Fast String Matching Algorithms Based Upon the Data Encoding Scheme

Tai-Ye Huang[#], Hui-Min Chen[*], Chien-Hung Huang[#], R. C. T. Lee[*], Hsien-Yang Liao[#]

[#] *Department of Computer Science and Information Engineering, National Formosa University, No.64, Wunhua Rd., Huwei Township, Yunlin County 632, Taiwan, R. O. C.*

[1]chhuang@nfu.edu.tw

[*] *Department of Computer Science, National Chi Nan University No.1, University Rd, Puli, Nantou County,54561 Taiwan, R.O.C.*

Abstract—**The string matching problem is to find all locations of a pattern string with length m in a text with length n. In this paper, we propose a new encoding method to shorten both the lengths of pattern and text, by substituting the substring between a special character by its length. Then we use the quick searching algorithm to solve the string matching problem on the encoded pattern and text [9]. By using the encoding method, the pattern and text can be shortened by a factor of, about, $\frac{2}{|\Sigma|}$, where $|\Sigma|$ denotes the size of the alphabet set containing all characters of text and pattern. In practice, it performs better than $\frac{2}{|\Sigma|}$. For instance, using an English sentence pattern, of length 50, as a pattern and a text, of length 200000, in average, they are shortened to about, respectively, 6% and 12.4% of the original sizes. Thus, the exact matching can be improved in a much shorter time.**

Keywords—**computational biology, algorithm, string matching, sliding window, data encoding scheme.**

1. Introduction

String matching is a fundamental operation, as such it has many practical applications in computational biology, data mining, intrusion detection, and data compression [5, 7].

The string matching problem is defined as follows. Given a text string $T = t_1t_2... t_n$ of size n, a pattern string $P = p_1p_2... p_m$ of size m, and the alphabet set Σ containing all characters of text T and pattern P, our purpose is to find all occurrences of P in T.

Efficient string matching algorithms have been proposed by many researchers [1-4, 6, 8-9]. Two well-known optimal, $O(m+n)$, time algorithms were proposed by Boyer et al. [1] and Knuth et al. [4]. Several string matching algorithms [2-3, 6, 8-9] were proposed to improve the hidden constant, by orders of magnitude.

In this paper, we first propose a new

encoding method to shorten both the lengths of pattern and text in $O(m+n)$. Then we solve the string matching problem by using the quick searching algorithm on the encoded pattern and text. To evaluate the effectiveness of the proposed algorithms, we implemented a prototype and compare it with two well-known algorithms [1, 4]. The results show that our algorithm is significantly better than the existing ones.

2. The Algorithms

2.1 The Quick Search Algorithm

The quick search algorithm was proposed by Sunday [9] and it is similar to the Horspool algorithm [2]. Let w be a character in the alphabet. The quick search algorithm consists of the following two phase. In the pre-processing phase, we construct a bad-character table which is used to shift pattern P. The bad-character table is used to find whether there exists a rightmost character w in P which is equal to the character t_{j+m} in a partial window $t_{j+1}...t_{j+m}$. If the character w exists in P, we record the position of w from the right end in the table. If w does not exist in P, we record it as $m+1$.

In the searching phase, compare text T and pattern P from left to right. Assume that p_i is aligned to t_j. If a match occurs in P, output the position i and look up the character t_{j+m} in the bad-character table to decide the number of steps to shift. If a mismatch occurs in P, we always look up the character t_{j+m} in the bad-character table to decide the number of steps to shift.

In Figure 1, we show an example of the quick search algorithm, where $T = atcacatcacaagtcat$ and $P = atcacat$. First, in the pre-processing phase, we compute the bad-character table as shown in Figure 1(a). Then, in the searching phase, we scan text T and pattern P from left to right. Because a match occurs, we output the position 1 and shift P 3 positions by looking up character t_8 of the bad-character table. This case is illustrated in Figure 1(b). We continue to scan the text T and the pattern P from left to right. When a mismatch occurs in p_2, we shift P 2 positions by looking up character t_{11} of the bad-character table. This case is also illustrated in Figure 1(c). By the same measure, a mismatch occurs in p_7, shift P 8 positions by looking up character t_{13}. If the window surpasses the length of text T, we stop the comparing. This case is illustrated in Figure 1(d). The pre-processing phase, it costs $O(m+s)$ time, where s is the number of alphabets in pattern P, the searching phase, takes $O(mn)$ time.

