Technology and Health Care 29 (2021) S115–S124
DOI 10.3233/THC-218012
IOS Press

A novel optimal multi-pattern matching method with wildcards for DNA sequence

Xinlu Wang[a,b], Ahmed A.F. Saif[a], Dayou Liu[a], Yungang Zhu[a,*] and Jon Atli Benediktsson[c]
[a]*College of Computer Science and Technology, Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education, Jilin University, Changchun, Jilin 130012, China*
[b]*College of International Education, Jilin University, Changchun, Jilin 130012, China*
[c]*Faculty of Electrical and Computer Engineering, University of Iceland, Reykjavik 102, Iceland*

Abstract.
BACKGROUND: DNA sequence alignment is one of the most fundamental and important operation to identify which gene family may contain this sequence, pattern matching for DNA sequence has been a fundamental issue in biomedical engineering, biotechnology and health informatics.
OBJECTIVE: To solve this problem, this study proposes an optimal multi pattern matching with wildcards for DNA sequence.
METHODS: This proposed method packs the patterns and a sliding window of texts, and the window slides along the given packed text, matching against stored packed patterns.
RESULTS: Three data sets are used to test the performance of the proposed algorithm, and the algorithm was seen to be more efficient than the competitors because its operation is close to machine language.
CONCLUSIONS: Theoretical analysis and experimental results both demonstrate that the proposed method outperforms the state-of-the-art methods and is especially effective for the DNA sequence.

Keywords: DNA sequence, multi-pattern matching, wildcards, packed string matching

1. Introduction

In recent years, the biological data has significantly increased. DNA sequence is typical biological data that contains basic information of species [1,2]. DNA sequencing is important in practical areas such as life and agricultural sciences. But nowadays, the DNA sequence data with massive size is a significant computational challenge [3]. Therefore, efficient DNA sequences processing is a major problem to cope with. Pattern matching between different species, i.e., pattern matching for DNA sequence has been a fundamental issue in biomedical engineering, biotechnology and health informatics.

In bioinformatics, DNA sequence alignment is one of the most fundamental and important operation to identify which gene family may contain this sequence. It can be inferred the biological function the sequence may have by DNA sequence alignment. The mechanism of DNA sequence alignment allows the target sequence to be compared with other sequences to find regions of local similarity [1], this process can be modelled as pattern matching problem, in which a pattern is a sequence of DNA characters (A, C, G, T) that is to be searched in a DNA sequence or chromosome [3].

*Corresponding author: Yungang Zhu, College of Computer Science and Technology, Jilin University, No. 2699 Qianjin Street, Changchun, Jilin 130012, China. Tel.: +86 13596429585; E-mail: zhuyungang@jlu.edu.cn.

Optimal multi pattern matching for DNA sequence is a challenging issue. For example, when we want to find the locational of biological virus in the genetic sequence, since the virus is not simply copied, some short sequence fragments may be inserted and deleted between two adjacent characters, in this case, pattern matching algorithm with wildcards is necessary [4].

In this paper, we propose a novel algorithm for optimal multiple pattern matching with wildcards for DNA sequence based on a packed string. The algorithm has a pre-processing time complexity of $O(|P|)$ which $|P|$ is the length of pattern P. In cases when the patterns are short enough such that every pattern can be encoded with a single machine word the algorithm runs in $O(n + nk)$ time in worst case, n is the length of main text, and k is the number of pattern strings. The performance of the algorithm can be improved by filtering the patterns according to how many different symbols the pattern is comprised of and the number of characters which occur only once, and the number of wildcard symbols. Special string-matching instructions can also improve the algorithm performance and extend the algorithm to approximate matching algorithm.

The remainder of this paper is organized as follows: Section 2 provides the problem definition and notations. The proposed method is described in Section 3, and then a theoretical analysis of the method is outlined. The experimental results are presented in Section 4. Lastly, conclusions and future works are summarized in Section 5.

