String Matching Algorithm based on Letters' Frequencies of Occurrence

Majed AbuSafiya
Faculty of Information Technology
Al-Ahliyya Amman University
Amman, Jordan
majedabusafiya@gmail.com

***Abstract*—In this paper, we utilize the varying frequencies of occurrence of letters in the words in natural languages to propose a new string matching algorithm. Unlike standard string matching algorithms that compares letters in the pattern against the text in fixed order, the proposed algorithm ranks the letters of the pattern according to their frequency of occurrence. The letters of patterns are then matched against text according to that ranking, starting with the letter with the least frequency of occurrence. The proposed algorithm significantly outperformed KMP algorithm in terms number of comparisons.**

Keywords—string matching; letter occurrence frequency, KMP

I. INTRODUCTION

String matching is a classical problem in computer science. It is widely used in fields like information retrieval, bio-informatics and intrusion detection systems. The inputs to the string matching algorithm are a string to match, called *pattern* (*P*) whose length is *m* and a text (*T*) to search in with length *n*. The output of a string matching algorithm is the locations in *T* where the pattern *P* exists.

It is known that letters of any natural language have a varying frequencies of occurrence in the words of this language. For example, in Arabic, a letter like 'ا' (named *Alef*), occurs much more frequently in Arabic text than, for example, the letter 'ظ'. In this paper, we propose Frequency of Occurrence Based string matching algorithm (FOB) that utilizes this variation. By comparing the letters of *P* against the current window on *T* in increasing order of their frequency of occurrence, we can faster reach to possibly matching window in *T* with less number of comparisons.

We have compared FOB with the well-known Knuth-Morris-Pratt (KMP) string matching algorithm [4] and found that the number of comparisons for FOB is significantly less than those for KMP algorithm.

This paper is organized as follows: section II introduces most known string matching algorithms. Section III presents the proposed FOB algorithm. Section IV presents the experimental study that was conducted to compare FOB with KMP. The paper ends with a conduction and list of references.

II. RELATED WORK

Literature is rich with many string matching algorithms. Brute Force Algorithm [1] is the naive string matching algorithm where the a window of the pattern length is shifted by one position in *T* for each iteration. Its time complexity is $O(m \times n)$. Other string matching algorithms does better than the Brute Force string matching algorithm by a preprocessing phase on *P* where a function(s) over *P* is found, stored in a data structure, and then used to faster-slide the comparison window over *T*. Boyer-Moore algorithm [3] generates two such functions in the preprocessing phase. It is time complexity is $O(m \times n)$. Knuth-Morris-Pratt (KMP) algorithm [4] has $O(m+n)$ complexity. In the preprocessing phase on *P*, the prefix function is created. The prefix function is used to avoid starting comparisons of *P* against *T* starting from the first letter in case of mismatch was found. Karp-Rabin algorithm [5] creates a hash function in the preprocessing phase to help in checking the similarity between *P* and the current window on *T*. Its complexity is $O(m \times n)$. Horspool algorithm [6] is a simplification of Boyer-Moore algorithm with search complexity of $O(m \times n)$. Quick search algorithm [7] is another simplification of Boyer-Moore with search complexity $O(m \times n)$. This algorithm is efficient in case of using short patterns and large Alphabets. Shift-Or algorithm [8] uses bit-wise techniques and is efficient if *P* is not longer than memory-word length of the machine. It is search complexity is $O(n)$. Raita Algorithm [9] compares the last letter of the pattern, the first, and then the middle letter. It has $O(m \times n)$ complexity. Berry-Ravendran Algorithm [10] performs windows shifts by considering the bad character shift for the two consecutive text characters to the right of the window. Its complexity is $O(m \times n)$.

After studying these string matching algorithms, we found that none of them have used the frequency of occurrence of language letters in speeding string matching. All these algorithms (except for Brute-force algorithm) need a preprocessing phase. They all have search complexity of $O(m \times n)$ except for KMP algorithm (with $O(m+n)$ complexity) and Shift-Or algorithm (with $O(n)$ complexity). However, Shift-OR algorithm is restricted to patterns that are short and less in length than the computer memory word length. KMP algorithm is a generic efficient algorithm that can work in any context. We have chosen KMP to evaluate the efficiency of FOB.

III. PROPOSED ALGORITHM

We will use the terminology and format used in [1] to present FOB algorithm. FOB requires statistics for frequency of occurrence of the Alphabet letters of the natural language of the text. We chose Arabic language for presenting the proposed

Authorized licensed use limited to: Birla Inst of Technology and Science Pilani-Dubai. Downloaded on February 25,2022 at 17:47:10 UTC from IEEE Xplore. Restrictions apply.

algorithm. However, the proposed approach can be applies to texts of any other language. We adopted the frequencies of Arabic letters given by the study in [3]. This reference calculated the frequencies of Arabic letters from several texts that contain together over five million letters. These frequencies may slightly vary if done with different set of texts. However, they give some notion of relative likelihood of a letter occurrence in Arabic text. We don't really care about the exact expected frequency of occurrence of letters. Given any two letters, we care to decide which letter is less likely to exist in text than the other.

```
FOB(T, P)
1  n = T.length
2  m = P.length
3  RANK(P)
4  first = GET-SEARCH-LETTER-WITH-RANK(0)
5  for t = 1 to n - m
6       if  T[t] == P[first].letter
7            SEARCH(0,t);
```

