Chapter 6 The MP and KMP Algorithms Based upon Rule E2-1

In this chapter, we shall introduce the MP and KMP Algorithms. Both algorithms are based upon Rule E2-1: the suffix to prefix rule.

Section 6.1 The Morris-Pratt Algorithm (MP Algorithm)

The MP Algorithm is named after Morris and Pratt. Unlike the previously introduced algorithms, the MP Algorithm scans the pattern and the window of the text from left to right. Consider Fig. 6.1-1.

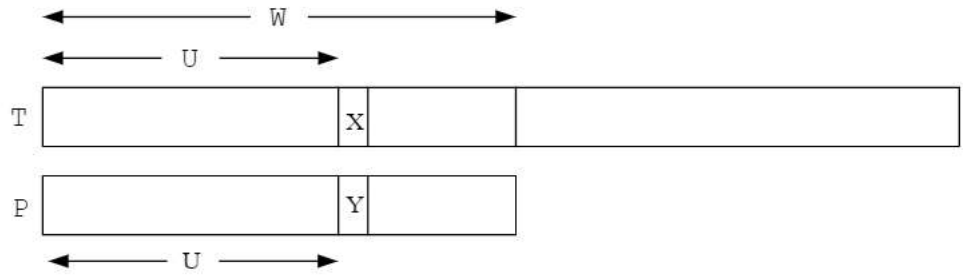


Fig. 6.1-1 A diagram related to the MP Algorithm

In Fig. 6.1-1, we assume that we scan from the left and we get a complete match up to the prefix U of the window W. That is, the prefix U of W is equal to a prefix of P and the next character x to U of W is not equal to the next character y to U of P as shown in Fig. 6.1-1. Therefore, we must shift P. We may consider U as a partial window. If there is a suffix of U of W which is equal to a prefix of U of P, let V the longest one, as shown in Fig. 6.1-2. In this case, we shift P to the right to such an extent that the prefix V of U of P is exactly aligned with the suffix V of U of W, as shown in Fig. 6.1-3, according to Rule E2-1.

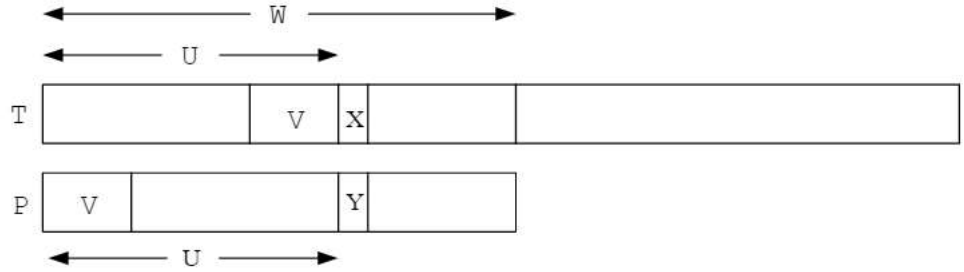


Fig. 6.1-2 The suffix to prefix relationship in the partial window U

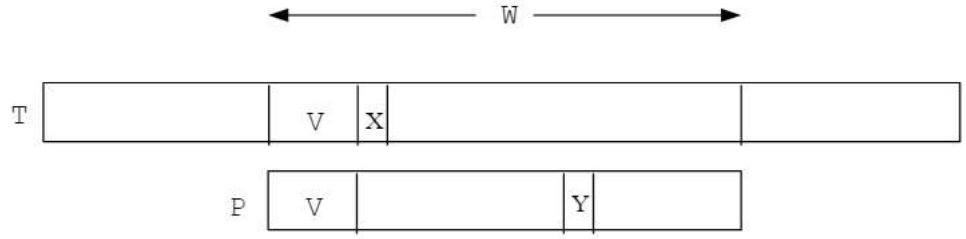


Fig. 6.1-3 One case of moving P in the MP Algorithm

If there is no suffix of U of W which is equal to a prefix of U of P, we shift P to the right as shown in Fig. 6.1-4.

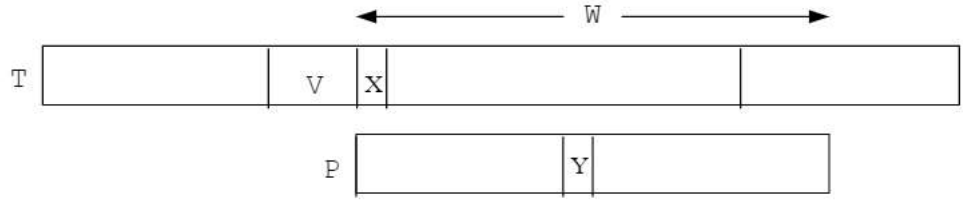
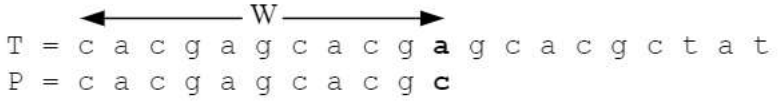


Fig. 6.1-4 Another case of moving P in the MP Algorithm

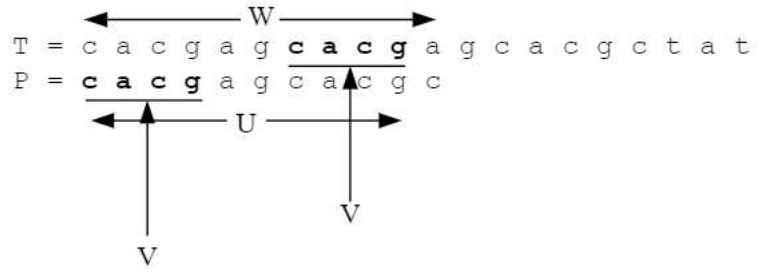
Example 6.1-1

Let $T = cacgagcacgagccacgctaa$ and $P = cacgagcacgc$

For the first window, we note that there is a mismatch as indicated by the bold faced characters, as shown in Fig. 6.1-5(a). We then note that $U = cacgagcacg$ and $V = cacg$ is the longest suffix of U of W which is equal to a prefix of U of P as shown in Fig. 6.1-5(b). We shift P so that the two identical V's are exactly aligned as shown in Fig. 6.1-5(c). As can be seen, an exact match is found.



(a)



(b)

```
T = c a c g a g c a c g a g c a c g c t a t
P =             c a c g a g c a c g c
                          (c)   An exact match is found.
```

Fig. 6.1-5 The behavior of the MP Algorithm for Example 6.1-1

We must understand that the MP Algorithm is efficient if there is a rather long U. If the prefix U is short, the MP Algorithm would shift the pattern P a very small number of steps, as demonstrated in the following example.

Example 6.1-2

Let $T = cacgtactg$ and $P = actg$. The MP Algorithm would behave as in Fig. 6.1-6.

```
T = c a c g t a c t g
P = a c t g
P =   a c t g
P =     a c t g
P =       a c t g
P =         a c t g
P =           a c t g     A matching is found.
```

Fig. 6.1-6 The behavior of the MP Algorithm for Example 6.1-2

As can be seen, the MP Algorithm in this case behaves like a brute force window

sliding algorithm with one step shifting all the time. The reader can see that if the text and the pattern are of random strings, it is very unlikely to have a long prefix U of W which is equal to a prefix of P.

It may appear that we have to find the suffix V of Prefix U of W in the run time. If this is so, the algorithm would not be efficient. Let us consider Fig. 6.1-2 again. If such a suffix V does exist in U of W, it would also be a suffix of Prefix U of P, as shown in Fig. 6.1-7 .

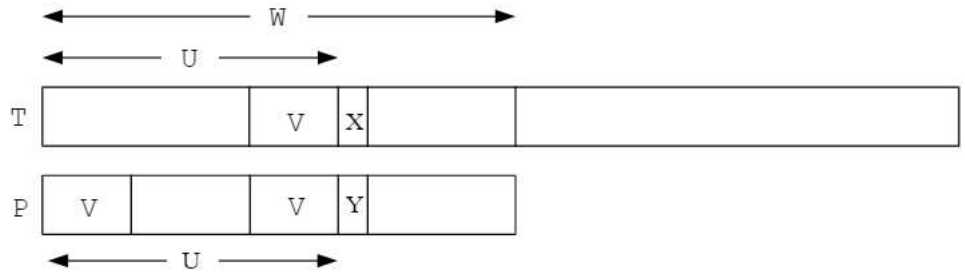


Fig. 6.1-7 The existence of a suffix V in P

Thus, to implement the MP Algorithm, we need to solve the following problem:

Definition 6.1-1 The Self Suffix to Prefix Problem:

The Self Suffix to Prefix Problem: Given a string $S = s_1 s_2 \cdots s_m$, for $1 \le i \le m$, find the longest suffix of $S(1,i) = s_1 s_2 \cdots s_i$ which is also a prefix of $S(1,i)$.

The reader who is familiar with the Reverse Factor Algorithm may find the self suffix to prefix problem to be a special version of the $LSP(X,Y)$ problem. We are given $S = s_1 s_2 \cdots s_m$ and we are to find $LSP(S(1,i), S(1,i))$ for $1 \le i \le m$.

Example 6.1-3

Let us consider the string S in Fig. 6.1-8.

1	2	3	4	5	6	7	8	9
a	c	g	t	a	g	t	a	c

Fig. 6.1-9 A string S for Example 6.1-3

Then, for $i = 5$, we have $S(1,5) = acgta$ and the longest suffix of $S(1,5)$

which is equal to a prefix of $S(1,5)$ is a. This suffix is just $LSP(S(1,5),S(1,5))$. For $i=9$, we have $S(1,9)=acgtagtac$ and the longest suffix of $S(1,9)$ which is equal to a prefix of $S(1,9)$ is ac. Again, this is simply $LSP(S(1,9),S(1,9))$. Of course, we do not care the contents of the suffix; we actually only care about the length of the suffix because this length of the suffix is used to determine the number of steps to shift the pattern.

We can use the methods to find $LSP(X,Y)$ we learned before to solve the self suffix to prefix problem. But it will not be efficient because there are m distinct $S(1,i)$'s and finding each $LSP(S(1,i),S(1,i))$ individually is quite time-consuming. **In the following, we shall show that after we have found $LSP(S(1,i-1),S(1,i-1))$, we can find $LSP(S(1,i),S(1,i))$.**

In the MP Algorithm, we perform a pre-processing and construct a prefix function table which stores the information about the solution of the self suffix to prefix problem on the pattern P. We shall discuss this topic in the following.

Definition 6.1-2 The MP Prefix Function

The MP Prefix Function: Given a string $S=s_1s_2\cdots s_n$, the MP Prefix Function $f(i)$ is the length of the longest suffix of $S(1,i)$ which is also a prefix of $S(1,i)$, for $1\le i\le n$. For $i=0$, $f(i)=f(0)=0$.

Example 6.1-4

Let $S=cgcgagcgcgc$. Then the prefix function of S is as shown in Table 6.1-1 below.

Table 6.1-1 The prefix function of $S=cgcgagcgcgc$

0	1	2	3	4	5	6	7	8	9	10	11
	c	g	c	g	a	g	c	g	c	g	c
0	0	0	1	2	0	0	1	2	3	4	3

Let us examine the entry where $i=3$. For $i=3$, $S(1,3)=cgc$. The longest suffix of $S(1,3)=cgc$ which is equal to a prefix of itself is c. Therefore, $f(3)=1$. For $i=10$, $S(1,10)=cgcgagcgcg$. The longest suffix of $S(1,10)$ which is equal to a prefix of itself is $cgcg$. Therefore, $f(10)=4$.

To construct the prefix function, we need to think recursively. There are two critical parameters which determine the prefix function. A boundary condition for $f(1)$ is $f(1)=0$. This boundary condition is so set to make sure that the longest suffix must be a proper suffix. Suppose we are computing $f(i)$. We first assume that $j=f(i-1)$ has already been computed. There are several possible cases.

Case 1: $s_i = s_{j+1}$.

In this case, because $j=f(i-1)$, we have $S(1,j)=S(i-1-j+1,i-1)=S(i-j,i-1)$. Since $s_i=s_{j+1}$, we have $S(1,j+1)=S(i-j,i)$, as shown in Fig. 6.1-10. Besides, $S(i-j,i)$ must be the longest suffix of $S(1,i)$ which is also a prefix of $S(1,i)$. Thus, we conclude that $f(i)=f(i-1)+1=j+1$.

