A Multipattern Matching Algorithm Using Sampling and Bit Index

Jinhui Chen, Zhongfu Ye
Department of Automation
University of Science and Technology of China
Hefei, P.R.China
jeffcjh@mail.ustc.edu.cn, yezf@ustc.edu.cn

Min Tang
Department of Computer Science and Technology
Tsinghua University
Beijing, P.R.China
tangmin@csnet1.cs.tsinghua.edu.cn

Abstract

Pattern matching is one of the basic problems in computer science. In this paper we propose a new multiple pattern matching algorithm. Unlike the well known Knuth-Morris-Pratt, Boyer-Moore, Karp-Rabin and their variants, our algorithm is derived from the ideas of sampling and bit index, sampling for efficiency and bit index for flexibility, as a result providing the simplest way to search for multiple patterns. Theoretical analysis and experimental results show that our algorithm is average-optimal with average complexity of O(n/m) for the search of patterns of length m in a text of length n. It provides a proper solution to such needs as matching long dispersed patterns and especially bit pattern matching (newly introduced in this paper) in data analysis of some private protocols' communication.

1. Introduction

Pattern Matching (PM) has many applications in most fields of computer science. In this paper we present a new PM algorithm for matching multiple patterns which we argue has the best average runtime whereas is very simple and flexible.

The PM problem consists in finding all occurrences of patterns in text. Since any alphabet ∑ can be coded in binary, in this paper we view both of text and patterns as bit stream but stored in bytes as usually. Let $T = T[0…n-1]=T_{0…n-1}$ be the text of length n, $P=P[0…m-1]=P_{0…m-1}$ the pattern of length m, and $P^1… P^r$ be the r patterns of at least m bytes. An I-substring (i-bitstring) is the string of continuous I bytes (bits) in T or P. So-called bit pattern matching--a new term introduced in the paper--is the problem that P need not start at some byte of T, but may begin at any bit of any byte in T.

Classical PM algorithms have KMP [1], BM [2], KR [3], and their variants [4-11] with different matching policies, and in practice BM style algorithms are considered the most efficient. Most PM algorithms have the general structure [12] in Fig. 1.

PM algorithms mainly differ in the computation of shift increment according to matching policies. Generally, there are some deficiencies in the algorithms above, including: 1) the efficiency of algorithms depends heavily on the alphabet; 2) the speed advantage of matching long pattern is not evident; and 3) they are not suitable for bit pattern matching.

To solve these problems, we propose a new algorithm known as SBI (Sampling and Bit Indexing). Unlike KMP, BM and KR, SBI is derived from the simple ideas of sampling [13] and bit index. Theoretical analysis and experimental results show that SBI is average-optimal with average running time of O(n/m) in the condition that the total length of patterns is no more than tens of thousands.

```
Generic_Pattern_Matcher ( T, P )
        n ← length ( T )    m ← length ( P )
        Precompute ( P )
        s ← 0
        while s ≤ n – m do
              s ← s + Shift_Increment ( s, T, P )
where
```

✧ Precompute (P) derives some data structure usually in form of tables or lists, which are later accessed by Shift_Increment.

✧ Shift_Increment (s, T, P) checks for a match at s and then computes a shift increment from s.

Figure 1. General structure of PM algorithm

In section 2 we describe the basic type of SBI in details. Section 3 contains the proof of the correctness and the analysis of time complexity and space complexity. Section 4 makes discussions of three variants of basic SBI. Some experimental results are presented in section 5 to verify our algorithm. The last section is the conclusion and relative work for further research.

2. Basic SBI Algorithm

We need some specific terms used by SBI.

Definition 1: A full match means that P is identical to some m-substring of T, while a partial match means some I-substring of P is identical to some I-substring of T, where $I \in 1...m$ is the length of partial match.

Definition 2: *Sampling* is a positioning operation, moving forward a constant distance (also called *jump*) over T or P every time. The I-substirng or i-bitstring taken at the sampling point of T or P is known as index (abbr. idx later).

2.1. Basic Ideas of SBI

The idea of SBI is simple: if there is a full match between T and P, it must also be a partial match. By means of sampling T to obtain the i-bitstring as a possible partial match, we can extend a partial match to a full match. This idea is illustrated in Fig. 2, where three partial matches as indices lie at the beginning, in the middle, and at the tail of P, respectively.

