Compressed Pattern Matching in DNA Sequences

Lei Chen, Shiyong Lu, and Jeffrey Ram
Wayne State University
ak3230@wayne.edu, shiyong@cs.wayne.edu, and jeffram@med.wayne.edu

Abstract

We propose derivative Boyer-Moore (d-BM), a new compressed pattern matching algorithm in DNA sequences. This algorithm is based on the Boyer-Moore method, which is one of the most popular string matching algorithms. In this approach, we compress both DNA sequences and patterns by using two bits to represent each A, T, C, G character. Experiments indicate that this compressed pattern matching algorithm searches long DNA patterns (length > 50) more than 10 times faster than the exact match routine of the software package Agrep, which is known as the fastest pattern matching tool. Moreover, compression of DNA sequences by this method gives a guaranteed space saving of 75%. In part the enhanced speed of the algorithm is due to the increased efficiency of the Boyer-Moore method resulting from an increase in alphabet size from 4 to 256.

1. Introduction

String pattern matching is a common important operation in many applications. Various good solutions have been presented for pattern matching. One of the most efficient methods is the Boyer-Moore algorithm (BM) [3, 5] that was developed by R. S. Boyer and J. S. Moore. The Boyer-Moore algorithm uses three clever ideas [10]: the right-to-left scan, the bad character shift rule, and the good suffix shift rule. Together, these ideas lead to a method that typically runs in "sublinear" time for sufficiently large alphabets and sufficiently long patterns.

An interesting application of string matching is in pattern matching in DNA sequences consisting of four characters A, C, G and T. DNA is the genetic blueprint that determines heritable traits of living organisms. A DNA sequence for an organism can be long and contain a lot of information. Large numbers of DNA sequences require efficient storage. Recently, the compressed pattern matching problem attracted special interest, with the goal of enhancing storage efficiency while speeding up the search time. The compressed pattern matching problem was first defined in the work of Amir and Benson [1] as the task of finding pattern occurrences in compressed sequence without first decompressing it. Using this technique we can compress DNA sequences to reduce their size and I/O overhead considerably and reduce the time to search patterns directly in compressed DNA sequences. Various compression methods have been extensively studied in the last decade from both theoretical and practical points of view [14, 2, 20, 6, 8, 13, 15, 16, 17, 18, 19].

In this paper, we propose a new compression method for DNA sequences and a new search method to search patterns directly in the compressed DNA sequences. In particular, we compress DNA sequences and patterns by using two bits to represent each A, T, C, G character [4]. The search method is our new compressed pattern matching algorithm d-BM, which is based on the Boyer-Moore method. We also present experiments showing that for long patterns our algorithm is more than 10 times faster than the exact match routine based on the Boyer-Moore-Horspool algorithm [12] in the software package Agrep [21, 22], which is known as the fastest available DNA pattern matching program.

Most existing text compression methods fall into two categories: statistical compression and dictionary-based compression. For the former category, Huffman encoding [15] is usually used. The one dealing with Huffman encoded files runs faster than the Aho-Corasick (AC) [4] algorithm compared to the original files by the same factor as the compression ratio. For the dictionary-based compression, Shibata [19] used byte-pair encoding (BPE) [9]. Utilizing a combined Boyer-Moore and BPE algorithm, DNA string matching runs about three times faster than the exact match routine of the software package Agrep. In [6], de Moura, et al. proposed a compression scheme that uses a Huffman coding on words. They presented an algorithm that runs twice as fast as Agrep. However,

Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference (CSB 2004)
0-7695-2194-0/04 $20.00 © 2004 IEEE

the compression method is not applicable to DNA sequences, which cannot be segmented into words.

The rest of the paper is organized as follows: Sections 2 and 3 describe the compression methods for DNA sequences and patterns respectively. Section 4 presents our proposed d-BM algorithm. Section 5 reports our experimental results and analysis, and Section 6 discusses and concludes the paper.

2. Compressing DNA sequences

Given a DNA sequence consisting of A, C, T, G characters, we use two bits to encode each character: "00" for A, "01" for C, "10" for T, and "11" for G. As a result, each byte can represent four DNA characters with each cell of 2 bits representing one DNA character. For example, DNA sequence GACCGTCT is encoded by binary sequence 11 00 01 01 11 10 01 10, which amounts to 8 cells or two bytes.

