DC: A highly efficient and flexible exact pattern-matching algorithm

Sérgio Deusdado and Paulo Carvalho

pmc@di.uminho.pt

Techn. Report DI-CCTC-09-10

2009, July

Computer Science and Technology Center
Departamento de Informática da Universidade do Minho
Campus de Gualtar – Braga – Portugal
http://cctc.di.uminho.pt/

DI-CCTC-09-10

DC: A highly efficient and flexible exact pattern-matching algorithm

by Sérgio Deusdado and Paulo Carvalho

Abstract

Aware of the need for faster and flexible searching algorithms in fields such as web searching or bioinformatics, we propose DC - a high-performance algorithm for exact pattern matching. Emphasizing the analysis of pattern peculiarities in the pre-processing phase, the algorithm encompasses a novel search logic based on the examination of multiple alignments within a larger window, selectively tested after a powerful heuristic called compatibility rule is verified. The new algorithm's performance is, on average, above its best-rated competitors when testing different data types and using a complete suite of pattern extensions and compositions. The flexibility is remarkable and the efficiency is more relevant in quaternary or greater alphabets.

Keywords: exact pattern-match, searching algorithms.

DC: A highly efficient and flexible exact pattern-matching algorithm

Sérgio Deusdado[a] Paulo Carvalho[b1]

[a]*Department of Exact Sciences, ESA, Polytechnic Institute of Bragança, Portugal*
[b]*Department of Informatics, Engineering School, University of Minho, Portugal*

Abstract

Aware of the need for faster and flexible searching algorithms in fields such as web searching or bioinformatics, we propose DC - a high-performance algorithm for exact pattern matching. Emphasizing the analysis of pattern peculiarities in the pre-processing phase, the algorithm encompasses a novel search logic based on the examination of multiple alignments within a larger window, selectively tested after a powerful heuristic called compatibility rule is verified. The new algorithm's performance is, on average, above its best-rated competitors when testing different data types and using a complete suite of pattern extensions and compositions. The flexibility is remarkable and the efficiency is more relevant in quaternary or greater alphabets.

Keywords: exact pattern-match, searching algorithms.

1. Introduction

Exact pattern-matching is a fundamental problem in computer science, the algorithm's efficiency is crucial in many search tasks present in everyday computation, such as exploring the web, an e-book or genomic data. An emergent implementation field for string matching algorithms application is network content security, including intrusion detection systems, anti-virus systems, and Web content filters.

Recent software and hardware advances have reduced the search response time to negligible values, however the exponential growth of available information has renewed the need for faster searching algorithms.

Basically, a pattern search algorithm intents to find all instances of a string-pattern p of length m in a text x of length n, being $n \geq m$. Strings p and x are built over a finite set of characters in a given alphabet $\sum$ of size σ.

A straightforward approach would analyze each character of the text as a possible initial character of the pattern and, for each attempt, summing positive comparisons till a failure or complete match. This basic approach, called Brute-Force algorithm, runs in $O(nm)$. Recent algorithms run substantially faster, in sub-linear time.

Most efficient algorithms operate in two stages or phases: the first phase includes the pre-processing or study of the pattern, being followed by the search or processing phase, where the text is objectively scanned by shifting the search window along it.

The key to achieve sub-linear performance is to pre-process the pattern in order to collect useful information to minimize redundant comparisons, boosting the detection of occurrences. The preliminary pre-processing phase deals exclusively with the pattern. The heuristics used in the next phase are based on pre-processed information, so its importance is enormous, even critical, because it may prevent spending redundant processing time in worthless tests. Subsequently, the processing phase, using the knowledge gathered in the previous phase, is devoted to iteratively identify exact matches in pattern-probable windows.

In this paper, we propose a new algorithm for exact pattern matching with a novel search logic, based on the examination of multiple alignments in a larger window, selectively tested after verified a compatibility rule. The window shift involves two cumulative components: a constant

[1] Corresponding author.
E-mail address: pmc@di.uminho.pt; Telephone:+351 253 604432; Fax: +351 253 604471

component, initially applied, safely repositions the window *m* characters ahead, and a variable component, cyclically incremented consulting the shift table, is used to shift the window in order to rapidly find a new occurrence of *p[m]*. Considering the first iteration, if the character *x[m]* does not appear in the pattern at all, it is safe to assume that no pattern occurrence will occur before *x[m+1]*, and in these conditions the shift value is maximal. The proposed algorithm extends this lemma to the next *m-1* characters since we can establish a window of *x[1..2m-1]* characters where a pattern *p[1..m]* does not exists if $x[m] \notin p[1..m]$. Otherwise, if $p[1..m] \supset x[m]$, up to *m* alignments of the pattern with *x[m]* are possible.

This paper is structured in five sections, including this introduction. In Section 2, existing pattern-matching algorithms are surveyed, given special attention to those we elect as references for performance comparison. In Section 3, the new algorithm is conceptually explained and its implementation described, concomitantly, a complete searching example is analyzed. The complexity analysis of the algorithm ends the section. In Section 4, the proposed algorithm's performance is assessed and compared with best-rated competitors providing empirical data. In Section 5, the results are discussed and the conclusions presented.

2. Survey on exact pattern-match algorithms

All representative pattern-matching algorithms scan the text iteratively. Each iteration comprises a window whose length equals the pattern length, thus each new window may contain zero or one pattern occurrence. The window is aligned with the pattern and each character involved is compared until a failure or a complete match occurs. The new window for the next iteration is initiated further in the text, and the shifted portion is safely ignored based on the results of the pre-processing phase. The cycle ends when the text has been integrally searched. Most efficient algorithms perform fewer comparisons and additively benefit from a simpler logic to evaluate the shift value for the next iteration.

