A Multi-pattern Matching Algorithm for Chinese-Hmong Mixed strings

Shun-Ping He, Li-Ping Mo* and Di-Wen Kang

College of Information Science & Engineering, Jishou University, Jishou 416000, Hunan, China
zmx89@jsu.edu.cn

Abstract. To solve the problem of rapid retrieval of Chinese-Hmong mixed text, a multi-pattern matching algorithm in double-bytes unit combined with the idea of AC algorithm and the mismatch processing strategy of Horspool algorithm is proposed for the Chinese-Hmong mixed strings. In this algorithm, a deterministic finite automaton is constructed based on the pattern-set according to the idea of AC algorithm, and the moving distance of the pattern is calculated by the bad-character rule of the Horspool algorithm, and the text is only traversed once to complete the quick search task of all patterns by using the finite automata. The experimental results show that the proposed algorithm has a good performance in multi-pattern matching for Chinese-Hmong mixed texts in different scale, even for the mixed texts containing more than 100,000 characters, the matching efficiency is also significantly higher than the AC algorithm.

Keywords: Natural Language Processing, Multi-pattern Matching, AC Algorithm, Horspool Algorithm.

1 Introduction

Square Hmong characters were created in the late Qing Dynasty of China and have been mainly used in the Hmong settlements such as Wuling Mountain District. Hmong informatization is one of great significances to promote the development of the local national cultural tourism industry and the digital protection of the intangible heritage of the Hmong culture. Information retrieval is a bottleneck to hinder the further research of the Hmong informatization. A high-performance pattern matching algorithm for strings is crucial to realize the fast retrieval of the square Hmong information.

According to the word formation, a square Hmong character represents a morpheme or word [1]. The square Hmong words in practical applications mainly containing single character or two characters, and few words containing 3 characters or more. Moreover, the square Hmong characters are usually mixed with Chinese characters, appear in the Chinese-Hmong songbook and script. The information retrieval mainly searches for a meaningful square Hmong string or Chinese-Hmong string from the mixed text. Obviously, compared to English, the mixed character set of the square Hmong characters and Chinese characters is a large character set, and the probability that the Chinese-

Hmong string is repeated in the text is very low, so mismatches are common. In previous research, Zeng L. et al. proposed a Horspool extension algorithm for matching the square Hmong string [2]. This algorithm directly expands the English character into words, and treats the double-bytes of square Hmong character as a whole. Only when the double bytes are completely equal, it is regarded as a square Hmong character matched. The experimental results show that the mentioned algorithm can solve the string search problem of square Hmong text. However, the algorithm is a single-pattern matching algorithm, which cannot search for multiple Chinese-Hmong mixed strings in text at one time. This paper presents an AC-EH algorithm, which combines the above Horspool extension algorithm with the AC algorithm, and can used to solve the multi-pattern matching problem of Chinese-Hmong mixed strings.

The rest of this paper is organized as follows. Section 2 introduces the basic principles of AC algorithm and Horspool algorithm. Section 3 depicts the proposed AC-EH algorithm. Section 4 verifies the feasibility of proposed algorithm by using the case study and experiment analysis. Section 5 concludes the paper.

2 Principles of AC Algorithm and Horspool Algorithm

2.1 AC Algorithm

Aho-Corasick automata algorithm (AC algorithm) is one of the most famous multi-pattern matching algorithms, and it utilizes the common prefix relationship among patterns to achieve efficient jumps in pattern mismatch. To some extent, AC algorithm can be regarded as an extension of the KMP algorithm in a multi-pattern context, but it has far better performance than the KMP algorithm [3-4]. Similar to the contribution of the KMP algorithm to the field of single pattern matching, AC algorithm has had a profound impact on the development of multi-pattern matching algorithms, it and its improved algorithms are still widely applied in various fields of pattern matching.

AC algorithm is based on the Deterministic Finite Automata (DFA) constructed according to the pattern-set, and converts the comparison of characters into the transition of the states of the DFA. The AC automata is represented as a six-tuple $M = (K, A, g, f, S, Z)$, where the following hold.