Since each byte can take 256 distinct values, the above compression method equivalently increases the size of an alphabet Σ from 4 to 256. As will be noted later, this increase of alphabet size significantly contributes to the enhanced performance of our BM-based pattern matching algorithm.

Using this simple encoding scheme, a DNA sequence T of size m is encoded by a compressed DNA sequence T' of $\lceil m/4 \rceil$ bytes. Notice that if $m \bmod 4 \neq 0$, the last byte of T' will only contain $m \bmod 4$ useful cells. For example, DNA sequence TACCGT will be encoded by 10 00 01 01 11 10 xx xx, with the last byte containing two do-not-care cells in the suffix, whose values are not our concern and denoted by "xx" in this paper.

Using this compression scheme, a DNA sequence T is represented as a pair $<T', A>$ where T' denotes the encoded byte sequence, and A is a non-negative integer, called *suffix adornment*, which indicates the number of do-not-care cells in the suffix of the last byte of T'. Obviously, A can only take one of the values from $\{0, 1, 2, 3\}$.

The above encoding scheme will guarantee a space saving approaching *75%.* This will save not only storage, but also I/O time when DNA sequence database is large.

3. Compressing DNA patterns

If we compress a DNA pattern using the same compression scheme described above, then there might exist an alignment mismatch between a compressed pattern P' and a compressed DNA sequence T', which results in not finding some occurrences of original DNA pattern P in DNA sequence T. Figure 1 illustrates such a situation: there exists an occurrence of the input pattern in the input DNA sequence (the shaded area). However, the location of the occurrence of the pattern in the DNA sequence is not at the boundary of bytes in the compression domain. As a result, the pattern will not be found if we apply a string matching algorithm directly.

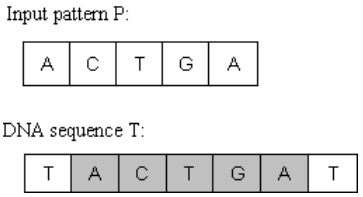

Figure 1 An example of alignment mismatch

To solve this problem, we represent each DNA pattern P by four compressed patterns corresponding to the four alignments, each of which is searched in the compressed DNA sequence. Each compressed pattern takes the form of $<P', A1, A2>$ where P' represents the encoded byte sequence, $A1$ is a non-negative integer, called *prefix adornment*, which indicates the number of do-not-care cells in the prefix of the first byte of P', and $A2$ is another non-negative integer, called *suffix adornment*, which indicates the number of do-not-care cells in the suffix of the last byte of P'. Obviously, both $A1$ and $A2$ can only take one of the values from $\{0, 1, 2, 3\}$. For example, the above "ACTGA" input pattern is represented as four compressed patterns: $<P1', 0, 3>$, $<P2', 1, 2>$, $<P3', 2, 1>$, and $<P4', 3, 0>$ where

- $P1' = 00\ 01\ 10\ 11\ 00\ xx\ xx\ xx$,
- $P2' = xx\ 00\ 01\ 10\ 11\ 00\ xx\ xx$,
- $P3' = xx\ xx\ 00\ 01\ 10\ 11\ 00\ xx$, and
- $P4' = xx\ xx\ xx\ 00\ 01\ 10\ 11\ 00$.

4. Compressed pattern matching (d-BM)

Our compressed pattern matching algorithm is based on the BM algorithm. In the following, we give a brief overview of the BM algorithm first, and then describe how we adapt the BM algorithm to compressed pattern matching in DNA sequences.

4.1. BM algorithm on uncompressed text

The BM algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of a mismatch it uses two preprocessed functions to shift the alignment to the right. These two

shift functions are called *the bad character rule and the good suffix rule*. In the bad character rule, mismatched characters at the right end of the pattern allow a large shift to the next occurrence of the character in the pattern *P*, or to the end of *P* if the character is not present in *P*. According to the good suffix rule, a mismatched character at the left end of a matched substring triggers a shift to the next occurrence of the substring or an identical left end of the substring. If no part of the substring is repeated in *P*, then *P* can be shifted its entire length. Since the bad character rule shift can be negative, the Boyer-Moore algorithm applies the maximum shift obtained with either the good suffix rule or the bad character rule. A detailed description of the Boyer-Moore algorithm is beyond the scope of this paper. Interested readers are referred to [3, 5] for such details.

