-1} has mismatched. The C table is defined very similarly to J, and it is used when internal characters are compared, since in that case a contiguous portion of P has been inspected (the situation when comparing the explicit characters is much more complex since an arbitrary subset of positions have been compared). This method, called "BM-complete" in our earlier paper [34], yielded poor results.

The second aims at eliminating the overhead of potentially accessing table J several times per block. Rather, we precompute the appropriate answers in tables $J1$ and $J2$. $J1(i,\ell,c)$ gives, for pattern substring $P_{i-\ell+1\ldots i}$, the latest previous occurrence in P of the same substring followed by character c. Formally,

$$\begin{array}{rll} J1(i,\ell,c) = & \max(& \{j,\ \ell \le j < i\ \wedge\ P_{j-\ell+1\ldots j} = P_{i-\ell+1\ldots i}\ \wedge\ P_{j+1} = c\} \\ & \cup & \{j,\ 0 \le j < \ell\ \wedge\ P_{1\ldots j} = P_{i-j+1\ldots i}\ \wedge\ P_{j+1} = c\} \\ & \cup & \{-1\}), \end{array}$$

which can be computed in $O(m^2\sigma)$ time. With $J1$ we do not have to use J repeatedly until we find an occurrence followed by a given character. The other case where we access J multiple times is when looking for a previous occurrence of some substring of P because the current one lies after the current block. For this sake we precompute

$$\begin{aligned} J2(i,\ell,p) = \; & \max(\; \{j,\ \ell \leq j \leq p \ \wedge \ P_{j-\ell+1\ldots j} = P_{i-\ell+1\ldots i}\} \\ & \quad \cup \quad \{j,\ 0 \leq j \leq p \ \wedge \ P_{1\ldots j} = P_{i-j+1\ldots i}\}), \end{aligned}$$

so that $J2(i,\ell,p)$ gives the last occurrence of $P_{i-\ell+1\ldots i}$ that ends before position p. $J2$ can be computed in $O(m^3)$ time.

This alternative is called "BM-blocks-2J", and removes the potential of repeated calls to J at the price of $O(m^2(m+\sigma))$ preprocessing time. It is not hard to see that the preprocessing time probably overweights the unlikely repeated calls to J, so it should not be a surprise that this alternative does not improve upon the simpler ones in practice.

Figure 19 compares the different block-based approaches. As it can be seen, BM-blocks is always better. The extra preprocessing time of BM-blocks-2J is more noticeable as the pattern length grows. Hence we will keep only BM-blocks for further comparisons.

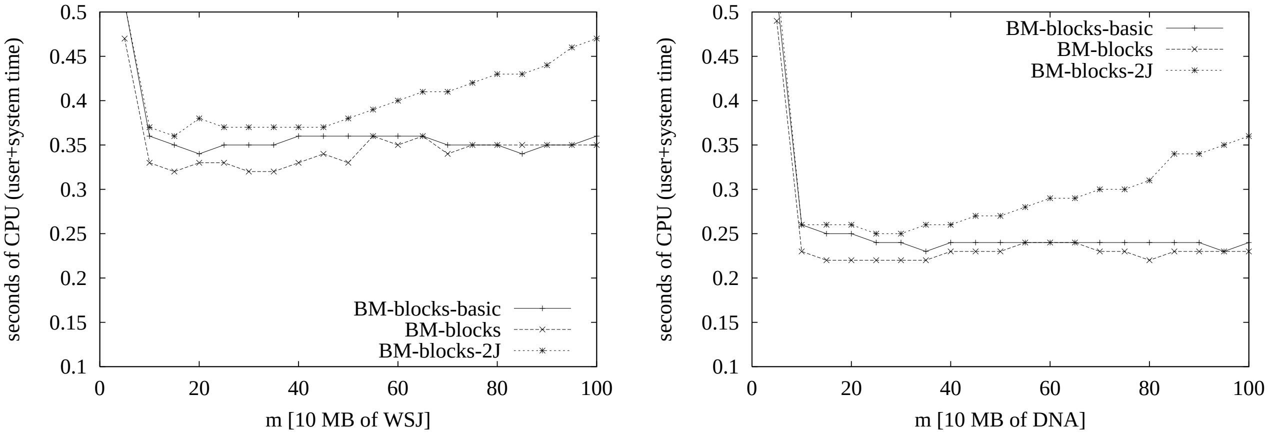


Figure 19: Direct searching with block-based Boyer-Moore approaches.

Figure 20 compares the best decompress-then-search approaches against BM-blocks. It can be seen that BM-blocks is competitive on DNA (except as usual for $m = 5$), beating the D77-nosearch lower bound, but far away from the DW-decode lower bound.

8 Direct Searching for Single Patterns

We compare now the best alternative of each kind, in order to obtain a recommendation on which is the best technique for compressed pattern matching. Alternative direct search algorithms, not based on Boyer-Moore [31, 19] were already shown to be 30% slower than our approach in earlier work [34]. Those experiments were performed on non-standard compression formats that resembled LZ78. Therefore, we do not believe that it is worth porting non-Boyer-Moore algorithms to the format of *compress*, as we already know that these are not competitive.

Figure 21 shows the results. On English text, the best choice is BM-simple or BM-simple-lf, although BM-simple is usually better. This algorithm takes 8%–20% less time than the best

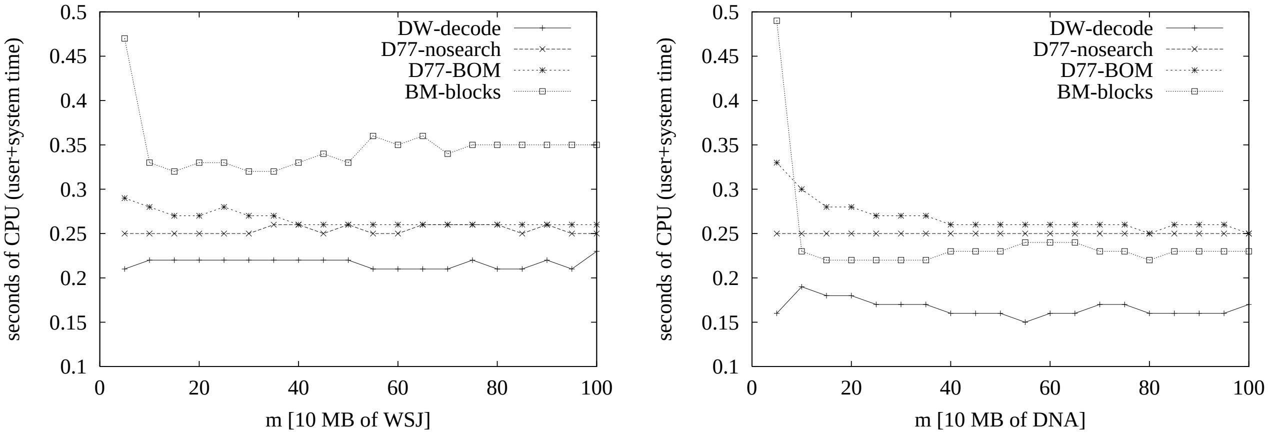


Figure 20: Comparison between decompress-then-search and direct searching with block-based Boyer-Moore approaches.

implementation of a decompress-then-search approach (which is already much better than *zgrep*). Other more sophisticated search techniques do not work well on English text, being even worse than D77-BOM. The latter is also unbeaten for very short patterns ($m = 5$).