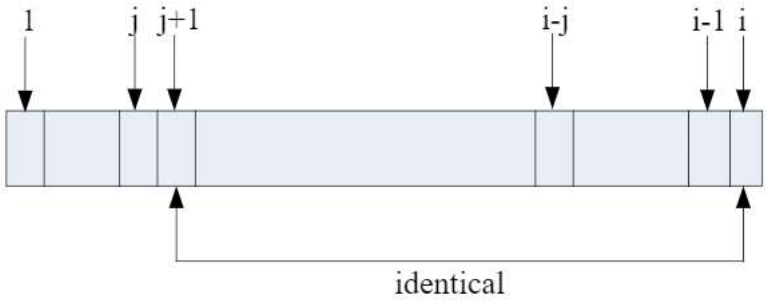


Fig. 6.1-10 Case 1 for the prefix function.

This case can be understood by viewing Table. 6.1-1. Consider $i=7$. $j=f(i-1)=f(6)=0$. $s_i=s_7=c=s_{j+1}=s_{0+1}=s_1$. Thus we have $f(i)=f(7)=j+1=0+1=1$ which is correct. Consider $i=10$. $j=f(i-1)=f(9)=3$ and $s_i=s_{10}=g=s_{j+1}=s_{3+1}=s_4=g$. Therefore, we have $f(i)=f(10)=j+1=3+1=4$ which is again correct.

Case 2: $s_i \neq s_{j+1}$

We first consider a sub-case where $j=0$.

Case 2(a): $s_i \neq s_{j+1}$ and $j=0$.

It is easy to see that in this case, $f(i)=0$

Consider $i = 6$ from Table 6.1-1. We note that $j = f(i-1) = f(5) = 0$ and $s_i = s_6 = g \neq s_{j+1} = s_{0+1} = s_1 = c$. Thus, $f(6) = 0$ Consider $i = 2$. $j = f(i-1) = f(2-1) = f(1) = 0$. $s_i = s_2 = g \neq s_{j+1} = s_{0+1} = s_1 = c$. Thus $f(2) = 0$.

Case 2(b) $s_i \neq s_{j+1}$ and $j \neq 0$.

In this case, we cannot say that $f(i) = 0$ as can be seen by $f(11)$ in Table 6.1-1. For $i = 11$, $j = f(i-1) = f(10) = 4 \neq 0$. But $s_i = s_{11} = c \neq s_{j+1} = s_{4+1} = s_5 = a$. Therefore, we cannot say that $S(7,11)$ is a suffix which is equal to $S(1,5)$. Yet note from Table 6.1-1, there is actually a shorter suffix, namely $S(9,10) = cg$, of $S(1,10)$ which is also a prefix of $S(1,10)$, namely $S(1,2)$. Besides, $s_3 = s_{11}$. Therefore, $S(9,11) = S(1,3) = cgc$. We may conclude that cgc is the longest suffix of $S(1,11)$ which is equal to a prefix of $S(1,11)$. We conclude that $f(11) = 3$ as can be seen in Table 6.1-1.

The question is: How can we find such a shorter suffix of $S(1,10)$ which is also a prefix of $S(1,10)$? This can be done through the following line of reasoning:

1. **From $f(10) = 4$, we know that $S(7,10) = S(1,4)$ and $S(7,10)$ is the longest such suffix of $S(1,10)$.**
2. **For any suffix of $S(1,10)$ which is shorter than $S(7,10)$, it must also be a suffix of $S(1,4)$.**
3. **Therefore, to find a shorter suffix of $S(1,10)$ which is also a prefix of $S(1,10)$, we just find such a suffix in $S(1,4)$.**
4. **That is, we look at $f(4)$. Since $f(4) = 2$, as can be found in Table 6.1-1, we conclude that there is indeed such a shorter suffix.**

The above discussion can be illustrated by the diagram shown in Fig. 6.1-11.

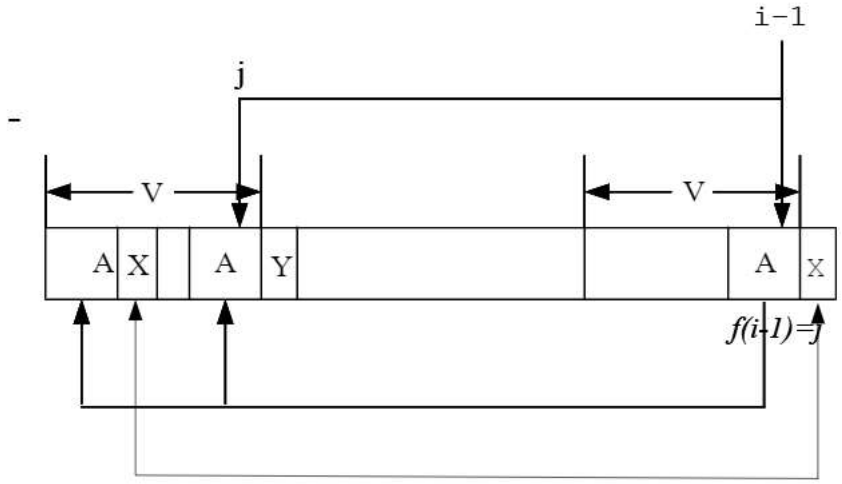


Fig. 6.1-11 An illustration of Case 2(b)

In general, when $s_i \neq s_{j+1}$ and $j \neq 0$ occurs, we concentrate our mind on the fact that $j \neq 0$. We know that there must be a substring V, which is a suffix, equal to a prefix of $S(1,i-1)$. To find a suffix A of $S(1,i-1)$ shorter than V which is also a prefix of $S(1,i-1)$, we simply search through the prefix V. This prefix V can be easily located by using $j = f(i-1)$ and the substring A can be easily found by using the formula $j = f(j)$. For example, in the above case, $j = f(i-1) = f(11-1) = f(10) = 4$. We then find $j = f(j) = f(4) = 2$. We know that there exists a shorter desired suffix whose length is 2. Of course, we have to find whether $s_i = s_{j+1}$ or not again. If the answer is yes, the process is terminated; otherwise, we simply set $j = f(j)$ again. The process is now illustrated as in the Algorithm 6.1.

The following is an algorithm to construct the MP Prefix Function

Algorithm 6.1 An Algorithm to Construct a Prefix Function

Input: A string S with length m

Output: The MP Prefix Function of S

$f(0) = 0$

$f(1) = 0$

For $i = 2$ to $i = m$

$j = f(i-1)$

Step A: If $s_i = s_{j+1}$, $f(i) = j+1$

Else if $j = 0$, $f(i) = 0$;

Else $j = f(j)$ and go to **Step A**.

.

Example 6.1-5

Let us consider Table 6.1-1 which is now displayed below again,

Table 6.1-1 The prefix function of $S = cgcgagcgcgc$

0	1	2	3	4	5	6	7	8	9	10	11
	c	g	c	g	a	g	c	g	c	g	c
0	0	0	1	2	0	0	1	2	3	4	3

Consider $i = 3$. $j = f(i-1) = f(3-1) = f(2) = 0.$ $s_i = s_3 = c = s_{j+1} = s_1$. Therefore, we have $f(i) = f(3) = j + 1 = 0 + 1 = 1.$

Consider $i = 5.$ $j = f(i-1) = f(5-1) = f(4) = 2.$ $s_i = s_5 = a \neq s_{j+1} = s_3 = c$. We further set $j = f(j) = f(2) = 0$. Since $j = 0$ and $s_i = a \neq s_{j+1} = s_1 = c$, $f(i) = f(5) = 0.$

Consider $i = 11$. $j = f(i-1) = f(11-1) = f(10) = 4.$ $s_i = s_{11} = c \neq s_{j+1} = s_{4+1} = s_5 = a$. We further set $j = f(j) = f4) = 2$. $s_i = s_{11} = c = s_{j+1} = s_{2+1} = s_3 = c.$ We have $f(i) = j + 1 = 3.$

Having constructed the MP Prefix Function table, we can use it to decide the number of steps to shift the pattern. Consider Fig. 6.1-13. We assume that the window of T starts from location i and the mismatching happens between t_{i+j-1} and p_j. Then the length of the longest suffix of $P(1, j-1)$ which is also a prefix of $P(1, j-1)$ is $f(j-1)$. Thus we should shift P $(j-1) - f(j-1)$ steps.

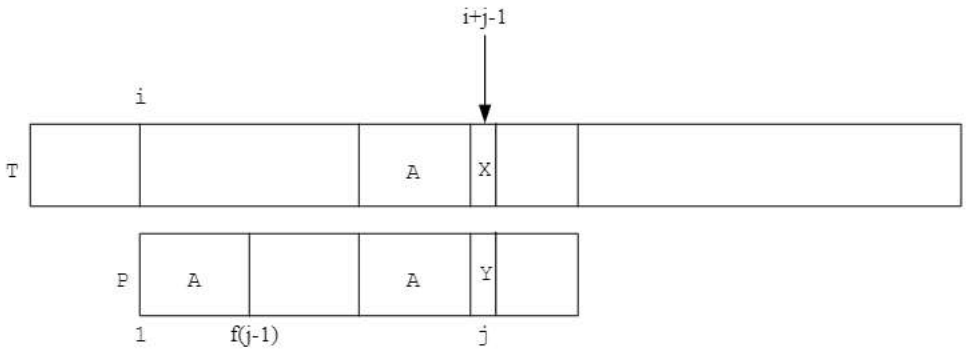


Fig. 6.1-13 The window before shifting

That is, the new window will start at location $i + (j-1) - f(j-1)$. This is shown in Fig. 6.1-14.

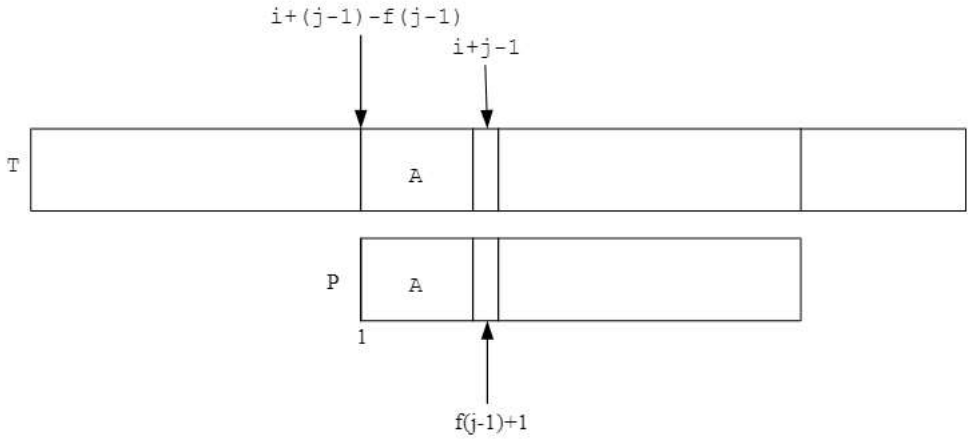


Fig. 6.1-14 The shifting of pattern P in the MP Algorithm

Example 6.1-6

Let us consider $T = agcgcgcgcta$ and $P = gcgcta$. Consider the instance in Table 6.1-2:

Table 6.1-2 An instance to illustrate the MP Algorithm

		1	2	3	4	5	6	7	8	9	10	11
T	=	a	g	c	g	c	g	c	g	c	t	a
P	=		g	c	g	c	t	a				
			1	2	3	4	5	6				

We can see that a mismatch occurs at location 5 of P. Thus $j = 5$ and $j-1 = 4$. It can be seen that $f(j-1) = f(4) = 2$ Thus, we shall shift the pattern $(j-1) - f(j-1) = 4 - 2 = 2$ steps as seen in Table 6.1-3.

Table 6.1-3 An illustration of the shifting of the pattern in the MP Algorithm

		1	2	3	4	5	6	7	8	9	10	11
T	=	a	g	c	***g***	***c***	g	c	g	c	t	a
P	=				***g***	***c***	g	c	t	a		
					1	2	3	4	5	6		

A mismatching again occurs at location 5 of P. We shift the pattern 2 steps again. The resulting situation is now shown in Table 6.1-4. An exact matching is now found.