From classic pattern-matching algorithms, KMP (Knuth-Morris-Pratt) [1] and BM (Boyer-Moore) [2] contributions improved significantly this nuclear computation recurrence in the late 1970s. Due to its performance outcome and proliferation, it is important to analyze BM features. The BM algorithm proceeds by sliding a search window of length *m* over the text. The text inside the window is checked against the pattern, from rightmost to leftmost character, and potentially encloses one pattern occurrence. If the whole window matches the pattern then a pattern replica has been discovered. A complete match or a character mismatch induces a window's shift. Two concurrent heuristics are used to shift the window, prevailing the best contribution facing the circumstantial conditions. The referred heuristics are pre-computed and the resulting shift values stored in the respective shift tables. In the search phase, facing the terminus of last attempt, the most advantageous shift value from the shift alternatives is used. The BM concurrent shift heuristics are:

Occurrence Heuristic: A failed comparison at any character in the window provides enough information to safely shift the search window. If the mismatched character is not part of the pattern, the next window could be moved just ahead of it, obtaining a maximal shift. If the mismatched character exists in the pattern then the next alignment is established between the mismatched character and its last occurrence in the pattern. If its last occurrence is *p[m]*, then its penultimate occurrence is considered. Thus, the shift value is the distance from the last occurrence of the mismatched character in *p[1..m-1]* to *p[m]*.

Match Heuristic: Being comparisons operated from right to left, if a partial or complete match is verified, a set of pattern characters are present in the text in the right sequence, respectively as a suffix or an occurrence. In both cases, it is possible to pre-compute a shift table containing the shift values applicable to all possible suffixes or the complete match of the pattern. This heuristic usually provides a better solution when shift limitations are present due to repetitive patterns.

Further investigation on BM searching originated several new versions. The most relevant are Horspool's variant [3] and Sunday's Quick Search algorithm [4], but many more are included in the BM family algorithms. BMH (Boyer-Moore-Horspool) is performance oriented and only uses the occurrence shift, applied to the last character in the search window. This is not always

the optimal choice but the exceptions, in general, do not justify the computation overhead. In factual performance evaluation, BMH stands even today, as referential for newer exact pattern search algorithms.

In [5] a simplified searching strategy based on a text partitioning scheme and constant window shifts was presented, leading to performance improvement comparatively with BM family algorithms. In this approach multiple alignments could occur per iteration.

Alternative pattern-matching algorithms use other approaches, such as suffix automata, bit parallelism or hashing. Representative examples are respectively, RF (Reverse Factor) [6], SO (Shift-Or) [7] and KR (Karp-Rabin) [8]. In fact, the number of comparisons required to complete the search task is lowered by some of these algorithms but the execution time performance is not always the best, due to a more complex logic to accomplish fewer comparisons.

Recent algorithms follow hybrid approaches with refinements, and incorporate the best features of past algorithms to achieve better performance, being FJS [9] a significant hybrid example of heuristic based algorithms, and BNDM [10] a significant hybrid example of bit parallelism and heuristic based algorithms.

FJS algorithm has adopted the main ideas of KMP and BM (resorting also to Sunday's contribution) in an effort to combine their best features. Algorithm FJS combines two well-known pattern-matching ideas:

(1) in accordance with the BM approach, FJS first compares *p[m]*, the rightmost character of the pattern, with the character in the corresponding text position *i*. If a mismatch occurs, a "Sunday shift" is implemented, moving *p* along *x* until the rightmost occurrence in *p* of the character $h = x[i+1]$ is positioned at $i+1$. At this new location, the rightmost character of *p* is again matched with the corresponding text position. Only when a match is found does FJS invoke the next (KMP) step; otherwise, another "Sunday shift" occurs;

(2) if $p[m] = x[i]$, KMP pattern-matching begins, starting (as KMP does) from the left-hand end *p[1]* of the pattern and, if no mismatch occurs, extending as far as $p[m-1]$. Then, whether or not a match for *p* is found, a "KMP shift" is eventually performed, followed by a return to step (1).

The Backward Nondeterministic DAWG Matching (BNDM) algorithm [10] has been developed from the backward DAWG matching (BDM) algorithm [11]. In the BDM algorithm, the pattern is preprocessed by forming a DAWG (directed acyclic word graph) of the reversed pattern. The text is processed in windows of size *m*. The window is searched for the longest prefix of the pattern from right to left with the DAWG. When this search ends, we have either found a match (i.e. the longest prefix is of length *m*) or the longest prefix. If a match was not found, the start position of the window can be shifted to the start position of the longest prefix. If a match was found we can shift on the second longest prefix (the longest one is the match we just found).

Further investigation in BNDM algorithm has produced a simplified and fastest version named SBNDM [12], without prefix searching. The advantage of bit-parallelism algorithms stems from fast bit operations in machine words. Since these algorithms need 1 bit per character, 32 bits and 64 bits architectures are very restrictive as greater patterns are very common in emergent pattern searching applications. It is possible to search long patterns using bit-parallelism by splitting the pattern and reusing the algorithm for the necessary machine words to cover the entire pattern, but the performance is penalized.

Recently, in [13], Lecroq proposed an adaptation of the Wu and Manber [14] multiple string matching algorithm to single string matching algorithm, including a new search strategy based on hashing *q*-grams. Experimental results showed state-of-the-art results for short patterns on small alphabets, however the presented versions take advantage on using *q*-grams with $3 \leq q \leq 8$.

Considering the specificity of biological sequences searching, we have proposed GRASPm - Genomic-oriented Rapid Algorithm for String Pattern-match [15], a novel algorithm, 2-grams based, that introduces an innovative searching strategy, allowing multiple alignments examination within a window, taking as reference the central duplet and verifying a compatibility rule that inhibits incompatible alignments previewed in the pre-processing phase. Additionally, uses a cumulative shift rule that includes a default and constant shift value plus a variable shift value, pre-processed similarly to the BM occurrence heuristic but duplet based.

Although GRASPm was conceived to search genomic data, in practice, it is an ultra-fast exact-pattern matching algorithm for small alphabets with $\sigma \leq 4$.
In order to benchmark the performance of the proposed algorithm, for simplicity reasons, all comparisons will be confined to four reference searching algorithms: the BMH algorithm, because it is generically considered the fastest of all classical algorithms; the FJS algorithm, because it has also demonstrated on average, the best performance at publication time (late 2005) in general purpose exact pattern-matching based on heuristics; the SBNDM algorithm, presented in 2003, because it combines the advantages of bit parallelism and shifting heuristics achieving top performance when $m \leq \omega$, being ω the number of bits in a computer word (typically 32 or 64); and the WML algorithm, recently published as the fastest algorithm on many cases, in particular on small size alphabets.
GRASPm will be included in DNA tests, since it is, based upon our best knowledge, the fastest exact pattern-matching in this field. For a comprehensive comparison of related algorithms we suggest [16] [17] [18].

