# Word Matching Algorithm based on Relative Positioning of Letters

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

***Abstract*—In this paper, we propose a word matching algorithm to find the valid shifts of a given word in a text string. The idea is based on using pre-calculated information about the likelihood of relative positioning of each pair of letters of the search word in the words of the language under consideration. This information helps in deciding an order in which the letters of the search word should be compared with the letters of the words in the text. This ordering eliminates mis-matching words in text in less number of comparisons than those needed in standard ordering of comparisons. Our algorithm showed better performance than KMP algorithm.**

***Keywords*—algorithm, string matching; letter position**

## I. Introduction

String matching is a classical problem in computer science. 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 valid shifts of *P* in *T*. In this paper, we will consider a special case of this problem where the pattern is a single word and the matching happens between this word and the words of *T*. We will name this problem *word matching*.

It is known that letters have a varying likelihood to exist in a given position in the words of any natural language. Also, a given pair of letters have varying likelihood to co-exist in certain positions in the words of the language. For example the likelihood of finding the pair of letters (ال) in the first and second positions of an Arabic word is much likely to occur than, for example, finding (ا) at position 2 and ل at position 6. By studying a large number of words of the language under consideration, we can calculate values that quantify the likelihood for a given pair of letters to co-exist in certain positions in the words of this language. This information can be used to decide an order in which the letters of the search word should be compared with the letters of the words in *T*. By ordering these comparisons based on the least likely to occur pair, mis-matching words of *T* can be eliminated in less number of comparisons.

This paper is organized as follows: section II introduces most known string matching algorithms. Section III presents the proposed algorithm. Section IV presents the experimental study that was conducted to compare the proposed algorithm with the well-known KMP algorithm. The paper ends with a conclusion 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 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 $O(n)$. Ratia 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 $O(m\times n)$.

Considering these string matching algorithms, we found that none of them used the likelihood of occurrence of the letter pairs in the word as basis to order the comparisons between the letters of the pattern and the the letters in the text. The work in [2] used the frequencies of letter occurrences in the words of the language to order the comparisons. However, it did not consider the location of the letter in the word.

## III. PROPOSED ALGORITHM

We will use the terminology and format used in [1] to present our proposed algorithm. We considered the Arabic language for our study. However, the idea can be applied to any natural language. The main algorithm is shown in Fig-1. WORD-MATCHING takes two parameters: the *word* to search for and the text string to search in *T*. The algorithm will *tokenize T* by taking the words of *T* one at a time. We will denote the current word of *T* being matched with *nextToken.* The algorithm then calls MATCH(*word*, *nextToken*) algorithm. If a match is found between *word* and *nextToken*, then a valid shift of *word* in *T* is found and the position of *nextToken* in *T* should be printed.

```
WORD-MATCHING(word, T)
1 FrequencyVector= BUILD-FREQUENCY-VECTOR(word)
2 while (T.hasMoreTokens())
3       nextToken= T.getNextToken()
4       if MATCH(word, nextToken)
5             print 'valid shift at ' nextToken.position()
```

Fig. 1. WORD-MATCHING Algorithm

The first step in the WORD-MATCHING algorithm is to build the frequency vector for the pairs of letters of the search word. This step corresponds to the preprocessing phase that is usually done by the known string matching algorithms. This happens by calling BUILD-FREQUENCY-VECTOR(*word*) algorithm (Fig-2).

```
BUILD-FREQUENCY-VECTOR(word)
1 for i=0 to word.length()-1
2       for j=i+1 to word.length()
3             first= word.charAt(i);
4             second=word.charAt(j);
5             freqRecord= GET-RECORD(first, second, i, j-i)
6             freqVector.add(freqRecord)
7 return freqVector
```