2. Definition and notations

The aim of this paper is to find the occurrence of each pattern from set $P = \{P^1, \ldots, P^k\}$ in a text T with $P^j = p^{j1}p^{j2}\ldots p^j_{mj}$ and $T = t_0t_1\ldots t_n$ such that $t_i \in \sum \cup \{*\}$ and $p^j_k \in \sum \cup \{*\}$. The notations are explained as follows:

- m_j denotes the length of the j_{th} pattern in the pattern set;
- $m_{\max}$ denotes the length of the longest pattern in the pattern set;
- $|P|$ denotes the sum length of pattern set P;
- n denotes the length of text, where $n > m_{\max}$;
- l denotes the length of matched blocks using the hash values;
- o denotes the number of matched blocks using the hash values;
- d denotes the number of wildcard symbols in the matched blocks that using the hash values including pattern partial matching.
- *occ* denotes the number of occurrences for pattern P in text T;
- $\sum$ denotes an alphabet;
- σ denotes the size of $\sum$, i.e., the number of symbols in the alphabet;
- ε denotes an arbitrary small constant, where $\varepsilon > 0$;
- α denotes an integer parameter satisfied $\alpha \leqslant m \leqslant n\alpha$;
- g denotes the number of wildcards in the given pattern;
- $L(P)$ denotes the total length of individual keywords in every pattern of the pattern set;
- D is the distribution of symbols over the patterns.

Let $\{P^1, P^2 \ldots, P^k\}$ be a set of k patterns, the notations, i.e., m, P, n, T, $\sum$ and σ are the same as above. A wildcard symbol can match any symbol including itself. The pattern $P^j = p^{j1}p^{j2}\ldots p^j_{mj}$ denotes a pattern where every character is saved in a single machine word while $P^{j'} = p^{j'1}p^{j'2}\ldots p^{j'}_{m'j}$ denotes the corresponding packed string pattern that several characters are packed into a machine word. $T = t_1t_2\ldots t_n$ denotes a text where every character is saved in a single machine word, and

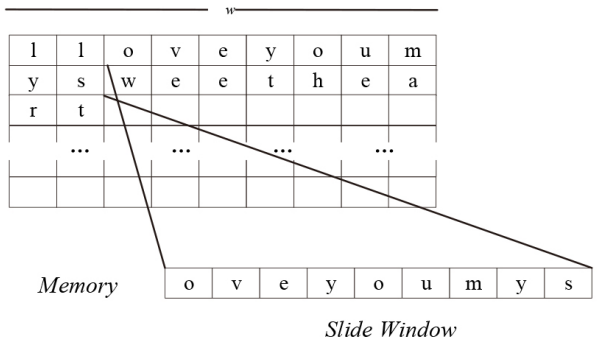

Fig. 1. Slide window on $w = 64$, $\gamma = 7$ and $\alpha = 9$ packed string of text.

$T = t[1]t[2] \dots t[n]$ denotes the corresponding packed string text. Multiple characters can be packed into a larger word with a machine word length. The packing process will be given in the following chapter. w denotes the machine word length, $t[w]$ is a sliding window, $\gamma = \log \sigma$ denotes the number of bits that is enough to store a single character in a single machine word, and $\alpha = w/\gamma$ denotes the number of characters that can be packed into a single machine word.

3. The proposed methodology

The main idea of the proposed algorithm is to pack each $w/\log_2 \sigma$ symbols of text and patterns into a single machine word. A single matching word slides along whole text, and in every shifting step, a packed string in a sliding window is compared with a stored packed string of patterns, as shown in Fig. 1. The comparison process includes two sub-processes. The first sub-process is to check if there is a match and the second sub-process is for double checking in order to avoid false matching. The first sub-process is based on shift and "&" called "AND bit-wise operations", while the second sub-process is based on shift and "|" called "OR bit-wise operations".

Assume that the machine word w is of a standard size i.e., of either 32 or 64 bits. Then, every machine word can contain up to $\alpha = w/\log_2 \sigma$ symbols, where σ is the alphabet size. Furthermore, $\gamma = \log_2 \sigma$ is the number of bits that is used to encode one symbol from the alphabet. In instance, the first 100 MB of Wikipedia English language (enwik8) has 127 different symbols, i.e., $\sigma = 127$, assume $w = 64$, then $\gamma = 7$, $\alpha = 9$. One machine word can include 9 characters.

The pre-processing phase is first executed to; encode every symbol in text and patterns with a unique integer number and wildcards with 0's. All the pattern symbols are packed to machine words. Let us assume now that every pattern can be encoded to a single machine word, i.e., $m < \alpha$, and every comparison process will only take $O(1)$ time. The packaging process is executed by shifting OR bitwise-operations as shown below:

$$\begin{aligned} &P_{[j]} = 0, P_{[j]} = P_{[j]}|p_{[j]1} \\ &P_{[j]} = P_{[j]} << \gamma, P_{[j]} = P_{[j]}|p_{[j]2} \\ &\dots \\ &P_{[j]} = P_{[j]} << \gamma, P_{[j]} = P_{[j]}|p_{[j]mj} \end{aligned} \tag{1}$$

The result is stored as a packed string pattern, $p[j]$, for every pattern that will be used in the comparison process phase.

