Start-to-End Algorithm for String Searching

Rawan A. Abdeen

Al- Balqa' Applied University, Salt, Jordan

Summary
String searching is a very important component of many problems, including text editing, text searching and symbol manipulation. In this paper a string searching algorithm is proposed as an improvement of the brute-force searching algorithm. The algorithm is named Start-to-End Algorithm. The proposed algorithm does not preprocess neither the *pattern* nor the *text to* perform searching.

Key words: String searching, pattern, start-to-end algorithm.

1. Introduction

Although we deal with data in a lot of forms, text remains the main form to exchange information and take advantage of it.

String searching sometimes called **string matching** is concerned in finding the occurrences of a **substring** (called the ***pattern***) of length **m** in a **string *(called the text)*** of length **n** (where $n \geq m$) [1-3].

In order to search for a pattern within a string, an algorithm is needed to find the pattern as well as to know the locations where it was found in a given sequence of characters.

A lot of algorithms were created to perform string searching. Each algorithm uses a specific strategy to perform the search. Some need to preprocess the pattern [4-6]. Others need to preprocess the text; also there are algorithms that require both the pattern and the text to be preprocessed before searching [7] and some do not perform preprocessing neither for the text nor for the pattern.

One of the simplest string searching algorithms is the Brute-force algorithm. It is the least efficient way to check whether one string occurs inside another.

Various string searching algorithms were created to improve the Brute-Force algorithm. From those algorithms: **the Knuth-Morris-Pratt** (KMP), **Boyer-Moore (BM)** and **Karp and Rabin** algorithms [1][8]. Still to determine which of the algorithms is the best to use depends on the application were the algorithm is to be applied.

The **Knuth-Morris-Pratt** (KMP) algorithm uses information about the characters in of the pattern to determine how much to move along that string after a mismatch occurs [9][10]. The **Rabin–Karp** algorithm computes a hash function to seek for a pattern within a given text [10]. **The Boyer-Moore** algorithm works by searching the target string from right to left, while moving it left to right [9].

2. Brute-Force Algorithm

Brute-force algorithm, which is also called the **"naïve"** is the simplest algorithm that can be used in pattern searching. It is probably the first algorithm we might think of for solving the pattern searching problem. It requires no preprocessing of the pattern or the text [11].

The *idea* is that the pattern and text are compared character by character [8][10]; in the case of a mismatch, the pattern is shifted one position to the right and comparison is repeated, until a match is found or the end of the text is reached [1].

The algorithm works with two pointers; a **"text pointer" i** and a **"pattern pointer" j**. For all (n-m) possibly valid shifts, pattern and text are compared; while text and pattern characters are equal, the pattern pointer is incremented. If a mismatch occurs, **i** is incremented, **j** is reset to zero and the comparing process is restarted. In case a match is found, the algorithm returns the position of the pattern; if not, it returns not found message [9, 11].

The worst case will happen if all the characters of the pattern were matched with the text segment except the last one.

Referring to the algorithm, the outer for-loop is executed at most **n-m+1** times and the inner loop is executed at most m times. Thus, the running time (*time complexity*) of the brute force algorithm is: **O((n-m+1)m)** which is **O(nm)** [8]. In the worst case, when **n** and **m** are equal, this algorithm has a quadratic running time [1].

3. Start-To-End Algorithm

This algorithm finds all the occurrences of the pattern in the text. It does not require performing preprocessing neither for the text nor for the pattern.

The *idea* is that the first and last characters of the pattern are first compared to the corresponding first and

Manuscript received February 5, 2011
Manuscript revised February 20, 2011

last characters of the segment taken from the text, in which we first start by comparing the first character in the pattern with the first character in the text, if they match, then we continue by comparing the last character in the pattern with the last character in the text, if a match occurs then proceed by matching the rest of the characters of the pattern with the rest of the characters of the segment, character by character. In the case of a mismatch while performing character by character comparison, we directly take the next segment of the text shifted one position from the previous one, else continue comparing. If all the characters of the pattern match with the characters of the segment then signal that the pattern was **found** and at which **location** in the text it was found. After that, proceed with the next segment to find other occurrences of the pattern in the text. If we have scanned all of the segments of the text without matching the pattern with any of the introduced segments a **not found** signal is performed. **Fig. 1** illustrates the algorithm in a flowchart to find the first occurrence of the pattern within the text.

