Three Strategies for the Dead-Zone String Matching Algorithm

Jacqueline W. Daykin[1,2,3,5], Richard Groult[4,3], Yannick Guesnet[3], Thierry Lecroq[3], Arnaud Lefebvre[3], Martine Léonard[3], Laurent Mouchard[3], Élise Prieur-Gaston[3], and Bruce Watson[5,6]

[1] Department of Computer Science, Aberystwyth Univ., Wales & Mauritius
[2] Department of Informatics, Kings College London, UK
[3] Normandie Univ., UNIROUEN, LITIS, 76000 Rouen, France
[4] Modélisation, Information et Systèmes (MIS), Univ. de Picardie Jules Verne, Amiens, France
[5] Department of Information Science, Stellenbosch Univ., South Africa
[6] CAIR, CSIR Meraka, Pretoria, South Africa

Abstract. Online exact string matching consists in locating all the occurrences of a pattern in a text where only the pattern can be preprocessed. Classical online exact string matching algorithms scan the text from start to end through a window whose size is equal to the pattern length. Exact string matching algorithms from the dead-zone family first locate the window in the middle of the text, compare the content of the window and the pattern and then recursively apply the same procedure on the left part and on the right part of the text while possibly excluding some parts of the text. We propose three different strategies for performing the symbol comparisons and, we compute the shifts for determining the left and right parts of the text at each recursive call.

Keywords: exact string matching algorithms, dead-zone strategy, online, recursion

1 Introduction

The string matching problem consists in finding one or, more usually, all the occurrences of a pattern $x = x[0\,..\,m-1]$ of length m in a text $y = y[0\,..\,n-1]$ of length n. It can occur for instance in information retrieval, bibliographic search and molecular biology. It has been extensively studied and numerous techniques and algorithms have been designed to solve this problem (see [9,3,5,6]). We are interested here in the problem where the pattern is given first and can then be searched in various texts. Thus a preprocessing phase is only allowed on the pattern.

Most string matching algorithms use a window to scan the text. The size of this window is equal to the length of the pattern. They first align the left ends of the window and the text. Then they check if the pattern occurs in the window (this specific work is called an *attempt*) and they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. String matching algorithms mostly differ in the way they compare the pattern and the window content, in the way they compute the length of the shifts and in the quantity of information they store from one attempt to the other, leading to a great number of algorithms.

As early as 1997 a new family of algorithms has been designed that do not fit in the sliding window strategy: it consists in first locating the window in the middle of the text, performing an attempt and then recursively applying the same procedure on the left part and on the right part of the text, while possibly excluding some parts of the text giving the "dead-zone" method [12,10,11,8]. Algorithms from this family are highly parallelizable.

This strategy has not attracted much attention. To address this, here we present three different methods for performing the symbol comparisons during the attempts and for computing the lengths of the shifts.

The remainder of the paper is organized as follows: in Section 2 we give the formal notions and notation used throughout the paper. Section 3 presents the basic techniques used for online string matching and recalls the famous Knuth-Morris-Pratt and Boyer-Moore algorithms. In the next three sections we give new strategies for the dead-zone method: a right-to-left memoryless strategy in Section 4, a right-to-left strategy with memory in Section 5 and an alternating strategy in Section 6. Finally we give our conclusion and perspectives in Section 7.

2 Notation

Consider a finite totally ordered alphabet Σ of constant size $|\Sigma| = \sigma$ which consists of a set of symbols. A *string* is a sequence of zero or more symbols over Σ. The set of all non-empty strings over Σ is denoted by Σ^+. The empty string is the null sequence of symbols (hence of zero length) and is denoted by ε; furthermore, $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$.

A string x over Σ of length $|x| = m$ is represented by $x[0\,..\,m-1]$, where $x[i] \in \Sigma$ for $0 \leq i \leq m-1$ is the $(i+1)$-th symbol of x.

The reverse of a string x (i.e. the string read from right to left) is denoted by $\tilde{x}$.

The concatenation of two strings x and y is defined as the sequence of symbols of x followed by the sequence of symbols of y and is denoted by $x \cdot y$ or simply xy when no confusion is possible. A string x is a *factor* of y if $y = uxv$, where $u, v \in \Sigma^*$; specifically a string $x = x[0\,..\,m-1]$ is a factor of y if $x[0\,..\,m-1] = y[j\,..\,j+m-1]$ for some j.