3. The New Algorithm: Description and Implementation

This paper presents a novel algorithm that improves significantly exact pattern-matching performance. Combining known ideas like large search window, multiple alignments per iteration and constant shifts [5], with a novel search strategy and an innovative compatibility rule heuristic, the proposed DC algorithm extends searching flexibility and efficiency, keeping a low space complexity and simplicity. BM descendants are popular in natural language applications while bit-parallelism based searching algorithms are well suited for small alphabets and moderate patterns (normally limited to $m \leq 32$). DC algorithm represents a novel heuristics based approach, surpassing in efficiency and flexibility the existing algorithms in this category. Traditionally, heuristic based approaches are dominant in natural language searching but are less competitive when searching text derived from small alphabets ($\sigma \leq 8$), whereas bit parallelism approaches obtain better results in this field. However, the proposed algorithm greatly reduces the performance gap, without pattern length limitations. The new algorithm acts in two sequential phases, the pre-processing phase where the pattern is analyzed, and the searching phase where the text is iteratively scanned in order to identify pattern replicas.

3.1. Basic concepts and definitions

Before detailing each phase, it is fundamental to define some concepts in order to prepare and sustain further explanations.

Lemma 3.1.1: If a pattern p of length m exists within a search window of $2m-1$ contiguous characters, p includes necessarily the window's central character $x[cc]$. In consequence, the searching phase is focused primarily on $x[cc]$.
Proof: Being $2m-1$ the length of the search window, any set of m consecutive characters within the window will include a common element - the central character of the window. By just verifying the $x[cc]$ character is enough to determine whether or not p occurrences are possible.
Large search window with eventual multiple alignments: A new search window, containing $2m-1$ characters and centered in $p[m]$, is only considered when an instance of $p[m]$ is found, which means that at least, one alignment needs to be tested. Each window may contain a maximum of m alignments to test. More concretely, each window includes a maximum of alignments equal to $p[m]$ occurrences in the pattern.

Central character (*cc*) as alignment reference: Based on the Lemma 3.1.1, only the central character of the window participates in all possible alignments of p, thus, it is considered the reference for alignments testing. As mentioned before, all the possible alignments will match a $x[cc]$ equal to the character in $p[m]$. Whenever a window is established $x[cc]=p[m]$. The cc value is also used as progression variable, as the searching task ends when $cc>n$.

Precedent character of *cc* as a parallel filter: The *x[cc-1]* character of the search window is used as sentinel for the compatibility rule. Any character in the alphabet could be a future *x[cc-1]*. Since *x[cc]* is always equal to *p[m]*, a pattern replica can only occur if *x[cc-1]* matches any precedent of *p[m]* occurrences in the pattern. Therefore, it is valuable to pre-compute which values of *x[cc-1]* are plausible to consider further verifications, with this definition is possible to discard several alignments just examining the *x[cc-1]* character. This definition constitutes the basis for the compatibility rule heuristic.

Compatibility rule: To avoid exhaustive alignments tests of all the occurrences of *p[m]* in the pattern with *x[cc]* is necessary to gather information to support selective decisions. In fact, not all possible alignments are compatible alignments. The compatibility rule is a selective rule that inhibits incompatible alignments. Except for the first character, a pattern has a precedent before each character. The proposed algorithm is interested in the precedents of the occurrences of *p[m]* in the pattern. These precedents are relevant because can be used to preview useful alignments compatibilities in the search phase, since future *x[cc-1]* characters will be aligned with the referred precedents. An alignment is compatible with a future *x[cc-1]* if it possesses an equivalence in the precedent. Therefore, the compatibility table contains, for each character in the alphabet, its list of compatible alignments. The incompatibility is determined if the *x[cc-1]* of the window does not precedes any occurrence of *p[m]* in the pattern. In searching phase, pre-processed compatibilities for the *x[cc]* under analysis are available, allowing selective alignments trials. The relevant overhead of processing the compatibility rule occurs in the pre-processing phase. The compatibility rule represents a good performance tradeoff as it avoids a huge number of superfluous comparisons, and additively their necessary setup.

Regular shift of *m* characters per window: After a search window is established and tested, it is always possible a default shift of *m* characters to reposition the next window further in the text. In fact, a new pattern instance could only occur beyond the last alignment tested, which means that, at least, it has to finish one character ahead. To include this alignment as the first of the next window, a shift of *m* characters is required.

Cyclic extra-shift: All iterations begin with a cycle of extra-shifts. A pre-processed shift table, based on the BM's bad character rule, provides extra-shift values to rapidly found *p[m]* occurrences in the text. Initially, $cc=m$ and, while $cc \leq n$ and $x[cc] \neq p[m]$, a shift cycle is maintained being *cc* successively incremented with *extra_shift(cc)*. When $x[cc]=p[m]$ the cycle is interrupted to establish a new window.

3.2. Pre-processing phase

This phase is mainly related to knowledge gathering through pattern analysis. Arbitrarily, it is initiated with the extra-shift table computation. Basically, this table contains the maximum shift value for each pattern character and *m* for the remaining characters of the alphabet that do not integrate the pattern. As the extra-shift function will be applied to the character that immediately follows the window, if this character matches the last character of the pattern then the shift value will be null. Otherwise, the maximum shift value is obtained by observing the distance from the last occurrence of a character in the pattern to *m*. Later, in the search phase it is possible to shift repeatedly the window analyzing only the shift table. While the central character *(cc)* of the next window does not match *p[m]*, or *extra_shift(cc)>0*, the next central character can be incremented iteratively without further verifications. When this window progression stops, *x[cc]* is necessarily equal to *p[m]*, hence only the last character alignments need pre-processing, reducing considerably the algorithm's space complexity. Considering *p="Albert Einstein"*, with *m=15*, the resulting shift table is shown in Table 1.

