INFORMATION TECHNOLOGY JOURNAL

Information Technology Journal 9 (3): 453-459, 2010
ISSN 1812-5638

An Efficient ASCII-Based Algorithm for Single Pattern Matching

Rami H. Mansi and Jehad Q. Alnihoud
Department of Computer Science, Faculty of Information Technology,
Al al-Bayt University, Mafraq, Jordan

Abstract: In this study, we propose a new single exact pattern matching algorithm, called ASCII-Based-RJ algorithm. Furthermore, a string matching tool (RJ-SMT) has been developed to simulate and test the proposed algorithm, the naïve (brute force) and Boyer-Moore algorithms. In its searching phase, the proposed algorithm improves the shifts in the naïve and Boyer-Moore algorithms by excluding the text's segments that contain a different character from pattern's characters. Based on the experimental results, the ASCII-Based-RJ algorithm outperformed the naïve algorithm by 35.3% and Boyer-Moore algorithm by 2.3%. We may conclude that adding some restrictions, or conditions, on text's characters during the preprocessing phase, increases the efficiency of the searching phase, which increases the efficiency of the algorithm as a result.

Key words: String matching, boyer-moore algorithm, naïve matching, ASCII-based matching, design of algorithms

INTRODUCTION

The string matching problem, also called pattern matching, is defined as the operation of finding one or all of the occurrences of a pattern of characters P of length m in a larger text T of length n. The problem has been extensively studied and many techniques and algorithms have been designed to solve this problem. String matching algorithms are mostly used in information retrieval, bibliographic search, molecular biology and question answering applications (Lecroq, 2007; Wu *et al*., 2007).

String matching is a very important issue in the domain of text processing and its algorithms are considered as the basic components used in implementations of practical software under most operating systems. Moreover, they emphasize programming methods that serve as paradigms in other fields of computer science (Charras and Lecroq, 2004; Watson, 2002).

In most of information retrieval and text-editing applications, it is necessary to the user and developer to be able to locate quickly some or all occurrences of a specific pattern of words and phrases in a text (Alqadi *et al*., 2007). Moreover, string matching has many applications including database query, DNA and protein sequence analysis. Therefore, the efficiency of string matching algorithms has a great impact on the performance of these applications (Crochemore and Lecroq, 2003). Although data are memorized in various ways, text remains the main and most efficient form to exchange information; since it is independent of the software and hardware that used in different systems (Kim and Kim, 1999; Sheu *et al*., 2008).

Basically, a string matching algorithm uses a window to scan the text. The size of this window is equal to the length of the pattern. It first aligns the left ends of the window and the text. Then it checks if the pattern occurs in the window (this specific work is called an attempt) and shifts the window to the right. It repeats the same procedure again until the right end of the window goes beyond the right end of the text (Amintoosi *et al*., 2006).

Exact string matching means finding one or all exact occurrences of a pattern in a text. Naïve (brute force) algorithm, as mentioned by Charras and Lecroq (2004), Boyer and Moore (1977), Morris and Pratt (1970) and Knuth *et al*. (1977) are exact string matching algorithms. Some of the exact string matching algorithms have been presented to solve the problem of searching for a single pattern in a text, such as Karp and Rabin (1987) algorithms. In the other hand, some have been presented to solve the problem of searching for multiple patterns in a text. Although the Knuth-Morris-Pratt algorithm has better worst-case running time than the Boyer-Moore algorithm, the latter is known to be extremely efficient in practice (Crochemore *et al*., 1994; Watson and Watson, 2003).

Since 1977, with the publication of the Boyer-Moore algorithm, there have been many papers published that deal with exact pattern matching and in particular discuss and/or introduce variants of Boyer-Moore algorithm. The

Corresponding Author: Jehad Q. Alnihoud, Department of Computer Science, Faculty of Information Technology, Al al-Bayt University, Mafraq, P.O. Box 130040, Mafraq 25113, Jordan

pattern-matching literature has had two main categories: Reducing the number of character comparisons required in the worst and average cases and reducing the time requirement in the worst and average cases (Danvy and Rohde, 2006; Franek *et al*., 2006).