4.2. d-BM algorithm on compressed DNA sequences

Our d-BM applies the BM algorithm to search the compressed DNA sequence using four compressed patterns instead of one.

As empty cells might appear in the suffix of a compressed DNA sequence and in the prefix and suffix of the compressed input pattern, special care must be taken when comparing bytes appearing in these areas so that matching against empty cells is masked. Consider the search of pattern $<P', A1, A2>$ of length m' in DNA sequence $<T', A>$ of length n'. We need to consider the following four cases:

1) When we match $P'[1]$ to $T'[j]$ ($1 \le j \le n'$-1), the first $A1$ cells of $P'[1]$ should be masked.
2) When we match $P'[i]$ ($2 \le i \le m'$-1) to $T'[j]$ ($1 \le j \le n'$-1), no masks need to be considered. It is a traditional character matching.
3) When we match $P'[m']$ to $T'[j]$ ($1 \le j \le n'$-1), the last $A2$ cells of $P'[m']$ should be masked.
4) When we match $P'[m']$ to $T'[n']$, both the last $A2$ cells of $P'[m']$ and the last A cells of $T'[n']$ should be masked.

If we use the entire length of a pattern to do pattern searching we have to consider all four cases in the whole pattern and the overhead to determine which case to compare will be significant. In order to reduce the overhead, we consider a pattern $<P', A1, A2>$ as $<Pf, Pm, Pb, A1, A2>$ where Pf is the first byte of P', Pb is the last byte of P', and Pm is the byte sequence of P' excluding the first and last bytes of P'. Similarly, we consider a DNA sequence $<T', A>$ as $<Tf, Tb, A>$ where Tb is the last byte, and Tf is the byte sequence of T' excluding the last byte.

Application of the d-BM algorithm is divided into two steps: In the first step, we search four different middle parts of patterns Pmi in the DNA sequence Tf using the *BM* algorithm. In this phase we only need to consider rule 2; if no match is found then the pattern is not present in the DNA sequence. If a match is found, then we start the second step which is extending comparison to Pf with $Tf[i]$ $(0 \le i \le n'-2)$ (rule 1) and to Pb with $Tf[i]$ $(0 \le i \le n'-2)$ *(rule 3)* or Pb with Tb (rule 4). If they match, then a final match is found, otherwise continue searching.

For example, consider searching pattern *P: "TACTTTGGA"* in DNA sequence *T: "GCTACTTTGGATGCT"*. Figure 2(a) shows the corresponding compressed DNA sequence T', Tf, and Tb. For readability, we represent the value of each cell by the character it encodes instead of its binary value. Note that we have $A = 1$.

The algorithm searches four compressed patterns corresponding to the four alignments. First, it searches the middle part $Pm1$ of the first compressed pattern $P1'$ in Tf, no match is found as shown in Figure 2(b).

Second, it searches the middle part $Pm2$ of the compressed pattern $P2'$ in Tf. Similarly, no match is found as shown in Figure 2(c).

Third, it searches the middle part $Pm3$ of the compressed pattern $P3'$ in Tf, a match is found; we continue with the second step which performs the extended comparisons between $Pf3$ and $Pb3$ and their counterparts in T'. Both extended comparisons result in matches since "xxxxTA" matches "GCTA" and "GGAxx" matches "GGAT" (rule 3). This is illustrated in Figure 2(d).

Finally, after searching all four encoded patterns in the encoded DNA text, we report all the occurrences of the patterns if they exist.

Clearly, the second step of d-BM will not be used if the middle patterns were not present in the DNA sequence as determined in the first step. Therefore, the time cost in the first step dominates the whole procedure. We sketch out the d-BM algorithm in pseudocode in Figure 3.

