Real-Time Data Pre-Processing Technique for Efficient Feature Extraction in Large Scale Datasets

Ying Liu[‡][*], Lucian V. Lita[†], R. Stefan Niculescu[†], Kun Bai[‡], Prasenjit Mitra[‡], C. Lee Giles[‡]

College of IST[‡]
The Pennsylvania State University
University Park, PA 16802
{yliu,pmitra,giles, kbai}@ist.psu.edu

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

ABSTRACT

Due to the continuous and rampant increase in the size of domain specific data sources, there is a real and sustained need for fast processing in time-sensitive applications, such as medical record information extraction at the point of care, genetic feature extraction for personalized treatment, as well as off-line knowledge discovery such as creating evidence based medicine. Since parallel multi-string matching is at the core of most data mining tasks in these applications, faster on-line matching in static and streaming data is needed to improve the overall efficiency of such knowledge discovery. To solve this data mining need not efficiently handled by traditional information extraction and retrieval techniques, we propose a Block Suffix Shifting-based approach, which is an improvement over the state of the art multi-string matching algorithms such as Aho-Corasick, Commentz-Walter, and Wu-Manber. The strength of our approach is its ability to exploit the different block structures of domain specific data for off-line and online parallel matching. Experiments on several real world datasets show how our approach translates into significant performance improvements.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]: Information Search and Retrieval – search process

General Terms

Algorithms

Keywords

multiple-pattern matching, block suffix shift, pre-processing, feature extraction

[*]Partial work done during Siemens internship.

1. INTRODUCTION

In recent years, the ever increasing amount of web data is exploited by more and more data mining applications, such as information extraction from electronic data [15], discovering interesting usage patterns in text collections, [12] finding advertising keywords on web pages [23], detecting spam web pages [20] and mining the shopping preferences of web surfers. Most of these data mining applications employ automatic or semi-automatic machine learning techniques that make use of features as the atomic units of information. Because for online web data-mining applications, speed is of essence, effective algorithms for feature extraction are of outmost importance. In particular, information extraction (parallel concept extraction from blogs, news, instance messages etc), online and off-line search (e.g. retrieval), and traffic scanning (e.g. network scanning [16], online query log processing [10]) are application classes in which massive concept sets are sought in large unstructured and semi-structured online datasets.

Data pre-processing, including feature extraction for statistical information extraction and classification tasks, is a considerable bottleneck. Existing advanced multi-pattern matching algorithms are crucial in time-sensitive processing of high volume data. The explosion in the size and number of information sources observed in recent years emphasizes the need for identifying large concepts sets in vast amounts of data. In most cases, fast parallel multi-pattern matching algorithms are needed for rapid and timely processing of data. In the news domain, in order to process large corpora, resource-driven information extraction relies on matching large free text databases against massive concept sets drawn from resources such as gazetteers, otologies, semantic networks etc. This processing step is often time consuming, yet inherently paralellizable.

One of the most frequent applications using DNA databases involves approximate and exact searching for specific sequences. Part of the task is identifying *all occurrences* of multiple sequence *simultaneously* [18]. Performing this task efficiently is often a requirement, especially considering the rate at which genomic databases (corpora) are growing. In processing large text corpora, resource-driven extraction relies on matching large free text databases against *sizable concept sets* drawn from resources such as gazetteers, otologies, semantic networks etc.

Words, n-grams and patterns (expressions) of interest are among the most frequently used features/concepts. To extract these kinds of features/concepts, the above applications need a fundamental but often neglected step: efficient

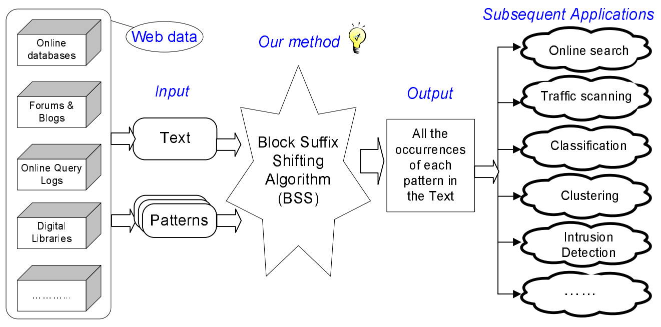


Figure 1: The role of the BSS algorithm in the web data extraction and analysis field

multi-string simultaneous matching which is also referred to as *parallel multi-concept extraction*, an often time consuming, yet inherently parallelizable process.

The multi-pattern search approach is extremely useful in situations when the phrases to be extracted are known while the *text* on which the search is performed is unknown ahead of time, very large, and continuously growing. Under these constraints, indexing is not a viable option: non-linear algorithms are not scalable and extraction cannot be performed online. Current state of the art multi-pattern extraction algorithms such as Aho-Corasick (AC) [6], Commentz-Walter (CW) [7], and Wu-Manber (WM) [22] provide solutions with theoretical complexity linear in the size of patterns and text to be matched against, but with non-negligible constants. In many online data sources and web datasets, additional domain-specific constraints have the potential to maintain the low theoretical complexity while significantly reducing the extraction/matching time. For example, a pervasive constraint takes advantage of the fact that across multiple domains, databases (*texts*) are naturally organized in terms of semantic units or blocks (genetic blocks, words, fields etc).