The brute force algorithm, as mentioned by Charras and Lecroq (2004), consists of checking, at all positions in the text between 0 and n-m, whether an occurrence of the pattern starts there or not. Then, after each attempt, it shifts the pattern by exactly one position to the right. The brute force algorithm requires no preprocessing phase and a constant extra space in addition to the pattern and the text. During the searching phase, the text character comparisons can be done in any order. The time complexity of the searching phase is O (mn), where m is the length of the pattern and n is the length of the text and the expected number of text character comparisons is 2n.

The Boyer-Moore algorithm scans the characters of the pattern from right to left beginning with the rightmost one. In case of mismatch (or a complete match of the whole pattern) it uses two pre-computed functions to shift the window to the right. These two shift functions are called the good-suffix shift and bad-character shift. Assume that a mismatch occurs between the character x(i) = a of the pattern and the character y(i+j) = b for the text during an attempt at 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 character different from x(i). If there exists no such segment, the shift consists in aligning the longest suffix v of y(i+j+1.. j +m-1) with a matching prefix of x. The bad-character shift consists in aligning the text character 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 include y(i+j) and the left end of the window is aligned with the character immediately after y(i+j), namely y(i+j+1). As discussed in (Boyer and Moore, 1977; Navarro and Fredriksson, 2004), tables bmBc and bmGs can be pre-computed in time $O(m + \sigma)$ before the searching phase and require an extra-space in $O(m + \sigma)$. The searching phase time complexity is quadratic O(mn), but at most 3n text character comparisons are performed when searching for a non periodic pattern. On large alphabets (relatively to the length of the pattern) the algorithm is extremely fast.

This study is an attempt to reduce the processing time of both naïve (brute force) and Boyer-Moore algorithms. A new exact single pattern matching algorithm is proposed, analyzed, implemented and tested. The proposed algorithm improves the length of the shifts of the naïve and Boyer-Moore algorithms. Consequently, enhances the speed of the performance as compared to these algorithms.

MATERIALS AND METHODS

Ascii-Based-Rj algorithm: The preprocessing phase of the ASCII-Based-RJ (ASCII-Based-Rami and Jehad) algorithm finds the indices of the characters in the text that not occur in the pattern. Suppose that a character at index (z) in the text does not occur in the pattern, then the pattern cannot start in the segment (z-m+1 ... z) in the text, where m is the length of the pattern. Thus, this segment and all such segments, will be excluded during searching for the first, middle and last characters of the pattern in the text. The preprocessing phase creates a zero-based array called ASCII_Arr of size (95) elements (indexed from 0 to 94). This size represents the number of the printable characters in the ASCII table (American Standard Code for Information Interchange). These characters are from (space), which has the code (32), to (~), which has the code (126), in the ASCII table. The algorithm scans the characters of the pattern and for each character it increments the value of the element in the ASCII_Arr array using the actual ASCII code for that character minus (32) as index. For example, if the character (m), which has the code (109) in the ASCII table, occurs in the pattern, then the element in the ASCII_Arr array at index (77), (109-32 = 77), will be incremented by one. Therefore, the index of the (space) is (0) and the index of (~) is (94) in the ASCII_Arr array.

After that, the algorithm creates a new array, called SKIP_Arr, to hold the indices of the text that the pattern cannot start occurring at. These indices are determined by scanning the text's characters from right to left and the segment (z-m+1 ... z) for each index (z) in the text that contains a character does not appear in the pattern will be ignored during searching for the pattern's first, middle and last characters in the text. The range from (z-m+1) to (z) represents (m), which is the length of the pattern. The algorithm determines whether a character in the text occurs in the pattern by checking the corresponding element in the ASCII_Arr array (ASCII code of that character minus 32). If the value of the element in that index is zero, then this character (z) of the text did not appear in the pattern. Thus, the segment (z-m+1 ... z) in the SKIP_Arr array will hold the value (-1) to denote that this segment will be ignored during searching for the pattern's first, middle and last characters.