Table 6.1-4 Another shifting in Example 6.1-6

		1	2	3	4	5	6	7	8	9	10	11
T	=	a	g	c	g	c	g	c	g	c	t	a
P	=						g	c	g	c	t	a
							1	2	3	4	5	6

There is a special case which we must be concerned with. That is, when an exact match is found, how should we shift? The answer is quite easy to obtain, we shift $m - f(m)$ steps.

There is a very important property of the MP Algorithm which should be pointed out here. Let us consider Table 6.1-3. The new window starts from location 4 of T. But, we do not have to compare t_4 and p_1 as well as t_5 and p_2 because we know that $T(4,5) = P(1,2)$. One may easily comprehend this by consulting Fig. 6.1-14. That is, after shifting the pattern P $(j-1) - f(j-1)$ steps, although the new window starts from location $i + (j-1) - f(j-1)$ of T, we start by comparing t_{i+j-1} with $p_{f(j-1)+1}$. This is illustrated in Fig. 6.1-15.

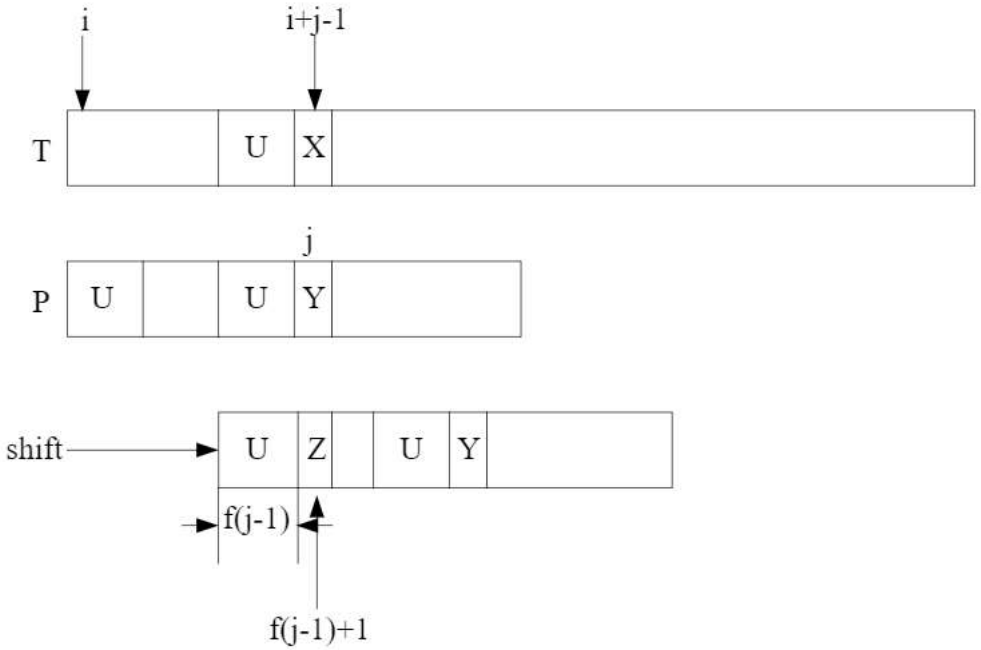


Fig. 6.1-15 The new scanning position in P and T

Let us summarize our previous discussion as follows:

Rule 6.1-1 :

Rules of Shifting for the MP Algorithm:
1. Assume that the present window of T starts at location i of T, scan the window and P from left to right until a mismatching occurs.
2. Assume that the mismatching occurs at location j of P.

3. Shift the window to the right so that it starts at location $i+(j-1)-f(j-1)$. Align the pattern with the new window.

4. Start scanning from location $i+j-1$ of T and location $f(j-1)+1$ of P.

5. After an exact match is found, we shift P $m-f(m)$ steps.

The following is the MP Algorithm.

Algorithm 6.2 The MP Algorithm

Input: Text string T and pattern string P with lengths n and m respectively.

Output: The occurrences of P in T.

Construct the prefix function f of pattern string P.

$i=1$

$j=1$

Step A If $i>n-m+1$, Stop (The window exceeds the text T).
$W=T(i,i+m-1)$
Align P with W.

Step B If $t_{i+j-1}\neq p_j$, $i=i+(j-1)-f(j-1)$, $j=f(j-1)+1$, go to Step A
. (Mismatching occurs. Shift the pattern and open a new window.)
Else
If $j<m$, $j=j+1$, go to Step B.
Else Report "$W=P$; an exact matching is found."
$i=i+m-f(m)$, $j=f(m)+1$, go to Step B.
(Shift the pattern and open a new window.)

For the MP Algorithm, note the following:

(1) All characters of T, except the last $n-m$ ones, will be compared at least once.
(2) After each shift, only one character which was previously compared will be compared again. Since the pattern is shifted from left to the right, such extra comparisons cannot exceed the number of shifts. Since there are at most n shifts, there are at most n such extra comparisons.

Combining the above two statements, we conclude that the total number of comparisons of the MP Algorithm is bounded by $2n-m$. Thus the MP Algorithm is a linear algorithm in the worst case.

The MP Algorithm can be obviously improved. Consider Fig. 6.1-16(a). If the MP Algorithm is used, we would move P immediately so that the prefix U of P is aligned with a substring U of T. But, actually, we should check whether $x=y$. If $x\neq y$, we move P as shown in Fig. 6.1-16(b). This concept will lead us to the KMP Algorithm which will be given in the next section.

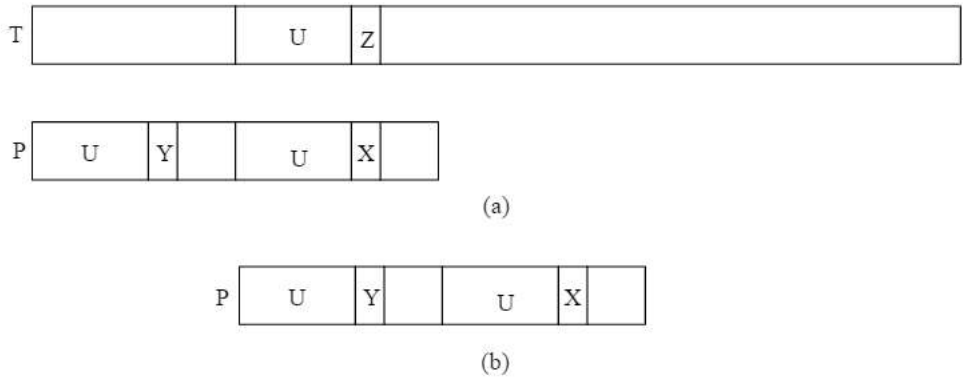


Fig. 6.1-16 The further improvement of the MP Algorithm

One may ask: Is the MP Algorithm an efficient algorithm? To answer this question, let us consider Fig. 6.1-17. Let us assume that the mismatching occurs at location j of P. If j is small, the MP Algorithm degenerates into a case that each shift is very short. The MP Algorithm will be efficient only if j is large. Unfortunately, a simple probabilistic analysis will tell us that this is quite unlikely because the probability that $(j-1)$ characters of the window match exactly with $(j-1)$ characters of the pattern is $\frac{1}{\sigma^{j-1}}$. We may say that the MP Algorithm is basically weak because it scans from the left.

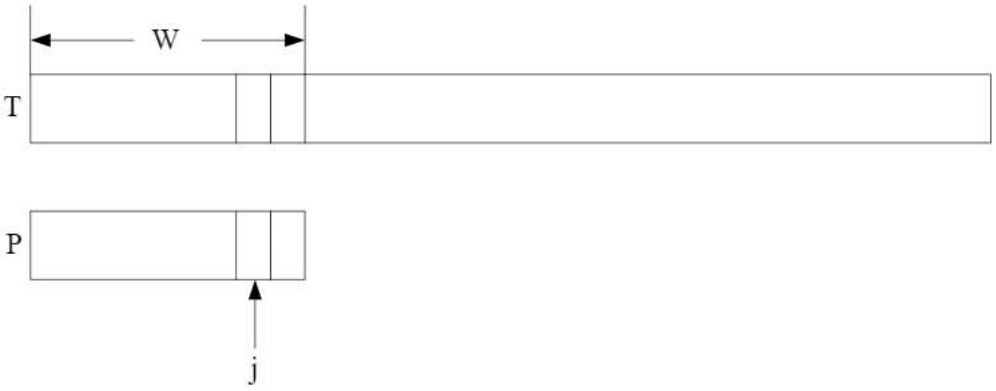


Fig. 6.1-17 The Basic idea of the MP Algorithm

Section 6.2 The Knuth-Morris-Pratt Algorithm (KMP Algorithm)

The Knuth-Morris-Pratt Algorithm (KMP Algorithm for short) further improves the MP Algorithm. Let us assume a mismatching occurs at location j of P and $f(j-1)=t$ where f is the MP function. The main idea of this algorithm

can be explained by the following four cases:

Case 1: $t = 0$ **and** $p_j = p_{t+1} = p_1$.

In this case, we shift $P \quad j$ **steps as shown in Fig. 6.2-1(a)**

Case 2: $t = 0$ **and** $p_j \neq p_{t+1} = p_1$.

In this case, we shift $P \quad (j-1)$ **steps as shown in Fig. 6.2-1(b).**

Case 3: $t \neq 0$ **and** $p_j \neq p_{t+1}$.

In this case, we shift $P \quad (j-1) - f(j-1)$ **steps as shown in Fig. 6.2-1(c).**

Case 4: $t \neq 0$ **and** $p_j = p_{t+1}$.

In this case, we set $t = f(t)$ **and recursively determine** $g(j)$ **as illustrated in Fig. 6.2-1(d).**

Note that in Case 4, h **may be 0.**

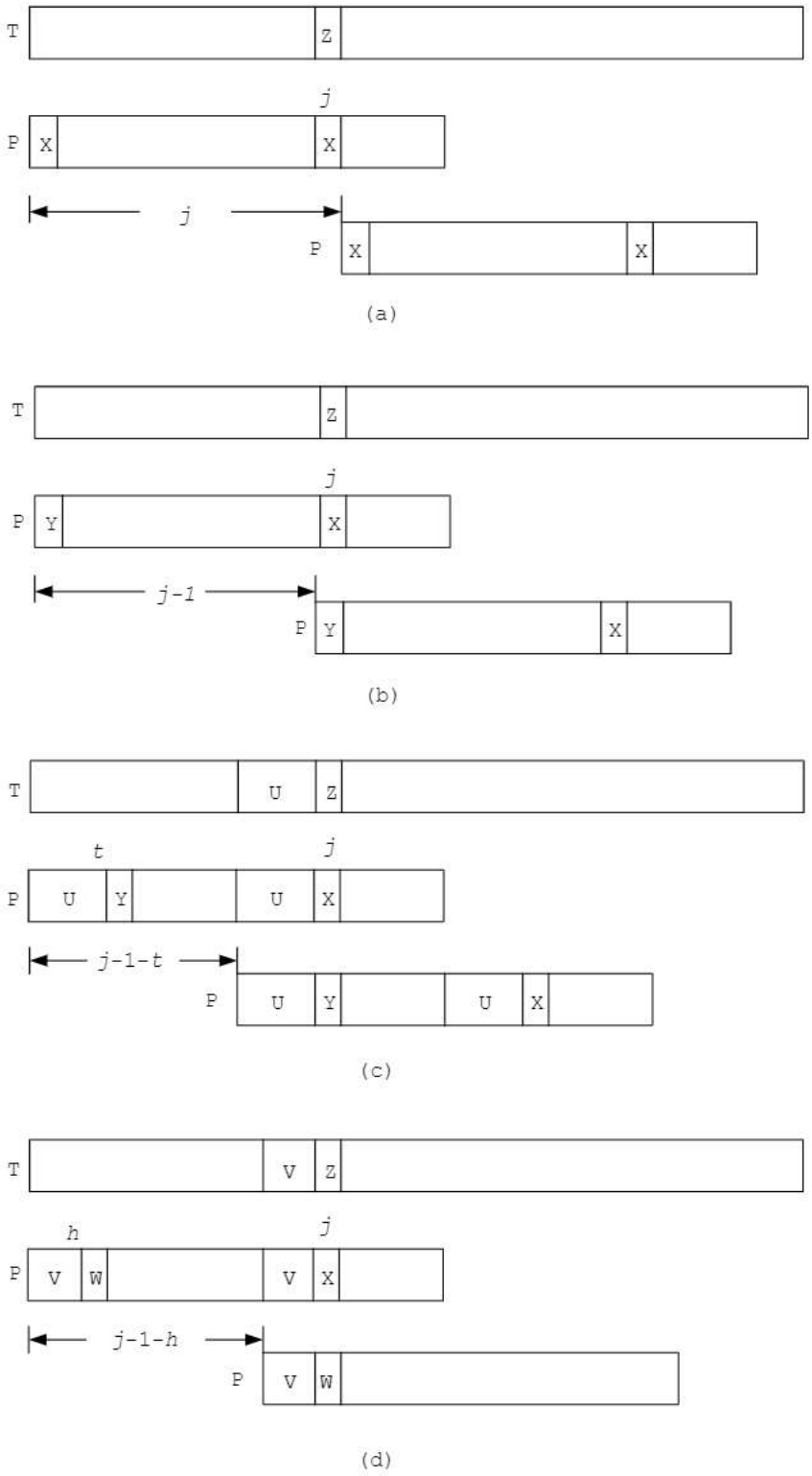


Fig. 6.2-1 An illustration of the KMP Algorithm

Example 6.2-1

Consider $T = gacgagacgactg$ and $P = gacgactg$.

