CHINESE STRING SEARCHING USING THE KMP ALGORITHM

Robert W.P. Luk
Department of Computing, Hong Kong Polytechnic University, Kowloon, Hong Kong
E-mail: csrluk@comp.polyu.edu.hk

Abstract

This paper is about the modification of KMP (Knuth, Morris and Pratt) algorithm for string searching of Chinese text. The difficulty is searching through a text string of single- and multi-byte characters. We showed that proper decoding of the input as sequences of characters instead of bytes is necessary. The standard KMP algorithm can easily be modified for Chinese string searching but at the worst-case time-complexity of *O(3n)* in terms of the number of comparisons. The finite-automaton implementation can achieve worst-case time complexity of *O(2n)* but constructing the transition table depends on the size of the alphabet, Σ, which is large for Chinese (for Big-5, $\Sigma > 13{,}000$). A mapping technique reduces the size the alphabet to at most $|P|$ where P is the pattern string.

1. Introduction

The alphabet size of Chinese (to be more precise Hanyu) is relatively large (e.g. about 55,000 in Hanyu Da Cidian) compared with Indo-European languages. Various internal codes (e.g. GB, Big5, and Unicode) have been designed to represent a selected subset (5000-16,000) which requires two or more bytes to represent. For compatability with existing single-byte text, the most significant bit of the first byte is used to distinguish between multi-byte characters and single-byte characters. For instance, Web browsers (e.g. Netscape) cannot interpret the annotations represented by their equivalent 2-byte characters. Thus, Chinese string searching algorithms have to deal with a mixture of single- and multi-byte characters.

This paper will focus in 2-byte characters because their internal codes are widely used. Two modified versions of the KMP algorithms are presented: the classical one and the finite-automaton implemenation. Finally, we discuss the practical situations in Chinese string searching.

2. The Problem

Directly using existing fast string searching algorithms (Knuth *et al.*,1977; Boyer and Moore,1977) for on-line Chinese text can lead to errors in identification as in using the find option of Netscape in Chinese window. For example, the pattern string, $P=$"" (i.e. AA,AA in hexidecimal) can successfully match with the second and third bytes of the text string, $T=$¥""B (i.e. A4,AA,AA,43 in hexidecimal) which is incorrect. The error occurs where the second byte of the character in T is interpreted as the first-byte of the pattern character. Thus, it is necessary to decode the input data as characters.

Two well-known string searching algorithms were discovered by Knuth, Morris and Pratt (1977) (KMP), and Boyer and Moore (1977) (BM). The KMP algorithm has better worst-case time complexity where as the BM algorithm has better average-case time complexity. Recently, there has been some interest in improving (Hume and Sunday, 1991; *Crochemore et al.*, 1994) the time complexity or proving a smaller bound (Cole, 1994) of the time-complexity of the BM algorithm, as well as in the efficient construction (Baeza-Yates *et al.*, 1994) of the BM algorithm. These algorithms derived from BM assumes that knowing the positional index, i, of the text string, T, can access and interpret the data, $T[i]$, as a character. However, with a text string of single- and multi-byte characters, i can point to the first-byte or the second-byte of a 2-byte character which the computer cannot determine in the middle of the text string. It has to scan left or right until a one-byte character, the beginning of the text string or the end of the text string is encountered. For example, the BM algorithm moves to position $i = 4$ ($= ||P||$) for matching in Table 1. At this position, $T[4]$ (= A4) does not match with $P[4]$. Since the computer cannot determine whether $T[4]$ is the first or second byte of the 2-byte character, it cannot use the delta tables to determine the next matching states. Even worst, for some internal code (e.g. Big-5), it is not possible to directly convert the byte sequence into the corresponding character sequence in the backward direction. Thus, as a first step, we focus on modifying the KMP for Chinese string searching.

i	1	2	3	4	5	6	7	8
T	¤£		¤¤		¤§		¤ß	
T[i]	A4	A3	A4	A4	A4	A7	A4	DF
P	<	¤¤		>				
P[i]	3C	A4	A4	3E				

Table 1: Matching between the text string, $T=$¤£¤¤¤§¤ß , and the pattern string, $P=$<¤¤>. Here, *T[]* and *P[]* shows the hexidecimal value of each byte in T and P.

