Faster Approximate String Matching over Compressed Text

Gonzalo Navarro* Takuya Kida† Masayuki Takeda†
Ayumi Shinohara† Setsuo Arikawa†

Abstract

Approximate string matching on compressed text was a problem open during almost a decade. The two existing solutions are very recent. Despite that they represent important complexity breakthroughs, in most practical cases they are not useful, in the sense that they are slower than uncompressing the text and then searching the uncompressed text. In this paper we present a different approach, which reduces the problem to multipattern searching of pattern pieces plus local decompression and direct verification of candidate text areas. We show experimentally that this solution is 10–30 times faster than previous work and up to three times faster than the trivial approach of uncompressing and searching, thus becoming the first practical solution to the problem.

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 $\{|x|,\ T = xPy\}$. Exact string matching has been a basic problem in computer science since its beginnings [CR94, AG97].

A generalization of the basic string matching problem is *approximate string matching* [Nav00]: an error threshold $k < m$ is also given as input, and we want to report all the ending positions of text substrings which match the pattern after performing up to k character insertions, deletions and replacements on them. Formally, we have to return the set $\{|xP'|,\ T = xP'y \text{ and } ed(P, P') \le k\}$, where $ed(P, P')$ is the "edit distance" between both strings, i.e. the minimum number of character insertions, deletions and replacements needed to make them equal. The best algorithms for this problem need in the worst case $O(u)$ time, and $O(ku)$ if the extra space must be polynomial in m. On average, they need $O(u(k + \log_\sigma(m))/m)$ time. Approximate string matching is an important problem in textual databases (e.g. for spelling checkers and to query low-quality databases), computational biology (e.g. to search on DNA and protein databases) and signal processing (e.g. to search approximate patterns in audio strings or for speech recognition).

On the other hand, text compression [BCW90] tries to exploit the redundancies of the text to represent it using less space. There are many different compression

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

†Dept. of Informatics, Kyushu University 33. Fukuoka 812-8581, Japan. {kida,takeda,ayumi,arikawa}@i.kyushu-u.ac.jp.

schemes, among which the Ziv-Lempel family [ZL77, ZL78] is one of the most popular in practice because of their good compression ratios combined with efficient compression and decompression time.

It is natural to think of merging search and compression. The *compressed matching problem* was first defined in the work of Amir and Benson [AB92] as the task of performing string matching in a compressed text without decompressing it. Given a text T, a corresponding compressed string $Z = z_1 \ldots z_n$, and a pattern P, the compressed matching problem consists in finding all occurrences of P in T, using only P and Z. A naive algorithm, which first decompresses the string Z and then performs standard string matching, takes time $O(m+u)$. An optimal algorithm takes worst-case time $O(m+n+R)$, where R is the number of matches (note that it could be the case that $R = u > n$).

The compressed matching problem is important in practice. Today's textual and DNA databases are an excellent example of applications where both problems are crucial: the texts should be kept compressed to save space and I/O time, and they should be efficiently searched. Moreover, as the CPU speed improves much faster than that of I/O devices, it is worthwhile to spend more and more CPU time for the sake of reduced I/O. Surprisingly, the combined requirements of having a searchable and compressed text 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.

In particular, *approximate* searching on compressed text was advocated in 1992 as an open problem [AB92]. Only very recently a couple of solutions have appeared [KNU00, MKT$^+$00]. Despite their theoretical achievements, which are nontrivial, the experimental results in both papers show that in practice they are slower than a decompression of the text followed by a state-of-the-art search on the uncompressed text.

In this paper we take a different approach. We reduce the problem of approximate searching to the problem of multipattern searching of a set of pattern pieces plus local decompression and direct verification of candidate text areas. Apart from the existing algorithm for multipattern searching on compressed text [KTS$^+$98], we propose a couple of new algorithms by adapting existing single-pattern search algorithms [NT00, NR99, KTS$^+$99].