On DNA, on the other hand, the alphabet is much smaller and the simple BM techniques do not perform well except for rather long patterns $m \geq 70$. In this case the best is, almost always by far, BM-multichar. BM-blocks, although elegant, is only interesting in very special cases. However, it is interesting to mention that BM-blocks is the fastest for $m = 10$. Overall, our techniques take 30%–50% less time than the best decompress-then-search approach. Again, D77-BOM is by far unbeaten for $m = 5$.

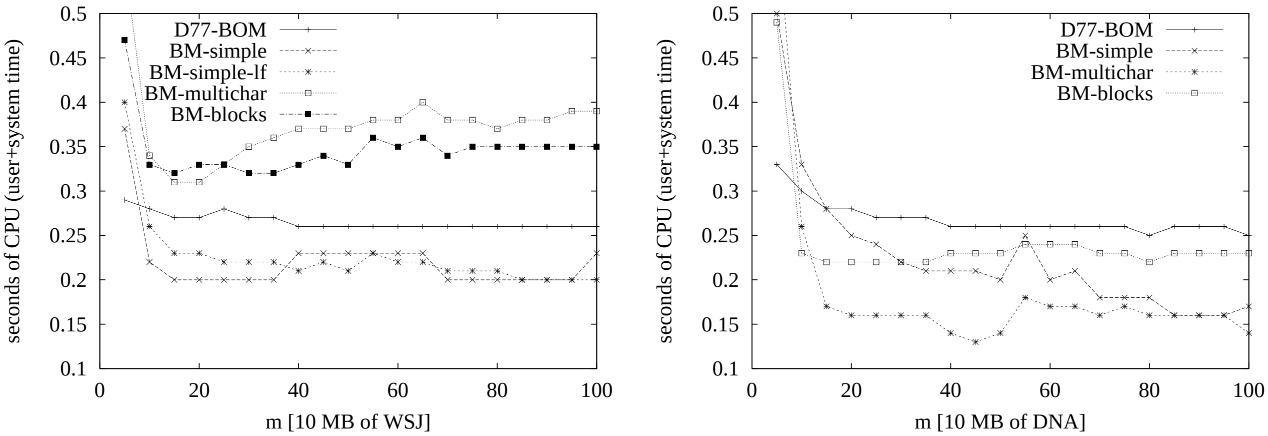


Figure 21: Comparison between decompress-then-search and direct searching using the best approach of each kind.

Our *LZgrep* search tool cannot know in advance which type of text it faces. So the default is to use BM-simple, which should work well in the most common cases. Users can override this default choice if they know the type of text. Note also that which is the best algorithm depends on the machine, for example, in our previous work [34], BM-blocks was better than BM-multichar on DNA text.

The case $m = 5$ deserves a special mention. None of our direct search algorithms have worked well on it, being D77-BOM by far the best choice. The reason is that 5 is smaller than the length of most blocks (the average block length is 10–12). Therefore we must "unroll" many blocks because in many cases the window is completely inside a block. For patterns of length 10 or more there is always at least one explicit character inside the text window.

9 Multipattern Matching

Let us now consider the case where we want to search for several patterns $P^1, \ldots, P^r$ simultaneously, in order to report all their occurrences. We show how the algorithms developed for single patterns can be extended to handle multiple patterns.

As shown in Section 4.2, there is not a single best decompress-then-search algorithm to compare against. Hence we have opted for creating a ficticious algorithm called "D-BEST", which is the best over the available algorithms for each case, shown in Section 4.2.

9.1 Simple Boyer-Moore

The first problem when trying to extend the window approach of Section 5 is that different patterns may have different lengths. So let us align them to the right and choose the window length as that of the shortest pattern, as illustrated in Figure 22.

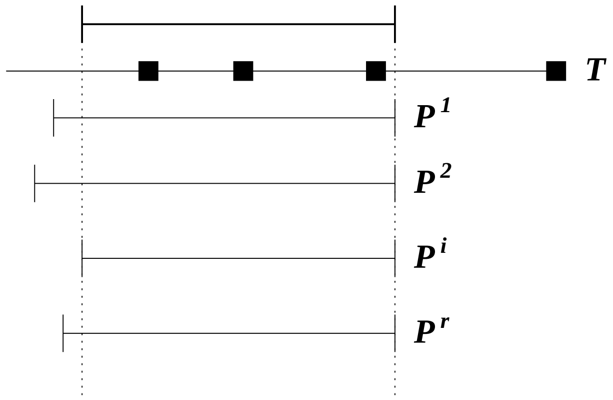


Figure 22: A window approach for multipattern matching over LZ78/LZW compressed text. It is a generalization of Figure 13.

The simplest way to extend BM-simple to handle multiple patterns is to define table $B(i, c)$ so that it takes into account all the patterns, that is

$$B(i, c) = \min_{k \in 1 \ldots r} \min(\{i\} \cup \{i - j,\ 1 \leq j \leq i\ \wedge\ P^k_j = c\})$$

where for simplicity we consider patterns truncated to the window length in this formula.

This way, $B(i,c)$ lets us shift the window by the minimum amount permitted among the patterns we search for, which guarantees that no occurrence will be missed. So we read the characters as in Section 5, until either $B(i,c) \neq 0$ for some character c read at window position i, or we read all the window. In the latter case, we must still check all our patterns against the text window one by one in order to report occurrences, because (1) we may have left out some parts of the patterns due to truncation, and (2) table B gives minimum shifts over the set of patterns, so no occurrence of any particular pattern is guaranteed. Let us call "BM-simple" this approach, as it is the most direct descendant of the single-pattern version.