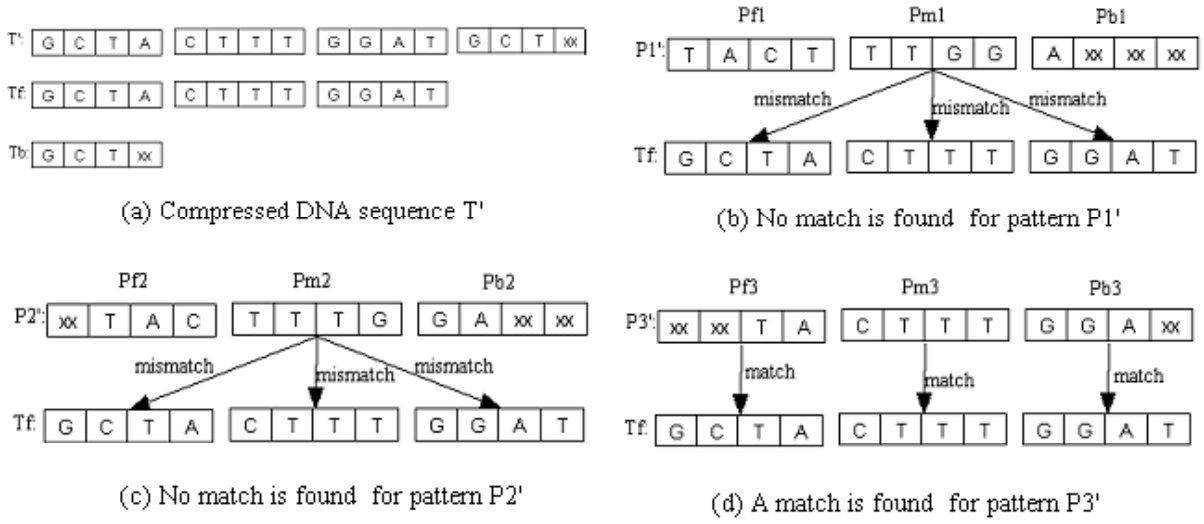

Figure 2 An example of searching pattern P in T

Algorithm d-BM
Input: pattern P and DNA sequence T
Output: report all occurences of P in T

Begin
/* Preprocessing */
Calculate the four compressed patterns <*Pfi, Pmi, Pbi, A1i, A2i*> *(1≤i≤4)*
with length *m'*
Calculate the compressed DNA sequence <*Tf, Tb, A*> with length *n'*
According to *the bad character rule,* compute four charJump tables
with different *Pmi, A1i, A2i*
According to *the good suffix rule,* compute four matchJump
tables with different *Pmi, A1i, A2i*

/* Compressed d-BM pattern matching */
for(four different matchJump tables, charJump tables and *Pm*)
if (BM find an occurrence of *Pm* in *Tf* with position i)
// extend comparison to *Pf* and *Pb*
if(i = *n'* – *m'* + 1) // occurrence present in the end of *Tf*
if(*Pf* ==*Tf*[i-1] && *Pb* == *Tb*) return i; // pattern find
else if(i < *n'* – *m'* + 1) // occurrence present in middle of Tf
if(*Pf* ==*Tf*[i-1] && *Pb* == *Tf*[i+*m'*-2]) return i; // pattern find
else continue;
else continue;
return; // cannot find pattern
End Algorithm

