Finding a Haystack in Haystacks – Simultaneous Identification of Concepts in Large Bio-Medical Corpora*

Ying Liu‡†, Lucian V. Lita†, R. Stefan Niculescu†, Prasenjit Mitra‡, C. Lee Giles‡
College of IST‡
The Pennsylvania State University
University Park, PA 16802
{yliu,pmitra,giles}@ist.psu.edu

Siemens Medical Solutions†
51 Valley Stream Parkway
Malven, PA 19335
{lucian.lita,stefan.niculescu}@siemens.com

Abstract

Since nearly all information is now created digitally, large text databases have become more prevalent than ever. Automatically mining information from these databases proves to be a challenge due to slow pattern/string matching techniques. In this paper we introduce a new, fast multi-string pattern matching method called the Block Suffix Shifting (BSS) algorithm, which is based on the well known Aho-Chorasick algorithm. The advantages of our algorithm include: the ability to exploit the natural structure of text, perform significant character shifting, avoid useless backtracking jumps, efficient matching time and avoid the typical "sub-string" false positive errors. Our algorithm is applicable to many fields with free text, such as the health care domain and the scientific document field. In this paper, we apply the BSS algorithm to health care data and mine hundreds of thousands of medical concepts from a large Electronic Medical Record (EMR) corpora simultaneously and efficiently. Experimental results show the superiority of our algorithm when compared with the top of the line multi-string matching algorithms (the Aho-Corasick and the Wu-Manber algorithm).

1 Introduction

Recent advances in computer hardware allow ever-increasing data storage capacity. For example, news agencies have built their own massive news archives, hospitals have their own patient databases and scientists collect a slew of genomics and proteomics data. While storing this data in large databases is now a relatively easy task, efficiently extracting and processing information from these databases is more than a challenge.

Recently, we see a government-coordinated push towards migrating hospital paper charts into electronic medical records (EMRs)[5], which are relatively large - sometimes hundreds of GBs of data - text based medical databases, covering thousands of patient records, where a full patient record can have a size of few tens of megabytes. Typically, a patient record contains data in structured format, including demographics, lab results, medications, as well as text notes. These include admission notes, history and physical profiles, operation reports, progress notes and discharge summaries. Recently, Medicare introduced the Quality Measures initiative, asking hospitals to provide a record on how well they treated patients with several prevalent conditions, including Acute Myocardial Infarction, Heart Failure, Surgical Infection and Pneumonia. Currently, hospitals employ a significant number of people who manually go through the paper charts, trying to answer questions relevant to these quality measures by tracking specific text patterns. To address this issue, several companies have built rule-based expert systems. One of the main challenges faced by these systems is large performing efficient pattern matching on the EMRs. Motivated by this challenge, we propose a fast, simultaneous algorithm to match a large set of patterns in a large text corpora. The algorithm seeks to identify all (possibly overlapping) instances of all items in the pattern set efficiently and to improve upon previous solutions.

Aho-Corasick (AC)[2], Commentz-Walter (CW)[3], and Wu-Manber (WM)[11] are representative multiple-string matching algorithms. In practice, they can seldom apply an important *shift* operation especially when there are numerous patterns and the text is large. The *shift* operation[8] refers to skipping several characters in the text without having to check against the pattern-derived data structure specific to an algorithm. The basic reason for this is that they work at the character level and treat both patterns and text as character

*This work is supported by Siemens Medical Solutions and NSF grant 0535656.

†Partial work done during Siemens internship.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

sequences without considering their natural structure.

In many pattern-matching applications, different characters can be treated differently. Due to the natural structure of text and patterns, we can view them as a set of *blocks* and *separators* between the *blocks*: Each *block* is composed of a variable size set of *characters*. In these constrained tasks, patterns may only start from the beginning of a *block*. When the AC algorithm is used to deal with the constrained multiple-pattern matching problem, will spend unnecessary matching time evaluating characters that are not allowed to start a pattern instance. The evaluation involves several comparisons and pattern data structure traversals that are not needed. Moreover, the AC algorithm spends additional time analyzing scenarios that are easily detectable as pattern instances that would violate the block constraints. To avoid these time-consuming explorations and to speed up the overall matching process, we introduce a new multi-pattern matching algorithm that integrates shifting functionality and is better suited for block constrained tasks.

Among the tasks that follow a natural *block/separator* structure, we focus on a frequently encountered task in the healthcare field. In particular, we focus on matching medical concepts in free text data such as: patient medical records and medical articles. Given patient medical records, several applications are needed to identify and extract informative features, process contextual information, and locate relevant concepts.

In this paper, we present a new online exact multi-pattern matching algorithm to search for multiple patterns simultaneously, the Block Suffix Shifting (*BSS*) algorithm. The BSS algorithm implements shifting functionality on segments of text while maintaining an exact match solution. It is faster than previous character-level algorithms and is scalable to a very large number of patterns. The BSS algorithm handles patterns and text with long blocks well, which is very often the case in the medical domain. We show that such a method can lead to efficient pattern matching performance, especially for free-text environments. Moreover, we present experiments with this new method in a medical domain setting and show improvements on real data and real applications. Besides efficiency, the method is also very flexible, allowing efficient solutions in different settings. The pattern data structure needs to be constructed only once and can be reused for multiple texts, i.e. multiple hospital patient record databases, medical articles, etc. The BSS algorithm has the following advantages:

- exploits the natural structure of text;
- allows significant character shifting;
- avoids backtracking jumps that are not useful;
- has efficient, overall less matching time;
- avoids the typical "sub-string" false positive errors;
- has multilingual applicability, such as alphabetic languages and ideographical languages.

In this paper, Section 2 discusses the related work. Section 3 elaborates the detailed state machine construction and the matching process of the BSS algorithm. Section 4 analyzes the algorithm complexity. Section 5 shows the improvements of the BSS algorithm comparing with the AC algorithm with an example. We analyze the experimental results in Section 6 and conclude in Section 7.

2 Related Work

Aho-Corasick (AC)[2], Commentz-Walter (CW)[3], and Wu-Manber (WM)[11] are representative multiple-string matching algorithms. They are widely used in several notable application areas including text processing, speech recognition, information retrieval, network security, and computational biology. The AC algorithm serves as the basis for the UNIX tool *fgrep*[1] while the WM algorithm is used in *glimpse* [9]. However, these algorithms seldom apply a *shift* operation in practice for numerous patterns and large text.