(1) K is a finite set, and each element of it is called a state.

(2) A is a finite alphabet, and each element of it is called an input symbol.

(3) g is a state transition function, which is an map on $K \times A \rightarrow K$, corresponding to the function *goto*.

(4) f is failure function, which is also an map on $K \times A \rightarrow K$, indicating a state transition when a pattern mismatch occurs, corresponding to the function *failure*.

(5) $S \in K$, is the initial state and unique.

(6) $Z \subset K$, is the final state set. The final state marked as double circles is also called an acceptable state or an ending state, corresponding to the function *output*.

The AC algorithm first preprocesses the pattern-set, establishes a function table corresponding to the functions *goto*, *failure* and *output*, and constructs a DFA used for pattern matching with a mismatch pointer accordingly. Then, with the DFA, the three functions described above are used to scan the text to be matched to find all occurrence positions of each pattern in the text. If the current state fails to match, it goes to the state indicated by the mismatch pointer of the current state, and the matching is continued.

AC algorithm eliminates the influence of the pattern-set size on the matching speed by preprocessing. For the text T to be matched with length n and the pattern-set $P=\{p_1, p_2, p_3,..., p_q\}$ with q patterns, AC algorithm only needs to scan T once to complete the searching for each pattern in P without backtracking, and can find all the patterns that have been successfully matched. Obviously, the time complexity of the AC algorithm is only related to the length n of T, is O(n).

2.2 Horspool Algorithm

The Horspool algorithm is derived from the BM algorithm. The BM algorithm with a time complexity of O(n) is one of the most famous single pattern matching algorithms, which calculates the maximum value of pattern shift distance using bad-character rule and good-suffix rule at the same time [5]. Due to skipping a lot of characters that don't need to match, the performance of the BM algorithm is 3-5 times better than that of the KMP algorithm in practical applications [6-7]. There are many improved versions of the BM algorithm [8-12], and the Horspool algorithm is one of them, which fixes the last character of the current matching-window as a bad-character, and only uses the bad-character rule to calculate the moving distance of the pattern [12]. Because the calculation of the distance based on good-suffix rule and the selection of the maximum value in BM algorithm are all avoided, its efficiency is significantly higher than that of the BM algorithm in practical applications.

For simplicity, the substring currently matched in T to be matched is denoted as $T[i]\ldots T[i+m\text{-}1]$. Where, i is the starting position of the substring, and $0\leq i\leq n\text{-}m$. $T[i]\ldots T[i+m\text{-}1]$ is also called a matching-window. Suppose $T[i+j]$ and $P[j]$ represent the currently processed character in T and P, respectively. Where, j is the matching position of the substring, and $0\leq j\leq m\text{-}1$. The algorithm takes the last character $T[i+m\text{-}1]$ in window as a bad-character, and compares it with the last character $P[m\text{-}1]$ of the pattern. If the matching successes, the algorithm continues to compare the remaining characters in T and P one by one from right to left until they are completely equal or there is a mismatch at a certain character position. If the matching fails, $T[i+m\text{-}1]$ acts as a bad-character, and the algorithm tries to find the last position k ($\text{-}1\leq k\leq m\text{-}2$) where the bad-character $T[i+m\text{-}1]$ appears in P, and then moves P to the position where $T[i+m\text{-}1]$ and $P[k]$ are aligned, and the next match will be continued.

To get the moving distance the pattern, the position of the last position k of each character in P and the distance of position k from $P[m\text{-}1]$ must be stored in advance. This distance is denoted as $shift[P[k]]$, which indicates the moving distance of P when $T[i+m\text{-}1]$ is a bad-character, and is calculated by the following Eq.(1).

$$shif[P[k]] = m\text{-}1\text{-}\ k \quad (\text{-}1 \leq k \leq m\text{-}2) \tag{1}$$

In Eq.(1), k=-1 means that the current character does not appear in P, and the moving distance P is m.

