ELSEVIER

Available online at www.sciencedirect.com
SCIENCE DIRECT®

Theoretical Computer Science 332 (2005) 391–404

Theoretical Computer Science

www.elsevier.com/locate/tcs

The wide window string matching algorithm

Longtao He[a,*], Binxing Fang[a], Jie Sui[b]

[a] *Research Center of Computer Network and Information Security Technology, Harbin Institute of Technology, Harbin 150001, PR China*
[b] *Graduate School of the Chinese Academy of Sciences, Beijing 100039, PR China*

Received 19 December 2003; received in revised form 11 November 2004; accepted 25 November 2004
Communicated by M. Crochemore

Abstract

Generally, current string matching algorithms make use of a window whose size is equal to pattern length. In this paper, we present a novel string matching algorithm named WW (for Wide Window) algorithm, which divides the text into $\lfloor n/m \rfloor$ overlapping windows of size $2m-1$. In the windows, the algorithm attempts m possible occurrence positions in parallel. It firstly searches pattern suffixes from middle to right with a forward suffix automaton, shifts the window directly when it fails, otherwise, scans the corresponding prefixes backward with a reverse prefix automaton. Theoretical analysis shows that WW has optimal time complexity of $O(n)$ in the worst, $O(n/m)$ best and $O(n(\log_\sigma m)/m)$ for average case. Experimental comparison of WW with existing algorithms validates our theoretical claims for searching long patterns. It further reveals that WW is also efficient for searching short patterns.
© 2004 Elsevier B.V. All rights reserved.

Keywords: Suffix automaton; Reverse prefix automaton; Bit parallelism; Wide window algorithm; String matching

1. Introduction

String matching is always one of the research focuses in computer science [18,12,22]. It is a very important component of many problems, such as text processing, linguistic

* Corresponding author.
E-mail addresses: hlt@hit.edu.cn (L. He), bxfang@mail.nisac.gov.cn (B. Fang)
URL: http://icuc.sf.net/cucme/.

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.12.002

Table 1
Four types of string matching algorithms

Pattern factor	Scanning forward	Scanning backward
Prefix	KMP, Shift-Or	RF, TRF, BNDM
Suffix	FDM, TNDM	BM and its variants

translating, data compression, search engine, speech recognition, information retrieval, computational biology, computer virus detection, network intrusion detection, and so on. Formally, the string matching problem consists of finding all occurrences (or the first occurrence) of a pattern in a text, where the pattern and the text are strings over some alphabet.

In this paper, the *pattern* and the *text* are denoted, respectively, as $x = x_1x_2 \dots x_m$ of length m and $y = y_1y_2 \dots y_n$ of length n. The *alphabet* is denoted as Σ of size σ. And the set of all suffixes of string x is denoted as $Suf(x)$. Likewise, the set of all prefixes is denoted as $Pre(x)$.

Current string matching algorithms work as follows [4]. They scan the text with a window whose size is generally m. They first align the left ends of the window and the text—this specific work is called an *attempt*—then inspect the symbols in the window with different strategies, and after a whole match of the pattern or a mismatch, they shift the window to the right. They repeat the same procedure until the right end of the window goes beyond the right end of the text. This mechanism is usually called the *sliding window mechanism*. According to the scanning strategies in the sliding window, string matching algorithms are classified into four categories (see Table 1):

(1) *Scanning pattern prefixes forward*: The algorithms such as KMP [15] and Shift Or [2], perform the inspections from left to right in the window. They keep the information about all matched pattern prefixes with some automata. The worst case time complexities of this kind of algorithms are $O(n)$, which is optimal in theory. But since they inspect the symbols one by one, their average case time complexities are bad.
(2) *Scanning pattern suffixes backward*: In order to exploit the inspected information, BM [3] and its variants [23] scan from right to left in the window. They shift the window mainly according to the inspected pattern suffixes. These suffixes, which are also the suffixes of the window, are able to improve the length of the shifts, so many symbols of the text are not necessary to inspect. Thus, though the worst case time complexities of the kind of algorithms are $O(mn)$, their average case time complexities are sublinear.
(3) *Scanning pattern prefixes backward*: This kind of algorithms, like RF [16], TRF [9], and BNDM [17], match pattern prefixes by scanning the symbols with the suffix automaton [1] [10] of the reverse pattern from right to left in the window, to improve the length of the shifts further. Their average case time complexities reach $O(n(\log_\sigma m)/m)$, which is theoretically optimal [25].
(4) *Scanning pattern suffixes forward*: FDM [4] is the first one in this category. But since it does not take full advantage of the suffix automaton, FDM has the same complexities as KMP. TNDM [19] is a two-way modification of BNDM. It scans a pattern suffix

[1] Also called DAWG for Directed Acyclic Word Graph.

forward before normal backward scan as BNDM. The experiments indicate that TNDM examines less symbols than BNDM on the average.

In this paper, we studied on the idea of scanning pattern suffixes forward. A new string matching algorithm (called WW for Wide Window) is proposed. We prove via theoretical analysis that the time complexities of this algorithm are optimal in worst, best and average cases: $O(n)$, $O(n/m)$ and $O(n(\log_\sigma m)/m)$. Experimental comparison of WW with existing algorithms validates our theoretical claims for searching long patterns. It further reveals that WW is also efficient for searching short patterns.