Strings $u = y[0\,..\,i]$ are called *prefixes* of y, and strings $v = y[i\,..\,m-1]$ are called *suffixes* of y. The prefix u (respectively, suffix v) is a proper prefix (suffix) of a string y if $y \neq u, v$.

A border of a non-empty string x is a factor u of x which is both a proper prefix and a proper suffix of x. We denote by $\mathit{border}(x)$ the longest border of x and by $\mathit{per}(x) = |x| - \mathit{border}(x)$ the smallest period of x.

For $0 \leq i \leq m-1$:

- $\mathit{pref}[i]$ = length of the longest common prefix of x and $x[i\,..\,m-1]$;
- $\mathit{suff}[i]$ = length of the longest common suffix of x and $x[0\,..\,i]$.

Given a string x of length m, the values $\mathit{border}(x)$ and $\mathit{per}(x)$ and arrays *pref* and *suff* can be computed in $O(m)$ time (see [4]).

We are interested in finding all the occurrences of a pattern $x = x[0\,..\,m-1]$ of length $|x| = m$ in a text $y = y[0\,..\,n-1]$ of length $|y| = n$. We will apply the Dead-Zone strategy.

3 Background on online exact string matching algorithms

Exact string matching consists in finding one, or more generally, all the occurrences of a string x (usually called a *pattern* in the literature) of length m in a *text* y of length n. In its online version only the pattern x can be preprocessed before the searching phase.

3.1 Sliding window mechanism

Online exact string matching algorithms work as follows. They scan the text with the help of a *window* whose size is generally equal to m. They first align the left ends of the window and the text, then compare the symbols of the window with the symbols of the pattern — this specific work is called an *attempt* — and after a whole match of the pattern or after a mismatch they *shift* the window to the right. They repeat the same procedure again until the right end of the window goes beyond the right end of the text. This mechanism is usually called the *sliding window mechanism.* We associate each attempt with the positions j and $j+m-1$ in the text when the window is positioned on $y[j\,..\,j+m-1]$: we say that the attempt is at the left position j and at the right position $j+m-1$.

3.2 Knuth-Morris-Pratt algorithm

Consider an attempt at a left position j, that is when the window is positioned on the text factor $y[j\,..\,j+m-1]$. Assume that the first mismatch occurs between $x[i]$ and $y[i+j]$ with $0 < i < m$. Then, $x[0\,..\,i-1] = y[j\,..\,i+j-1] = u$ and $a = x[i] \neq y[i+j] = b$. When shifting, it is reasonable to expect that a prefix v of the pattern matches some suffix of the portion u of the text. Moreover, if we want to avoid another immediate mismatch, the symbol following the prefix v in the pattern must be different from a. The longest such prefix v is called the tagged border of u (it occurs at both ends of u followed by different symbols in x). This introduces the notation: let $bp_x[i]$ be the length of the longest border of $x[0\,..\,i-1]$ followed by a symbol c different from $x[i]$ and -1 if no such tagged border exists, for $0 < i \leq m$. Then, after a shift, the comparisons can resume between symbols $x[bp_x[i]]$ and $y[i+j]$ without missing any occurrence of x in y, and avoiding a backtrack on the text. The table bp_x can be computed in $O(m)$ space and time (see [4]) before the searching phase, applying the same searching algorithm to the pattern itself, as if $x = y$.

The best prefix array bp_x for x with $m+1$ elements is defined as follows for $0 \leq i \leq m$:

$$bp_x[i] = \begin{cases} -1 & \text{if } i = 0 \\ |border(x[0\,..\,i-1])| & \text{if } x[|border(x[0\,..\,i-1])|] \neq x[i] \\ bp_x[|border(x[0\,..\,i-1])|] & \text{otherwise.} \end{cases}$$

3.3 Boyer-Moore algorithm

The Boyer-Moore algorithm scans the symbols of the pattern from right to left beginning with the rightmost one. In the case of a mismatch (or a complete match of the whole pattern) it uses two precomputed functions to shift the window to the right. These two shift functions are called the *good suffix shift* (also called matching shift) and the *bad character shift* (also called occurrence shift).