The AC algorithm is a linear-time algorithm based on an automata approach. It combines automata with an extension of the Knuth-Morris-Pratt (KMP) algorithm[6] by a method that includes an automaton based approach using suffix tree-like links. The AC algorithm constructs a state machine using the pattern set, successively reading individual characters in the text and concomitantly traversing the state machine through predefined goto and failure functions, and occasionally emitting outputs.

The AC algorithm scans one character at a time from text T and compares it with the current state in the pattern-built state machine. The reason for the impossibility of a shift is that the AC algorithm works at the character level: it treats both patterns and text as character sequences without considering their natural structure. The underlying alphabet is considered to be some small set of characters, such as ASCII or a DNA alphabet. Each edge of the AC state machine is labeled with a character. This provides a natural way to apply the AC algorithm and the small alphabet size can take advantage of the simple *byte operations* implemented at the machine level. As part of the matching process, the AC algorithm checks each character successively, that all the characters in the text are treated equally and

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

the assumption is that pattern instances can start from any character in the text. If we encounter a mismatch of a character c_i in the text T, AC still needs to check the next character c_{i+1} since it can start a new pattern instance, regardless of the location of c_{i+1}.

The Commentz-Walter (CW) algorithm combines the Boyer-Moore (BM) [4] method with the Aho-Corasick algorithm. The CW algorithm is only faster than the AC algorithm on small pattern sets and long minimum pattern lengths.

The Wu-Manber(WM) algorithm presents a different approach that is also based on the idea of the Boyer-Moore algorithm. The WM algorithm only uses the *bad-character shift* of the Boyer-Moore algorithm and considers the characters in the text to be separated in blocks of size B instead of one by one. In order to preserve the size of the alphabet, B cannot be large. In practice, they use either B=2 or B=3. As we increase the number of patterns, each block in the text will have multiple patterns that match it and the performance of the WM algorithm is heavily constricted by the length of the shortest pattern.

Although The WM algorithm claims a capability to shift due to its derivation from the Boyer-Moore algorithm, with increasing pattern size the possibility that the current block in the text matches some patterns increases dramatically and the shift does not happen (shift value=0). For the WM algorithm, the shift value was zero 5% of the time for 100 patterns, 27% for 1000 patterns, and 53% for 5000 patterns [11]; Moreover, when a block of size B in the text matches with the suffixes of some patterns, the WM algorithm has to check these patterns one by one and the advantage of shifting can not be shown.

3 Block Suffix Shift (BSS) algorithm

Formally, the multi-pattern-matching problem can be presented as follows:

- Let $\sum$ be a fixed alphabet, $|\sum| = \sigma$ whose elements we will refer to as *characters* – in practice very often the elements of σ are actual characters.
- Let $P = \{p_1, p_2, ..., p_k\}$ be a finite set of patterns, which are arbitrary strings of characters from $\sum$.
- Let $m = \sum_{i=1}^{k} |p|$. We assume that any two patterns may partially overlap. We define m as the total size of the pattern set P.
- Let $T = t_1, t_2, ..., t_n$ be a large text, again consisting of characters from the same alphabet $\sum$. n refers to the total size of the text T.

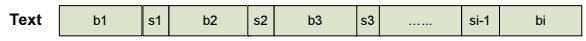


Figure 1: Text in the block/seperator property.

- The goal is to locate and identify all occurrences of all the patterns of P in T. The matched substrings in T may also overlap with one another.

3.1 Approach The BSS algorithm is derived from the AC algorithm because the AC algorithm is very robust and works well for large text and many patterns. The most important improvement is that the BSS algorithm shifts over the text T on many segments to heavily reduce the matching time without missing any result.

We propose this idea based on an interesting observation: both the text and the patterns can be viewed as a sequence of *blocks (b_1, b_2, ...)* and the *separators* between the *blocks (s_1, s_2, ...)* (see Figure 1). We call it the *"block/separator"* property. The blocks can be in variable lengths and the separators can be in different ways. This property is applicable to many applications. In this paper we focus on an medical domain task: efficient massive parallel medical concept extraction from large patient record corpora.

Instead of the *character* sequences, we view the text and patterns as *block/separator*-structured data. According to the nature of the free text, each *block* is a *word*. The *separators* can be space characters, punctuation as well as user-specified text, and even null for the equal-length blocks, which are not relevant to the task at hand. In order to identify *blocks*, the document tokenization may need to be performed. There are two main tokenization categories: orthography-oriented tokenization and dictionary-based tokenization[7]. Dictionary-based tokenization technique uses a dictionary to look for possible word strings/ combinations. It is normally used for tokenizing ideographical languages such as Chinese, Japanese, and Korean. Through tokenization we can detect the boundary of each word using orthographical clues, such as space,punctuation marks, etc. In this paper we focus on searching patterns in the English free-text and use orthography-oriented tokenization techniques. We define the *separator* as *space* character. In the preprocessing stage we remove the punctuation as well as the redundant spaces.

The free text structure imposes an important constraint: *Any pattern instance may only start with the beginning of a block*. In another sentence, it is impossible to start a new pattern in the middle of a *block*.

For the example in Section *2*, if c_{i+1} is not the first

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

character of a word w_j, there is no need to compare c_{i+1} with the pattern data structure. Moreover, the constraint provides the opportunity to shift over all the entire suffix of w_j (from c_{i+1} to the last character of w_j) and start a new matching iteration from the beginning of the next word w_{j+1}. This novel view of the data can speed up the entire pattern matching process for a more efficient performance. The BSS algorithm enables to skip over many segments in the text T without missing any matches. For the free-text, the skipped segments are the suffixes of all the mismatched words. Most often, the probability to mismatch is heavily larger than the probability to match, and therefore the amount of such segments in T is large. Besides being fast, the method is also very flexible, allowing efficient solution for many text variation as long as both the text and pattern share the *block/separator* property.

Similar to the AC algorithm, the behavior of the BSS algorithm is dictated by three functions: a goto function g, a failure function f, and an output function *output*. Both *failure* function and *goto* function map a state into a state. *Failure* function is enacted whenever the *goto* function reports *fail*. The *output* function associates the *patterns* (possibly empty) with every *state*. Comparing with the AC algorithm, the BSS algorithm makes important modifications in the state machine construction and the matching stage.

3.2 Goto Function At beginning, the BSS algorithm determines the states and the *goto* function. In order to take advantage of the simple *byte operations*, the BSS algorithm determines the states and constructs the *goto* functions at the character level using the exactly same method as that of the AC algorithm. We begin with the *goto* graph consisting of one vertex that represents the root state: state *0*. We then get a pattern p_i from P and add it into the graph by inserting a directed path that begins at the state *0*. Initially, *i=1*. In this directed path, each edge is labeled with a character and there is a new state node between every two edges. The pattern p_i is added to the output function of the state at which the path terminates. From the root state to the end of this directed path, we can spell out the pattern p_i. In the second round, we do the same thing to the next pattern p_{i+1}. The BSS algorithm adds new edges to the graph only when necessary (no existing edges to share).