In this paper, we present a new online exact parallel 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. Once the pattern set is given, the BSS algorithm achieves efficient pattern matching performance for the constantly changing and growing online data, especially for free-text environments. Moreover, we present experiments in several popular online data sets 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 query log databases. The BSS algorithm has the following advantages:

- exploits the natural structure of dataset;
- allows significant character shifting;
- avoids backtracking jumps that are not useful;
- is efficient in terms of overall matching time;
- avoids the typical "sub-string" false positive errors;
- works extremely well on the web data in the free text, the semi-structured, or the structured format;
- has multilingual applicability, such as alphabetic languages and ideographical languages.

In Section 2, we discuss the related work. Section 3 elaborates the detailed state machine construction and the matching process of the BSS algorithm. In Section 3.4.1, we show the improvements of the BSS algorithm compared with the AC algorithm on an example. Section 4 analyzes the algorithm complexity. We analyze the experimental results in Section 5 and conclude in Section 6.

2. RELATED WORK

Aho-Corasick (AC) [6], Commentz-Walter (CW) [7] [24], and Wu-Manber (WM) [22] 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* [5] while the Wu-Manber(WM) algorithm is used in *glimpse* [19]. However, these algorithms seldom apply a *shift* operation in practice for numerous patterns and large text.

The Aho-Corasick (AC) algorithm [6] is a linear-time algorithm based on an automata approach: combines automata with Knuth-Morris-Pratt algorithm. It combines automata with an extension of the Knuth-Morris-Pratt (KMP) algorithm [13] by a method that includes an automaton based approach using suffix tree-like links [11] [9] 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 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 [7] combines the Boyer-Moore (BM) [8] method with the Aho-Corasick algorithm. Although it claim that it enables AC to shift, the Commentz-Walter algorithm is only faster than the AC algorithm on small pattern sets and long minimum pattern lengths.

The Wu-Manber(WM) algorithm [25] presents a different approach that also based on the idea of the Boyer-Moore algorithm. The Wu-Manber(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, for each block there are multiple patterns that match it, the algorithm still needs to evaluate them one by one and thus the advantage of shifting can not be shown.. The performance of the Wu-Manber(WM) algorithm is heavily constricted by the length of the shortest pattern.

Although the Wu-Manber(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 Wu-Manber(WM) algorithm, the shift value was zero 5% of the time for 100 patterns, 27% for 1000 patterns, and 53% for 5000 patterns [22]; Moreover, when a block of size B in the text matches with the suffixes of some patterns, the Wu-Manber (WM) algorithm has to check these patterns one by one and the advantage of shifting can not be shown.

3. BLOCK SUFFIX SHIFT ALGORITHM

3.1 Problem Statement

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.

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.

3.2 The Block/Separator Property

Although some literatures [22] [17] claim that the performance of the Wu-Manber (WM) algorithm is better than the AC algorithm in practice, no literature gives concrete experimental results to show the difference. Moreover, the new SNORT [21] release note officially shows that the WM algorithm is slower than the AC algorithm and the former will be deprecated. Our Block Suffix Shifting (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

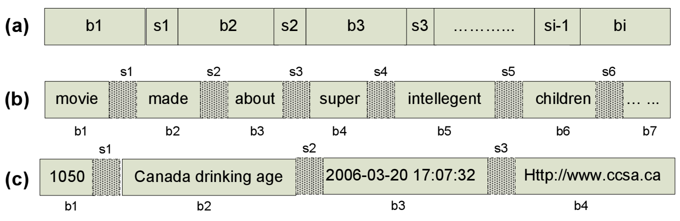


Figure 2: (a): The general block/seperator property; (b): The example T in the basic block/seperator property; (c): The example T in the composite block/seperator property;

of *blocks* (b_1, b_2, ...) and the *separators* between the *blocks* (s_1, s_2, ...) (see Figure 2 a). We call it the *"block/separator"* property. The blocks can have variable lengths and the separators can be situated in different positions. This property holds for many applications. According to the different data structures, we further define the property in two levels: the *basic "block/separator"* property and the *composite "block/separator"* property.

The *basic "block/separator"* property is applicable to any free texts, e.g., the online documents in the digital libraries, the web pages, and the blogs etc. Because *word* is the shortest meaningful unit in the free-text, we define each *block* in the *basic "block/separator"* property as a *word* (see Figure 2b). All blocks share the equal chance to match a pattern.