The rest of this paper is organized as follows. In Section 2, the basic idea and two important automata are introduced. Section 3 presents the detailed algorithm and a step-by-step example. In Section 4, the time and space complexities are analyzed. Experimental results are given in Section 5, followed by conclusions in Section 6.

2. Basic idea and two useful automata

2.1. Basic idea

After each alignment, the symbol y_i being inspected in the text is not only related to the $m - 1$ symbols $y_{i-m+1}, y_{i-m+2}, \dots, y_{i-1}$ before (according to which, the algorithms such as BM scan from right to left in the window of size m), but also the $m - 1$ symbols $y_{i+1}, y_{i+2}, \dots, y_{i+m-1}$ after (according to which, the algorithms such as KMP scan from left to right in the window of size m). Well then, current inspected symbol and its left and right $m - 1$ symbols compose the window of size $2m - 1$. Charras et al. [5] have studied the similar idea. They presented three algorithms: Skip, KMPSkip, and AlphaSkip. The algorithms perform well for small alphabets and very long patterns. But since they make the attempts in an isolated manner in the same alignment, the most useful information is lost. Thus, many improvements can be made on this idea.

In the window, we can scan the pattern suffixes from middle to right with a suffix automaton first, and then scan corresponding pattern prefixes from middle to left with a reverse prefix automaton if necessary. In this way, the useful information relevant to the middle symbol can be scanned completely, and all occurrences containing the middle symbol in the window can be found, so that the loss of the information is minimized.

Our new algorithm makes use of two kinds of automata, respectively: forward suffix automaton and reverse prefix automaton.

2.2. Suffix automaton (*SA*)

The suffix automaton of a string x of length m is defined as the minimal deterministic (non-necessarily complete) automaton that recognizes the (finite) set of suffixes of x [6]. It is denoted by $SA(x)$ in this paper.

An example of suffix automaton is displayed in Fig. 1, which accepts the set of suffixes of string *aabbabb*: {ε, *b*, *bb*, *abb*, *babb*, *bbabb*, *abbabb*, *aabbabb*}.

The suffix automaton is a well-known structure [6,11,20,1,8]. The size of $SA(x)$ is linear in m (counting both nodes and edges), and a linear on-line construction algorithm exists

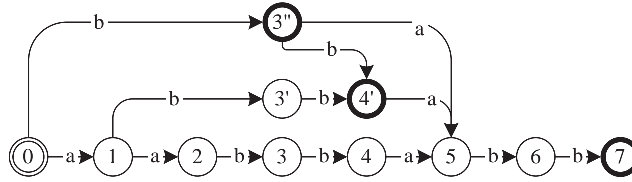

Fig. 1. $SA(aabbabb)$.

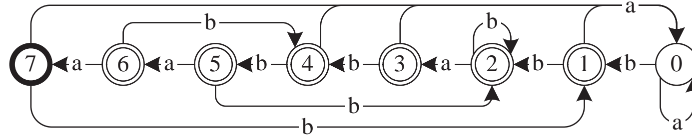

Fig. 2. $RPA(aabbabb)$.

[6]. A very important fact for our algorithm is that this automaton can not only be used reversedly to recognize the pattern prefixes [16,11], but also be used forward to recognize the pattern suffixes.

2.3. *Reverse prefix automaton* (*RPA*)

Reverse prefix automaton mentioned in this paper is in fact the deterministic finite automaton of reverse pattern, except for running in the reverse direction. The algorithm runs *RPA* only if *SA* has matched at least one non-null suffix. *RPA* does not start from the traditional initial state, but from the corresponding state of the maximal suffix matched by *SA*. Formally:

Definition 1. The RPA of a string x is a quintuple $A(Q, \Sigma, \delta, S, T)$, where

$Q = Suf(x) = \{\varepsilon, x_m, x_{m-1}x_m, \ldots, x_2 \ldots x_{m-1}x_m, x\}$. In practice, state $q = x_t x_{t+1} \ldots x_m$ is usually represented by its length $|q| = m - t + 1$;

Σ is the set of all symbols appearing in the text and pattern;

$\delta : Q \times \Sigma \to Q$ is the transition function. $\delta(q, a) = aq$, if and only if $aq \in Suf(x)$; or $\delta(q, a) = p$, where p is the maximum in $Suf(x) \cap Pre(aq)$;

$S = Q - \{\varepsilon\}$ is the set of initial states, $\varepsilon \notin S$ because scanning with *RPA* is not necessary when the suffix matched by *SA* is ε;

$T = \{x\}$ is the set of terminal states, denoting a prefix (which concatenates the suffix matched by *SA* into a pattern) is matched.

As an example, $RPA(aabbabb)$ is given in Fig. 2. Starting from different initial states 1, 2, 3, 4, 5, 6 and 7, the *RPA* can accept different prefixes in a reverse way, respectively: *aabbab*, *aabba*, *aabb*, *aab*, *aa*, *a* and ε.

The construction of *RPA* is similar to that of the string matching automaton [7]. The time complexity is $O(m+\sigma)$, and the space complexity is $O(m\sigma)$.

