International Journal of Recent Research and Review, Vol. VII, Issue 2, June 2014
ISSN 2277 – 8322

Implementation of an Unique & Fast String Matching Algorithm

Suresh Fatehpuria, Ankur Goyal
Computer Science Engg., Rajasthan Technical University, Yagyavalkya Institute of Technology, India
Email: sureshjecit@gmail.com

***Abstract*— There are many number of fields where the string matching application may be needed. The fields like processing of signals in telecommunication, searching DNA patterns, searching a word pattern in a Word document or over the web are some of the examples of string matching. The algorithm presented in this paper is a very unique idea for finding a string in the given text. The algorithm preprocesses the whole string by indexing all the characters of the string followed by storing the indexes in a two dimensional array. The resulting two dimensional array contains the index entry of every occurrence of every character present in the string. In the second phase i.e. the matching phase, the pattern is rolled over the given string until the match is found (or a match doesn't occur). Initially or when the mismatch occurs then for shifting the pattern on every mismatch, the help of two dimensional array is taken since it contains all the indexes of all the characters present in the text and consequently it ensures relevant shifts only. The time complexity for matching processes is very much less as compared to other proposed algorithms because of a very few numbers of shifts are made with the help of two dimensional array.**

***Keywords*— Pattern, shifting, matching, efficient, complexity.**

I. INTRODUCTION

The problem of pattern matching is one of the extensively studied problems in Computer Science because of its large number of applications in different areas like text processing, image processing, speech analysis, data compression, bio informatics etc. The problem of the pattern matching is to find all the occurrences of a pattern P of length m in text T of length n [1].

Due to being very important concept, there have been numerous algorithms to match a pattern in the text. Some of those are less efficient and are in less use; some of those are more efficient and consequently are in more use. Some of those are very specific to the situation where one should use those. The efficiency and performance of an algorithm is measured with the help of some techniques named time complexity and the space complexity. The notations like O, Θ and Ω are used in finding the efficiency of an algorithm or code.

As the time is passing on, people are trying to execute everything in faster manner. In such a situation reducing the time complexity is really required. Although the space complexity doesn't matter in some of the cases since the memories are getting cheaper day by day, still the researchers are trying to reduce the space complexity too.

II. LITERATURE SURVEY

There have been a big number of algorithms for finding a pattern in a given text. The most basic algorithm that doesn't include any pre-processing of data is the Naïve or Brute Force algorithm [1]. The time complexity of the searching phase of Brute Force algorithm is O (mn). In practical, almost all the algorithms perform pre-processing of the pattern and then find all the valid shifts. So in general, a pattern matching algorithm have two of the phases: one is "pre-processing phase" in which the pattern is pre-processed and another one is "matching phase" in which a match is tried to find. A brief of some famous algorithms are stated below.

The Rabin Karp algorithm [2] proposed by Michal O. Rabin and M. Karp in 1987 uses hashing to find a pattern in a given text. Although the worst case running time for the algorithm is as much as Naïve algorithm's complexity is, still it works better in average case. Rabin and Karp assume that each and every character of the alphabet $\sum$ is a decimal digit. In general case we can assume that each character is a digit in radix-d notation, where $d = |\sum|$ and $\sum = \{0, 1, 2\ldots9\}$. Then a string of k consecutive characters as representing a length - k decimal number can be viewed [1]. The time complexity of preprocessing phase is O (m) and of searching phase is O (n+m).

The Knuth-Morris-Pratt algorithm [3] or KMP algorithm was first thought by Donald Knuth & Vaughan Pratt and independently by James H. Morris in 1974. They all three people published it jointly in 1977. In this algorithm, when the pattern is tried to match with the text then the partial part (but not the full pattern) of the pattern that got matched (if any), is remembered with the use of "prefix function". With the help of that partial part we can determine the corresponding text characters. This allows us to determine that certain shifts are invalid. This phenomenon of skipping certain shifts avoids the repetitive full text searching and consequently reduces the complexity. The time complexity of preprocessing phase is *O* (m) and of searching phase is *O* (nm).

The Boyer Moore Algorithm [4] or BM algorithm was proposed by Boyer and Moore in 1977. The algorithm is considered as a benchmark in the industry. Although many times it has been modified and improved by different authors. The algorithm starts the matching process form the right end of the pattern and the pattern is shifted from left to right. For shifting the pattern the algorithm takes the help of good suffix and bad character heuristics. These heuristics allow the algorithm to skip many of the characters from matching attempt. The time and space complexity of preprocessing phase is *O* (m+|Σ|) and the worst case running time of searching phase is *O* (nm + |Σ|). The best case of the algorithm is *O* (n/m).