2.2 The Encoding Algorithms

In this section, we propose a new approach to solve the string matching problem. The main idea of our algorithm is to shorten both the lengths of the pattern P and the text T. Note that the both lengths of encoded pattern P and text T are much shorter than the original ones in general case. Then apply any string matching algorithms [1-4, 6, 8-9] to solve the string matching problem on the encoded pattern and text. Since the string lengths are reduced, it is obvious that the exact matching of pattern P will be much faster. Hence our approach has the following properties:

1. The substring P' is coded in such a way that it becomes much shorter than P. The coded P' is denoted as P'_{en}.
2. The text T will also be coded by using the same coding method to code P'. The coded text is denoted as T'_{en}. It will be shown that T'_{en} is also shortened and will contain P'_{en} if T contains P.
3. Since both T'_{en} and P'_{en} are short now, it becomes easy to determine whether P'_{en} occurs in T'_{en}.

Our encoding algorithm contains three phases: the encoding phase, the searching phase and the examination phase. For the sake of completeness, the detailed algorithm (DEA) is listed in the following.

Algorithm DEA

Input: $T = t_1 t_2 ..t_n, P = p_1 p_2 ... p_m$

Output: All occurrences of P in T.

Step 1: /The Encoding Phase/

Step 1.1: Choose a character x whose frequency in P is as small as possible and exclude the following cases.

Case 1: x only appears once in P.

Case 2: x appears twice and is adjacent in P.

If there is no such *x*, randomly select a character to be x.

Step 1.2: Let P'_{en} be the longest substring of P that begins with x and ends with x. If the length of the substring between two nearest x is greater than 0, replace the substring with its length.

Step 2: /The Searching Phase/

Use the quick search algorithm to find all occurrences of P'_{en} in T'_{en} from left to right. Record the position i's where P'_{en} occurs in T'_{en}.

Step 3: /The Examination Phase/

Let q denote the leftmost side position of x in P, r denote the leftmost side position of x in T, and $Pos(i)$ denote the corresponding position of T' while P'_{en} matches at position i of T'_{en}.

For every position i, align P with the substring $T_{r+Pos(i)-q,\, r+Pos(i)-q+m-1}$ to examine whether there exists an exact matching or not.

2.2.1 The Encoding Phase

The first step of the encoding phase is to select a character x from some alphabet set of P, and we delete all characters to the first x in P and all characters to the right of the last x in P. Then the result is a substring enclosed by x. This substring must be of the form of $x^{a_1} S_1 x^{a_2} S_2 \mathbf{L} S_k x^{a_{k+1}}$ for some k where x^i denotes a string of i $x's$, and the length of S_i could not be zero. We show an example of how to encode the pattern P. Let

$P = aacdstaaceedrcab$. Suppose we select character c, then we can get $P' = cdstaaceedrc$.

The second step is to encode P' into a shorter string P'_{en}. As mentioned above, the form of P' is $x^{a_1} S_1 x^{a_2} S_2 \ldots S_k x^{a_{k+1}}$, let d_i denote the size of S_i. Then $P'_{en} = x^{a_1} d_1 x^{a_2} d_2 \ldots x^{a_k} d_k x^{a_{k+1}}$. If the length of S_i is zero, we don't record it in P'_{en}. For example, we select a substring $P' = dcstddacceercd$, and we have $P'_{en} = d3dd7d$.

Finally, we use the same method to encode T. Suppose x is the character that we select to encode string P. Let T' be a substring of text T, and enclosed by x. If character x doesn't exist in T, T' shall be empty. If character x can be found in T, we encode T' as the form of $x^{a_1} d_1 x^{a_2} d_2 \ldots x^{a_k} d_k x^{a_{k+1}}$, which is T'_{en}.