In addition, some other web data has structured or semi-structured format, e.g., the online query log records, the XML files, and the DNA database etc. Figure 2c shows an randomly downloaded query log record. Considering such structure information can facilitate the data matching performance. Once the patterns are given, we can quickly decide which segments in T we should check and which we should not. If the patterns are the query keywords, we only have to check "b2" in Figure 2c and shift over all the others. We say that these web data follows the *composite "block/separator"* property and such a segment is the *composite "block"* because it can consist several *words*.

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[14]. 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, etc. 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 from the web data in English and use orthography-oriented tokenization techniques. We define the *separator* as *space* characters and we preprocessed the text to remove the punctuation as well as the redundant spaces.

3.3 The Pattern Starting Constraint

The data structure imposes an important constraint: *if the text follows a "block/separator" property, 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*. This constraint is applicable to the text in both the *basic* block and the *composite* block.

For the example in Section *2*, if c_{i+1} is not the first 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 text following the *basic "block/separator"* property, the skipped segments are the suffixes of all the mismatched *words*. For the text following the *composite "block/separator"* property, the skipped segments are the suffixes of all the mismatched *composite blocks*. 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*. At beginning, the BSS algorithm determine 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. Because of the limited space, please see the detailed goto function and the output function in [6];

3.4 The Failure Function of the BSS algorithm

3.4.1 Example with the AC algorithm and the BSS algorithm

Example 1. p_1="rent acura car"*;* p_2="acura car repair"*;* p_3="car deal"*;* p_4="carseat safety"*;*

Figure 3 shows the state machine generated by the AC algorithm and the BSS algorithm for Example *1*, which each pattern is a randomly selected query keyword. The state machine is a rooted directed tree with fifty states totally. Each state is represented by a number from *0* to *49*. State 0 is designated as the root state s_0. We only display the failure functions that do not go to the root state.

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 web data domain.

3.4.2 When should the BSS algorithm shift?

Let us consider another example shown as follows.

Example 2. p1="eel", T="heel hurt".

According to the AC algorithm (Figure 4a), 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, 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 (*"hurt"*).*

3.4.3 Failure functions connecting characters with different character index number (CIN)

Let us consider the third example shown as follows.

Example 3. p1="eel", T="ear eeel".

AC will generate another "substring matching" result for *Example 3* (Figure 4a): *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 remaining word suffix (*"r"*) and jump to the beginning of the next word. For the second word *"eeel"*, the failure functions of AC in *Figure 4a* ($f(s_2) = s_1$) 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 4a*, the *CINs* connected by the failure function are *1* and *2 (1≠2)*. *Figure 4c* and *Figure 4d* 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 4b and Figure 4d*):

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

3.4.4 The redundant failure functions and matchings

Let us consider the following example.

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 *Figure5*). 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 additional useless operations: $f(s_5) = s_1$ and $t \neq b$, which compounds for a lot of wasted time when the T grows. Suppose s_1 has another child besides *b* (see Figure 5c), the BSS algorithm should keep the failure function $f(s_5) = s_1$

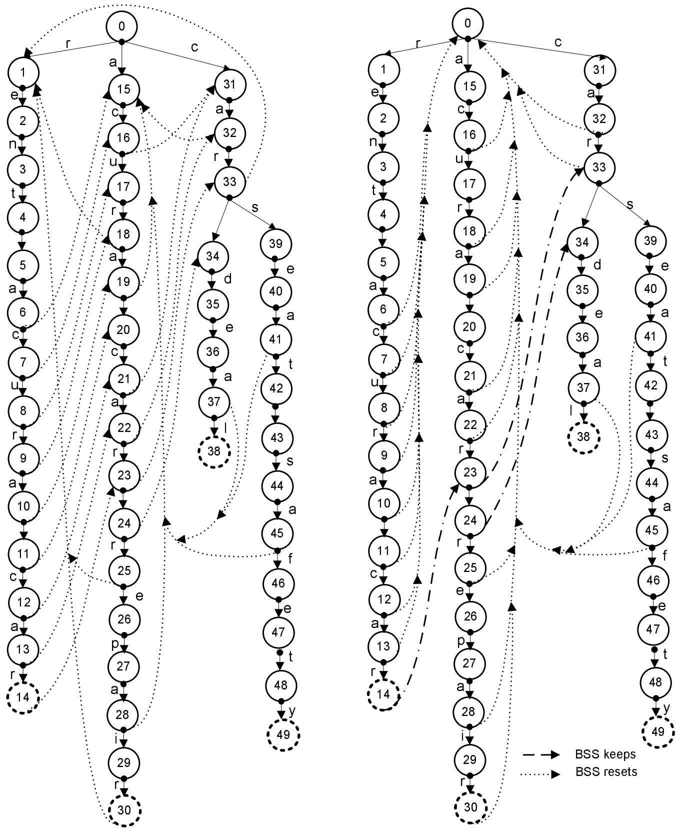


Figure 3: The state machine of Example 1 generated by the AC algorithm (left) and the BSS algorithm (right)