An efficiency problem of BM-simple is that, as explained, even if we have read a sequence of characters that do not match any of our patterns, it might be that table B does not let us shift the window. Imagine for example that we have patterns *abab* and *baba*, and read any text window formed by a combination of a's and b's, say *bbaa*. Since $B(i,a) = B(i,b) = 0$ for any i, we will always verify those text windows.

We study three alternatives to alleviate this problem. They are all based on precomputing a table

$$M(i,c) = [P_i^1 = c]\ [P_i^2 = c]\ \ldots [P_i^r = c]$$

where $[P_i^k = c]$ is a bit with value 1 if the logical condition holds and zero otherwise. Hence, $M(i,c)$ is a bit mask whose k-th bit tells whether P^r has a character c at position i. We also maintain a bit mask of r bits, $mask$, whose k-th bit tells whether P^k has matched the current text window up to now. Hence $mask$ is initialized as $11\ldots1$, and after we read character c at window position i we set $mask \leftarrow mask \& M(i,c)$, where "&" is the bitwise "and".

The three strategies work as follows. BM-simple-2 updates $mask$ every time $B(i,c) = 0$ (that is, as long as the window cannot be shifted). If at some moment $mask = 00\ldots0$ (all zeros), then we know the window can be shifted, although we do not know by how much. Hence we shift the window by one position and restart the process. BM-simple-3 also updates $mask$, but it does not shift the window prematurely. Rather, if we reach the end of the window without shifting, we use $mask$ to determine which patterns must be checked (that is, those P^k such that the k-th bit of $mask$ is 1). Moreover, only the part of the patterns that lies outside the window must be checked, as the characters inside the window are known to match P^k. Finally, BM-simple-4 does not use $mask$ initially, but rather after reaching the beginning of the window without obtaining any shift. At this point the window is retraversed computing $mask$ and the surviving patterns are verified.

Figure 23 compares the different alternatives. We considered only up to 100 patterns, because for $r = 1,000$ the performance was not competitive at all. As it can be seen, plain BM-simple is the best choice on WSJ and for long DNA patterns ($m > 40$ for 10 patterns), while BM-simple-2 was by far better for shorter DNA patterns.

Figure 24 compares the best BM-simple approaches against the best decompress-then-search technique. As it can be seen, BM-simple is competitive on WSJ for $r = 10$ patterns, and also for $r = 100$ patterns if these are long enough ($m > 70$). On DNA, on the other hand, no BM-simple variant is useful.

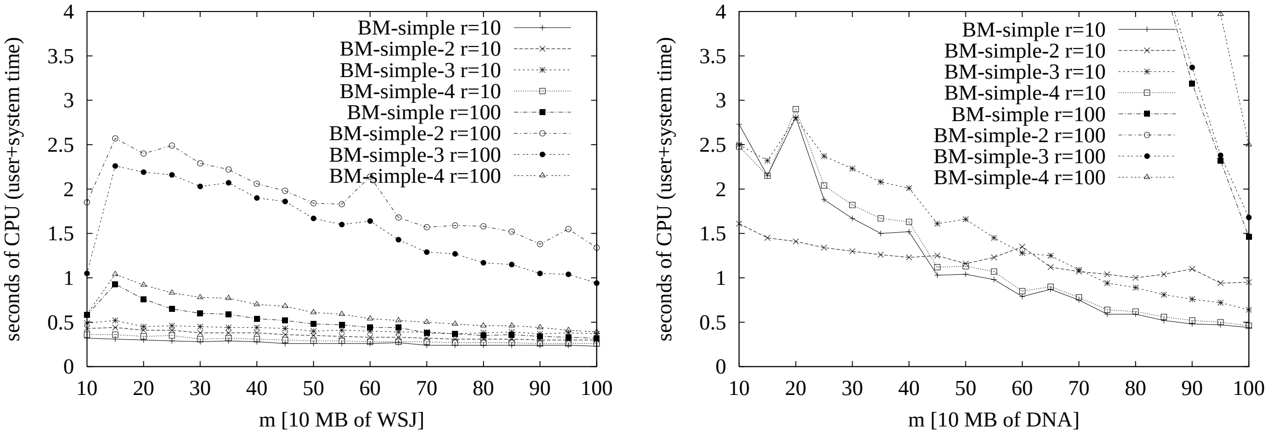


Figure 23: Direct multipattern searching with simple Boyer-Moore approaches.

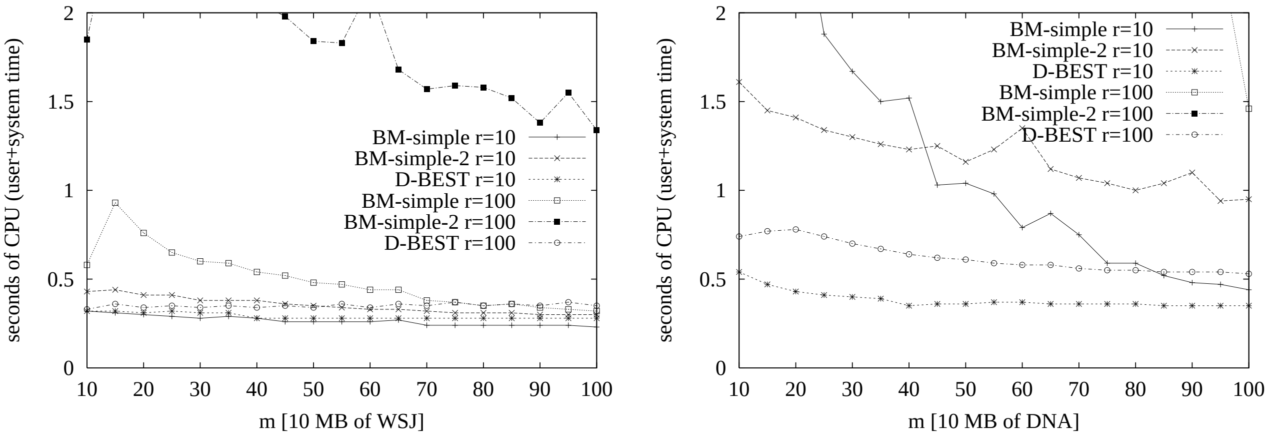


Figure 24: Comparison between multipattern decompress-then-search and direct searching with simple Boyer-Moore approaches.

9.2 Multicharacter Boyer-Moore

We adapt the idea of shifting using q-grams rather than simple characters, as in Section 6. We still use $q = 4$ and $N = 1,017$. With more patterns we also tried larger N values (and also larger q), but we obtained no improvement in doing that, up to $r = 1,000$.