We show an example, where $T = cactacgactactcgacgcta$ and $P = actacgact$. Suppose we select a character t in P. Then $P' = tacgact$ and $P'_{en} = t5t$. We also obtain $T = tacgactactcgacgct$ and $T'_{en} = t5t2t6t$ by using the same encoding method.

Now we describe the conditions that how we select an appropriate character to encode P and T into P'_{en} and T'_{en}. Let $fre(x)$ denote the frequency of character x in P. We select a character x with the minimal frequency. As long as $fre(x)$ conforms with the following two cases, the character x should be abandoned.

Case 1: $fre(x) = 1$.

Case 2: $fre(x) = 2$ and $|P_{l,r}| = 2$,

where l = the leftmost position of x in P and

r = the rightmost position of x in P.

We illustrate these two cases in the following:

Case 1: $fre(x) = 1$.

If we select the character whose $fre(x) = 1$, the length of string P'_{en} will be equal to one. It is like that we just take one character of pattern P to compare with text T. Of course, the exact matching probability will be larger than the probability of taking the entire P to compare with T. But in reality, string P and T may not have so many matches. Let's take an example as shown below, we are given a pattern $P = bcdcbcc$ and $T = dacbcdcbacdaac$. We have $fre(d) = 1$, then select it as our encoding character. In Figure 2(a), we can see that P'_{en} occurs three times in T'_{en}. But P does not exist in T at all, as shown in Figure 2(b).

Case 2: $fre(x) = 2$ and $|P_{l,r}| = 2$.

In Case 2, the length of string P'_{en} will become two. The exact matching probability will rise if we just take a substring with length two of P instead of the complete pattern P to compare with text T. The result is not satisfactory. We hope that the length of P'_{en} is moderate, not too short or too long. If the length is very short, it will rise the probability of getting the wrong matching; if the length is very long, it will rise the comparison times. Both of them are not we want to see. We show an example where $P = baaccbcbbb$ and $T = aacbaaccbcbbbedaaaad$. In Figure 3(a), P'_{en} occurs 5 times in T'_{en}. But in reality, P only occurs one time in T, as shown in Figure 3(b).

If we select the character c which doesn't obey Case 1 and Case 2 in the same P and T, where $P = baaccbcbbb$ and $T = aacbaaccbcbbbedaaaad$, the occurrence of P'_{en} in T'_{en} reduces to one, as shown in Figure 4(a). In Figure 4(b), we can see that the occurrence of P in T is really one time. The case of wrong matching is reduced now.

2.2.2 The Searching Phase

After the encoding phase, T and P have been encoded to T'_{en} and P'_{en}. In the searching phase, we need to find all occurrences of P'_{en} in T'_{en}. This is a string matching problem. Any string matching algorithms [1-4, 6, 8-9] can be used. But, since both text and pattern strings are reduced, we recommend to use the quick search algorithm [9]. Hence, the time complexity is $\mathrm{O}(|P'_{en}| \times |T'_{en}|)$, better than compare the entire pattern P with text T, whose time complexity is $\mathrm{O}(|P| \times |T|)$.

We show an example with $P = ccabbab$ and $T = ababcccabbabbccccabb$ in Figure 5. First, if we select "a" which has the smallest frequency in P, we have $P' = abba$, $P'_{en} = a2a$ and $T'_{en} = a1a4a2a6a$. The bad-character table can be obtained as shown in Figure 5(a). Then, we scan text T'_{en} and pattern P'_{en} from left to right. Figure 5(b) shows when a mismatch occurs in position 2, we shift P'_{en} 4 positions by looking up position 4 of T'_{en} of bad-character table. We continue to scan text T'_{en} and pattern P'_{en} from left to right. When a match occurs, we output the position 5 and move P'_{en} 4 steps to the right. The position is out of the window, terminate the matching phase. This case is shown in Figure 5(c).

Note that there are many algorithms which can be used to locate all occurrences of P'_{en} in T'_{en}. We also admit that the quick search algorithm is not the most efficient one in time complexity. For instance, the KMP algorithm only costs $\mathrm{O}(m+n)$ time in searching phase. In the procedure of encoding pattern P and text T into P'_{en} and T'_{en}, many numbers which exist in T'_{en} aren't in the alphabet of P'_{en}. In bad-character shift table, those numbers can shift more steps than English character. It is the reason that we choose the quick search algorithm in our searching phase.