Horspool algorithm performs character comparison from right to left, shifts the pattern from left to right, and fixes the last character of the current matching-window as a bad-character, and only needs to scan the text *T* to be matched once to find all the substrings that match the pattern, so the time complexity is O(*n*). Obviously, the more common the character mismatch, the better the performance of the Horspool algorithm.

3 Proposed AC-EH Algorithm

3.1 Basic Principle of AC-EH Algorithm

The proposed AC-EH algorithm takes double-bytes as a matching-unit to execute matching for Chinese-Hmong mixed strings by combining the basic idea of AC algorithm with the mismatch processing strategy of Horspool algorithm. Similar to the AC algorithm, the AC-EH algorithm preprocesses the pattern-set before performing matching, that is, constructs three lookup tables corresponding to functions *goto*, *output*, and *shift* based on the pattern-set. The values of the three tables will be filled in during the calculation of the three functions. And then, the algorithm includes three stages. Firstly, the DFA is constructed and the corresponding functions *goto* and *output* are calculated according to the basic AC algorithm. Secondly, the function *shift* of the pattern is generated by using the bad-character heuristic rule of Horspool algorithm. Finally, the pattern matching is operated on the DFA. In the matching process, the functions *goto*, *output* and *shift* are used to scan the text to be matched and search for the patterns by moving DFA. When a mismatch occurs, the last character in the matching-window is fixed as a bad-character according to the mismatch processing strategy of the Horspool algorithm and the bad-character-based heuristic rule, and the pattern is moved based on the *shift* value of the current bad-character, and the matching is continued.

3.2 Construction of DFA and Calculation of Function *goto*

Construction of DFA and calculation of function *goto* are crossed. AC-EH algorithm creates an initial state 0 of the DFA, and starts from the initial state 0, for each pattern $p_i[1...Len]$ of the pattern-set *P* ($1 \leq i \leq q$), generates the value of the function *goto* and other states of DFA. Suppose that the length of the text *T* to be matched is *n*, the pattern-set with *q* patterns is $P=\{p_1, p_2, p_3,..., p_\mathrm{q}\}$, and the maximum and minimum lengths of each pattern in *P* are *MaxLen* and *MinLen*, respectively. Let's denote the current state as D_k, the steps for generating DFA and function *got*o are as follows.

Step1: Take the initial state 0 as D_k.

Step2: When D_k faces the input character $p_i[j]$ ($1 \leq j \leq Len$), it is checked whether there is such a state D_t in the direct successors of D_k (ie $goto(D_k, p_i[j])=D_t$). If it does, goto Step3, otherwise, goto Step4.

Step3: $D_k=D_t$, take the next character $p_i[j+1]$ as the current input character, and goto Step2.

Step4: Construct *goto*(D_k, $p_i[j]$)=D_t and check if the current input character is the last character $p_i[Len]$ of the pattern p_i. If yes, goto Step5; otherwise, goto Step3.

Step5: D_t is denoted as the final state.

3.3 Calculation of Function *Output* and Function *shift*

Assume that the current state D_k is a final state, and there is a path from the initial state 0 to the state D_k. If all characters on the path are sequentially connected to obtain the string t_1, then *output*(D_k)= t_1.

In the process of pattern matching with DFA, if the input character *a* is faced in the current state D_k, and a state D_t that makes *goto*(D_k, *a*)=D_t is not found, the *shift* value is calculated by the following Eq.(2).

$$shift(a)=\begin{cases} Min\{\, j \mid P_k[j]=a, 1< j \le MinLen, 1\le k\le q)\} & a \text{ is as a non-first character} \\ & \text{of string in the pattern} \\ MinLen & \text{others} \end{cases} \tag{2}$$

3.4 Description of AC-EH Algorithm

The AC-EH algorithm steps of multi-pattern matching using DFA for Chinese-Hmong mixed strings are as follows:

Step1: Preprocess pattern-set, generate functions *goto* and *output*, and construct DFA.