3. Knuth-Morris-Pratt Algorithm.

3.1 Searching

Figure 1 is the listing of the modified version of KMP algorithm (Knuth *et al.*, 1977) for searching

Chinese string. Here, i is the positional index of the text string but the position is specified in terms of bytes. By comparison, j is the positional index of the pattern string, P, and the position is in terms of characters. Characters in P are stored in two arrays $P1[]$ and $P2[]$. Here, $P1[]$ stores the first-byte and $P2[]$ stores the second byte of two-byte characters in P. If there are single-byte characters in P, they are stored in $P1[]$ and the data in corresponding positions of $P2[]$ are undefined. Here, we assumed that a NULL character is patched at those positions. For example, if P=<¤£<¤£¥i>, then the values in $P1[]$ and $P2[]$ are shown in Table 2.

```
1  function Chinese_KMP
2
3  {  int i=1; j=1;
4
5     while ((j <= |P|) && (i <= ||T||)) {
6        if one-byte-character(T[i])
               /* decode single- or 2-byte characters */
7           {  while ((j!=0) && (T[i]!=P1[j]))
               /* 1-byte character matching */
8              j = next[j];   /* failure link */
9              i++;           /* update i position */
10             }
11       else { while    ((j!=0)    &&    ((T[i]!=P1[j])    ||
(T[i+1]!=P2[j])))             /* matching */
12             j = next[j];   /* failure link */
13             i+= 2;         /* update i position */
14             }
15       j += 1;              /* update j position */
16       }                    /* while-loop ends */
17    if (j > |P|) then  return(i-||P||);
                            /* compute matched position */
18       else  return(0);     /* no matched position */
19    }
```

Figure 1: A modified version of KMP for Chinese string searching. The function, one-byte-character, determine whether the current input is a single or 2-byte character, by testing whether the converted integer value of $T[i]$ is positive or negative. If the converted value is negative, then $T[i]$ is the first-byte of a 2-byte character. Here, $|T|$ and $||T||$ are the length of the text string, T, in terms of characters and bytes, respectively.

The program in Figure 1 determines (in line 6) whether the current input character is a single- or two-byte character. If it is a single-byte character, the standard KMP algorithm operates for that single-byte character, $T[i]$, in line 7 to 10. Otherwise, i is pointing at a two-byte character. This implies that: (a) matching 2-byte characters is carried out where the data in $T[i+1]$ is the second byte of the character (line 11); and (b) i is incremented by 2 instead of 1, because it is counting in terms of bytes (line 12). Since j is counting in terms of characters, the increment of j (lne 15) is one whether the characters in P are single or two bytes. When the pattern string is found in T, the position of the first matched character in T is returned. Since the position is in terms of bytes, it is the last matched position, i, minus the length of P in terms of bytes (i.e. $||P||$).

Character $P[j]$	<	¤£	<	¤£	¥i	>
j	1	2	3	4	5	6
$P1[j]$	3C	A4	3C	A4	A5	3E
$P2[j]$	NULL	A3	NULL	A3	69	NULL
$f(P[j])$	<	a	<	a	b	>
$next[j]$	0	1	0	1	3	0

Table 2: The values of the patterns indexed by j. Here, $P[]$ is a conceptual array which can hold both single- and 2-byte characters. This array is implemented as two arrays: $P1[]$ and $P2[]$ which stores the first and second byte of the 2-byte characters, respectively. The function, $f()$, maps two byte characters into single-byte characters, simplifying the generation of values in the array, $next[]$, and the failure links in $fl[]$.

3.2 Generating next[]

The array, $next[]$, contains the failure link values which can be generated by existing algorithms (Standish, 1980) for single-byte characters. The basic idea is to map the 2-byte characters in P to single-byte characters and then use existing algorithms. The mapping is implemented as an array, $f[]$. Each character in P is scanned from left-ro-right. Whenever an unseen character is found, it is assigned a character value that is the negative of the amount of different 2-byte characters seen so far. For example, the third unseen 2-byte character is mapped to a one-byte character, the value of which is (char) -3.

