DOI: https://doi.org/10.26682/sjuod.2020.23.1.14

Journal of University of Duhok, Vol. 23, No.1 (Pure and Eng. Sciences), Pp 134-143, 2020

EMPIRICAL PERFORMANCE EVALUATION OF KNUTH MORRIS PRATT AND BOYER MOORE STRING MATCHING ALGORITHMS

SARHAN S. DAWOOD and SAMAN A. BARAKAT

*Dept. of Computer Science, College of Science, University of Duhok. Kurdistan Region-Iraq

(Received: September 22, 2019; Accepted for Publication: May 19, 2020)

ABSTRACT

Many algorithms have been proposed for string matching in order to find a specific pattern in a given text. These algorithms have been used in many applications such as software editors, genetics, Internet search engines, natural language processing, etc. The aim of this paper is to evaluate the performance of two popular algorithms: Boyer Moore (BM) and Knuth Morris Pratt (KMP) in terms of execution time. The algorithms have been programmed using Java and Java Microbenchmark Harness to evaluate their execution time using a number of experimental test scenarios. Results show that the BM algorithm outperformed the KMP algorithm in all test scenarios.

KEYWORDS: Boyer Moore, Knuth Morris Pratt, String Matching, Performance Evaluation

1. INTRODUCTION

The subject of exact text matching (searching in texts) is one of the important topics in the field of modern text analysis or text searching. The main objective is to find all occurrences of a pattern (P) with size (M) in a text (T) of size (N), where (M<N) and P, T ∈ Σ, where Σ is a set of finite elements (symbols) taken from alphabet of a give language (called alphabet). Many solutions to this problem have been proposed, most of which are conveniently applicable only in very specific cases (Catania, 2018)(Charras & Lecroq, 2004). Nowadays, string matching algorithms which are widely used in most text editors, genetics, internet search engines, natural language processing, etc. Since the exact text matching algorithms are used instead of binary and linear algorithms, many algorithms where proposed, some of them use new techniques in searching and other refinement for pre-proposed algorithms. Most of these algorithms, perform the search in one of two ways: either from right to left, or from left to right, keeping in mind that a little number of algorithms may use some other specific order in searching or in any order (Charras & Lecroq, 2004). In this paper we will evaluate the performance of two widely used algorithms, Boyer Moore (BM) algorithm and Knuth Morris Pratt (KMP) algorithm in terms of time required to execute each algorithm~~,~~ using number of experimental test scenarios. Java Microbenchmark Harness (JMH) (Oracle, 2019) has been used to measure the performance of both algorithms. JMH is a Java toolkit that provides a very solid foundation for writing and running benchmarks. It is flexible, easy to use, and takes care of the JVM warm-up and code-optimization paths which lead to produce accurate results.

The paper is organized as follows: Section 2 presents the related work. The BM algorithm is highlighted in section 3 and the KMP Algorithm is provided in section 4. The test methodology is presented in section 5 and results and discussions are shown in section 6 and 7. Finally, section 8 is devoted to conclude the paper.

2. RELATED WORK

Since the first string matching algorithm was proposed, they become a target for interested researchers in this important area. Some of these researchers proposed new algorithms, while others suggested certain modifications on the work of existent algorithms. Some of these works will be reviewed to highlight these attempts in evaluating the algorithms referred to in this paper.

- (De V. Smit, 1982) examines and compares time complexities of three strings matching algorithms, straightforward, KMP and BM. The comparison is made based on experimental data using their actual average behaviour. It is shown that the BM algorithm is much better in most cases, and on average the KMP algorithm is not remarkably better performance than the straightforward algorithm.