Step2: Traverse DFA, calculate the *shift* value of each character in *T* according to Eq.(2), and record it in the *shift* table.

Step3: Align the last character of the pattern p_{MinLen} with a length of *MinLen* in DFA with the last character of *T*, take the character *a* in the position where *T* is aligned with the first character of p_{MinLen} as the current character, and let the matching pointer point to *a*.

Step4: Take the initial state 0 as the current state D_k, record the position *Pos* of the character *a* in *T*, and move the matching pointer from *a*, then match characters one by one from left to right.

Step5: If match fails, move both pointer and DFA to the left based on the *shift* value at the position of character *a*, and then goto Step4. Otherwise, D_k=*goto*(D_k, *a*), that is, take the next state *goto*(D_k, *a*) as the current state.

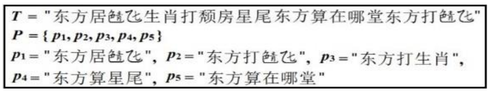
Step6: Check if state D_k is a final state. If yes, call the function *output*, store the value of *output*(D_k) and *Pos* value in the *output* table, then move the pointer and DFA to the left to *MinLen* characters, and then goto Step4. Otherwise, take the next character in *T* as the current character *a*, start the next comparison.

Step7: End the matching process. At this time, if the *output* table is empty, it means that *T* does not contain any pattern in *P*. Otherwise, the contents of the *output* table are just those patterns in *P* that appear in *T*.

4 Case Study and Related Experiment Analysis

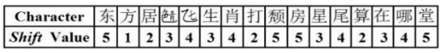
4.1 Case Study

Given the Chinese-Hmong mixed text *T* to be matched and the pattern-set *P*={p_1, p_2, p_3, p_4, p_5} as shown in Fig.1. The *MinLen* value of the pattern in *P* is 5. The *shift* value of each character in *T* is calculated according to Eq.(2) as shown in Fig.2.



T = "东方居遒飞生肖打颓房星尾东方算在哪堂东方打遒飞"
P = { p_1, p_2, p_3, p_4, p_5 }
p_1 = "东方居遒飞", p_2 = "东方打遒飞", p_3 = "东方打生肖",
p_4 = "东方算星尾", p_5 = "东方算在哪堂"

Fig. 1. The content of the text *T* and the pattern-set *P*



Character	东	方	居	遒	飞	生	肖	打	颓	房	星	尾	算	在	哪	堂
***Shift* Value**	5	1	2	3	4	3	4	2	5	5	3	4	2	3	4	5

Fig. 2. The *shift* value of each character in *T*

The following Fig.3 to Fig.7 show the execution of the AC-EH algorithm.

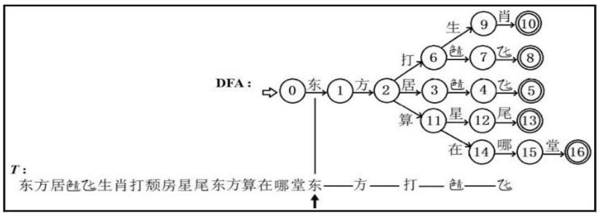


Fig. 3. The 1st diagram of algorithm execution case

In Fig.3, since the *MinLen* value is 5, the last 5 characters of *T* are aligned with the characters of the shortest pattern in the DFA. Starting from the initial state 0 in DFA and the current character '东' in *T*, the pointer is moved from left to right. The comparison in a character-by-character manner is done. When the final state 8 is reached, the pattern p_2 matches successfully. At this time, the *output* value of the final state 8 is p_2, and the position where p_2 appears in *T* is 19. Then, values p_2 and 19 are stored in the output table. And then, according to *MinLen* value, the pointer and the DFA are both moved 5 character-positions to the left as shown in Fig.4.