2.2.3 The Examination Phase

After the searching phase, we find all occurrences of P'_{en} in T'_{en}. In examination phase, we need to know the corresponding position of T which should be examined. We define a function $Pos(i)$ to calculate the corresponding position of T' while P'_{en} matches at position i of T'_{en}. The function $Pos(i)$ is denoted as following:

$$Pos(i) = \sum_{x \le i} f(T'_{en}(x)), \quad \text{where } f(T'_{en}(x)) = \begin{cases} 1 & \text{, if } T'_{en}(x) \text{ is a character.} \\ T'_{en}(x) & \text{, if } T'_{en}(x) \text{ is a number.} \end{cases}$$

When P'_{en} matches at position i of T'_{en}, the function $Pos(i)$ accumulates the value from the first character of T'_{en} to the character of position i. If $T'_{en}(x)$ is a character, add 1 to the function; otherwise add $T'_{en}(x)$ to the function. Consider the example mentioned in Section 2.2.2, the text $T'_{en} = a1a4a2a6a$, suppose a match occurs at position 5 of T'_{en}. The function $Pos(5) = f(T'_{en}(1)) + f(T'_{en}(2)) + f(T'_{en}(3)) + f(T'_{en}(4)) + f(T'_{en}(5)) = 1+1+1+4+1=8$.

The function $Pos(i)$ is to calculate the position of T' that the first character of P' should be aligned, not to calculate the position of T which P should be aligned. Because we delete all characters until the encoding character x when P encoded into P'. We don't know how many characters are deleted by $Pos(i)$. If we want to know the position that P should be aligned to T, some shift value should be considered. Let q denote the leftmost side position of x in P, r denote the leftmost side position of x in T. Thus, we align P to match with $T_{r+Pos(i)-q,\, r+Pos(i)-q+m-1}$.

Given a complete example: Let $T = aaatbaaaatabataaatababa$ and $P = aaatababa$. After the searching phase, we have $T'_{en} = b6b7b1b$, $P'_{en} = b1b$ and $m = 9$. We also record that P'_{en} occurs at position 5 of T'_{en}. The leftmost side position of b in P is 6, that is $q = 6$. The most left side position of b in T is 5, that is $r = 5$. The function $Pos(5) = f(T'_{en}(1)) + f(T'_{en}(2)) + f(T'_{en}(3)) + f(T'_{en}(4)) + f(T'_{en}(5))$ $= 1+6+1+7+1=16$. As shown in Figure 6(a), P' occurs at position 16 of T'. We need to examine whether P occurs at position $(r + Pos(5) - q) = (5+16-6) = 15$ of T or not. That is, we need to align P to match with the substring $t_{15,23}$. We align P with t_{15} to verify whether there exists an exact string matching. Figure 6(b) shows that a match really occurs at t_{15}.

Figure 6(c) shows another example, where $T = atctcagtcagaaccctggtc$ and $P = agaccgacc$. We can see that $T'_{en} = g3g6gg$ and $P'_{en} = g3g$ and $m = 9$. Suppose from the searching phase, we know that P'_{en} occurs at position 1 of T'_{en}. $Pos(1) = 1$, $q = 2$ and $r = 7$. We align P to match with $t_{6,14}$. We align P with t_6 to verify whether there exists an exact string matching or not. And a mismatch occurs at t_6.

2.2.4 Analysis of Complexities

In this following, we analyze the time and space complexity of the proposed algorithms. In the encoding phase, the time complexity of encoding a pattern string P is $O(m)$ and encoding a text string T is $O(n)$. Assume that the character x is the best choice. We encode P and T into P'_{en} and T'_{en} respectively and this step can be done by linear scan. So we can see that the time complexity of encoding string P and string T is $O(m+n)$. The space complexity is the same as time complexity, $O(m+n)$.