At this stage, the algorithm checks the elements of the SKIP_Arr array to search for the segments of the text that their first, middle and last characters equal the pattern's first, middle and last characters. If the value of the element is (0), the initial value, then it checks the text at index equal to the current index of the SKIP_Arr array and if the first, middle and last characters of the text's segment at that index equal the pattern's first, middle and last characters, the index will be saved in the Occurrence_List using a variable (i). The element will be skipped if it is (-1). This technique decreases the number of text character comparisons. Furthermore, it decreases the number of expected occurrences of the pattern in the text. As a result, it decreases the time complexity of searching for a pattern in a text.

Pseudocode of the ASCII-Based-RJ algorithm: The pseudocode of the preprocessing phase is expressed as follows:

```
procedure PRE-ASCII-BASED(array T(n),array P(m))
    var j:=i:=y:=z:=0, x:=n-1, mid=floor(m/2) as integer
    Create array: ASCII_Arr(95) initialized by 0's
    Create array: SKIP_Arr(n) initialized by 0's
    Create array: Occurrence_List(n-m+1) initialized by 0's
    for j from 0 to m-1 do
      Increment ASCII_Arr(ASCII_CODE(P(j))-32)
    for x from n-1 downto 0 do
      if ASCII_Arr(ASCII_CODE(T(x))-32)==0 AND x >= m-1 then
        for y from x-m+1 to x do
          if SKIP_Arr(y) == 0 then
            SKIP_Arr(y):=-1
          else
            Break the loop
      else
      if ASCII_Arr(ASCII_CODE(T(x))-32)==0 AND x < m-1 then
        for y from 0 to x do
          if SKIP_Arr(y) == 0 then
            SKIP_Arr(y):=-1
          else
            Break the loop
    for z from 0 to n-m do
      if SKIP_Arr(z) ? -1 AND T(z)==P(0) AND T(z+mid)=P(mid)
                              AND T(z+m-1)=P(m-1) then
        Occurrence_List(i):= z
        i:=i+1
    SEARCH-ACII-BASED(T(n), P(m), i, Occurrence_List(n-m+1))
end procedure
```

The searching phase of the ASCII-Based-RJ algorithm is as follows:

```
procedure SEARCH-ACII-BASED(array T(n), array P(m), i,
                        array Occurrence_List(n-m+1))
    if i > 0 then
      if m==1 then output the content of Occurrence_List()
      else
        var c:=x:=0, count:=1, as integer
        var value as Boolean
        while c < i do
          value:= true
          for x from Occurrence_List(c)+1
          to Occurrence_List(c)+ m-1 do
            if T(x)?P(count) then
              value:= false
              break the for loop
            count:= count+1
          if value==true then
            output(Occurrence_List(c))
          c:=c+1
          count:=1
    else output("The pattern is not found!")
end procedure
```

Table 1: The values of ASCII_Arr array (indexed from 0 to 94)

Characters	Index in ASCII_Arr (ASCII code-32)	Frequency
A	65-32 = 33	2
M	77-32 = 45	1
B	66-32 = 34	1
C	67-32 = 35	1
O	79-32 = 47	1

Example: Assume that we have the following text and pattern and we want to find all occurrences of the pattern in the text.

Text
AMACCOAMBACHAMABCOAMALCO
Pattern
AMABCO

Step one: The algorithm scans the characters of the pattern and for each character in the pattern it increments by one the value of the ASCII_Arr array at index equals to the code of that character in the ASCII table minus (32). In other words, the value of the corresponding element in the ASCII_Arr array of each character in the pattern will be incremented by one, using the ASCII code of that character minus (32) as index in the ASCII_Arr array. Table 1 shows the ASCII_Arr array after incrementing the corresponding elements of each character in the pattern.