3.1. Sliding window updating process

The text T is packed in the same way as the patterns are packed. A computer machine word is used as a sliding window $t[w]$ with α being the size of the characters. The α-sliding window is initially constructed from the first α characters of the text t. Then, the packed string in the sliding window is compared with the packed string in the stored patterns.

3.2. Comparison process

Comparison of a packed string in a sliding window with a packed string of stored patterns can be achieved as follows. Let us assume that the size of the sliding window and the size of each pattern are identical, and that the wildcard bits in a packed string of patterns are all 1's and the wildcard bits in the text are all 0's. Then, we have

$$\begin{aligned} &p^y = t[w]\&(\sim P^{[j]}) \\ &\text{if } \textit{Temp} = 0, \text{pattern } P^{[j]} \text{ has a match} \end{aligned} \tag{2}$$

The complement of the packed string of a pattern is first computed. Then, an AND bit-wise operation is executed between the packed string of the pattern and the sliding window, where the bits of wildcards are all 0's, both in the sliding window and in the complement of the packed string of the pattern. A match exists for the pattern at position i if the result at position i of the text is 0. Otherwise, the match could not exist.

The second process is called as double checking. In this process, the OR bit-wise operation is executed between the complement of a packed string of pattern and a packed string of text. During the OR-ing process, the wildcard bits should all be 1's. If the bits of a resulting word are all 1's, then the pattern $p[j]$ has a match. It is important to notice in the AND bit-wise process, the wildcard bits are all 0's, while in the OR bit-wise process they are all 1's:

$$\begin{aligned} &\textit{Temp} = t[w]\&(\sim P^{[j]}) \\ &\text{if } \textit{Temp} = 111\ldots1, \text{pattern } P^{[j]} \text{ has a match} \end{aligned} \tag{3}$$

To check the match of the remaining part of the word if a pattern cannot be packed into a single machine word, i.e., $m > \alpha$, an additional process with a partial matched pattern is needed. A pseudo code of the proposed algorithm is described as follows:

A Pseudo Code of the Proposed Algorithm

(1) Input a set of patterns $\{P^1, P^2 \ldots P^k\}$ and text T.
(2) Encode all the symbols in patterns and text with unique number and wildcards with 0.
(3) Pack each α character of patterns into a machine word.
   a. γ bits left shift $P'[] = P'[] << \gamma$
   b. OR the new character with shifting result. $P' = P'|p_i$
(4) Pack the first α character of text into machine word as a sliding window.
   a. γ bits left shift $T[] = T[] << \gamma$.
   b. OR the new character with shifting result. $T[] = T[]|t_j$.
(5) Compare the packed string in the sliding window with the packed string of every patterns.
   a. Change wildcards bits to 0's for both the sliding window and the complement of pattern.
   b. AND the packed string of sliding window the complement of every patterns.
   c. If the result is 0.

Table 1
Comparison of algorithms solving the problem of pattern matching with fixed number of wildcards

Based-algorithm	Running time	LP	BA	NPU
Packed string	$O(n + nk + occ)$	×	×	√
Euclidian distance	$O(occ \log k + n \log \lvert P \rvert)$	×	×	√
Humming distance	$O(occ \log k + n \log \lvert P \rvert \log \sigma)$	×	×	√
Prime number encoding	$O(occ \log k + n \log m)$	×	×	√
Euclidian distance and hash function	$O(d + o + kn \log l)$	√	√	√
Bit-parallel	$O(d(m/w)(n/\sigma) + n)$	√	√	√

c_1. Change the wildcard of the sliding window and the complement of patterns to all of 1.
c_2. OR packed string of sliding window and the complement of every pattern.
c_3. If the bits of the result of pattern p^y are all 1, report that p^y has a match.

(6) Shift sliding window by γ and go to step 4.

3.3. Algorithm analysis

Assuming that pattern set P with k patterns has a total length $|P|$. In every shifting step, one character is packed. The packing process consists of two bit-wise operations (left shift and OR bit-wise operations), both with $O(1)$ time. Therefore, the complexity of the packing process is $2|P|$ in terms of time and can be written as $O(|P|)$ time. Thus, the complexity of the pre-processing algorithm is $O(|P|)$.