Table 1 – Shift table example for *p="Albert Einstein".*

ASCII	…	32	…	65	…	69	…	98	…	101	…	105	…	108	…	110	...	114	115	116	…
Char.	…	spc	…	A	…	E	…	b	…	e	…	i	…	l	…	n	…	r	s	t	…
Max. Shift	15	8	15	14	15	7	15	12	15	2	15	1	15	13	15	0	15	10	4	3	15

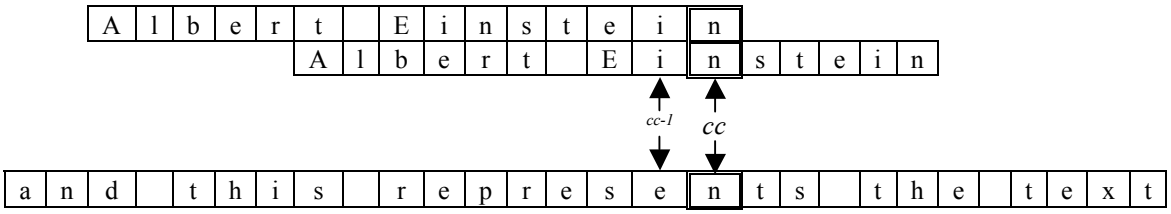
The pre-processing phase aims to comprehend the peculiarities of the pattern, not only to compose a shift table, but also to broadly optimize pattern recognition in the processing phase. As the only alignments that will be necessary to study are those which involve *p[m]*, the occurrences of *p[m]* in *p* are registered in a vector in two ways: the number of occurrences in the pattern (first cell), and each specific position or index of occurrence (the following cells). Table 2 illustrates an example for the pattern *p="Albert Einstein"*, where the character *p[m]='n'* (ASCII code 110) has two occurrences, at 10^{th} and 15^{th} characters.

Table 2 – Pattern study detailing *p[m]='n'* occurrences for *p="Albert Einstein"*.

Occurrences	Occurrences' Indexes			
2	10	15	0	…

The pre-processing phase could now attain the compatibility rule's analysis, which represents the classification of the pattern alignments as compatibles or incompatibles under certain circumstances. The compatibility rule plays a major role in the proposed algorithm, being crucial to assure both performance and flexibility. Since the proposed algorithm uses a search window of *2m-1* characters, frequently examines multiple alignments within the window, thus the compatibility rule aims to reduce the number of attempts per iteration inhibiting incompatible alignments. As the pre-processing phase only focuses on the pattern, it is possible to study all the pattern alignments of *p[m]*, grouping them (see Table 3), and analyze the conditions that should occur later in the search phase to effectively test an alignment.
Reusing *p="Albert Einstein"*, we have two alignments for the character in *p[m]*, as shown in Table 3. In the next phase, for each search window will be considered a central character (*x[cc]*) as reference for *p[m]* alignments, and if *x[cc]* will always meet a character *p[m]*, the previous character (*x[cc-1]*) will always meet the precedent characters of *p[m]* occurrences, previewed by the compatibility rule. Therefore, these characters can be explored to differentiate the possible alignments as compatibles or incompatibles facing the future *x[cc-1]*. In the example in Table 3, the alignments are only compatible with a future *x[cc-1]='i'* (ASCII code 105), for any different *x[cc-1]*, the algorithm considers an incompatible alignment and will not waste processing time.

Table 3 – Alignments with the character in *p[m]* for *p="Albert Einstein"*.



The compatibility rule table (see Table 4) is pre-processed and will contain the compatible alignments' specifications, supplying the necessary parameterization to proceed with alignment tests in the searching phase. The compatibility rule is not effective to dismiss alignments when *x[cc]=p[1]* since no previous character exist. In these cases no alignments can be excluded. The alignments' indexes are stored backwards; this is required to find eventual pattern replicas in the correct sequence. In fact, greater indexes correspond to earlier pattern occurrences.

Table 4 – Compatibility pre-processed table for *p="Albert Einstein"*.

ASCII	...	105	…
Align. 1	0	15	0
Align. 2	0	10	0
…	…	...	...
Align. n	0	0	0

An implementation proposal, in C language, of this phase of the algorithm is presented in Fig. 1.

```
#define DIM_ALPHABET 256
#define MAX_PATTERN 256

unsigned int lco[DIM_ALPHABET+1][MAX_PATTERN+1];//last character occurrences table
unsigned int compatibility[DIM_ALPHABET+1][MAX_PATTERN+1];//compatibility table
unsigned int xshift[DIM_ALPHABET+1];//extra shift table

//*****************************************************************************
void imply_all(){
int i,j;
for (j=0;j<=DIM_ALPHABET;j++){
    i=1;
    while (compatibility [j][i]>0) i++;
    compatibility [j][i]=1;
  }
}
//*****************************************************************************
void imply_alignment(unsigned int ic,unsigned int v){
  int i;
  i=1;
  while (compatibility [ic][i]>0) i++;
  compatibility [ic][i]=v;
}
//*****************************************************************************
void DC_Preprocessing(char *pattern, unsigned int m){
 unsigned int i, n, index, ic;
 // extra shift table update
 for (i =0;i<=DIM_ALPHABET; i++) xshift [i] = m;
 for (i=0; i<=m-1 ;i++) xshift[pattern[i]]=m-i-1;
 // p[m] occurrences table update
 for (n=1;n<=DIM_ALPHABET;n++) lco [n][1]=0;
 for (n=0;n<=m-1;n++){
    index = pattern[n];
    lco[index][1]++;
    if (lco[index][1]>MAX_PATTERN) printf ("Pattern length overflow!");
    lco[index][lco[index][1]+1]=n+1;
   }
 // compatibility table update
 ic=pattern[m-1];
 for (n=lco[ic][1];n>=1;n--){
        if (lco[ic][n+1]>1) // All the alignments that have a precedent character
          imply_alignment(pattern[lco[ic][n+1]-2],lco[ic][n+1]);
          else imply_all();
    }
}
```

Fig. 1 – Pre-processing phase of the DC algorithm: an implementation in C language.

3.3. Searching phase

The searching phase is based on alignment trials over the iterative searching windows used to discover instances of the pattern within the text. However, the first search window may not coincide with the initial text. In fact, there is no need of window definition until an alignment probability is detected. Initially $cc=m$ and an extra-shit cycle is performed until $x[cc]=p[m]$ or $cc>n$.

In the best case the searching phase will end without testing any alignment and using always the maximal shift. However, in the average-case, after a short shift cycle the first window is established, then the characters $x[cc]$ and $x[cc-1]$ are used to evaluate the compatibility of the alignments to test them selectively. An important advantage of this algorithm relies mainly in the fact that, no matter the number of alignments to test within a window, the characters $x[cc]$ and $x[cc-1]$ involving all pattern occurrences will be tested only once, saving redundant computation. The validation relies on a different search strategy based on the assumption expressed in Lemma 3.1.1.