Actually, the ASCII_Arr is of length (95), indexed from (0) to (94). But for simplicity, we have shown only the indices of the characters of the pattern in Table 1, while the rest of indices will contain (0), which is the initial value of the elements of the ASCII_Arr array.

As shown before, for each character in the text, there is a corresponding character in the ASCII_Arr at index equals to the ASCII code of that character minus (32).

Step two: After that, the algorithm scans the characters of the text from right to left, starting at the last character. If the corresponding value of the current character (z) of the text in the ASCII_Arr is zero, then this means that this character (in the text) does not occur in the pattern. Thus, the range from (z-m+1…z) in the text will be ignored during the searching phase, since the pattern cannot starts at any index of this range in the text. The algorithm saves this range in the SKIP_Arr array, which is of length n, where n is the length of the text. The algorithm overwrites the values of the elements of indices of the range (z-m+1…z) in the SKIP_Arr array by the value (-1) to be skipped during searching for the first, middle and last characters of the pattern. The SKIP_Arr array is shown in Table 2. Note that the indices in the table that have a gray background will be excluded during the searching phase, since the pattern can not occur in these indices.

Table 2: The SKIP_Arr array

Index	Value	Index	Value
0	0	12	0
1	0	13	0
2	0	14	0
3	0	15	0
4	0	16	-1
5	0	17	-1
6	-1	18	-1
7	-1	19	-1
8	-1	20	-1
9	-1	21	-1
10	-1	22	0
11	-1	23	0

Since the SKIP_Arr is of length n, which is the length of the text, the algorithm uses it to search for the pattern's first, middle and last characters in the text only at indices that have the value (0) in the SKIP_Arr array, while ignoring the indices that have the value (-1).

Step three: The algorithm saves the indices of the text's segments that have first, middle and last characters equal the pattern's first, middle and last characters in the Occurrence_List array to be used during the searching phase. The Occurrence_List array will be as follows:

- Occurrence_List 12

Step four: The algorithm uses the Occurrence_List array to search the text for the pattern, as follows:

First attempt (at index 12, five character comparisons, Match):
A M A C C O A M B A M H A **M A B C O** A M A L C O
A **M A B C O**

The algorithm performed (5) character comparisons in the example.

ANALYSIS OF THE ASCII-BASED-RJ ALGORITHM

The worst case of the preprocessing phase of ASCII-Based-RJ algorithm arises when each character of the text does not occur in the pattern, or, when for each m characters in the text, only the last character of that segment does not occur in the pattern. In this case, the lines (from 7 to 14) of the pseudocode of the preprocessing phase of ASCII-Based-RJ algorithm take O(2n) time, which is simplified as O(n).

The best case of the preprocessing phase of ASCII-Based-RJ algorithm occurs when all characters of the text appear in the pattern. This means, there is no any character exists in the text and does not occur in the pattern. In this case, the preprocessing phase takes O(n) time.

The preprocessing phase builds the Occurrence_List array, which holds the indices of the expected occurrences of the pattern in the text. During the searching phase, the algorithm uses the Occurrence_List to match the expected occurrences of the pattern in the text with the characters of that pattern.

In the best case, when there is no any expected occurrence of the pattern in the text, the searching phase of ASCII-Based-RJ algorithm takes a constant time O(1). Therefore, in the best case, the overall time complexity of the algorithm is O(n).

In the worst case, the searching phase of the algorithm scans (m-1) characters (i) times, where (i) is the number of expected occurrences of the pattern in the text. Thus, the algorithm takes $O((i\times m)-i)$ time in the worst case. The ASCII-Based-RJ algorithm requires (95) additional space for the ASCII_Arr array, (n) space for the SKIP_Arr array and (n-m+1) space for the Occurrence_List array. So, it needs (2n-m+96), which is O(n-m) extra space, in addition to the original text and pattern.

This study was conducted in 2009 as a part of the MSc thesis of Rami H. Mansi at the department of computer science at Al al-Bayt University, under supervision of Jehad Q. Alnihoud.