We have again two versions, the basic BM-multichar, and also BM-multichar-2, which corresponds to the use of *mask* just as in BM-simple-2. Figure 25 shows that the use of q-grams improved the time to search for many patterns in both texts. It can also be seen that, except for the (anyway not competitive) case of $r = 100$ and $m \leq 20$ on DNA, BM-multichar is always better than BM-multichar-2. So we will preserve only BM-multichar for future experiments.

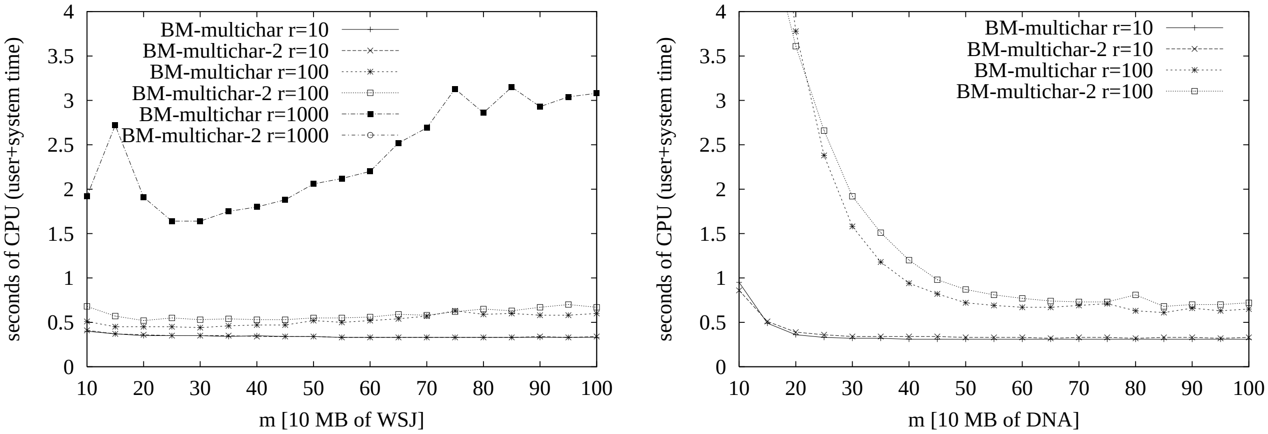


Figure 25: Direct multipattern searching with multicharacter Boyer-Moore approaches.

Figure 26 compares BM-multichar against the best decompress-then-search approaches. This time we improve upon the naive algorithm only on DNA, for $r = 10$ patterns and $m \geq 15$.

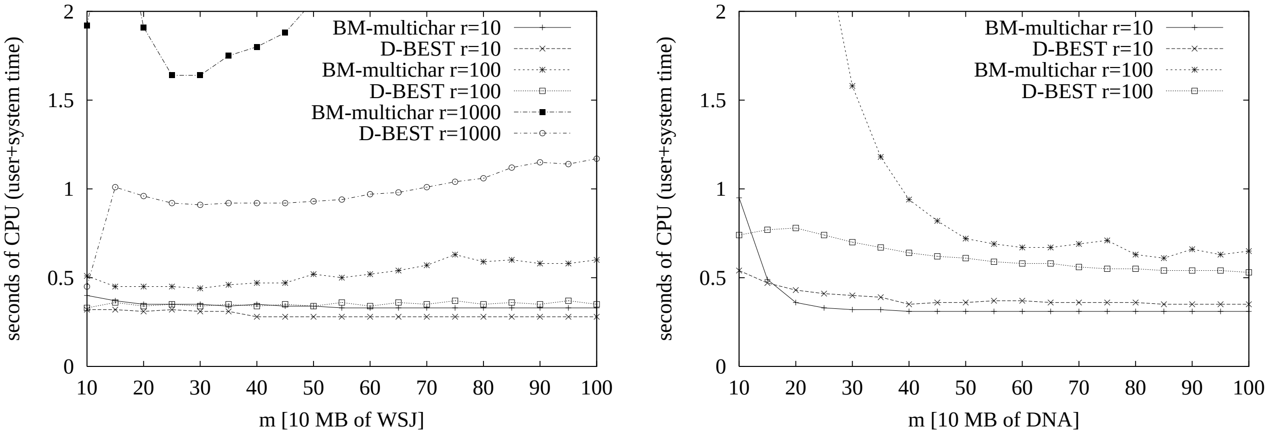


Figure 26: Comparison between multipattern decompress-then-search and direct searching with multicharacter Boyer-Moore approaches.

9.3 Shifting by Complete Blocks

We try to adapt the idea of Section 7 to multiple patterns. However, this turns out to be rather difficult. With a single pattern P we can compute the last occurrence of a text block inside P, by considering the candidate position given by the referenced block and then iterating using table $J(i, \ell)$ until finding that last occurrence. Then, if the last occurrence happens to be ahead the block, $J(i, \ell)$ is used again to find previous occurrences.

With multiple patterns, the last occurrence of the referenced block is still a single window position, but it may appear in several patterns at that same position. Finding which of them can be extended by the last character of the current block can be a time-consuming task. The same can be said about moving from such a set of positions to the "previous" set of positions, which might also appear in several patterns.

What we need is a data structure where every different substring of every pattern is represented at a single place, so as to store at that place the last occurrence position in the pattern set. The natural choice is a trie data structure where we store not only the patterns, but also every suffix of every pattern, that is, the set of strings $P^k_{i...m}$, for $1 \leq k \leq r$ and $1 \leq i \leq m$. Since the trie data structure stores one node for each prefix of each string stored, it follows that there will be one node for each prefix of each suffix of every pattern, or which is the same, one node for each substring of each pattern in the set.

Figure 27 gives an example for the pattern set formed by four words: `"para"`, `"pare"`, `"hola"` and `"arar"`. We have inserted the patterns and their suffixes (as shown in the top part of the figure). We have numbered the nodes as they were created when inserting the pattern suffixes in the trie. The dotted arrows are the so-called *suffix links*, connecting the node representing substring aw to that representing substring w, where a is a single character.

On top of each node we have drawn a list of numbers. These are the final positions where the substring w represented by the node appears in some pattern of the set. We also include final positions (that is, lengths) of pattern prefixes that match a suffix of w. The lists are stored in decreasing order and without repetitions. Although we draw a hyphen to separate full occurrences of w in the patterns from suffixes of w that match pattern prefixes, it is easy to distinguish them anyway because the former cannot be smaller than $|w|$ and the latter are smaller than $|w|$.