Figure 3 The pseudocode of d-BM

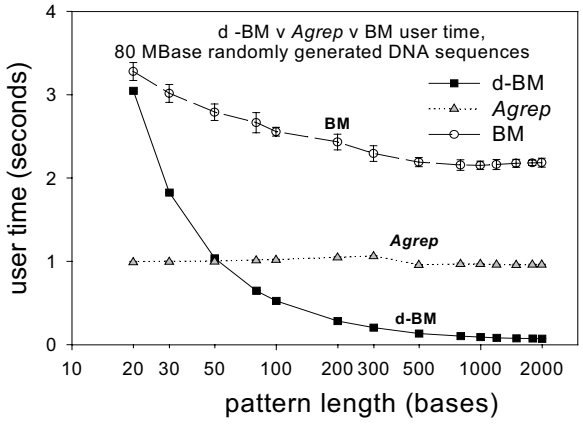

Figure 4 Comparison of d-BM to uncompressed BM and Agrep on synthetic data

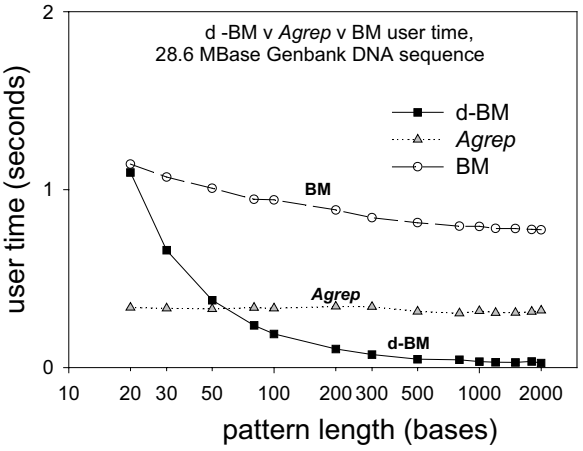

Figure 5 Comparison of d-BM to uncompressed BM and Agrep on real data

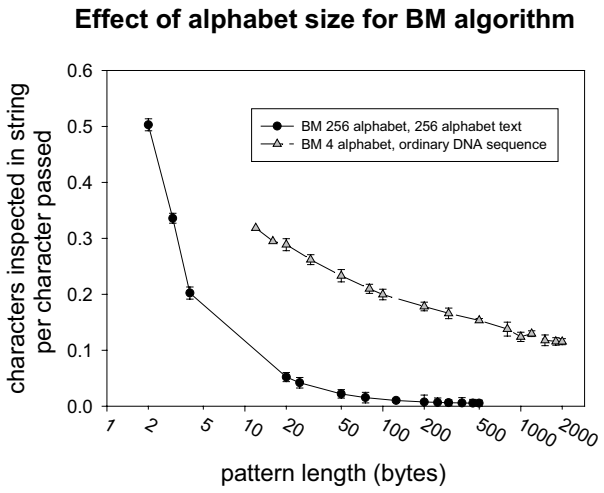

Figure 6 Effect of alphabet size on BM speed

5. Experimental results

5.1. Comparison of d-BM to uncompressed BM and Agrep

Experiments were run on an Intel Pentium III 575MHz machine running Linux RedHat7.2. DNA files that were tested included 80 megabase randomly generated by an in-house program, as well as a 28.6 megabase DNA sequence randomly chosen from Genbank.

To gather the test patterns we wrote a program that randomly selects a substring of a given length from a source string. We used this program to select 30 patterns of length M, for each M from 20 to 2000 and analyzed user times (seconds) in all of the experiments. As illustrated in Figures 4 and 5, d-BM is faster than BM analysis on uncompressed data for patterns > 20 bases in length and as much as 14 times faster than Agrep for patterns > 50 bases in length.

5.2. Effect of alphabet size on BM algorithm speed

Boyer-Moore have mentioned in their paper the total number of instructions executed in order to pass over a character decreases as the length of the pattern increases, and that this effect is enhanced by larger character alphabets. These effects were illustrated by comparison with alphabets of 2, 26, and 100 characters [3].

As described in the section 4.2, d-BM consisted of two steps, and the first step BM search dominates the speed of the algorithm. However, instead of searching in the uncompressed DNA sequence with the alphabet size of 4, we search in the compressed DNA sequence with the alphabet size of 256. Therefore, we examined the BM advantage of enlarging the size of alphabet Σ from 4 to 256. Figure 6 shows that characters inspected in string per character passed of BM is smaller with a 256-character alphabet.

6. Discussions and conclusions

In this paper, we propose a new derivative Boyer-Moore approach to search for long patterns in

compressed DNA files. By working in the compression domain, the algorithm provides a guaranteed space saving of approximately 75% for both DNA file and match pattern and an enhanced pattern matching speed compared to several search methods applied to the same sequences in uncompressed format (Figures 4 and 5). A significant proportion of the improved performance for long patterns can be attributed to the advantage of using a larger alphabet in a BM algorithm (Figure 6).