Note that the last pattern's character is always an eventual alignment but not always a compatible one. The remaining occurrences of *p[m]* in the pattern are the other candidates to compatible alignments. By consulting the compatibility table where the compatible alignments for a particular *x[cc-1]* are described, further tests are performed selectively avoiding excessive computation. By only testing the character *x[cc-1]* it is possible to avoid several character comparisons to decide which ones are not viable. This feature contributes to enhance efficiency.
For each compatible alignment, it is necessary to retrieve the index of alignment of the pattern with the central character from the compatibility table. So, if a window presents, at least, one compatible alignment, the necessary variables are adjusted to initiate character comparisons. As characters *x[cc]* and *x[cc-1]* are pre-tested, there is no need to repeat redundant comparisons. When the pattern is aligned with the beginning of the search window the verification process will imply just a prefix. Normally, there will be a prefix before *x[cc-1]* and a suffix after *x[cc]* to be verified. In the cases of compatible alignments where *x[cc]* is aligned with *p[1]* only the suffix needs verification. The rule is to verify the prefix first, and if the possibility of a complete match subsists, the suffix is also verified, in both cases from left to right. If the number of successful comparisons equals *m*, then a pattern occurrence is reported. When all the compatible alignments are tested the iteration is terminated. Subsequently, a regular shift of *m* characters is summed to *cc* as a first increment to reposition the window further in the text. The new *x[cc]* is then evaluated by the extra-shift function, and another shift cycle starts (as described at the beginning of this subsection) to find the next pattern-probable window. The searching phase ends when all the text has been scanned.
An implementation proposal for the new algorithm's searching phase, in C language, is shown in Fig. 2.

Note that in the proposed algorithm very small patterns (*m=1* and *m=2*) are considered particular cases. When *m=1*, the BF approach was used. When *m=2*, a new search strategy was created to improve performance, especially for small alphabets. An implementation proposal for *m=2* cases is presented in [12].

```
void DC_Search(char *text, long n, char *pattern, unsigned int m)
{
 int cc,na,precedent,ia,iap,j,prefix;
 char b;

 b=pattern[m-1]; //Last character in pattern
 cc=m-1; // First cc
 while ((text[cc]!=b) && (cc<=n)) cc+=xshift[text[cc]];  // First extra-shift cycle
 while (cc<=n)
   {
     precedent=text[cc-1];
     ia=1;
     while ((na=compatibility[precedent][ia])>0)
         {
            prefix=na-2; // Prefix length
            iap=cc-prefix-1; // Position to align the pattern
            j=0;
            while ((j<prefix) && (text[iap+j]==pattern[j])) j++; // Prefix test
            if (j>=prefix)
              {
               j=prefix+2; //cc-1 and cc are pre-tested, advance to suffix
               while ((j<m) && (text[iap+j]==pattern[j])) j++;   // Suffix test
               if (j>=m) printf ("\nPattern at position %d.", iap);
              }
            ia ++;  // Advance to the following compatible alignment
         }
     cc+=m;  // regular shift
     while ((text[cc]!=b) && (cc<=n)) cc+=xshift[text[cc]]; // Extra-shift cycle
   }
}
```

Fig. 2 – Searching phase of the DC algorithm: an implementation in C language.

An illustrative example is now provided to better understand the behavior of the new algorithm during the processing phase. The searching example uses the text *x*="*This text includes the pattern Albert Einstein once.*", with *n*=52 and the pattern *p*="*Albert Einstein*", with *m*=15, already analyzed in the pre-processing phase section.

The first *cc* is initialized with *m*, thus *x[cc]='u'* (see Fig. 3), while *x[cc]≠p[m]* the extra shift table is recurrently consulted to increment *cc* in order to rapidly find the next occurrence of *p[m]* in the text.

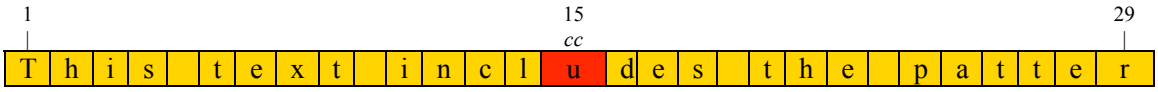


Fig. 3 – The beginning of the first iteration and the first shift cycle.

Consulting Table 1, *extra_shift('u')* is 15, thus next *cc=30*. As the new *x[cc]='n'=p[m]*, the first search window is established (see Fig. 4).

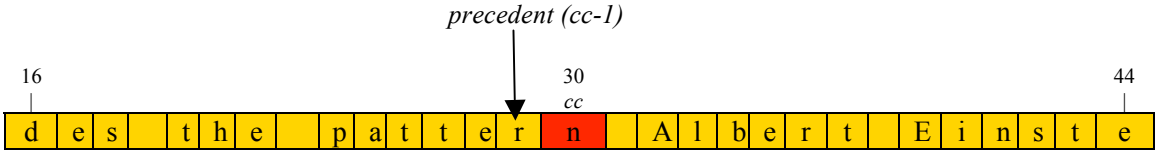


Fig. 4 – The first search window is established, the characters *x[cc]* and *x[cc-1]* are analysed.

Analyzing the compatibility conditions within the first window (see Table 4), the compatibility rule states, facing a *precedent='r'*, that no alignments are compatible, therefore, no further tests are needed. The constant shift component is applied (advancing *m* characters), so the new *cc=45* and the next iteration begins with *x[cc]='i'* (see Fig. 5).

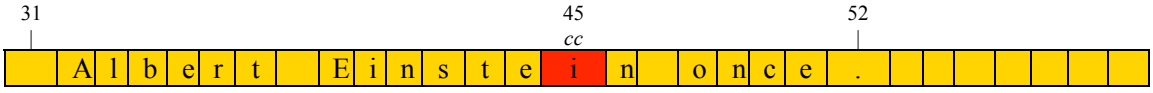


Fig. 5 – The beginning of the second iteration and the second shift cycle.

Consulting Table 1, *extra_shift('i')* is 1, thus next *cc=46*. As *x[cc]='n'=p[m]*, the second search window is established (see Fig. 6).