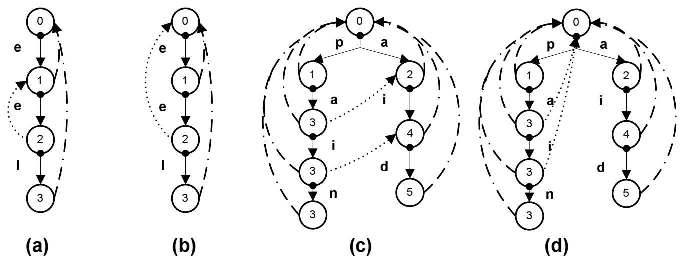


Figure 4: Sample state machines generated by the AC algorithm (a,c) and the BSS algorithm (b,d)

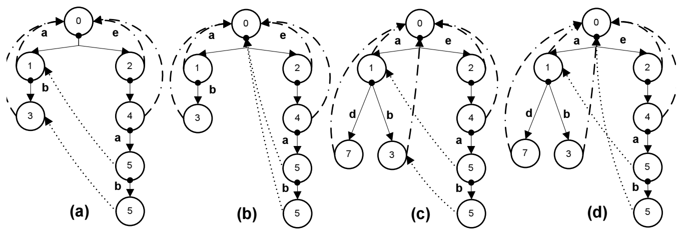


Figure 5: Examples of the redundant failure functions of the AC algorithm (a, c) and the modification of the BSS algorithm (b, d)

(see Figure 5d) 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 Figure5d).*

3.4.5 The propagated failure functions

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 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 3 (left),the state machine generated by the AC algorithm for the Example 1, there are five sets of the propagated failure functions: (s_1, s_{33}, s_{23}, s_{14}), (s_{15}, s_{32}, s_{22}, s_{13}), $(s_{15}, s_{19}, s_{10}), (s_{31}, s_{16}, s_7), and (s_1, s_{18}, s_9)$. If we modify $p2$ in the Example 1 from *"acura car repair"*to *"acura car"*, we should delete the states from s_{24} to s_{30} correspondingly. However, we still keep the propagation set (s_{33}, s_{23}, s_{14}) because although s_{14} and s_{23} have the same goto function (null) and the same CIN number, $f(s_{23}) = s_{33}$ enables us to reach s_{33} and s_{33} has different goto functions: *goto*(s_{33}, "s")=s_{39}, and *goto*(s_{33}," ")=s_{34}. Suppose *T="rent acura car deal"*, we can arrive at s_{38} to get an output along the propagated failure functions. 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 $f(s_\alpha) = s_\beta$.*

