# Application of a Modified Convolution Method to Exact String Matching

K.W. Liu[1], R.C.T. Lee[2] and C.H. Huang [3*]
[1, 2] Department of Computer Science, National Chi Nan University, Puli, Nantou Hsieh, Taiwan 545
[3] Department of Computer Science and Information Engineering, National Formosa University, 64,Wen-Hwa Road, Hu-wei, Yun-Lin, Taiwan 632.
* Corresponding author: chhuang@sunws.nfu.edu.tw

## Abstract

*The problem of exact string matching is to find all locations at which a query of length m matches a substring of a text of length n. In this paper, we first find out all relative suffixes of this query on the text, and we look backward to find out the corresponding prefixes of this query on the text. In order to get this effort, we make use of a wide window[R.1] whose size is equal to 2m-1. Logic operation in CPU is the main process for all calculations. We directly use the logic operations to speed up the matching. Because the query can be represented as the bit vector, we save the space complexity. We get the optimal solution for exact string matching.*

*Keywords:* convolution method; bit-vector; Wide Window approach; String matching

## 1. Introduction

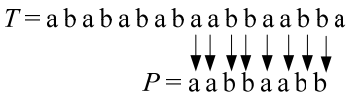
Many approaches use automaton[R.4] and suffix tree[R.5] to get the exact string matching. Using automaton and suffix tree methods have the advantage of theoretical explanation. But we need a complicated programming to appear the idea of automaton and suffix tree. In practice we can use the convolution method [R.2]to get the exact string matching. But the original convolution method is taken the disadvantage of big space and time complexity. In this paper, we use bits (1 and 0) and bit level operation (in this paper we use AND and SHIFT) to simulate as convolution method. Among all of the mathematical operation, the bit level operations are faster than any other operations in CPU. Good preprocessing of pattern makes the string matching speed up. More of the string matching algorithms need to make the complicated preprocessing of pattern. In this paper we make a simple preprocessing of pattern. We use bits (0, 1) to representing the pattern in preprocessing. Time complexity for our preprocessing is O($m$). Space complexity for our preprocessing is O($m$) bit.

The exact string matching problem is defined as follows: Given a text string $T = t_1t_2t_3...t_n$ and a pattern string $P = p_1p_2p_3...p_m$. The length of the pattern string is always smaller the length of the text string. We find all occurrences of the pattern string within the text string.

Example: Given: a text string $T$ and a pattern string $P$

$T$ = abababababaabbaabbababababa

$P$ = aabbaabb




Sliding window method is the very simple method to solve the exact string matching problem. See Figure 1.1 for an example.

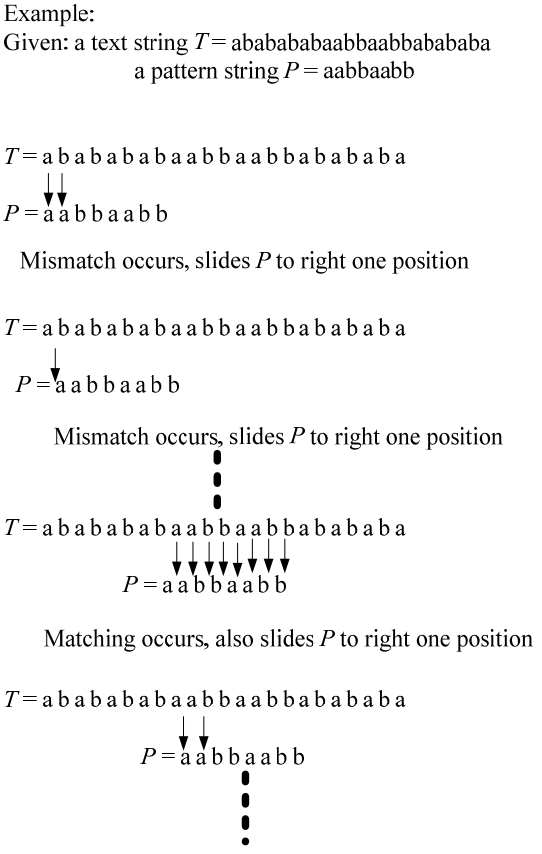



Figure 1.1

## 2. Basic Idea of the Wide-Window Approach

We open a window with size 2|*P*|-1 on the text string. We divide it into two parts, we denote the first one as $T_1$ and the second part as $T_2$. The length of $T_1$ is |*P*|-1. The length of $T_2$ is |*P*|.

