Using Abstraction Methods to Improve String Matching Algorithms

Article · March 2012

CITATIONS
0

READS
48

2 authors:

Raaid Alubady
University of Babylon
18 PUBLICATIONS **21** CITATIONS
SEE PROFILE

Mehdi Ebady Manna
University of Babylon
12 PUBLICATIONS **34** CITATIONS
SEE PROFILE

Journal of Babylon University/Pure and Applied Sciences/ No.(1)/ Vol.(22): 2012
College of Science/Babylon University Scientific Conference

Using Abstraction Methods to Improve String Matching Algorithms

Raaid N. Alabaedy, Mehdi Ebadi Manaa, Adib M. Monzer Habbal
University of Babylon
raaid_alabaedy@yahoo.com , meh_man12@yahoo.com , adib@uum.edu.my

Abstract-

Enhancing or upgrading the existing algorithms needs big efforts to implement the existing one and after to on the problem or the limitation in these algorithms. As such, we need to enhance or upgrade this algorithm or its step to be efficient. Many updating revised algorithms for string matching have been done in many areas. All the work relies on of how enhance the time matching to be efficient. This work introduces a good tool to enhance the existing Boyer-Moore algorithm. On the other hand, we have designed two new algorithms for string matching called RAM1 & RAM2 and developed the existing one called "enhance_BMA". The new algorithm depends on abstraction string method in Java's tool, which helps to find great results even in the worst case of matching. Finally, most of the Object-Oriented programming concepts have been achieved in this work.

We propose in this paper new algorithm using Java tool which we called it RAM1 & RAM2. The existing Boyer-Moore has problems when the pattern length is long. This leads to increase the time complexity which causes unstably for this algorithm. On the other hand, we enhance the Boyer-Moore Algorithm (BMA) for a large given pattern in the text by using abstraction methods.

Keywords: String Matching Algorithm, Boyer-Moore Algorithm, efficient time, Abstraction methods, JAVA tools.

1. INTRODUCTION

STRING matching problem scale has much attention over the years due to its importance in various computer applications such as text processing, information retrieval, computational biology and intrusion detection. All those applications require a highly efficient algorithm to find all the occurrences of a given pattern in the text [1]. String-matching algorithms is a very important subject in the huge data of text processing. They play a main role in theoretical computer science by providing the enhancement and challenge to develop these algorithms. Due to the large amount of data that save in linear files these days, we need to make the string matching algorithms efficient to search in these files, even if the processor speed and capacity of storage of computers increase regularly. String-matching algorithms look for an occurrence of a string (called a pattern) in a text.

In this paper, the pattern is referred to by x=x [0...m-1] of characters; while its length is m. The text is referred by y=y [0...n-1]; its length is equal to n. We need to find the pattern in a given text in finite set of characters called the alphabet (A). The text matching or searching in text patterns considers the main thing in the algorithm domain. We need to search the pattern in linear file or a large text in both ways either the searching for exact pattern or closer for it in words. Though the sequential and binary search algorithms are used to find one key inside a large text or digits [1], it is not useful for our paper here. On the other hand, string matching algorithms is considered the main concepts to execute software process inside the operating system. It also has a main role in computer science to solve complex problems [2]. Finally, in this paper, we need to search for a certain pattern (group of keys and characters) in a large text or other closer pattern [3] using abstraction methods by Java tool to satisfy the efficient time.

String matching is a traditional application of partial evaluation, and obtaining the search phases of linear-time algorithms has become a standard benchmark [8, 9]. The obtained algorithms include several efforts and non-trivial ones, notably the Brute left-to-right string-matching algorithm and simplified variants of the Boyer-Moore right-to-left string-matching algorithm play a good role in time efficiently [10,11].