To compare between the performance of our algorithm with the naïve (brute force) and Boyer-Moore algorithms; we have built a string matching tool (a text editor) using Visual Basic 6.0 (Rami and Jehad-String Matching Tool (RJ-SMT)). In this tool, the ASCII-Based-RJ, Naïve and Boyer-Moore algorithms have been implemented and compared. The RJ-SMT is available at http://www.rjstringmatching.webs.com.

We have extensively tested the proposed algorithm on random test data. A simple program is developed to create random test patterns with different lengths (1 to 14) of characters. Both characters of patterns and strings were in the main memory, rather than a secondary storage medium. The total number of instructions that got executed and execution time in seconds were considered in the evaluation process. For each pattern length, 300 randomly selected samples were tested and averaged, while the total string length was 10,000 of randomly generated characters.

RESULTS AND DISCUSSION

Table 3 shows the experiment results of the tested algorithms. The execution time in seconds is denoted by (T) while the number of executed instructions is abbreviated by (Inst) for each algorithm.

In Table 3, the execution time (T) and the number of executed instructions (Inst) of an algorithm represent the average of 300 runs of the algorithm using the same pattern length (m) and random characters of the pattern at each run. Figure 1 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 utilizing ASCII-Based-RJ algorithm.

Table 3: Experimental results of the tested algorithms

	ASCII-Based-RJ		Brute force		Boyer-Moore	
Pattern length	T	Inst	T	Inst	T	Inst
1	3.45	10100	3.30	30000	3.50	29470
2	2.91	7000	3.50	30414	3.20	15592
3	3.11	9991	4.36	30421	3.61	10635
4	2.80	10000	4.85	30425	3.49	9225
5	2.90	9975	4.95	30461	3.20	9645
6	3.20	7000	4.98	30452	3.40	7945
7	3.00	10009	5.20	30480	3.20	6967
8	3.20	9996	5.30	30459	3.51	7672
9	3.60	10000	5.40	30501	3.70	7400
10	3.70	10012	5.50	30507	3.80	6660
11	3.80	7300	5.80	30539	4.10	6230
12	4.10	8400	6.00	30543	4.40	7549
13	4.20	9900	6.30	30528	4.35	8566
14	4.40	10016	6.80	30535	4.50	10000

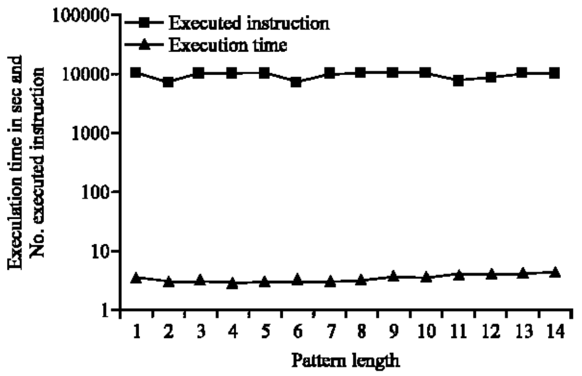

Fig. 1: Experimental results of ASCII-Based-RJ algorithm

It is apparent that the best performance of the ASCII-Based-RJ algorithm is when the pattern was relatively long (more than 3 characters). This result is reasonable, because the segments of the text that will be excluded during the searching phase increase as the pattern gets longer. Figure 2 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 utilizing brute force algorithm.

The best performance of the brute force algorithms is when the length of the pattern was relatively short. Since the algorithm compares almost m characters at each index of the text, the execution time increases as m gets larger.

Figure 3 shows the average number of the executed instructions and the average execution time in seconds for each patterns sample of each pattern length from 1 to 14 when the Boyer-Moore algorithm is used.