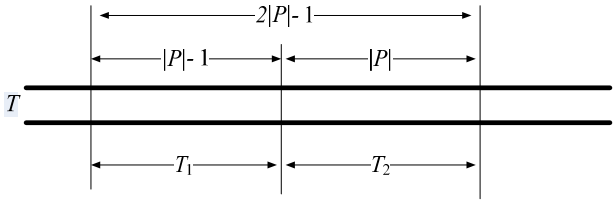



Figure 2.1

Since $|T_1|<|P|$, some suffix of *P* must be in $T_2$ if it exists. First we find all prefixes of $T_2$ which are also suffixes of *P*. We can be sure that one part of $T_2$ can be ignored as shown in flowing Figure.

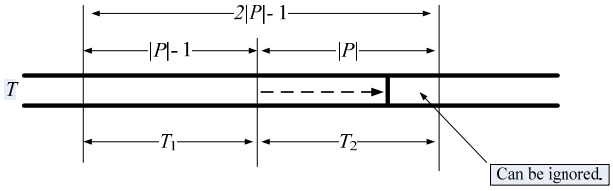



Figure 2.2

For every prefix of $T_2$ which is a suffix of *P*, we should find whether there exists a suffix in $T_1$ which is also a prefix of *P* as shown in Figure 2.3.

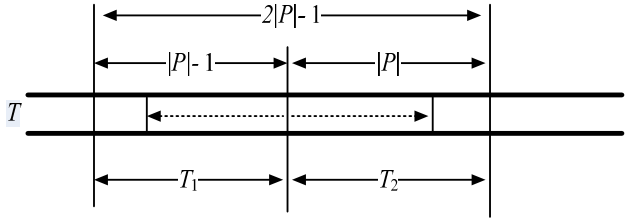



Figure 2.3

To simplify the description, we give the following definitions.

Definition 1. Let *n-suffix to be* the suffix of string *S* with length *n*, where $-1 < n \leq |S|$ .

Example: *S* = abcde
0-*suffix* of *S* = $\varepsilon$
1-*suffix* of *S* = e
2-*suffix* of *S* = de
3-*suffix* of *S* = cde
4-*suffix* of *S* = bcde

Definition 2. Let *n*-prefix to be the prefix of string S with length *n*, where $-1 < n \leq |S|$

Example: *S* = abcde
0-*prefix* of *S* = $\varepsilon$
1-*prefix* of *S* = a
2-*prefix* of *S* = ab
3-*prefix* of *S* = abc
4-*prefix* of *S* = abcd
5-*prefix* of *S* = abcde

See Figure 2.4 ~ Figure 2.5 for an example of the wide window approach.

Given: *T* = aababcbdcea
*P* = abcbd
Let us produce a wide window whose length is
|*P*|-1+|*P*| = 2|*P*| -1
In this case,
|*P*|=5
2|*P*| -1 = 9

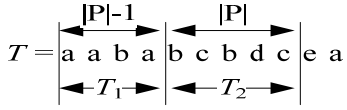



Figure 2.4

We first find all prefixes of $T_2$ which are equal to some suffixes of *P*.

In this case, we obtain bcbd whose length is 4. We found a 4-*suffix* of *P* is the 4-*prefix* of $T_2$.

4-*suffix* of *P* = bcbd
4-*prefix* of $T_1$ = bcbd
|*P*|-4 = 5 – 4 = 1

If the 1-*suffix* of $T_1$ is the 1-*prefix* of *P*, we have found a matching.

1-*suffix* of $T_1$ = a
1-*prefix* of *P* = a

∴ 1-*suffix* of $T_1$ = 1-*prefix* of *P*. Thus we conclude that a matching is found

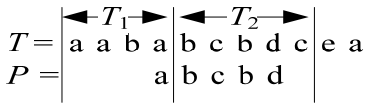



Figure 2.5

The next question is how to find all prefixes of $T_2$ which are equal to some prefixes *P*. The convolution method will work in this issue.

## 3. The Modified Convolution Method

We may use the convolution method to find all prefixes of $T_2$ which are equal to some suffixes of *P*. Consider Figure 3.1.