Assume that a mismatch occurs between the character $x[i] = a$ of the pattern and the character $y[i+j] = b$ of the text during an attempt at the left position j. Then, $x[i+1\,..\,m-1] = y[i+j+1\,..\,j+m-1] = u$ and $x[i] \neq y[i+j]$. The good suffix shift consists in aligning the segment $y[i+j+1\,..\,j+m-1] = x[i+1\,..\,m-1]$ with its rightmost occurrence in x that is preceded by a symbol different from $x[i]$. If no such segment exists, the shift consists in aligning the longest suffix v of $y[i+j+1\,..\,j+m-1]$ with a matching prefix of x.

Formally we define two conditions, for $0 \le i \le m-1$ and $1 \le d \le m$ as follows. The suffix condition $\mathit{suffCond}_x$:

$$\mathit{suffCond}_x(i,d) = \begin{cases} 0 < d \le i+1 \text{ and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x \\ \text{or} \\ i+1 < d \text{ and } x[0\,..\,m-d-1] \text{ is a suffix of } x. \end{cases}$$

The occurrence condition $\mathit{occCond}_x$:

$$\mathit{occCond}_x(i,d) = \begin{cases} 0 < d \le i \text{ and } x[i-d] \ne x[i] \\ \text{or} \\ i < d. \end{cases}$$

Then the good suffix array for x is defined as follows, for $0 \le i \le m-1$:

$$gs_x[i] = \min\{d \mid \mathit{suffCond}_x(i,d) \text{ and } \mathit{occCond}_x(i,d) \text{ are satisfied }\}.$$

The bad character shift consists in aligning the text symbol $y[i+j]$ with its rightmost occurrence in $x[0\,..\,m-2]$. If $y[i+j]$ does not occur in the pattern x, no occurrence of x in y can overlap with $y[i+j]$, and the left end of the window is aligned with the symbol immediately after $y[i+j]$, namely $y[i+j+1]$.

Note that the bad character shift can be negative, thus for shifting the window, the Boyer-Moore algorithm applies the maximum between the good suffix shift and the bad character shift.

4 Right-to-left searching strategy for the Dead-Zone method

One searching strategy for the Dead-Zone method consists in scanning the symbols of the window from right to left in the same manner as in the Boyer-Moore algorithm [2]. Then, when a mismatch occurs or when an occurrence of the pattern is found, a right shift and a left shift has to be applied in order to compute the right and left parts where the recursion will apply. The general situation is the following: a suffix v of the pattern has been matched in the text and a mismatch occurred between a symbol a at position i in the pattern and a symbol d in the text. The right shift consists in finding a re-occurrence of v in the pattern preceded by a symbol b different from a, and the left shift consists in finding the longest suffix v' of the pattern preceded by a symbol c different from a (see Figure 1).

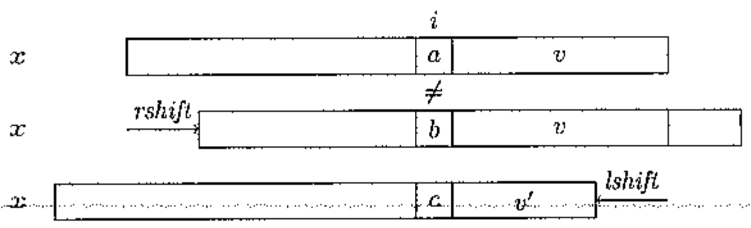

Figure 1. When a suffix v of the pattern has been matched in the text and a mismatch occurred between a symbol a at position i in the pattern. The right shift (*rshift*) consists in finding a re-occurrence of v in the pattern preceded by a symbol b different from a, and the left shift (*lshift*) consists in finding the longest suffix v' of the pattern preceded by a symbol c different from a.

The right shift can be computed by the good suffix array of the Boyer-Moore algorithm while the left shift can be computed by the best prefix array of the Knuth-Morris-Pratt algorithm [7] for the reverse pattern (see [4]).