3.4.6 The failure function rules of 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 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$\in \sum$;
    /*AC-Construct-GOTO()*/;
    for each $p_k \in P$ do
        for $c_i \in p$; i=1,..,m do
            trie.add($c_i$);
    /*BSS-build-NFA()*/;
    for each a$\in \sum$ such that (s=goto(0,a))$\neq$0 do
        s$\rightarrow$Queue; fail(s)$\leftarrow$0;
    while (r$\leftarrow$Queue)$\neq \emptyset$ do
        for each a$\in \sum$ such that (s=goto(r,a))$\neq$fail do
            s$\rightarrow$Queue; state$\leftarrow$fail(r);
            while (next=goto(state,a))=fail do
                state$\leftarrow$fail(state);
            /*BSS reset fail function*/;
            if next$\neq$0 and next.cin=s.cin then
                if $\forall$a$\in \sum$, goto(next,a)=goto(s,a) and fail(next)=0 then
                    fail(s)$\leftarrow$0;
                else
                    fail(s)$\leftarrow$next;
            else
                fail(s)$\leftarrow$0;
    /*AC-Convert-NFA-DFA()*/;
end
```

Table 1 elaborates the steps that the BSS algorithm resets the *do-not-go-to-the-root-state* failure functions for the Example 1. The detailed reset reasons are also displayed.

3.5 The 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.

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 *block*, shifts to the beginning of the next *block*, 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 *block* in T. In another sentence, when the failure function goes to s_0, the shift value is greater than zero. Otherwise, the shift value is zero. This value reflects the length of the segment to be shifted.

Figure 6a displays the match phase of the AC algorithm for T= *"movie made about super intelligent children"* and the patterns p_1= *"motor"*, p_2= *"stock"*, p_3= *"childrensplace.com"*; According to the AC algorithm, we have to scan all the 43 characters without any shift. Figure 6b displays the match phase of the BSS algorithm for the same T and P. Because this T follows the *basic "block/separator"* property,

Table 1: The failure function changes of the BSS algorithm on the Example 1

Level	state	AC	BSS	Reasons
2	s_{16}	s_{31}	s_0	$CIN(s_{16}) \neq CIN(s_{31})$, see 3.4.3
2	s_{32}	s_{15}	s_0	$CIN(s_{32}) \neq CIN(s_{15})$, see 3.4.3
3	s_{33}	s_1	s_0	$CIN(s_{33}) \neq CIN(s_1)$, see 3.4.3
4	s_{18}	s_1	s_0	$CIN(s_{18}) \neq CIN(s_1)$, see 3.4.3
5	s_{19}	s_{15}	s_0	$CIN(s_{19}) \neq CIN(s_{15})$, see 3.4.3
6	s_6	s_{15}	s_0	redundant failure function, see 3.4.4
6	s_{41}	s_{15}	s_0	$CIN(s_{41}) \neq CIN(s_{15})$, see 3.4.3
7	s_7	s_{16}	s_0	redundant failure function, see 3.4.4
7	s_{21}	s_{31}	s_0	redundant failure function, see 3.4.4
7	s_{37}	s_{15}	s_0	$CIN(s_{37}) \neq CIN(s_{15})$, see 3.4.3
8	s_8	s_{17}	s_0	redundant failure function, see 3.4.4
8	s_{22}	s_{32}	s_0	redundant failure function, see 3.4.4
9	s_9	s_{18}	s_0	redundant failure function, see 3.4.4
9	s_{23}	s_{33}	s_{33}	s_{33} has different goto(), see 3.4.4
10	s_{10}	s_{19}	s_0	redundant failure function, see 3.4.4
10	s_{24}	s_{34}	s_{34}	s_{34} has different goto(), see 3.4.4
10	s_{45}	s_{15}	s_0	redundant failure function, see 3.4.4
11	s_{11}	s_{20}	s_0	redundant failure function, see 3.4.4
11	s_{25}	s_1	s_0	redundant failure function, see 3.4.4
12	s_{12}	s_{21}	s_0	redundant failure function, see 3.4.4
13	s_{13}	s_{22}	s_0	redundant failure function, see 3.4.4
14	s_{14}	s_{23}	s_{23}	s_{23} has different goto(), see 3.4.4
14	s_{28}	s_{15}	s_0	$CIN(s_{28}) \neq CIN(s_{15})$, see 3.4.3
16	s_{30}	s_1	s_0	$CIN(s_{30}) \neq CIN(s_1)$, see 3.4.3

the BSS algorithm shifts on the block suffix of each mismatched blocks. With these shifts, we only have to check 16 over 43 characters. Figure 6c displays the match phase of the BSS algorithm for another example T with patterns p_1= *"candidacy exam"*, p_2= *"stock"*, and p_3= *"ramada.com"*. The T contains three randomly selected AOL query logs that follows the *composite "block/separator"* property.

Using Example 1, suppose T= *"check out these promotions at selected thrifty car rental locations the acura mdx is all-new for 2007 and always check the carseat safety"*. The AC algorithm has to scan all the 137 characters without any shift. However, the BSS algorithm only checks 53 characters and shifts over all the underlined suffixes:

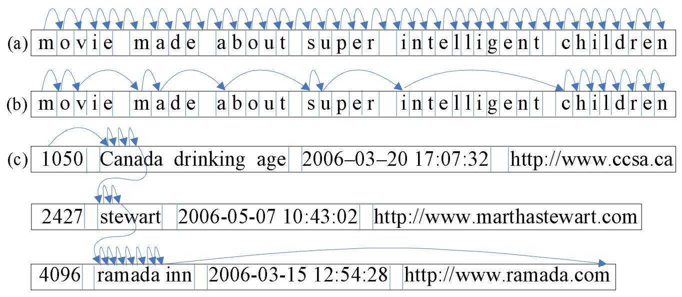
"check out these promotions at selected thrifty car rental locations the acura mdx is all-new for 2007 and always check the carseat safety".

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 mismatches are encountered on the first character in each block and the blocks are of equal length t. 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.



(a) movie made about super intelligent children
(b) movie made about super intelligent children