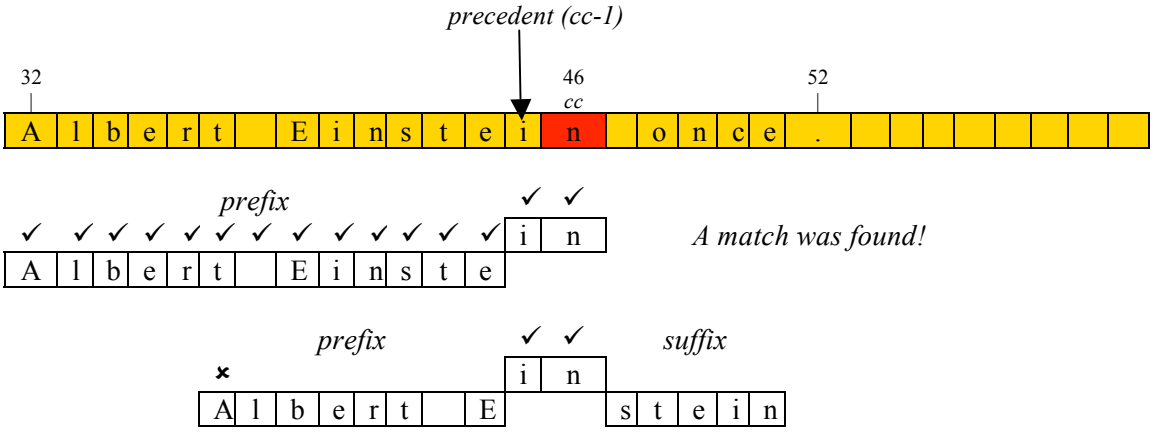


Fig. 6 – The second window is established, the characters *x[cc]* and *x[cc-1]* are analyzed.

Accordingly with the compatibility rule (see Table 4), facing a *precedent='i'*, it is necessary to test two compatible alignments, the first includes the pattern aligned with *x[cc]* by the 15th character, and the second includes the pattern aligned with *x[cc]* by the 10th character. Character comparisons are initiated to confirm or not prefix and suffix correspondences with the text. For the first alignment all characters match, therefore a pattern occurrence is revealed. The second alignment verification fails at the first character comparison. As with the addition of the constant shift component the new *cc* exceeds *n*, the text was completely scanned and the algorithm is ended.

3.4. Complexity Analysis

The space and time complexity in the pre-processing phase is the sum of several pre-computations, namely: the extra-shift table composition, the *p[m]* occurrences table and the compatibility table. The extra-shift table composition requires time $O(\sigma+m)$ and space $O(\sigma)$. The last character occurrences study requires $O(2m)$ time and the resulting table needs $O(m+1)$ space.

Comparatively with BM type algorithms, the new algorithm requires a supplementary table to maintain the compatibility rule data. Concretely, considering an alphabet with 256 symbols, and a pattern up to 256 characters, the required memory resources are 64KB, thus an irrelevant dimension facing the current hardware capabilities and the obtained benefit. The compatibility rule pre-processing requires $O(m)$ time and $O(\sigma m)$ space.

The time complexity in the searching phase is in the best case $O(n/m)$, this case occurs when the initial shift-cycle is only interrupted when the end of the text is reached, and during the cycle the shift function always returns maximum shifts. In the worst case, we can only count on the regular shifting, and each window will contain the maximum number of alignments (*m*), all compatibles and complete matches, requiring *m-2* characters comparisons each (*cc* and *cc-1* are special cases, in the processing phase these two characters are only tested once independently of the alignments present in the window). In this case the time complexity is $O(n/m)(2+m(m-2))$, considerably lower than $O(nm)$. In the average-case and mainly in small alphabets, the compatibility rule acts as a parallel test in the search phase, reducing time complexity. Thus, by just verifying one element of the compatibility table, it is possible to discard (test) several alignments.

4. Experimental Results and Comparisons

As mentioned in Section 2, the selected contenders for performance comparisons are all best-rated searching algorithms in its categories: BMH, representing the efficiency and simplicity of classical algorithms; FJS, representing the recent hybrid heuristics based approach; SBNDM, as the top performing bit parallelism algorithm; WML as the most recent reference; and GRASPm, the ultimate genomic-oriented exact pattern-matching algorithm. Since the available versions of the WML algorithm are based on *q*-grams ($3 \leq q \leq 8$), and the algorithms in competition are unigram based - except for GRASPm which is 2-gram based - an adaptation of WLM algorithm was done to work with 2-grams in order to keep equality. For simplicity reasons this version will be designated here as WML2. The FAOSO algorithm [19], a descendant of Shift-Or, appears in the literature as an ultra-fast algorithm. A genuine implementation was kindly provided by the authors for comparison purposes. However, FAOSO includes a parameterization *k*, which is context dependent and determines greatly the algorithm performance. In fact, an incorrect *k* ruins the algorithm's performance. Despite our effort, it was impossible to automatically generate the best *k* facing the variables present in searching problems. Thus, adjusting *k* is trial based and costly. As the optimization of *k* is prohibitive in real applications, FAOSO was excluded from our list of competitors.

The DC algorithm was coded in C language. Genuine implementations for the other algorithms, also in C language, were compiled in the same way to achieve a significant comparison. An implementation of BMH is included in [17], an implementation of FJS is provided by the authors in [9] and, SBNDM and WML2 implementations used were a courtesy of the authors.