To decide whether or not an i-bitstring is a partial match, we need sample P to gather all its i-bitstrings in advance. Hence, SBI algorithm has two stages:

✧ the first stage is preprocessing of P to store in table the information of all its i-bitstrings as indices which may be partial matches;

✧ the second stage is to sample T to obtain corresponding index, and with the help of preprocessing information extend partial matches to full matches if there are any.

Following are the two stages of basic SBI in details in the case of single pattern (i.e. r =1).

2.2. The Preprocessing Stage

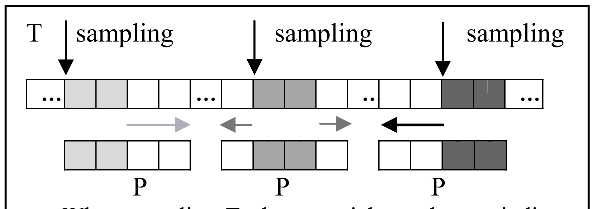
In this stage we preprocess P to create an array V of linked lists of index of P.

The first thing is to compute related parameters. According to n, m, and available memory, we can obtain three parameters: i--the length of index in bit, I--the length of index in byte, and J --the jump of sampling T in byte. The computation formula and related restrictions are as follows:

$$2^i = \rho \times n / m$$
$$I = (i + 7) / 8 \quad (1 \leq I \leq m / 2)$$
$$J = m - I + 1 \quad (r \times J \leq 2^i)$$

where n and m are the lengths of text and pattern respectively, i and I are the index width in bit and byte respectively, J is the const jump of sampling text, and ρ is called compression ratio ($0<\rho\leq1$).

For example, $n = 2^{28}$, $m = 2^5$, $r = 1$, we choose $\rho=1/2^7$, then $i = 16$, $I = 2$, $J = 31$. Note that i is not necessarily the multiple of 8. In practice we may choose i=8~24, and i=16~20 is usually a good choice.



When sampling T, three partial matches as indices are found. At these points, we try to extend partial matches forward and backward to possible full ones.

Figure 2. Ideas of SPM: extend partial match to full match

```
Function PP ( n, m, P )
1. compute i, I, J
2. allocate memory for V
3. for p = 0 to $2^i$-1 do  V[p] ← NULL
4. for p = 0 to J-1 do
5.     get i-bitstring of P as idx at byte p
6.     if V[idx] equals NULL then
7.         create linked list V[idx]
8.     insert node of offset p into V[idx]
```

Figure 3. Pseudo code of function PP

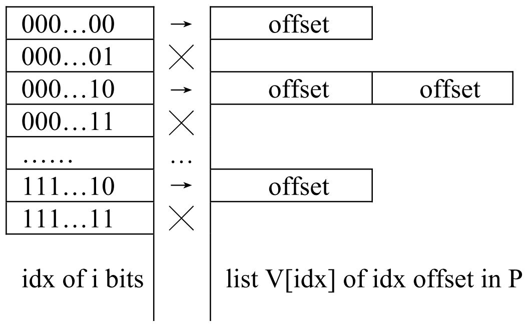


Figure 4. Structure of array V

Then what we need do is to create an array V of linked lists of index sampled from P. In other words, V is a pointer array of size 2^i, whose element is either NULL or a pointer to a linked list. Every node in the list V[idx] stores the offset in P of the index idx, and V[idx] consists of nodes with the same idx sampled with jump 1 byte at different point of P.

The preprocessing stage can be described by function PP (PreProcessing) in Fig. 3.

We illustrate the structure of the array V in Fig. 4.

2.3. The Sampling and Matching Stage

In this stage of sampling and matching, we take the following steps to find all possible full matches:

- ✧ sample T with jump J bytes to get index idx
- ✧ decide whether idx is valid, i.e. V[idx] = NULL ?
- ✧ if V[idx] != NULL, then extend every partial match recorded in the node of linked list V[idx] to full match if there is any.

This procedure can be described by the pseudo code of function SM (Sample and Match) in Fig. 5.

Complete SBI algorithm consists of two functions, i.e. PP and SM. Clearly, our simple algorithm uses the techniques of sampling and indexing to guarantee the efficiency (with constant jump J=m-I+1 every time to finish scanning T in n/J steps), and by means of bit index supports the flexibility of matching (such as single/multiple pattern matching, and bit pattern matching discussed in section 4).

To better understand the algorithm, we give an example of matching procedure. Given P = “pattern” and m=7, we choose i=16, I=2, J=m-I+1=6. The matching procedure of SBI algorithm is illustrated in Fig. 6.

