Journal of Computer Science 2(1): 82-85, 2006
ISSN 1549-3636
© 2006 Science Publications

Occurrences Algorithm for String Searching Based on Brute-force Algorithm

Ababneh Mohammad , Oqeili Saleh and Rawan A. Abdeen
Department of Information Technology, Al- Balqa' Applied University, Salt, Jordan

Abstract: This study proposes a string searching algorithm as an improvement of the brute-force searching algorithm. The algorithm is named as, *Occurrences algorithm*. It is based on performing *preprocessing* for the *pattern* and for the *text* before beginning to search for the pattern in the text.

Key words: Pattern, substring, string searching, occurrences algorithm

INTRODUCTION

Although we deal with data in a lot of forms, text remains the main form to exchange information and take advantage of it, thus a lot of operations have been made on texts.

Operations on strings often involves searching for the existence and the location of a substring within a sequence of characters.

String searching is concerned in finding the occurrences of a **substring** (called a ***pattern***) of length **m** in a ***text*** of length **n**. This process is an important step towards solving many problems, including text editing, text searching and symbol manipulation.

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.

There are a lot of algorithms that were created as improvements of the brute-force algorithm. Each algorithm tries to avoid problems that were encountered in the existed algorithms. Still to determine which of the algorithms is the best to use depends on the application were the algorithm is to be used.

From the string searching algorithms is the **Karp and Rabin** algorithm. It uses hashing which provides a simple method that avoids the quadratic number of character comparisons in most practical situations[1-3].

Assuming that the pattern length is no longer than the memory-word size of the machine, the **Shift Or** algorithm is an efficient algorithm to solve the exact string-matching problem and it adapts easily to a wide range of approximate string-matching problems[1,4]. **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[1,3,5].**The Boyer-Moore** algorithm works by searching the target string from right to left, while moving it left to right[1,3].

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[1,3,5-7].
The *idea* is that the pattern and text are compared character by character; 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.

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[1,2,6,7].

The worst case will happen if all the characters of the pattern matches 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)**. In the worst case, when **n** and **m** are equal, this algorithm has a quadratic running time.

OCCURRENCES ALGORITHM

This algorithm finds all the occurrences of the pattern in the text. It requires performing preprocessing of the pattern and the text before searching. Thus, the searching technique used in this algorithm is based on three processes. These processes are preprocessing of the pattern, preprocessing of the text and then depending on the results of the preprocessing, the searching process is performed.

Let ***cmax*** denotes the **character** with the highest number of occurrences in the pattern, which means ***cmax*** refers to the character that has the highest number of repentances in the pattern.

Correspondent Author: Ababneh Mohammad, Department of Information Technology Al- Balqa Applied University, Jordan

* **s:** is the array that contains the indexes of the segments that will be taken into consideration in the comparisons.
* **sc:** is the number of elements in the **s** array.

The *number of the elements* denotes the number of segments that are to be taken into consideration while matching.

Preprocessing the pattern: Preprocessing the pattern algorithm includes calculating the number of occurrences, that is the number of repetitions, for each character in the pattern (*note that*: the text in which you search in is to be divided into segments). After that, the algorithm finds the character that is of the highest occurrence in the pattern. Figure 1 shows the flowchart of this process.

Preprocessing the pattern

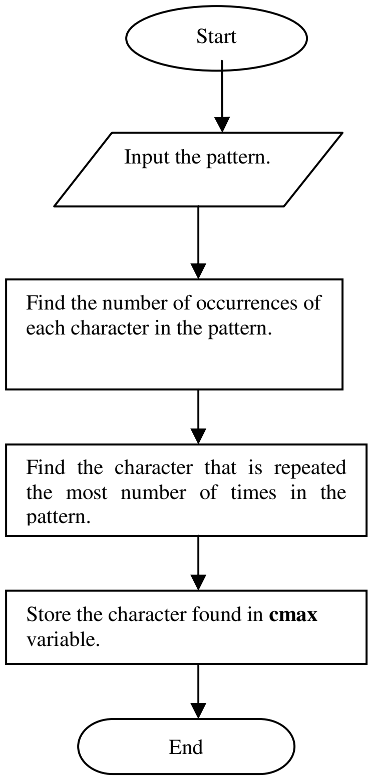
Start
Input the pattern.
Find the number of occurrences of each character in the pattern.
Find the character that is repeated the most number of times in the pattern.
Store the character found in **cmax** variable.
End