Shortening the length of the DNA sequence in which the pattern is being matched shortens the search time (compare Figure 4 to Figure 5). However, the 75% fewer bytes used for the compressed DNA sequence compared to the uncompressed sequence does not provide a direct time savings in the d-BM algorithm. Although an encoded pattern could be searched in 1/4 the time in the compressed domain, the algorithm requires that the encoded pattern be run four times, in each of its alignments, P1' to P4'. Nevertheless, due to the smaller number of bytes needed to store the compressed DNA sequence, the input/output time might be saved.

The demonstrated algorithm is less efficient for short patterns, but much more efficient for long pattern. In part the lower efficiency for short patterns is due to the fact that the BM algorithm must examine more characters per character passed when matching short patterns (Figure 6). In fact, the algorithm exaggerates this effect because the number of characters in the pattern being searched is greatly reduced in the compression domain. For example, if the uncompressed DNA pattern m = 16 to 19, the compressed patterns m' = 16/4 = 4 or m' = 16/4 + 1 = 5, and the middle portion Pm that is actually examined has length of Pm = 4 - 2 = 2 or length of Pm = 5 - 2 = 3, respectively. As shown in Figure 6, characters inspected in string per character passed is about 0.3 for the uncompressed pattern of length = 20, but about 0.33 in pattern length 3 and 0.5 in pattern length 2 when alphabet size is 256. This explains why the user time comparing d-BM to application of BM to the uncompressed sequence (Figures 4 and 5) extrapolates to a greater user time for a DNA pattern length of <20. The Agrep program is more efficient for pattern length < 50; however, Agrep does not become more efficient with longer pattern lengths. Hence, the increased efficiency of the d-BM method for long patterns using a 256-character alphabet enables it to operate more than 10 times faster than Agrep.

This 10-fold improvement in exact pattern match searches in DNA sequences presented in this paper is greater than for previously described compressed pattern matching methods. For example, Shibata et al. [19], proposed a BM-type algorithm on genetic data compressed by their BPE algorithm and demonstrated only a 3-fold improvement over Agrep. They described results of their algorithm only for short patterns (length < 30), so it's possible that further improvements for longer patterns may occur; however, they have not reported such results. The BPE compression algorithm achieves only a 30% compression with genomic data and therefore lacks some of the data storage and input/output advantages of the present algorithm. Similarly, Navarro and Tarhio [17] tested a BM string matching approach applied to genomic data compressed by a Ziv-Lempel method. Their fastest method for genomic data (BM-blocks) achieved increases in speed compared to decompressing their DNA sequences followed by searching with Agrep of only about 30%. We obtained an internet accessible version of Navarro and Tarhio's software (www.dcc.uchile.cl/~gnavarro/software) and can confirm that when tested on our computer with the same sequences that we have searched experimentally for Figures 4 and 5 of this paper that our d-BM algorithm is more than 10 times faster than searches using BM-blocks for patterns >100 bp in length. Furthermore, Ziv-Lempel compression achieved only 40% compression for genomic sequences [17].

A final question concerns the types of data in which exact pattern searches may be useful. Approximate pattern matching is a familiar procedure in programs such as BLASTN, for genetic database searches, and CLUSTALW, for multisequence alignments. For distant interspecies comparisons, such as between human and mouse, in fact, it is rare that the distance between consecutive nucleotide substitutions would be as long as 25 bases (e.g., Figure 3 of [7]), and for such comparisons exact pattern matches would usually be inappropriate. However, intraspecies comparisons, such as among groups of human individuals, or interspecies comparisons at the taxonomic order level (e.g., among primates) would be expected to have much longer sequences in common. An example, is the regulatory element of the alpha-globin gene cluster compared among primates [11] which has interspecies and intrahuman exact matches >100 bp in length, along with a significant number of single nucleotide polymorphisms bordering the exact matches. Particularly in the non-coding regions, such long stretches of conserved bases may be indicative of important regulatory elements or other unknown functions. In addition, searches for large mobile genetic elements and insertions and gene rearrangements may be assisted by efficient exact pattern matching algorithms.

Currently, we are investigating a compression scheme so that, instead of searching four compressed patterns, we only need to search for one compressed pattern without losing any match. The result is very encouraging and will be reported in a forthcoming paper.