Table 6.2-1 An instance to explain the KMP Algorithm

T	=	g	a	c	g	a	g	a	c	g	a	c	g	t
P	=	***g***	***a***	c	***g***	***a***	c	g	t					
		1	2	3	4	5	6	7	8					

At location 6 of P, there is a mismatching. Let us use the prefix function of the MP Algorithm. We note $f(5)=2$. But $p_6 = p_3 = c$. Thus, we shift P as shown in Table 6.2-2.

Table 6.2-2 A shifting of P in the KMP Algorithm

T	=	g	a	c	g	a	g	a	c	g	a	c	g	t
P	=						g	a	c	g	a	c	g	t
							1	2	3	4	5	6	7	8

We have now found an exact match.

From the above case, we can see that the prefix function of the KMP Algorithm will be different from that of the MP Algorithm. We first give the KMP Prefix Function, denoted as $g(j)$, a formal definition as follows:

Definition 6.2-1 The KMP Prefix Function $g(j)$

KMP Prefix Function $g(j)$: Suppose a mismatching occurs at location j of P. Then $(j-1)-g(j)$ is the number of steps to shift P. In other words, $g(j)$ is $(j-1)-$ (the number of steps to shift). In other words, the number of shifts of the KMP Algorithm is $j-1-g(j)$.

To compute $g(j)$, we shall use the MP prefix function $f(j)$ discussed in the previous section. Let $t=f(j-1)$ in the following discussion. The following cases over all possible cases:

The Rules of the Computation of the KMP Prefix Function $g(j)$

Case 1: $t=0$ and $p_j = p_1$.

In this case, we shift P j steps as shown in Fig. 6.2-1(a). Thus $g(j)=(j-1)-(j)=-1$.

Case 2: $t = 0$ **and** $p_j \neq p_1$.

In this case, we shift P $(j-1)$ **steps as shown in Fig. 6.2-1(b).. Thus** $g(j) = (j-1) - (j-1) = 0$.

Case 3: $t \neq 0$ **and** $p_j \neq p_{t+1}$.

In this case, we shift P $(j-1) - f(j-1)$ **steps as shown in Fig. 6.2-1(c). Thus** $g(j) = f(j-1)$.

Case 4: $t \neq 0$ **and** $p_j = p_{t+1}$.

In this case, we set $t = f(t)$ **and recursively determine** $g(j)$.

Example 6.2-2

Let $P = bcbabcbaebc$. The MP prefix function f and the KMP prefix function g are displayed in Table 6.2-3 below:

Table 6.2-3 An example of the KMP prefix function

j	1	2	3	4	5	6	7	8	9	10	11	12
P	b	c	b	a	b	c	b	a	e	b	c	
f	0	0	1	0	1	2	3	4	0	1	2	
g	-1	0	-1	1	-1	0	-1	1	4	-1	0	2
j-1-g	1	1	3	2	5	5	7	6	4	10	10	9

Let us show all of the possible cases.

Case 1: $t = 0$ and $p_j = p_1$.

This occurs at $j = 5$ . . For $j = 5$, $t = f(j-1) = f(4) = 0$ and $p_j = p_5 = b = p_1$. Thus $g(j) = g(5) = -1$ according to Rule Case 1. As can be seen, this is correct as we must shift $j - 1 - g(j) = 5 - 1 - g(5) = 4 - (-1) = 5$ steps if a mismatching occurs at p_5.

This also occurs at $j = 7$. At the very beginning, this belongs to Case 4 where $t = f(7-1) = f(6) = 2 \neq 0$ and $p_{t+1} = p_{2+1} = p_3 = p_j = p_7 = b$. We conduct a recursive computation which will set $t = f(t) = f(2) = 0$, find $p_{t+} = p_1 = p_j = p_7 = b$ and finally produce $g(7) = -1$ according to Rule Case 1. We shift $j - 1 - g(j) = 7 - 1 - (-1) = 7$ steps.

Case 2: $t = 0$ and $p_j \neq p_1$.

This occurs at $j = 2$. For $j = 2$, $t = f(j-1) = f(1) = 0$ and $p_j = p_2 = c \neq p_1 = b$. Thus, $g(j) = g(2) = 0$ according to Rule Case 2. This is again correct as we can only shift P $j - 1 - g(j) = 2 - 1 - g(2) = 1 - (0) = 1$ step.

Case 3: $t \neq 0$ and $p_j \neq p_{t+1}$.

This occurs at $j = 9$. For $j = 9$, $t = f(j-1) = f(8) = 4 \neq 0$ and $p_j = p_9 = e \neq p_{t+1} = p_5 = b$. Thus $g(j) = g(9) = t = 4$ according to Rule Case 3. We shift $j - 1 - g(j) = 9 - 1 - g(9) = 8 - 4 = 4$ steps. We can easily that this is correct as this case can actually be handled by the MP Algorithm.

Case 4: $t \neq 0$ and $p_j = p_{t+1}$.

This occurs at $j = 6$. For $j = 6$, $t = f(j-1) = f(5) = 1 \neq 0$ and $p_j = p_6 = c = p_{t+1} = p_2$. According to Rule Case 4, we set $t = f(t) = f(1) = 0$ and $p_j = p_6 = c \neq p_{t+1} = p_{0+1} = p_1 = b$. Thus, we now have Case 2 and conclude that $g(j) = g(6) = 0$. We shift $j - 1 - g(j) = 6 - 1 - g(6) = 5 - 0 = 5$ steps.

Before giving the entire algorithm for constructing the KMP Prefix Function, let us note two boundary conditions:

(1) **Boundary Condition for** $g(1)$.

If a mismatching occurs at location $j = 1$ of P, we of course have to shift P 1 step. Thus $j - 1 - g(1) = 1 - 1 - g(1) = 0 - g(1) = -g(1) = 1$. Therefore $g(1) = -1$.

(2) **Boundary Condition for** $g(m+1)$

If there is an exact match, we must shift P $m - f(m)$ steps as we did in the MP Algorithm. So, we may think that there is a mismatching at $j = m + 1$ of P. Thus $j - 1 - g(j) = m - g(m+1) = m - f(m)$. Therefore, $g(m+1) = f(m)$.

The following is the algorithm to construct the KMP prefix function.

Algorithm 6.3: An Algorithm to Construct the KMP Prefix Function
Input: A string S with length m
Output: The KMP prefix function
Compute the MP prefix function f
Set $g(1) = -1$ and $g(m+1) = f(m)$.

For $j = 2$ to $j = m$

$t = f(j-1)$

Step A: If $t \neq 0$,

If $s_j = s_{t+1}$, $t = f(t)$, go to Step A.

Else $g(j) = t$. Stop

Else

If $s_j = s_1$, $g(j) = -1$. Stop.

Else $g(j) = 0$. Stop.

Example 6.2-3

Let us consider the data in Table 6.2-3 which is redisplayed below where $P = bcbabcbaebc$.

Table 6.2-3 An example of the KMP prefix function

j	1	2	3	4	5	6	7	8	9	10	11	12
P	b	c	b	a	b	c	b	a	e	b	c	
f	0	0	1	0	1	2	3	4	0	1	2	
g	-1	0	-1	1	-1	0	-1	1	4	-1	0	2

We shall see how Algorithm 6.3 will find the KMP prefix function in this case. Assume that the MP prefix function f has been calculated. Initially, we set $g(1) = -1$.

When $j = 2$, we have $t = f(j-1) = f(2-1) = f(1) = 0$. In this case, we found that $p_j = p_2 = c \neq p_1 = b$ and we then set $g(2) = 0$ according to Rule Case 2.

When $j = 3$, we have $t = f(j-1) = f(3-1) = f(2) = 0$. In this case, we found that $p_j = p_3 = p_1 = b$ and we then set $g(3) = -1$ according to Rule Case 1.

When $j = 4$, we have $t = f(j-1) = f(4-1) = f(3) = 1 \neq 0$. In this case, we found that $p_j = p_4 = a \neq p_{t+1} = p_2 = c$ and we then set $g(4) = t = 1$ according to Rule Case 3.

When $j = 5$, we have $t = f(j-1) = f(5-1) = f(4) = 0$. In this case, we found that $p_j = p_5 = p_1 = b$ and we then set $g(5) = -1$ according to Rule Case 1.

When $j = 6$, we have $t = f(j-1) = f(6-1) = f(5) = 1 \neq 0$. In this case, we found that $p_j = p_6 = c = p_{t+1} = p_2$. We then set $t = f(t) = f(1) = 0$ according to

Rule Case 4. We recursively have $p_j = p_6 = c \neq p_{t+1} = p_{0+1} = p_1 = b$ and we will conclude that $g(j) = g(6) = 0$ according to Rule Case 2.

When $j = 7$, we have $t = f(j-1) = f(7-1) = f(6) = 2 \neq 0$. In this case, we found that $p_7 = b = p_{2+1} = p_3$. Then we have $t = f(t) = f(2) = 0$ according to Rule Case 4. . We recursively have $p_j = p_7 = b = p_{t+1} = p_{0+1} = p_1 = b$ and we will conclude that $g(j) = g(7) = -1$ according to Rule Case 1.

.

When $j = 8$, we have $t = f(j-1) = f(8-1) = f(7) = 3 \neq 0$. In this case, we found that $p_j = p_8 = a = p_{3+1} = p_4$. Then we have $t = f(t) = f(3) = 1$ according to Rule Case 4. We recursively have $p_j = p_8 = a \neq p_{t+1} = p_{1+1} = p_2 = c$ and we will conclude that $g(j) = g(8) = t = 1$ according to Rule Case 3.

.

When $j = 9$, $t = f(j-1) = f(8) = 4 \neq 0$ and $p_j = p_9 = e \neq p_{t+1} = p_5 = b$. Thus $g(j) = g(9) = t = 4$ according to Rule Case 3.

When $j = 10$, we have $t = f(j-1) = f(10-1) = f(9) = 0$. In this case, we found that $p_j = p_{10} = p_1 = b$. We then set $g(j) = g(10) = -1$ according to Rule Case 1.

When $j = 11$, we have $t = f(j-1) = f(11-1) = f(10) = 1 \neq 0$. In this case, we found that $p_{11} = c = p_{t+1} = p_2$. According to Rule Case 4, we set $t = f(t) = f(1) = 0$. We recursively have $p_j = p_{11} = c \neq p_1 = b$ and $g(j) = g(11) = 0$ according to Rule Case 2.

Finally, we set $g(12) = f(m) = f(11) = 2$.

We can see that our computation based upon Algorithm 6.3 yields exactly the same result shown in Table 6.2-3.