The Figure2 shows the process of the goto function construction for the pattern set P in Example 1:

- p_1=*"painless"*;
- p_2=*"pain of eye"*;
- p_3=*"acute pain"*;

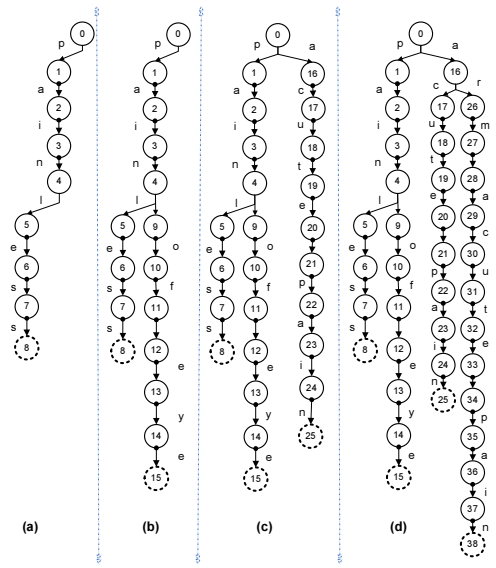


Figure 2: The process of the goto function construction for the Example 1 based on the BSS algorithm

- p_4=*"arm acute pain"*.
- T =*"experience acute pain of ear together stomach acute pain after painkiller currently feel painless"*

The graph is a rooted directed tree. To complete the construction of the goto function, we add a loop from state 0 to state 0 on all input other than p or a. This pattern matching machine consists of thirty-nine states totally. Each state is represented by a number from *0* to *38*. State 0 is designated as the start state.

3.3 Failure Function Defining and specifying failure functions is a critical part of the BSS algorithm. Before elaborating on the algorithmic details, we show reasons for the low performance of the AC algorithm, which does not work well for the medical application. We present several examples, analyze algorithm performance, and identify desirable modifications.

3.3.1 When should we shift? Consider Example 2: *p1='eel', T='heel eye'*.

According to the AC algorithm (Figure 3a), we have an occurrence of *p1* in the *T='h**eel** eye'*. However, it is not a real occurrence. *'eel'* is an independent word in *p1*. If we find a match in T, the matched *'eel'* should also be an independent word, instead of a substring of a word. For these cases, the BSS algorithm makes the first modification:

- Modification 1: for each mismatch (e.g., $h \neq e$), if there is no path to continue in the state machine,

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

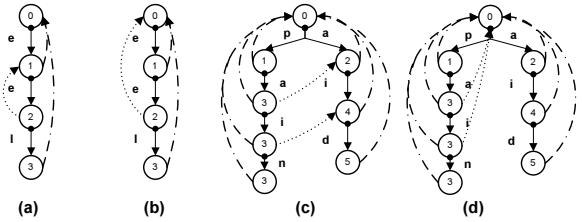


Figure 3: Sample state machines generated by the AC algorithm (a,c) and the BSS algorithm (b,d)

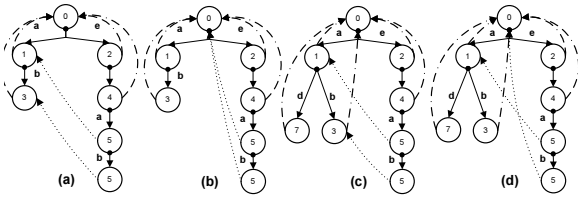


Figure 4: Examples of the redundant failure functions of the AC algorithm (a, c) and the modification of the BSS algorithm (b, d)

the failure function of the current state points to the root state ($f(s_0, h) = s_0$). BSS then aborts the matching process for the current word (*'heel'*) in T, shifts the reminder word suffix (*'eel'*), and jumps to the beginning of the next block/word (*'eye'*).

3.3.2 Failure functions connecting characters with different character index number (CIN) Consider Example 3: *p1='eel', T='ear eeel'*.

AC will generate another "substring matching" result for *Example 3* (Figure 3a): *T='ear e**eel**'*. For the first word *'ear'* in T, there is a mismatch ($a \neq e$) and $f(s_1) = s_0$. We stop the matching process for the word *'ear'*, skip the reminder word suffix (*'r'*) and jump to the beginning of the next word. For the second word *'eeel'*, the failure functions of AC in *Figure 3a* ($f(s_2) = s_1$) breaks the block boundary constraint: if a pattern matches a section of the text, every matched character pair from both the pattern and the text sections should have the same corresponding character index number *(CIN)*. However, in *Figure 3a*, the *CINs* connected by the failure function are *1* and *2 (1≠2)*. *Figure 3c* and *Figure 3d* show such failure functions between two patterns *'pain'* and *'aid'* with both AC and BSS.

In order to avoid such mismatches, the BSS algorithm makes the second modification based on the AC algorithm (see *Figure 3b and Figure 3d*):

- Modification 2: the BSS algorithm deletes such failure functions and reset them to the root state s_0 directly.

3.3.3 The redundant failure functions and matchings Consider Example 4: *p1="ab", p2="e ab", T="e at"*.

For those failure functions that connect two characters with the same *CINs* in the AC algorithm, should we keep all of them? The answer is no because we notice that some of them generate useless backward jumps and waste time on worthless matching operations without possibility to get any result. The BSS algorithm can avoid such wastes and jump to the next word earlier.

According to the Aho-Corasick algorithm, Example 4 has two failure functions between *p1* and *p2* (see *Figure4*). The matching process of the T over the trie is as following: $goto(s_0, e) = s_2, goto(s_2, \text{“ ”}) = s_4, goto(s_4, a) = s_5$. Because $t \neq b, f(s_5) = s_1$; Because $t \neq b, f(s_1) = s_0$; Because $t \neq e$ and $t \neq a, f(s_0) = s_0$. We stop at the root state s_0. Neither *p1* nor *p2* is found.