In each shifting time of updating the sliding window, only one character is added to the sliding window, so the time complexity of updating sliding window is$O(|n|)$. The comparing process consists of two main processes, AND-ing and OR-ing that are executed between a packed string of sliding window and a packed string of k patterns. After the execution of the AND-ing process, other process checks are done if the bits of the result are zeros. And after the OR-ing process, the other process-checks are done if the bits of result are all 1's. The complexity of the AND-ing process is 2 nk time, which can be written as $O(nk)$ time complexity, and the same time for the OR-ing process. In case all the pattern lengths are less than α, the complexity of the comparison process is $O(nk)$ time. In contrast, the time complexity is $O(nk + occ)$ time if the pattern lengths are longer than α. In the worst case, the total complexity of the algorithm performance is $O(n + nk + occ)$ time. Pattern filtering, with character variation and the number of characters being the main filtering factors, can speed up the algorithm.

Table 1 shows a theoretical comparison between our algorithm and other five recently proposed algorithms. The comparison includes the algorithmic performance and support for matching with Long Pattern (LP), Big Alphabet size (BA) and Non-Interrupted Pattern Updating (NPU).

4. Experimental results

Three data sets were used to test the performance of the proposed algorithm. The first two data sets source from natural language data and the last one sources from a DNA sequence data set. The first nature language data set is from enwik8 (Wiki.) corpus that compromises the first 100 MB of Wikipedia in English language with an alphabet size of 127 different symbols. The second natural language data set is the 4.245 MB King James Version (KJV) of the Bible with an alphabet size of 79 different symbols. The third data set is 96.7 MB hs13.chr file corresponding to the plain ASCII form of the human chromosome 13 (DNA) and the alphabet size is 10 different symbols. Human chromosome 13 is selected because it has moderate size, and it is related to breast cancer, which makes it significant to be analyzed.

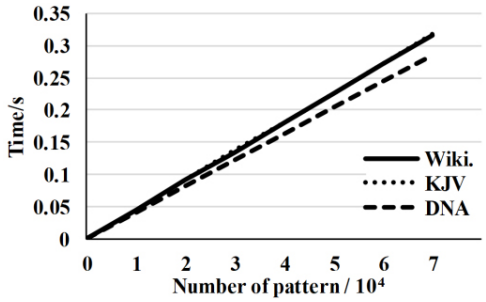

Fig. 2. Pre-processing time for the proposed algorithm applied on three data sets.

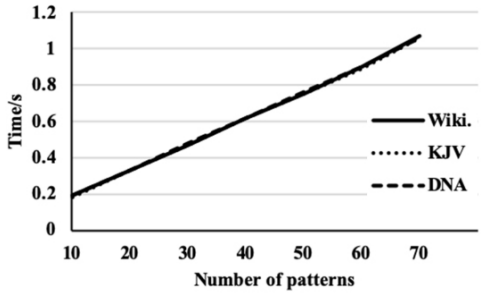

Fig. 3. Performance of the proposed algorithm on three data sets.

The experiments were conducted on a 2.16 GHz Pentium (R) dual 64-bit CPU computer with 2 G RAM and 32-bit Microsoft Windows 8 operating system; we utilized C++ language and Microsoft Visual Studio 2005 to implement the algorithms. We conducted the bench marking of the algorithms in a consistent and controlled environment, and the wildcard character was selected randomly.

The performance of the proposed algorithm was tested against k patterns. Two types of experiments were performed. In these two experiments, a full machine word with 64 bits was used. In the first experiment we used an alphabet size of 256, i.e., $\gamma = 8$ bits, and then 8 characters of maximum pattern-length that can deal with the three data sets. The second experiment was only on the DNA data set but in that data set the size of alphabet is 16, i.e., $\gamma = 4$ bits. In the implementation of the algorithm, simple and basic bit-wise instructions were used such as OR, AND, shift and complement bit-wise operations. We did not use any complex or special string instructions in order to guarantee that every operation executes in $O(1)$ time. Because the compared algorithms belong to exact matching, they are not approximate matching or based on random algorithms such as genetic algorithm or machine learning algorithms, i.e., the precisions of matching are almost equal, so the main difference on performance is the running time of the algorithms.