Fig. 2. BUILD-FREQUENCY-VECTOR Algorithm

The BUILD-FREQUENCY-VECTOR algorithm builds the frequency vector that is needed to determine the order in which the letters of *word* should be compared with the letters of *nextToken*. To show how the BUID-FREQUENCY-VECTOR works, we will explain it through an example. Let the word to be searched for *word*="الرحمن" . Table-I shows the frequency vector for this word. This table contains a row for each pair *(x,y)* of letters of *word,* where *x* represents the first, and *y* represents the second. The letter *x* always proceeds *y* in *word. xLocation* represents the location of *x* in *word.* Note that the position of the first letter of *word* is 0. *distance* represents the distance of *y* from *x* in *word.* For example, refer to the first row in Table-I, *x*=(ا), *y*=(ل), *xLocation* is the location of *x* in the word "الرحمن" which is 0, *y* location is 1 which is found by adding *xLocation* and *distance*. In general, the *Freq* column contains a scalar value that represents the likelihood that the letter *x* will exist at *xLocation* in an Arabic word and the letter *y* to co-exist in the same word with *distance* letters away from *x*. *Freq* values are calculated by studying a large number of Arabic words. We have stored these values in a normal text file. BUILD-FREQUENCY-VECTOR(*word*) builds the frequency vector for *word* by querying the frequency file for the frequencies for all possible values of *x*, *y*, *xLocation*, *distance* for *word.* This is done by calling the procedure GET-RECORD (Fig-2). The output will be the frequency vector shown in Table-I. The *Row* and *Example* columns are added for clarification.

TABLE I. FREQUENCY VECTOR FOR WORD="الرحمن"

| Row | Example | | | | | | *x* | *xLocation* | *y* | *distance* | *Freq* |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | ن | م | ح | ر | ل | ا | | | | | |
| 1 | - | - | - | - | ل | ا | ا | 0 | ل | 1 | 1203 |
| 2 | - | - | - | ر | - | ا | ا | 0 | ر | 2 | 66 |
| 3 | - | - | ح | - | - | ا | ا | 0 | ح | 3 | 43 |
| 4 | - | م | - | - | - | ا | ا | 0 | م | 4 | 58 |
| 5 | ن | - | - | - | - | ا | ا | 0 | ن | 5 | 66 |
| 6 | - | - | - | ر | ل | -- | ل | 1 | ر | 1 | 55 |
| 7 | - | - | ح | | ل | - | ل | 1 | ح | 2 | 46 |
| 8 | - | م | - | - | ل | - | ل | 1 | م | 3 | 124 |
| 9 | ن | - | - | - | ل | - | ل | 1 | ن | 4 | 90 |
| 10 | - | - | ح | ر | - | - | ر | 2 | ح | 1 | 28 |
| 11 | - | م | - | ر | - | - | ر | 2 | م | 2 | 65 |
| 12 | ن | - | - | ر | - | - | ر | 2 | ن | 3 | 98 |
| 13 | - | م | ح | - | - | - | ح | 3 | م | 1 | 17 |
| 14 | ن | - | ح | - | - | - | ح | 3 | ن | 2 | 44 |
| 15 | ن | م | - | - | - | - | م | 4 | ن | 1 | 25 |

The MATCH algorithm is shown in Fig-3. It returns **true** if *word* matches *nextToken*. It returns **false** immediately if *word* and *nextToken* are of different length (lines 1-2). We describe a letter from *word* to be *finished* in case it already was compared with the corresponding letter in *nextToken*. In the worst case, the **while** loop will iterate a number of times equals to the length of the word. It may iterate less number of times. The first line in the **while** loop (line 5), the next record of frequency vector with minimum frequency is retrieved. For our example, for *word* = "الرحمن", the first record will be taken is row 13 in

Table-I. This means that the first letters of *word* to be compared with the corresponding letters in *nextToken* are the letters *x*=(ح) which exists in *xLocation*=3 and *y*= (م) which exists in location 4. The variable *firstPos* will take the value 3 while *secondPos* will take the value 4 (lines 6-7). The letter at *firstPos* of *nextToken* will be compared with the letter at *firstPos* in *word.* Also the letter at *secondPos* of *nextToken* will be compared with the letter at the *secondPos* in *word* (line 8). If either of these comparisons returns false, the MATCH algorithm will terminate and return false meaning the *nextToken* will not match *word* (line 9). If execution reaches line 10, this means that both *x* and *y* of *word* were matched with corresponding letters in *nextToken*, so we mark them as *finished* (lines 10-15). The while loop then iterates to get the next minimum frequency record from the frequency vector. The next minimum frequency will be for the pair *x*=(م) and *y*=(ن) (row 15 in Table-I). The while loop will iterate until all the letters of *word* are *finished* or a mis-match is found.

```
MATCH(word, nextToken)
1  if (nextToken.length() != word.length())
2      return false
3  finishedCounter=0
4  while (finishedCounter != word.length())
5      minFreqRecord = word.freqVector.getNextMin()
6      firstPos = minFreqRecord.xLoc
7      secondPos = minFreqRecord.xLoc + minFreqRecord.distance
8      if (nextToken.charAt(firstPos) != word.charAt(firstPos)
           or
          nextToken.charAt(secondPos) != word.charAt(secondPos))
9          return false;
10     if (word.charAt(firstPos).finished=false)
11         word.charAt(firstPos).finished=true
12         finishedCounter= finishedCounter+1
13     if (word.charAt(secondPos).finished=false)
14         word.charAt(secondPos).finished=true
15         finishedCounter= finishedCounter+1
16 return true
```

Fig. 3. MATCH algorithm

The order in which the letters of *word*=”الرحمن” are matched against *nextToken* is shown in Table-II. Note that we chosen *nextToken* to be the same as *word* to consider the worst-case for the algorithm (i.e. a match is found).