saman.barakat@uod.ac

- (Sunday, 1990) illustrates a substring search algorithm that has better performance than the BM algorithm and it is not based on scanning the pattern in any specific order. He used three various pattern scan Techniques: a Quick Search, a Maximal Shift, and Optimal Mismatch algorithm. He showed that, for short pattern strings this algorithm has about 20% or greater increase in search speed for normal English text. No complexity results are proven.
- (Ersin, Carus, & Mesut, 2007) tested eight different texts of the same size from eight different natural languages through six different matching algorithms and recorded search times. The aim—is to show that the variation in performance of these algorithms does not only depend on the number of symbols in alphabets, but also depends on the constitutional changes of these natural languages. Their work shows also that the performance of some languages is not efficient in long and medium length strings while they are better in short length strings or vice versa.
- (Wahlström, 2013) evaluates five string searching algorithms; Brute Force, BM, KMP, Karp-Rabin and the Horspool algorithm, through presenting how they work, when they work and when each one of them will be the best choice for a particular problem. He shows also that the choice of algorithm depends on: The length of the pattern, type of the alphabet, length of the searched text, and pattern length.
- (Chandraseta, 2017) analyzed the difference between BM algorithm and its derived variants and presented an optimization toward the BMHS algorithm. He found that the improvements algorithms are pretty good optimizations that also have their own worst case that performs worse than BM. The best optimization is achieved by combining BM good suffix rule with some other improvement algorithms (BMHS0, BMHS).
- (Tsarev et al., 2016) presents a composed algorithm, which has been created based on of KMP and BM string matching algorithms. They show that combining these two algorithms, allows earning larger shift in case of pattern and string characters' mismatch.

Although previous works revolves around the same field of research of this paper. However, this paper differs from previous works in two ways: (i) using different text and (ii) using different pattern sizes or text language. For example, Authors in (De V. Smit, 1982) evaluated and compared the average performance of three algorithms (straightforward, KMP and BM) using a fixed and small length African language text (500 characters) with short patterns no longer than (14) characters. While authors in (Sunday, 1990) describes an improved substring search algorithm that has better performance than the other algorithm including BM and KMP algorithms, by using various pattern scan (quick, maximal shift, and optimal mismatch) algorithms on just three various patterns. Therefore, the new addition of this paper is to compare empirically between BM and KMP algorithms with regard to their execution time.

3. BOYER MOORE (BM) ALGORITHM

The Boyer Moore (BM) exact string matching algorithm was proposed by (Boyer & Moore, 1977). It is considered as one of the most efficient algorithms for strings searching applications such as text editors when searching and replacing, search engines, plagiarism detection (Rahim, Zulkarnain, & Jaya, 2017). The algorithm starts examining characters in the pattern from right to left, if the character being inspected in the pattern does not match the corresponding character in the text, the BM algorithm uses two predetermined tables prior processing to determine how the pattern must be skipped forward (Choudhary, Rasool, & Khare, 2012). These tables are constructed by using the "**bad character shift rule**" and "**good suffix shift rule**" (Gurung, Chakraborty, & Sharma, 2016) (Wahlström, 2013) (AbdulRazzaq, Rashid, Hasan, & Abu-Hashem, 2013). The time complexity for pre-processing of a text with length **n** and pattern with length **m** is O(m + size of alphabet). The best case performance time complexity is O(n/m), and worst case time complexity is O(nm).

3.1. Bad character shift rule

The bad character shift rule operates as follows:

For each character (X) in the alphabet, let L(X) represents the location of the right-most occurrence of character (X) in the pattern P, where L(X) is assigned to (0) if (X) is not found in the pattern. If the first mismatch is found at index (i) in the pattern and the corresponding text character T(j) mismatches the pattern character P(i), then shift the pattern right by **Max(1, i - L(T(j)).**

Example:

Let Text (T) = ACTGACTAACTCA and Pattern (P) = ACTCA

j	1	2	3	4	5	6	7	8	9	10	11	12	13
T	A	C	T	G	A	C	T	A	A	C	T	C	A
P	A	C	T	C	A								
i	1	2	3	4	5								

The L(X) table for the alphabet (right-most position of character X in the pattern)

X	A	C	G	T
L(X)	5	4	0	3

L(A) = 5, L(C) = 4, L(G) = 0, and L(T) = 3
Starting from the right of the pattern (P), i=j=5 and T(j)=A matched with P(i)=A, **then we skip the text and the pattern 1 position to the left.**
When i = J = 4 and T(4) = G Mismatched With P(4) , then shift the pattern right by **max(1, 4 - L(G)) which is = max(1,4 -0)=4**

j						1	2	3	4	5	6	7	8	9
T	A	C	T	G	A	C	T	A	A	C	T	C	A
P					A	C	T	C	A				
i					1	2	3	4	5				

And so on until all characters of the patter matches the corresponding characters of the text.