There are many strings matching algorithm that performs the same think by finding an occurrence of pattern n in a given text m. So, due to the huge data that occurs in these days such as files for biological DNA, marketing and data warehousing, we need to find an efficient time for searching. For example: given a text string T and a pattern string P, find the pattern inside the text. For example

T: "If student wants to take good mark, so he musts to study"

P: "take go"

It is a simple example and after applying this input to any algorithm, it will give us the result (the position where the pattern is begun) we also can calculate the time for matching P in T, but if the text is too long and the pattern is being increased, it certainly will take more time to execute. This is a problem that we need to solve. It becomes necessary to find a new algorithm or enhance the existing one.

2. LITERATURE REVIEW

The concept of string matching algorithms has been used in many areas: for the text editor, Suleyman used the efficient approximate and dynamic matching of patterns using a labeling paradigm [4]. For security, Anti-virus uses string matching algorithms and plays an important role in today's Internet communication security.

Virus scanning is usually performed on email, web and file transfer traffic flows at intranet security gateways. The performance of popular anti-virus applications relies on the pattern matching algorithms implemented in these security devices [5]. More importantly, the user query such as Google's engine search has used the string matching algorithms. Search engines are resources to assist users in information retrieval. It is inherently predicated on users searching for information from their "user information needs" that result from the interpretation of the decisional problem [6].

The huge data that are concerned with DNA sequence file has been used widely in string matching algorithms. String matching algorithms are very important in bioinformatics. The [7]] adopted the Approximate String Matching in DNA Sequences, which performs better than a suffix array if the data structure can be stored entirely in the memory.

3. PROJECT OBJECTIVES

- Defining the project requirements by executing and implementing string matching algorithms such as Brute Force Algorithm, Boyer-Moore Algorithm and as standard tools for last evolution.
- Developing new algorithms as mentioned above which we have decided to call them RAM1 & RAM2 and enhancing the existing one (Boyer-Moore Algorithm).
- To evaluate our work, we need to compare the obtained results from new or enhanced algorithms with step no. 1
- Performing the OOP concepts on the proposed algorithm.
- Determining the limitations in the proposed algorithm and finally running the algorithms to record the final results.

4. DESIGN AND DEVELOPMENT

This study has designed two methods called RAM1, RAM2, and developed an existing BMA called "enhance_BMA" by using abstraction methods in OOP environment. Both the design and the development of the algorithms are discussed in the following subsections.

4.1 DESIGN

This section is devoted to the designing of two algorthims RAM1 and RAM2.

4.1.1 RAM1 ALGORITHM

Firstly, we assume that the pattern has paired elements that need to be checked in text T to see the matcher return a true condition, in this case, we call abstraction method ***substring(first_position, last_position-1***) that returns the part of the string S from the position I to position j-1 and its length same as pattern's length if the two comparisons have the same ASCII code. We called this method (building method) that compares the ASCII of the pattern with the matched ASCII (subtext) in the string. If true, it returns to the position otherwise, it calls another abstraction method that returns the first element occurrence after checked position. The abstraction method is ***indexOf(first_char_in_anotherString,current_ position)*** The pseudo code algorithm for this Algorithm as in figure (1) below:

```
RAM1(text, pattern) {
//EFECTS : If next or pattern are empty throw //EmpityStringException, otherwise
check pattern is in text, //return an index where first pattern is stored, otherwise return
// -1.
n←——   text length
m←——   pattern length
if(m>n)    // if true return -1
if(m==1)   // if true return the position of m in
             //the text
patt←——    rasc(pattern);   // call building method that
                              //calculate pattern ASCII
repeat
if(( pattern.charAt(0)==text.charAt(i) &&
     pattern.charAt(1)==text.charAt(i+1))
      &&(rasc(text,substring(i,m+i))==patt))   // check if the
                        //first and second element in pattern are
                        // the same in text if true, then check the
                        //ASCII is the same pattern's ASCII if also
                        //true then match
i←——   text.indexOf(pattern.charAt(0),i+1)    // otherwise
                                    //update the position in text
until (i<n&&n-i>=m)
return -1          // not match
```