3. The wide window algorithm

The main characteristic of the algorithm in this paper is the use of a wide window (of size $2m-1$), so we call it the WW algorithm.

3.1. The WW algorithm

The initialization of the algorithm is to construct two automata, respectively: $SA(x)$ and $RPA(x)$. Existing construct algorithms are applied in this paper.

Before the scanning stage is introduced, some definitions are given as follows:

Definition 2. The *attempt positions* are the *i*th $\{i \mid i = km, 1 \leqslant k \leqslant \lfloor n/m \rfloor\}$ positions in the text. An *attempt window* is defined as a slice window of size $2m-1$ whose middle is the attempt position [2]. In the window, the series of $m-1$ symbols before the attempt position is denoted as *left window*, and the rest is denoted as *right window*.

In this way, WW divides the text into $\lfloor n/m \rfloor$ overlapping windows of size $2m-1$. Each window has $m-1$ same symbols as previous and next windows, respectively, and each attempt position occurs only in one window. WW attempts on the windows $y_{(k-1)m+1} \ldots y_{km} \ldots y_{(k+1)m-1}$ in turn, where k ranges from 1 to $\lfloor n/m \rfloor$.

In the windows, the inspections consist of two phases. To facilitate the discussion, the following variables are introduced: r, R which denote the next inspecting position and the current state of $SA(x)$ respectively, and l, L which denote the next inspecting position and the current state of $RPA(x)$. L also records the maximal matched suffix in the first stage. Because the states of $RPA(x)$ are the length of the maximal matched suffix, L is just the initial state of $RPA(x)$ after the first stage. All the initial values of r, R, l and L are set to 0.

(1) As shown in Fig. 3, the algorithm scans the right window $y_{km} \ldots y_{(k+1)m-1}$ from left to right with $SA(x)$. When $SA(x)$ reaches a terminal state, the next scanning position r which is the length of matched suffix is recorded in the variable L. It goes until the transition for the current symbol in the current state is undefined. If $L > 0$, $y_{km} \ldots y_{km+L-1}$ is the maximal matched suffix, and then the algorithm turns into the second phase, else the maximal matched suffix is ε, and the algorithm can shift to next window directly.

(2) As shown in Fig. 4, the algorithm scans the left window $y_{(k-1)m+1} \ldots y_{km-1}$ from right to left with $RPA(x)$ starting from the state L. When the automaton reaches the terminal state, an occurrence is found at current scanning position l. This stage continues when $m - L \leqslant m - 1 - l$ (namely $L > l$) because there will not be any occurrence remained in the window as $RPA(x)$ consumes at least $m - L$ symbols to transfer from state L to terminal state m, but there are $m - 1 - l$ symbols remained. Because $RPA(x)$ scans the

[2] The last window is an exception, which has only $n - m\lfloor n/m \rfloor + m$ symbols, and has very little effect on the algorithm, so we will have no supplementary explanation about it later.

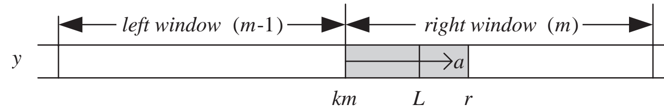

Fig. 3. Searching forward for the pattern suffixes with $SA(x)$ till there is no transition at a. Record next r in L when a terminal state is reached.

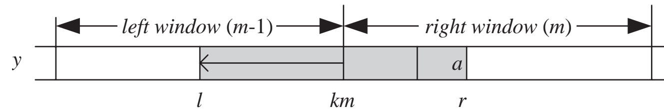

Fig. 4. Searching backward for corresponding prefixes with $RPA(x)$ till no more occurrence exists. Output an occurrence at l when the terminal state is reached.

```
WW(x = x_1 x_2 ··· x_m, y = y_1 y_2 ··· y_n)
 1.   Preprocessing
 2.       Build SA(x) and RPA(x)
 3.   Search
 4.       For k ∈ 1 ··· ⌊n/m⌋ do
 5.           R ← 0, L ← 0, r ← 0, l ← 0
 6.           While R ≠ null do
 7.               R ← δ_SA(R, y_{km+r})
 8.               r ← r + 1
 9.               If R is terminal then L ← r
10.           End of while
11.           While L > l do
12.               If L is terminal then
13.                   report an occurrence at km − l
14.               End of if
15.               l ← l + 1
16.               If l = m then break
17.               L ← δ_RPA(L, y_{km−l})
18.           End of while
19.       End of For
```

Fig. 5. The pseudo-code of WW.

left window from right to left, it outputs in reverse order when more than one occurrence of pattern exists in the window.

The pseudo-code of WW is shown in Fig. 5. We note $\delta_{SA}(p, c)$ the transition function of $SA(x)$. $\delta_{SA}(p, c)$ is the node that we reach if we move along the edge labeled by c from the node p. If such an edge does not exist, $\delta_{SA}(p, c)$ is null. Also we note $\delta_{RPA}(q, c)$ the transition function of $RPA(x)$. Some optimizations and boundary checks done on the real code are not shown for clarity.