The proposed method ran efficiently in the pre-processing step on the three data sets. Figure 2 shows that for the natural language data set, the pre-processing time was almost identical for both the Wiki and KJV datasets, but it was less for the DNA data set. The figure also shows that the pre-processing time was linear as a function of the number of patterns for all three data sets. In a numerical form, the pre-processing time in sec. for the three data sets, the proposed algorithm can process one hundred thousand of patterns in less than half a second of where every pattern has 8 characters. This means that the algorithm can process almost one million characters in less than half a second.

Figure 3 shows that the algorithm performance on the three data sets. From the figure it can be observed

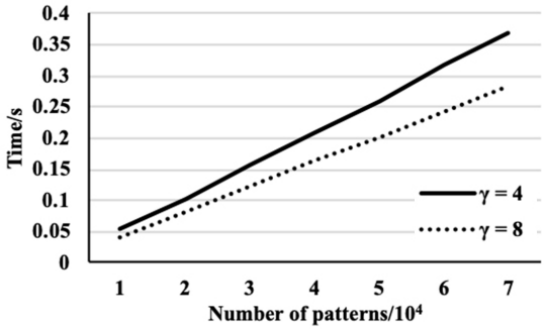

Fig. 4. Comparison of pre-processing on the DNA data set where $\gamma = 4$ and $\gamma = 8$ for a big alphabet size data set.

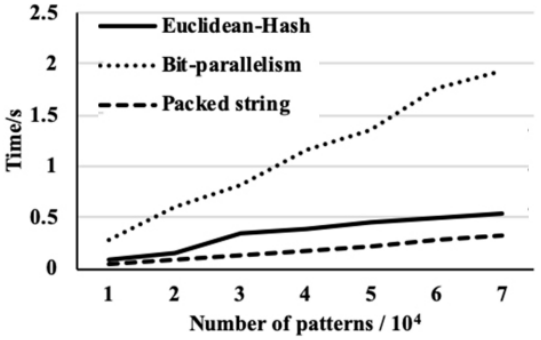

Fig. 5. The time of pre-processing for three algorithms on the Wikipedia data set.

that the time complexity for the three data sets is identical and linear. The experimental results shown in Fig. 4 are achieved on the data set that has the smallest alphabet size, i.e., the DNA data set. The dataset is with 10 different symbols from an alphabet where $\gamma = 4$ bits can represent all the symbols, and 64 bits can store up to 16 characters. Figure 4 shows the pre-processing time complexity comparison for a big alphabet size data set when $\gamma = 4$ and $\gamma = 8$. The pre-processing time complexity is the same for the two cases. For the case $\gamma = 8$, the algorithm runs more efficiently than when $\gamma = 4$, because the number of processed characters in the case of a $\gamma = 4$ machine word can save 16 characters (each character needs only one shift and one OR bit-wise operation).

We then compare the performance of the proposed algorithm with the performance of two recent algorithms. The first one is based on the Euclidean distance and the hash function [5] and the second is based on bit-parallelism [6]. Figures 5–7 show the pre-processing time of the three algorithms against the number of patterns for the three data sets. The figures also show that the bit-parallelism based algorithm is less efficient in the pre-processing step than the other two algorithms where the packed string-based algorithm is the most efficient. While the pre-processing time of the packed string-based algorithm is close in pre-processing time with the Euclidean distance and the hash function-based algorithm.

Tables 2–4 show the performance of the algorithms in seconds as a function of the number of patterns on the first 1 MB of the three data sets. As shown in the tables, for the three data sets, the bit-parallelism-based algorithm and the packed string-based algorithm demonstrate a close performance. Of the two, a better performance is observed for the packed string-based algorithm proposed in this paper. Again, it should be stressed that the Euclidean distance and hash function-based algorithm has less efficiency

Table 2
The performance (time/s) of three algorithms on the Wikipedia data set

Number of patterns	Euclidean and hash	Bit-parallelism	Packed string
10	134.4	0.414	0.188
20	292.8	0.715	0.328
30	451.2	1.004	0.476
40	609.6	1.281	0.613
50	768	1.578	0.758
60	926.4	1.877	0.899
70	1084.8	2.127	1.070

Table 3
The performance (time/s) of three algorithms on the KJV data set

Number of patterns	Euclidean and hash	Bit-parallelism	Packed string
10	145.28	0.419	0.184
20	303.36	0.737	0.332
30	461.44	1.054	0.473
40	619.52	1.378	0.614
50	777.6	1.705	0.754
60	935.68	2.026	0.894
70	1093.76	2.343	1.059