In the searching phase, we construct a table: bad-character table. Let $|\Sigma|$ denote the alphabet size. The size of the bad-character table is $|\Sigma|$. It is obviously that the time complexity of generating the table is $\mathrm{O}(|P'_{en}|+|\Sigma|)$ and its space complexity is $\mathrm{O}(|\Sigma|)$. Then we scan the current window from left to right. When a mismatch occurs, we look up the bad-character table to decide our shift. The worst case of the time complexity in this phase is $\mathrm{O}(|P'_{en}|\times|T'_{en}|)$.

In the examination phase, the function $Pos(i)$ is used to calculate the corresponding match position in T. The function scans string T'_{en} and accumulates its value until position i which is obtained from the searching phase. It is obviously that the time complexity in this phase is $\mathrm{O}(|T'_{en}|)$.

In the following, we consider the compression ratio of our compression scheme. In general case, assume that the probability of each c in T and in P is equal for all $c \in \Sigma$. Let a denote the frequency of each c in T, b denote the frequency of each c in P. Because we assume that the frequency of each c is equal, a can also be represented as $\left\lfloor \frac{|T|}{|\Sigma|} \right\rfloor$, and b as $\left\lfloor \frac{|P|}{|\Sigma|} \right\rfloor$.

Suppose we select character x to encode P'_{en} and T'_{en}, where the number of x in T is a and the number of x in P is b. Then, the number of numerals in T'_{en} is at most $a-1$, and that in P'_{en} is at most $b-1$. Thus $|T'_{en}| \le a+(a-1)=2a-1$ and $|P'_{en}| \le b+(b-1)=2b-1$. By definition, $|T| \ge a|\Sigma|$ and $|P| \ge b|\Sigma|$. Therefore, we have $\frac{|T'_{en}|}{|T|} \le \frac{2a-1}{a|\Sigma|} \cong \frac{2a}{a|\Sigma|} = \frac{2}{|\Sigma|}$ and $\frac{|P'_{en}|}{|P|} \le \frac{2b-1}{b|\Sigma|} \cong \frac{2b}{b|\Sigma|} = \frac{2}{|\Sigma|}$.

This analysis shows that our compression scheme is quite effective. For the DNA type data, both text and pattern are compressed to half of their sizes. If the data are English language sentences, both text and pattern will be compressed to roughly 7.7% of their original sizes.

3. The Experimental Results

In this section, we conducted several experiments to demonstrate the performance of our algorithm. These experiments run on a Pentium IV 3GHz with 1GB of RAM under Windows XP. We make comparison with two well-known algorithms (i.e., KMP algorithm [4] and BM algorithm [1]) both in the comparison times and the compression ratio. All compared algorithms are implemented in the C++ programming language.

We classify the pattern into DNA type data and natural language data according to the size of pattern. Each experiment randomly selects a part of a DNA sequence and an English article with length 2×10^5 to be text T, and a part of it with length 10, 20, 30, 40 and 50 to be pattern P. Each result is the average of 100 experiments with different P's and T's. The *x*-axis represents the length of pattern. The *y*-axis represents the comparison times in 2×10^5 text length.

(1) DNA type data

We show the experiments with alphabet size is 4 (i.e., $|\Sigma| = 4$), and the patterns of sizes are 10, 20, 30, 40 and 50. Figure 7 shows the experimental result.

It is easy to see that our methodology has improved the comparison time to a large extent. For instance, when $|P| = 50$ and $|T| = 200000$ in Figure 7(a), the average comparison times of KMP algorithm, BM algorithm and our encoding method are 231181ms, 44875 ms and 9160 ms. In other words, the comparison time of our encoding method is only 3.96% and 20.4% that of KMP algorithm and BM algorithm respectively. Furthermore, with the same lengths of pattern and text, Figure 7(b) also shows that the pattern is shortened to 24% of its original length and the text is shortened to 38.1% of its original length. Thus, the exact matching can be done in a much shorter time.

(2) Natural language data

Figure 8 shows the experiments with alphabet is 26 (i.e.,$|\Sigma| = 26$). When $|P| = 50$ and $|T| = 200000$ in Figure 8(a), the average comparison times of KMP algorithm, BM algorithm and our encoding method are 201892 ms, 12769 ms and 5200 ms respectively. That is, the comparison times of our encoding method can be reduced to 2.57% and 40.7% that of KMP algorithm and BM algorithm respectively. With the same lengths of pattern and text, Figure 8(b) also shows that the pattern is shortened to 6% of its original length and the text is shortened to 12.4% of its original length. Hence, our encoding method is much superior to these two well-known algorithms.

