DNA 서열을 위한 빠른 매칭 기법
(Fast Matching Method for DNA Sequences)

김 진 욱 [†] 김 은 상 [††] 안 용 기 [†††] 박 근 수 [††††]
(Jin Wook Kim) (Eunsang Kim) (Yoongki Ahn) (Kunsoo Park)

요 약 DNA 서열은 각 종을 나타내는 근본적인 정보이며, 다른 종 간의 DNA 서열 비교는 중요한 작업이다. DNA 서열은 길이가 매우 길며 또 종의 종류도 다양하기 때문에, DNA 서열 비교에서는 빠른 매칭 뿐만 아니라 효율적인 저장도 중요한 요소이다. 즉, 인코딩 된 DNA 서열에 적합한 빠른 문자열 매칭 방법이 필요하다. 본 논문에서는 매칭 시 디코딩이 필요하지 않은 인코딩 된 DNA 서열을 위한 빠른 매칭 알고리즘을 제시한다. 제시하는 알고리즘은 네 문자 한 바이트 인코딩을 이용하며 서픽스 기법과 다중 패턴 매칭 기법을 접목하고 있다. 실험 결과로는 본 논문에서 제시하는 방법이 AGREP보다 약 다섯 배 빠름을 보이는데, 이는 알려진 알고리즘들 중에서 가장 빠른 결과이다.
키워드 : 문자열 매칭, 인코딩된 DNA 서열, 다중 패턴 매칭

Abstract DNA sequences are the fundamental information for each species and a comparison between DNA sequences of different species is an important task. Since DNA sequences are very long and there exist many species, not only fast matching but also efficient storage is an important factor for DNA sequences. Thus, a fast string matching method suitable for encoded DNA sequences is needed. In this paper, we present a fast string matching method for encoded DNA sequences which does not decode DNA sequences while matching. We use four-characters-to-one-byte encoding and combine a suffix approach and a multi-pattern matching approach. Experimental results show that our method is about 5 times faster than AGREP and the fastest among known algorithms.
Key words : string matching, encoded DNA sequences, multi-pattern matching

1. Introduction

In molecular biology, DNA sequences are the fundamental information for each species and a comparison between DNA sequences is an interesting and basic problem. Since a DNA sequence is represented by a sequence of four bases - A, C, T, and G, the comparison problem is the same as a *matching problem between strings*. There are various kinds of comparison tools and the famous two are BLAST [1] and FASTA [2]. These tools provide approximate matching. In fact, however, they are based on exact matching to speed up.

The exact matching problem is to find all the occurrences of a given pattern P in a large text T, where both T and P are sequences of characters from a finite alphabet Σ. Many algorithms have been developed for exact matching and they are divided into three approaches [3]: Prefix approach, suffix approach, and factor approach. For the prefix approach, there are the Knuth-Morris-Pratt (KMP) algorithm [4], the Shift-Or algorithm [5] and its variants [6]. For the suffix approach, there are the

· This work was supported by FPR08-A1-021 of the 21C Frontier Functional Proteomics Project from the Korean Ministry of Education, Science and Technology.

† 비 회 원 : 인하대학교 컴퓨터정보공학부 교수
gnugi@inha.ac.kr
†† 비 회 원 : 서울대학교 컴퓨터공학부
eskim@theory.snu.ac.kr
††† 학생회원 : 서울대학교 컴퓨터공학부
ykahn@theory.snu.ac.kr
†††† 종신회원 : 서울대학교 컴퓨터공학부 교수
kpark@theory.snu.ac.kr
논문접수 : 2009년 3월 19일
심사완료 : 2009년 5월 15일

Boyer-Moore (BM) algorithm [7] and its variants - the Horspool algorithm [8], the Sunday algorithm [9], and the hybrid algorithm [10]. For the factor approach, there are the Backward Nondeterministic Dawg Matching (BNDM) algorithm [11] and the Backward Oracle Matching (BOM) algorithm [12]. In addition, there exists some results for small alphabets [13-15]. Kim and Shawe-Taylor (KS) and Tarhio and Peltola (TP) use q-gram for shifts in [14] and [15], respectively. The approximate pattern matching tool, AGREP [16,17], has the exact match routine which has been widely used. For more information, see [3].