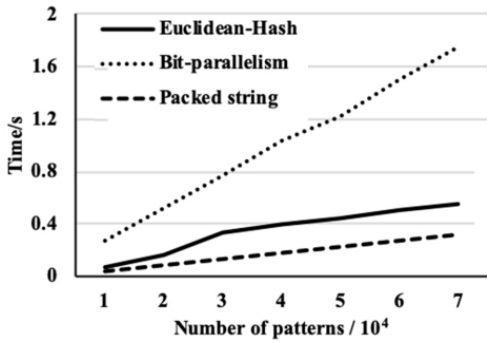

Fig. 6. The time of pre-processing for three algorithms on the KJV data set.

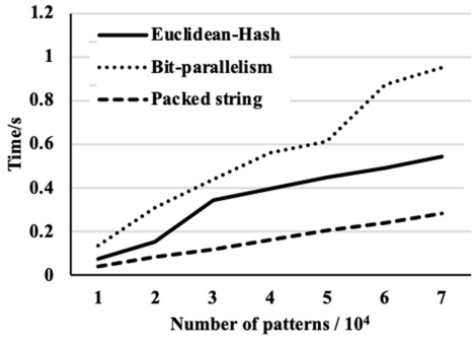

Fig. 7. The time of pre-processing for three algorithms on the data set.

Table 4
The performance (time/s) of three algorithms on the DNA data set

Number of patterns	Euclidean and hash	Bit-parallelism	Packed string
10	154.88	0.327	0.191
20	310.4	0.578	0.352
30	465.92	0.824	0.477
40	621.44	1.063	0.617
50	776.96	1.312	0.762
60	932.48	1.566	0.902
70	1088	1.825	1.063

because it uses the Fast Fourier Transform (FFT) which has many interior variables which slows down the algorithm.

5. Conclusions and future works

Pattern matching for DNA sequence has been an essential issue in biomedical engineering and health informatics. The main finding of this article is to propose a faster pattern matching for DNA sequence, i.e., a practical and efficient algorithm for multi-pattern matching with a fixed number of wildcards is proposed. The main idea of the proposed method is to consider matching patterns as packed string patterns with a sliding window. The window slides along the given packed string text, matching against stored packed string patterns. Three data sets are used to test the performance of the proposed algorithm and compare with the state-of-the-art methods. The proposed algorithm was seen to be more efficient than the competitors because its operation is close to machine language (the algorithm operates machine word, so CPU can directly interpret the data), the proposed algorithm is especially effective for the DNA sequence. In future work, the efficiency of algorithms can be improved by merging the proposed bit-parallelism and string packing algorithm with one or more mechanisms such as Euclidean distance and Hash function [7]. In particular, the proposed algorithm can be improved by using specialized word-size packed string-matching instructions. We also plan to improve the work by combining uncertain reasoning models such as evidence theory [8].

Conflict of interest

None to report.

References

[1] Bhat N, Wijaya EB, Parikesit A. Use of the "DNAChecker" algorithm for improving bioinformatics research. Makara Journal of Technology. 2019; 23: 72–77.
[2] Tahir M, Sardaraz M, AzizIkram A. EPMA: efficient pattern matching algorithm for DNA sequences. Expert Systems with Applications. 2017; 80: 162–170.
[3] Najam M, Rasool RU, Ahmad HF, Ashraf U, Malik AW. Pattern Matching for DNA Sequencing Data Using Multiple Bloom Filters. BioMed Research International. 2019; Article ID 7074387.
[4] Saif A, Hu L. Multi pattern matching algorithm with wildcards based on Euclidean distance and hash function. Proceedings of 16th International Conference on Computational Science and Its Applications, Beijing, China. 2016.
[5] Faro S, Kulekci OM. Fast and flexible packed string matching. Journal of Discrete Algorithms. 2014; 61–72.
[6] Saif AAF, Hu L, Chu J. Multi pattern matching algorithm with wildcards based on bit-parallelism. Wuhan University Journal of Natural Sciences. 2017; 22: 178–184.

[7] Kulekci OM. Filter based fast matching of long patterns by using SIMD instructions. In Proceedings of the Prague Stringology Conference. 2009; pp. 118–128.
[8] Liu DY, Zhu YG, Ni N. Ordered proposition fusion based on consistency and uncertainty measurements. Science China Information Sciences. 2017; 60: 082103.