4. Concluding Remarks

In this paper, we have proposed an encoding approach to compress pattern and text. Then the quick search algorithm is used to locate all occurrences of pattern in text. Since the input is shorter than the original one, it is obvious that the matching will be much faster. An examination method is also introduced to calculate the corresponding position of the original pattern and text while an exact match occurs. Furthermore, based on the experimental results, the proposed encoding algorithm is quite efficient for different type data, especially in natural language data.

5. Acknowledgement

This work is supported by the National Science Council of R.O.C. under the grant of NSC 97-2221-E-150-063.

References

[1] Boyer R. S. and Moore J. S., A Fast String Search Algorithm. *Communication of the ACM*, 1977; 20: 762–772.

[2] Horspool R. N., Practical Fast Searching

in Strings. *Software-Practice & Experience*, 1980; 10: 501-506.

[3] Hume A. and Sunday D. M., Fast String Searching. *Software-Practice & Experience*, 1991; 21: 1221-1248.

[4] Knuth D. E., Morris J. H. and V. R. Pratt V. R., Fast Pattern Matching in Strings. *SIAM Journal on Computing*, 1977; 6(2): 323-350.

[5] Pevzner P. P. *Computational Molecular Biology-An Algorithmic Approach*. MIT Press, 2000.

[6] Raita T., Tuning the Boyer-Moore-Horspool String Searching Algorithm. *Software - Practice & Experience*, 1992; 22(10): 879-884.

[7] Sayoood K. *Introduction to Data Compression*. second edition, Morgan Kaufmann, 2000.

[8] Smith P. D., Experiments With a Very Fast Substring Search Algorithm. *Software - Practice & Experience*, 1991; 21(10): 1065-1074.

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

w	*c*	*a*	*t*	*
shift	3	2	1	8

(a) After the pre-processing phase, the bad-character table.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T=	*a*	*t*	*c*	*a*	*c*	*a*	*t*	*c*	*a*	*c*	*a*	*a*	*g*	*t*	*c*	*a*
	⇕	⇕	⇕	⇕	⇕	⇕	⇕									
P=	*a*	*t*	*c*	*a*	*c*	*a*	*t*									
	⟶	⟶	*P*=	*a*	*t*	*c*	*a*	*c*	*a*	*t*						

(b) During the searching phase, an exact match occurred.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T=	*a*	*t*	*c*	*a*	*c*	*a*	*t*	*c*	*a*	*c*	*a*	*a*	*g*	*t*	*c*	*a*
				⇕	✗											
			P=	*a*	*t*	*c*	*a*	*c*	*a*	*t*						
			⟶	⟶	*P*=	*a*	*t*	*c*	*a*	*c*	*a*	*t*				

(c) During the searching phase, a mismatch occurred.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
T=	*a*	*t*	*c*	*a*	*c*	*a*	*t*	*c*	*a*	*c*	*a*	*a*	*g*	*t*	*c*	*a*
						⇕	⇕	⇕	⇕	⇕	⇕	✗				
					P=	*a*	*t*	*c*	*a*	*c*	*a*	*t*				
											⟶	⟶	*P*=	*a*	*t*	*c*

(d) The completion of the searching phase.

Figure 1: An illustration of the quick search algorithm for $T = atcacatcacaagtcat$ and $P = atcacat$.

T'_{en} =	d	4	d	4	d
	⇕		↑		↑
P'_{en} =	d		↓		│
			d		↓
					d

(a) The occurrence of T'_{en} and P'_{en}.

T =	d	a	c	b	c	d	c	b	a	c	d	a	a	c
						✕	⇕							
P =	b	c	d	c	b	c	c							
									✕	⇕				
		P =		b	c	d	c	b	c	c				
													✕	⇕
						P =		b	c	d	c	b	c	c

(b) The occurrence of T and P.

Figure 2: An illustration of Case 1 for $fre(x) = 1$.