Specifically, when a mismatch occurs at position i of the pattern the right shift is given by $gs_x[i]$ and the left shift is given by $m-1-i-bp_{\tilde{x}}[m-1-i]$ (where $bp_{\tilde{x}}$ is the best prefix array for $\tilde{x}$ the reverse of x). Algorithm DZ-R2L below implements this strategy. Parameters b and e are respectively the beginning and the end positions on the text and $C \geq 1$ is any small constant that will enable the recursion to stop. When the recursion stops the search is done by a brute force algorithm BF.

```
DZ-R2L($x, m, y, b, e$)
 1  if $e - b < Cm$ then
        ▷ Stop the recursion
 2      BF($x, m, y, b, e$)
 3  else $j \leftarrow (e + b + 1)/2 - m/2$
 4      $i \leftarrow m - 1$
        ▷ Scan
 5      while $i \geq 0$ and $x[i] = y[j + i]$ do
 6          $i \leftarrow i - 1$
 7      if $i < 0$ then
 8          OUTPUT($j$)
 9          $rshift \leftarrow per(x)$
10          $lshift \leftarrow per(x)$
11      else $rshift \leftarrow gs_x[i]$
12          $lshift \leftarrow m - 1 - i - bp_{\tilde{x}}[m - 1 - i]$
        ▷ Left recursive call
13      DZ-R2L($x, m, y, b, j + m - 1 - lshift$)
        ▷ Right recursive call
14      DZ-R2L($x, m, y, j + rshift, e$)
```

The following theorem can be easily proved.

Theorem 1. *The algorithm* DZ-R2L$(x, m, y, 0, n-1)$ *finds all the occurrences of* x *in* y *and runs in* $O(mn)$ *time.*

5 Right-to-left strategy with memory for the Dead-Zone method

For the Dead-Zone method, it is possible to implement a strategy with memory simi-lalry to the Apostolico-Giancarlo (AG) algorithm [1] for the Boyer-Moore algorithm. The AG algorithm stores for each rightmost position of the window the length ℓ of the longest suffix of the pattern ending at that position. Then if, during the right-to-left scan of an attempt, it reaches a position where this length ℓ is non-null, the algorithm can make a decision and end the attempt without any further symbol comparisons as in most cases, otherwise it can jump over the factor of the text of length ℓ. In that case, symbols of the text are positively compared only once.

The main difference here is that for the AG algorithm there is only one window and we may access only the rightmost position of a previously matched suffix of the pattern, while for the Dead-Zone method we may access any symbol inside a previously matched suffix of the pattern. Here, during an attempt at left position j when a mismatch occurs with position i of the pattern or $i = -1$ because an occurrence of the pattern has been found, when a symbol at a position k of the pattern x is matched with a symbol at position $j+k$ of the text y during the attempt we need to store, for this position $j+k$, the position k and the length $k-i$. These items of information are stored in arrays $skip_1$ and $skip_2$ respectively. Because when a match occurs at position k, the mismatch position i is not known yet, a stack is used during each attempt for storing all the matching positions. Then at the end of the attempt these positions are popped in order to fill the corresponding values of arrays $skip_1$ and $skip_2$.

Note that during an attempt at left position j, if a position $i+j$ is reached such that $k = skip_2[i+j] > 0$, it means that $x[k-\ell+1\mathinner{..}k] = y[i+j-\ell+1\mathinner{..}i+j]$ with $\ell = skip_1[i+j]$, and furthermore $x[k-\ell] \neq y[i+j-\ell]$ if $k \geq \ell$. We need to know whether $y[i+j-\ell+1\mathinner{..}i+j] = x[i-\ell+1\mathinner{..}i]$ and thus we need to know whether $x[k-\ell+1\mathinner{..}k] = x[i-\ell+1\mathinner{..}i]$: this information can be obtained by computing the longest common prefix of the suffixes of $\tilde{x}$ starting at positions $m-1-k$ and $m-1-i$. This can be done in constant time by computing the suffix array and the longest common prefixes (LCP) array of $\tilde{x}$ and preparing the latter for answering Range Minimum Queries (RMQ).

Then, let q be the length of the longest common prefix of $\tilde{x}[m-1-k\mathinner{..}m-1]$ and $\tilde{x}[m-1-i\mathinner{..}m-1]$. If $q = \ell$ then $x[k-\ell+1\mathinner{..}k] = x[i-\ell+1\mathinner{..}i]$ and if $k \geq \ell$ it also holds that $x[k-\ell] \neq x[i-\ell]$ then $y[i+j-\ell+1\mathinner{..}i+j] = x[i-\ell+1\mathinner{..}i]$ and $x[i-\ell]$ and $y[i+j-\ell]$ need to be compared.