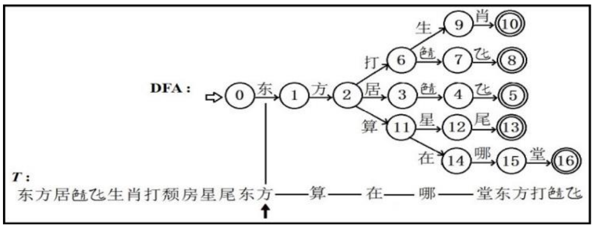


Fig. 4. The 2nd diagram of algorithm execution case

In Fig.4, matching is performed from the initial state 0. At this time, the current character ‘方’ in *T* does not match the character ‘东’ in the aligned position of DFA. Since the *shift* value of the character ‘方’ is 1, the pointer and the DFA are both moved 1 character-position to the left as shown in Fig.5.

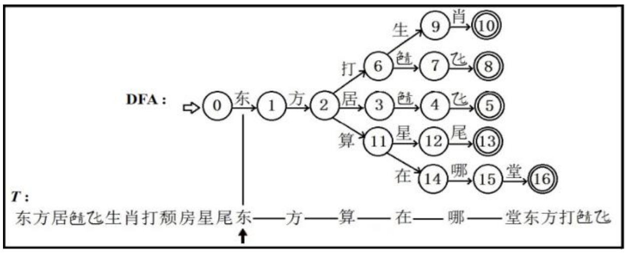


Fig. 5. The 3rd diagram of algorithm execution case

In Fig.5, matching is performed from the initial state 0. At this time, the current character ‘东’ in *T* matches the character ‘东’ of the aligned position in the DFA. Starting from the character ‘东”, the pointer is moved from left to right, and the comparison in a character-by-character manner is done again. When the final state 16 is reached, the pattern p_5 matches successfully. At this time, the *output* value of the final state 16 is p_5, and the position where p_5 appears in *T* is 13. Then values p_5 and 13 are stored in the *output* table. According to *MinLen* value, the pointer and the DFA are both moved 5 character-positions to the left as shown in Fig.6.

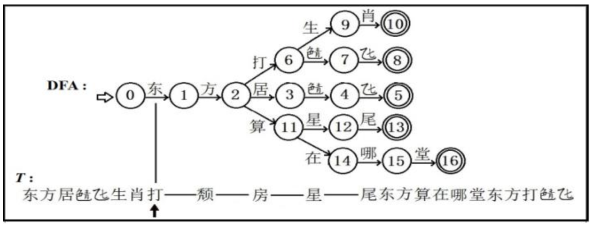


Fig. 6. The 4th diagram of algorithm execution case

In Fig.6, similar to the method described above, the pointer and the DFA are both moved to the left by 1, 2, and 3 character-positions in sequence until the current character ‘东’ in *T* matches the character ‘东’ of the aligned position in the DFA (as shown in Fig.7). Starting from the character ‘东’, the matching pointer is moved from left to right, the comparison in a character-by-character manner is done again. When the final state 5 is reached, the pattern p_1 matches successfully. At this time, the *output* value of the final state 5 is p_1, and the position where p_1 appears in *T* is 1. Then, values p_1 and 1are also stored in the output table. Finally, the matching is completed and the result is obtained by *output* table, that is, {(p_2,19), (p_5, 13) , (p_1,1)}.