3.1 Algorithm Steps

Step 1: Divide the text into segments in which the first segment begins from element at index 0, the second segment begins at the element of index 1 and so on. That is each segment to be taken is shifted one character than the previous one.

Step 2: Compare the first character of the pattern with the corresponding first character of the segment taken, **if a match occurs**, then go to the **next step**. If a **mismatch** occurs, then take the **next segment** and **repeat step 2**.

Step 3: Compare the last character of the pattern with the corresponding last character of the segment taken, if a match occurs, then go to the next step, else if a mismatch occurs, then take the **next segment** of the text and go to **step 2**.

Step 4: Perform character by character comparison for the rest of the characters of the pattern with the rest of the characters of the segment taken (that is without considering the first and last characters in the comparison). If a mismatch is encountered while matching in any step of the comparison, then we stop comparing and proceed with the next segment for comparison and go to step 2, else continue comparing. If all the characters of the pattern match with the characters of the segment then signal that the pattern was **found** and at which **location** in the text it was found. After that, proceed with the **next segment** and repeat **step 2**, to search for other occurrences of the pattern in the text.

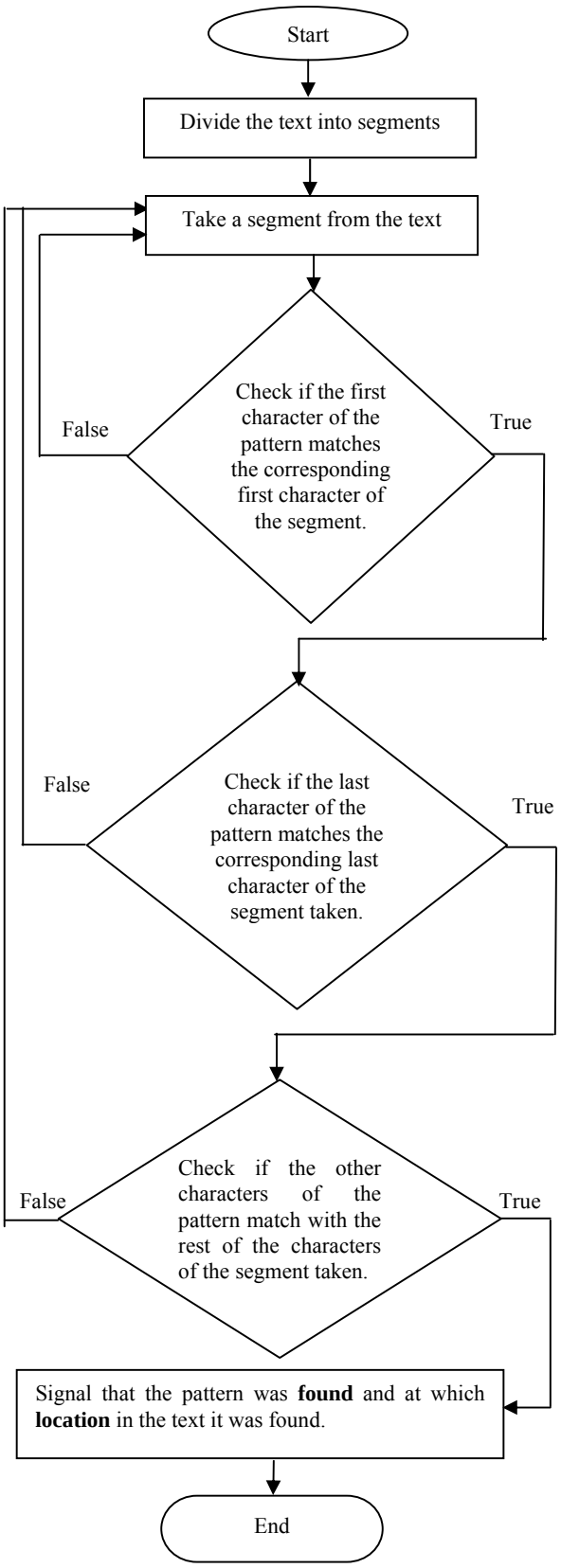

Fig. 1: A flowchart for the proposed algorithm to find the first occurrence of the pattern within the text.