	1	2	3	4	5	6	7	8	9	10
$T'_{en}=$	a	a	2	a	a	9	a	a	a	a
$P'_{en}=$	a	a								
	→	→	→	a	a					
	→	→	→	→	→	→	a	a		
	→	→	→	→	→	→	→	a	a	
	→	→	→	→	→	→	→	→	a	a

(a) The occurrence of T'_{en} and P'_{en}.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
T=	a	a	c	b	a	a	c	c	b	c	b	b	b	e	d	a	a	a	a	d
										╳										
P=	b	a	a	c	c	b	c	b	b	b										
			P=	b	a	a	c	c	b	c	b	b	b							
																				╳
														P=	b	a	a	c	c	b

(b) The occurrence of T and P.

Figure 3: An illustration of Case 2 for $fre(x)=2$ and $|P_{l,r}|=2$.

$T'_{en}=$	*c*	3	*c*	*c*	1	*c*
			⇕	⇕	⇕	⇕
$P'_{en}=$			*c*	*c*	1	*c*

(a) The occurrence of T'_{en} and P'_{en}.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
T=	*a*	*a*	*c*	*b*	*a*	*a*	*c*	*c*	*b*	*c*	*b*	*b*	*b*	*e*	*d*	*a*	*a*	*a*	*a*	*d*
				⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕							
			P=	*b*	*a*	*a*	*c*	*c*	*b*	*c*	*b*	*b*	*b*							

(b) The occurrence of T and P.

Figure 4: An illustration out of Case 1 and Case 2.

P'_{en} =	*a*	2	*a*

w	*a*	2	*
shift	1	2	4

(a) The bad-character table.

	1	2	3	4	5	6	7	8	9
T'_{en} =	*a*	1	*a*	4	*a*	2	*a*	6	*a*
	⇕	✗							
P'_{en} =	*a*	2	*a*						

(b) A mismatch occurred.

	1	2	3	4	5	6	7	8	9
T'_{en} =	*a*	1	*a*	4	*a*	2	*a*	6	*a*
					⇕	⇕	⇕		
	→	→	P'_{en}	=	*a*	2	*a*		

(c) The matching phase terminated.

Figure 5: An illustration of the searching phase for $P = ccabbab$, $x = 'a'$ and $T = ababcccabbabbccccabb$.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
T' =	*b*	*a*	*a*	*a*	*a*	*t*	*a*	*b*	*a*	*t*	*a*	*a*	*a*	*t*	*a*	*b*	*a*	*b*
															P' =	*b*	*a*	*b*

(a) An occurrence of P' in T'.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
T:	*a*	*a*	*a*	*t*	*b*	*a*	*a*	*a*	*a*	*t*	*a*	*b*	*a*	*t*	*a*	*a*	*a*	*t*	*a*	*b*	*a*	*b*	*a*
															⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕	⇕
P:															*a*	*a*	*a*	*t*	*a*	*b*	*a*	*b*	*a*

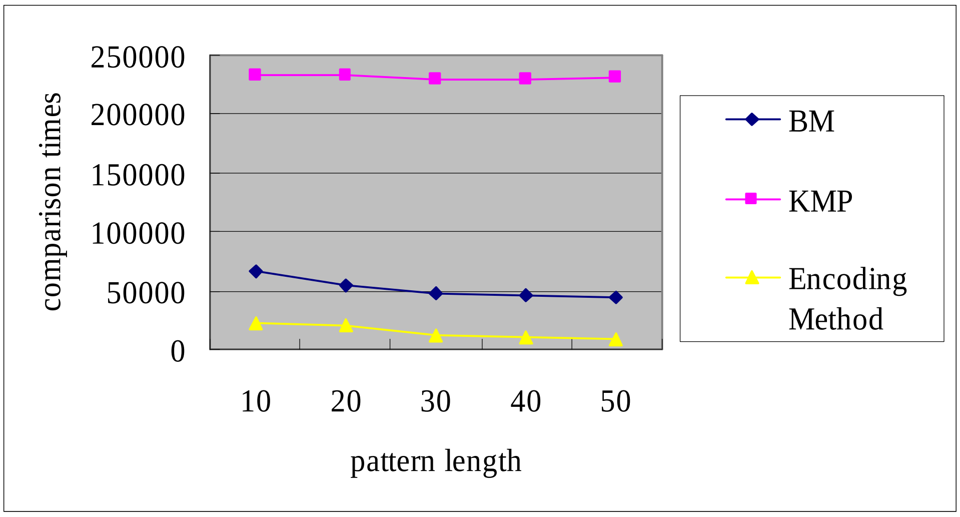
(b) An examination of P in T.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
T:	*a*	*t*	*c*	*t*	*c*	*a*	*g*	*t*	*c*	*a*	*g*	*a*	*a*	*c*	*c*	*c*	*t*	*g*	*g*	*t*	*c*
													✕	⇕							
P:						*a*	*g*	*a*	*c*	*c*	*g*	*a*	*c*	*c*							