3. Analysis of SBI Algorithm

In this section we analyze the correctness and performance of our algorithm SBI in the basic type. Further discussions of its three variants are made in Section 4.

3.1. The Correctness Analysis

To analyze the correctness, we first have the following lemma. We suppose index length (in bytes) $I \le m/2$ below.

Lemma 1: Suppose X is any m-substring of T, we can always get an I-substring of X as index when sampling T with jump $J \le m-I+1$.

```
Function SM ( T )
  // call function PP then execute the following
1. for t = 0 to n-1 step J do
2.     get i-bitstring as idx at byte t
3.     if V[idx] != NULL then
4.         while V[idx] not ends
5              get offset p from node of V[idx]
6.             compare $T_{t-p...t-p+m-1}$ against $P_{0...m-1}$
                 // or extend forward and backward from idx
7.             if match completely then report it
```

Figure 5. Pseudo code of function SM

Proof: Since jump $J \le m-I+1$ ($\le m$) when sampling T, there must be a sampling point j inside X, where $j \in 0...m-1$. There are two cases: 1) If $j \in 0...m-I$, then we directly take I-substring of X at j; 2) If $j \in m-I+1...m-1$, then there is another sampling point $j' = j+I-m-1 \in 0...I-2$, and we take I-substring of X at j'. Anyway, we can get an I-substring of X as index. □

Due to Lemma 1, we can prove the following sampling theorem which is critical to SPM algorithm.

Theorem 1: (Sampling Theorem) If jump (of sampling T) $J \le m-I+1$, then SPM algorithm never discards any full match.

Proof: Suppose there is a m-substring, X, in T to match P. By Lemma 1, we can take an I-substring of X as index idx at some sampling point. It is clear that idx also exists in P since V[idx] is not NULL. Based on idx (it is a partial match), we extend backward and forward in T and P, finally to succeed in finding a full match, which is in fact X (in T) and P. □

Since SPM samples T with jump J=m-I+1, theorem 1 ensures the correctness of our algorithm. It is not difficult to see that if J > m-I+1, then we might get no I-substring of X as index such that SPM algorithm fails. In actual implementation, we only take as index i-bitstring instead of I-substring, where I=(i+1)/8, to reduce the memory requirement.

It is worth pointing out that ALGO1 in [4] is in fact a special case of SBI when I=1.

3.2. The Performance Analysis

To analyze the performance, we prove the theorem below.

Theorem 2: The running time of PM algorithm has a lower bound of $\Omega(n/m)$.

Proof: We divide $T = T_{0...n-1}$ of length n into [n/m] contiguous and non-overlapping m-substrings, and disregard the few last text bytes that may not complete