Fig. 1. OFB Algorithm

A listing of FOB algorithm can be found in (*Fig. 1)*. OFB algorithm first initializes variables *n* and *m* with the lengths of *T* and *P* respectively. Note that we maintain the information about *P* by defining *P* in the algorithm as an array of records. One record for each letter in the pattern. *P*[*i*] record contains the fields: (1) *letter*: that represents the letter at location *i* in the pattern, (2) *frequency*: that represents the frequency of occurrence of the *i*-th letter of the pattern, (3) *rank:* that represents the rank of the *i*-th letter in *P* among the other letters (in *P*) according to its frequency of occurrence and (4) *location:* which will be the value of *i.* Note that the indexing of *P* and *T* starts from 1 as found in [1].

OFB calls RANK(*P*) algorithm (*Fig. 2).* RANK(*P*) represents the preprocessing phase of the FOB algorithm. The task of RANK is to set the *rank* fields in *P* according to their frequency of occurrence. In lines 2-4 in RANK(*P*), the *rank* field is initialized to zero. Also, the *frequency* field of every letter in *P* is set with its frequency of occurrence. This is done through GET_FREQ_OF_OCCURANCE that will get the frequency of the letter from a file that contains the frequency of occurrence values for all letters of the language used.

```
RANK(P)
1    m = P.length
2    for i = 1 to m
3        P[i].rank = 0
4        P[i].frequency
             = GET_FREQ_OF_OCCURANCE(P[i].letter)
5    for i = 1 to m
6        lessThanCounter = 0
7        for j = 1 to i
8            if P[i].frequency> P[j]. frequency
9                 lessThanCounter = lessThanCounter +1
10           else
11                P[j].rank= P[j].rank + 1
12           P[i].rank=lessThanCounter
```

Fig. 2. RANK Algorithm

For example, let the pattern be *P*="مالك", at the end of RANK execution, the fields of *P* will be as shown in *TABLE I.* Note that RANK(*P*) is called only once for the pattern. Note that if *P* contains a letter that has no frequency of occurrence value in the file, it will get the rank zero.

TABLE I. P[I] FIELDS

i	P fields			
	P[i].letter	*P[i]. frequency*	*P[i].location*	*P[i].rank*
1	م	8.08%	1	1
2	ا	13.7%	2	3
3	ل	11.55%	3	2
4	ك	3.17%	4	0

Returning to OFB algorithm (*Fig. 1*), and after calling RANK, GET-SEARCH-LETTER-WITH-RANK will be called with argument 0 to get the index of the letter in *P* that has the lowest rank. That is, the variable *first* will contain the index of the letter of the least frequency of occurrence among the letters in *P*. In our example with *P*="مالك", the letter of least frequency is the letter 'ك', so GET-SEARCH-LETTER-WITH-RANK will return the value 4.

In line 5, OFB runs a loop searching for an occurrence of *P*[*first*].*letter* in *T*. Once found, aa call to SEARCH(*0*, *t*) is made. Refer to *Fig. 3*.

```
SEARCH(rank, t){
1   current = GET-SEARCH-LETTER-WITH-RANK(rank)
2   if  rank == P.length
3    if  T[t]==P[current]
4        print "Pattern occurs at shift"  t - P[current].location
5        rank=0
6   else
7    if  P[current] == T[t]
8          rank = rank + 1
9          next=GET-SEARCH-LETTER-WITH-RANK(rank)
10         t = t - (P[current].location – P[next].location)
11        SEARCH(rank, t)
```

Fig. 3. SEARCH Algorithm

SEARCH is a recursive algorithm. It takes two arguments. The first argument (i.e. *rank*) is the current rank that is being searched for its letter. The second argument (i.e. *t*) is the start search index in *T*.

The recursion stopping condition for SEARCH happens when *rank* value reaches to the length of the *P*. If this happens, this means that *m-1* letters of the pattern are already matched except the last letter, which is the letter with the highest rank in *P*. We mean by last here: last by rank and not last by position in *P*. Now we make a final check (in line 3) to decide if the last letter is matched. If so, current position of *P* in *T* is printed.

IV. COMPARISON WITH KMP

We compared FOB with KMP algorithm. We implemented KMP algorithm, as described in [1]. The comparison is based on comparing the number of if-statement comparisons that are

Authorized licensed use limited to: Birla Inst of Technology and Science Pilani Dubai. Downloaded on February 25,2022 at 17:47:10 UTC from IEEE Xplore. Restrictions apply.

run by each of the two algorithms. This was done by injecting both algorithms with a counter (*numberOfComparisons*) that is initialized by zero and incremented wherever an if-statement comparison is executed. We have chosen *T* to be the text of the Holy Quran which contains 77,439 of words. The experiment is based on randomly selecting a word from *T* and then searching for it using both algorithms. We repeated this process 3,000 times adding up the number of comparisons. By random selection of the words, we eliminated the bias in the results for the word length or repetition. So it gives a generic measure for the number of comparisons without predefined assumptions of the selected matched words. The results can be seen in TABLE II. Note that FOB has significantly less number of comparisons than those of KMP.