Although the AC algorithm does not generate any wrong result, there are redundant backward jumping and matching actions by doing the $r \neq b$ judgement twice: when we fail at s_5 with the input *"t"* over the goto function *"b"*, it is unnecessary to jump to s_1 because the only goto function of s_1 is also "b". Using the Aho-Corasick algorithm, we do two useless operations: $f(s_5) = s_1$ and $t \neq b$. If we do not fix this problem, with the increasing of the size of T, suppose we fail at s_5 $10,000$ times, this will waste a lot of time on $10,000 \times 2 = 20,000$ useless operations. Suppose s_1 has another child besides b (see Figure4c), the BSS algorithm should keep the failure function $f(s_5) = s_1$ (see Figure4d) because the new child does not equal to b, if we fail at s_5, we may can continue at the new child. To deal with such problems, our BSS algorithm does the following change:

- Modification 3: $\forall s_\alpha, s_\beta, f(s_\alpha) = s_\beta$, if s_β has different *goto* function as that of s_α, and both of them have the same CINs, BSS keeps this failure function (see Figure4d).

3.3.4 The propagated failure functions Using Example 1 we see in Figure 5 as the state machine generated by the AC algorithm.

Based on above examples, can we make the following statement? For any failure function in the AC algorithm $f(s_\alpha) = s_\beta$, if both s_α and s_β have the same *CINs* and the same goto functions, should we reset this failure function of s_α to the root state $f(s_\alpha) = s_0$, in order to reduce the redundant jumping and match? The answer is no because we should consider the possible match through the propagated failure functions.

The propagated failure function is defined such that

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

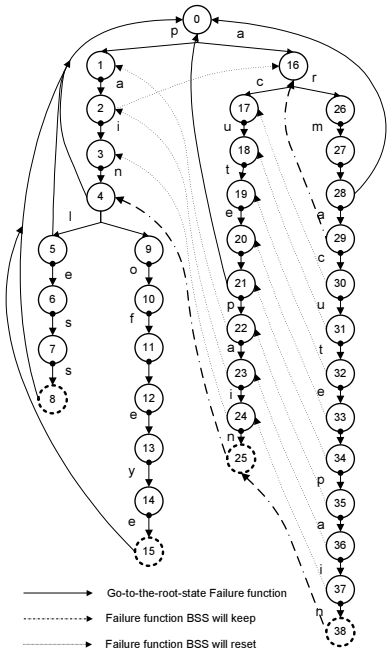


Figure 5: Aho-Corasick State machine and goto function for Example 1

there are a set of non-zero states $S = s_1, s_2, ..., s_i, i \geq 3$, which satisfy the condition: $f(s_i) = s_{i-1}, f(s_{i-1}) = s_{i-2}, ..., f(s_2) = s_1$.

In Figure 5, there are four triples that construct the propagated failure functions: (s_1, s_{22}, s_{35}), (s_2, s_{23}, s_{36}), (s_3, s_{24}, s_{37}), and (s_4, s_{25}, s_{38}). For the last triple, although s_{25} and s_{38} have the same goto function *(null)*, we keep the failure function $f(s_{38}) = s_{25}$, because $f(s_{25}) = s_4$ enables us to reach s_4 and s_4 has different goto functions: $goto(s_4, "l") = s_5$, and $goto(s_4, "") = s_9$. Suppose *T="arm acute painless"*, we can arrive at s_8 to get an output along the propagated failure functions from s_{38} to s_{25} to s_4.

In order to find all the possible matches, the BSS algorithm do the following modification comparing with the AC algorithm:

- Modification 4: $\forall s_\alpha$ and $s_\beta, f(s_\alpha) = s_\beta$, both of them have the same CINs. If s_β has the same *goto* function as that of s_α, but $f(s_\beta) \neq s0$, the BSS algorithm keeps this failure function $f(s_\alpha) = s_\beta$.

3.3.5 Setting failure functions in the BSS algorithm For all the failure functions of the AC algorithm, we classify them into two groups: *go-to-the-root-state* failure functions, and *do-not-go-to-the-root-state* failure functions. The BSS algorithm modifies a subset of the second group and leaves the first group intact.

For each state s_α ($s_\alpha \neq s_0$), according to the AC algorithm, $f(s_\alpha) = s_\beta$. If $s_\beta = s_0$, we allow this failure function or else we will have to decide whether allow it or reset it.

For a better understanding, we summarize the failure function settings of the BSS algorithm using the following concrete modification and adaptation of the AC algorithm. We only keep the failure functions that satisfy the condition:

1. s_α and s_β have the same character index numbers *(CINs)*; and
2. s_β has different *goto* function that s_α does not have, OR although s_β and s_α have the same *goto* functions, the failure function of s_β i is not the root state s_0.

For all the others, the BSS algorithm resets to the root state s_0. If we arrive at s_0 through a failure function, it follows that a mismatch occurred in the text, allowing the BSS algorithm to shift. The pseudocode of the BSS trie construction is shown in Algorithm 1.

Algorithm 1: BSS Trie Construction Pseudo Code

```
begin
    P = (p_1, p_2, ..., p_k), p = (a_1, a_2, ..., a_m), m∈ ∑;
    /*AC-Construct-GOTO()*/;
    for each p_k ∈ P do
        for c_i ∈ p; i=1,..,m do
            trie.add(c_i);
    /*BSS-build-NFA()*/;
    for each a∈ ∑ such that (s=goto(0,a))≠0 do
        s→Queue; fail(s)←0;
    while (r←Queue)≠ ∅ do
        for each a∈ ∑ such that (s=goto(r,a))≠fail do
            s→Queue; state←fail(r);
            while (next=goto(state,a))=fail do
                state←fail(state);
            /*BSS reset fail function*/;
            if next≠0 and next.cin=s.cin then
                if ∀a∈ ∑, goto(next,a)=goto(s,a) and
                fail(next)=0 then
                    fail(s)←0;
                else
                    fail(s)←next;
            else
                fail(s)←0;
    /*AC-Convert-NFA-DFA()*/;
end
```

3.4 Output Function The BSS algorithm has a similar output function to that of the AC algorithm: we associate the output $p1, p2, p3, p4$ with state *2, 15, 25, 38* respectively in *Figure 5*. The path from state 0

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

to each output state defines a pattern: the path from state 0 to state 8 spells out the pattern 1, the path from state 0 to state 15 spells out the pattern 2, the path from state 0 to state 25 spells out the pattern 3, and the path from state 0 to state 38 spells out the pattern 4.

3.5 The Matching Stage – Shifts After the BSS algorithm constructs the three functions based on the pattern set, the matching stage is ready to begin. All the shifts occur in this stage. The larger the average length of the words is, the more mismatches there are, the earlier the mismatches happen, the more segments in the text T the BSS algorithm can shift, and the more performance improvement the BSS algorithm can make, thus increasing the practical performance gap when compared to the AC algorithm.