3.2. Good suffix shift rule

The good suffix shift rule is guided by finding the longest suffix characters X (X>0) of the pattern (P) that matches the corresponding characters of the text (T), which is denoted by suff(X) and is represented by the suffix of size (X) of the pattern. If suff(X) is not occurred in the pattern then it is shifted by the suffix size. However, if there is a prefix of size (l<X) within the pattern that matches the suffix of the same size, then the pattern is shifted by the distance between the prefix and the suffix. On the other hand, if suff(X) occurred in the pattern and not proceeded by the same character that resulted the mismatch then the pattern is shifted by a distance equal to suff(X) and its rightmost occurrence (Gurung et al., 2016).

Example

Text (T) = BIZFIZIBIZFIZBIZ
Pattern (P) = FIZBIZ

Good suffix shift table for (P) = FIZBIZ:

IZBIZ	ZBIZ	BIZ	IZ	Z	
6	6	6	3	6	1

Find the longest suffix that matches
- if that suffix appears to the left in **P** preceded by a different char, shift to align
- if not, then shift the entire length of the word

B	I	Z	F	I	Z	I	B	I	Z	F	I	Z	B	I	Z	
F	I	Z	B	I	Z											**IZ** suffix matches, **IZ** appears to the left, so shift by 3 to align (good-Suffix-Table (**IZ**) = **3**)
B	I	Z	F	I	Z	I	B	I	Z	F	I	Z	B	I	Z	
			F	I	Z	B	I	Z								no suffix match, so shift **1** spot (good-Suffix-Table () = **1**)
B	I	Z	F	I	Z	I	B	I	Z	F	I	Z	B	I	Z	
				F	I	Z	B	I	Z							**BIZ** suffix matches, doesn't appear again so full shift (good-Suffix-Table (**BIZ**) = **6**)
B	I	Z	F	I	Z	I	B	I	Z	F	I	Z	B	I	Z	
										F	I	Z	B	I	Z	🡐 Found

Last function:
Input: Text (**T**) with size **(n)** characters and Pattern (**P**) with size **(m)** characters
Output: Index of the first substring of (**T**) matching (**P)**

The **Last(X)** function takes a letter **(X)** from the alphabet and determines how much the pattern is to be shifted if a letter equal to (**X)** is found in the text that does not match the pattern.

$$\textbf{last(X)} = \begin{cases} \textbf{index of the last occurrence of (X) in pattern (P)} & \textbf{if (X) is in (P)} \\ \textbf{-1} & \textbf{otherwise} \end{cases}$$

Boyer Moore (BM) Algorithm

```
1:   Compute function last( )
2:   i ← m - 1
3:   j ← m - 1
4:   repeat
5:   if ( P[j] = T[i] )
6:         if (j = 0)
7:             return i  "A match"
8:         else
9:             i ← i - 1
10:            j ← j - 1
11:   else
12:      i ← i + m - min(j, 1 + last(T[i])
13:      j ← m - 1
14:  until( i  >  n  - 1 )
15:  return "No Match"
```

4. KNUTH MORRIS PRATT (KMP) ALGORITHM

The Knuth Morris Pratt (KMP) algorithm was introduced by (Knuth, Morris, Jr., & Pratt, 1977). It was the first linear-time algorithm developed for the exact pattern matching problem. It compares the pattern characters one by one with text from left-to-right (Rahim et al., 2017). This algorithm is based on the idea that: Every character in Pattern (P) is compared with characters in Text (T), if all characters in (**P)** are matching with characters in (**T**) then the search is successful, if any mismatch found then, shift the **(P)** according to the longest proper prefix that is also suffix (henceforth LPS) table which is prepared before the comparison process started between the pattern and text (Tsarev et al., 2016). The time complexity for pre-processing of the pattern with length m can be done in *O*(m), and the worst case time complexity for searching phase is *O*(n + m).