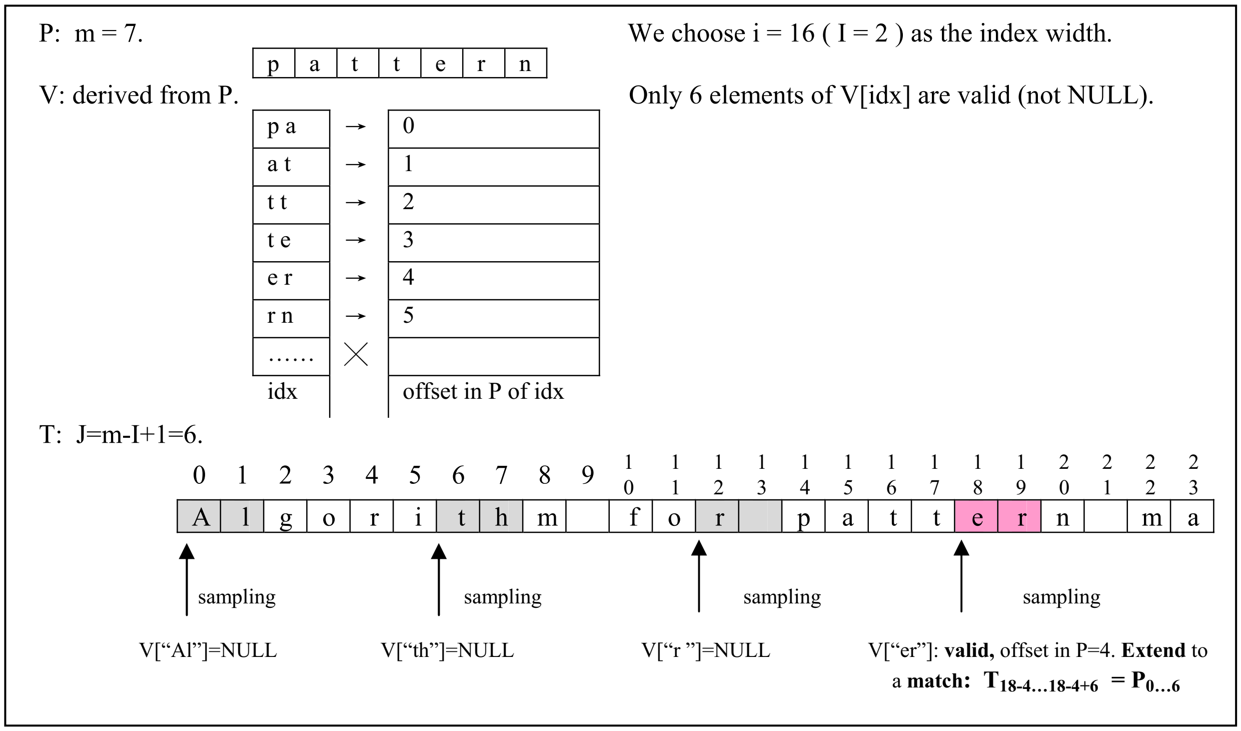


Figure 6. Eample of SPM matching procedure

a full m-substring. According to [15], the time of matching P in T is no less than the total time of matching P inside [n/m] m-substrings. Since these m-substrings are independent, one can not finish the whole work of matching in less than [n/m] "processes". Since every "process" spends no less than one time unit, i.e. $\Omega(1)$, the total time of matching is [n/m] $\times\Omega(1)= \Omega(n/m)$. □

To facilitate the analysis of running time of SPM, we need the following lemma.

Lemma 2: Suppose $X=X_{0...k}$ and $Y=Y_{0...k}$ are two strings of same length k+1, the event $X_i=Y_i$, $i\in 0...k$ has the same probability p, $0<p\leq 1/2$. Then the comparison times needed to decide whether or not X equals Y (i.e. X=Y) are at most 1/(1-p) on average.

Proof: Let q=1-p. Note that function $g(x) = xp^x$ has its peak value $g_p = -1/(e\ln p)$, so $g_{1/2} = 1/(e\ln 2) = 0.53$, $g_{1/256} = 1/(8e\ln 2) = 0.07$. Therefore we may omit the small value of $p^{k+1}(k+1)$ in the following computation. The average comparison times to decide whether or not X=Y are given by

$$\begin{aligned} f(p) &= q\times 1 + pq\times 2 + p^2q\times 3 + \dots p^k\times(k+1) \\ &= q\times 1 + pq\times 2 + \dots p^kq\times(k+1) + p^kp\times(k+1) \\ &\approx q\times(1+p^1\times 2+\dots +p^k\times(k+1)) \\ &\leq q\times(1+p^1\times 2+\dots+p^k\times(k+1) +\dots) \qquad (1) \\ &= q\times 1/q^2 = 1/q \\ &= 1(1-p). \end{aligned}$$

Here in (1) we reference the sum of well-known series $1+x^1\times 2+x^2\times 3+x^3\times 4+\dots=1/(1-x)^2$. □

In this paper, p=1/256 and 1/2 for byte and bit, respectively.

We now give the analysis of average running time of SBI algorithm, denoted by t(n, m), assuming that both T and P are random strings with uniform distribution.

The total time t(n, m) consists of $t(n,m)_p$--preprocessing time, and $t(n,m)_s$--sampling and matching time. According to function PP in Fig. 3, we have

$$t(n, m)_p = O(2^i + J) = O(n / m)$$

But $t(n, m)_s$ is much more complicated and needed to be investigated in detail. The line 2 of function SM in Fig. 4 (denoted by SM_Line2) spends constant time of c_1. The condition of SM_Line3 has a probability of $J/2^i$ to be true. If SM_Line3 is true, it is easy to see that the loop statement of SM_Line4 is run once on average because the average length of valid linked list in V is one. By Lemma 2, SM_Line5~7 spends constant time of c_2 on average. Therefore, the mathematical expectation of $t(n, m)_s$ is:

$$\begin{aligned} E(t(n, m)_s) &= n / J \times (c_1 + (J / 2^i) \times 1 \times c_2) \\ &= O(n / J) = O(n / m), \text{ if } J = O(2^i). \end{aligned}$$

Note that $J = O(2^i)$ is to assure $J/2^i = O(1)$.