After we have the KMP prefix function, the number of shifts is $(j-1) - g(j)$ if a mismatching occurs at location j of P. Our question now is: Where will be the new scanning start? To answer this question, we must know the number of steps we shift and which new location of T is aligned with p_1. In the following, we shall first compare the MP Algorithm and the KMP Algorithm to show that the KMP Algorithm is somehow more complicated.

Let us give the rules about the new scanning of the MP Algorithm as follows:

Summary of Shifting Rules for the MP Algorithm. Assume that the window starts at location *i* of T and a mismatch occurs at location j of P. We shift $j-1-f(j-1)$ steps, p_1 is aligned with $t_{i+(j-1)-f(j-1)}$ and the new scan

starts at location $f(j-1)+1$ **of** P. **That is, the new** $j'=f(j-1)+1$. **This is shown in Fig. 6.2-2 below.**

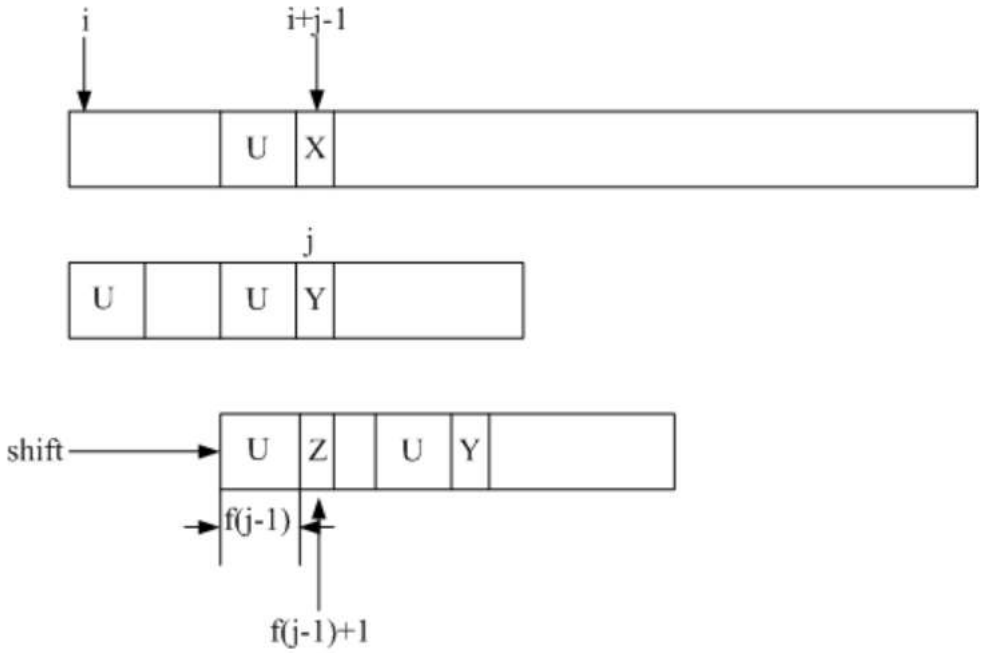


Fig. 6.2-2 The new scanning position of the MP Algorithm

To facilitate our discussion, we display Table 6.2-3 in below. We assume that the window starts with location 1 of the text T and **we first use the MP Algorithm.**

j	1	2	3	4	5	6	7	8	9	10	11	12
P	b	c	b	a	b	c	b	a	e	b	c	
f	0	0	1	0	1	2	3	4	0	1	2	
g	-1	0	-1	1	-1	0	-1	1	4	-1	0	2

Assume $j=6$. In this case, we shift $j-1-f(j-1)=5-1=4$ steps, p_1 is aligned with $t_{1+(j-1)-f(j-1)}=t_{1+4}=t_5$ and the new scan starts at location $f(j-1)+1=1+1=2$ of P. That is, the new $j'=f(j-1)+1=1+1=2$.

Assume $j=7$. In this case, we shift $j-1-f(j-1)=6-2=4$ steps, p_1 is aligned with $t_{1+(j-1)-f(j-1)}=t_{1+4}=t_5$ and the new scan starts at location $f(j-1)+1=1=1=2$ of P. That is, the new $j'=f(j-1)+1=2+1=3$.

We now consider the KMP Algorithm. We shall give the following rules "temporarily" because we shall modify them later:

A Temporary Summary of Shifting Rules for the KMP Algorithm. Assume

that the window starts at location 1 of T and a mismatch occurs at location j of P. We shift $j-1-g(j)$ steps, p_1 is aligned with $t_{1+(j-1)-g(j)}$ and the new scan starts at location $1+g(j)$ of P. That is, the new $j'=1+g(j)$.

We now assume that the KMP Algorithm is used.

Assume $j=6$. In this case, we shift $j-1-g(j)=5-0=5$ steps. p_1 is aligned with $t_{1+(j-1)-g(j)}=t_{1+5}=t_6$ and the new scan starts at location $1+g(j)=1+0=1$ of P. That is, the new $j'=1=g(j)=1+0=1$.

So far so good. But the KMP Algorithm behaves differently as compared with the MP Algorithm, as expected. As we shall see, the temporary rule does not work for $j=7$ where $g(j)=g(7)=-1$.

Assume $j=7$. In this case, we shift $j-1-g(j)=6-(-1)=7$ steps, p_1 is aligned with $t_{1+(j-1)-g(j)}=t_{1+7}=t_8$ and the new scan starts at location $1+g(j)=1+(-1)=0$ of P. That is, the new $j'=1+g(j)=1+(-1)=0$.

The reader can see a clear problem in the above statement because the new scan starts at location 0 of which does not make sense. Why does this happen? Note there are three kinds of shifts in the KMP Algorithm, as shown in the following three cases.

Case 1: $g(j)=0$. As shown in Fig. 6.2-3, we shift P $(j-1)-g(j)=j-1-0=j-1$ steps and the scanning starts from location 1 of P.

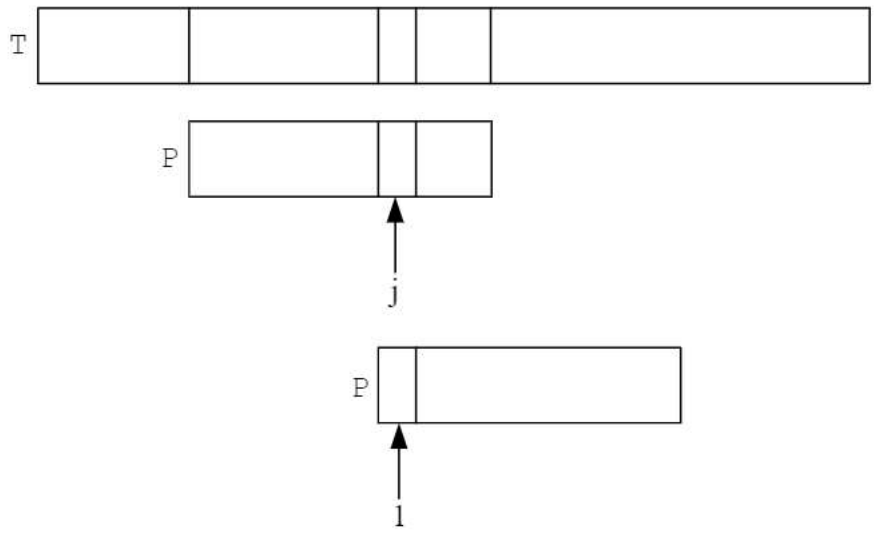


Fig. 6.2-3 The starting point of scanning in P for $g(j)=0$

Case 2: $g(j) = -1$. As shown in Fig. 6.2-4, we shift P $j-1-g(j) = j-1-(-1) = j-1+1 = j$ steps and the scanning again starts from location 1 of P.

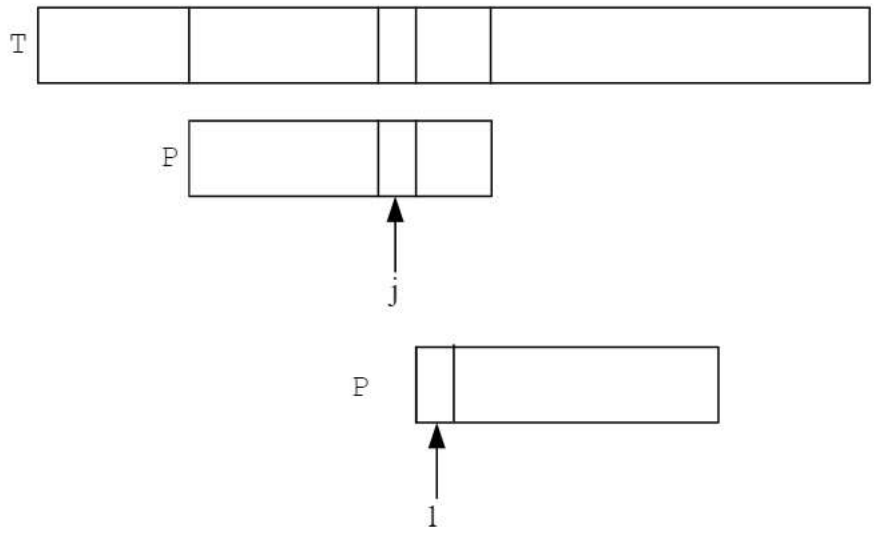


Fig. 6.2-4 The starting point of scanning in P for $g(j) = -1$

Case 3: $g(j) \neq 0$ and $g(j) \neq -1$. In this case, we shift P $(j-1)-g(j)$ steps as shown in Fig. 6.2-5. The scanning starts at location $g(j)+1$ of P.

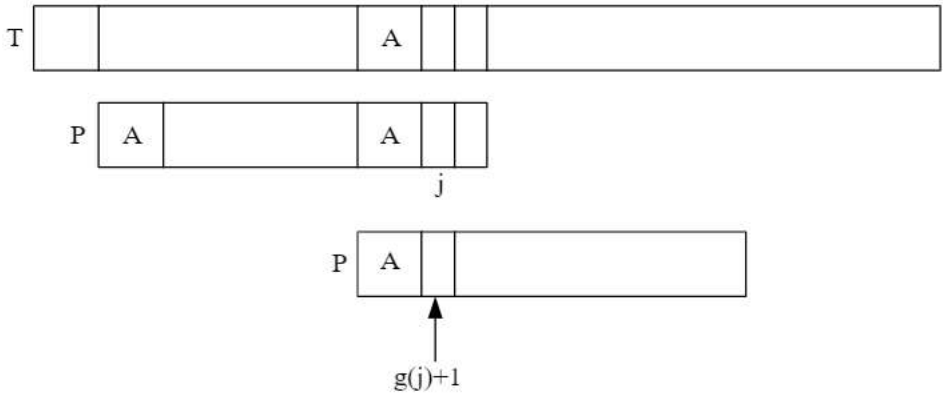


Fig. 6.2-5 The starting point of scanning in P after shifting $(j-1)-g(j)$ steps

Based upon the discussion, we know that in two cases, the scanning starts from location 1 in P and in one case, the scanning starts from location $g(j)+1$, where $g(j) \geq 1$, in P. Note that we cannot use $g(j)+1$ to generally represent the starting location for scanning in P because if $g(j) = -1$, $g(j)+1 = -1+1 = 0$ which is not appropriate. However, we may use $\max(1, g(j)+1)$ to represent the starting location of scanning in P as this will guarantee that the scanning does not start from location 0 in P.

The reader may note the case shown in Fig. 6.2-4 where $g(j) = -1$ does not happen in the MP Algorithm.

The KMP Algorithm is quite similar to the MP Algorithm and is displayed below.

Algorithm 6.4 The KMP Algorithm

Input: Text string T and pattern string P with lengths n and m respectively.

Output: The occurrences of P in T.

Construct the KMP prefix function g of pattern string P.

$i = 1$

$j = 1$

A If $i > n - m + 1$, Stop (The window exceeds the text T).
$W = T(i, i + m - 1)$
Align P with W.