III. PROPOSED UNIQUE SOLUTION

The algorithm proposed in this paper consists of two phases:

1. Pre-processing of text (instead of pre-processing of pattern).
2. Matching of pattern in the given text.

A. Phase 1: Pre-processing of the text

This phase pre-processes the given text. For pre-processing the text, a two dimensional array arr[][] is taken. The array should have ∑ numbers of rows where ∑ is the size of the alphabet (number of identical characters in the text), no matter how many columns are there. Each row of the array (from 0^{th} row) is associated with exactly one character of the alphabet. For example: row 0 may be associated with character *a*, row 1 may be associated with character *b*, and so on. Now the whole given text is also considered to be in an array and consequently the text characters are assumed to be indexed starting from 0 for the first character. Now first character of the text is traversed and the index of the character is stored in the two dimensional array corresponding to that particular character. This process is repeated for every character present in the text. The outcome of this whole process is the two dimensional array that contains the index entry for every occurrence of every character of the text. Formation of this matrix ensures the gaining of all the knowledge about every character present in the text viz. the occurrence of the characters along with their places (indexes).

B. Phase 2: Matching of the pattern in the given text

In this phase, first of all, the first character of the pattern is checked. After getting the first character of the pattern, the index of that character's first occurrence is known from the two dimensional array by searching the index in corresponding row. Once the index is retrieved, the matching process is started from the right next index of the retrieved one. This process ensures skipping all the words those does not start from the first character of the pattern. For instance, if the alphabet size ∑ is let say 26, then all the words those start from at least 25 different characters will be skipped. During the matching process, if the mismatch occurs, the next occurrence of the pattern is retrieved from the two dimensional array and again the matching process is started at the newly retrieved index.

We understand the working of the algorithm with an example: we consider a text and the pattern with alphabet size ∑ being 27, i.e., ∑={'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x','y','z','_'}. The text and the pattern are as shown below.

Text all_systems_have_to_be_similar

Pattern sim

The given pattern *sim* has to be found in the given string *all_systems_have_to_be_similar*. The above

described procedure is used to find the pattern in the text.

A. Phase 1: Pre-processing of the text

In this phase the whole text is indexed starting from the first character by putting the text in an array and the index for all the characters of the text is put in a two dimensional array corresponding to the character itself.

a	l	l	_	s	y	s	t	e	m	s	_	h	a	v
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
e	_	t	o	_	b	e	_	s	i	m	i	l	a	r
15	16	17	18	19	20	21	22	23	24	25	26	27	28	29

Fig.1 The text after indexing

	0	1	2	3	4
a	0	13	28		
b	20				
c					
d					
e	8	15	21		
f					
g					
h	12				
i					
j	24	26			
k					
l	1	2	27		
m	9	25			
n					
o	18				
p					
q					
r	29				
s	4	6	10	23	
t	7	17			
u					
v					
w					
x					
y	5	14			
z					
__	3	11	16	19	22

Fig. 2 The two dimensional array containing the index of the characters of the text

The above figure shows the two dimensional array containing the index entry for every character present in the text.

B. Phase 2: Matching of the pattern in the given text

Now as the two dimensional array is prepared, the matching of the pattern can be started. For matching process, the first letter of the pattern *sim* is observed. This is *s*. Now the first occurrence of *s* can easily be found in the two dimensional array. In two dimensional array we can see that the first occurrence of *s* in the text is at position 4. So, the search can be started from the character indexed with 4.

Text all_systems_have_to_be_similar

Pattern sim

But, since the first character will match for sure, the matching process can be started from the right next character of the pattern with the corresponding character in the text.

Text all_systems_have_to_be_similar

Pattern sim

As it can be seen that a mismatch occurred at the second character, so again two dimensional array is looked up for getting the index of next occurrence of *s* and it is 6. So the pattern is kept under the index of 6 of the text and search is started from the character indexed with 7.

Text all_systems_have_to_be_similar

Pattern sim

Again a mismatch occurred at second character of the pattern and the next occurrence of *s* is at index 10. So the matching process is started from the character indexed 11.

Text all_systems_have_to_be_similar

Pattern sim

Again a mismatch occurred at second character of the pattern and the next occurrence of *s* is at index 23. So the matching process is started from the character indexed 24.

Text all_systems_have_to_be_**sim**ilar

Pattern **sim**

And finally the full match occurred and the pattern found at the position 23 in the text.

After going through this example we can easily see that algorithm takes least number of steps in matching process as compared to other algorithms. Our algorithm took only two shifts for matching the shifts successfully. The algorithm skips most of the words in the text while searching a pattern. The most beneficial thing is that if there is more than one occurrences of the pattern in the text then all of them can be found in a very short period of time.

IV. PROPOSED ALGORITHM

A. Phase 1: Pre-processing of the text

```
1.    n ← length (text)
2.    ∑ ← alphabet size
3.    for i ← 0 to n-1
4.          pos ← index of text [i] in array alphabet []
5.          j ← a [pos]
6.          arr [pos] [j] ← i
7.          a[pos] ← a[pos] + 1
8.    return arr[][]
9.    for i ← 0 to ∑ - 1
10.         do if (alphabet [i] = pattern [0])
11.               then nowset ← i
12.   end of loop
```

B. Phase 2: Matching of the pattern in the given text

```
1.    for i ← 0 to EleInRow - 1
2.          jump ← jump + 1
3.          count ← 1
4.          do for j ← 1 to m – 1
5.                do if (pattern [j] = Text [(arr[nowset][i]+j)])
```

6. then count ← count + 1
7. if (count = m)
8. then Print "pattern found at position" arr[nowset][i] "with" jump "number of jumps".
9. k ← 1
10. end inner for loop
11. else
12. end of the inner for loop
13. if (k ≠ 1)
14. then print "given pattern is not found"

V. TIME COMPLEXITY ANALYSIS

The time complexity for the above described two phases are as described below.