Performance tests were executed using a system based on an Intel Pentium IV - 3,4 GHz - 512KB cache - 1GB DDR-RAM, under Windows XP Professional SP2 OS. Applications were compiled with *gcc* (full optimization) and execution times were collected in milliseconds using the *timeGetTime()* function, provided by an OS library (*libwinmm.a*).
The tests have comprised multiple texts built over several alphabets. Concretely, the participant algorithms were tested searching binary data ($\sigma=2$), DNA data ($\sigma=4$), protein data ($\sigma=20$) and natural language (based on ASCII) data ($\sigma=256$). The main underlying idea was to probe the algorithms in searching about 50 MB of each of these data types, in order to analyze the overall results and evaluate the most flexible and efficient one.
To search binary text, ~50 MB of data were randomly generated. The DNA sequence used in the tests was part of the Human Chromosome 1 (the initial 50 MB), downloaded from UCSC[1] biological databases. The proteins sequence used was the result of merging four of the largest proteomes available. The gathered file, include the *Homo Sapiens*, *C. Elegans*, *A. Thaliana* and *Mouse Musculus* proteomes, which were downloaded from Integr8[2] databases. In the merged file, FASTA tags were cleared, conserving only the amino-acid sequences, resulting in nearly 50 MB of raw data. The natural language text resulted from a compilation of 37 e-books, mainly from European Literature, including Charles Dickens, Victor Hugo, Sir Arthur Conan Doyle, Jules Verne, etc., based on ASCII plain text, and obtained from Project Gutenberg[3]. The merged text length is also about 50 MB.
For each data type, a pattern collection containing 700 different patterns, based on 100 samples by length class, with *m=2, 4, 8, 16, 32, 64* and *128*, were randomly generated (except for natural language patterns) and stored in a file for test purposes. The natural language patterns consist of English words when $m \leq 8$, the larger patterns are complete or incomplete sentences randomly chosen from the text. The mean execution time in milliseconds, comprising pre-processing and searching phase, was used to establish the following performance comparison.
We have run 2800 tests for each algorithm, employing 100 pattern samples multiplied by 7 different pattern lengths multiplied by 4 alphabet types. The above tables (Tables 5-8) contain a summary of the results of these tests as they contain, per algorithm, the 28 average runtimes measured. DC and SBNDM are clearly leaders in exact pattern-matching using 1-gram algorithms. Considering 28 different competitions, DC collects 17 winnings. If we reduce the competition to $m \leq 32$ patterns in order to include SBNDM in equality, in 20 competitions DC accumulates 11 wins while SBNDM gets 7 wins. SBNDM is dominant in binary data (see Table 5), in genomic data (see Table 6), on average, DC is slightly superior to SBNDM. In protein data (see Table 7) as in natural language data (see Table 8) DC is clearly dominant and SBNDM appears only in third place. FJS is a good option for the specific case of very small patterns with $m < 4$. WML2 is a good option for long patterns considering alphabets with $\sigma \geq 4$ as it takes advantage of 2-grams.

Table 5 – Runtimes for binary data ($\sigma=2$), using ~50 MB of randomly generated data.

	DC		BMH		FJS		SBNDM		WML2	
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank
2	**309**	**1**	604	4	553	2	562	3	1632	5
4	629	4	**565**	**1**	582	2	621	3	885	5
8	530	2	531	3	634	4	**413**	**1**	643	5
16	482	2	601	3	668	5	**225**	**1**	623	4
32	435	2	619	3	684	4	**121**	**1**	700	5
64	**389**	**1**	614	2	653	*3*	>ω	?	828	4
128	**358**	**1**	617	2	657	*3*	>ω	?	1154	4

[1] hgdownload.cse.ucsc.edu/downloads.html
[2] www.ebi.ac.uk/integr8/
[3] www.gutenberg.org

Table 6 – Runtimes for DNA data ($\sigma=4$), using part (initial 50 MB) of the Human Chromosome 1.

	DC		BMH		FJS		SBNDM		WML2		*GRASPm*
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	*Time (ms)*
2	**267**	**1**	406	3	337	2	452	4	898	5	***267***
4	**258**	**1**	279	2	306	3	315	4	353	5	***227***
8	191	3	220	4	290	5	184	2	**179**	**1**	***124***
16	142	3	201	4	283	5	**107**	**1**	115	2	***70***
32	113	3	194	4	294	5	**61**	**1**	95	2	***42***
64	92	2	201	3	296	4	>ω	?	**90**	**1**	***28***
128	**77**	**1**	210	3	300	4	>ω	?	114	2	***20***

Table 7 – Runtimes for proteins data ($\sigma=20$), using several merged proteomes (~50MB).

	DC		BMH		FJS		SBNDM		WML2	
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank
2	**173**	**1**	271	4	174	2	177	3	672	5
4	**111**	**1**	149	4	120	2	132	3	235	5
8	**67**	**1**	86	3	77	2	96	4	111	5
16	**44**	**1**	56	3	52	2	61	5	60	4
32	32	2	42	5	40	4	**31**	**1**	37	3
64	**26**	**1**	33	4	32	3	>ω	?	27	2
128	**22**	**1**	31	4	30	3	>ω	?	**22**	**1**

Table 8 – Runtimes for natural language ($\sigma=256$), using an ASCII e-books compilation (~50MB).

	DC		BMH		FJS		SBNDM		WML2	
m	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank	Time (ms)	Rank
2	160	3	237	4	149	2	**144**	**1**	586	5
4	**94**	**1**	127	4	95	2	101	3	197	5
8	**58**	**1**	75	3	65	2	84	4	96	5
16	**44**	**1**	55	3	52	2	64	4	52	2
32	**31**	**1**	39	4	39	4	37	3	32	2
64	25	2	32	3	29	2	>ω	?	**24**	**1**
128	**17**	**1**	21	4	19	3	>ω	?	**17**	**1**

Table 9 – Ranking sums, considering tests using all patterns and patterns with $m \leq 32$ only.

	DC		BMH		FJS		SBNDM		WML2	
	all	*m≤32*	*all*	*m≤32*	*all*	*m≤32*	*all*	*m≤32*	*all*	*m≤32*
$\sigma = 2$	**13**	11	18	14	23	17	?	**9**	32	24
$\sigma = 4$	**14**	**11**	23	17	28	20	?	12	18	15
$\sigma = 20$	**8**	**6**	27	19	18	12	?	16	25	22
$\sigma = 256$	**10**	**7**	25	18	17	12	?	15	21	19
$\sum$	**45**	**35**	93	68	86	61	?	52	96	80

In Table 9, an overall analysis is presented by summing the individual rankings obtained searching each specific data type. In order to include SBNDM in this analysis, the sums also consider separately the tests using patterns with $m \leq 32$. The conclusion is evident; in general terms, DC stands as the most efficient algorithm for alphabets with $\sigma \geq 4$, also being

competitive in binary data searching when compared with heuristics based algorithms, emerging consequently as the most flexible algorithm for exact-pattern matching. Additionally, DC is not limited by computer word size and its implementation is very simple.
Note that in genomic data, GRASPm demonstrates clear supremacy in all pattern lengths, however it is considered a genomic-oriented algorithm, not usable in large alphabets, and therefore is not considered in the overall analysis. GRASPm is based on 2-grams and uses a logic similar to DC, the main difference is the absence of the shift cycle. Using 2-grams during the compatibility rule pre-processing and to compose the sentinel, GRASPm does a better use of the compatibility rule, improving efficiency and consequently the searching times.
SBNDM could also be used to handle long patterns as described in [12], however the achieved performance is not as competitive as in the basic version since the pattern needs to be divided and searched by parts due to word size limitation.
To complement the performance analysis and assess the efficiency of the proposed algorithm, the number of character comparisons (*ncc*) was also evaluated. Reusing the same data and patterns sets, the *ncc* executed by the leading algorithms were measured. The average values obtained were organized in Table 10.