For the text T, the BSS algorithm traverses it character by character to take advantage of the simple byte operation of the computer for the efficient performance. For each input character, the BSS algorithm checks the goto function of the current state: if *fail*, it follows the failure function to the new state. If the new state is the root state s_0, the BSS algorithm skips the reminder suffix of the current *word*, shifts to the beginning of the next *word*, and restarts the matching process from s_0. The *go-to-the-root-state* failure function means that no possible pattern in P can match with the current *word* in T. In other words, when the failure function goes to s_0, the shift value is greater than zero. Otherwise, the shift value is zero. The value reflects the length of the suffix to be shifted.

4 Complexity Analysis

In this section, we analyze the time complexity of our algorithm. As the BSS algorithm encounters mismatches, the advantage of the algorithm becomes apparent. In a practical setting of a multi-pattern matching problem, generally the probability for a mismatch is considerably larger than the probability for a match, especially when the text is large.

4.1 Best Case Scenario The best case scenario for the BSS algorithm occurs when the blocks are of equal length t and when mismatches are encountered on the first character in each block. Thus, the time complexity is $O(m+ (n/t) + z)$, as compared with $0(m + n + z)$ of AC, where z is the number of pattern occurrences in T. Although the complexity is still linear, since most of the time $n \gg m$ and $n \gg z$, the matching time is decided by n – which is the target of the BSS algorithm. Moreover, because of the equal-length blocks, it becomes trivial to skip entire blocks without identifying separators, yielding a considerable speedup.

4.2 Worst Case Scenario The AC algorithm can be viewed as the worst case scenario of the BSS algorithm since the former has to check and match every character in the text. For the BSS algorithm, this scenario appears in the following two cases. However, in free-text contexts, due to word diversity, these boundary cases are very infrequent.

(1)There is no mismatched block in the text. Every block in the text belongs to at least one pattern. In this case, the BSS algorithm must traverse the entire text and evaluate every character in the same fashion as the AC algorithm;

(2)If a mismatch exists, the mismatch happens in the last character of a block/word.

Table 1: The failure function of the Aho-Corasick algorithm for Example 1

state s	0	1	2	3	4	5	6	7	8	9
f(s)	0	0	**16**	0	0	0	0	0	0	0
state s	10	11	12	13	14	15	16	17	18	19
f(s)	0	0	0	0	0	0	0	0	0	0
state s	20	21	22	23	24	25	26	27	28	29
f(s)	0	0	**1**	**2**	**3**	4	0	0	0	16
state s	30	31	32	33	34	35	36	37	38	
f(s)	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	25	

Table 2: The goto function and the output function of the Aho-Corasick algorithm for Example 1

input	e	x	p	e	r	i	e	n	c	e	
state	0	0	1	0	0	0	0	0	0	0	0
output											
input	a	c	u	t	e		p	a	i	n	
state	16	17	18	19	20	21	22	23	24	25	9
output										**p3**	
input	o	f		e	a	r		t	o	g	e
state	10	11	12	13	16	0	0	0	0	0	0
output											
input	t	h	e	r		s	t	o	m	a	c
state	0	0	0	0	0	0	0	0	0	16	17
output											
input	h		a	c	u	t	e		p	a	i
state	0	0	16	17	18	19	20	21	22	23	24
output											
input	n		a	f	t	e	r		p	a	i
state	25	9	16	0	0	0	0	0	1	2	3
output	**p3**										
input	n	k	i	l	l	e	r		c	u	r
state	4	0	0	0	0	0	0	0	0	0	0
output											
input	r	e	n	t	l	y		f	e	e	l
state	0	0	0	0	0	0	0	0	0	0	0
output											
input		p	a	i	n	l	e	s	s		
state	0	1	2	3	4	5	6	7	8		
output									**p1**		

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

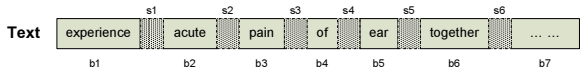


Figure 6: The text T of the Example 1 in the Block and Separator View

5 Example

In this section, we use Example 1 to explore the difference between the BSS algorithm and the AC algorithm.

In Example 1, the alphabet $\sum$={*a, c, e, f, h, i, l, m, n, o, p, r, s, t, u, x, y*} and σ=17. Both patterns and text are processed as a sequence of characters, see Figure 6.

5.1 Match With the Aho-Corasick Algorithm

Table 1 shows the *failure* function of the example as well as Table 2 shows the goto *function* and the *output* function. Both *failure* and *goto* function map a state into a state. *Failure* function is enacted whenever the *goto* function reports *fail*. The *output* function associates the *patterns* (possibly empty) with every *state*.

Initially, the current state of the machine is the start state *state 0*. The first input character in T is *"e"*. Because *"e"* is neither *"p"* nor *"a"*, we consult the failure function in *Table 2* and the new current state is still the state *0*. According to the Aho-Corasick algorithm, we should continue the operating cycle on the next character *"x"* because it is possible to match with a pattern that starts with the character *"x"*. It fails again and the new input character is "p". This time the *goto* function works (see *Table 2*), we arrive at *state 1*. The next input character "e" makes us fail again and return to the state *0*. The matching process continues like this until we finish the last input character in T. Along the whole matching process, we found two occurrences for $p3$ and one occurrence for $p1$.

5.2 Match With the BSS Algorithm

Still use the Example 1, we show the BSS pattern matching machine (trie) in Figure7. Table 3 shows the updated *failure* function based on the BSS algorithm while Table 4 displays the corresponding *goto* function and *output* function.

In the text T, the underlined characters are the segments the BSS algorithm can shift in the matching stage: T ="*experience acute pain of ear together stomach acute pain after painkiller currently feel painless*"

Comparing with the Aho-Corasick algorithm, the BSS algorithm skips 51/97 = 52.577% of the text T.

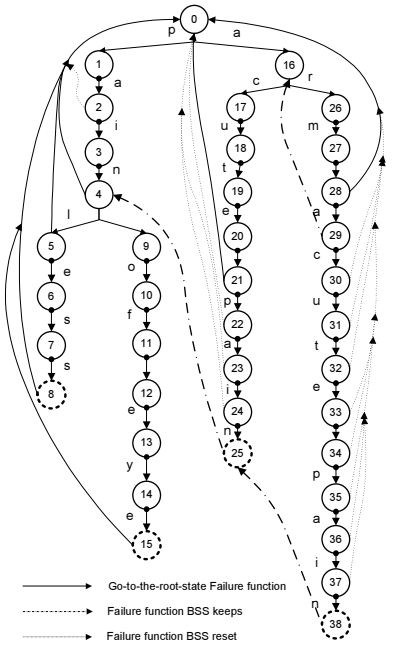


Figure 7: BSS algorithm state machine and goto function

Comparing the Table 2 with the Table 4, we notice that for the same patterns and text, the BSS algorithm only need to walk through the trie states in 36 times while the Aho-Corasick algorithm has to do it in 97 times.