Finally we get the average running time of basic SBI as follows:

$$\begin{aligned} E(t(n, m)) &= E(t(n, m)_p) + E(t(n, m)_s) \\ &= O(n / m) + O(n / m) \\ &= O(n / m), \text{ if } J = O(2^i). \end{aligned}$$

The formula of $t(n,m)_s$ also gives the possible worst-case runtime of SBI algorithm as follows:

$$\begin{aligned} t(n, m)_s &= n / J \times (c_1 + (1/1) \times J \times m) \\ &= O(n / J) + n \times m \\ &= O(n\, m). \end{aligned}$$

This is encountered when $T = a^n$ and $P = a^m$.

The best-case running time of SBI is clearly achieved when none of i-bitstrings of P occurs in T, and in this situation we have

$$\begin{aligned} t(n, m) &= n / J \times (c_1 + 0) \\ &= O(n / J) \\ &= O(n / m). \end{aligned}$$

It is necessary to point out the fact that SBI is average-optimal with average runtime O(n/m) dose not contradict the theoretical result of [15], which shows that based on the symbol comparison, the average complexity of multipattern matching is $\Omega(n\log_\sigma(rm)/m)$ accesses to T. By using the techniques of sampling and indexing, our algorithm SPM reduces the time of $\Omega(\log_\sigma(rm))$ spent on every block in [15] to O(1). This is in fact an application of usual time-space tradeoff, because SBI algorithm needs an extra memory space for the array V of linked list of index, whose size is $O(2^i + m)$.

4. Variants of SBI Algorithm

For the basic type of SBI algorithm in section 2, in this section we discuss its three variants to meet such needs as multiple patterns, bit pattern matching and less memory space.

4.1. Variant for Multiple Patterns

SBI can extend naturally to multiple patterns. Given r patterns, $P^1 \ldots P^r$, of lengths $m_1 \ldots m_r$, respectively, $m = \min\{m_k\}$.

The function PP in Fig.1 needs to be modified to PP2 in Fig.7. PP2 has two differences from PP: one is the addition of a loop of line 4 for multiple patterns; the other is the inserted node with more information of k and p.

The function SM in Fig.3 needs minor modification, including SM_Line5 (more information of k and p) and SM_Line6 (ompare T against P^k). New version SM2 is given in Fig. 8.

It is easy to see that average running time of SBI in the case of multiple patterns is

```
Function PP2 (n, m, r, P^1… P^r)
1. compute i, I, J
2. allocate memory for V
3. for  p = 0 to $2^i$-1  do V[p] ← NULL
4. for  k = 0  to  r-1  do
5.     for  p = 0  to  J-1  do
6.         get i-substring of $P^k$ as idx at byte p
7.         if V[idx] equals NULL then
8.             create linked list V[idx]
9.         insert node of k and p into V[idx]
```

Figure 7. Pseudo code of function PP2

```
Function SM2 ( T )
    // call function PP2 then execute the following
1. for  t = 0  to  n-1  step J  do
2.     get i-bitstring as idx at byte t
3.     if V[idx] != NULL then
4.         while V[idx] not ends
5              get k and p from node of V[idx]
6.             compare $T_{t-p...t-p+m-1}$ against $P^k_{0...m-1}$
                   // or extend forward and backward from idx
7.             if match completely then report it
```

Figure 8. Pseudo code of function SM2

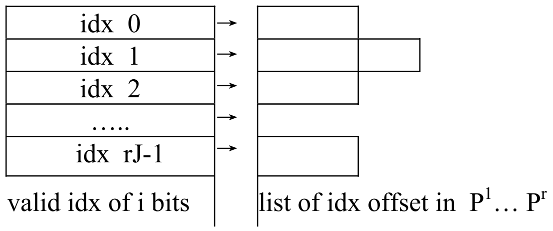


Figure 9. Structure of packed V

$$\begin{aligned} t(n, m) &= t(n, m)_p + t(n, m)_s \\ &= O(2^i + rJ) + n/J \times (c_1 + (rJ/2^i) \times 1 \times c_2) \\ &= O(n/m), \text{ if } rJ = O(2^i). \end{aligned}$$