In Figure 27, the list of node 6 (`"ar"`) is 4,3,2, which means that it appears in some pattern of the set finishing at those positions. It has a suffix link to node 8 (`"r"`). Node 7 (`"ara"`) occurs at positions 4 and 3, but also its suffix of length 1 is a prefix of some pattern in the set (`"arar"`).

It is easy to build the trie by first inserting each full pattern and then its shorter and shorter suffixes, adding the suffix links at the same time. The lists of full substring occurrences are also created at the time we insert the suffixes of each pattern. Finally, the final parts of the lists, of node suffixes that match pattern prefixes, is computed by a level-wise traversal over the trie. Note that all the suffixes of w that are prefixes of a pattern are also suffixes of aw that are prefixes of a pattern. So, to compute the final section of the list for a node representing aw, we use the suffix link to retrieve the final section of the list for the node representing w. The only extra action needed is to add $|w|$ to the list if w itself is a prefix of some pattern. This is easily known by checking whether the node representing string w has $|w|$ in its list of full occurrences.

This trie is used to replace table $J(i, \ell)$ as follows. For each new block we first find whether it is a substring of some pattern, by finding out which node it corresponds to. The first empty

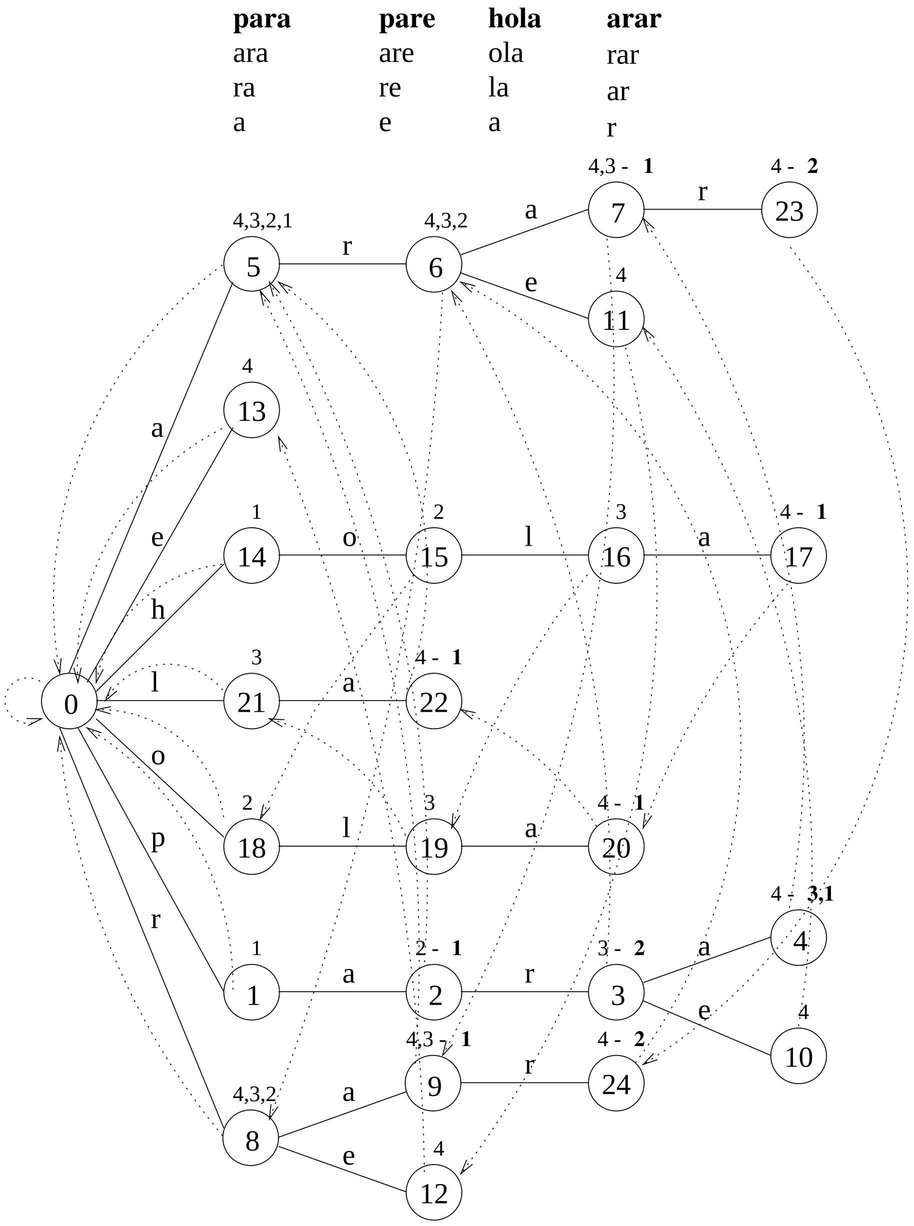


Figure 27: A trie data structure built over the suffixes indicated on top, plus some additional information needed by the algorithm.

block clearly corresponds to the root of the trie (that represents the empty string). For a new block $b = (s, c)$, we find out the trie node n_s corresponding to block s, and see if one can follow by an edge labeled c. If we can, then the child node n_b corresponds to b, otherwise b is not a substring of any pattern. If there is such a node n_b, then we can find all the final positions where b occurs inside any pattern, in the list associated to node n_b. This list is conveniently sorted in decreasing order so we can find the largest useful position, that is, the one not exceeding the position of b in the current window. With this information we can determine whether a shift is possible or not.

The above technique must be slightly complicated to account for partial matches, that is, for cases where block b does not occur inside any pattern, but its suffix matches a pattern prefix. For each block, we do not only store its corresponding trie node, but also an indication telling whether the block appears completely or just its suffix appears as a prefix. If $b = (s, c)$ and s appears partially, then b can only appear partially. To find its appropriate node n_b, we try to descend from n_s by c. We can descend only if the appropriate edge exists and the child node is a prefix of some pattern (that is, if it represents string w, then $|w|$ must appear in its list). If we can descend, we are done and this is a partial occurrence for b. If we cannot, it still might be that we can find a proper node by following suffix links from b_s and trying to descend by character c, under the same condition of arriving at a node that is a pattern prefix. If we finally arrive at the root node and still cannot descend by c, then we associate the root node to b, and can shift the window until completely surpassing block b. A similar process is followed if n_s is a complete occurrence for s, but we cannot find a descendant by character c. Since b cannot have a complete occurrence, we use the same mechanism of following suffix links in order to find a partial occurrence. Note that if block b turns out to have only partial occurrences, then its occurrences in the pattern set correspond only to the last part of the list of node n_b.