(c) 1050 Canada drinking age 2006–03–20 17:07:32 http://www.ccsa.ca
2427 stewart 2006-05-07 10:43:02 http://www.marthastewart.com
4096 ramada inn 2006-03-15 12:54:28 http://www.ramada.com

Figure 6: (a): The match phase for the example free text based on the AC algorithm; (b): The match phase for the example text in the "basic" Block/Seperator property based on the BSS algorithm; (c): The match phase for the example text in the "composite" Block/Seperator property based on the BSS algorithm;

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.

- 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;
- If a mismatch exists, the mismatch happens in the last character of a block/word.

5. EXPERIMENTAL RESULTS

We implemented experiments comparing the BSS algorithm with two famous multi-pattern matching algorithms (the AC and the WM algorithm) on the real web data. In addition to compare the matching performance in terms of the speed, we also investigate the impact of algorithm on 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, the modification steps of the BSS algorithm, and the level of the *"block/separator"* property;

We are interested in quantifying the average text scanning speed (MB/second) on the test data and the average number of the shifted characters per block. For each experimental setting, we randomly select T and P from the web data, perform twelve separate runs and show the averages.

5.1 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[21] and are implemented in JAVA. We also implement the BSS algorithm in JAVA, by modifying the available version of the AC algorithm.

5.2 Test data

In order to investigate performance of each algorithm as well as its advantages and disadvantages, our experiments were performed mainly on the the online web data: the query logs and the electronic documents in the digital libraries. The query logs originate from low to high-traffic web sites such as well known search engines, retail store web portals, news web sites, social networks, personal web sites etc. This data covers rich information such as the query itself, date, time, source ip, type of redirection etc, which is used extensively for statistics gathering, clustering, as well as query log search. For all of these tasks, fast, user-driven access to specific information from query logs is necessary, especially when considering the fact that this type of data is constantly changing and constantly growing. To illustrate the immensity of these data sets, consider the *daily* query volume of US users observed [2] based on March 2006 in several high-traffic search engines Google: 91 million, Yahoo: 60 million, MSN: 28 million, AOL: 16 million, Ask: 13 million. Given such large data sets, there is a growing need for swift and efficient processing while at the same time allowing fast search and extraction.

The second data source are the scientific chemistry papers collected from the digital library (Royal Chemistry Society [4]). Another important data set for our experiment is the KDD Cup datasets [3] 1997. All the free-text data follows the basic Block/Seperator property while all the query log data follows the composite Block/Seperator property.

All the test data has three different alphabet sizes: the English free text (σ=96) and the DNA data (σ=4).

English alphabet: In our experiment, the English free texts come from two useful web data: The size of the alphabet σ is 96 (all visible characters). The total AOL query log size is $650MB$. The randomly selected $100,000$ patterns fall under two categories: the query keywords and the clicked URLs; The total size of the electronic chemical documents is 500 MB and we randomly select $50,000$ chemical names to construct the pattern set.

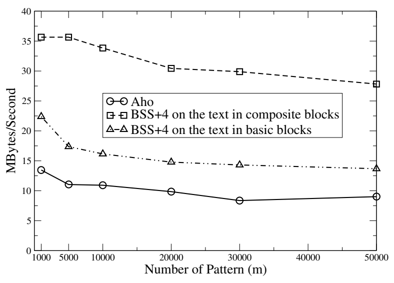
DNA alphabet(σ=4): All test data comes from the DNA data set [1] in *gcg* format. The alphabet $\sum = \{A, C, G, T\}$. The total size of the text is 250MB and the the total number of patterns we used is $50,000$.

5.3 Results of the web data in the composite "block/ separator" property

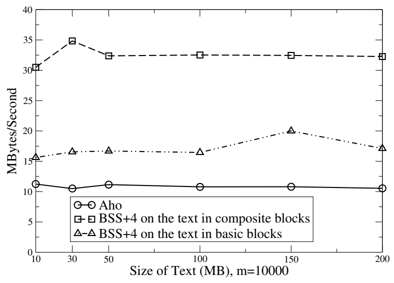
5.3.1 *Effect of m on the Match Performance*

Figure 7(a) shows the results of the performance of the BSS algorithm vs. the AC algorithm on the AOL query log, 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 query log data, the BSS algorithm maintains a high performance over AC for every m: the improvement rates of the BSS algorithm are 165.1%, 222.9%, 209.6%, 209.04%, 257.38%, and 208.75% respectively. The "BSS+4" refers to the implemented BSS algorithm with all four modification steps based on the AC algorithm (see Section 3.4).

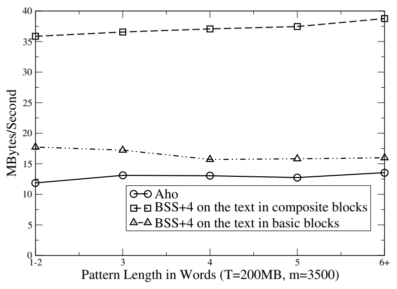
We notice that the BSS algorithm has a performance-decreasing trend along the growth of the number of patterns. It is not surprising because as the increasing of m, the state machine is enlarged with more states, which provide more