Fig. 1: Preprocessing the Pattern

Preprocessing the text: The algorithm finds the character that is of the highest occurrence in the pattern. This process depends on the character found in the previous process (preprocessing of the pattern process), which has the highest number of occurrences in the pattern. If this character is found in the segment of the text, then calculate the number of occurrences of that character in that segment. If the number calculated is equivalent to the number of occurrences of that character in the pattern, then store the *segment index* in an array. Figure 2 shows the steps through which the process is performed. At the end of the preprocessing of the text, the array will contain the indexes of the segments that will be considered in the comparison process.

Searching algorithm: After preprocessing the pattern and the text, the comparison will be done. This comparison will be between the pattern and the segments that their indexes are stored in the array. Figure 3 illustrates the searching process.

When the pattern does not include a character repeated more than once, then the algorithm depends on the first character in the pattern to search for it. Searching will be done in a similar manner as when searching for a pattern having characters repeated more than once.

IMPLEMENTATION

The main advantage of the proposed algorithm is its simplicity. Any programming language can easily implement it.

Occurrences algorithm has improved the way the brute-force algorithm searches for the pattern. Preprocessing the pattern and the text is performed before searching for the pattern in the text, which reduces the time complexity. When the pattern does not include a character repeated more than once, then the algorithm depends on the first character in the pattern to perform searching. Table 1 illustrates the differences in the time complexity between both algorithms, depending on a given text as an example.

RESULTS

The improvement that the occurrences algorithm has offered over the brute-force algorithm, is that it preprocesses the pattern and the text before performing the searching process. The preprocessing processes give additional information that can be used in order to facilitate the searching process. Eventually after the preprocessing processes have been executed, an array is created. This array will be used to determine which segments of the text will be compared with the pattern; thus in the searching process the segments that will be compared with the pattern are only those determined by the array, without having to traverse all the segments of the text to find the pattern.

Let ***cmax*** denotes the **character** with the highest number of occurrences in the pattern, which means ***cmax*** refers to the character that has the highest number of repetitions in the pattern.

* **s:** is the array that contains the indexes of the segments that will be taken into consideration in the comparisons.
* **sc:** is the number of elements in the **s** array.