B If $t_{i+j-1} \neq p_j$, $i = i + (j-1) - g(j)$, $j = \max(1, g(j) + 1)$, go to Step A.
(Mismatching occurs. Shift the pattern and open a new window.)
Else
If $j < m$, $j = j + 1$, go to Step B.
Else Report "$W = P$; an exact matching is found."
$i = i + m - g(m+1)$, $j = g(m+1) + 1$, go to Step B.
(Shift the pattern and open a new window.)

Example 6.2-3

In this example, we shall demonstrate how the KMP Algorithm works. First, we let $P = abcabcd$. The KMP prefix function is now constructed as in Table 6.2-4

Table 6.2-4 The KMP Prefix Function of $P = abcabcd$

1	2	3	4	5	6	7	8
a	*b*	*c*	*a*	*b*	*c*	*d*	
-1	0	0	-1	0	0	3	0

Let $T = abcabcabcabcd$. The KMP Algorithm works as shown in Table 6.2-5.

Table 6.2-5 The initialization of the KMP Algorithm for Example 6.2-3

T	=	*a*	*b*	*c*	*a*	*b*	*c*	***a***	*b*	*c*	*d*
P	=	*a*	*b*	*c*	*a*	*b*	*c*	***d***			
		1	2	3	4	5	6	7			

As seen in Table 6.2-5, there is a mismatching at location 7 of P. Thus $j = 7$. From Table 6.2-4, we have $g(j) = g(7) = 3$. Thus we move P $(j-1) - g(j) = (7-1) - 3 = 6 - 3 = 3$ steps as shown in Table 6.2-6. We have now found an exact matching.

Table 6.2-6 A Shifting of P in the AMP Algorithm for Example 6.2-3

T	=	a	b	c	***a***	***b***	***c***	***a***	***b***	***c***	***d***
P	=				***a***	***b***	***c***	***a***	***b***	***c***	***d***
					1	2	3	4	5	6	7

Fig. 6.2-6 The working of the KMP Algorithm on
$T = abcabcabcabcd$ and $P = abcabcd$

The time-complexity of the KMP Algorithm is the same as the MP Algorithm.

Section 6.3 The Simon Algorithm

The Simon Algorithm further improves the KMP Algorithm. Consider Fig. 6.3-1. Let A denote the prefix of P which is equal to a suffix of a partial window of T. The Simon Algorithm checks whether the character immediately after A in P is equal to the character after A in T. If not, we move even further.

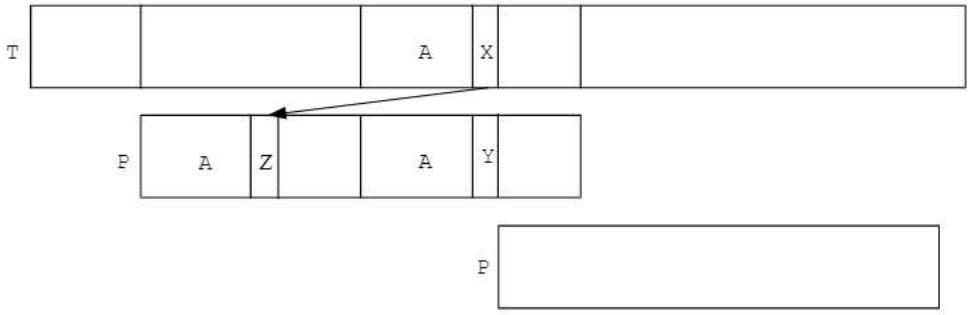


Fig. 6.3-1 The Simon Algorithm

The further checking is done in run-time. Since it takes a constant time to do the checking, the time-complexity of the Simon Algorithm is the same as that of the MP Algorithm.

Section 6.4 The Time-Complexity Analysis of the KMP Algorithm

The worst case time-complexity of the KMP Algorithm is the same as that of the MP

Algorithm, namely $O(n)$.

Before we introduce a Markov Chain analysis of the average case time-complexity of the KMP Algorithm, let us consider the following example in Table 6.4-1.

Table 6.4-1 An example to explain the two classes of text characters in the KMP Algorithm

		1	2	3	4	5	6	7	8	9	10	11
T	=	a	c	t	a	a	g	t	c	a		
P	=	a	c	t	a	g						
					a	c	t	a	g			
						a	c	t	a	g		

We can see that all characters in T, except $t_5 = a$, is compared exactly only once while t_5 is compared more than once. It is first compared with p_5 and then with p_2 and p_1. We therefore may divide all of the characters in T into two classes: those which are compared only once in the KMP and those which are compared more than once. We note that there are three states during the execution of the KMP Algorithm:

S_1 : t_i is compared only once and compared with p_1

S_2 : t_i is compared only once and compared with some character other than p_1.

S_3 : t_i is compared more than once.

The relationship among these three states is now displayed in Fig. 6.4-1.

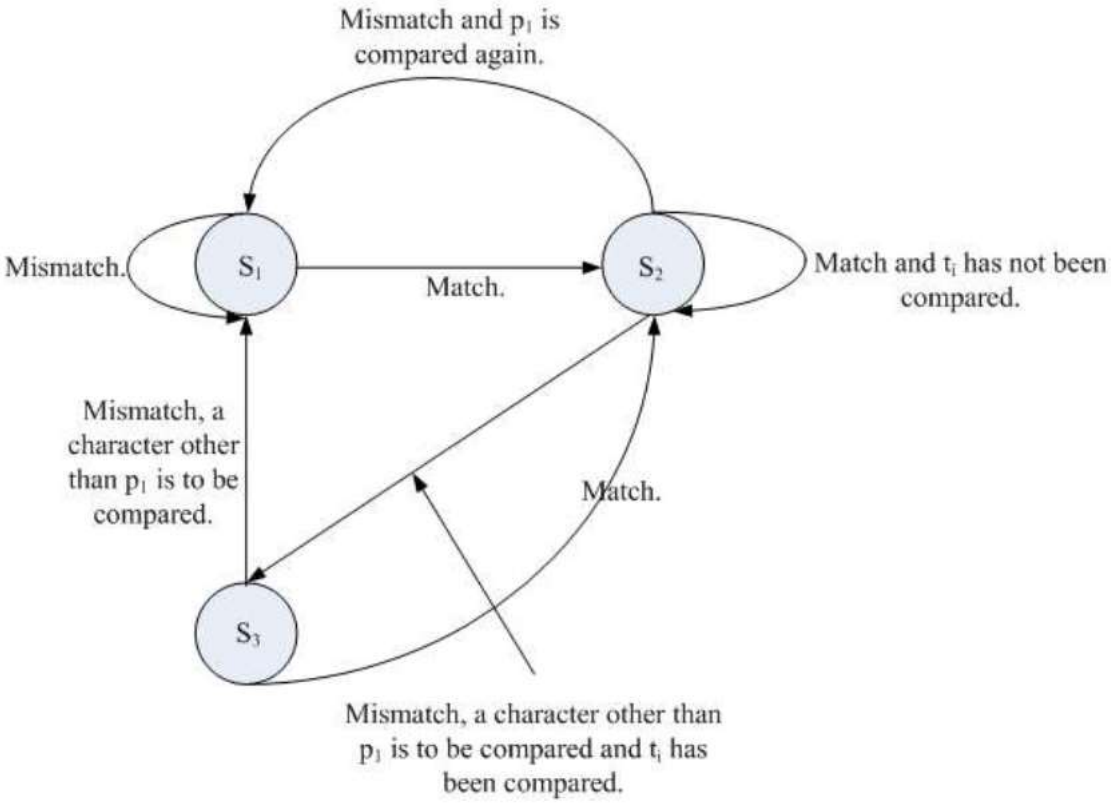


Fig. 6.4-1 The three Markov chain states of the KMP Algorithm

In Fig. 6.4-1, each edge corresponds to the situation that the KMP Algorithm compares another text character. An example is now given to describe the diagram.

Example 6.4-1 An example to explain the Markov chain approach for analyzing the average case time-complexity of the KMP Algorithm.

Consider the data as well as the execution of the KMP Algorithm in Table 6.4-2

Table 6.4-2 The three states in a KMP Algorithm execution

		1	2	3	4	5	6	7	8	9	10	11	12	13

T	=	a	c	t	a	c	c	a	c	t	a	c	g	
P	=	a	c	t	a	c	g							
		s_1	s_2	s_2	s_2	s_2	s_2							
					a	c	t	a	c	g				
							s_3							
							a	c	t	a	c	g		
							s_3							
								a	c	t	a	c	g	
								s_1	s_2	s_2	s_2	s_2	s_2	

We specify three state transition probabilities, x, y and z as shown in Fig. 6.4-2.

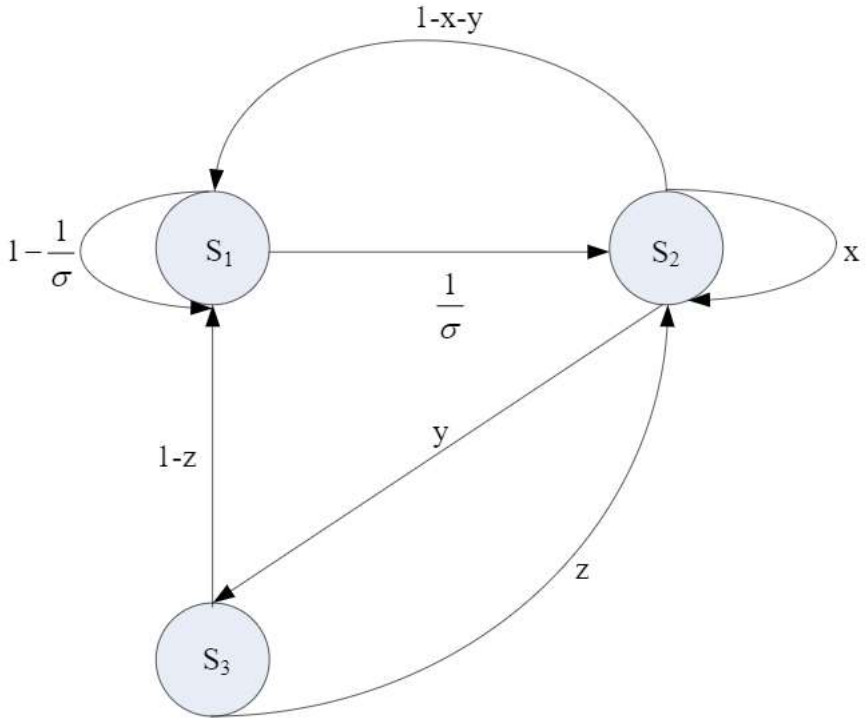


Fig. 6.4-2 The transition probabilities of the KMP Algorithm.

We do not know the values of x, y and z. Yet, it is easy to see the bounds of them as below:

$$0 \le x \le \frac{1}{\sigma}$$

$$0 \le y \le 1 - \frac{1}{\sigma}$$

$$\frac{1}{\sigma - 1} \le x \le 1$$

Let the probability of staying in S_i be denoted as P_i for $i = 1,2,3$. Then we have:

$$P_1 + P_2 + P_3 = 1 \tag{6.4-1}$$

$$P_1 = (1 - \frac{1}{\sigma})P_1 + (1 - x - y)P_2 + (1 - z)P_3 \tag{6.4-2}$$

$$P_2 = \left(\frac{1}{\sigma}\right)P_1 + xP_2 + zP_3 \tag{6.4-3}$$

$$P_3 = yP_2 \tag{6.4-4}$$

Solving these equations, we have:

$$P_3 = \frac{y}{\sigma + 1 + y(1 - \sigma z) - \sigma x} \tag{6.4-5}$$

Let N_i be the average number of times to stay in State S_i and $\overline{C_n}$ be the average number of comparisons for different instances of T with length n. We cannot directly find N_i and S_i because we can not enumerate all instances of T with length n. We can, however, have some feeling about these terms by considering the case shown in Example 6.4-1. Instead of N_i, $\overline{C_n}$ and P_i, we shall use N_i', $\overline{C_n}'$ and P_i' because we are talking about a particular case. In this case, we have:

$$N_1' = 2$$
$$N_2' = 10$$
$$N_3' = 2$$

$$\overline{C_n}' = 12 + 2 = 14$$

$$P_1' = \frac{2}{14}$$
$$P_2' = \frac{10}{14}$$
$$P_3' = \frac{2}{14}$$

We can easily see that in this case, $N_i = \overline{C_n}' \times P_i'$ for $i = 1,2,3$. We can now derive some formulas. Based upon the above definitions, we have

$$P_i = \frac{N_i}{\overline{C_n}} \tag{6.4-6}$$

For the KMP Algorithm,

$$N_1 + N_2 = n. \tag{6.4-7}$$

Therefore, we have:

$$P_1 + P_2 = \frac{N_1 + N_2}{\overline{C_n}} = \frac{n}{\overline{C_n}} \tag{6.4-8}$$

Combining (6.4-8) and (6.4-1), we obtain:

$$\overline{C_n} = \frac{n}{1 - P_3} \tag{6.4-9}$$

It can be proved that $\overline{C_n}$ will be maximized by setting $x = \frac{1}{\sigma}$, $y = 1 - \frac{1}{\sigma}$ and $z = 1$

Thus,

$$\overline{C_n} \le \left(2 - \frac{1}{\sigma}\right) n \tag{6.4-10}$$

We conclude that the average case time-complexity of the KMP Algorithmis $O(n)$.