Algorithm DZ-R2L-WITH-MEMORY below implements this strategy.

```
DZ-R2L-WITH-MEMORY$(x, m, y, b, e)$
 1 if $e - b < Cm$ then
      ▷ Stop the recursion
 2    BF$(x, m, y, b, e)$
 3 else $(i, j) \leftarrow (m-1, (e+b+1)/2 - m/2)$
 4    $S \leftarrow \emptyset$
      ▷ Scan
 5    while $i \geq 0$ do
 6       $(k, \ell) \leftarrow (skip_1[i+j], skip_2[i+j])$
 7       if $\ell > 0$ then
 8          $q \leftarrow$ RMQ$(SA_{\tilde{x}}, m-1-k, m-i-1)$
 9          if $\ell \neq q$ then
10             $i \leftarrow i - \min\{\ell, d\}$
11             break
12          else $i \leftarrow i - q$
13       else if $x[i] = y[i+j]$ then
14          PUSH$(S, i)$
15          $i \leftarrow i - 1$
16       else break
17    while $S \neq \emptyset$ do
18       $k \leftarrow$ POP$(S)$
19       $(skip_1[j+k], skip_2[j+k]) \leftarrow (k, k-i)$
20    if $i < 0$ then
21       OUTPUT$(j)$
22       $(rshift, lshift) \leftarrow (per(x), per(x))$
23    else $(rshift, lshift) \leftarrow (gs_x[i], m-1-i-bp_{\tilde{x}}[m-1-i])$
      ▷ Left recursive call
24    DZ-R2L-WITH-MEMORY$(x, m, y, b, j+m-1-lshift)$
      ▷ Right recursive call
25    DZ-R2L-WITH-MEMORY$(x, m, y, j+rshift, e)$
```

We can conjecture that, as with the AG algorithm, the DZ-R2L-WITH-MEMORY algorithm runs in linear time in the worst case.

6 Alternating searching strategy: right – left for the Dead-Zone method

In order to try to optimize the lengths of the right and left shifts, it is possible to design an alternating strategy for the Dead-Zone algorithm. At each attempt, the window is placed is the middle of the text. The scanning is performed by comparing alternately the right and the left ends of the window until a complete match or a mismatch occurs. After each attempt two recursive calls are performed on the left and on the right parts of the text.

Then, at each attempt, a mismatch can occur either on the left or on the right and accordingly a shift has to be computed in the left and in the right which leads to 4 shift functions stored in 4 arrays:

- right shift after a right mismatch stored in array *rsrm*;
- left shift after a right mismatch stored in array *lsrm*;

– right shift after a left mismatch stored in array *rslm*;
– left shift after a left mismatch stored in array *lslm*.

Let us define two conditions $\mathit{suffCond}'_x$ and $\mathit{occCond}'_x$ as follows. For $0 \le i \le m-1$ and $1 \le d \le m$,

$$\mathit{occCond}'_x(i,d) = (0 < d \le i \text{ and } x[i-d] \ne x[i]) \text{ or } (i < d)$$

$$\begin{aligned}\mathit{suffCond}'_x(i,d) = (\ & 0 < d \le m-2-i \\ & \text{and } x[0\,..\,m-2-i-d] = x[d\,..\,m-2-i] \\ & \text{and } x[i-d+1\,..\,m-d-1] = x[i+1\,..\,m-1]\ ) \\ \text{or} \\ (\ & m-2-i < d \le i+1 \\ & \text{and } x[i-d+1\,..\,m-d-1] = x[i+1\,..\,m-1]\ ) \\ \text{or} \\ (\ & i+1 < d \text{ and } x[0\,..\,m-d-1] = x[d\,..\,m-1]\ )\end{aligned}$$