Our experimental results show that, for moderate error level k/m, the new algorithms are faster than the naive approach of uncompressing plus searching (up to 3 times faster for low enough error levels). For this moderate error level, the new algorithms are about 10–30 times faster than previous work. Therefore, this is the first *practical* result for this problem.

2 Related Work

Two different approaches exist to search compressed text. The first one applies to compression methods based on single symbol replacement, such as Huffman coding [Huf52]. Efficient solutions exist for this case [Man97, MNZBY00], but in general the compression ratio is not so good or the functionality is limited (e.g. to natural language text).

The second approach considers Ziv-Lempel compression, which is based on finding repetitions in the text and replacing them with references to similar strings previously appeared. LZ77 [ZL77] is able to reference any substring of the text already processed, while LZ78 [ZL78] and LZW [Wel84] 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 is from 1994 [ABF96], which presents a compressed matching algorithm for LZ78 which simulates a KMP machine [KMP77] and solves the existence problem (i.e. just determining whether the pattern appears or not) in time and space $O(m^2 + n)$. For LZ77 a randomized algorithm has been presented [FT98] to solve the same problem in time $O(m + n\log^2(u/n))$.

An extension of the former work to multipattern searching on LZ78/LZW was later presented [KTS$^+$98]. Based on Aho-Corasick [AC75], they achieve $O(m^2 + n + R)$ time and $O(m^2 + n)$ space to find all the occurrences of the patterns, where this time m is the total length of all the patterns.

Later work [NR99] presented a general scheme to search on Ziv-Lempel compressed texts (simple and extended patterns) and specialized it for some particular formats (LZ77, LZ78, etc.) Their approach is based on bit-parallelism, a technique to pack many values in the bits of a computer word of w bits and manage to update all them in parallel. A similar result, for LZW, was independently found [KTS$^+$99].

Approximate string matching on compressed text is an open problem advocated in 1992 [AB92]. Very recently, it has been solved for the LZ78/LZW formats in $O(mkn + R)$ worst case and $O(k^2n + R)$ average case time using dynamic programming techniques [KNU00] and in $O(nmk^3/w)$ worst case time using bit-parallelism [MKT$^+$00]. However, both solutions are very slow in practice. The aim of this paper is to present the first practical solution to this problem for the LZ78/LZW formats.

3 The LZ78/LZW Compression Formats

We introduce some notation for the rest of the paper. A string S is a sequence of characters over an alphabet Σ of size σ. The length of S is denoted as $|S|$, therefore $S = s_1 \dots s_{|S|}$ where $s_i \in \Sigma$. A substring of S is denoted as $S_{i \dots j} = s_i s_{i+1} \dots s_j$, and if $i > j$, $S_{i \dots j} = \varepsilon$, the empty string of length zero. In particular, $S_i = s_i$. P and T, the pattern and the text, are strings of length m and u respectively.

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. We are particularly interested in the LZ78 and LZW formats.

The LZ78 format [ZL78] is based on a dictionary of blocks, to 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 $T_{1 \dots j}$ of T has been already compressed into a sequence of blocks $Z = b_1 \dots b_r$, all of them in the dictionary, then we look for the longest prefix of the rest of the text $T_{j+1 \dots u}$ 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 $Z = b_1 \dots b_r b_{r+1}$, and we add the block to the dictionary. It is easy to see that any prefix of a dictionary element is also in the dictionary, and a natural way to represent it is a trie.

Given a block $b_r = (s, c)$ we call $ref(r) = s$ and $char(r) = c$. It is also easy to know the length $len(r)$ as $len(0) = 0$ and $len(r) = len(ref(r)) + 1$.

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 compress using limited memory. A particularly interesting variant is from Welch, called LZW [Wel84], which is used by Unix's *Compress* program. In this case, the extra letter (second element of the pair) is not coded, but it is obtained as the first letter of the next block (the dictionary is started with one block per letter). Hence any algorithm designed for LZ78 is trivially adapted to LZW, which we do not consider separately in our algorithm description (despite that our actual implementations work on LZW).