A comparison between DNA sequences of different species is an important task. Since DNA sequences are very long and there exist many species, not only fast matching but also efficient storage is an important factor for DNA sequences. Thus, a fast string matching method suitable for encoded DNA sequences is needed.

In fact, various encoded string matching algorithms have been developed for general string matching after the compressed matching problem was first defined by Amir and Benson [18]. Manber [19] used an encoding method which compresses the phrase of length 2 to one byte. However, the problem of this method is that the pattern may have more than one encoding. Moura et al. [20] proposed an algorithm which consists of the Sunday algorithm with a semi-static word-based modeling and a Huffman coding. It is 2 times faster than AGREP. Amir, Benson, and Farach [21] gave the first matching algorithm for Ziv-Lempel encoded texts. Recently, for Ziv-Lempel encoding method, Navarro and Raffinot [22] used the Shift-Or algorithm, and Navarro and Tarhio [23] used the BM algorithm. For byte pair encoding method, Shibata et al. [24] used the Knuth-Morris-Pratt algorithm and Shibata et al. [25] used the BM algorithm. The algorithm in [24] is 2 times faster than AGREP for GenBank data set with $m \leq 16$ where m is the pattern length and the algorithm in [25] is 3 times faster with $m \leq 30$.

Recently there have been efforts to develop practical and fast string matching algorithms just for DNA sequences [26,27]. The following algorithms use a similar encoding method which is suitable for small alphabet. Fredriksson [26] used an encoded character called a super-alphabet. A super-alphabet of size σ^s is defined as packing s symbols of text T to a single super-symbol where σ is the size of the alphabet. Fredriksson's Shift-Or algorithm with a super-alphabet of size $4^4 = 256$ is 5 times faster than the original Shift-Or algorithm with DNA sequences. Chen, Lu, and Ram [27] used the following encoding method: each DNA base is encoded to two-bit code and thus four bases can be considered at a time. Chen, Lu, and Ram's d-BM algorithm makes four encoded patterns and then does matching using BM for each encoded pattern. It is faster than AGREP with $m > 50$.

In this paper, we present a fast string matching method for encoded DNA sequences. Our method uses encoded texts for matching and does not decode them. We use four-characters-to-one-byte encoding and combine a suffix approach and a multi-pattern matching approach, i.e., a combination of a multi-pattern version of the Sunday algorithm and a simplified version of the Commentz-Walter algorithm [28,29]. Through this combination, we get the most efficient string matching method for encoded DNA sequences. We implement various algorithms and compare with our method. Experimental results show that our method is about 5 times faster then AGREP and the fastest string matching method for encoded DNA sequences among known algorithms.

2. Preliminaries

We first give some definitions and notations that will be used in this paper. Let Σ be an alphabet and σ be the size of Σ. A string is concatenations of zero or more characters from alphabet Σ. The length of a string S is denoted by $|S|$. Let $S[i]$ denote ith character of a string S for $1 \leq i \leq |S|$ and $S[i..j]$ denote a substring $S[i]S[i+1]\cdots S[j]$ of S for $1 \leq i \leq j \leq |S|$.

Let T be a text string of length n and P be a pattern string of length m. The string matching problem is defined as follows: Given a text T and a pattern P, find all occurrences of P in T.

In this paper, we consider DNA sequences. There are four characters, A, C, T, and G, i.e., $\Sigma = \{A, C, T, G\}$ and thus $\sigma = 4$. The problem we consider in this paper is as follows.

Problem 1. Let T be a text, P be a pattern and T' be an encoded text of T. Given T' and P, find all occurrences of P in the original text T without decoding T'.