The LPS table is in fact a one-dimensional array, with a number of elements equal to the number of characters in the pattern, each element specifies "How many positions the pattern has to shift" when we find a mismatch. The first element of LPS table is always **0** (i.e. LPS (0) =0), the other elements will be calculated as follows, taking into account the repetition of these steps until the **LPS** table is filled.

LPS Table (Prefix Table)
Input: Pattern with size (m) characters
Output: Failure function LPS for P (i to j)

1: Define a one dimensional array with the size equal to the length of the pattern. (LPS[m])
2: Define variables **i & j**. Set i = 0, j = 1 and LPS[0] = 0.
3: Compare the characters at **P[i]** and **P[j].**
4: If they match then set **LPS[j] = i+1** and increment i & j values by one. And Go to Step 3.
5: If they are mismatched then: Check the value of 'i'. If it is 0 then set **LPS[j] = 0** and increment 'j' value by 1, if it is not '0' then set **i = LPS[i-1]**. Go to Step 3.
6: Repeat above steps until all the values of LPS[] are filled

Note that the failure function F is used to create the LPS table for P, maps j to the length of the longest prefix of P that is a suffix of P[1 . . j], encodes repeated substrings inside the pattern itself.

Example :

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
Text(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
	0	1	2	3	4	5	6															
Pattern (P)	X	Y	Z	W	X	Y	W				A											

n = size of Text = 22
m = size of Pattern =7

First, create the LPS table for the previous pattern according LPS algorithm:

0	1	2	3	4	5	6
0	0	0	0	1	2	0

Second, start implementing the KMP algorithm

Start comparing P[0] with T[0] starting from left to right

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
	0	1	2	3	4	5	6															
(P)	X	Y	Z	W	X	Y	W	A														

Up to P[2] the characters matched. At P[3] mismatch occurred , LPS[2] value is considered, since the value is 0 we need to compare the first character in P with next character in T

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
					0	1	2	3	4	5	6											
(P)					X	Y	Z	W	X	Y	W											

At P[6] mismatch occurs , so we will consider LPS[5] value, since the value is '2' we need to compare P[2] character with mismatched character in T.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
									0	1	2	3	4	5	6							
(P)									X	Y	Z	W	X	Y	W							

At P[2], a mismatch occurs, so LPS[1] value is considered. Since its value is 0, the first character P[0] in P is compared with the next character in T.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
												0	1	2	3	4	5	6				
(P)												X	Y	Z	W	X	Y	W				

At P[6] mismatch occurs, so we consider LPS[5] value. Since its value is “2”, we need to compare the P[2] character in P with mismatched character in T

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
(T)	X	Y	Z	A	X	Y	Z	W	X	Y	A	X	Y	Z	W	X	Y	Z	W	X	Y	W
																0	1	2	3	4	5	6
(P)	**Matched (the Pattern found) ➔**															X	Y	Z	W	X	Y	W

Here all P characters matched with a substring in T starting from index value 15

Knuth Morris Pratt (KMP) Algorithm:

```
1:   f ← the element in LPS table of pattern P
2:   i ← 0
3:   j ← 0
4:   While ( i < length[T] )
5:       if (P[j] = T[i])
6:           if (j = m-1)
7:               return i - m + 1 "A match"
8:           i ← i + 1
9:           j ← j + 1
10:      else if ( j > 0 )
11:          j ← f(j -1)
12:      else
13:          i ← i + 1
14:  return "No match"
```

5. METHODOLOGY

This paper uses similar methodology to (Sarhan & Gawdan, 2017) research paper. The methodology consists of (1) the test scenarios, (2) the test conditions, (3) the test metrics, and (4) the test setup. The performance of the BM and KMP algorithms has been compared empirically in terms of execution time by using this methodology.