4 Our Search Approach on LZ78/LZW

Most algorithms to search on compressed text borrow their ideas from classical algorithms on uncompressed text and adapt them to work on a sequence of Ziv-Lempel blocks rather than on a sequence of characters. The solutions for approximate string matching are not an exception. Four basic approaches exist on classical approximate pattern matching [Nav00], three of which are of interest for this paper: (1) Dynamic Programming, (2) Bit Parallelism, and (3) Filtration. (Refer to the survey [Nav00].) In the present work, we adapt a simple but powerful filtration technique due to [WM92, NBY99] to work on compressed text. The idea is that if a pattern is split in $k+1$ nonoverlapping pieces, then at least one of the pieces must appear unaltered inside every occurrence with at most k errors. The reason is that each error can alter at most one piece.

We split the pattern in $k+1$ equal-length pieces trying to maximize the length of the shortest piece. Hence their length is $\lfloor m/(k+1) \rfloor$. We run a multipattern search algorithm on the compressed text, looking for the positions where any of the pieces appears. We discuss later the multipattern search algorithms that can be used.

Each time a piece is found, we decompress the candidate text area and apply a classical algorithm over the decompressed area (of length $m + 2k$). Say that pattern piece $P_{i \dots i'}$ begins at text position j. Then the text area to check is $T_{j-i+1-k \dots j+m-i+k}$. We have chosen Myers' algorithm [Mye98] for the verification.

In order to decompress locally we need to store, as we process the compressed text blocks b_r, the information on $ref(r)$, $char(r)$ and $len(r)$ in arrays. We also need to keep track of the current text position j, which is easy by accumulating the lengths of the blocks. If the last block processed b_r finishes at text position j and we find a pattern piece ending inside b_r, then we can obtain the text characters $T_{j-len(r)+1 \dots j}$ in reverse order as $char(r)$, $char(ref(r))$, $char(ref(ref(r)))$, etc. If former characters are

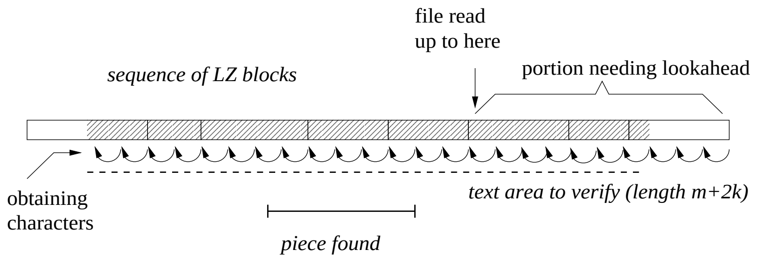

Figure 1: Our search method on compressed text.

needed then we look at the previous block $r-1$ and obtain $T_{j-len(r)-len(r-1)+1\dots j-len(r)}$ in reverse order as $char(r-1)$, $char(ref(r-1))$, $char(ref(ref(r-1)))$, and so on.

However, it may be necessary to obtain some characters ahead of the last block read. This is implemented with a lookahead mechanism, where the module that reads the compressed text is $m+k$ blocks ahead. This guarantees that the required block information is always present. Figure 1 illustrates.

It is common to have overlapping verification requirements when the error level k is not very low. To avoid reverifying many times the same text area, we keep track of the last text position verified and the state of the verification algorithm at that point. If the new requirement overlaps with the previous one we restart only from the last verified text position using the stored search state, instead of initializing it.