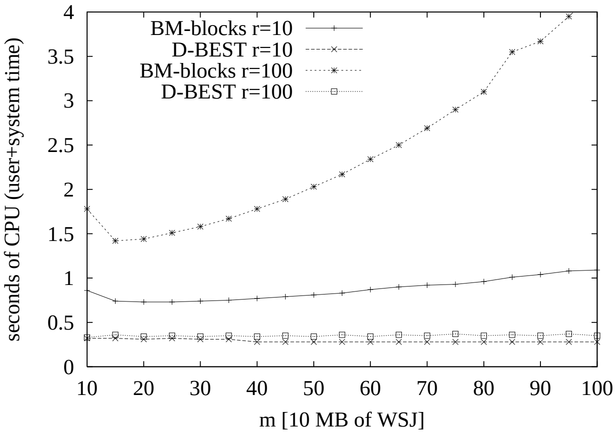
The rest of the algorithm is the same as in Section 7. Figure 28 compares this algorithm against decompress-then-search. Although the idea is elegant, it is clear that the overhead for constructing and managing the trie quickly becomes dominant as m grows (note that the construction takes time $O(rm^2)$). However, the algorithm is rather attractive for small and few patterns $r = m = 10$ on DNA text. This is the only point where BM-multichar could not beat decompress-then-search.

10 Direct Searching for Multiple Patterns

It turned out that each of the three alternatives studied has its own area where it is the only one beating the naive decompress-then-search approach. BM-simple dominates on WSJ text, while on DNA the best is BM-blocks for $m = 10$ and BM-multichar for larger m. What is less clear is which is the range of r values where each algorithm is useful.

Figure 29 shows the results for BM-simple on WSJ and BM-multichar on DNA, comparing against decompress-then-search, on a more refined scale of r values. It can be seen that, the larger r, the larger m is needed by BM-simple to beat decompress-then-search. On the other hand, on DNA there is a range of m values with a minimum and maximum pattern length, and the range narrows as more patterns are searched for. The result is shown in Figure 30.

Overall, it can be seen that we have suceeded only moderately in efficient searching for multiple patterns. We beat decompress-then-search only for a handful of patterns. There are, however, several applications where this is precisely the case of interest.



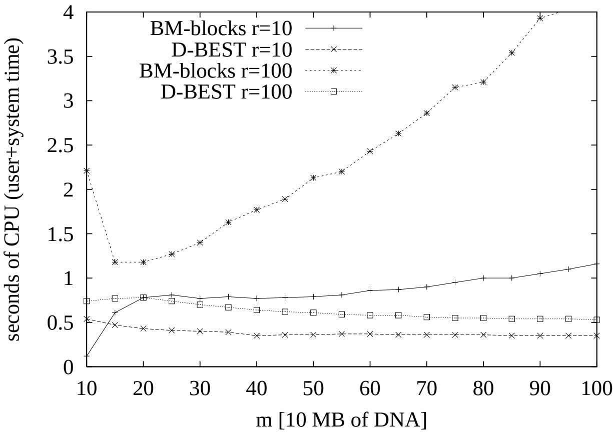
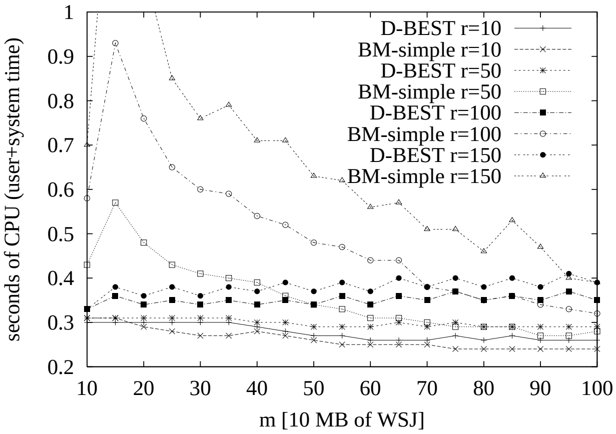


Figure 28: Comparison between multipattern decompress-then-search and direct searching with block-based Boyer-Moore approaches.



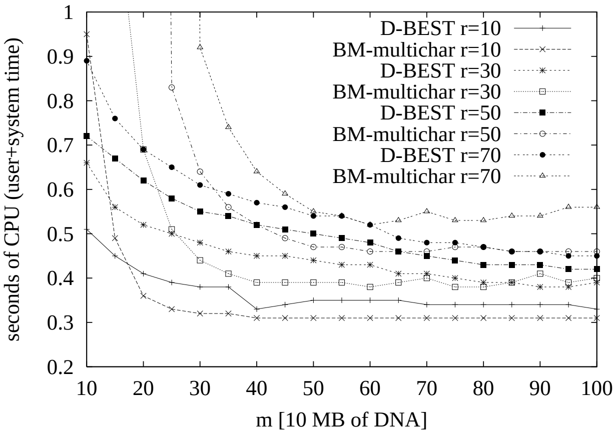
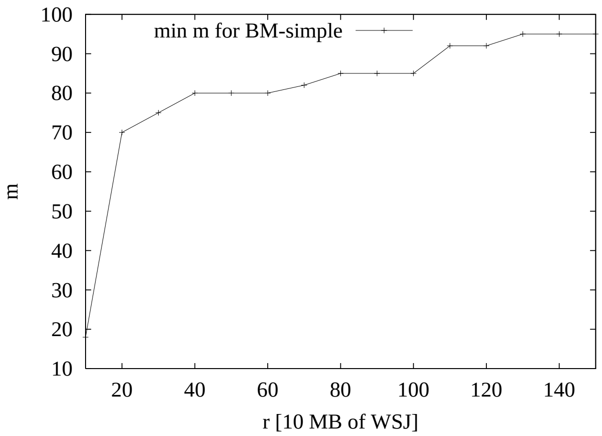


Figure 29: Comparison between decompress-then-search and direct searching using the best approach of each kind.



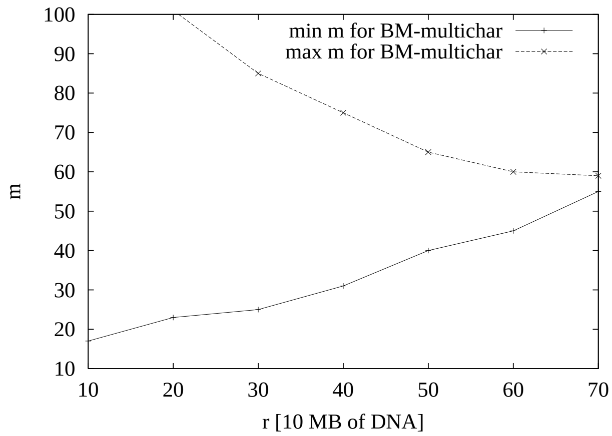


Figure 30: The ranges of m and r values where our algorithms are superior to the decompress-then-search approach.