Given: $T_2$ = bcbdc , *P* = abcbd , $T_2^r$= cdbcd

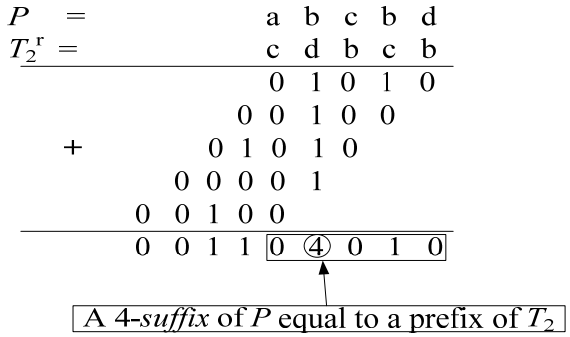



Figure 3.1

We check the numbers among the results. If the value is equal to its position, we conclude that a suffix of $P$ equal to a prefix of $T_2$.
The convolution method works, but the complexity is not good enough. With the convolution method, the time complexity is $O(n^2)$ and $O(mn)$ additional space. To reduce the overhead, we proposed a modified convolution method. As in Figure 3.2, to speed up, the "multiplication" and "addition" operations can be replaced with "shift" and "and" operations.

We may also use the logic operator (AND &) to find all prefixes of $T_2$ which are equal to some suffixes of $P$.

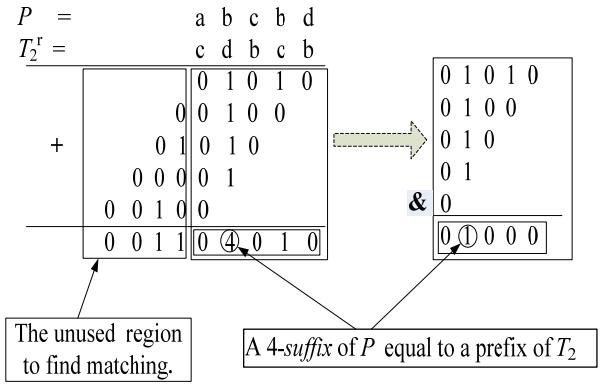



Figure 3.2

Similarly, we may use the modified convolution method to find all suffixes of $T_1$ which are equal to some prefixes of as Figure 3.3.

$T_1$ = aaba , $P$ = abcbd , $P^r$ =dbcda

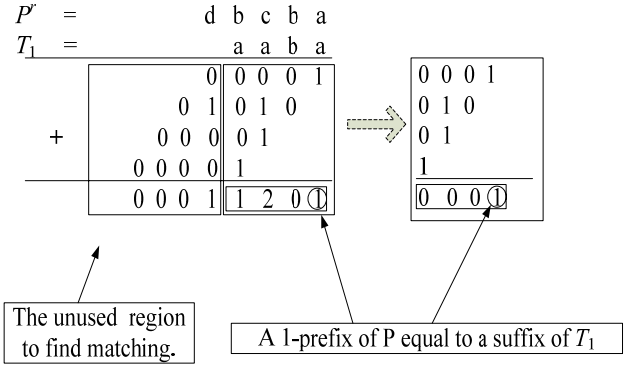



Figure 3.3

## 4. The Algorithm

Let us consider the following case:

T = bcbdc
P = abcbd

Our job is to determine whether there is a prefix in $T$ which is a suffix of $P$. Indeed, in this case, we have 4-prefix of T (bcbd) which is also the 4-suffix of P.

As indicated before, we may use modified convolution.

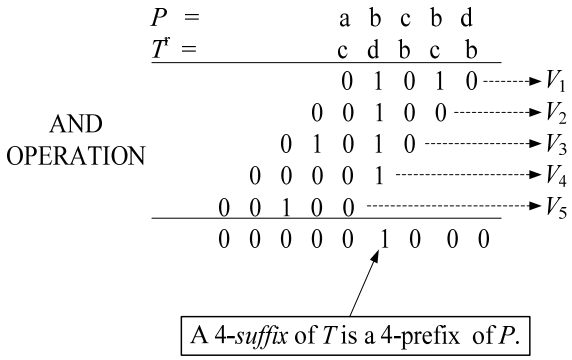



Figure 4.1

Definition 3. Given a string $S = s_1s_2\ldots s_n$ and a character $\alpha$, the *α-bit pattern* of $S$ is defined as $b_1b_2\ldots b_n$ where $b_i$=1 if $s_i = \alpha$ and $b_i$=0 if otherwise.
Taking $S$ = abcbd as an example:

*a-bit* pattern of $S$ = 1 0 0 0 0
*b-bit* pattern of $S$ = 0 1 0 1 0
*c-bit* pattern of $S$ = 0 0 1 0 0
*d-bit* pattern of $S$ = 0 0 0 0 1

We can now observe that

1. $V_1$ = *b-bit* pattern of $P$ as we are comparing $T$[1] = b with $P$,
2. $V_2$ = *c-bit* pattern of $P$ as we are comparing $T$[2] = c with $P$,
3. $V_3$ = *b-bit* pattern of $P$ as we are comparing $T$[3] = b with $P$,
4. $V_4$ = *d-bit* pattern of $P$ as we are comparing $T$[4] = d with $P$,
5. $V_5$ = *c-bit* pattern of $P$ as we are comparing $T$[5] = c with $P$.