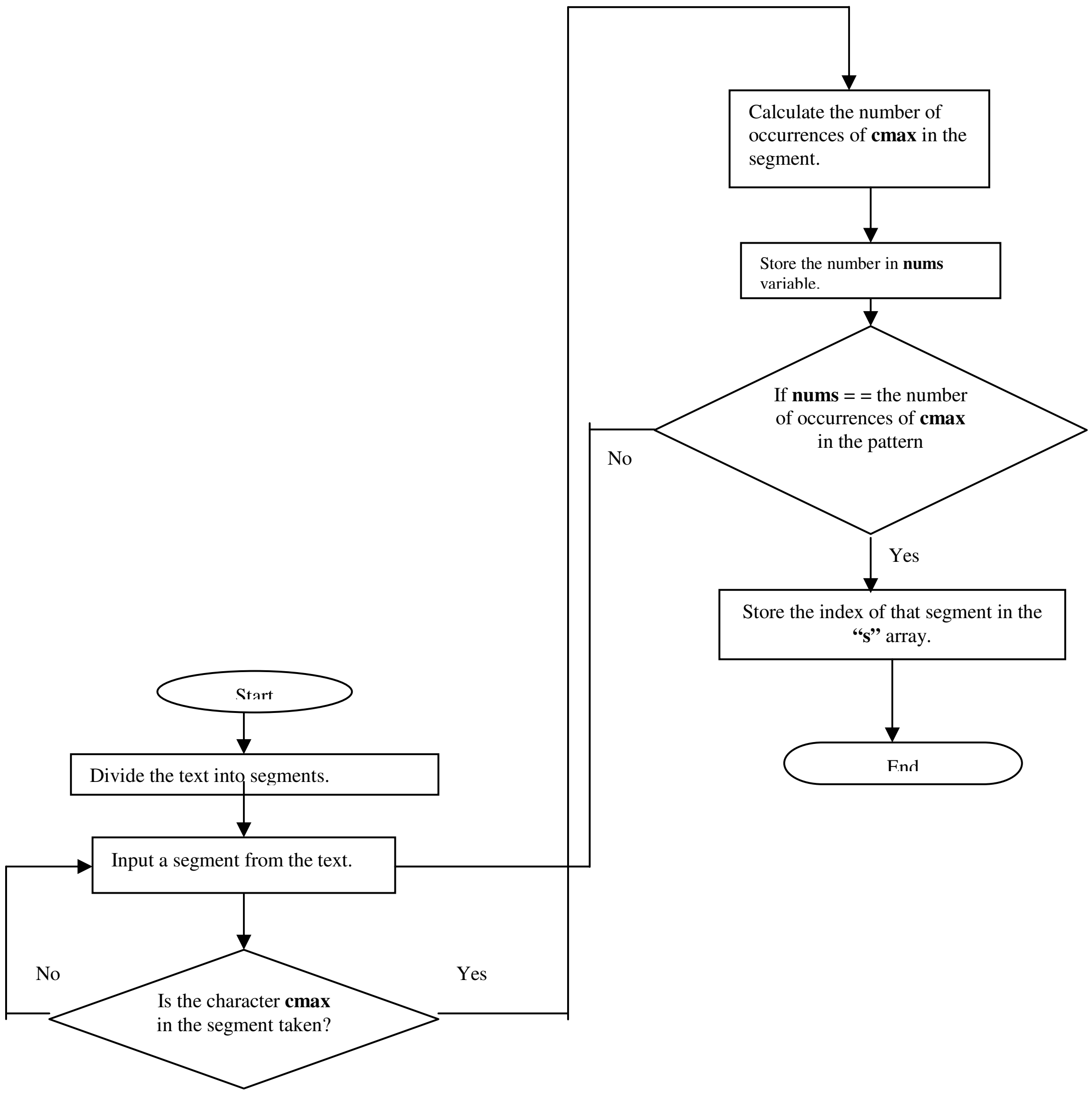
Calculate the number of occurrences of **cmax** in the segment.
Store the number in **nums** variable.
If **nums** = = the number of occurrences of **cmax** in the pattern
No
Yes
Store the index of that segment in the **"s"** array.
Start
End
Divide the text into segments.
Input a segment from the text.
No
Yes
Is the character **cmax** in the segment taken?

Fig. 2: Preprocessing the text

Table 1: The differences in the time complexity between both algorithms, depending on a given text as an example

Description	Example (the pattern)	Brute-force Time Complexity O((n-m+1)*m)	Occurrences Time Complexity O(sc*m)
If the pattern exists in the text and has a character that is repeated more than once.	Room	(20 - 4 + 1) * 4 17 * 4 = 68.	2 * 4 = 8.
If the pattern exists in the text and does not have a character that is repeated more than once.	om	19 * 2 = 38.	4 * 2 = 8.
If the pattern is not in the text and has a character repeated more than once.	moon	17 * 4 = 68.	2 * 4 = 8.
If the pattern is not in the text and does not have a character that is repeated more than once.	fb	19 * 2 = 38.	2 * 2 = 4.

The *number of the elements* denotes the number of segments that are to be taken into consideration.