3.2. Search example

To illustrate WW , we give an example by searching all occurrences of the pattern *aabbabb* in the text *ababababababaabbabba*.

We first build $SA(aabbabb)$ as Fig. 1 and $RPA(aabbabb)$ as Fig. 2, respectively. We note the current window between square brackets, also the current maximal matched suffix between vertical lines, and inspected letters underlined. We begin with $[ababab||S_0abababa]$ $abbabba$, where S_q (P_q) denotes that *SA* (*RPA*) is running in state q, and reading the symbol next to S_q (P_q).

(1) $[ababab||S_0abababa]abbabba \Longrightarrow$
$[ababab||\underline{a}S_1bababa]abbabba \Longrightarrow$
$[ababab||\underline{ab}S_{3'}ababa]abbabba \Longrightarrow$
$[ababab||\underline{aba}S_{null}baba]abbabba$. Because the matched suffix is ε, the algorithm shifts to next window directly. We search again:

(2) $ababab a[bababa||S_0abbabba] \Longrightarrow$
$abababa[bababa||\underline{a}S_1bbabba] \Longrightarrow$
$abababa[bababa||\underline{ab}S_{3'}babba] \Longrightarrow$
$abababa[bababa|\underline{abb}|S_{4'}abba] \Longrightarrow$
$abababa[bababa|\underline{abb}|\underline{a}S_5bba] \Longrightarrow$
$abababa[bababa|\underline{abb}|\underline{ab}S_6ba] \Longrightarrow$
$abababa[bababa|\underline{abbabb}|S_7a] \Longrightarrow$
$abababa[bababa|\underline{abbabb}|\underline{a}S_{null}]$. The maximal matched suffix is *abbabb*, and its corresponding state in *RPA* is 6. We resume the scan:

(3) $abababa[babab P_6a|\underline{abbabb}|a] \Longrightarrow$
$abababa[baba P_7b|\underline{aabbabb}|a]$. Because 7 is a terminal state, output an occurrence at current position 13. In addition, the shortest (non-ε) path from current to terminal state is of length 7, and only 5 symbols remain in the window. There are no more occurrences in current window. Backward scanning stage ends. And the algorithm ends too.

Hence, WW reads 11 symbols totally in the text and reports a match at 13.

4. Theoretical analysis

4.1. Theoretical verification

Theorem 1. *WW can only find all occurrences of the pattern in the text.*

Proof. Firstly, we show that all occurrences in the window can be found in each attempt. Based on the definition of *SA*, WW can recognize the position of the maximal matched suffix which includes x_{km} (i.e. recognize all suffixes) in the first stage. In the second stage, WW only scans the matched result from the first stage. Thus, in the window, it is impossible to match an occurrence which is not ended in current right window $x_{km}x_{km+1}\ldots x_{(k+1)m-1}$. In the second stage, WW just runs like a reversed DFA, whose correctness is self evident. Therefore, WW can match all the occurrences correctly in the window. On the other hand, according to the algorithm, though all the windows are overlapped, their right windows are

non-overlap and consecutive. So every occurrence in the text can always be found in only one attempt. □

WW scans the windows from left to right, but because of attempting m possible positions in parallel in one window, it outputs in reverse order when more than one occurrence of pattern exists in the window. This is different from traditional algorithms, which output positions in order. So WW is suitable for the applications that are not sensitive to the output sequence of occurrence positions, such as finding all the occurrences, checking the existence, or counting the occurrences of a pattern.

By making slight modification to the algorithm, we can adjust the output sequence of occurrence positions. During the second phase, we add a stack of size m (there are not more than m occurrences in one window). When the automaton reaches a terminal state, the algorithm pushes current scanning position into the stack instead of outputting it directly. When the second phase ends, WW pops and outputs all the elements in the stack one by one. In this way, it has the same output sequence as traditional algorithms. Because the modification has no effect on the time complexities of the algorithm, original algorithm is considered as follows for clarity.

4.2. Analysis of complexities

WW makes use of two automata: $SA(x)$ and $RPA(x)$, and their space complexities both are $O(m\sigma)$, then:

Proposition 1. *The space complexity of WW is* $O(m\sigma)$.

The preprocess of WW is mainly to construct two automata: $SA(x)$ and $RPA(x)$. Both their constructing time complexities are $O(m)$, therefore:

Proposition 2. *The preprocess time complexity of WW is* $O(m)$.

It is proved as follows that the worst, best and average time complexities of WW are all theoretically optimal.

Theorem 2. *The worst case time complexity of WW is* $O(n)$.

Proof. WW inspects every symbol once at most in every attempt window (of size $2m-1$). On the other hand, after each attempt, the algorithm will shift the window for a length of m fixedly, that is to say, the algorithm attempts only $\lfloor n/m \rfloor$ times. Consequently, the algorithm scans the text $(2m-1)\lfloor n/m \rfloor \leqslant 2n - n/m < 2n$ times at most. Hence, the worst case time complexity of WW is $O(n)$. □

The upper bound of scanning symbols is reached in the case of searching all occurrences of a^m in a^n.

Theorem 3. *The best case time complexity of WW is* $O(n/m)$.