There are various encoding methods: Ziv-Lempel, Huffman, BPE, etc. If an encoding method E such that $E(a) = a$ for $a \in \Sigma$ is used, then $T' = T$ and Problem 1 is the same as the string matching problem.

3. Proposed Method

We want to solve Problem 1 for DNA sequences as fast as possible. To do this, we will propose an algorithm that uses the following methods:

- Encoding method : A fixed-length encoding method.
- Matching method : A suffix approach and a multi-pattern matching approach.

We first explain the encoding method of our algorithm, and then explain the matching method of our algorithm.

3.1 Encoding

We use a fixed-length encoding method. Since $\Sigma = \{A, C, T, G\}$, i.e., $\sigma = 4$, we need only two bits for each character. Thus we can define a mapping M:

$$M(\mathrm{A}) = 00,\ M(\mathrm{C}) = 01,$$
$$M(\mathrm{T}) = 10,\ M(\mathrm{G}) = 11.$$

Since there are eight bits in one byte, we can encode four characters to one byte. Given a substring $S[i..i+3]$, we define an encoding method E such that

$$E(S[i..i+3]) = M(S[i]) \parallel M(S[i+1]) \parallel M(S[i+2]) \parallel M(S[i+3])$$

where $\parallel$ is the concatenation operator. Then, the size σ' of the encoded alphabet is $4^4 = 256$.

We explain text encoding and pattern encoding.

3.1.1 Text Encoding

Given a text T, an encoded text T' is defined as $[T_m, T_b, Mask_T]$ where T_m is an encoded byte sequence of length $\lceil n/4 \rceil - 1$, T_b is the last encoded byte and $Mask_T$ is a bit mask for T_b.

The encoded byte sequence T_m is defined as

$$T_m[i] = E(T[4i-3..4i])$$

for $1 \le i \le n'$ and $n' = \lceil n/4 \rceil - 1$. The last encoded byte T_b is defined as

$$T_b = M(T[4n'+1]) \parallel \cdots \parallel M(T[4n'+r]) \parallel \overbrace{00 \parallel \cdots \parallel 00}^{4-r}$$

where $r = n - 4n'$, $1 \le r \le 4$. The bit mask $Mask_T$ is defined as

$$Mask_T = \overbrace{11 \parallel \cdots \parallel 11}^{r} \parallel \overbrace{00 \parallel \cdots \parallel 00}^{4-r}.$$

3.1.2 Pattern Encoding

Given a pattern P, we make four encoded patterns. Since we will match an encoded text and an encoded pattern, using only one encoded pattern we cannot find all occurrences but only some positions such that their positions modulo 4 are all the same. Thus we need four encoded patterns such that the possible occurrence positions of them modulo 4 are 0, 1, 2 and 3, respectively.

An encoded pattern P^i for $0 \le i \le 3$ is defined as $[P_f^i, P_m^i, P_l^i, Mask_f^i, Mask_l^i]$ where P_m^i is an encoded byte sequence of length $\lceil (n+i)/4 \rceil - 2$, P_f^i and P_l^i are the first and last encoded bytes and $Mask_f^i$ and $Mask_l^i$ are bit masks for P_f^i and P_l^i, respectively.

The encoded byte sequence P_m^i is defined as

$$P_m^i[j] = E(P[4j+1-i..4j+4-i])$$

for $1 \le j \le m_i$ and $m_i = \lceil (n+i)/4 \rceil - 2$. The first encoded byte P_f^i is defined as

$$P_f^i = \overbrace{00 \parallel \cdots \parallel 00}^{i} \parallel M(P[1]) \parallel \cdots \parallel M(P[4-i])$$

where $0 \le i \le 3$. The bit mask $Mask_f^i$ is defined as

Table 1 Four encoded patterns for P=ATCAACGAGAGATC

i	P_f^i	P_m^i	P_l^i	$Mask_f^i$	$Mask_l^i$
0	00100100	00011100 11001100	10010000	11111111	11110000
1	00001001	00000111 00110011	00100100	00111111	11111100
2	00000010	01000001 11001100	11001001	00001111	11111111
3	00000000	10010000 01110011 00110010	01000000	00000011	11000000

$$Mask_f^i = \overbrace{00 \| \cdots \| 00}^{i} \| \overbrace{11 \| \cdots \| 11}^{4-i}.$$