We are ready to present our algorithm.

## The KRC Algorithm

**Input**: $T = t_1 t_2 \ldots t_m$ , $P = p_1 p_2 \ldots p_n$

**Output** : All occurrences of $P$ on $T$

**Preprocessing:**

Find the character set of $P$

Build the character_bit pattern of $P$

the character_*r*bit pattern of inversed $P$

**Searching:**

For each $k \in 1..\left\lfloor \frac{n}{m} \right\rfloor$ do

Open a wide window whose length is $2m$-1 and its center point is at $km$

Let the window be denoted as $a_1 a_2 \ldots a_{2m-1}$

Let $a_1 a_2 \ldots a_{m-1}$ be denoted as $T_1$

Let $a_m a_{m+1} \ldots a_{2m-1}$ be denoted as $T_2$

Find out all prefixes of $T_2$ which are the suffix of $P$ by using bit pattern approach.

For every prefix of $T_2$ which is a suffix of $P$, we should find whether there exists a suffix in $T_1$ which is also a prefix of $P$ by using bit pattern approach. If we found, we found a matching.

End For

## End

In the following, we give an example to explain our algorithm step by step.

Example: $T$ = aababcbdc

$P$ = abcbd

Let us produce a wide window whose length is

$|P| - 1 + |P| = 2|P| - 1$

In this case,

$|P|=5$ , $2|P| - 1 = 9$

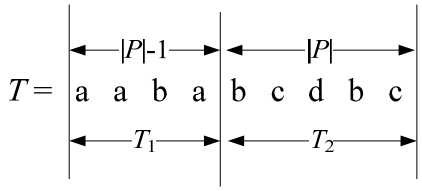



Figure 4.2

## Preprocessing step

Build character bit pattern of $P$

$P$ = abcbd

Find all bit patterns of $P$, $P$ is composed of a, b, c, b, d. The character set of $P$ = {a, b, c, b, d}

*a_bit* = 1 0 0 0 0

*b_bit* = 0 1 0 1 0

*c_bit* = 0 0 1 0 0

*d_bit* = 0 0 0 0 1

Having constructed the character bit pattern of $P$, we may use the character bit pattern of $P$ to find whether the prefix of $T_2$ is equal to the suffix of $P$.

Find all character bit patterns of reversed $P$, $P$ is composed of a, b, c, b, d.

*a_rbit* = 0 0 0 0 1

*b_rbit* = 0 1 0 1 0

*c_rbit* = 0 0 1 0 0

*d_rbit* = 1 0 0 0 0

Having constructed the character bit pattern of reversed $P$, we may use the character bit pattern of $P$ to find whether the suffix of $T_1$ is equal to the prefix of $P$ as shown in Figure 4.3 to Figure 4.7.