4. Results

This algorithm finds all the occurrences of the pattern in the text. The improvement process for the Brute-Force algorithm within the proposed (Start-to-End) algorithm does not require performing preprocessing for the pattern as the other algorithms that has improved the Brute-Force algorithm. **Table 1** summaries the algorithms that has improved the brute-force algorithm with their time complexity.

The proposed algorithm begins the search process by comparing the first character of the pattern with the first character of the segment taken, if they match, then it compares the last character of the pattern with the last character of the segment, if a match occurs, then it will allow to perform character by character matching between the segment and the pattern, for the rest of the characters that remain without comparing.

The time complexity for the Start-to-End algorithm can be detailed as follows:

- If the first and last characters of the pattern does not match with the first and last characters of all the segments in the text, then the ***time complexity*** would be:
  O(((n-m)+1) * (m-2)).

- If the first and last characters match, then the ***time complexity*** would be: **O((n-m)+1).**

Table 2 illustrates the differences in the time complexity between the brute-force algorithm and the proposed start-to-end algorithm depending on an example where the number of characters of the text is 14 and of the pattern is 4.

Table 1: A summary for the algorithms that has improved the brute-force algorithm with their time complexity

Algorithm	Preprocessing the Pattern	Time Complexity
Brute-Force Algorithm	No preprocessing	O((n-m+1)*m)
Start-to-End Algorithm	No preprocessing	O(((n-m)+1)*(m-2))
Rabin-Karp Algorithm	Preprocesses the pattern	O(nm)
Knuth-Morris-Pratt algorithm	Preprocesses the pattern	O(n+m)
Boyer-Moore algorithm	Preprocesses the pattern	O(nm)

Table 2: The differences in the time complexity between the brute-force algorithm and the proposed start-to-end algorithm depending on an example where the number of characters of the text is 14 and of the pattern is 4

Description	Brute-force Time Complexity	Start-to-end Time Complexity
If the first and last characters of the pattern does not match with the first and last characters of all the segments in the text.	44	22
If the first and last characters match with one of the text segments	44	11

5. Conclusion

In conclusion, this paper has proposed a string searching algorithm as an improvement for the brute-force algorithm without the need to preprocess neither the pattern nor the text. The improvement that this algorithm has offered over the brute-force algorithm is that it does not allow performing character by character matching between the segment taken from the text and the pattern only after it checks that the first and last characters in the pattern match the first and last characters in the segment taken from the text. This process would improve the time of searching of the brute-force algorithm.

References

[1] Thierry Lecroq, "Experimental Results on String Matching Algorithms", SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 25(7), 727–765, 1995.

[2] Stephen G., "String Searching Algorithms", World Scientific, Singapore, 1994.

[3] Apostolico, "A, Galil Z. Pattern Matching Algorithms", Oxford University Press, 1997.

[4] Liu Z, Du X,and Ishii N., "An improved adaptive string searching algorithm", Software Practice and Experience, 1988, 28(2):191–198.

[5] Sunday D., "A very fast substring search algorithm", Communications of the ACM, 1990, 33(8):132–142.

[6] Bruce W., Watson, E., "A Boyer-Moore-style Algorithm for Regular Expression Pattern Matching", Science of Computer Programming, 2003, 48: 99-117.

[7] Fenwick P., "Fast string matching for multiple searches", Software–Practice and Experience, 2001, 31(9):815–833.

[8] Ohdan Masanori, Takeuchi Ryo And Satou Tadamasa, "An Evaluation of String Search Algorithms at Users Standing", Proceedings of the 3rd WSES International Conference on Mathematics and Computers in Mechanical Engineering (MCME), 2001, pp. 4231-4236, ISBN: 960-8052-35-1.

[9] Softpanorama, "Searching Algorithms", 2010, http://www.softpanorama.org/Algorithms/searching.shtml. Accessed on 8 Jan., 2011.
[10] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", 3rd edition, 2009, MIT Press.
[11] Michael T. Goodrich and Roberto Tamassia, "Algorithm Design", 2002, John Wiley and Sons, Inc.
[12] Hume and Sunday, "*Fast String Searching"* SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 21(11), 1221–1248, 1991.

Rawan A. Abdeen received the B.S. degree in Information Technology and M.S. degree in Computer Science from Al-Balqa' Applied University.