And the last encoded byte P_l^i is defined as

$$P_l^i = M(P[4(m_i+1)-i+1]) \| \cdots$$
$$\| M(P[4(m_i+1)-i+r]) \| \overbrace{00 \| \cdots \| 00}^{4-r}$$

where $r = m - 4(m_i+1) - i,\ 1 \le r \le 4$. The bit mask $Mask_l^i$ is defined as

$$Mask_l^i = \overbrace{11 \| \cdots \| 11}^{r} \| \overbrace{00 \| \cdots \| 00}^{4-r}.$$

Note that P_f^i and P_l^i are non-empty bytes.

For example, given a pattern $P=$ ATCAACGAGAGATC, four encoded patterns are shown in Table 1.

3.2 Matching

We combine a suffix approach and a multi-pattern matching approach, i.e., a combination of a multi-pattern version of the Sunday algorithm and a simplified version of the Commentz-Walter algorithm. After the encoding stage, we have one encoded text T' and four encoded patterns P^i for $0 \le i \le 3$. Since we do not decode T' while matching, we must use four encoded patterns. Thus, to get the most efficient performance for matching T' with four encoded patterns P^i, we adopt a multi-pattern matching approach. For each encoded pattern, we adopt a Boyer-Moore approach which shows the best results among known string matching algorithms.

The matching stage consists of two phases: preprocessing phase and searching phase. We first explain the preprocessing phase, and then explain the searching phase.

3.2.1 Preprocessing

In the preprocessing phase, we make a shift table Δ for encoded patterns. The role of the shift table Δ is to find the nearest candidate position for the next occurrence of any one of four encoded patterns at any position.

We make a shift table Δ via two steps. First, we compute a shift candidate table d^i for each encoded pattern P^i. The shift candidate table d^i is in fact a shift table for one pattern matching. We use the method that there must be at least one byte shift at any time [9]. For each P^i, we compute d^i using P_m^i such that

$$d^i[\alpha] = \min\{m_i+1, \min\{m_i+1-k \mid P_m^i[k]=\alpha,\ 1 \le k \le m_i\}\}$$

where α is an encoded character, $0 \le \alpha \le 255$.

Then, we compute a shift table Δ from d^i. Since each d^i means the minimum offset for the next candidate position of P^i, we choose the minimum of d^i to find the next candidate position of four encoded patterns, i.e.,

$$\Delta[\alpha] = \min\{d^i[\alpha] \mid 0 \le i \le 3\}.$$

In addition, we make an index list for each encoded character α. When α appears in the last position of P^i, i.e., $P_m^i[m_i]=\alpha$, i is inserted in the index list for α. Using this, we can search for only the encoded patterns that P_m ends with α.

3.2.2 Searching

In the searching phase, we find all occurrences of four encoded patterns in an encoded text. The searching phase is divided into two parts: match part and shift part. Figure 1 shows a pseudocode for the searching phase.