Section 6.5 The Improved KMP Algorithm

At the end of the original KMP Paper, there is an improved version of the KMP Algorithm. This improved algorithm uses the basic idea of the filtering concept introduced in Section 4.10 which is based upon Rule E1 displayed below.

Rule E1: If P exactly appears in W, every substring of W will exactly appear in P. Equivalently, if any substring of W does not appear in P, we may conclude that P does not appears in W.

The improved KMP Algorithm uses a suffix r of the window with length $|r| = \lfloor 2\log_{\sigma} m \rfloor$. If r does not appear in pattern P, we shall skip the window as shown in Fig. 6.5-1. We can see that Rule E1 is used as a filtering mechanism so that many windows may be ignored. Whether r appears in P can be determined by using the suffix tree approach or the bit vector approach introduced in Chapter 1.

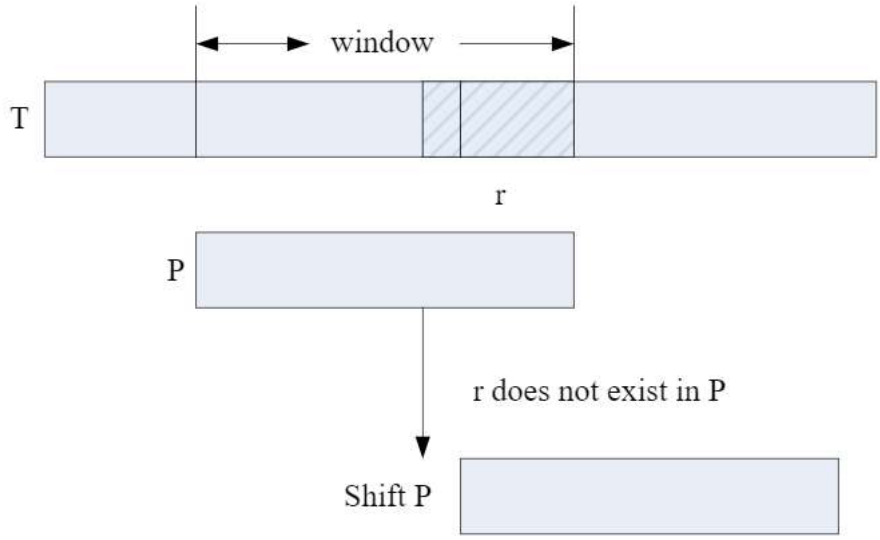


Fig. 6.5-1 The shifting in the improved KMP Algorithm

For the improved KMP Algorithm, one peculiar aspect of it is to open a wider window to KMP Algorithm as shown in Fig. 6.5-2.

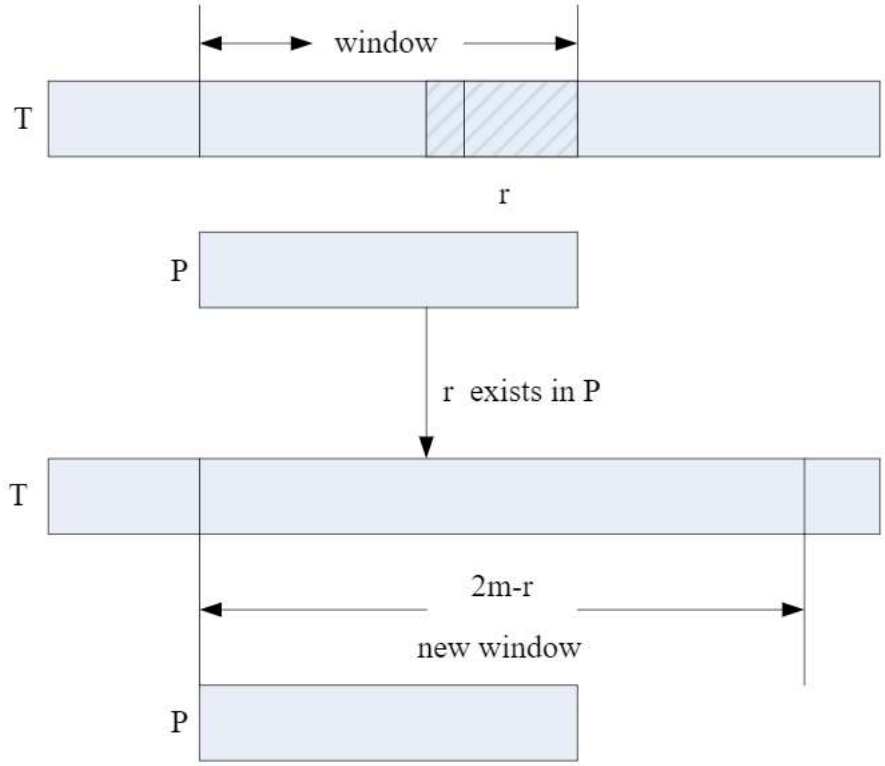


Fig. 6.5-2 The wide window for the improved KMP Algorithm

The Improved KMP Algorithm thus consists of two stages: the filtering stage and the KMP stage. The reason for the use a wider window will be explained later when we analyze the time-complexity of the algorithm.

Example 6.5-1

Consider the following data:

T	:	a	g	c	a	g	**c**	**a**	g	c	t	a	g	a	g	c	a	t	c	t
P	:	a	g	**c**	**a**	g	c	t												

In this case, $m = 7$ and $\sigma = 4$. We have $|r| = \lfloor \log_4 7 \rfloor = 2$. We first find that the suffix $r = ca$ of the window appears in P. A window of length $2m - |r| = 14 - 2 = 12$ is opened. The KMP Algorithm is now executed and we shift the pattern as shown below:

T	:	a	g	c	a	g	c	a	g	c	t	a	g	a	g	c	a	t	c	t
P	:				a	g	c	a	g	c	t									

An exact match is found and the pattern is shifted as shown below:

T	:	a	g	c	a	g	c	a	g	c	t	a	g	a	g	c	a	t	c	t
P	:											a	g	c	a	g	c	t		

This time, $r = at$, it is not found in P and we do not have to perform the KMP Algorithm. As can be seen, the pattern cannot be shifted any more as it will be out of the boundary.

We now try to answer the question: Why do we need to open a wider window to execute the KMP Algorithm? Suppose we use a smaller window, as shown in Fig. 6.5-3.

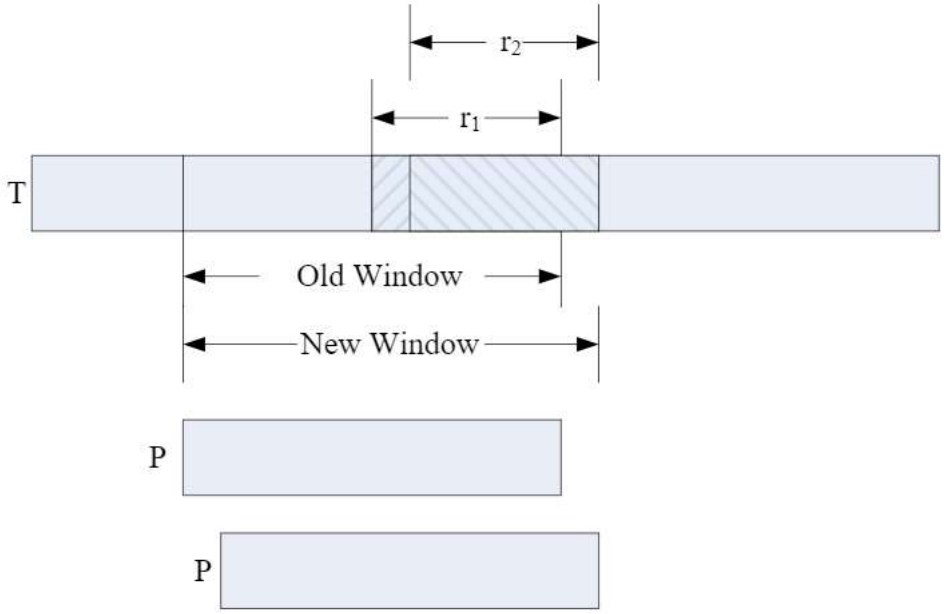


Fig. 6.5-3 The problem of having a small window for the improved KMP Algorithm

As one can see, the two suffixes overlap. This creates a problem. That is, we can no longer assume that the second suffix is random any more. For our wider window, the situation will be as shown in Fig. 6.5-4. There is no overlapping any more.

Fig. 6.5-4 The two suffixes in a wider window,

The following is the Improved KMP Algorithm.

Algorithm 6.5 The Improved KMP Algorithm:

Input: Text string T and pattern string P with lengths n and m respectively

and σ being the size of the vocabulary.

Output: The occurrences of P in T.

$i = 1$

While $i \le n - m$ do

If $r = T(i + m - \lfloor 2\log_\sigma m \rfloor, i + m - 1)$ is a substring of P. Apply the KMP Algorithm with window with length $2m - |r|$ starting from i.

Else $i = i + m - |r| + 1$

End of While

The Average Case Time-Complexity Analysis of the Improved KMP Algorithm

In average case analysis, when $|r| = \lfloor 2\log_\sigma m \rfloor$, there are $\sigma^{2\log_\sigma m} = m^2$ possible permutations of $|r|$ characters. Consider each substring of P with length $|r|$. We want to know whether this substring is r. This substring must be one of the m^2 permutations. This situation is as if we are taking $m - |r| + 1 = O(m)$ substrings with length $|r|$ from m^2 permutations and we want to know the probability that the taken substring is r.

Thus Pr(r occurs in P) $\le \frac{m}{m^2} = O\left(\frac{1}{m}\right)$.

If r does not appear in P, no more comparisons are needed. We just shift.

Let $\overline{C_{|W|,m}}$ be the expected number of character comparisons for an iteration. The number of comparisons of the filtering process to determine whether r appears in P by using suffix tree of P is $1 \times |r| = 2\log_\sigma m$.

The expected number of comparisons needed by the KMP algorithm is less than $O\left(\frac{1}{m}\right)(2m - |r|) = O\left(\frac{1}{m}\right)O(m) = O(1)$.

Thus $\overline{C_{|W|,m}} = 2\log_\sigma m$.

The Filtering Stage

The filtering stage introduced in the above is not only good for the KMP Algorithm. It is good for almost every window sliding algorithm. The critical action needed is to decide whether a short suffix of a window appears in the pattern or not. Although we may consider this problem an ordinary string matching problem, we actually may not think so because the suffix is always very short. We may modify any algorithm introduced in Chapter 4 to solve the *LSP* finding problem.

In the following, let us give an example to present the modified Chu's Algorithm to solve the short suffix finding problem.

Assume that $T = acctgtta$, $P = aatacttg$ and $S = ta$.

Step 1: We set $D = (0,0,0,0,0,0,0,0)$ to start with. Since $s_2 = a$ appears in locations 1, 2 and 4 of P, we set

$$D = (1,1,0,1,0,0,0,0).$$

Step 2: We check whether $s_1 = t$ appears in locations 1 and 3 of P. It appears in location 3. So, we report that $S = ta$ appears in P.