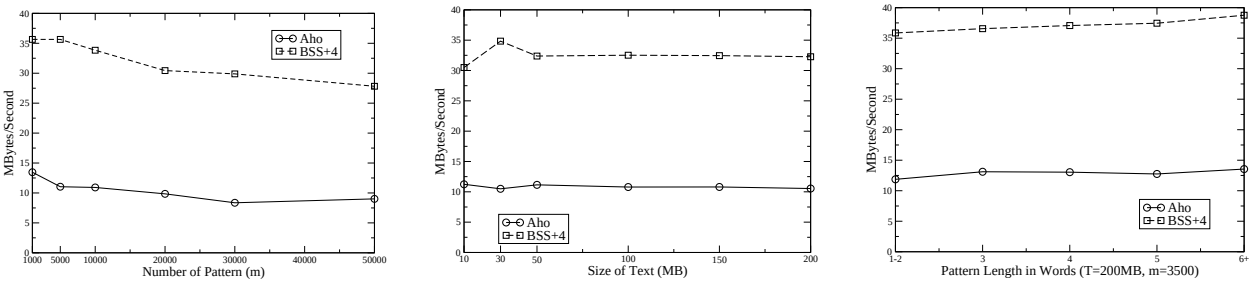
Table 3: The updated failure function of the BSS algorithm for Example 1

state s	0	1	2	3	4	5	6	7	8	9
f(s)	0	0	**0**	0	0	0	0	0	0	0
state s	10	11	12	13	14	15	16	17	18	19
f(s)	0	0	0	0	0	0	0	0	0	0
state s	20	21	22	23	24	25	26	27	28	29
f(s)	0	0	**0**	**0**	**0**	4	0	0	0	16
state s	30	31	32	33	34	35	36	37	38	
f(s)	**0**	**0**	**0**	**0**	**0**	**0**	**0**	**0**	25	

6 Experimental results

In this section, we present experiments comparing the BSS algorithm with two famous multi-pattern matching algorithms (the AC algorithm and the WM algorithm) on real data. In addition to compare the matching performance in terms of speed, we also investigate the impact of algorithm in terms of the following parameters: the size of the pattern set $P - m$, the size of the text $T - n$, the diversity of the pattern length, the average mismatch location in the blocks, the diversity of the block length in T, the size of the shortest pattern, the alphabet size σ, and modification steps of the BSS algorithm. We are interested in quantifying the average

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited



(a) (b) (c)

Figure 8: Text scanning speed of the BSS algorithm as a function of the size of the pattern set m, the size of the text m, and the pattern length

Table 4: The updated goto function and the output function of the BSS algorithm for Example 1

input	e	a	c	u	t	e		p	a	i
state	0	16	17	18	19	20	21	22	23	24
output										
input	n		o	f		e	a	t	s	a
state	25	9	10	11	12	13	16	0	0	16
output	**p3**									
input	c	u	t	e		p	a	i	n	
state	17	18	19	20	21	22	23	24	25	9
output									**p3**	
input	a	f	p	a	i	n	k	c	f	p
state	16	0	1	2	3	4	0	0	0	1
output										
input	a	i	n	l	e	s	s			
state	2	3	4	5	6	7	8			
output							**p1**			

text scanning speed (MB/s) on the test data and the average number of the shifted characters per block. For each experimental setting, we randomly select text and pattern set from the test data, and we perform twelve separate runs and averages are given.

6.1 Test environment All the experiments were conducted on a computer with Intel Core Due 1.83GHz CPU, 1 Gigebytes of RAM. The operation system is Windows XP. The classic AC and WM algorithms adopt the code presented in SNORT[1][10] and are implemented in JAVA. We implement the BSS algorithm by modifying the AC algorithm.

6.2 Test data In order to investigate performance of each algorithm as well as its advantages and disadvantages, our experiments were performed on test data with two different alphabet sizes: the English free text and the alphabet of size $|\sum|=\sigma=4$.

English alphabet: The English free texts come from two sources: the electronic medical records (EMR) and the scientific chemistry papers (Royal Chemistry Society[2]). The size of the alphabet σ is 96 (all visible characters). For the EMR data, the total text size is above 640MB. For the overall medical text, we use a database of $71,140$ patterns that fall under three categories: (i) signs and symptoms, (ii) diseases, and (iii) clinical drugs. The total size of the second data source is 500 MB and we randomly select $50,000$ chemical names to construct the pattern set.

DNA alphabet(σ=4): All test data comes from the DNA data set[3] in *gcg* format. The alphabet is $\sum = \{A, C, G, T\}$. The total size of the text is 250MB and the the total number of patterns we used is $50,000$.

6.3 Effect of m on the Match Performance *Figure 8(a)* shows the results of the performance of the BSS algorithm vs. the AC algorithm on the free text EMR, against six different sizes of the pattern sets: $m = \{1000, 5000, 10000, 20000, 30000, 50000\}$. For each set, we randomly select the patterns with diverse lengths. We apply the algorithms on the same text T $(50MB)$ for each pattern set. On the real medical data, the BSS algorithm maintains a high performance over AC for every m: the improvement rates of the BSS algorithm are 100%, 65.8%, 40.3%, 40.1%, 40.2%, and 35.1% respectively.

We notice that the BSS algorithm has a performance-decreasing trend comparing with the stable curve of AC, along the growth of the number of patterns. This trend is not surprising in that m increases, the number of states in the finite state machine increases, which provide more chances for T to traverse. The ratio of the matched segments in T will be increased and less block suffixes the BSS algorithm

[1] http://www.nersc.gov/

[2] http://www.rsc.org/

[3] ftp://genome-ftp.stanford.edu/pub/yeast/

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

can shift. When m reaches a very large value that all the blocks in T match with P, it results in a worst case scenario for the BSS algorithm and its performance curve will meet with the curve of AC in the figure.

6.4 Effect of n on the Match Performance We performed similar experiments in order to investigate the effect of n on the run time. The six value of n are: $10M$, $30M$, $60M$, $100M$, $200M$, and $300M$. *Figure 8(b)* shows the graph of the run time of the BSS algorithm versus the run time of AC with the same pattern size $m = 10000$. The results confirm that the performance improvement of the BSS algorithm is steady as the increasing of the text size n.