The mapping scheme is practical. First, the number of different characters that can be represened with a negative value is 127 and usually $|P| < 128$ characters. Second, the time-complexity of mapping, $O(||P||)$, can be done in linear time with respect to $|P|$ and in constant time with respect to $|T|$. This is important because it is added to the total time-complexity of searching. To achieve $O(||P||)$, the function, $found()$, uses an array, $f[]$, of size $|\Sigma|$ (where Σ is the alphabet) to store the equivalent single-byte characters. A perfect hash function (section 4), $h()$, converts the 2-byte characters into an index of $f[]$. After searching, it is necessary to clear $f[]$. This can be done in $O(||P||)$ by assigning NULL characters to the locations in $f[]$ corresponding to 2-byte characters in P.

4. Finite automaton implementation.

Since $||T||$ is large, reducing its multiplicative factor in the time complexity would be attractive. In Knuth *et al.*, (1977), this was done using a finite automaton which searches in $O(||T||)$ instead of $O(2||T||)$. Standish (1980) provided an accessible algorithm to build the automaton, M. First, failure link values are computed (similar to computing values in $next[]$) as in Algorithm 7.4 (Standish, 1980) and then the state transitions are added as in Algorithm 7.5 (Standish 1980). A direct approach is to compute the conceptual automaton, Mc, which regards the 2-byte characters as

one-byte and then convert the automaton for multi-byte processing. Since the space-time complexity in constructing the automaton depends on the size of the alphabet (i.e. $O(|\Sigma| \times |Qc|)$ where Qc is the set of states of Mc) which is large, this approach is not attractive. For instance, if $|Qc| = 10$ and $|\Sigma| = 10{,}000$, then about 100,000 units of storage (integers) are needed! Further processing is needed to convert the automaton for 2-byte processing!

4.1 Automaton Implementation.

Another approach uses the different characters in P as the reduced alphabet, Σr, which is much smaller than $|\Sigma|$. We use a mapping function as discussed in section 3.2 to build a mapping of 2-byte characters to one-byte. These one-byte characters and the standard one-byte characters (e.g. ASCII) form Σr. The NULL character, λ, represents all the characters in Σ but not in $\Sigma r - \{\lambda\}$ ($= \Sigma * (\Sigma r - \{\lambda\})'$). Given that the multi-byte string, P, is transformed into a single-byte string, P', existing algorithms can be used to construct the automaton.

For each pattern string, P, string searching will execute the following steps:

(a) convert 2-byte characters to one-byte in P to form P' (i.e. form Σr) using mapping as in section 3.2;
(b) compute the failure link values of P' using Algorithm 7.4 in (Standish, 1980);
(c) compute the success transitions and store them in $\delta()$ as in (Standish, 1980);
(d) compute the failure transitions using the failure link values using Algorithm 7.5 in (Standish, 1980) and store the transitions in $\delta()$;
(e) use the automaton, M, with state transition function $\delta()$, to search for P' in T;
(f) output the matched position, if any;
(g) clear that mapping function that forms Σr using P.

4.2 Constructing the automaton.

For step (c) and (d), the operation of Algorithm 7.5 was illustrated with an example of a binary alphabet in (Standish, 1980). Here, we illustrate the use of a larger alphabet, Σr, and $\lambda \in \Sigma r$. Suppose the pattern string, P, is as shown in Table 2 which also contains the corresponding P' and failure link values, $fl[j]$. The success transitions are added to $\delta()$ as $\delta(j\text{-}1, P'[j] \leftarrow j$ (e.g. $\delta(0,<) \leftarrow 1$ and $\delta(1,a) \leftarrow 2$). The failure transitions are computed from 0 to $|P'|$ because $fl[j] < j$. For state 0, $\delta(0,\alpha) \leftarrow 0$ if $\alpha \neq P'[1]$ and $\alpha \in \Sigma r$ (i.e. $\delta(0,a) \leftarrow 0$, $\delta(0,b) \leftarrow 0$, $\delta(0,>) \leftarrow 0$, $\delta(0,\lambda) \leftarrow 0$ but $\delta(0,\alpha) \leftarrow 1$). For other states, $\delta(j,\alpha) \leftarrow \delta(fl[j],\alpha)$ if $\alpha \neq P'[j]$ and $\alpha \in \Sigma r$ (e.g. $\delta(1,a) \leftarrow \delta(fl[1],a) = \delta(0,a) = 0$ and $\delta(1,<) \leftarrow \delta(fl[1],<) = \delta(0,<) = 1$). Effectively, the states in $\delta(fl[j],.)$ are copied across to the corresponding entries in $\delta(j,.)$ except for the successful transition from j. Figure 2 illustrates how a row of entries in $\delta(fl[j],.)$ are copied across to compute $\delta(j,.)$.