Proof. If the symbols at all attempt positions are not included in the pattern, the algorithm will scan only one symbol per window. So the algorithm inspects $\lfloor n/m \rfloor$ symbols in all. Thus, the best case time complexity is $O(n/m)$. □

The lower bound of scanning $\lfloor n/m \rfloor$ symbols is reached in the case of searching all occurrences of b^m in a^n.

The average running time of string matching algorithm is generally analyzed in the situation where the text is random. The probability that a specified symbol occurs at any position in the text is $1/\sigma$. And this does not depend on the context of the symbol.

Theorem 4. *Under independent equiprobability condition, the average case time complexity of WW is* $O(n(\log_\sigma m)/m)$.

Proof. Firstly, count the average number of symbol inspections at each attempt.

Let $r = 2\lceil \log_\sigma m \rceil$. There are not less than m^2 possible values for the strings of length r. The pattern has not more than m substrings of length r. Illustrated with the window $y_{(k-1)m+1} \ldots y_{km} \ldots y_{(k+1)m-1}$, three cases are discussed as follows:

(1) *SA* inspects more than r symbols in the right window. Then the string $y_{km} \ldots y_{km+r-1}$ must be a substring of the pattern. The probability is less than $m/m^2 = 1/m$. In this case, the number of inspections is less than $2m$. (The worst behavior of the algorithm scanning all the $2m - 1$ symbols in the window.)

(2) *RPA* inspects more than r symbols in the left window. Then the string $y_{km-r+1} \ldots y_{km}$ must be a substring of the pattern too. The probability is less than $1/m$. The number of inspections is less than $2m$ too.

(3) Both *SA* and *RPA* inspect not more than r symbols. The number of inspections is bounded by $2r$. The probability is less than 1 of course.

The expected number of inspections at an attempt is thus less than

$$\frac{1}{m} \times (2m) + \frac{1}{m} \times (2m) + 1 \times (4 \log_\sigma m + 4) \tag{1}$$

which is $O(\log_\sigma m)$.

Since the algorithm attempts only $\lfloor n/m \rfloor$ times, the average case time complexity is $O(n(\log_\sigma m)/m)$. □

5. Experimental results

We ran extensive experiments on random and real-world texts in order to show how efficient our algorithm is in practice. The experiments were run on a Pentium III 933 MHz dual-processor computer with 1 GB of RAM, and a computer word of 32 bits, under Linux. We measured CPU time with gprof and turned on the compiler optimizations.

All the algorithms were implemented with an uniform I/O interface. The code of existing algorithms is from ESMAJ [3] with the exception of TNDM. We made our best coding effort

[3] http://www-igm.univ-mlv.fr/~lecroq/string/.

to implement all the algorithms. We compared the following algorithms.

- *KMP*: the famous Knuth–Morris–Pratt algorithm [15] has a linear worst case time complexity.
- *SO*: the more efficient variant of Shift-And algorithm [2] has a linear worst case time complexity provided that the pattern is not longer than the computer word.
- *BM*: the famous Boyer–Moore algorithm [3] is the first sublinear algorithm in the average [24].
- *QS*: the Quick Search algorithm [23] is very fast in practice for short patterns and large alphabets.
- *TBM*: the Tuned Boyer–Moore algorithm [14], which is an implementation of a simplified version of BM, is very fast in practice.
- *AS*: the Alpha Skip Search algorithm [5] uses buckets of positions for each factor of length $\log_\sigma m$ of the pattern.
- *RF*: the classical Reverse Factor Matching algorithm [11] is optimal in the average.
- *TRF*: the Turbo Reverse Factor Matching algorithm [9], which is a refinement of the Reverse Factor algorithm, is optimal both in the average and the worst.
- *BNDM*: the Backward Nondeterministic Dawg Matching algorithm [17] Which can be seen as a bit-parallelism version of RF, is optimal in the average provided that the pattern is not longer than the computer word.
- *TNDM*: the Two-way Nondeterministic Dawg Matching algorithm [19] is a two-way modification of BNDM. Their experiments show that this change of direction will decrease the number of the symbol inspections. But it just matches only one suffix before backward scan stage, this makes TNDM still a quadratic worst case time complexity algorithm.
- *WW*: Our Wide Window String Matching algorithm is also optimal both in the average and the worst. And our WW algorithm can be further improved by using bit-parallel technique too. Here we refer the bit-parallel WW as BWW.

The texts which we used were of size 10 MB, over which we searched for 5000 patterns. We ran experiments on random text with uniformly distributed alphabets of sizes 2, 4, 8, 16, 32, 64, 128 and 256. The patterns were randomly generated on the corresponding alphabet. While for real-world texts, we ran experiments on an English text (from the TREC Wall Street Journal collection) and a network traffic (from the 1999 DARPA Intrusion Detection Evaluation Data Set). The patterns were randomly selected from the corresponding text (at word beginnings in the case of English text). For random texts we searched for short (from 2 to 32) and long (multiples of 32 from 64 to 1024) patterns, while for real-world texts we searched for short patterns only, because English words and network signatures are usually shorter than 32.

To make the plots more readable, we rescale the y axes to the most interesting values. For example, KMP is often outside the range of interesting values(13–26 s/GB).