6.5 Effect of the Pattern Length on the Match Performance In order to investigate the distribution of the pattern length and its effect on the match performance, we further divide the $71,140$ patterns from the Unified Medical Language System (UMLS) into four groups according to the pattern length in term of words: one-word patterns (6,176), two-word patterns (16,866), three-word patterns (21,221), and four-and-above-word patterns (26,877). This distribution shows that in the health care field, there are many multiple-word patterns, and a considerable number of medical concepts is four words or more (e.g. "pain in the lower right leg").

To study the effect of pattern length on matching performance, we randomly select 1000 patterns from each group and apply both the BSS algorithm and the AC algorithm on the same text ($50MB$). The results are shown in *Figure 8(c)*. For all the four groups covering both short and long pattern lengths, the performance gain for the BSS algorithm over AC is 32.25%, 35.56%, 20.99%, and 28.99% respectively.

Although the patterns in other fields may not contain multiple words, they share a similar property: many patterns are long in terms of the number of characters. Figure 9 shows detailed statistics on the pattern length of the $50,000$ patterns from the chemistry domain.

6.6 Effect of the Block Size on the Match Performance We believe that the larger the average block size is, on average, the more characters in T the BSS algorithm can shift and the more significant the performance improvement that the BSS algorithm can achieve. In order to test our conjecture, we investigate the distribution of the block length first (see Figure 10) based a randomly selected $350MB$ scientific chemistry papers.

We divided all the 34.54 millions of blocks in the T into four categories according to their block size:

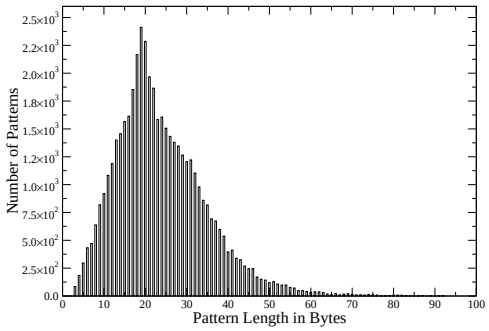


Figure 9: The Distribution of pattern length

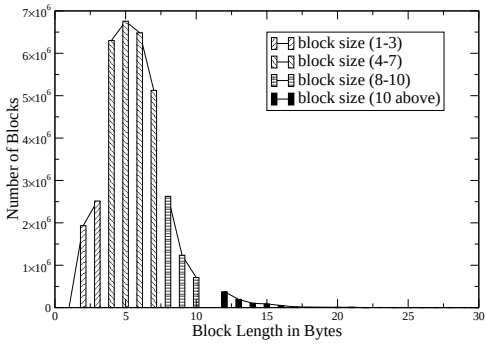


Figure 10: Block size distribution over text

between $1-3$, between $4-7$, between $8-10$, and above 10. The block size is defined as the number of the characters in a block. We randomly select blocks from each category to construct the test data. Within the same testing environment, we evaluate the improvement of the BSS algorithm by calculating how many characters in T are shifted by the BSS algorithm. Figure 11 shows both the average block length of each category and the average number of the shifted characters the BSS algorithm achieves. The fifth column represents the results for the DNA data with equal block length (10). In addition to figure 11, detailed experimental setting and results are listed in Table 5.

Table 5: The average block length and the character number shifted by BSS for each block

	1-3	4-7	8-10	> 10
Size of P - m	4500	4500	4500	4500
Size of T – n (MB)	50	50	50	50
Average block length in T	2.5	5.4	8.6	12.3
Chars/block BSS shifts	0.19	1.6	3.9	5.8

Our experimental result confirms our conjecture that as the increasing of the block length, the BSS algorithm makes more performance improvement comparing AC: for the four categories, the BSS algorithm shifts

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

7.6%, 29.3%, 45.3%, and 47.1% respectively over the whole text. We notice that the BSS algorithm has a performance-increasing trend, along the growth of the average block size in T (see Figure 12). It is not surprising because as the increasing of the block size, with the same text size m, we have less blocks. With the same mismatch possibility, there are much larger segments in T that the BSS algorithm shifts.

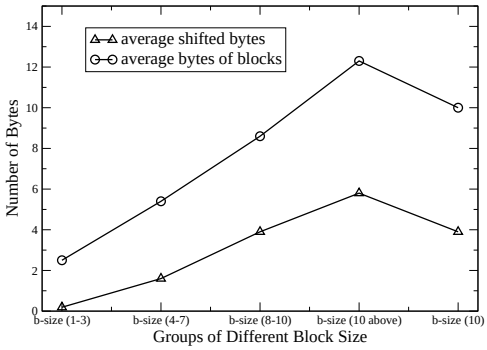


Figure 11: Average character shift of the BSS algorithm as a function of the block size

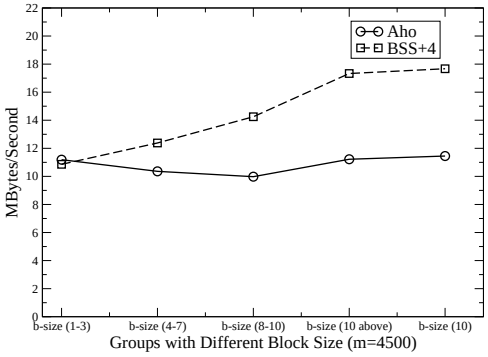


Figure 12: Text scanning speed of the BSS algorithm as a function of the length of the shortest pattern

6.7 Effect of the length of the shortest pattern We pay attention to this parameter because it is heavily related to the performance of the Wu-Manber algorithm. The shift value can not be larger than the length of the shortest pattern. However, we believe that the performance of the BSS algorithm is not restricted by the shortest pattern.

In this section, we still use the four data sets prepared in the Section 6.6, which the length of the shortest pattern are 3, 4, 8, and 11 respectively. In order to confirm our conjecture, we manually reduce the length of the shortest pattern in each data set to 3 by adding a 3-character pattern "DNA" to each pattern set. The experimental results show that with such a small pattern, the BSS algorithm still keep the same average number of the shifted characters for each block as displayed in Figure 11 and the same performance improvement over AC in Figure 12.

6.8 Effect of the mismatched location We believe that the earlier the mismatches happen, the more shifts the BSS algorithm can do and the more improvements the BSS algorithm achieves. Figure 13 shows the performance difference between the BSS algorithm and the AC algorithm over three different mismatch locations: the end of each block, the middle of each block, and the beginning of each block.