Table 10 – Number of character comparisons executed by the leading algorithms using DNA and Protein data types.

	DNA Data (~50MB)		Proteins Data (~50MB)	
m	DC	SBNDM	DC	SBNDM
4	**26102884**	31699532	**16254579**	17773507
8	18771859	**18479193**	**9002221**	10025841
16	13367854	**10202793**	**5229691**	5654516
32	10046115	**5731745**	3581488	**3358118**
64	**7566082**	?	**2596050**	?
128	**6123337**	?	**2128837**	?

Ratifying the execution time results, the *ncc* necessary to complete the different searching tasks are, in general terms, favorable to SBNDM on small size alphabets and favorable to DC on greater alphabets.

5. Discussion and conclusions

We have presented a novel algorithm for general-purpose exact pattern matching. The main goal was to design a new algorithm, highly efficient regardless the data type to search or pattern length. As the results demonstrate, the new algorithm is highly efficient and flexible, standing as a good choice for all kind of alphabets, except for binary data, and undoubtedly the best choice when $\sigma \geq 20$. For smaller alphabets, SBNDM, and mainly GRASPm, are the best choices.
The proposed algorithm is heuristic based, not limited in pattern length, introducing new heuristics and a novel search strategy. The most valuable contributions are the compatibility rule which enhances the multi-alignments windows searching strategy. The space complexity is reduced since only the *p[m]* occurrences' alignments are pre-processed and analyzed. The compatibility rule is particularly useful in small and medium alphabets and in presence of long patterns as it allows parallel verifications to decide selectively the alignments to test. Furthermore, it includes a shift cycle to enhance performance when dealing with large alphabets, where alignments occur more rarely.
SBNDM is well suited for binary data, but inadequate when searching data from large alphabets, also presenting poor performance for small alphabets and small patterns ($m \leq 4$). In the overall ranking, DC wins by a considerable margin.
Analyzing the performance stability aspects, the proposed algorithm does not suffer notoriously from pattern composition variations. On the contrary, BMH is perceptibly affected by pattern composition variations. In addition, the new algorithm presents a progressive behavior, i.e., as pattern length increases the algorithm's performance improves gradually, always taking

advantage of pattern length. On the contrary, the competitors, except SBNDM, do not consistently exhibit this behavior.
Comparatively with BM type algorithms, the new algorithm requires a supplementary table to maintain the compatibility rule pre-processing data. Concretely, considering an alphabet with 256 symbols, and a pattern up to 256 characters, the required memory resources are 64KB, thus an irrelevant size facing the current hardware capabilities.
In practice, the complexity analysis for the proposed algorithm evinces a sub-linear behavior in the average case but, further analysis is necessary to theoretically demonstrate it.
In summary, attending to the innovative search strategy and high performance achieved, the proposed algorithm is a relevant contribution regarding flexible and efficient exact pattern-matching.

Acknowledgment

This work was supported in part by a grant from the Portuguese Government – PRODEP/Action 5.3.

References

[1] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," *SIAM J. Comput.*, vol. 6(2), pp. 323-350, 1977.
[2] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," *Commun. Assoc. Comput. Mach.*, vol. 20(10), pp. 762-772, 1977.
[3] R. N. Horspool, "Practical fast searching in strings," *Software - Practice & Experience*, vol. 10(6), pp. 501–506, 1980.
[4] D. M. Sunday, "A very fast substring search algorithm," *Commun. Assoc. Comput. Mach.*, vol. 33(8), pp. 132-142, 1990.
[5] S. Kim, "A new string-pattern matching algorithm using partitioning and hashing efficiently," *Journal of Experimental Algorithmics (JEA)*, vol. 4(2), 1999.
[6] M. Crochemore, A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski, and W. Rytter, "Speeding up Two String Matching Algorithms," *Algorithmica*, vol. 12(4/5), pp. 247-267, 1994.
[7] R. A. Baeza-Yates and G. H. Gonnet, "A new approach to text searching," *Commun. ACM*, vol. 35(10), pp. 74-82, 1992.
[8] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM J Res Dev.*, vol. 31(2), pp. 249-260, 1987.
[9] F. Franek, C. G. Jennings, and W. F. Smyth, "A Simple Fast Hybrid Pattern-Matching Algorithm," *Lecture Notes in Computer Science*, vol. 3537, pp. 288-297, 2005.
[10] G. Navarro and M. Raffinot, "Fast and Flexible String Matching by Combining Bitparallelism and Suffix automata," *ACM Journal of Experimental Algorithms*, vol. 5(4), pp. 1-36, 2000.
[11] M. Crochemore and W. Rytter, *Text algorithms*: Oxford University Press, 1994.
[12] H. Peltola and J. Tarhio, "Alternative Algorithms for Bit-Parallel String Matching," in Proceedings of SPIRE '03, 10th Symposium on String Processing and Information Retrieval, 2003.
[13] T. Lecroq, "Fast exact string matching algorithms," *Information Processing Letters*, vol. 102, pp. 229-235, 2007.
[14] S. Wu and U. Manber, "A fast algorithm for multi-pattern searching," TR-94-17, Department of Computer Science, University of Arizona, Tucson 1994.
[15] S. Deusdado and P. Carvalho, "GRASPm: an efficient algorithm for exact patter-matching in genomic data," *Int. Journal of Bioinformatics Research and Applications*, vol.5(4), 2009.
[16] P. D. Michailidis and K. G. Maragaritis, "On-line String Matching Algorithms: Survey and Experimental Results," *International Journal of Computer Mathematics*, vol. 76(4), pp. 411-434, 2001.

[17] T. Lecroq, "Experimental Results on String Matching Algorithms," *Software - Pratice and Experience*, vol. 25(7), pp. 727-765, 1995.
[18] B. Smyth, *Computing Patterns in Strings*: Pearson Addison-Wesley, 2003.
[19] K. Fredriksson and S. Grabowski, "Practical and Optimal String Matching," in Proceedings of SPIRE '05, 12th Symposium on String Processing and Information Retrieval, 2005.