Time complexity

* If ("sc**"** = = 0) (which means that the pattern is not in the text), then the *time complexity* would be: **O(1)**.
* **"cmax"** is found in the segment and is repeated in the segment in a number of times that is equal to its repentance in the pattern, that is **"sc"** is **not equal to** zero and "**s"** array includes the segments' numbers that are to be considered in comparison, then the ***time complexity*** would be:

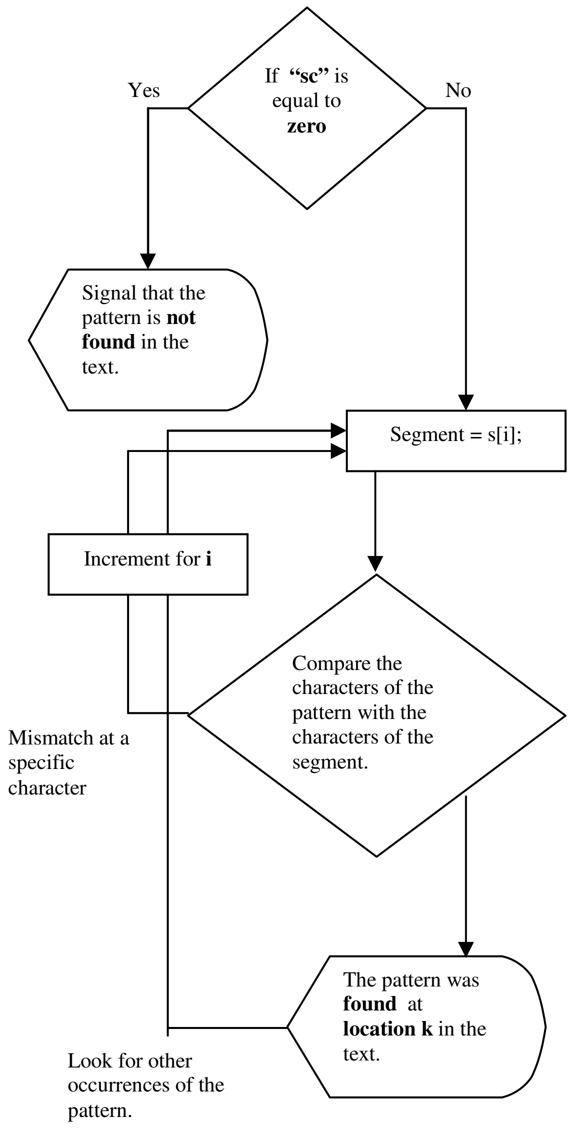
If "sc" is equal to zero
Yes
No
Signal that the pattern is not found in the text.
Segment = s[i];
Increment for i
Compare the characters of the pattern with the characters of the segment.
Mismatch at a specific character
The pattern was found at location k in the text.
Look for other occurrences of the pattern.

Fig. 3: Searching Process

O((sc)*m), where **sc** as mentioned earlier, is the number of elements in the **s** array, **"sc"** is either less than or equal to **n-m+1.**

CONCLUSION

In conclusion, this study has proposed a string searching algorithm as an improvement of the brute-force algorithm. **Brute-force** algorithm requires no preprocessing on the pattern or on the text.

While the **occurrences** algorithm requires performing preprocessing on the pattern and on the text before searching. It depends on the repetition of a character that is found to be repeated the most in the pattern.

Furthermore, because the quantity of available data in the fields where string searching is used tend to double by time, the algorithms to be used should be efficient even if the speed and capacity of storage of computers increase regularly. Thus, we always need to create algorithms that can perform in a faster and more efficient manner than those existing.

REFERENCES

1. Christian Charras. Introduction. http://www-igm.univ-mlv.fr/~lecroq/string/node2.html#SECTION0020. Accessed on 1 Nov., 2003.
2. Cormen, Leiserson, Rivest, 1990. Introduction to Algorithms. MIT Press.
3. National Institute of Standards and Technology (NIST), 2004. String Matching. http://www.nist.gov/dads/HTML/stringMatching.html. Accessed on 20 Jul., 2004.
4. National Institute of Standards and Technology (NIST), "Shift-Or", 2004, http://www.nist.gov/dads/HTML/shiftOr.html. Last visited: 20 July 2004.
5. Alison Cawsey, 1998. String Searching. http://www.cee.hw.ac.uk/~alison/ds98/node74.html Accessed on 21 Jun., 2004.
6. Christian Charras, "Brute Force algorithm. http://www-igm.univ-mlv.fr/~lecroq/string/node3.html#SECTION0030. Accessed on 1 Nov., 2003.
7. Michael T. Goodrich and Roberto Tamassia, 2002. Algorithm Design. John Wiley and Sons, Inc.