We can use an even more efficient method. Note that P is also short. There are therefore only a small number of substrings of P with the same length of the suffix. In the above case, the suffix length is 2. We can list all of the substrings of P as follows:

$$(aa, at, ta, ac, ct, tt, tg)$$

Let us now sort them alphabetically as follows:

$$(aa, ac, at, ct, ta, tg, tt).$$

For every suffix with length 2, we can quickly report whether it appears in P by conducting a binary search of the above list. For instance, we can quickly see that ga does not appear in P because it does not exist in the above list.

Of course, we may also use hashing mechanism.

Section 6.6 The Colussi Algorithm

In this section, we shall introduce the Colussi Algorithm which is a further improvement of the KMP Algorithm. Let us recall in Section 6.2, a function $g(j)$ is defined. According to Definition 6.2-1, the number of steps to shift in the KMP Algorithm is $j-1-g(j)$ when a mismatch occurs at location j. The smallest value of $g(j)$ is -1. Thus $j-1-g(j) = j-1-(-1) = j$ is the maximum number of steps one can shift in the KMP Algorithm. The Colussi Algorithm focuses on this particular case where $g(j) = -1$.

Let us recall, from the discussion in Section 6.2 that $g(j) = -1$ if there is no suffix of $P(1, j-1)$ which is equal to a prefix $P(1, x-1)$, $p_i \neq p_x$ and $p_j = p_1$.

Table 6.6-1 displays an example of $g(j)$

Table 6.6-1 An example of $g(j)$

j	1	2	3	4	5	6	7	8	9	10	11
P	b	c	b	a	b	c	b	a	e	b	c
$g(j)$	-1	0	-1	1	-1	0	-1	1	4	-1	0

Let us consider the case shown in Table 6.6-2.

:

Table 6.6-2 Another example of $g(j)$

		1	2	3	4	5	6	7	8	9
T	=	a	c	c	a	b	c	b	a	a
P	=	b	c	b	a	a				
$g(j)$	=	-1	0	-1	1	0				

Suppose that the KMP Algorithm is used. We would have the process shown in Table 6.6-3. An exact match is found at the 5th step.

Table 6.6-3 The KMP Algorithm applied to the data in Table 6.6-2

	a	c	c	a	b	c	b	A	a
1	b	c	b	a	a				
2		b	c	b	a	a			
3			b	c	b	a	a		
4				b	c	b	a	a	
5					b	c	b	a	a

Let us now temporarily ignore all of the locations where $g(j) = -1$. In the above case, in locations 2, 4 and 5, $g(j) \neq -1$. The process would be as shown in Table 6.6-4

:

Table 6.6-4 Table 6.6-2 redisplayed

j		1	2	3	4	5	6	7	8	9				
T	=	a	c	c	a	b	c	b	a	a				
P	=	b	c	b	a	a								
$g(j)$	=	-1	0	-1	1	0								

1. At location 2, $t_2 = p_2$. No shift.
2. At location 2, $t_4 = p_4$. No shift.

3. At location 5, $t_5 \neq p_5$. We shift $j-1-g(j)=5-1-0=4$ steps. The result is as shown in Table 6.6-5:

Table 6.6-5 The process of the Colussi Algorithm applied to the data in Table 6.6-4

j		1	2	3	4	5	6	7	8	9				
T	=	a	c	c	a	b	c	b	a	a				
P	=					b	c	b	a	a				

An exact match is found.

We can see that by ignoring the locations where $g(j)=-1$, we still can obtain the same result. Let us consider another example.

Example 6.6-1

Consider the data in Table 6.6-6:

Table 6.6-6 Data for Example 6.6-1

j	1	2	3	4	5	6	7	8	9	10	11
W	a	c	c	a	b	c	b	a	e	b	b
P	b	c	b	a	b	c	b	a	e	b	c
$g(j)$	-1	0	-1	1	-1	0	-1	1	4	-1	0

We first match locations 2, 4, 6, 8 and 9 where $g(j)\neq -1$. There are exact matches at all of these locations. A mismatch occurs at location 11. Then we shift $j-1-g(j)=11-1-0=10$ steps which is correct.

We now give the basic principle of the Colussi Algorithm.

The Basic Principle of the Colussi Algorithm

Given a window W and a pattern P, we may examine, from left to right, the locations where $g(j)\neq -1$, shift if necessary, before we examine, from left to right, the locations where $g(j)=-1$.

When $g(j)\neq -1$, there are two possible cases: Case 1: $g(j)=0$ and Case 2: $g(j)>0$ This basic principle of the Colussi Algorithm is formally stated in two lemmas.

Lemma 6.6-1 Case 1 of the Colussi Algorithm

Given a window W and a pattern P, if a mismatch occurs at location j where $g(j)=0$, for all of locations x's before location j where $g(x)\neq -1$, $w_x = p_x$, and all locations x's where before location j where $g(x)=-1$ have not been examined, we may shift $j-1-g(j)$ steps.

Proof: We essentially will prove that for $1 \le i \le j-1$, $W(i,j) \neq P(1, j-i-+1)$. For any location $k < j$ in P, there are the following three possible cases:

Case 1.1 $p_k \neq p_1$.
Case 1.2 $p_k = p_1$ and there exists an $i, k < i < j$, $P(k,i-1) = P(1,i-k)$ and $p_i \neq p_{i-k+1}$.
Case 1.3 $p_k = p_1$ and no such $i < j$ exists.
.

We now prove case by case.

Case 1.1. Since $p_k \neq p_1$, $g(k) \neq -1$. According to our assumption, $w_k = p_k \neq p_1$. Thus $W(k,j) \neq P(1, j-k-+1)$.

Case 1.2. $p_k = p_1$ and there exists an $i, k < i < j$, $P(k,i-1) = P(1,i-k)$ and $p_i \neq p_{i-k+1}$. In this case, $g(i) \neq -1$. According to our assumption, $w_i = p_i \neq p_{i-k+1}$. Thus $W(k,j) \neq P(1, j-k-+1)$.

Case 1.3. $p_k = p_1$ and no such $i < j$ exists.. Note that the condition of Case 1.2 can be viewed as $\exists i(A(i) \text{and } B(i))$. The negation of this condition is therefore $\forall i(\neg A(i) \vee \neg B(i)) \equiv \forall i(A(i) \rightarrow (\neg B(i)))$. Let us assume that $P(k,i-1) = P(1,i-k)$. Thus according to the condition of Case 1.3, $p_i = p_{i-k+1}$. Therefore, $P(k,i) = P(1,i-k+1)$. Let $i = j-1$. We have $P(k,j-1) = P(1,j-k)$. Because $g(j)=0$, $p_j = p_{j-k+1}$. Thus $P(k,j) = P(1,j-k+1)$. As assumed, $w_j \neq p_j = p_{j-k+1}$. In conclusion, $W(k,j) \neq P(1,j-k+1)$. Q.E.D.

In the following, we present some examples to illustrate the physical meaning of Lemma 6.6-1

Example 6.6-2 An Example of Case 1.1 in Lemma 6.6-1

In this case, $p_k \neq p_1$. Consider the data for Example 6.6-1 as shown below:

		$k=2$		$k=4$		$k=6$		$k=8$	$k=9$		$j=11$
j	1	**2**	3	**4**	5	**6**	7	**8**	**9**	10	11
W	a	***c***	c	***a***	b	***c***	b	***a***	***e***	b	b
P	b	***c***	b	***a***	b	***c***	b	***a***	***e***	b	c
$g(j)$	-1	**0**	-1	**1**	-1	**0**	-1	**1**	**4**	-1	0

The mismatch occurs at location 11. Before locations 11, for location x which is equal to 2, 4, 6, 8 and 9, $p_x \neq p_1$. Therefore $g(x) \neq -1$. Thus $p_x = w_x \neq p_1$.

Example 6.6-3 An Example of Case 1.2 in Lemma 6.6-1

In this case, $p_k = p_1$ and there exists an $i, k < i < j$, $P(k, i-1) = P(1, i-k)$ and $p_i \neq p_{i-k+1}$. Again, consider the data for Example 6.6-1 as shown below:

					$k=5$				$i=9$	$j=10$	
J	1	**2**	3	**4**	5	**6**	7	**8**	**9**	10	11
W	a	***c***	c	***a***	b	***c***	b	***a***	***e***	b	b
P	b	***c***	b	***a***	b	***c***	b	***a***	***e***	b	c
$g(j)$	-1	**0**	-1	**1**	-1	**0**	-1	**1**	**4**	-1	0

For $k=5$, there exists $i=9$ such that $P(5,8) = P(1,4)$ and $p_5 \neq p_9$. Because $g(9) \neq -1$, $p_5 \neq p_9 = w_9$. Therefore $W(5,11) \neq P((1,7)$.

Example 6.6-4 An Example of Case 1.3 in Lemma 6.6-1

In this case, $p_k = p_1$ and no such i exists. Again, consider the data for Example 6.6-1 as shown in Table 6.6-7:

Table 6.6-7 An Example of case 1.3 in Lemma 6.6-1

						$k=7$				$j=10$	
J	1	**2**	3	**4**	5	**6**	7	**8**	**9**	**10**	11
W		***c***	c	***a***		***c***		***c***		***b***	
P	b	***c***	b	***a***	b	***c***	b	***c***	b	***a***	b
$g(j)$	-1	**0**	-1	**1**	-1	**0**	-1	**0**	-1	**0**	-1

For $k=7$, we have $P(7,10) = P(1,4)$. But $p_4 = p_{10} \neq w_{10}$ because a mismatch occurs at . $j=10$. Therefore, $W(7,10) \neq P(7,10)$. Case 1.3 occurs at $k=7$.

In the following, we shall give the lemma concerning with Case 2 where $g(j) > 0$.

Lemma 6.6-2 Case 2 of the Colussi Algorithm

Given a window W and a pattern P, if a mismatch occurs at location j where $g(j) > 0$, for all of locations x's before location j where $g(x) \neq -1$, $w_x = p_x$, and all locations x's where before location j where $g(x) = -1$ have not been examined, we may shift $j - 1 - g(j)$ steps.

The proof of this lemma is quite similar to that of Lemma 6.6-1 and will not be given.

Reference

[AC91] Apostolico, A. and Crochemore, M., Optimal canonization of all substrings of a string, Information and Computation, Vol. 95, No. 1, 1991, pp.76.95.

[C91] COLUSSI L., Correctness and efficiency of the pattern matching algorithms, Information and Computation, Vol. 95, No. 2, 1991, pp.225-251.

[CLP98] Charras, C., Lecroq, T. and Pehoushek, J. D., Very Fast String Matching Algorithm for Small Alphabets and Long Patterns, Proceeding of the 9th Annual Symposium on Combinatorial Pattern Match, Lecture Notes in Computer Science, Vol. 1448, 1998, pp. 55-64.

[FJS2007] Franek, F., Jennings, C. G., Smyth, W. F., A simple fast hybrid pattern-matching algorithm, Journal of Discrete Algorithms, Vol.5 No.4, 2007, pp.682-695.

[GG92] Galil, Z., Giancarlo, R., On the exact complexity of string matching: upper bounds, SIAM Journal on Computer, Vol. 21, No.3, 1992, pp. 407-437.

[H93] Hancart, C., On Simon's string searching algorithm, Information Processing Letters, Vol. 47, No. 2, 1993, pp. 95-99.

[KMP77] Knuth, D. E., Morris, J. H. and Pratt, V. R., Fast Pattern Matching in Strings, SIAM Journal on Computing, Vol. 6, No.2, 1977, pp. 323-350.

[L96] Luk, R. W. P., Chinese string searching using the KMP algorithm, Proceedings of the 16th conference on Computational linguistics, 1996, Pages 1111-1114.

[MP70] Morris, J. H. and Pratt, V. R., A linear pattern-matching algorithm, Technical Report 40, University of California, Berkeley, 1970.

[R89] Regnier, M., Knuth-Morris-Pratt algorithm: an analysis. In Proceedings of the 14th Symposium on Mathematical Foundations of Computer Science, 1989, pp. 431-444.
[S93] Simon, I., Simon, I., String Matching Algorithms and Automata, Proceedings of the Colloquium in Honor of Arto Salomaa on Results and Trends in Theoretical Computer Science, 1994, pp. 386.395.