A. Phase 1: Time complexity of the first phase

This phase runs a *for* loop for *n* times (0 to n-1), (where *n* is the length of the text) that puts the index of each and every character of the text in a two dimensional array *arr*[][]. Once the *for* loop is executed completely we get a two dimensional array containing all the indexes of the characters present in the text. So the time complexity due to *for* loop is *O* (n).

This phase also contains the process of finding the character in the alphabet *alphabet*[][] that matches with the first character of the pattern *pattern*[0]. This forces us to traverse the only row in two dimensional array that contains the indexes of the first character of the pattern and that's the glory of our algorithm. The process of finding this association takes **at most** *alphabet_size* numbers of steps using a *for* loop. So, the complexity of this step is *O* (alphabet_size) or *O* D(∑). so, the whole phase results the complexity of *O* (n+∑) with the assumption that all other steps of the algorithm can be executed in constant time.

B. Phase 2: Time complexity of the second phase

The second phase of the algorithm is responsible for matching the pattern in the text with the help of two dimensional array *arr* [][]. This phase consists of two *for* loops: one outer and one inner. Both of two are responsible for extracting the index entries of the first character of the pattern one by one from the two dimensional array and matching the pattern's characters in the text with the help of the extracted index.

For extracting the index from the two dimensional array, the outer *for* loop runs for at most *EleInRow* times, where the *EleInRow* is the number of elements (index entries) in the row that keeps the index entries of the first characters of the pattern. For example, if the pattern is ***a****pproved* and the text contains 10 number of **a'**s, then there will be 10 index entries in **a**'s row of the two dimensional array and the value of *EleInRow* will be 10.

For matching the pattern in the given text, the inner *for* loop runs for *m-1* times (1 to m-1), for each time the outer *for* loop runs. All other assignments, condition checks and print functions can be assumed to be done in constant time and consequently can be ignored while calculating the complexity.

Thus the total time complexity for this phase is *O* (*EleInRow-(m-1))*, which is very low as compared to *O* (*mn*) (i.e. the complexity of many other algorithms).

VI. COMPARISON

The following table, table 1 gives a tabular comparison of the complexities of various algorithms with the proposed algorithm.

TABLE I
COMPARISOIN OF COMPLEXITITES OF MAJOR ALGORITHMS WITH THE NEWLY PROPOSED ONE

Algorithm	**Pre-processing**	**Matching Time**
Naïve String Search	No Preprocessing	Θ ((n-m+1)m)
Rabin- Karp algorithm	Θ (m)	Θ (n+m)
		Θ((n-m+1)m)
Knuth-Morris-Pratt algorithm	Θ (m)	Θ (n)
Boyer Moore Horspool algorithm	Θ (m+|∑|)	*O*(nm)

Jumping Algorithm	$O(n+\sum)$	$O((m-1)$ EleInRow)

VII. CONCLUSION

This paper presents a very new idea for finding a pattern in a given text by pre-processing the text. The pre-processing phase of Jumping Algorithm helps the matching phase in taking the jumps. The asymptotic analysis shows that the pre-processing phase that takes $O(n+\sum)$ time for putting the indexes in two dimensional arrays may be a little bit costly but the matching phase is amazingly cheaper in comparison of other existing algorithms. For small texts, the matching phase is almost constant. This less number of matching is what the key idea behind the algorithm. Since the algorithm shows its excellent behaviour in terms of complexity, it can be adopted in string matching's practical applications.

VIII. FUTURE WORK

Although the algorithm proves its effectiveness up to the mark in the matching phase, still the scope of improvement is always there. The pre-processing phase that takes $O(n+\sum)$ time can be improved somehow. Furthermore pre-processing phase needs a two dimensional array for storing the indexes of the characters of the text. The number of rows of this two dimensional array should be equal to the alphabet size $\sum$ and so, the number of rows can easily be decided, but it is difficult to say how many columns should be taken as we can't say that how many times a particular character may occur in the text. This problem can perhaps be reduced by taking some other data structures like linked list so that the number of required spaces for storing the indexes could be decided dynamically and ultimately the storage need could be reduced.

IX. REFERENCES

[1] Thomas H. Corman, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, *Introduction to Algorithms* (2nd Ed.), ISBN 81-203-2141-3, Prentice Hall of India, 2004

[2] Richard M Karp, Michael O. Rabin, "Efficient randomized pattern-matching algorithms", *IBM Journal on Research Development*, Vol. 31, No. 2, 1987, pp. 249-260

[3] Donald Knuth, Jr. H. Morris, Vaughan Pratt, "Fast pattern matching in strings," *SIAM Journal on Computing*, Vol. 6, No. 2, doi: 10.1137/0206024, 1977, pp.323–350

[4] R.S. Boyer, J.S. Moore, "A fast string searching algorithm", *Communication of the ACM*, Vol. 20, No. 10, 1977, pp.762–772