In the match part, we match each character of encoded patterns and the encoded text. Let i be a pointer to the encoded text T_m. When the index list for the character of $T_m[i]$ contains some entries, we try to match T_m and encoded patterns which are indicated by the index list. Since the

```
while $i \le n'$
  for $r$ in the index list of $T_m[i]$          // match part
    $k \leftarrow i-1$, $j \leftarrow m_r - 1$
    while $j > 0$ and $T_m[k] = P_m^r[j]$
      $k \leftarrow k-1$, $j \leftarrow j-1$
    end while
    if $j = 0$
      if $i < n'$
        if $P_f^r = T_m[k]\,\&Mask_f^r$ and $P_l^r = T_m[i+1]\,\&Mask_l^r$
          pattern occurs at position $4*k-(3-r)$
        end if
      end if
      else if $P_f^r = T_m[k]\,\&Mask_f^r$ and $P_l^r = T_b\&Mask_l^r$
        pattern occurs at position $4*k-(3-r)$
      end if
    end if
  end if
  do                                              // shift part
    $i \leftarrow i + \Delta[T_m[i+1]]$
  while $i \le n'$ and $\Delta[T_m[i]] \ne 1$
end while
```

Fig. 1 Pseudocode for searching phase

first and last characters of the encoded patterns have their bit masks, a text character is masked by using the bit mask of the pattern. If we find a match, output the match position at the original text. After that, we start the shift part.

In the shift part, we move a pointer i from left to right via a shift table Δ as far as possible. At first, i is shifted by $\Delta[i+1]$ [9]. Then, if the encoded text character at the shifted position i has no entry in its index list, shift again. This guarantees that at least one shift occurs at each iteration. Lemma 1 shows the correctness of our algorithm.

Lemma 1. Using a shift table Δ, we can find every candidate position for four encoded patterns.

Proof. Let i be the current position of the encoded text T_m and let α be the encoded character at the position $i+1$ of T_m. There are four values $d^i[\alpha]$ for $0 \le i \le 3$ and, W.L.O.G., let $\Delta[\alpha] = d^0[\alpha]$. Suppose that an encoded pattern P^1_m is matched at $i+d^1[\alpha]-m_1+1$ of T_m and $\Delta[\alpha] < d^1[\alpha]$. Then to prove this lemma, it is sufficient to show that the current position comes to $i+d^1[\alpha]$. Since the current position is now updated to $i+\Delta[\alpha]$, the next shift added to $i+\Delta[\alpha]$ is $\Delta[\beta]$, where $\beta = T_m[i+\Delta[\alpha]+1]$. Because P^1_m occurs at $i+d^1[\alpha]-m_1+1$, we get $\beta = P^1_m[m_1-(d^1[\alpha]-\Delta[\alpha])+1]$. Thus, $d^1[\beta] \le d^1[\alpha]-\Delta[\alpha]$ and $\Delta[\beta] \le d^1[\beta]$. Therefore $\Delta[\alpha]+\Delta[\beta] \le d^1[\alpha]$ and $i+\Delta[\alpha]+\Delta[\beta] \le i+d^1[\alpha]$. Since Δ is larger than 1, after repeating the above step, the current position comes to $i+d^1[\alpha]$, eventually.

3.3 Analysis

The worst case time complexity is $O(n'km')$, where n' is the length of the encoded text T_m, m' is the maximum of the lengths of four encoded pattern P^i_m and k is 4, and the best case time complexity is $O(n'/m'+m'occ)$, where occ is the number of all occurrences of the pattern in the text.

4. Experimental Results

We had experiments with our algorithm FED (fast matching with encoded DNA sequences) and the following algorithms: BM (Boyer-Moore) [7], Horspool [8], Sunday [9], AGREP [16,17], TP (Tarhio and Peltola) [15], TP08 (a new variant of TP for DNA) [30], KS (Kim and Shawe-Taylor) [14], BOM (Backward Oracle Matching) [12], BNDM (backward nondeterministic dawg matching) [11], BB (BM on byte pair encoding) [25], SASO (super-alphabet shift-or) [26], and d-BM [27]. BB is the compressed pattern matching algorithm on byte-pair encoding and SASO, d-BM and FED are the compressed pattern matching algorithms on fix-length encoding. Others are the original pattern matching algorithms.