Figs. 6 and 7 shows the results for short and long patterns over random texts, respectively. The x-axis is the length of the patterns, and the y-axis shows the average running time in second per pattern per GB text of each algorithm. It needs to note that BWW, SO, BNDM and TNDM are only available for searching patterns shorter than the computer word.

For short patterns, WW and BWW are the most efficient algorithms for large alphabets. BWW is always among the most efficient algorithms.

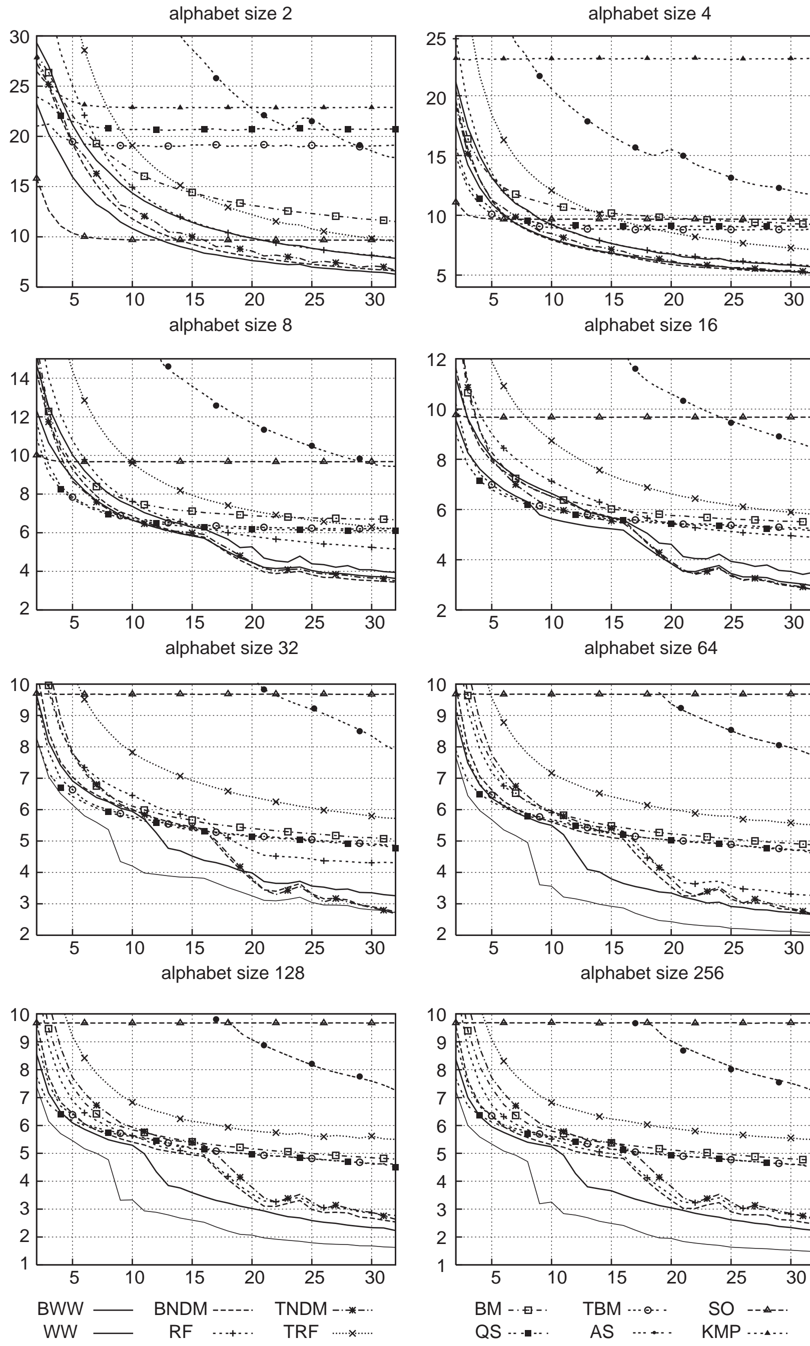

Fig. 6. Running time for random text and short pattern length, over different alphabet sizes.