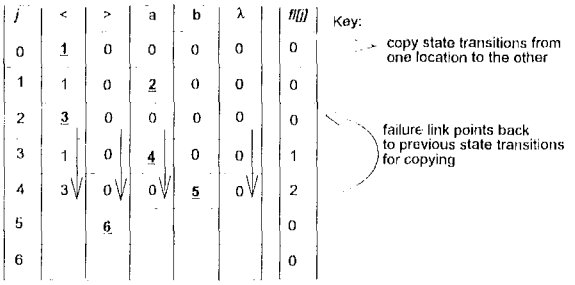

Figure 2: An illustration of constructing the failure transitions of M. Here, $j = 4$ and the failure link of j (i.e. $fl[4] = 2$) is used to determine which of the previous row of the state transition table, $\delta()$, is used for updating the values of the current row in $\delta()$. The underlined entries are the success transistions.

Figure 3 shows the program that computes the state transitions using the failure links. The program computes for state 0, the last states and the other states separately. The last state is distinguished because it has no success transitions where as the other has one for each state. The program for generating failure links is not given because:

(1) it is similar to computing *next[]*;
(2) a version is available (Algorithm 7.4 in Standish, 1980) which does not need any modification.

```
1  void build_transitions()
2
3  {
4     int i=0, j=0, k=0;
5
6     for (i=-|Σ2|;i<=|Σ1|;i++) /* build transistions at j = 0 */
7        if ((char) i == P[1])  δ(0,i)=1;
8           else  δ(0,i)=0;
9     for(j=1;j < |P|; j++) {   /* build other transitions which has
success tranistions */
10       k = fl[j];
11       for (i=-|Σ2|;i<=|Σ1|;i++)
12       if ((char) i == P[j+1])  δ(j,i)=j+1;
13          else  δ(j,i)= δ(k,i); }
14       k = fl[|P|];/* failure transitions for j = |P| */
15       for (i=-|Σ2|;i<=|Σ1|;i++) /*  there  is  no  success
transition in this case */
16          δ(j,i)= δ(k,i);
17}
```

Figure 3: Building the state transitions given that the failure links are known. Note that the algorithm assumed that $\Sigma r = \Sigma 1 \cup \Sigma 2$ where $\Sigma 1$ and $\Sigma 2$ are the one-byte (e.g. ASCII) character alphabet and the transformed 1-byte character alphabet representing the different two-byte characters in P, respectively. Futhermore, since $|\Sigma 2| < 128$ and $\Sigma 2 \subseteq \Sigma$. A multiplicative factor of the space-time complexity can be reduced if mapping is also carried out for single-byte as well as 2-byte characters in P. The correctness of the above program can be shown by mapping all the characters not in Σr to Σ because they have identical state transition values (i.e. dividing the alphabet into equivalent classes of identical transition values).

4.3 Searching.

Searching is implemented as state transitions of M (Figure 4). Initially, the state of the automaton, M, is set to 0. The next state is determined by the current character read from the text string, T, at position i and the current state. If the current state is equal to $|P'|$, then P is in T at position $i - ||P||$.

```
1  intFA_KMP()
2  {
3     int i=1; state=0;
4     while ((state != |P|) && (i <= ||T||)) {
5        if one-byte-character(T[i])        /* decoding front-end */
6           input_character = (int) T[i];
7        else  { input_character = found(T[i],T[i+1]);
8                i++};        /* update for 2-byte character */
9        state = δ(state, input_character);
10          i++;
11       }
12       if (state == |P|) return (i - ||P||);
13          else  return(0);
14    }
```

Figure 4: String searching of multi-byte characters using the finite automaton.