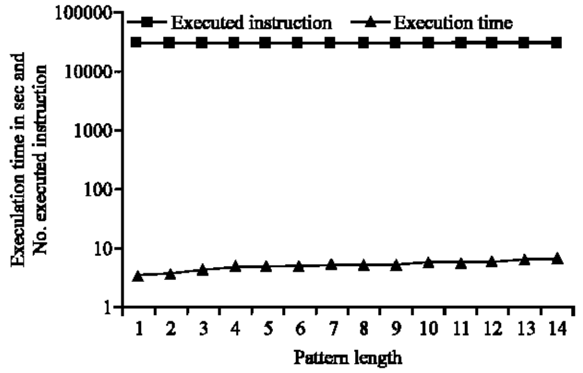

Fig. 2: Experimental results of the brute force algorithm

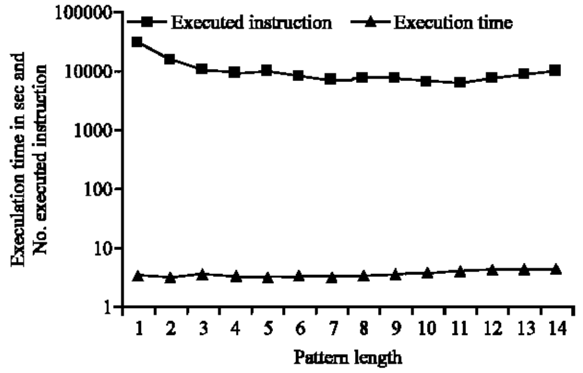

Fig. 3: Experimental results of the Boyer-Moore algorithm

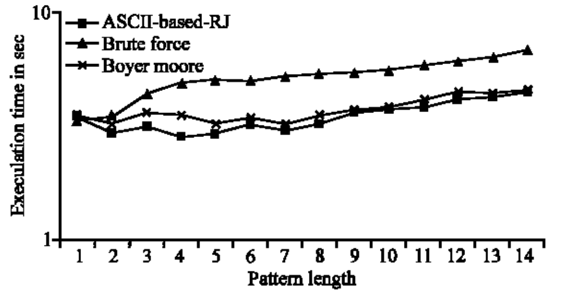

Fig. 4: Execution times in seconds of the tested algorithms

It is apparent that the best performance of the Boyer-Moore algorithm is when the pattern was relatively long (more than 4 characters). This result is logical, because the algorithm collects more information about the pattern when it is long.

Figure 4 shows a comparison between execution times of the ASCII-Based-RJ, Brute Force and Boyer-Moore algorithms for each patterns sample of each pattern length from 1 to 14.

It is apparent that the ASCII-Based-RJ algorithm outperforms the performance of the brute force and Boyer-Moore algorithms. It is clear that the proposed algorithm enhances the execution time of string matching as compared to the brute force and Boyer-Moore algorithms (Table 4). This enhancement is calculated by considering the differences in execution times of the algorithms to search for 14 patterns samples as recorded in Table 3. The enhancements of the proposed algorithms as compared to the naïve (brute force) and Boyer-Moore algorithms were 35.3 and 2.3%, respectively.

Table 4: Percentages of enhancements in execution time

Algorithm	Enhancement on brute force (%)	Enhancement on boyer-moore (%)
ASCII-Based-RJ	35.3	2.3