That is to say, the average running time for multiple patterns is still O(n/m) in the reasonable condition of $rJ=O(2^i)$. In practice, we can choose the proper i such that $rJ=O(2^i)$ as follows:

if	rJ < 20000	then	let i = 16
else if	rJ < 60000	then	let i = 17
else if	rJ < 100000	then	let i = 18
else if	rJ < 300000	then	let i = 19
else			let i = 20

4.2. Variant for Bit Pattern Matching

This paper for the first time introduces the concept of bit pattern matching, which is not yet fully studied by other PM algorithms. SBI algorithm can meet the need of bit pattern matching with minor changes in function PP and SM as follows:

PP_Line1: J= m-I+1 →
J=m-I

PP_Line4: for p=0 to J-1 do →
for p=0 to 8J-1 do

PP_Line5: get i-bitstring of P as idx at byte p →
get i-bitstring of P as idx at bit p

SM_Line7: compare $T_{t-p\ldots t-p+m-1}$ and $P_{0\ldots m-1}$ →
compare T and P bitwisely from idx

The average running time of SBI in the case of bit pattern matching is

$$\begin{aligned} t(n, m) &= t(n, m)_p + t(n, m)_s \\ &= O(2^i + 8J) + n/J \times (c_1 + (8J/2^i)\times 1\times c_2) \\ &= O(n/m), \quad \text{if } 8J = O(2^i). \end{aligned}$$

Clearly it is still average-optimal.

In the data analysis of some private protocols such as banking key management protocol ISO8732 [14] and tactical data link in military communication, bit pattern matching is necessarily applied. Our algorithm presents a proper solution to such needs, while other PM algorithms so far are not suitable for bit pattern matching.

4.3. Variant for Less Memory Space

SBI needs extra memory space of size $O(2^i + rm)$ to store the array V. Since most elements of V are NULL, the memory requirement can be reduced to $O(rm + rm) = O(rm)$ by only storing the information of valid indices of r patterns, resulting in the packed type of V with the structure in Fig. 9, where all rJ indices are sorted. In this case it depends on the binary search to decide whether or not index of T is valid. The average runtime of this variant accordingly becomes $O(n\log_2(rm)/m)$, which is consistent with the theoretical result of [15].

5. Experimental Results

This section presents some experimental results of comparison between SBI, KMP, BM and AC-BM.

The test platform is Windows Server 2003 with configuration of CPU P4 3.20GHz and main memory 1GB. The texts used have two different sources, one being actual IP-based packets and the other being encrypted data. The patterns of different lengths are randomly generated in advance. All runtimes reported are averages of 10 different runs with no consideration of data loading time. The time unit is ms.

In the tables and figures, PLEN denotes pattern length, PNUM pattern number, SBI_bit SBI's variant for bit pattern matching, and * means the time used is too long to need comparison with others.

5.1. Average Runtime of Single Pattern

In the case of single pattern, basic SBI is used to compare against KMP and BM both of which are only suitable for single pattern matching.

The used text is actual IP packets of size 300MB. The actual runtimes of KMP, BM and SBI are shown in Table 1 with different pattern lengths from 4 to 40. Based on the same data in Table 1, the form of plotted curves is presented in Fig.10, where KMP/3 denotes the one-third of actual runtime of KMP because KMP consumes much more time than BM and SBI.

It is clear that SBI runs twice faster than BM on average.

5.2. Average Runtime of Multiple Patterns

In the case of multiple patterns, we compare the multiple pattern version of SBI with AC-BM.

The text for test is also actual IP-based packets of total size 300MB. The actual runtimes of SBI and AC-BM are given in Table 2 with two cases of PLEN=8 and PLEN=16. Part of data from Table 2 are plotted in Fig. 11, where "AC-BM m=16" and "SBI m=16" stand for the case of PLEN=16 for both algorithms, respectively.

From Fig.11, we see that the advantage of SBI over AC-BM is great.

5.3. Average Runtime of Bit Pattern

For the test of bit pattern matching, we still use the text of actual IP packets of size 300MB for different pattern length from 4 to 40 to compare SBI_bit against basic SBI. Table 3 lists their average runtimes, which are plotted in Fig. 12.

Fig. 12 shows that SBI_bit has the same average-case performance as basic SBI for patterns whose lengths are no less than 12, i.e. $m \geq 12$. When $m = 4, 8$, there are clear gaps between two plotted lines. The reason is that the jump J for basic SMP equals m-I+1, while the jump J for SPM_bit equals m-I, where I=2, such that the difference of the sampling times (i.e. n / (m-I+1) and n / (m-I), respectively) is not negligible when m is small.