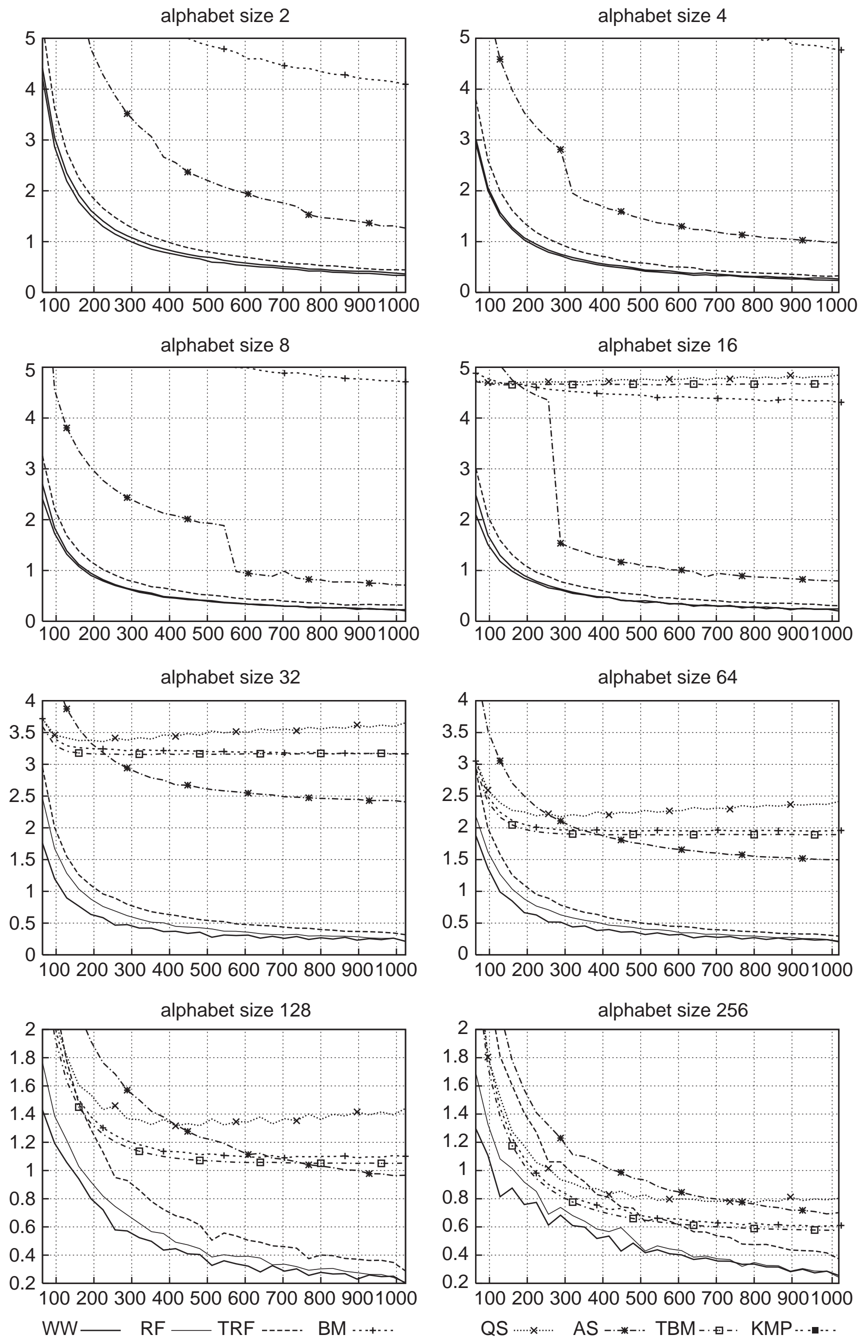

Fig. 7. Running time for random text and long pattern length, over different alphabet sizes.

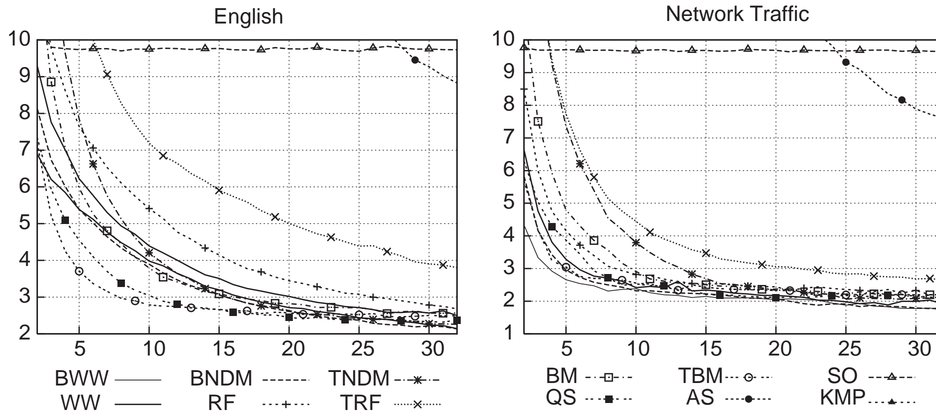

Fig. 8. Running time for real-world texts: English and network traffic.

For long patterns, WW is always the most efficient algorithm, especially for large alphabets. As the pattern length grows, the difference between WW and RF diminishes. The results are consistent with the fact that both WW and RF are theoretical optimal in average.

Among the worst case linear time algorithms: WW, TRF, KMP, SO, and BWW, BWW is the best one in most cases for searching short patterns. Only SO is better for very short pattern over very small alphabets. WW is the best one for searching long patterns.

Fig. 8 shows the results for English and network traffic texts. The results are very similar to random text for $\sigma = 16$ and $\sigma = 32$, respectively. That is, WW and BWW is reasonably competitive on English and network traffic.

6. Conclusions and future work

We present a new exact string matching algorithm called WW, which makes use of a wide window (of size $2m - 1$) to attempt m positions in parallel. WW divides the text into $\lfloor n/m \rfloor$ overlapping windows, in which the inspections consist of two phases. The algorithm scans the pattern suffixes from middle to right with the forward suffix automaton of the pattern, and then scans corresponding prefixes from middle to left with the reverse prefix automaton of the pattern. Theoretical analysis shows that WW has optimal time complexities in the worst, best and average cases: $O(n)$, $O(n/m)$ and $(O(n(\log_\sigma m)/m))$. Experimental comparison of WW with existing algorithms validates our theoretical claims for searching long patterns in average case time complexity. It further reveals that WW and its bit-parallel variant are very competitive for searching short patterns. Thus, WW not only suits for off-line pattern matching, but also fits in high-speed online pattern matching.