Let us consider the average complexity of this scheme. A probably optimal multipattern search algorithm [KTS+98] takes time $O(m^2+n)$. For each occurrence of a piece we need to work $O(m(m+2k)) = O(m^2)$ with a classical algorithm, or $O(m^2/w)$ using Myers' algorithm (note that we cannot use average time figures here because a candidate text area is not random). We use $O(m^2)$ for simplicity because the result is basically the same. The number of text positions where some piece matches is $(k+1)u/\sigma^{\lfloor m/(k+1)\rfloor}$. This gives a total search time of

$$O\left(m^2+n+\frac{ukm^2}{\sigma^{\lfloor m/(k+1)\rfloor}}\right)$$

which is competitive against the existing $O(k^2n)$ complexity for

$$k+1 \quad \leq \quad \frac{m}{\log_\sigma(u/n)+\log_\sigma m}(1+o(1))$$

and competitive against an $O(u)$ time decompression followed by an optimal search algorithm for

$$k+1 \quad \leq \quad \frac{m}{3\log_\sigma m}(1+o(1)) \quad < \quad \frac{m}{3\log_\sigma m}$$

This shows that our approach is good in general when the error level permitted k/m is low enough. When compared to previous algorithms to search on compressed text, a high compression ratio u/n goes against our algorithm because of our need to uncompress candidate text areas. The influence, however, is quite small (logarithmic).

It is worthwhile to notice that the memory we need for searching is the same necessary to decompress the file. When the Ziv-Lempel dictionary is reinitialized we can do the same. We describe now three possible techniques for the multipattern search. These are later compared in the experiments.

4.1 An Aho-Corasick Technique

The multipattern search algorithm on LZ compressed text [KTS+98] is based on the simulation of the Aho-Corasick pattern matching machine [AC75]. A Mealy type sequential machine is built which processes the blocks one by one. It consists of two functions: *Jump* and *Output* defined on the domain $Q \times D$, where Q is the set of states of the AC machine and D is the set of Ziv-Lempel blocks. For every block b_r, the machine makes just one state transition using *Jump*, which corresponds to the consecutive state transitions of the AC machine caused by the string represented by b_r. The AC machine being simulated may, however, pass through a state with outputs during the consecutive state transitions. The function *Output* is used to avoid missing such outputs.

The function *Jump* can be built in $O(m^2 + n)$ time using $O(m^2 + n)$ space, so that it returns its value in $O(1)$ time. The function *Output* can also be built at the same time and space complexities, so that it returns in $O(1)$ time a linear size representation of the set of outputs. On the other hand, the text scanning takes $O(n + R)$ time, where R is the number of pattern occurrences. Thus, the algorithm runs in $O(m^2 + n + R)$ time using $O(m^2 + n)$ space. The details are involved and the reader interested in them is referred to the original paper [KTS+98].

4.2 A Boyer-Moore Technique

An existing Boyer-Moore technique to search a single pattern on compressed text [NT00] can be adapted to multipattern searching. The Boyer-Moore technique [BM77] consists in aligning the pattern in a text window and comparing the window text characters with the corresponding pattern characters. If a mismatch occurs, then a *safe shift* is computed, which permits to slide the window forward in the text without risk of missing an occurrence. If no mismatch occurs, then a pattern occurrence is reported and the window is slid by 1.

A number of techniques to adapt the idea to Ziv-Lempel compressed text have been presented [NT00]. The general idea is to use the explicit characters c of the blocks $b_r = (s, c)$ to try to shift. Among the many shift functions proposed, we have been able to adapt the one which precomputes a table

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

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

If the explicit characters do not permit a shift (i.e. $B_P(i, c) = 0$), then we start to use the characters of the referenced blocks: $char(s)$, $char(ref(s))$, etc. on all the blocks that overlap with the text window. If no window character permits a shift, a match is reported.

To adapt the technique to search a set of pattern pieces $P^0 \dots P^k$, we precompute a pessimistic B table which permits the minimum shift among all the patterns:

$$B(i,c) = \min_{0 \le r \le k} B_{P^r}(i,c)$$

which is easily precomputed and clearly cannot lose any match. On the other hand, the fact that no text window character permits a shift does not immediately imply that one of the patterns has appeared. For example we can search for `"abc"` and `"def"`, and a text window `"aec"` will not permit a positive shift.