We implemented all algorithms by ourselves, except AGREP, BOM, and BNDM. All the algorithms had been implemented in C, compiled with gcc 4.3.2. We ran the experiments in 2.4GHz Intel Core2 Quad CPU with 8 GB RAM, running GNU/Linux 2.6.27.12.

We had experiments on ten real DNA sequence data sets from NCBI: The sizes of data sets are varied from 7.6MB to 220MB. For each data set, the patterns were randomly extracted from the texts, and each test was repeated 50 times. Since the experimental results are similar, we show typical two examples: *Mus Musculus* fragments set (220MB) and *Homo sapiens* chromosome 1 long sequence (210MB). We report the average time in milliseconds which includes the pattern encoding, preprocessing and searching times. Figures 2-5 shows the experimental results.

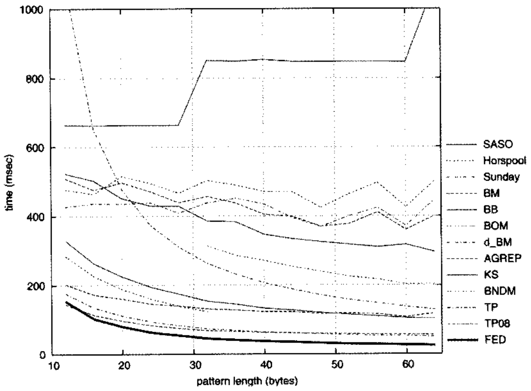


Fig. 2 Running time for *Mus Musculus* fragment set with the pattern length between 12 and 64

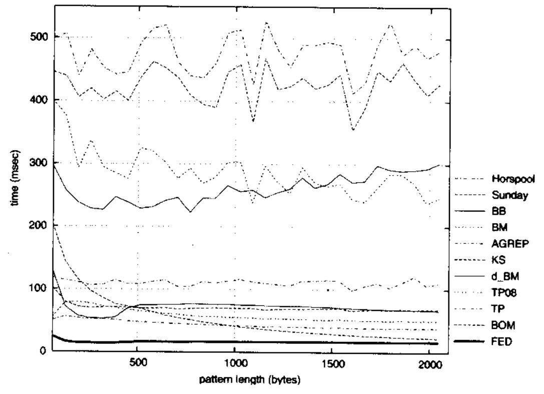


Fig. 3 Running time for *Mus Musculus* fragment set with the pattern length between 64 and 2016

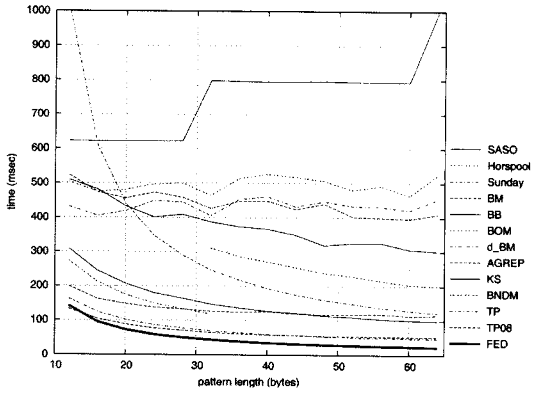


Fig. 4 Running time for *Homo sapiens* chromosome 1 with the pattern length between 12 and 64

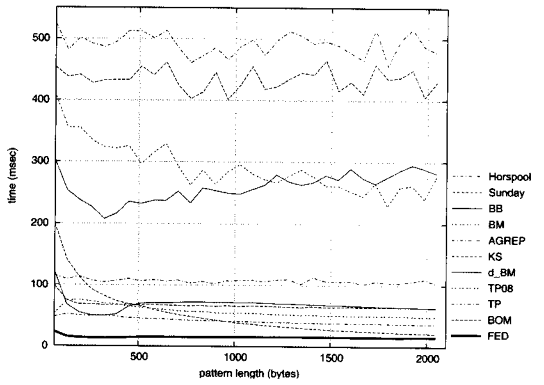


Fig. 5 Running time for *Homo sapiens* chromosome 1 with the pattern length between 64 and 2016