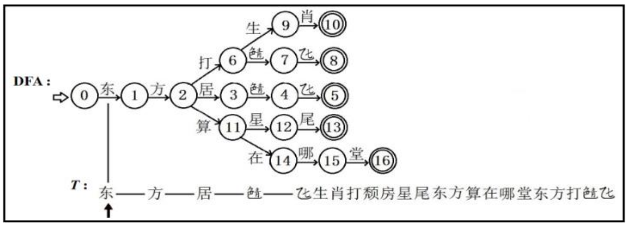


Fig. 7. The 5th diagram of algorithm execution case

4.2 Experiment Analysis

AC-EH algorithm was implemented in Java, all multi-pattern matching experiments were carried out under the conditions of Intel(R) Core(TM) i5-3470 CPU @ 3.20GHz, 4G memory and Win7 operating system. For the pattern-set composed of 5 patterns of length 1-10 and 3 mixed texts T_1, T_2 and T_3 to be matched with lengths of 5185, 37770 and 147645 words, both AC-EH algorithm and AC algorithm had an accuracy of 100%. Time-consuming data of two algorithms in experiments are shown in Table 1. The *TNo*, *TLen*, *PLen*$_1$-*PLen*$_5$, *AC-EH_Time*(ms), and *AC_Time*(ms) in the table respectively represent the number and length of the text to be matched, length of the five patterns, and time-consuming data of the AC-EH algorithm and the AC algorithm.

Table 1. Time-consuming comparison of two algorithms

TNo	*TLen*	*PLen*$_1$	*PLen*$_2$	*PLen*$_3$	*PLen*$_4$	*PLen*$_5$	*AC-EH_Time*	*AC_Time*
T_1	5185	1	1	1	1	1	6.770030	6.825193
		2	2	2	2	2	5.965028	6.885809
		3	3	3	3	3	5.200117	6.930709
		4	4	4	4	4	5.007366	6.795045
		5	5	5	5	5	5.067660	6.975610
		6	6	6	6	6	4.819424	6.507041
		7	7	7	7	7	4.885172	7.446423
		8	8	8	8	8	4.778373	7.305949
		9	9	9	9	9	5.080809	7.562844
		10	10	10	10	10	5.071509	7.551619
T_2	37770	1	1	1	1	1	13.082075	10.677013
		2	2	2	2	2	9.653281	10.356616
		3	3	3	3	3	8.607421	11.226723
		4	4	4	4	4	7.597803	10.621849
		5	5	5	5	5	8.734105	11.134998
		6	6	6	6	6	7.187924	10.582401
		7	7	7	7	7	7.051941	11.020822
		8	8	8	8	8	6.687605	10.877140
		9	9	9	9	9	7.169965	11.508313
		10	10	10	10	10	6.850209	11.037500

T_3	147645	1	1	1	1	1	31.151309	19.064743
		2	2	2	2	2	26.267099	16.522093
		3	3	3	3	3	21.268714	16.583350
		4	4	4	4	4	19.912399	17.339282
		5	5	5	5	5	17.442873	16.643965
		6	6	6	6	6	16.167058	16.615101
		7	7	7	7	7	16.362695	17.449288
		8	8	8	8	8	15.448009	17.432931
		9	9	9	9	9	15.641402	19.183729
		10	10	10	10	10	14.888356	17.025298

According to table 1, the time-consuming line chart of the two algorithms for matching different length patterns is shown in Fig.8.

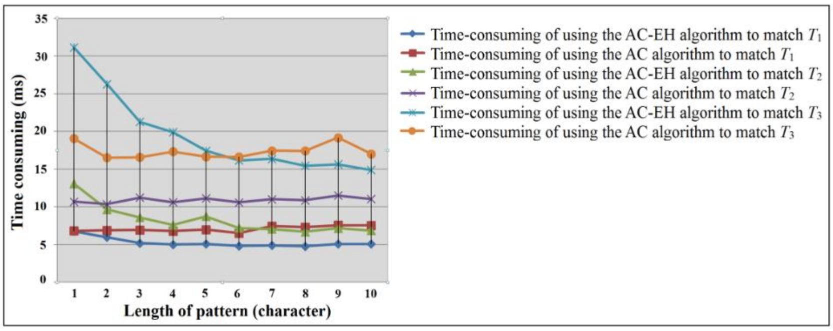


Fig. 8. Time-consuming of two algorithms for patterns with different length

From Table 1 and Fig.8, it is easy to find that: (1) When the text such as T_1 to be matched is very short, the time performance of the AC-EH algorithm is significantly better than that of the AC algorithm regardless of the pattern length. (2) When the text to be matched is longer but the pattern is relatively shorter (for case, T_2 matches the pattern with length 1, T_3 matches the pattern of length 1-5), the time performance of the AC-EH algorithm is slightly inferior to the AC algorithm. (3) When the length of the pattern is gradually increased (for case, T_2 matches the pattern with the length 2 to 10, T_3 matches the pattern with the length 6 to 10), the time performance of the AC algorithm remains basically stable, and that of the AC-EH algorithm is gradually improved, which is significantly better than that of the AC algorithm.