## Searching Step

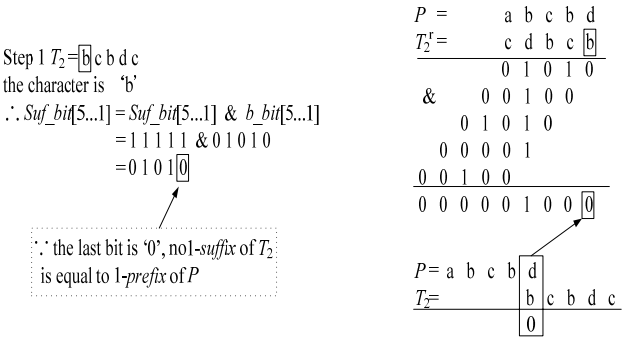



Figure 4.3

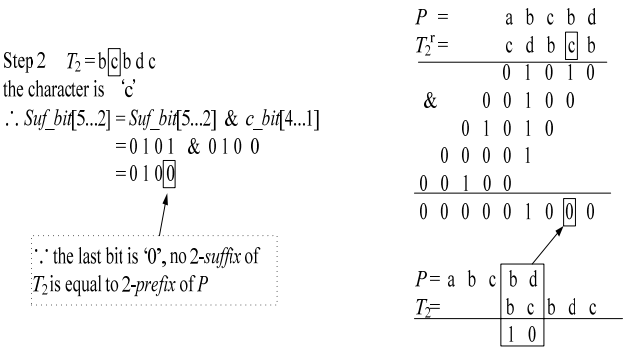



Figure 4.4

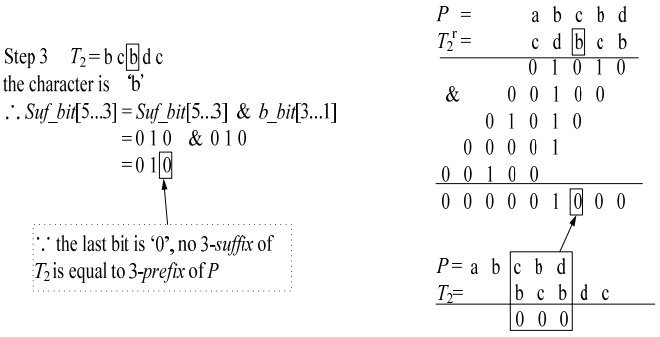



Figure 4.5

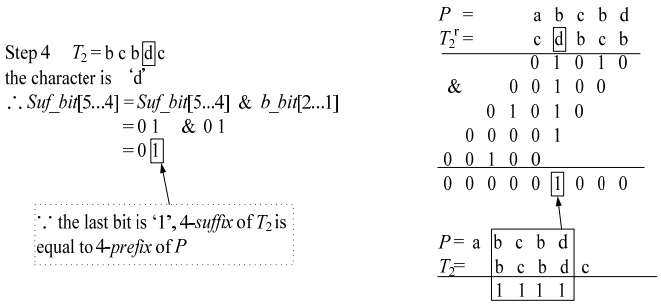



Figure 4.6

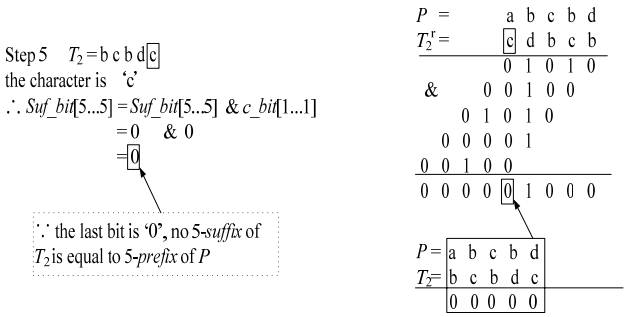



Figure 4.7

We have found one suffix which is 4-*suffix*. The corresponding prefix which we need to find is (|*P*|-4)-*prefix*. If we found, we got a matching.

Having constructed the character bit pattern of reversed *P*, we may use the character bit pattern of reversed *P* to find whether the suffix of $T_1$ is equal to the prefix of *P* as shown in Figure 4.8 to Figure 4.11.