The proposed algorithm FED is faster than all the others from short patterns to long patterns. Figures 2 and 4 show the running times for short patterns whose lengths are from 12 to 64. FED is 2~5 times faster than AGREP and 2~3.5 times faster than TP. In addition, FED is at least 3 times faster than BNDM. A recent algorithm FAOSO (fast average optimal shift or) in [6] is reported about 2 times faster than BNDM on DNA sequences, and thus FED is still faster than FAOSO. Figures 3 and 5 show the running times for long patterns whose lengths are from 64 to 2016. FED is 5 times faster than AGREP and 2.5~5 times faster than TP. As the lengths of patterns get larger, BOM nears FED but FED is still faster than BOM even with pattern length 4000.

5. Conclusions

We have presented a string matching algorithm suitable for encoded DNA sequences and shown that our algorithm is the fastest among known algorithms. In addition, since the matching process is done with the encoded text as it is, we can save the time and space overhead for decoding.

References

[1] BLAST, http://www.ncbi.nlm.nih.gov/BLAST

[2] FASTA, http://www.ebi.ac.uk/fasta

[3] Gonzalo Navarro and Mathieu Raffinot. Flexible Pattern Matching in Strings: Practical On-Line Search Algorithms for Texts and Biological Sequences. Cambridge University Press, 2002.

[4] D. E. Knuth, J. H. Morris Jr, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 6:323-350, 1977.

[5] Ricardo Baeza-Yates and Gaston H. Gonnet. A New Approach to Text Searching. *Communications of the ACM*, 35(10):74-82, 1992.

[6] Kimmo Fredriksson and Szymon Grabowski. Practical and Optimal String Matching. *12th International Symposium on String Processing and Information Retrieval*, Lecture Notes in Computer Science, 3772:376-387, 2005.

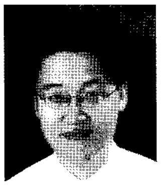
[7] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762-772, 1977.

[8] R. Nigel Horspool. Practical Fast Searching in Strings. *Software Practice and Experience*, 10(6): 501-506, 1980.

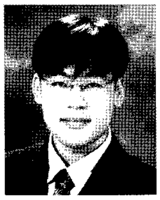
[9] Daniel M. Sunday. A Very Fast Substring Search Algorithm. *Communications of the ACM*, 33(8): 132-142, 1990.

[10] Frantisek Franek, Christopher G. Jennings, and W.