The latter can be rewritten as

$$\begin{aligned}\mathit{suffCond}'_x(i,d) = (\ & 0 < d \le m-2-i \\ & \text{and } x[d\,..\,m-2-i] \text{ is a prefix of } x \\ & \text{and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x\ ) \\ \text{or} \\ (\ & m-2-i < d \le i+1 \\ & \text{and } x[i-d+1\,..\,m-d-1] \text{ is a suffix of } x\ ) \\ \text{or} \\ (\ & i+1 < d \text{ and } x[0\,..\,m-d-1] \text{ is a suffix of } x\ ).\end{aligned}$$

Then the array *rsrm* can be defined as follows for $0 \le i \le m-1$:
$\mathit{rsrm}[i] = \min\{d \mid \mathit{occCond}'_x(i,d) \text{ and } \mathit{suffCond}'_x(i,d) \text{ are satisfied}\}$.
Three cases can arise as shown in Figure 2.

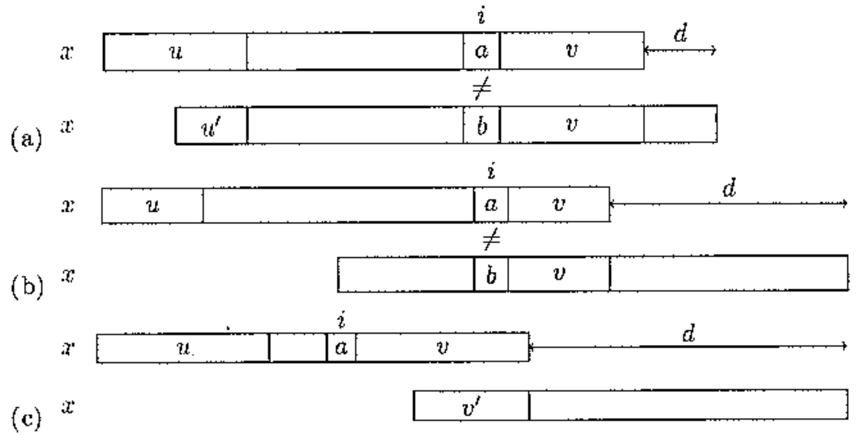

Figure 2. Right shift after a right mismatch: assume that prefix u and suffix v (of the same length) of x match the text and that a mismatch occurs with symbol a at position i of x: (a) in this case the suffix v of x reoccurs preceded by a symbol b different from a and a prefix u' of x matches a suffix of u; (b) in this case only a suffix v of x reoccurs preceded by a symbol b different from a; (c) in this case only a prefix v' of x matches a suffix of v.

Arrays *lsrm*, *rslm* and *lslm* can be defined similarly to array *rsrm*. Then algorithm DZ-ALT given below implements the alternating strategy.

```
DZ-ALT$(x, m, y, b, e)$
 1 if $e - b < Cm$ then
      ▷ Stop the recursion
 2    BF$(x, m, y, b, e)$
 3 else $j \leftarrow (e + b + 1)/2 - m/2$
 4    $i \leftarrow 0$
 5    while $i \leq m/2$ do
         ▷ Right scan
 6       if $x[m-1-i] \neq y[j+m-1-i]$ then
 7          $rshift \leftarrow rsrm[m-1-i]$
 8          $lshift \leftarrow lsrm[m-1-i]$
 9          break
         ▷ Left scan
10       if $x[i] \neq y[j+i]$ then
11          $rshift \leftarrow rslm[i]$
12          $lshift \leftarrow lslm[i]$
13          break
14       $i \leftarrow i + 1$
15    if $i > m/2$ then
16       OUTPUT$(j)$
17       $rshift \leftarrow per(x)$
18       $lshift \leftarrow per(x)$
      ▷ Left recursive call
19    DZ-ALT$(x, m, y, b, j + m - 1 - lshift)$
      ▷ Right recursive call
20    DZ-ALT$(x, m, y, j + rshift, e)$
```

6.1 Right Shift after a Right Mismatch

We will now show how to compute the array *rsrm* efficiently.

Lemma 2. *For* $0 \leq i \leq m-1$: $rsrm[i] \leq m - suff[j]$ *where* $j = \max\{k \mid 0 \leq k < m-1-i$ *and* $suff[k] = k+1\}$.