Hence, when a shift is not possible we have to obtain the characters of the text window and check directly for the presence of any of the patterns in the set. Obtaining the window characters has already been done when trying to shift the window, and the direct check can be done in time proportional to the window length by storing all the patterns P^r in a trie data structure.

4.3 A Bit Parallel Technique

There exists a bit-parallel technique [NR99, KTS$^+$99] to search a single pattern on Ziv-Lempel compressed text. The technique uses bit-parallelism to store a set of pattern positions. For each block of the compressed text we store the set of pattern prefixes which match a block suffix and the set of pattern suffixes which match a block prefix. The list of positions inside each block where there is an occurrence of the whole pattern is also maintained. Finally, the state of the search consists of the current text position and a bit mask indicating the set of pattern prefixes which have matched a suffix of the text read up to now.

When a new block $b_r = (s,c)$ appears, we carry out three actions: (i) compute its prefixes, suffixes and internal matches; (ii) report the new occurrences; (iii) update the search state. The first part is done using the prefixes, suffixes and internal matches of s, to which c has to be added at the end. The second part is obtained by joining a pattern prefix which matches the text suffix read up to now with a pattern suffix that matches a prefix of the new block. To those matches we have to add the positions inside the new block where the whole pattern appears. Finally, the search state is updated by knowing the length of the new block and the set of pattern prefixes which match a suffix of the new block. The reader is referred to the original papers [NR99, KTS$^+$99] for more details.

The idea for multipattern searching is to carry out all the $k+1$ searches in parallel by packing the sets of pattern positions of all the patterns in a single computer word. The number of bits required for the bit-parallel simulation with one pattern is proportional to the pattern length m. Since we are now searching for $k+1$ subpatterns of length $\lfloor m/(k+1) \rfloor$, the total length is at most m, so the space required is similar to that of an exact search for the original pattern.

All the operations done on the bit masks to handle sets of pattern prefixes and suffixes can be easily adapted to the case where the information of several patterns is packed together. It is necessary to exercise some care to make sure that the information about one pattern does not overflow onto the bit area of another pattern, but avoiding this with the use of the appropriate bit masks is an easy exercise.

5 Experimental Results

We have implemented our algorithms on LZW, modifying Unix's *Compress* program (so we are able to search files with ".Z" extension). We ran our experiments on an Intel Pentium III of 550 MHz and 64 Mb of RAM running Linux. The word length is $w = 32$ bits. We have compressed 10 Mb of Wall Street Journal articles (WSJ) and 10 Mb of DNA. WSJ was compressed to 42.59% of its size and DNA to 27.71%.

We have compared our three algorithms against previous work [KNU00, MKT$^+$00] and against the naive approach of decompressing plus searching with the best available algorithms on uncompressed text.

We have tested $m = 10$ to 30, and $k = 1$ to $m/2$. For each pattern length, we selected 100 random patterns from the text and used the same patterns for all the algorithms. As the system times are negligible because of caching, we report user times only. We test the following algorithms: the previous work based on dynamic programming over compressed text [KNU00] (**DP**), the previous work based on bit-parallelism over compressed text [MKT$^+$00] (**BP**), the naive approach of decompressing plus searching using partitioning into $k + 1$ pieces [NBY99] (**U+PP**) and using bit-parallelism [Mye98] (**U+BP**), and our new algorithms using different multipattern search techniques: the Aho-Corasick method [KTS$^+$98] (**PP/AC**), our adaptation of the Boyer-Moore method [NT00] (**PP/BM**) and our adaptation of the bit-parallel method [NR99, KTS$^+$99] (**PP/BP**).