(c) An examination of P in T.

Figure 6: An illustration of the examination phase.

Length Text=200000	**KMP**	**BM**	**Encoding Method**
Pattern = 10	232979	65986	22825
20	233615	55081	20110
30	229804	47987	12243
40	228902	45268	10885
50	231181	44875	9160



(a) The comparison times.

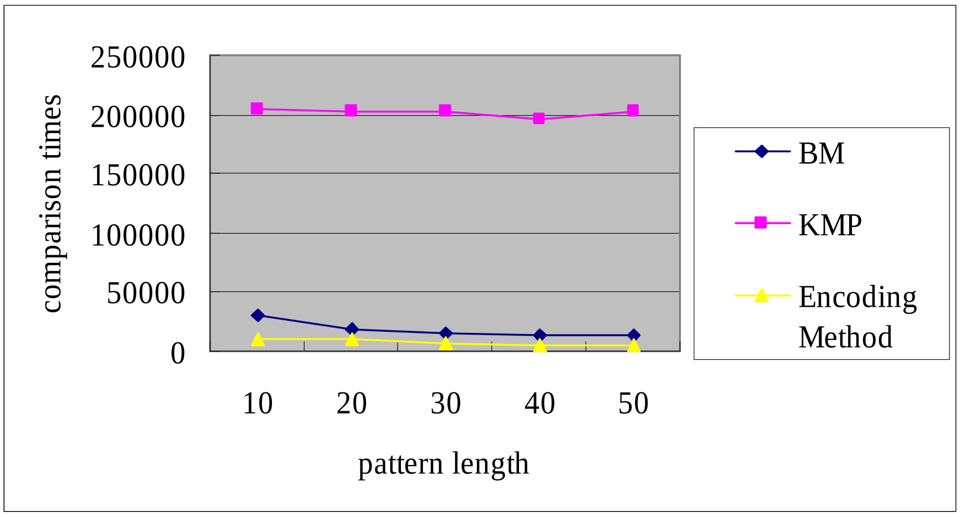
$\|P\|$	$\|T\|$	$\|P'_{en}\|$	$\|T'_{en}\|$	$\Re$ **of** P	$\Re$ **of** T
10	200000	4	82362	40%	41.1%
20	200000	4	82954	20%	41.4%
30	200000	7	82588	23.2%	41.2%
40	200000	11	77322	27.5%	38.6%
50	200000	12	76321	24%	38.1%

$\Re$: compression ratio

(b) The compression ratio.

Figure 7: The experiment of DNA type data for comparison times and the compression ratio.

Length Text=200000	**KMP**	**BM**	**Encoding Method**
Pattern = 10	204315	29583	10831
20	202376	18354	10126
30	202989	15356	7105
40	196612	13334	4712
50	201892	12769	5200



(a) The comparison times.

P	T	P'_{en}	T'_{en}	**ℜ of** P	**ℜ of** T
10	200000	4	46551	40%	23.2%
20	200000	3	35042	15%	17.5%
30	200000	3	31203	10%	15.6%
40	200000	3	27329	7.5%	13.6%
50	200000	3	24800	6%	12.4%

ℜ : compression ratio

(b) The compression ratio.

Figure 8: The experiment of natural language type for comparison times and the comparison ratio.