Proof. See Figure 3. We will show that $occCond'_x(j, d)$ and $suffCond'_x(j, d)$ are both satisfied with $j = i$ and $d = m - k - 1$. If $0 \leq k < m - 1 - i$ then $i < m - k - 1$. If $suff[k] = k + 1$ it means that $x[0\,..\,k]$ is a suffix of x. Then $occCond'_x(i, m-k-1)$ and $suffCond'_x(i, m-k-1)$ are both satisfied. Thus $rsrm[i] \leq m - k - 1$. □

The following two lemmas can be proved similarly.

Lemma 3. *For* $0 \leq i \leq m-1$: $rsrm[m-1-suff[i]] \leq m-1-i$ *if* $m-1-i \geq suff[i]$.

Lemma 4. *For* $0 \leq i \leq m-1$: $rsrm[m-1-suff[i]] \leq m-1-i$ *if* $m-1-i < suff[i]$ *and* $pref[m-1-i] \geq suff[i] - m + 1 + i$.

Then the following algorithm RSRM computes the array *rsrm*.

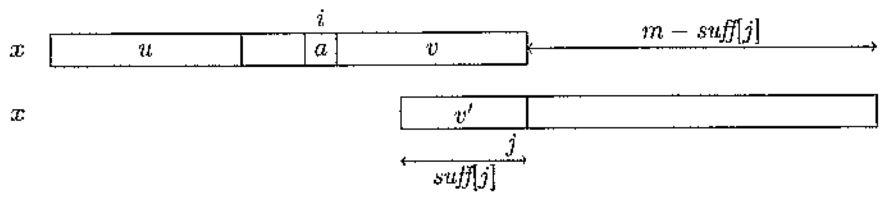

Figure 3. Right shift after a right mismatch: assume that prefix u and suffix v of x match the text and that a mismatch occurs with symbol a at position i of x then the length of the right shift cannot be larger than $m - suff[j]$.

```
RSRM($x, m, suff, pref$)
 1  $j \leftarrow 0$
    $\triangleright$ Lemma 2
 2  for $i \leftarrow m-2$ downto $-1$ do
 3     if $i = -1$ or $suff[i] = i+1$ then
 4        while $j < m-1-i$ do
 5           $rsrm[j] \leftarrow m-1-i$
 6           $j \leftarrow j+1$
    $\triangleright$ Lemmas 3 and 4
 7  for $i \leftarrow 0$ to $m-2$ do
 8     if $m-1-i \geq suff[i]$ or $pref[m-1-i] \geq suff[i]-m+1+i$ then
 9        $rsrm[m-1-suff[i]] \leftarrow m-1-i$
10  return $rsrm$
```

Lemma 5. *Algorithm* RSRM($x, m, suff, pref$) *is correct and runs in linear time.*

Proof. Correctness comes from Lemmas 2–4. Time complexity analysis is similar to the analysis of the good suffix array (see [4]). □

6.2 Right Shift after a Left Mismatch

Let *rpref* be the array *pref* for $\tilde{x}$ and let *rsuff* be the array *suff* for $\tilde{x}$. The following four lemmas can be proved similarly to Lemmas 2–4.

For $0 \leq i \leq m-1$:

Lemma 6. $rslm[i] < m - suff[j]$ *where* $j = \max\{k \mid 0 \leq k \leq i$ *and* $suff[k] = k+1\}$.

Lemma 7. $rslm[i] \leq \min\{m-j \mid suff[j] \geq m-i\}$.

Lemma 8. $rslm[m-1-i+rsuff[i]] \leq m-1-i$ *if* $rpref[m-1-i] \geq m-1-i+rsuff[i]$.

Lemma 9. $rslm[m-1-suff[i]] \leq m-1-i$ *if* $pref[m-1-i] \geq suff[i]+m-1-i$.