Figure 2 shows some results. Our methods improve by far over previous work. In the area where they are interesting (i.e. better than the naive approach) they are 10 to 30 times faster than previous work. In order to enhance visibility we have had to leave out of the plots those DP and BP algorithms,but they take at least (i.e. for $m = 10, k = 1$) more than 6 seconds. A rough and optimistic approximation of DP's search time is $2 + 0.65m + 0.2mk$ seconds on WSJ and $1 + 0.25m + 0.15mk$ on DNA, while BP is even slower.

On the other hand, the new algorithms are faster than the naive approach of decompressing and searching when k/m is low enough, where "low enough" depends also on the type of text and pattern length, e.g. the limit is more tolerant on shorter patterns and larger alphabets, as predicted by the analytical condition $k/m \leq 1/3 \log_\sigma m$. The maximum acceptable k/m value on WSJ ranges from 40% for $m = 10$ to 30% for $m = 30$, while on DNA it is 20%. These ranges include most of the interesting cases in practical applications. Observe that in the best cases (lowest k) we are about three times faster than the best naive approaches. We have included also the time necessary just to decompress, since up to that time our approach is guaranteed to be faster than any decompress plus search approach.

Finally, let us compare the different implementations of our algorithm. In general, the PP/BP version is better, which is reasonable because the length of the pieces is $\lfloor m/(k+1) \rfloor$, which is normally not enough for the PP/BM variant to take advantage of the pattern length. The only cases where PP/BM is superior is on WSJ when the error level is very low, i.e. k/m equal to 10% or less. PP/AC is slower than PP/BP, but this should change for longer patterns, as the bit-parallel simulation needs $\lceil m/w \rceil$ operations.

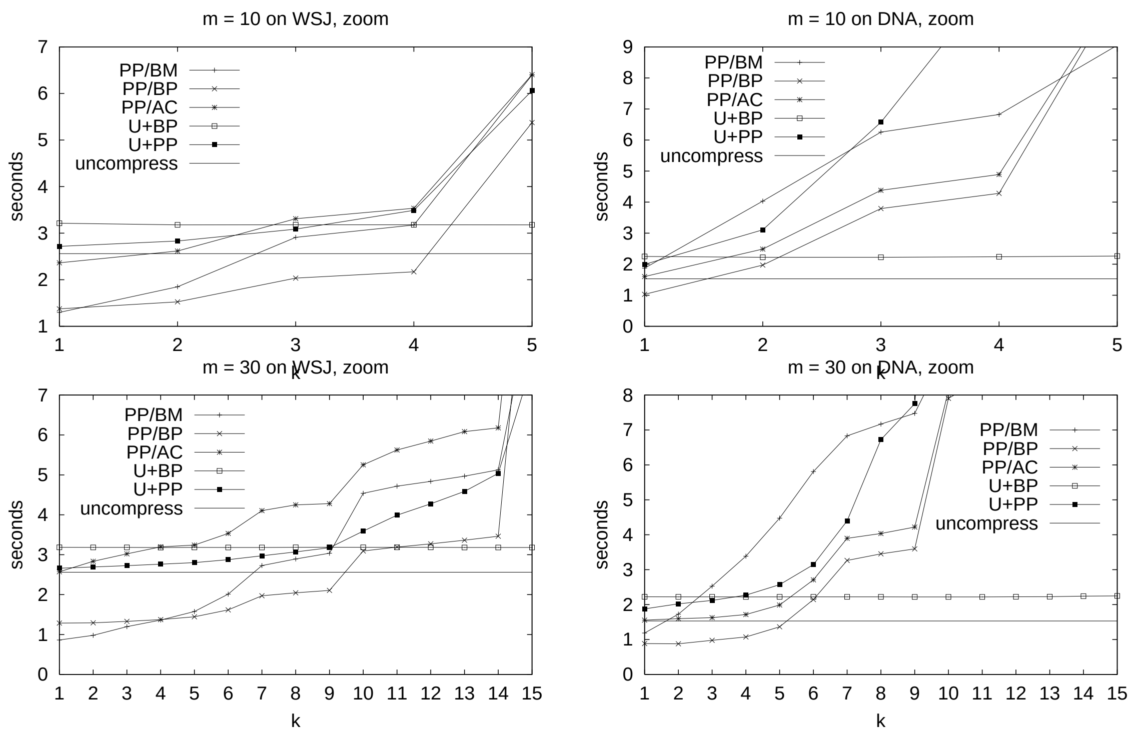

Figure 2: Time in seconds for the different algorithms on different texts (English on the left and DNA on the right), for $m = 10$ (top) and 30 (bottom) and $k = 1$ to $m/2$.