5. Practical considerations.

The KMP algorithm (Knuth *et al.*, 1977) was considered to perform better when the pattern string has recurrence patterns. Otherwise, it is about the same as the brute-force implementation with quadratic time-complexity. For Chinese string searching, it is not uncommon to search for reduplicating words (e.g. ¦³¨S¦³ and §Ö§Ö¼Ö¼Ö) (Chen *et al.*, 1992) which has recurrence patterns. Such repetition to form words is used in making emphasis as well as an essential part of yes-no questions. Otherwise, recurrence patterns in P occur only incidentally (e.g. ¤¤¤j¤¤¤å¾Ç¾Ç¨t translated as the Department of Chinese, Chinese University of Hong Kong).

Apart from recurrence, if there are a lot of backing up operations, the KMP algorithm would perform better than the brute-force implementation. Such cases occur where a proper prefix of the pattern string has high occurrence frequency in the text string (e.g. function words). In Chinese string searching, this will happen for technical terms that have a high frequency prefix constituent. For instance, Chinese law articles have many terms beginning with the word ¤¤°ê (i.e. China). A search through the Chinese law text for P=¤¤°ê¤H will require many backing up (or committing a false start) in the brute-force implementation when words or phrases like ¤¤°ê²k«ß, ¤¤°ê²º»â¤g, ¤¤°ê¥DÅv and ¤¤°ê¦D²k are encountered.

Sometimes, patterns which are words can match with text where the matched string of the text is not functioning as a word. For example, ¤j·| (which means conference) can be regarded as a word but in the phrase, ²^¤j·|¶}¤@¶¡·s²º¤À¦æ, the first character (underlined) of the matched string (in italics) is part of a name and the second character (in italics) function as a verb. Thus, Chinese text is often pre-segmented and string searching has to patch delimiters to the beginning and end of the pattern, P. However, the searching accuracy depends on the segmentation algorithm which is usually implemented as a dictionary look-up procedure. If a dictionary has poor coverage, the text tends to be over-segmented (Luk, 1994) and the recall performance of searching will drop drastically. Such cases occur if a general dictionary is used in segmenting technical articles (e.g. in law, medicine, computing, etc).

REFERENCES

BAEZA-YATES, R.A., C. CHOFFRUT, & G.H. GONNET (1994) "On Boyer-Moore automata", *Algorithmica*, **21**, pp. 268-292.

BOYER, R. & S. MOORE (1977) "A fast string searching algorithm", *Communications of ACM*, **20**, pp. 72-772.

CHEN, F-Y., R-P. J. MO, C-R. HUANG, K-J. CHEN (1992) "Reduplication in Mandarin Chinese: their formation rules, syntactic behavior and ICG representation", *Proc. of R.O.C. Computational Linguistics Conference V*, Taipei, Taiwan, pp. 217-233.

COLE, R. (1994) "Tight bounds on the complexity of the Boyer-Moore string matching algorithm", *SIAM Journal of Computing*, **23:5**, pp. 1075-1091.

COLUSSI, L. (1994) "Fastest pattern matching in strings", *Journal of Algorithms*, **16**, pp. 163-189.

CROCHEMORE, M. A. CZUMAJ, L. GASIENIEC, S. JAROMINEK, T. LECROQ, W. PLANDOWSKI, & W. RYTTER (1994) "Speeding up two string-matching algorithms", *Algorithmica*, **12**, pp. 247-267.

HUME, A. AND D.M. SUNDAY (1991) "Fast string searching", *Software - Practice and Experience*, **21:11**, pp. 1221-1248.

KNUTH, D.E., J. MORRIS & V. PRATT (1977) "Fast pattern matching in strings", *SIAM Journal of Computing*, **6**, pp. 323-350.

LUK, R.W.P. (1994) Chinese word segmentation using maximal matching and bigram techniques, *Proc. of R.O.C. Computational Linguistic Conference VII*, Hsinchu, Taiwan, pp. 273-282.

STANDISH, T.A. (1980) *Data Structure Techniques*, Addison-Wesley: Reading, Mass.