For the first location, the BSS algorithm should scan every characters in the text without the capability to shift. This is the worst scenario and the performance of the BSS algorithm is equal to that of the AC algorithm. Because of the diversity of the free text, we treat our previous experiments as the second case – mismatches happen in the middle of blocks on average. To experiment with a realistic scenario for the third mismatch location, we artificially enhance the test data by adding a special character in the beginning of each block in T. When the BSS algorithm reads such character, it fails and shifts to the beginning of the next block. The same thing happens until the end of T. Figure 13 confirms our conjecture.

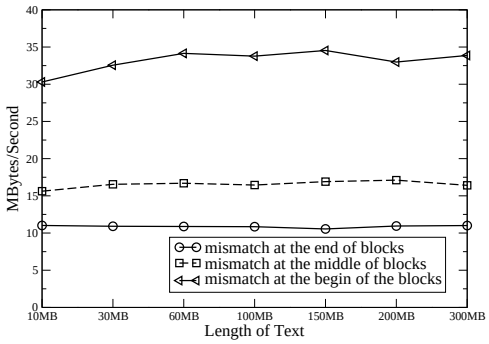


Figure 13: Text scanning speed of the BSS algorithm as a function of the mismatched location

6.9 Experiment on equal-length blocks We get the DNA data from Stanford University[4]. The alphabet size is four (A, C, T, G) and each block is with the fixed length 10. Because of the fixed block length, we can calculate the shift value for each state in the state machine and store the value in advance. Such

[4] $ftp://genome-ftp.stanford.edu/pub/yeast/data\_download/sequence/genomic_sequence/chromosomes/gcg/$

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited

operation can make the BSS algorithm more efficient by avoiding the time to judge where is the beginning of the next block in the match stage. With the same text size (n=10MB), we test the BSS algorithm and the AC algorithm in six different values of *m: 1,000, 5,000, 10,000, 20,000, 30,000, 50,000*. The text scanning speeds of AC are 10.21, 8.18, 7.5, 6.72, 5.35, 4.33 (MB/second) while the text scanning speeds of the BSS algorithm are 18.25, 15.04, 13.83, 12.48, 11.64, 8.98 (MB/second). respectively.

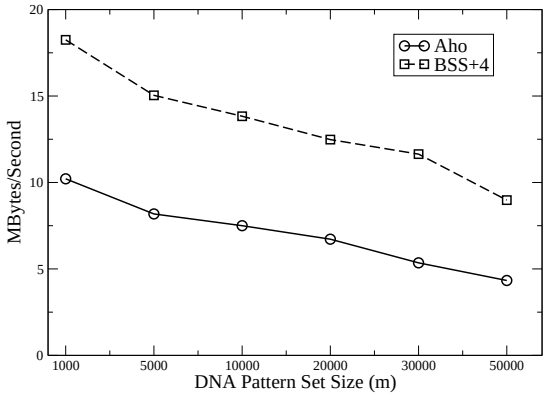


Figure 14: Text scanning speed of the BSS algorithm on DNA dataset

6.10 Effect of each modification step in the BSS algorithm In order to see the detailed effect of each modification the BSS algorithm implemented, we test it in two versions: the BSS algorithm with only the first two modification steps (Section 3.3.1 and 3.3.2, see the curve "BSS+2" in Figure 15), and the BSS algorithm with all the four modification steps (Section 3.3, see the curve "BSS+4" in Figure 15). We also compare these two versions with the AC algorithm and the WM algorithm. The results show that the first two modification steps increase AC by 30% on average, and the last two modification steps increase AC by 5% more.

7 Conclusion

We present a new online exact multi-pattern matching algorithm to search for multiple patterns simultaneously, the Block Suffix Shifting (*BSS*) algorithm. The BSS algorithm is faster than previous character-level algorithms and is scalable to a very large number of patterns. The most important improvement of the BSS algorithm is the significant shifting functionality on segments of text which exploits the natural structure of text itself. In addition, the BSS algorithm avoids the typical 'substring' false positive errors. We apply the BSS algorithm on healthcare data. Our experimental

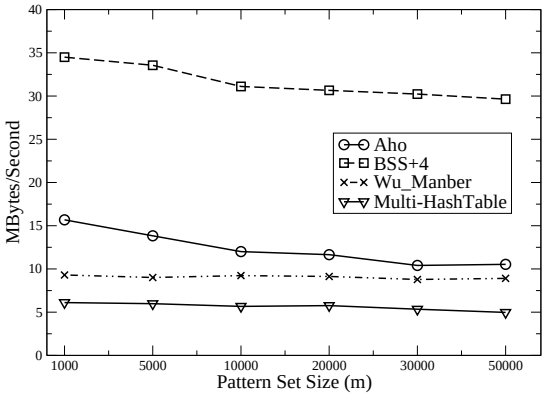


Figure 15: Comparison of the text Scanning speeds of the BSS algorithm as a function of modification steps

results show significant improvements in text scanning speed and reduce redundancy failure functions which lead to useless backward jumping.

References

[1] http://www.unet.univie.ac.at/aix/cmds/aixcmds2/fgrep.htm.

[2] C. M. Aho AV. Efficient string matching: an aid to bibliographic search. In *Communications of the ACM 18*, pages 333–340, June 1975.

[3] C.-W. B. A string matching algorithm fast on the average. In *Proc. 6th International Colloquium on Automata, Languages, and Programming*, pages 118–132, 1979.

[4] M. J. Boyer RS. A fast string searching algorithm. In *Communications of the ACM20*, pages 762–772, 1977.

[5] D. W. Catharine W. Burt, Esther Hing. Electronic medical record use by office-based physicians. In *National Center for Health Statistics*, 2005.

[6] V. P. Donald Knuth; James H. Morris, Jr. Fast pattern matching in strings. In *SIAM Journal on Computing*, pages 323–350, 1977.

[7] K. Fredriksson. On-line approximate string matching in natural language. In *Fundamenta Informatica*, pages Volume 72, Issue 4, 453–466, 2006.

[8] N. Horspool. Practical fast searching in strings. In *Software Practice and Experience*, page 10, 1980.

[9] U. Manber. Agrep, an approximate grep. In *http://www.tgries.de/agrep/*, 2005.

[10] M. Roesh. Snort: Lightweight intrusion detection for networks. In *in Proceedings of the 13th Systems Administration Conference*, 1999, USENIX.

[11] U. M. Sun Wu. A fast algorithm for multi-pattern searching. In *Technical Report TR 94-17, University of Arizona at Tuscon*, May 1994.

Copyright © by SIAM. Unauthorized reproduction of this article is prohibited