Table 1. Average runtime of single pattern matching

PLEN	KMP	BM	SBI
4	1725	1073	674
8	1721	559	295
12	1668	385	209
16	1667	310	151
20	1665	262	125
24	1671	223	109
28	1678	203	98
32	1700	198	90
36	1679	176	84
40	1710	161	87

Table 2. Average runtime of multiple patterns

PNUM	SBI (PLEN =8)	SBI (PLEN =16)	AC_BM (PLEN =8)	AC_BM (PLEN =16)
1	302	151	776	421
10	301	156	903	567
20	301	159	1126	728
40	307	165	1473	1142
80	318	179	2573	2229
160	348	208	6319	4557
320	382	246	*	*
640	459	312	*	*
1000	536	395	*	*
2000	773	500	*	*

Table 3. Average runtime of SBI and SBI_bit

PLEN	basic SBI	SBI_bit
4	657	926
8	295	350
12	204	228
16	148	159
20	120	131
24	109	117
28	98	101
32	93	92
36	82	85
40	79	84

Table 4. Average runtime of basic SBI for different length of text and different numbers of patterns

PNUM	n=600MB	n=400MB	n=200MB
1	600	398	201
10	604	404	203
20	612	409	204
40	625	418	207
80	646	435	217
160	701	468	232
320	798	529	265
640	965	639	318
1000	1150	757	378
2000	1653	1100	543

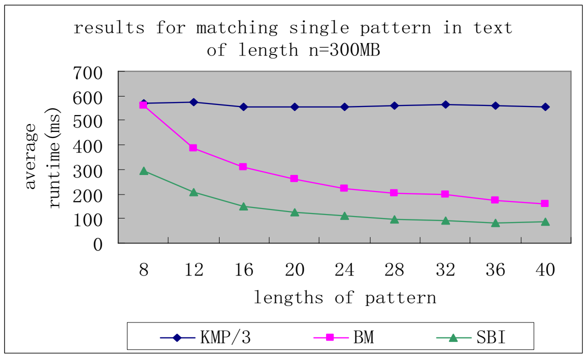


Figure 10. Runtimes of KMP, BM and SBI for single pattern

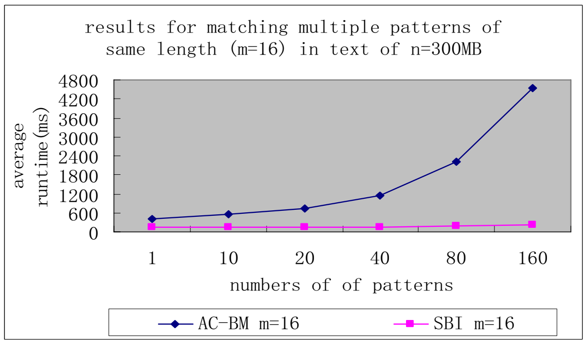


Figure 11. Runtimes of SBI and AC-BM for mutiple patterns

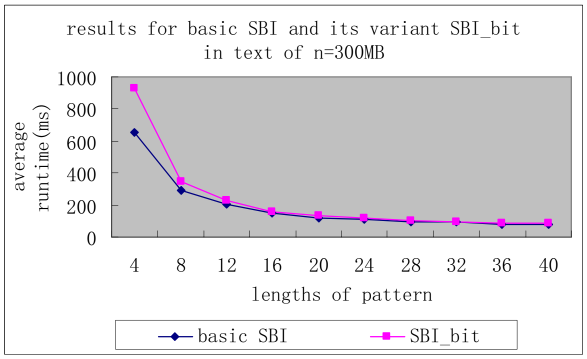


Figure 12. Runtimes of basic SBI and SBI_bit

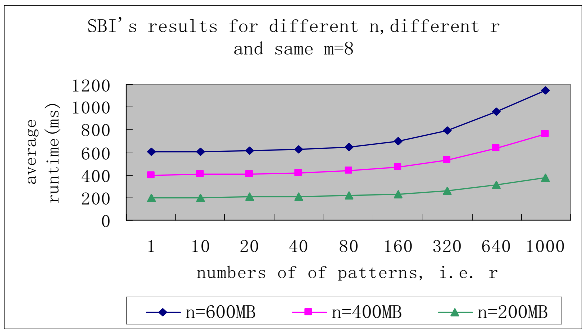


Figure 13. Runtimes of basic SBI for different n, r and same m

5.4. Runtime of SPM for Different n and m

In this test, we use the encrypted communication data of different length n as text. Table 4 presents the runtimes of basic SPM compared against itself, finding patterns of different numbers (r=1~2000) and same length (m=8) in text of different length (n=600/400/200 MB). Fig.13 gives plotted curves.