5.1. Test Scenarios

The performance of the **BM** and **KMP** algorithms is compared empirically according to test scenarios. This test uses six text sizes (1 KB, 10 KB, 100 KB, 1 MB, 100 MB and 200 MB). For the text sizes (1 KB to 1 MB), three pattern sizes have been used (10, 20, and 30 characters) and for the text sizes (100 MB and 200 MB) three different pattern sizes have been used (30, 50 and 75 characters). It should be noted that the test scenarios stated earlier have been selected to cover all aspects of each algorithm's performance.

5.2. Test Conditions

This study has considered the following conditions:

- Each algorithm has been tested with the same test scenario mentioned earlier.
- The test program is executed on the same computer.
- All applications have been closed except test program.
- During the test process, the computer is disconnected from the Internet.
- The time required to find the pattern within the text has been considered.
- Every test scenario has been recorded using the following JMH properties:

– Forks: 5 (It means five times the program will do warm up and measurement process).
– Warm-up: 5 iterations, 10 second each (It means that the program will do five warm up iterations each iterate will take 10 seconds).
– Measurement: 5 iterations, 10 second each (It means that the program will do five measurement iterations each iterate will take 10 seconds).
– Threads: 1 thread (The program will use only one thread).
– Benchmark mode: Average time (the time required to execute all measurements will be averaged at the end of the benchmark).
– Time unit: milliseconds

5.3. Test Metrics

The time required to run the algorithms have been stated as a measurement to evaluate and compare empirically the performance of both algorithms. Therefore, the algorithm with less execution time is considered as the best algorithm with regard to performance.

5.4. Test Setup

The specifications of software and hardware that has been considered in this study are shown in Tables 2.

Table (2): Software and Hardware Specifications

Software	Version
Java Development Kit (JDK)	1.8.0_211
Eclipse IDE	2019-03 (4.11.0)
Java Micro Benchmark Harness (JMH)	1.21
Operating System (Microsoft Windows)	10 Pro (64-bit)
Hardware	**Detail**
Computer System Model	Dell OptiPlex- 9010
CPU Type	Intel Core i7-3770
CPU Speed	3.4 GHz
CPU Cores	8
RAM	8 GB

6. Experimental Results

Figures 1-6 and Table 3 summarizes the results obtained from implementing the **BM** and **KMP** algorithms. Figures (1 – 6) represent the execution time of both algorithms for each text size in the test scenarios. Each figure presents the execution time in milliseconds as y-axis and pattern size (number of characters) as x-axis. Table 3 shows the execution time in milliseconds of both algorithms on the given texts and patterns samples.

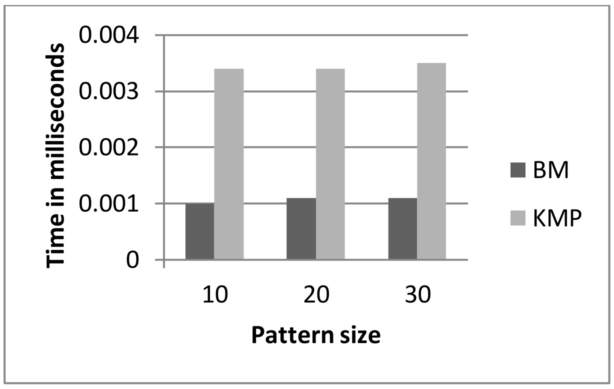

Fig. (1): The execution time of BM and KMP algorithms (Text size 1KB)