We have chosen simple default choices for our *LZgrep* search tool. We use BM-simple until 10 patterns, DW-WM until 100 patterns and DW-SBOM for more than 100 patterns. Users can override this default choice if they wish.

11 LZgrep: A Direct Compressed Text Search Tool

Using the best direct search algorithms developed, we built a compressed text search tool called *LZgrep*, with the aim of replacing the simpler but slower *zgrep*. *LZgrep* can search files compressed with Unix *compress* (a LZW compressor) and with Unix *gzip* (a LZ77 compressor), both of which are public. In order to use the best algorithm, we resort at times to a decompress-then-search approach, especially for multipattern search and necessarily on LZ77.

For the sake of replacing *zgrep*, we have to be as compatible as possible with *Gnu grep*. In particular, *grep* handles regular expressions, which we have not addressed. If *LZgrep* receives such kind of unsupported patterns or is requested to use an unsupported option, it simply invokes *zgrep*. This guarantees that *LZgrep* is faster than *zgrep* whenever possible, and at the same time ensures its full compatibility.

The main difference in the behavior of the search algorithms when we simulate *grep* is that we do not have to output the text positions that match, but the contents of the text lines that contain an occurrence of the pattern(s). Therefore, upon finding an occurrence, we uncompress the current line by accessing the contiguous blocks ahead and behind until we uncompress a newline. Then we send the uncompressed line to the standard output and shift the window to the beginning of the next line.

Other differences in the search behavior can be obtained through the search options of *grep*. One of the main changes in the search algorithms made to accomodate them was that we remember, for each block, the number of newlines inside it and the byte offset of the first newline with respect to the beginning of the block. This is easily computed for each new block read.

The options considered follow.

-A *num*, **-B** *num*, **-C** *num*: Print several lines preceding and following an occurrence. We avoided uncompressing text lines more than once, by storing the last uncompressed lines.

-b: Print the byte offset of each line reported, counting from the beginning of the file. This is easily obtained by remembering the byte offset of the current block and adjusting it as we uncompress the text that has to be shown.

-c: Count the number of matching lines rather than output them. Instead of uncompressing the surrounding blocks in order to find the next newline, we skip all the blocks without newlines that follow, and position the window right after the first newline of the next block.

-d *action*: Handle subdirectories by skipping them ($action$ = skip) or entering recursively ($action$ = recurse).

-D *action*: Handle sockets by skipping them ($action$ = skip) or reading them ($action$ = read).

-i: Ignore upper/lower case. This is elegantly handled in the LZW format, by changing the meaning of the initial default blocks 0–255, so that block codes corresponding to upper case letters are mapped to their lower case versions. The result is that the uncompressed text will be seen all as lower case. Any search pattern is mapped to lower case too.

-f *file*, **-F:** Search for a set of patterns separated by newlines, either from *file* (-f) or from the same pattern string (-F).

-h, -H: Print (-H) or do not print (-h) the file names containing the matches found.

-l, -L: Only output file names that contain (-l) or do not contain (-L) occurrences. For this sake we stop the search on a file upon finding the first occurrence and proceed accordingly.

-m *num*: Stop after finding *num* occurrences. We extend the previous mechanism to stop after *num* instead of after the first occurrence.

-n: Print also the line numbers of the lines output. This is easily handled by keeping the current line number, thanks to the information maintained on the newlines inside blocks.

-o: Show only the part of the line that matched. This is especially simple for *LZgrep* because only exact matching is supported. The only exception for this rule is when ignoring case (option -i).

-r, -R: Recurse inside subdirectories.

-v: Output the lines that do not contain occurrences. It is rather difficult to avoid uncompressing most of the file under this scenario, so we opted for not implementing this option, but rather switching to *zgrep*.

-w, -x: These options imply inexact searching, so we did not implement them, but switch to *zgrep* if they appear.

There are other options trivially handled such as -s (supress error messages), -V (print version), -q (supress output), -a and -I (handling of binary files).

It is rather difficult to choose the best search algorithm as the default. For example, we have seen that, depending on the text type (English or DNA), the correct option changes. Worse than that, there is no easy way to determine which is the type of the text we are going to search. Reading the first bytes of the compressed file we can know that it was compressed using LZ77 or LZW, but nothing else. Hence we have chosen the defaults to be the search algorithms that with higher probability would behave reasonably well on different types of texts.

On LZW, for single pattern matching we use BM-simple. For multipattern matching we use BM-simple until 10 patterns, DW-WM until 100 patterns and DW-SBOM for more than 100 patterns. On LZ77, for single patterns we use D77-BOM. For multipattern matching we use D77-WM until 100 patterns, and D77-SBOM for more patterns.

In case the algorithm chosen is not the best for a particular purpose, and also in order to ease the use of *LZgrep* for research purposes, we added an option **−alg=ALGORITHM**, where "ALGORITHM" permits choosing any of the algorithms we have considered in this work. The possible algorithm names are:

D-alg, where "alg" is BM, BOM or KMP for single pattern searching and WM, SBOM or AC for mutipattern searching. "D" stands for DW or D77 depending on the format the input text is compressed.

BM-simple[-ext], which applies BM-simple algorithm for LZW-compressed text. Possible values for "-ext" are: empty, "-basic" or "-lf" on single pattern searching, and "-multi", "-multi-2", "-multi-3" or "-multi-4" on multipattern searching.

BM-multichar[-ext], which applies BM-multichar algorithm for LZW-compressed text. On single pattern searching "-ext" must be empty, while on multipattern searching it must be either "-multi" or "-multi-2".

BM-blocks[-ext], which applies BM-blocks algorithm for LZW-compressed text. On single pattern searching, "-ext" can be empty, "-basic" or "-2J", while on multipattern searching it can only be "-multi".

12 Conclusions

We have presented several practical algorithms for direct searching of single and multiple patterns on LZW compressed text. Most of the research on this topic is more theoretical and involved. Our algorithms are much simpler and, in practice, faster than previous work. There exist some competitive practical alternatives on other compression formats, but these formats have not (yet) popularized enough to make these alternatives interesting for a wide audience.