6 Conclusions

We have presented the first practical solution for approximate pattern matching over Ziv-Lempel compressed text. Our algorithm splits the pattern in $k+1$ pieces and performs a multipattern search of the pieces on the compressed text. The candidate text areas are locally decompressed and verified with a classical algorithm. We use one existing multipattern search algorithm and propose other two. Our experimental results show that our algorithms are at least ten times faster than the previous solutions and up to three times faster than u decompressing plus searching.

Despite that we have concentrated on the edit distance, many more complex variants can be accommodated under the $(k+1)$-pieces filter, e.g. different costs for the operations, transpositions, block moves, etc. [Nav00].

References

[AB92] A. Amir and G. Benson. Efficient two-dimensional compressed matching. In *Proc. DCC'92*, pages 279–288, 1992.

[ABF96] 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.

[AC75] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Comm. of the ACM*, 18(6):333–340, June 1975.

[AG97] A. Apostolico and Z. Galil. *Pattern Matching Algorithms.* Oxford University Press, 1997.

[BCW90] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, 1990.

[BM77] R. Boyer and J. Moore. A fast string searching algorithm. *CACM*, 20(10):762–772, 1977.

[CR94] M. Crochemore and W. Rytter. *Text Algorithms.* Oxford University Press, 1994.

[FT98] M. Farach and M. Thorup. String matching in Lempel-Ziv compressed strings. *Algorithmica*, 20:388–404, 1998.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.

[KMP77] D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 6(1):323–350, 1977.

[KNU00] J. Kärkkäinen, G. Navarro, and E. Ukkonen. Approximate string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, pages 195–209, 2000.

[KTS$^+$98] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa. Multiple pattern matching in LZW compressed text. In *Proc. DCC'98*, pages 103–112, 1998.

[KTS$^+$99] 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.

[Man97] U. Manber. A text compression scheme that allows fast searching directly in the compressed file. *ACM TOIS*, 15(2):124–136, 1997.

[MKT$^+$00] T. Matsumoto, T. Kida, M. Takeda, A. Shinohara, and S. Arikawa. Bit-parallel approach to approximate string matching in compressed texts. In *Proc. SPIRE'2000*, pages 221–228. IEEE CS Press, 2000.

[MNZBY00] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM TOIS*, 18(2):113–139, 2000.

[Mye98] G. Myers. A fast bit-vector algorithm for approximate pattern matching based on dynamic progamming. In *Proc. CPM'98*, LNCS 1448, pages 1–13, 1998.

[Nav00] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 2000. To appear. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/survasm.ps.gz`.

[NBY99] G. Navarro and R. Baeza-Yates. Very fast and simple approximate string matching. *Information Processing Letters*, 72:65–70, 1999.

[NR99] 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.

[NT00] G. Navarro and J. Tarhio. Boyer-Moore string matching over Ziv-Lempel compressed text. In *Proc. CPM'2000*, LNCS 1848, pages 166–180, 2000.

[Wel84] T. A. Welch. A technique for high performance data compression. *IEEE Computer Magazine*, 17(6):8–19, June 1984.

[WM92] S. Wu and U. Manber. Fast text searching allowing errors. *Comm. of the ACM*, 35(10):83–91, 1992.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory*, 23:337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable length coding. *IEEE Trans. Inf. Theory*, 24:530–536, 1978.