TABLE II. SUMMARY OF MATCH ITERATIONS FOR WORD=”الرحمن” AND NEXTTOKEN=”الرحمن”

| *nextToken* | ن | م | ح | ر | ل | ا | Finished |
|---|---|---|---|---|---|---|---|
| *word* | ن | م | ح | ر | ل | ا | |
| First Iteration | | م | ح | | | | ح ، م |
| Second Iteration | ن | م | | | | | ح ، م، ن |
| Third Iteration | | | ح | ر | | | ح ، م، ن ، ر |
| Fourth Iteration | | | ح | | | ا | ح ، م ، ن ، ر ، ا |
| Fifth Iteration | | | ح | | ل | | ح ، م ، ن ، ر ، ا ، ل |

## IV. COMPARISON WITH KMP

To evaluate the performance of our algorithm, we compared it with the known KMP algorithm. The comparison was based on the total time needed to match a given word against the prefixes (of a long text string which is the text of the Holy Quran) with growing number of letters. We had to make a small modification on the KMP so that it matches a search word against single words of the text string so that we have a fair comparison with our algorithm (WMA). Fig-4 shows the comparison results. The x-axis shows the sizes of prefixes of the text string that were used to conduct the experiments. The y-axis shows the time needed to find the valid shifts for both algorithms for *word*=“الرحمن” in nanoseconds in these prefixes. This time includes both: the preprocessing time needed to build the frequency vector and the matching time needed to match *word* in the prefixes of text string. Note that the per-processing is needed once in WMA for every experiment. In every experiment, both algorithms were run repeatedly on the prefixes (of varying length) of the same text string. We started with the prefix of length of 1000 letters for a text string, then with the prefix of length 2000 letters for the same text string and so on. This is repeated up to the prefix of size 400,000 letters for the same text string.

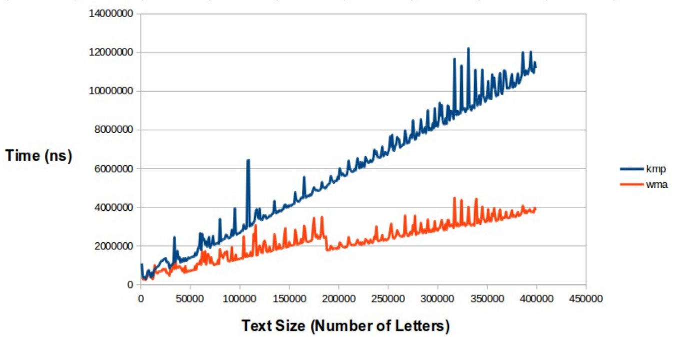

Fig. 4. Comparison with KMP

We studied the preprocessing time for both algorithms for *word* =“الرحمن” and we found that the preprocessing time needed for our algorithm was much longer (6730 *ms*) than the KMP preprocessing time (201 *ms).* However, the out-performance of WMA algorithm is due to the shorter matching time. As shown in the figure, the out-performance of WMA is more obvious for larger text size.

## V. CONCLUSION

In this paper, we proposed a word matching algorithm where a search word is matched against the words of a search text. The proposed algorithm reduces the number of comparisons needed to eliminate mismatching words by comparing pairs of letters of the search word in a different ordering than the ordering used in the known string matching algorithms. This ordering is determined by exploiting the knowledge about the likelihood of co-existence of a pair of letters in certain positions in the words of the language under consideration. We compare pairs of letters of the search word against the corresponding letters in text words, starting with those that are least likely to co-exist in the words of the language in the corresponding positions. As future work, we look forward extending the idea of the proposed algorithm to search for arbitrary strings and not only for single words.

## REFERENCES

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

[2] M. AbuSafiya, String Matching Algorithm based on Letters' Frequencies of Occurrence. 8Th International Conference on Computer Science and Information Technology, (CSIT), 2018.

[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] 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, Stringlogy, pp.16-28, 1999.