Table 4 and Fig.13 again verify the conclusion that SPM is average-optimal when the number of patterns is not very large.

From the experimental results of four tests above, we find out that SPM operates at a much higher efficiency than BM style algorithms, and that the average runtime of SPM matches very well to the result of our theoretical analysis, i.e. O(n/(m-I+1)) = O(n/m), where I=2 or 3 in practice, in the condition that the total length of patterns is no more than tens of thousands bytes.

6. Conclusion

This paper has proposed a simple, efficient PM algorithm known as SPM and its three variants. Derived from the simple ideas of sampling and bit index, SPM achieves the average-optimal performance and great flexibility.

Due to array V, SPM can support addition and deletion of some patterns even in the execution procedure. In the case of small alphabet ∑, if we code ∑ in binary (bit) in advance when storing the text and pattern, then we can use SPM_bit to finish the pattern matching. SPM can also be easily parallelized to meet the need for distributed computation.

We recommend SPM when the patterns are long and dispersed because of the fastest speed in this case. The drawback of SPM is the increased memory space required for array V, while this is generally affordable at present. Another deficiency is its worst runtime O(nm), which is worth further study to improve it.

7. References

[1] D. E. Knuth, J. H. Morris, and V. R. Pratt, ‘‘Fast pattern in strings, ” *SIAM J.Comput.*, vol. 6, pp.323-350, June 1977.

[2] R. S. Boyer and J. S. Moore, “A fast string searching algorithm,” *Commun. ACM*, vol. 20, pp. 762-772, Oct. 1977.

[3] R.M. Karp and M.O. Rabin, “Efficient randomized pattern-matching algorithms,” *IBM Journal of Research and Development*, vol. 31, pp. 249 – 260, 1987.

[4] A.V. Aho and M.J. Corasick, “Efficient string matching: an aid to bibliographic search,” *Commun. ACM*, vol.18, pp.333-340, 1975.

[5] R.N. Horspool, “Practical fast searching in string,” *Software – Practice and Experience*, vol.10, pp.501-506, 1980.

[6] D.M. Sunday, “A very fast substring search algorithm,” *Commun. ACM*, vol. 33, pp.132-142, 1990.

[7] S. Wu and U. Manber, “A Fast Algorithm for Multi-Pattern Searching,” *Technical Report TR 94-17*, Department of Computer Science, University of Arizona, May 1994.

[8] K. Sun, “A New String-Pattern Matching Algorithm Using Partitioning and Hashing Efficiently,” *ACM Journal of Experimental Algorithmics*, March 1998.

[9] FENG CAO, “PAMA: A Fast String Matching Algorithm and Its Application in DNA Sequence Search,” Thesis Submitted to the Graduate School of Wayne State University, Detroit, Michigan in partial fulfillment of the requirements for the degree of master of sicience, 2004.

[10] Charras, C., Lecroq, T., and Pehoushek, J. D. “A Very Fast String Matching Algorithm for Small Alphabets and Long Patterns,” *Proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching*, M. Farach-Colton ed., Piscataway, New Jersey, Lecture Notes in Computer Science 1448, pp.55-64, Springer-Verlag, 1998.

[11] S. S. Sheik, Sumit K. Aggarwal, Anindya Poddar, N. Balakrishnan, and K. Sekar, “A Fast Pattern Matching Algorithm, ” *J. Chem. Inf. Comput. Sci.* vol.44, pp. 1251-1256, 2004.

[12] D. Cantone and S. Faro, “Fast-Search: A new efficient variant of the Boyer-Moore string matching algorithm,” *Proc. of the 2nd Int’l Workshop on Experimental and Efficient Algorithms*. Lecture Notes in Computer Science 2647, Heidelberg: Springer-Verlag, pp.47−58, 2003.

[13] U. Vishkin, “Deterministic Sampling -- A New Technique for Fast Pattern Matching,” *SIAM J. Comput.,* 20, pp.22-40, 1991.

[14] “Banking - Key management (wholesale)”, ISO8732, Geneva (1988).

[15] G. Navarro and K. Fredriksson, “Average Complexity of Exact and Approximate Multiple String Matching,” *Theoretical Computer Science*, pp.283-290, 2004.