Figure 1. RAM1 algorithm pseudo code

4.1.2. RAM2 ALGORITHM

We use another abstraction method call compareTo(Object), compares the string S with string P, if it returns 0 then S equal to T otherwise returns a positive number or negative. We see that this method reduces the matching time better than ASCII method that used by RAM1. The pseudo code of the RAM2 algorithm is shown in figure (2)

```
RAM2(text, pattern) {
//EFECTS : If next or pattern are empty throw //EmpityStringException, otherwise
check pattern is in text, //return an index where first pattern is stored, otherwise return
// -1.
n◄——   text length
m◄——   pattern length
if(m>n)   // if true return -1
if(m==1)  // if true return the position of m in
           //the text
repeat
if(( pattern.charAt(0)==text.charAt(i) &&
    pattern.charAt(1)==text.charAt(i+1))
     && (text,substring(i,m+i).compareTo(pattern))
                  // check if the first and second element in
                  //pattern are the same in text if true, then
                  //check the ASCII is the same pattern's
                  //ASCII if also true then match
i◄——   text.indexOf(pattern.charAt(0),i+1)   // otherwise
                                    //update the position in text
until (i<n&&n-i>=m)
return -1         // not match
```

Figure 2. RAM2 algorithm pseudo code

4.2 DEVELOPMENT

This section deals with enhancing the Boyer-Moore algorithm. This algorithm uses these techniques:

- Finding P in T by checking the last elements and moving backwards through P, starting at its end. If it's ok, then the researchers will compare the contents of the pattern with a portion of the text that is equal to the pattern's length (compared by using abstraction methods).
- When a mismatch occurs at Text.charAt(i) == x, the character in pattern "Pattern.charAt(j)" is different as "Text.charAt(i)" then apply Boyer-Moore jump techniques. This means that we remove this part from the original algorithm:

```
if (Pattern.charAt(j) == Text.charAt(i))
            if (j == 0)
             return i;  match
             else { // looking-glass technique
              i--;
              j--;
```

and added this part :

```
if (pattern.charAt(j) == text.charAt(i)&&
    text.substring(k,i+1).compareTo(pattern)==0 )
```

This code shows that using two abstraction methods substring (first_position, last_position-1) and compare to (another_ string). The first one returns the part of the string S starting from the position i to position j-1 and the second compares the string S with string P if this method returns zero that means pattern's elements are the same subtext's elements. The enhanced Boyer-Moore algorithm is effective when the

pattern's length is long because it reduces the number the time of test by compare just one at a time.

5. VALIDATION AND EVALUATION

The user interface consists of two main interfaces. The first one shows many strings matching algorithm options that used in this work as an initial step for our evaluation work, which including two TextBox fields. These fields allow to input the text and the pattern to find time complexity and location for the first element. This position shows the matched position for the pattern in a given text.

The second interface shows the figure that contains the time result for these algorithms. The length of different patterns has been used to be searched in a given text such as (pattern with length =5,10,15,20,25,30,35,40,45,50 and so on) with a given text length with 100 characters as an example. Figure (3,4 and 5) shows this one.