(a)

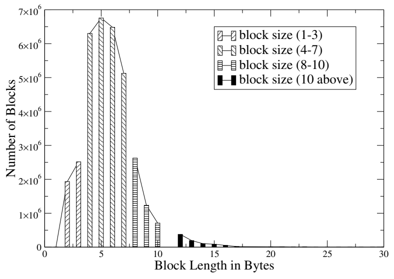


(b)

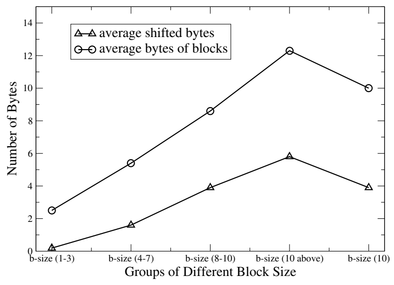


(c)

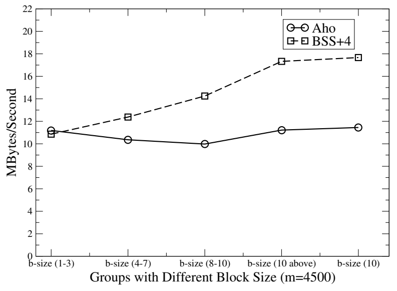
Figure 7: The text scanning speed of the BSS algorithm as a function of (a): The size of the pattern set m; (b): The size of the text n; (c): The pattern length. Based on the AOL query log data set that follows the *composite "block/separator"* property



(a)

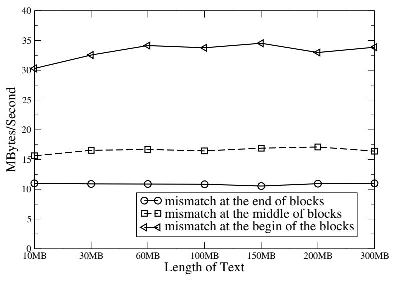


(b)

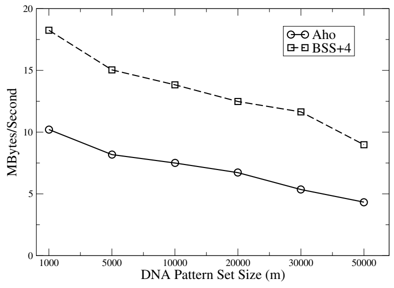


(c)

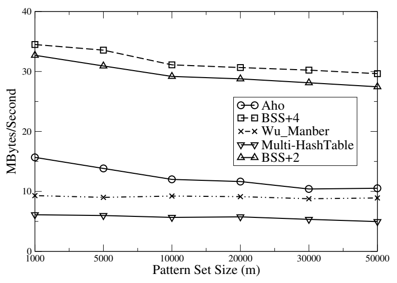
Figure 8: (a): The distribution of the pattern length in terms of the number of the blocks; (b): The average character shift of the BSS algorithm as a function of the block size; (c): The text scanning speed of the BSS algorithm as a function of the length of the shortest pattern. Based on the online scientific documents that follow the *basic "block/separator"* property



(a)



(b)



(c)

Figure 9: Text scanning speed of the BSS algorithm as a function of (a): The mismatched location on the online scientific documents that follow the *basic "block/separator"* property; (b): the size of the pattern set m of the the equal-length DNA data set; (c) The modification steps of the BSS algorithm. Based on the online scientific documents that follows the *basic "block/separator"* property

chances for T to traverse. The ratio of the matched segments in T will be increased and less block suffixes the BSS algorithm can shift. When m reaches a very large value that all the blocks in T match with P, a worst case scenario happens for the BSS algorithm and its performance curve will meet with the curve of AC in the figure.

5.3.2 Effect of n on the Match Performance

We performed similar experiments on the AOL query logs in order to investigate the effect of n on the run time. The six value of n are: $10M$, $30M$, $50M$, $100M$, $150M$, and $200M$. *Figure 7(b)* shows the run time of the BSS algorithm versus the run time of AC with the same pattern size $m = 10000$. The results confirm: (i) the performance improvement of the BSS algorithm is steady as the increasing of n; (ii) the performance of the BSS algorithm is linear to n.

5.3.3 Effect of the Pattern Length on the Match Performance

In order to investigate the distribution of the pattern

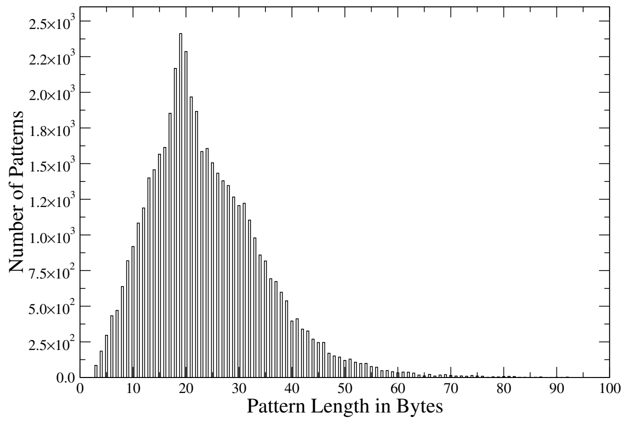


Figure 10: The Distribution of pattern length

length and its effect on the match performance, we further divide the randomly selected 100,000 patterns from the AOL query log into four groups according to the pattern length in term of words: one-or-two-word patterns (50145), three-word patterns (21100), four-word patterns (12982), and five-and-above-word patterns (15772). It shows that in the query log field, a considerable number of query concepts are short (one or two words).

To study the effect of pattern length on matching performance, we randomly select 250 patterns from each group and apply both the BSS algorithm and the AC algorithm on the same text ($200MB$). The results are shown in *Figure 7(c)*. For all the four groups covering both short and long pattern lengths, the performance gain for the BSS algorithm over AC is 202.1%, 178.9%, 184.3%, 193.7% and 186.1s% respectively.

Comparing with the query logs, the patterns in the chemistry field share a similar property: many patterns are short in terms of the number of characters. Figure 10 shows the detailed statistics on the pattern length of the 50,000 patterns from the online documents in the chemistry domain.

5.3.4 Effect of the level of the "Block/Seperator" property

In the three diagram of Figure 7, we compare the text scanning speed of the BSS algorithm on the texts in different "Block/Seperator" property levels. Each pair of the comparison are implemented with the same text size n and the same pattern set size m. The experimental results show that the BSS algorithm works extremely well on the text in the composite "Block/Seperator" property. It is not surprising because as long as we know the structure of the text in advance, we can easily know which blocks we should scan and which blocks we can shift over. With a minor change on one single line in the matching phase source code, we can efficiently speed up the performance.

5.4 Results of the web data in the basic "block/separator" property

5.4.1 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 the BSS algorithm can achieve. In order to test our conjecture, we investigate the distribution of the block length first (see Figure 8(a)) based a randomly selected $350MB$ online scientific chemistry papers.

We divided all the 34.54 millions of blocks in the T into four categories according to their block sizes: 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 8(b) 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).