TABLE II. NUMBER OF COMPARISONS FOR 3,000 PATTERNS EXISTING IN *T*

	KMP	***FOB***	***Improvement***
Number of Comparisons	2,581,533,934	1,558,591,453	39.6%

The second experiment is conducted to compare the performance between the two algorithms in case *P* does not exist in *T*. This experiment is the same as the first experiment except that after randomly selecting *P* from *T*, we randomly chose one letter from *P* and replace it with a letter that *does not* exist in *T*. The results are shown in TABLE III. OFB showed even higher performance in case of *P* does not exist in *T*.

TABLE III. NUMBER OF COMPARISONS FOR 3,000 PATTERNS THAT DOES NOT EXIST IN *T*

	KMP	***FOB***	***Improvement***
Number of Comparisons	2,553,122,670	1,233,277,061	51.7%

The third experiment was conducted to see the effect of the pattern length. We ran KMP and FOB algorithm with a set of randomly selected 1,000 words from *T,* each of length 3. We repeated the experiment for a randomly selected 1,000 words of length 4, up to 10. OBF clearly outperformed KMP (*Fig. 4*).

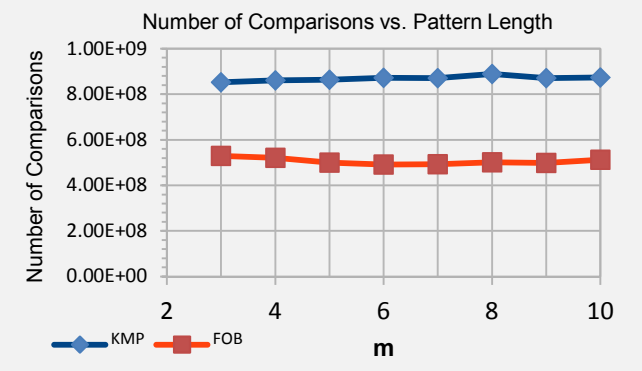

Fig. 4. Number of Comparisons vs. pattern length

Definitely there are many advantages of KMP over FOB. KMP is a generic string matching algorithm. It does not need any extra information other than *P* and *T* while FOB needs to know the frequencies of occurrence of the letters to work. However, overhead of finding the frequencies of occurrence is needed only once. KMP requires a preprocessing on *P* to compute the prefix function. Also, FOB requires a preprocessing of *P* to find the ranks of letters in *P*.

V. CONCLUSION

In this paper, we proposed a string matching algorithm that utilizes the variation of letter frequency of occurrence of letters in natural languages. Using frequency of occurrence statistics helped to improve the speed of string matching as shown. By starting to compare the letters of *P* of least frequency of occurrence, we can reject the mismatching window in the search text faster. We suggest as future work, working on analytical analysis of the proposed algorithm. The significant reduction in the number of comparisons, deserves a detailed study of complexity of the algorithm. A great benefit is expected if we consider the likelihood of relative positioning of a given letter with another specific letter in Text.

REFERENCES

[1] T. Cormen, C. Leiserson, R. Rivest, C. Stein, Introduction to Algorithms, 3rd ed., MIT Press, 2009.

[2] Arabic Letter Frequency, https://en.wikipedia.org/wiki/Arabic_letter_frequency

[3] R. Boyer, J. Moore, A fast string searching algorithm. Commun. ACM, Vol. 20(10), pp. 762-772, 1977.

[4] D. Knuth, J. Morris, V. Pratt, Fast Pattern Matching In Strings, SIAM journal on computing, Vol. 6(2), pp. 323-350, 1977.

[5] R. Karp, M. Rabin, Efficient randomized pattern matching algorithms. IBM journal of Research and Development, Vol. 31(2), pp. 249-260, 1987.

[6] R. Horspool, Practical fast searching in strings, Software: Practice and Experience, Vol. 10(6), pp. 501-506, 1980.

[7] D. Sunday, A very fast substring search algorithm. Commun. ACM, Vol. 33(8), pp. 132-142, 1990.

[8] R. Baeza-Yates, G. Gonnet, A new Approach to text searching. Commun. ACM, Vol. 35(10), pp. 74-82, 1992.

[9] T. Ratia, Tuning the Boyer-Moore-Horspool string searching algorithm. Software: Practice and Experience, Vol. 22(10), pp. 879-884, 1992.

[10] T. Berry, S. Ravindran, A fast string matching algorithm and experimental results, Stringlology, pp.16-28, 1999.

Authorized licensed use limited to: Birla Inst of Technology and Science Pilani-Dubai. Downloaded on February 25,2022 at 17:47:10 UTC from IEEE Xplore. Restrictions apply.