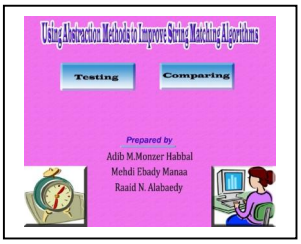

Figure 3. Entrance interface

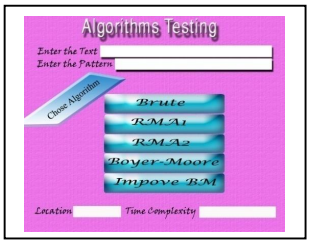

Figure 4. Algorithms Interfaces

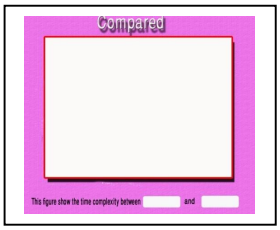

Figure 5. Compare Time Interface

If you choose testing bottom, the second interface appears to have many search methods and many fields. Firstly, we enter the text that we need to search on it. Secondly, we enter the pattern that we want to search it in text and we enter two search methods. This work has five string methods. that two of them such as Brute and Boyer_Moore, and two new methods are designed as RAM1 and RAM2. The last one is the enhancing of Boyer_Moore. In the same time, the result shows in textBox field that returns the location for matching and -1 for mismatch. Execution time will be shown in the second textbox which called time complexity (This time represents the amount of time that this method needs to search). By repeating the attempts, we can get many different time complexities. As an example, the figures (6, 7) show the different execution time between Brute method and RAM1 methods as shown below:

Text= i will i for will sall defill bagnrll dorhll

Pattern= dorhll

By running two algorithms, we got execution time for 5.634098E-4 for Brute method and 4.2778882E-4 for the proposed method RAM1. So, the last one is better than brute by 1.36E-04.

Figure 6. Brute Time Complexity

Figure 7. RAM1 Time Complexity

The important thing to show here is that the time for the implementation in term of 1000 times. Another execution will be specified for another method (RAM2) for the

same text and pattern. We got execution time 2.97055E- for this method. Therefore, the last one is better than RAM1 & Brute methods. See figures (7 and 8).

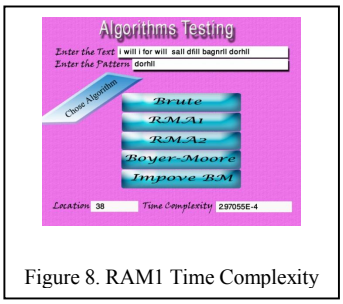

Figure 8. RAM1 Time Complexity

Another good example that we used here, is the execution time between the existing method of Boyer-Moore and the proposed one that we enhance it Boyer-Moore (Enhance). Let us take the following example.

Text= aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa

Pattern= baaaaaaaaaaaaaaaaaa

When we select Boyer-Moore method, the executed time appears is 0.0047010374, and when we select Boyer-Moore (Enhance) method the execution time is 0.0024318534. So we got the result that the difference between two methods is 2.27E-03. This is amazing work because the last one is better than the existing algorithm. See figures (9 and 10).

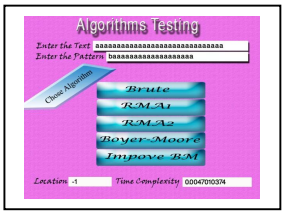

Figure 9. Boyer-Moore Time

Figure 10. Boyer-Moore (Enhance) Time Complexity

Comparing results can be obtained by pressing the compare button. The comparing interface will appear. Entering two algorithms as an example between Boyer-Moore and Boyer-Moore (Enhance) methods or RAM2 and Brute methods. The obtained results will be shown for 10 time complexity by using 10 input pattern's length. See the figure (11) below.

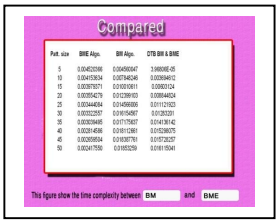

Figure11. Different Time Complexities for 10 Pattern's Length

Before evaluating our work, we need to explain the meaning of time complexity. Time complexity means that the algorithm takes more than one case (best, average and worst case), will focus on the worst case if the algorithm vibrates in many cases because it contains many complexities can also get rid of the difficulties in cases where the selected transactions (Properties example) are not appropriate or sufficient alone to determine the time adjusted . The three cases can be described as below:

- First: The Best Case, the lowest time needed to implement the algorithm (in our work when the pattern is at the beginning of the text).