Then the following algorithm RSLM computes the array *rslm*.

```
RSLM($x, m, suff, rpref, rsuff$)
$j \leftarrow m/2 - 1$
for $i \leftarrow m - 2$ to $-1$ do
    if $i = -1$ or $suff[i] = i + 1$ then
        $rslm[j] \leftarrow m - 1 - i$
        $j \leftarrow j - 1$
$i \leftarrow 1$
$j \leftarrow m - 2$
while $i < m/2$ do
    while $j \geq 0$ and $suff[j] < i$ do
        $j \leftarrow j - 1$
    if $j < 0$ then
        break
    else while $i < m - j$ do
            $rslm[i - 1] \leftarrow m - 1 - j$
            $i \leftarrow i + 1$
        $j \leftarrow j - 1$
for $i \leftarrow 0$ to $m - 2$ do
    if $rpref[m - 1 - i] \geq rsuff[i] + m - 1 - i$ then
        $rslm[rsuff[i] + m - 1 - i] \leftarrow m - 1 - i$
return $rslm$
```

Lemma 10. *Algorithm* RSLM($x, m, suff, rpref, rsuff$) *is correct and runs in linear time.*

Proof. Similar to the proof of Lemma 5. □

Arrays *lsrm* and *lslm* can be computed in linear time using similar methods. Thus the preprocessing phase of algorithm DZ-ALT requires linear time.

7 Conclusion and perspectives

We presented three strategies for the dead-zone exact string matching method: a memoryless right-to-left strategy, a right-to-left with memory strategy and an alternating strategy. It remains to state the time complexity exactly for the right-to-left with memory strategy that we conjectured to be linear. That would be the first linear strategy designed for the dead-zone method. An experimental study will also be necessary to assess the practical performances of the different strategies both in terms of running times and the number of symbol comparisons. Also the fact that such algorithms are highly parallelizable should be exploited.

Acknowledgements

The first author was part-funded by the European Regional Development Fund through the Welsh Government.

References

1. A. Apostolico and R. Giancarlo: *The Boyer-Moore-Galil string searching strategies revisited.* SIAM J. Comput., 15(1) 1986, pp. 98–105.
2. R. S. Boyer and J. S. Moore: *A fast string searching algorithm.* Commun. ACM, 20(10) 1977, pp. 762–772.
3. C. Charras and T. Lecroq: *Handbook of exact string matching algorithms*, King's College London Publications, 2004.
4. M. Crochemore, C. Hancart, and T. Lecroq: *Algorithms on strings*, Cambridge University Press, 2007.
5. S. Faro and T. Lecroq: *The exact online string matching problem: A review of the most recent results.* ACM Comput. Surv., 45(2) 2013, pp. 13:1–13:42.
6. S. Faro, T. Lecroq, S. Borzi, S. D. Mauro, and A. Maggio: *The string matching algorithms research tool*, in Proceedings of the Prague Stringology Conference 2016, Prague, Czech Republic, August 29-31, 2016, J. Holub and J. Zdárek, eds., Department of Theoretical Computer Science, Faculty of Information Technology, Czech Technical University in Prague, 2016, pp. 99–111.
7. D. E. Knuth, J. H. Morris, Jr, and V. R. Pratt: *Fast pattern matching in strings.* SIAM J. Comput., 6(1) 1977, pp. 323–350.
8. M. Mauch, D. G. Kourie, B. W. Watson, and T. Strauss: *Performance assessment of dead-zone single keyword pattern matching*, in 2012 South African Institute of Computer Scientists and Information Technologists Conference, SAICSIT '12, Pretoria, South Africa, October 1-3, 2012, J. H. Kroeze and R. de Villiers, eds., ACM, 2012, pp. 59–68.
9. G. Navarro and M. Raffinot: *Flexible Pattern Matching in Strings – Practical on-line search algorithms for texts and biological sequences*, Cambridge University Press, 2002.
10. B. Watson and R. Watson: *A new family of string pattern matching algorithms.* South African Computer Journal, 30 2003, pp. 34–41.
11. B. W. Watson, D. G. Kourie, and T. Strauss: *A sequential recursive implementation of dead-zone single keyword pattern matching*, in Combinatorial Algorithms, 23rd International Workshop, IWOCA 2012, Tamil Nadu, India, July 19-21, 2012, Revised Selected Papers, S. Arumugam and W. F. Smyth, eds., vol. 7643 of Lecture Notes in Computer Science, Springer, 2012, pp. 236–248.
12. B. W. Watson and R. E. Watson: *A new family of string pattern matching algorithms*, in Proceedings of the Prague Stringology Club Workshop 1997, Prague, Czech Republic, July 7, 1997, J. Holub, ed., Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University, 1997, pp. 12–23.