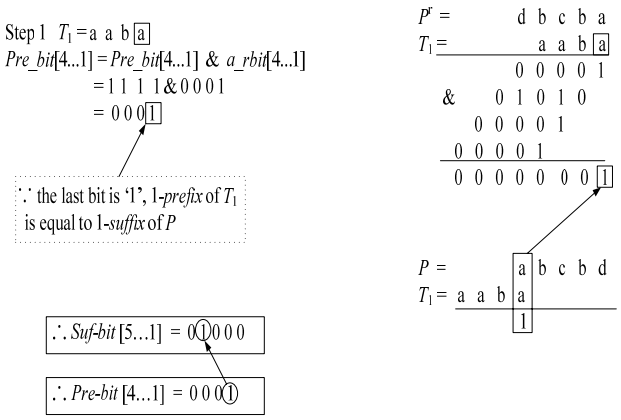



Figure 4.8

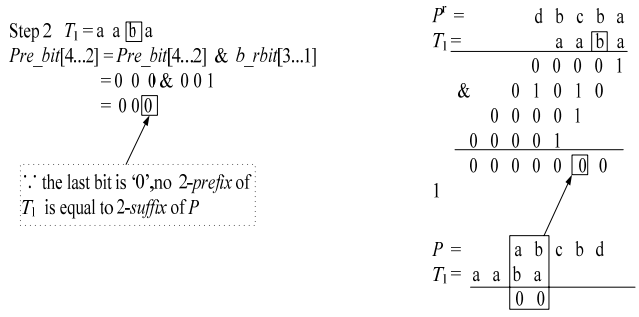



Figure 4.9

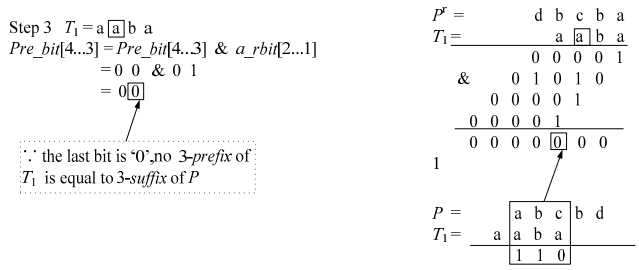



Figure 4.10

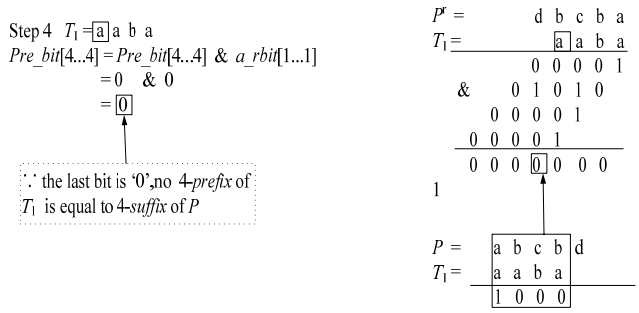



Figure 4.11

## 5. Analysis of Complexities

In the preprocessing, we make the bit pattern of pattern *P* and the bit pattern of reversed *P*. We represent *P* as the bit pattern whose length is |*P*|.

**Proposition 1.**. The space complexity for preprocessing is *O(m)* bits.

Preprocessing is linear time complexity, since the entire preprocessing just need to read *P* one time.

**Proposition 2.** The time complexity for preprocessing is *O(m)*.

The length of text string |*T*| is *n* and the length of pattern string |*P*| is *m*. Therefore we have *n*/*m* wide windows. For each wide window, we need 2*m* comparisons in worst case. So the total time needed is O(*n*) in worst case ( 2*m* × *n*/*m* = 2*n*).

**Proposition 3.** The time complexity for searching is *O(n)*.

To sum up, we have the following lemma.

**Lemma 1.** The KRC algorithm runs in *O(m)* time complexity with *O(n)* additional space.

**Experimental results**

We implemented our algorithm in C programming language. Obviously, according to our experiment, the total number of character comparison of KMP string matching algorithm is at least the length of the text string, almost twice; it is not rely on the length of pattern string. But KMP remembers the substring of text which have recently compared. Therefore we made our algorithm comparing with BM string matching algorithm. We used a lot of DNA sequences n our experiment. We compared our algorithm with BM; the total number of character comparison of our algorithm is less than the BM. This means that our algorithm is better BM and KMP methods. The following is the result of our experiment. The value of x-axis is the length of pattern string. The value of y-axis is the length of text string.

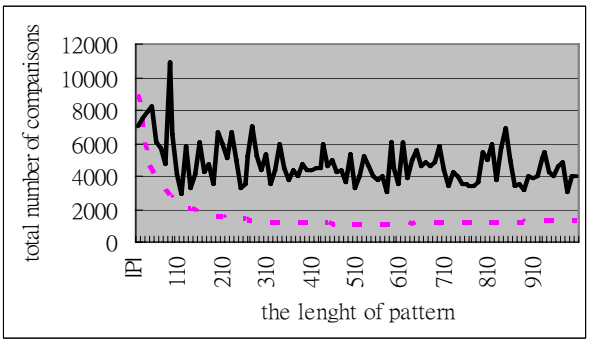



Figure experimental results

The solid line is the result of BM method. The dotted line is the results of our algorithm. Obviously our algorithm has the less character comparisons than BM method.

## 6. Summary

We use the bit level operation and our algorithm is very easily to be implemented. Time complexity of our algorithm is *O(n)*, so it is the optimal one in exact string algorithms. For further work, we will try multiple string matching, approximation string matching by using modified convolution method.

## 7. References


[R.1] Longtao He, Binxing Fang and Jie Sui. The wide window string matching algorithm. *Theoretical Computer Science, 2005*.

[R.2] B.H.Wu and R.C.T Lee. Convolution and its applications to sequence analysis*, 2004*.

[R.3] Gene Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. *Journal of the ACM, 1999.*

[R.4] Mark Nelson. Fast String Searching with suffix trees. *Dr. Dobb's Journal*, 1996.

[R.5] M. Crochemore, W. Rytter, *Jewels of Stringology*, World Scientific, Singapore, 2002.