- Second: The worst case is the maximum time needed to implement the algorithm (in our work when the pattern is non-existent).
- Third: The average case, its the medium time to implement the algorithm when the algorithm is not the best or worst case (in our work when the pattern be anywhere between the beginning or end of text)

Tables below show us the time complexity between all methods that our work implemented. Table (1) shows the best case for all methods. Table (2) shows the average case, and Table(3) shows the worst case as in below.

Table (1): Best Case for String Matching Algorithm

size	Brute	RMA1	RMA2	Boyer-Moore	Boyer-Moore (Enhance)
5	7.75E-05	1.70E-04	1.28E-04	1.11E-03	6.07E-04
10	1.15E-04	2.63E-04	1.19E-04	1.14E-03	6.60E-04
15	1.44E-04	3.48E-04	1.33E-04	1.18E-03	6.34E-04
20	1.82E-04	4.43E-04	1.43E-04	1.27E-03	6.73E-04
25	2.10E-04	5.58E-04	1.63E-04	0.001295411	6.71E-04
30	2.40E-04	6.78E-04	1.71E-04	0.001381326	7.15E-04
35	2.77E-04	7.23E-04	1.85E-04	0.001415913	7.66E-04
40	3.13E-04	7.96E-04	1.95E-04	0.001475956	7.73E-04
45	3.43E-04	8.78E-04	2.15E-04	0.001515563	8.54E-04
50	3.69E-04	1.00E-03	2.23E-04	0.001579749	8.32E-04

Table (2): Average Case for String Matching Algorithm

Pattern Size	Brute	RMA1	RMA2	Boyer-Moore	Boyer-Moore (enhance)
5	2.55E-04	5.48E-04	4.14E-04	1.17E-03	6.97E-04
10	4.35E-04	8.53E-04	5.05E-04	1.16E-03	7.75E-04
15	5.89E-04	1.17E-03	6.03E-04	1.24E-03	8.52E-04
20	7.96E-04	1.49E-03	7.03E-04	1.31E-03	8.77E-04
25	9.89E-04	1.78E-03	7.81E-04	1.38E-03	9.15E-04
30	1.16E-03	2.08E-03	8.59E-04	1.48E-03	9.37E-04
35	1.35E-03	2.42E-03	9.96E-04	1.58E-03	9.77E-04
40	1.53E-03	2.72E-03	1.04E-03	1.65E-03	9.88E-04
45	1.69E-03	3.04E-03	1.14E-03	1.74E-03	1.02E-03
50	1.90E-03	3.38E-03	1.24E-03	1.82E-03	1.05E-03

Table (3): Worst Case for String Matching Algorithm

size	Brute	RMA1	RMA2	Boyer-Moore	Boyer-Moore (Enhance)
5	0.003446268	0.007976771	0.005604596	0.004560047	0.004520366
10	0.005984154	0.011884106	0.006829948	0.007848246	0.004153634
15	0.008555052	0.015569291	0.008016096	0.010010611	0.003979371
20	0.010878398	0.017921148	0.008905751	0.012399103	0.003554279
25	0.012306026	0.020836372	0.009922604	0.014566006	0.003444084
30	0.013604702	0.02296475	0.010350681	0.016154567	0.003322557
35	0.014682132	0.02422422	0.010608391	0.017175637	0.003039495
40	0.015168933	0.02543041	0.010826495	0.018112661	0.002814586
45	0.016123831	0.026798157	0.011472889	0.018387761	0.002659504
50	0.016139185	0.02607698	0.011050945	0.01853259	0.002417550

The evolution step is shown clearly in the following charts. It is a good tool to recognize that by the abstraction, we got efficient results. See the figure (12) below:

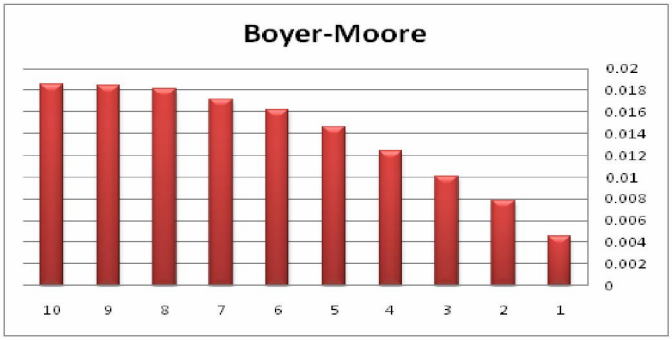

Figure12. Boyer-Moore Execution Time Chart

In original Boyer-Moore, when the pattern's length reduces, the time will be increased. But this situation is inversed in Boyer-Moore Enhanced. When the pattern's length reduces, the time will be decreased. Finally, Boyer- Moore Enhanced achieves what is the purpose from this work.

Figure13. Boyer-Moore Execution (Enhance) Time Chart

6. CONCLUSION

This work has added a good contribution for enhancing the string matching algorithm. This contribution has been done by using abstraction method in Java which is considered as a very helpful language in such algorithms. We got good results for the new RAM1 & RAM2 and enhance_BMA that is developed from the BM original in this work. But, the amazment of obtained results has been found by enhancing the Boyer-Moore algorithm using these abstraction methods such as "charAt", "substring", "compareOf" and "indexof". Finally, we evaluated our results according to the existing algorithms using different pattern's length as shown in tables above and this step is done by executing two existing algorithms such as Boyer-Moore and brute. Our conclusion can be summarized in the following steps:

- Enhancing the existing string matching algorithms by using abstraction methods and to reduce execution time when pattern's pattern's length is long (enhanced Boyer-Moore algorithm).
- Reducing execution time in a given small alphabetic by using two new algorithms (RAM1, RAM2).
- These algorithms provide more stability for string matching algorithms by using abstraction methods.

References

[1] Naijie, Xiaohu, & Gang. (2005). A Practical Distributed String Matching Algorithm Architecture and Implementation. (pp. 196- 200). World Academy of Science, Engineering and Technology.

[2] Rohde, H. K., & Danvy, O. (2005). On Obtaining the Boyer-Moore String-Matching Algorithm by Partial Evaluation. Basic Research in Computer Science (pp. 1-9). Denmark: BRICS Report Series.

[3] Oregon, O. O. (2003). Performance Analysis of String Pattern Matching Algorithms. Prosiding Seminar Penyelidikan Jangka Pendek , 1-5.

[4] ahinalp, C. S. (1996). Efficient Approximate and Dynamic Matching of Patterns Using a Labeling Paradigm. IEEE , 320-328.

[5] Zhou, X., Xu, B., Qi, Y., & Li1, J. (2008). A Fast Pattern Matching Algorithm for Anti-virus Applications. IEEE , 256-261.

[6] Onifade, F. W., Thiéry, O., Osofisan, ,. O., & Duffing, G. (2010). Dynamic Fuzzy String-Matching Model for Information Retrieval Based on Incongruous User Queries. Proceedings of the World Congress on Engineering 2010 Vol I. London: WCE.

[7] Cheng, L.-L., Cheung, D. W., & Yiu, S. M. (2003). Approximate String Matching in DNA Sequences. Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA'03). IEEE.

[8] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. Partial Evaluation and Automatic Program Generation. Prentice-Hall International, London, UK, 1993. Available online at http://www.dina.kvl.dk/~sestoft/pebook/.

[9] Morten Heine Sørensen, Robert Gl¨uck, and Neil D. Jones. A positive supercompiler. Journal of Functional Programming, 6(6):811–838, 1996.

[10] Christian Charras and Thierry Lecroq. Exact string matching algorithms. http://www-igm.univ-mlv.fr/~lecroq/string/, 1997.

[11] Christian Charras and Thierry Lecroq. Exact string matching algorithms. http://www-igm.univ-mlv.fr/~lecroq/string/, 1997.