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 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 8(c)). 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.

5.4.2 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: each 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 this parameter. 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 8(b) and the same performance improvement over AC in Figure 8(c).

5.4.3 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 can achieve. Figure 9(a) 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.

In order to obtain the experimental results for the third mismatch location, we manually generate the testing data by adding a special character at 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 9(a) confirms our conjecture.

5.5 Experiments on equal-length blocks

In this section we test the BSS algorithm on the text and the pattern set that are composed of the equal-length blocks. We get the DNA data from the online data set [1]. The size of the alphabet (A, C, T, G) is four. 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 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 (see Figure 9(b)).

5.6 Effect of each modification step in the BSS algorithm

In order to see the effect of the modification steps of the BSS algorithm, we test two versions: the BSS algorithm with the first two modification steps (Section 3.4.2 and 3.4.3, see the curve "BSS+2" in Figure 9(c)), and the BSS algorithm with all four modification steps (Section 3.4.4 and 3.4.5, see the curve "BSS+4"). We also compare these two versions with the AC algorithm and the WM algorithm using the same AOL query data set (n=50MB). The results show that the first two modification steps increase AC algorithm by 150% on average, and the last two modification steps increase the "BSS+2" by 5%.

6. CONCLUSIONS

In this paper we presented the Block Suffix Shifting (*BSS*), a new online exact multi-pattern matching algorithm to search for multiple patterns simultaneously. 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. Experimental results show significant improvements in text scanning speed and reduced redundancy failure functions which lead useless backward jumping.

7. REFERENCES

[1] ftp://genome-ftp.stanford.edu/pub/yeast/data_download/sequence/genomicsequence/chromosomes/gcg/.

[2] http://searchenginewatch.com.

[3] http://www.kdnuggets.com/datasets/kddcup.html.

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

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

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

[7] C.-W. B. A string matching algorithm fast on the average. In *Proc. 6th ICALP*, pages 118–132, 1979.

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

[9] C.-H. Chang and S.-C. Lui. Iepad: information extraction based on pattern discovery. In *WWW*, pages 681–688, 2001.

[10] J. Chen and T. Cook. Mining contiguous sequential patterns from web logs. In *WWW*, pages 1177–1178, 2007.

[11] H. Chim and X. Deng. A new suffix tree similarity measure for document clustering. In *WWW*, pages 121–130, 2007.

[12] A. Don, E. Zheleva, M. Gregory, S. Tarkan, L. Auvil, T. Clement, B. Shneiderman, and C. Plaisant. Discovering interesting usage patterns in text collections: integrating text mining with visualization. In *CIKM*, pages 213–222, 2007.

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

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

[15] V. Gedov, C. Stolz, R. Neuneier, M. Skubacz, and D. Seipel. Matching web site structure and content. In *WWW (Alternate Track Papers & Posters)*, pages 286–287, 2004.

[16] S. Kals, E. Kirda, C. Krügel, and N. Jovanovic. Secubat: a web vulnerability scanner. In *WWW*, pages 247–256, 2006.

[17] S. Kim and Y. Kim. A fast multiple string-pattern matching algorithm. In *Proc. of 17th AoM/IAoM Conference on Computer Science*, Aug. 1999.

[18] Y. Liu, L. V. Lita, S. Niculescu, P. Mitra, and C. L. Giles. Finding a haystack in haystacks – simultaneous identificcation ofconcepts in large bio-medical corpora. SIAM SDM 2008.

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

[20] A. Ntoulas, M. Najork, M. Manasse, and D. Fetterly. Detecting spam web pages through content analysis. In *WWW*, pages 83–92, 2006.

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

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

[23] W. tau Yih, J. Goodman, and V. R. Carvalho. Finding advertising keywords on web pages. In *WWW*, pages 213–222, 2006.

[24] B. W. Watson and R. E. Watson. A new family of string pattern matching algorithms. *South African Computer Journal*, 30:34–41, 2003.

[25] S. Wu and U. Manber. Fast text searching with errors. Technical Report TR-91-11, 1991.