7. Acknowledgements

We are thankful to the anonymous reviewers for their constructive comments and helpful suggestions that help improve the quality of this paper.

8. References

[1] A. Amir, and G. Benson, "Effcient two-dimensional compressed matching", In *Proc. DCC'92*, pp. 279-288, 2002.

[2] A. Amir, G. Benson and M. Farach, "Let sleeping files lie: Pattern matching in Z-compressed file", *Journal of Computer and System Sciences*, 52, pp. 299-307, 1996.

[3] R. S. Boyer and J.S. Moore, "A fast string searching algorithm", *Communications of the ACM*, 20, 762-772, 1977.

[4] X. Chen, S. Kwong, and M. Li, "A compression algorithm for DNA sequences and its applications in genome comparison", In *Proc. of the 10th Workshop on Genome Informatics (GIW'99)*, pp. 52-61, 1999.

[5] G. Davies and S. Browsher, "Algorithm for pattern matching", *Software-Practice and Experience*, 16, pp. 575-601, 1986.

[6] E. S. de Moura, G. Navarro, N. Ziviani and R. Baeza-Yates, "Direct pattern matching on compressed text", In *Proc. 5th International Symp. on String Processing and Information Retrieval*, IEEE Computer Society, pp. 90-95, 1998.

[7] E.T. Dermitzakis and A. Reymond, R. Lyle, N. Scamuffa, C. Ucla, S. Deutsch, B.J. Stevenson, V. Flegel, P. Bucher, C.V. Jongeneel, and S.E. Antonarakis, "Numerous potentially functional but non-genic conserved sequences on human chromosome 21", *Nature*, 420, 578-582, 2002.

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

[9] P. Gage, "A new algorithm for data compression", *The C Users Journal*, 12, 193, 1994.

[10] D. Gusfield, "Algorithms on Strings, Trees, and Sequences", Cambridge University Press, New York, 1997.

[11] C. L. Harteveld, M. Muglia, G. Passarino, M. Kielman, and L.F. Bernini, "Genetic polymorphism of the major regulatory element HS-40 upstream of the human alpha-globin gene cluster", *British Journal of Haematology*, 119, 848-854, 2002.

[12] R. N. Horspool, "Practical fast searching in strings. *Software-Practice and Experience*, 10, 501-506, 1980.

[13] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa, "Multiple pattern matching in LZW compressed text. In *Proc. Data Compression Conference (DCC'98),* IEEE Computer Society, pp. 103-112, 1998.

[14] U. Manber, "A text compression scheme that allows fast searching directly in the compressed file", In *Proc. 5th Ann. Symp. on Combinatorial Pattern Matching*, Springer-Verlag, pp. 113-124, 1994.

[15] M. Miyazaki, S. Fukamachi, M. Takeda, and T. Shinohara, "Speeding up the pattern matching machine for compressed texts" *Transactions of Information Processing*, Society of Japan, 39, 2638-2648, 1998.

[16] 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*, Springer-Verlag, pp. 14-36, 1999.

[17] G. Navarro and J. Tarhio, "Boyer-Moore string matching over Ziv-Lempel compressed text", In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, Springer-Verlag, pp. 166-180, 2000.

[18] Y. Shibata, T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, and S. Arikawa, "Speeding up pattern matching by text compression", In *Proc. 4th Italian Conference on Algorithms and Complexity*, Springer-Verlag, pp. 306-315, 2000.

[19] Y. Shibata, T. Matsumoto, A. Takeda, T. Shinohara and S. Arikawa, "A Boyer-Moore type algorithm for compressed pattern matching", In *Proc. 11th Ann. Symp. on Combinatorial Pattern Matching*, Springer-Verlag, pp. 181-194, 2000.

[20] M. Takeda, "Pattern matching machine for text compressed using finite state model", Technical Report DOI-TR-CS-142, Department of Informatics, Kyushu University.

[21] S. Wu and U. Manber, "Agrep : a fast approximate pattern-matching tool" In *Usenix Winter 1992 Technical Conference*, pp. 153-162, 1992.

[22] S. Wu and U. Manber, "Fast text searching allowing errors", *Comm. ACM*, 35, 83-91, 1992.