CONCLUSIONS

In this study, a new algorithm for improving the performance of single exact pattern matching is proposed. The ASCII-Based-RJ algorithm creates an array called ASCII_Arr of size (95), which equals the number of printing characters in the ASCII table. Then, it scans the characters of the pattern and for each character it increments by one the value of the ASCII_Arr at index equals to the code of that character in the ASCII table minus 32. After that, the algorithm scans the characters of the text from right to left starting at the last one. If the corresponding value of the current character (z) of the text in the ASCII_Arr array is zero, then this means that this character does not occur in the pattern. Thus, the range from (z-m+1…z) in the text will be ignored during the searching phase, since the pattern cannot start occurring at any index of this range in the text. The algorithm saves this range of indices in the SKIP_Arr array, which is of length *n*, where *n* is the length of the text. The algorithm overwrites the values of the elements of indices of the range (z-m+1…z) in the SKIP_Arr array by the value (-1) to be skipped during searching for the first, middle and last characters of the pattern. At this stage, the algorithm saves the indices of the text's segments where their first, middle and last characters equal the pattern's first, middle and last characters, respectively, in the Occurrence_List array to be used during the searching phase. We developed a string matching tool (RJ-SMT) to simulate and test the proposed algorithm, in addition to the naïve (brute force) and Boyer-Moore algorithms and we have extensively tested these algorithms on random test data. A simple program is developed to create random test patterns with different lengths (1 to 14) of characters. The total number of instructions that got executed and execution time in seconds were considered in the evaluation process. For each pattern length, 300 randomly selected samples were tested and averaged, while the total text string length was 10,000 of randomly generated characters. Based on the experimental results, the ASCII-Based-RJ algorithm outperformed the naïve algorithm by 35.3% and Boyer-Moore algorithm by 2.3%.

REFERENCES

Alqadi, Z., M. Aqel and I. El-Emary, 2007. Multiple-skip multiple-pattern matching algorithm (MSMPMA). IAENG Int. J. Comput. Sci., 34: 14-20.

Amintoosi, M.H.Y., M. Fathy and R. Monsefi, 2006. Using pattern matching for tiling and packing problems. Eur. J. Operat. Res., 183: 950-960.

Boyer, R.S. and J.S. Moore, 1977. A fast string searching algorithm. Commun. ACM., 20: 762-772.

Charras, C. and T. Lecroq, 2004. Handbook of Exact String-Matching Algorithms. 1st Edn., Kings College, London, ISBN: 978-0-7546-6498-7, pp: 19-24.

Crochemore, M., A. Czumaj, L. Gasieniec, S. Jarominek, T. Lecroq, W. Plandowski and W. Rytter, 1994. Speeding up two string matching algorithms. Algorithmica, 12: 247-267.

Crochemore, M.C.H. and T. Lecroq, 2003. A unifying look at the apostolico-giancarlo string-matching algorithm. J. Disc. Alg., 1: 37-52.

Danvy, O. and H. Rohde, 2006. On obtaining the boyer-moore string-matching algorithm by partial evaluation. J. Inform. Proc. Lett., 99: 158-162.

Franek, F., C. Jennings and W.F. Smyth, 2006. A simple fast hybrid pattern-matching algorithm. J. Disc. Alg., 5: 682-695.

Karp, R.M. and M.O. Rabin, 1987. Efficient randomized pattern-matching algorithms. IBM. J. Res. Dev., 31: 249-260.

Kim, S. and Y. Kim, 1999. A fast multiple string-pattern matching algorithm. Proc. 17th AoM/IAoM Int. Conf. Comput. Sci., 17: 44-49.

Knuth, D.E., J.H. Morris and V.R. Pratt, 1977. Fast pattern matching in strings. SIAM J. Comput., 6: 323-350.

Lecroq, T., 2007. Fast exact string matching algorithms. J. Inform. Process. Lett., 102: 229-235.

Morris, J. and V. Pratt, 1970. A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley.

Navarro, G. and K. Fredriksson, 2004. Average complexity of exact and approximate multiple string matching. J. Theor. Comput. Sci., 321: 283-290.

Sheu, T.F., N.F. Huang and H.P Lee, 2008. Hierarchical multi-pattern matching algorithm for network content inspection. J. Inform. Sci., 178: 2880-2898.

Watson, B., 2002. A new regular grammar pattern matching algorithm. J. Theor. Comput. Sci., 299: 509-521.

Watson, B. and R. Watson, 2003. A Boyer-Moore-style algorithm for regular expression pattern matching. J. Sci. Comput. Prog., 48: 99-117.

Wu, Y.C., J.C. Yang and Y.S. Lee, 2007. A weighted string pattern matching-based passage ranking algorithm for video question answering. J. Expert Syst. Appl., 34: 2588-2600.