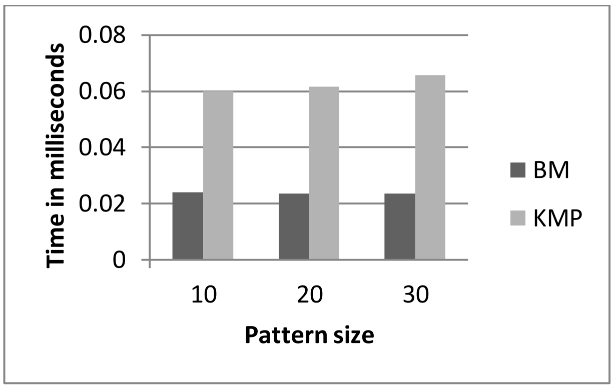

Fig.(1): The execution time of BM and KMP algorithms (Text size 10 KB)

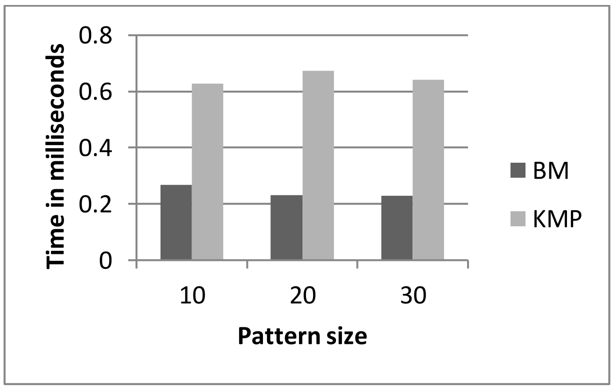

Fig. (2): The execution time of BM and KMP algorithms (Text size 100 KB)

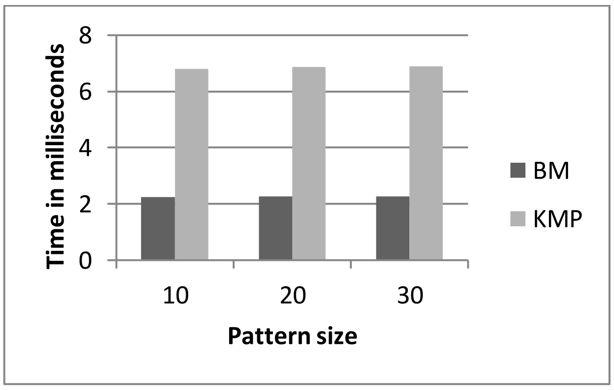

Fig. (3): The execution time of BM and KMP algorithms (Text size 1MB)

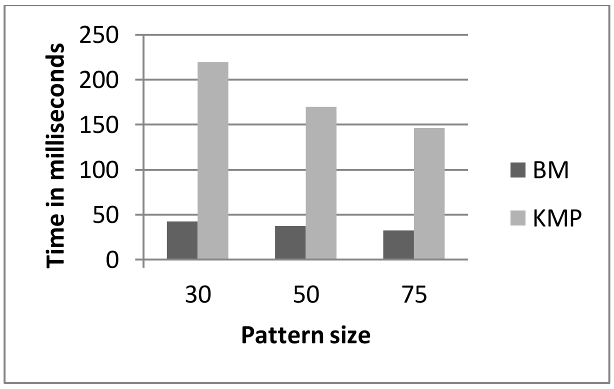

Fig. (4): The execution time of BM and KMP algorithms (Text size 100 MB)

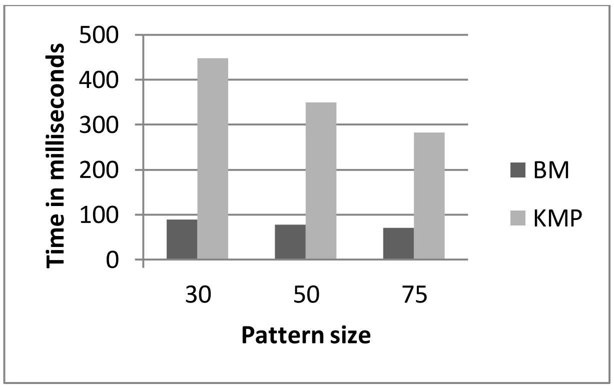

Fig. (6): The execution time of BM and KMP algorithms (Text size 200 MB)