Our goal was the development of a widely applicable compressed text search tool. This is of great interest in order to maintain all the user's files usually in compressed form, uncompressing them only when necessary. The growing gap between CPU and disk times makes this idea more and more appealing as technology evolves. In order to support this scenario in a form that is comfortable for general use, it is imperative to be able of searching the compressed files directly without the need to manually uncompress them before the search.

Such a tool, *zgrep*, exists at this moment in the form of a very simple script that uncompresses the text and sends it to a pattern matching software, *grep*. We have shown that it is possible to be up to 50% faster than *zgrep*, by searching the compressed text directly without decompression.

As a result, we have developed *LZgrep*, a free program[2] designed to replace *zgrep*. *LZgrep* solves a (significant) subset of the search problems addressed by *grep*, namely exact single and multiple pattern searching, and it resorts to *zgrep* in case of an unsopported search problem. This ensures full compatibility and at the same time improved performance in the most common cases. We note that, although we have focused on the LZW format, we have in passing obtained decompress-then-search algorithms for the more popular LZ77 format that are much faster than *zgrep*, because we avoid the overhead of communication between two unrelated programs (the uncompressor and *grep*). These capabilities are also incorporated into LZgrep, which makes it appealing to search LZ77 compressed files as well.

Note that there exist currently several environments that intercept all the communication to the file system so as to store the files in compressed form in a way that is transparent to the user. A text search is naturally solved by decompressing the file (by means of reading it from disk) and then searching it. Tools like *LZgrep* could be incorporated to those environments in order to provide a more efficient native search over the compressed search.

Our current plans are to extend *LZgrep* to support more sophisticated search problems, in particular approximate searching and regular expression searching. In the former case, there exists already a practical search algorithm based on direct multipattern search on compressed text [29], which could be easily adapted to our case. Since each approximate occurrence involves the exact occurrence of some substrings of the pattern, the idea is to split the pattern into subpatterns, search the subpatterns simultaneously, and uncompress the text areas surrounding each subpattern occurrence in order to find complete approximate occurrences. For regular expression searching, there exists already an algorithm [28], but we believe that it would be much more practical to resort again to multipattern searching for the main part of the search [38], using the algorithms developed in this paper, just as it was done for approximate searching. Finally, we must keep up to date with the best developments in plain text searching, so as to try to adapt them to compressed text searching, and also to use them on the uncompress-then-search portions of *LZgrep*. A recent promising algorithm for multipattern searching is [21].

Acknowledgements

We thank Marcos Rojas (University of Chile) and Carlos Avendano-Pérez (INAOEP, Mexico), for their help in implementing *LZgrep*.

References

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, 1975.

[2] Use of it for commercial advantage requires explicit permission from the authors.

[2] C. Allauzen, M. Crochemore, and M. Raffinot. Efficient experimental string matching by weak factor recognition. In *Proc. 12th Ann. Symp. on Combinatorial Pattern Matching (CPM'01)*, LNCS 2089, pages 51–72, 2001.

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

[4] 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.

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

[6] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Comm. of the ACM*, 20(10):762–772, 1977.

[7] B. Commentz-Walter. A string matching algorithm fast on the average. In *Proc. 6th Intl. Coll. on Automata, Languages and Programming (ICALP'79)*, LNCS 71, pages 118–132, 1979.

[8] M. Crochemore, A. Czumaj, L. Gąsieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter. Speeding up two string matching algorithms. *Algorithmica*, 12(4/5):247–267, 1994.

[9] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

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

[11] E. Fiala and D. Greene. Data compression with finite windows. *Comm. of the ACM*, 32(4):490–505, 4 1989.

[12] L. Gasieniec, M. Karpinksi, W. Plandowski, and W. Rytter. Efficient algorithms for Lempel-Ziv encodings. In *Proc. 5th Scandinavian Workshop in Algorithmic Theory (SWAT'96)*, 1996.

[13] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

[14] R. N. Horspool. Practical fast searching in strings. *Software Practice and Experience*, 10:501–506, 1980.

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

[16] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 166–180, 2000. Extended version to appear in *J. of Discrete Algorithms.*

[17] 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.

[18] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. 8th IEEE Data Compression Conference (DCC'98)*, 1998.

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

[20] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(1):323–350, 1977.

[21] J. Kytöjoki, L. Salmela, and J. Tarhio. Tuning string matching for huge pattern sets. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching (CPM'03)*, LNCS 2676, pages 211–224, 2003.

[22] 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.

[23] U. Manber and S. Wu. Glimpse: a tool to search through entire file systems. Technical Report 93–34, Dept. of Computer Science, Univ. of Arizona, October 1993.

[24] 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.

[25] M. Miyazaki, A. Shinohara, and M. Takeda. Speeding up the pattern matching machine for compressed texts. *Trans. of Information Processing Society of Japan*, 39(9):2638–2648, 1998.

[26] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

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

[28] G. Navarro. Regular expression searching over Ziv-Lempel compressed text. In *Proc. 12th Ann. Symp. on Combinatorial Pattern Matching (CPM'01)*, LNCS 2089, pages 1–17, 2001. Extended version to appear in *J. of Discrete Algorithms*.

[29] 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.

[30] G. Navarro, E. Moura, M. Neubert, N. Ziviani, and R. Baeza-Yates. Adding compression to block addressing inverted indexes. *Information Retrieval*, 3(1):49–77, 2000.

[31] G. Navarro and M. Raffinot. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proc. 10th Ann. Symp. on Combinatorial Pattern Matching (CPM'99)*, LNCS 1645, pages 14–36, 1999. Extended version to appear in *J. of Discrete Algorithms*.

[32] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics (JEA)*, 5(4), 2000.

[33] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, 2002.

[34] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 166–180, 2000.

[35] H. Peltola and J. Tarhio. String matching in the DNA alphabet. *Software Practice and Experience*, 27(7):851–861, 1997.

[36] Y. Shibata, T. Matsumoto, M. Takeda, A. Shinohara, and S. Arikawa. A Boyer-Moore type algorithm for compressed pattern matching. In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching (CPM'00)*, LNCS 1848, pages 181–194, 2000.

[37] D. Sunday. A very fast substring search algorithm. *Comm. of the ACM*, 33(8):132–142, 1990.

[38] B. Watson. A new regular grammar pattern matching algorithm. In *Proc. 4th European Symposium on Algorithms (ESA'96)*, LNCS 1136, pages 364–377, 1996.

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

[40] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes*. Morgan Kaufmann Publishers, New York, second edition, 1999.

[41] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Dept. of Computer Science, University of Arizona, 1994.

[42] 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.

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

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