Bit-parallelism [2,17,19] is a general way to simulate nondeterministic automata using the bits of the computer word. We have combined the bit-parallel technique with our idea of wide window [13]. Since reverse suffix automaton has good applications to multiple string matching [21], applying forward suffix automaton to multiple string matching is worth

studying too. Each attempt of WW is completely independent from all the others so that it could be easy to get a parallel version of WW. As a new idea of string matching, wide window opens many problems for further research.

References

[1] C. Allauzen, M. Raffinot, Simple optimal string matching algorithm, J. Algorithms 36 (1) (2000) 102–116.
[2] R. Baeza-Yates, G.H. Gonnet, A new approach to text searching, Comm. ACM 35 (10) (1992) 74–82.
[3] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Comm. ACM 20 (10) (1977) 62–72.
[4] C. Charras, T. Lecroq, Handbook of Exact String Matching Algorithms, King's College London Publications, 2004.
[5] C. Charras, T. Lecroq, J.D. Pehoushek, A very fast string matching algorithm for small alphabets and long patterns, in: M. Farach-Colton (Ed.), Proc. of the 9th Ann. Symp. on Combinatorial Pattern Matching, Lecture Notes in Computer Science, Vol. 1448, Springer, Piscataway, NJ, USA, 1998, pp. 55–64.
[6] M. Crochemore, Transducers and repetitions, Theoret. Comput. Sci. 45 (1) (1986) 63–86.
[7] M. Crochemore, Off-line serial exact string searching, in: A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Oxford University Press, Oxford, 1997, pp. 1–53, (Chapter 1).
[8] M. Crochemore, Reducing space for index implementation, Theoret. Comput. Sci. 292 (1) (2003) 185–197.
[9] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, W. Rytter, Speeding up two string-matching algorithms, Algorithmica 12 (4/5) (1994) 247–267.
[10] M. Crochemore, C. Hancart, Automata for matching patterns, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, Vol. 2, Linear Modeling: Background and Application, Springer, Berlin, 1997, pp. 399–462 (Chapter 9).
[11] M. Crochemore, W. Rytter, Text algorithms, Oxford University Press, Oxford, 1994, 412pp.
[12] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, Singapore, 2002.
[13] L. He, B. Fang, Linear nondeterministic dawg string matching algorithm, in: A. Alberto, M. Massimo (Eds.), String Processing and Information Retrieval, 11th Internat. Symp. (SPIRE 2004), Lecture Notes in Computer Science, Vol. 3246, Springer, Padova, Italy, 2004, pp. 70–71.
[14] A. Hume, D. Sunday, Fast string searching, Software Pract. Exper. 21 (11) (1991) 1221–1248.
[15] D.E. Knuth, J.H. Morris, V.R. Pratt, Fast pattern matching in strings, SIAM J. Comput. 6 (2) (1977) 323–350.
[16] T. Lecroq, A variation on the Boyer–Moore algorithm, Theoret. Comput. Sci. 92 (1) (1992) 119–144.
[17] G. Navarro, M. Raffinot, Fast and flexible string matching by combining bit-parallelism and suffix automata, ACM J. Exp. Algorithmics (JEA) 5 (4) (2000) 1–36.
[18] G. Navarro, M. Raffinot, Flexible Pattern Matching in Strings—Practical On-line Search Algorithms for Texts and Biological Sequences, Cambridge University Press, Cambridge, 2002.
[19] H. Peltola, J. Tarhio, Alternative algorithms for bit-parallel string matching, in: M.A. Nascimento, E.S. de Moura, A.L. Oliveira (Eds.), Proc. 10th Internat. Symp. on String Processing and Information Retrieval (SPIRE'03), Lecture Notes in Computer Science, Vol. 2857, Springer, Manaus, Brazil, 2003, pp. 80–94.
[20] M. Raffinot, Asymptotic estimation of the average number of terminal states in dawgs, in: R. Baeza-Yates (Ed.), Proc. 4th South American Workshop on String Processing, Carleton University Press, Valparaíso, Chile, 1997, pp. 140–148.
[21] M. Raffinot, On the multi backward dawg matching algorithm (MultiBDM), in: R. Baeza-Yates (Ed.), Proc. 4th South American Workshop on String Processing, Carleton University Press, Valparaíso, Chile, 1997, pp. 149–165.
[22] W.F. Smyth, Computing Patterns in Strings, Pearson Addison Wesley, 2003.
[23] D.M. Sunday, A very fast substring search algorithm, Comm. ACM 33 (8) (1990) 132–142.
[24] T.-H. Tsai, Average case analysis of the boyer–moore algorithm, in: http://www.stat.sinica.edu.tw/chonghi/stat.htm, 2003.
[25] A.C.C. Yao, The complexity of pattern matching for a random string, SIAM J. Comput. 8 (3) (1979) 368–387.