In summary, the matching speed of the AC-EH algorithm is significantly faster than that of the AC algorithm when the length of Chinese-Hmong mixed text and pattern is increased to a certain extent. Even for the Chinese-Hmong mixed text containing more than 100,000 words, the performance of the AC-EH algorithm can be better. Considering that the length of Chinese-Hmong mixed texts in the actual application are rarely more than 100,000 words, and the patterns that need to be searched (such as a lyrics) is usually longer, the AC-EH algorithm is suitable for solving the problem of multi-pattern matching for Chinese-Hmong mixed strings.

5 Conclusions and Future Work

This paper proposes a multi-pattern matching algorithm for Chinese-Hmong mixed strings by combining AC algorithm with Horspool extension algorithm. This algorithm is simple, easy to implement, has high matching efficiency, and suitable for realizing the rapid retrieval technology of Chinese-Hmong mixed text. In the future, we intend to study the parallel fuzzy search algorithm for Chinese-Hmong mixed strings with fuzzy Petri net.

Acknowledge

This work was supported by the National Natural Science Foundation of Hunan Province (No. 2019JJ40234), the Natural Science Foundation of China (No. 61462029), the Research Study and Innovative Experimental Project for College Students in Hunan Province (No. 20180599) and the Research Study and Innovative Experimental Project for College Students in Jishou University (No. JDCX20180122).

References

1. Yang, Z.B., Luo, H.Y.: On the folk coinage of characters of the Miao people in Xiangxi area. Journal of Jishou University (social sciences edition) 29(6), 130–134 (2008).
2. Zeng, L., Mo, L.P., Liu, B.Y., et al.: Extended Horspool algorithm and its application in square Hmong string pattern matching. Journal of Jishou University (Natrual Sciences Edition) 39(4), 150–156 (2018).
3. Aho, A.V., Corasick, M.J.: Efficient string matching: an aid to bibliographic search. Communications of ACM 18(6), 333-340 (1975).
4. Han, G.H., Zeng, C.: Theoretical research of KMP algorithm. Microelectronics ＆ computer 30(4), 30–33 (2013).
5. Boyer, R.S., Moore, J.S.: A fast string searching algorithm. Communication of ACM 20(10), 762–772 (1977).
6. Cole, R., Hariharan, R., Paterson, M., Zwick, U.: Tighter lower bounds on the exact complexity of String Matching. SIAM Journal of Computing 24(6), 30–45 (1995).
7. Cole, R., Hariharan, R.: Tighter upper bounds on the exact complexity of string matching. SIAM Journal of Computing 26(3), 803–856 (1997).
8. Zhao, X., He, L.F., Wang, X., et al.: An efficient pattern matching algorithm for string searching, Journal of Shanxi University of Science & Technology (Natural Science Edition) 35(1), 183–187 (2017).
9. Guibas, L. J., Odlyzko, A.M.: A new proof of the linearity of the Boyer-Moore string searching algorithm. Siam Journal of Computing 9(4), 672–682 (1980).
10. Sunday, D.M.: A very fast substring search algorithm. Communications of the ACM 33(8), 132–142 (1990).
11. Wang, W.X.: Research and improvement of the BM pattern matching algorithm. Journal of Shanxi Normal University (Natural Science Edition) 32(1), 37–39 (2017).
12. Horspool, R.N. Practical fast searching in strings. Software-Practice and Experience 10(6), 501–506 (1980).