F. Smyth. A Simple Fast Hybrid Pattern-Matching Algorithm. *16th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 3537:288-297, 2005.
[11] Gonzalo Navarro and Mathieu Raffinot. Fast and Flexible String Matching by Combining Bit-Parallelism and Suffix Automata. *ACM Journal of Experimental Algorithmics*, 5(4), 2000.
[12] Cyril Allauzen, Maxime Crochemore, and Mathieu Raffinot. Efficient experimental string matching by weak factor recognition. *12th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 2089:51-72, 2001.
[13] Christian Charras, Thierry Lecroq, and Joseph Daniel Pehoushek. A Very Fast String Matching Algorithm for Small Alphabets and Long Patterns. *9th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 1448:55-64, 1998.
[14] J.Y. Kim and J. Shawe-Taylor. Fast String Matching Using an n-gram Algorithm. *Software: Practice and Experience*, 24(1):79-88, 1994.
[15] Jorma Tarhio and Hannu Peltola. String Matching in the DNA Alphabet. *Software-Practice and Experience*, 27(7):851-861, 1997.
[16] Sun Wu and Udi Manber. Fast Text Searching Allowing Errors. *Communications of the ACM*, 35(10):83-91, 1992.
[17] Sun Wu and Udi Manber. AGREP - A Fast Approximate Pattern-matching Tool. *the Winter 1992 USENIX Conference*, pp.153-162, 1992.
[18] Amihood Amir and Gary Benson. Efficient Two-Dimensional Compressed Matching. *Data Compression Conference*, pp.279-288, 1992.
[19] Udi Manber. A Text Compression Scheme That Allows Fast Searching Directly in the Compressed File. *ACM Transactions on Information Systems*, 15(2):124-136, 1997.
[20] Edleno Silva de Moura, Gonzalo Navarro, Nivio Ziviani, and Ricardo Baeza-Yates. Direct Pattern Matching on Compressed Text. *5th International Symposium on String Processing and Information Retrieval*, IEEE Computer Society, pp. 90-95, 1998.
[21] Amihood Amir, Gary Benson, and Martin Farach. Let Sleeping Files Lie: Pattern Matching in Z-compressed Files. *5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp.705-714, 1994.
[22] Gonzalo Navarro and Mathieu Raffinot. Practical and Flexible Pattern Matching over Ziv-Lempel Compressed Text. *Journal of Discrete Algorithms*, 2(3):347-371, 2004.
[23] Gonzalo Navarro and Jorma Tarhio. LZgrep: a Boyer-Moore String Matching Tool for Ziv-Lempel Compressed Text. *Software-Practice and Experience*, 35(12):1107-1130, 2005.
[24] Yusuke Shibata, Takuya Kida, Shuichi Fukamachi, Masayuki Takeda, Anymi Shinohara, Takeshi Shinohara, and Setsuo Arikawa. Speeding Up Pattern Matching by Text Compression. *4th Italian Conference on Algorithms and Complexity*, Lecture Notes in Computer Science, 1767:306-315, 2000.
[25] Yusuke Shibata, Tetsuya Matsumoto, Masayuki Takeda, Ayumi Shinohara, and Setsuo Arikawa. A Boyer-Moore Type Algorithm for Compressed Pattern Matching. *11th Annual Symposium on Combinatorial Pattern Matching*, Lecture Notes in Computer Science, 1848:181-194, 2000.
[26] Kimmo Fredriksson. Shift-Or String Matching with Super-Alphabets. *Information Processing Letters*, 87(4):201-204, 2003.
[27] Lei Chen, Shiyong Lu, and Jeffrey Ram. Compressed Pattern Matching in DNA Sequences. *IEEE Computational Systems Bioinformatics Conference (CSB 2004)*, pp.62-68, 2004.
[28] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. *6th International Colloqium on Automata, Languages, and Programming*, Lecture Notes in Computer Science, 71:118-132, 1979.
[29] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. Technical Report TR 79.09.007, IBM Germany, Heidelberg Scientific Center, 1979.
[30] Petri Kalsi, Hannu Peltola, and Jorma Tarhio. Comparison of Exact String matching Algorithms for Biological Sequences. *2nd International Conference, Bioinformatics Research and Development*, pp.417-426, 2008.



김 진 욱
1998년 서울대학교 수학과 학사. 2000년 서울대학교 컴퓨터공학과 석사. 2006년 서울대학교 전기·컴퓨터공학부 박사. 2006년~2009년 ㈜에이치엠연구소 책임연구원. 2009년~현재 인하대학교 컴퓨터정보공학부 연구교수. 관심분야는 컴퓨터이론, 알고리즘, 웹검색, 생물정보학, 암호학



김 은 상
2005년 서울대학교 컴퓨터공학부 학사 2005년~현재 서울대학교 전기·컴퓨터공학부 박사과정. 관심분야는 컴퓨터이론, 알고리즘, 웹검색, 암호 및 보안

안 용 기
2007년 서울대학교 컴퓨터공학부 학사
2007년~현재 서울대학교 전기·컴퓨터공학부 석사과정. 관심분야는 컴퓨터이론, 알고리즘, 웹검색

박 근 수
정보과학회논문지 : 시스템 및 이론 제 36 권 제 2 호 참조