Table (3): Experimental results of BM and KMP algorithms

Text Size	Pattern Size (Characters)	BM Search Time (MS)	KMP Search Time (MS)
1 KB	10	**0.0010**	0.0034
	20	**0.0011**	0.0034
	30	**0.0011**	0.0035
10 KB	10	**0.0241**	0.0601
	20	**0.0235**	0.0616
	30	**0.0236**	0.0657

100 KB	10	**0.2679**	0.6286
	20	**0.2314**	0.6723
	30	**0.2289**	0.6418
1 MB	10	**2.2403**	6.7932
	20	**2.2601**	6.8723
	30	**2.2616**	6.8811
100 MB	30	**42.7821**	219.5554
	50	**37.3412**	170.0776
	75	**32.7880**	146.5829
200 MB	30	**88.5497**	448.3035
	50	**78.1193**	349.7801
	75	**70.5093**	282.3131

7. DISCUSSION

From Table 3 (bold represents better results), it is shown that BM algorithm outperform KMP algorithm in all test scenarios. Figure 1–4 shows that BM algorithm has executed approximately in 0.001, 0.02, 0.2 and 2 milliseconds for text sizes 1 KB, 10 KB, 100 KB and 1 MB respectively for all patterns. Also, Figure 1–4 shows that KMP has executed approximately in 0.003, 0.06, 0.6 and 6-milliseconds for text sizes 1 KB, 10 KB, 100 KB and 1 MB respectively for all patterns. It can be noted that these four test scenarios have not been affected by the pattern size very much. Also, it can be seen that BM algorithm three times faster than KMP algorithm for these test scenarios. However, these results have been changed when the text and pattern sizes have been increased.

Figure (5) presents the execution of BM and KMP algorithms over text size (100 MB). The BM algorithm has executed approximately in (42, 37 and 32) milliseconds for pattern sizes (30, 50 and 75 characters) respectively. While, The KMP algorithm has executed approximately in (219, 170 and 146) milliseconds for pattern sizes (30, 50 and 75 characters) respectively. Also, the figure (6) presents the execution of BM and KMP algorithms over text size (200 MB). The BM algorithm has executed approximately in (88, 78 and 70) milliseconds for pattern sizes (30, 50 and 75 characters) respectively. While, The KMP algorithm has executed approximately in (448, 349 and 282) milliseconds for pattern sizes (30, 50 and 75 characters) respectively. It can be seen that the execution time of both algorithms have been decreased when the pattern size increased. Especially KMP algorithm, in which the BM algorithm executed around (5) times faster than KMP algorithm when the pattern size was (30) characters; While, this value is reduced when the pattern size became (75) for both test scenarios.

8. CONCLUSION

This research introduced an empirical study on the performance of BM and KMP algorithms in terms of time required to execute the algorithm. Different text and pattern sizes have been used to achieve this goal. These algorithms are programmed in Java and the experiment has been done using Java Microbenchmark Harness (JMH) tool. The overall performance evaluation showed that the BM algorithm outperformed the KMP algorithm in all test scenarios. The test scenarios (1 – 4) showed that BM algorithm is around (3) times faster than KMP algorithm and the pattern size did not effect on the performance of both algorithms very much. However, in the test scenarios (5 - 6) when the text and pattern sizes have increased the execution times of both algorithms have decreased. In the future, more work can be done as: (a) evaluating the algorithms with different languages such as Arabic or Kurdish language (b) measuring the effect of using different text and pattern sizes on the performance of each algorithm.

REFERENCES

– AbdulRazzaq, A. A., Rashid, N. A. A., Hasan, A. A., & Abu-Hashem, M. A. (2013). The exact string matching algorithms efficiency review. *Global Journal on Technology*, *4*(2), 576–589.
– Boyer, R. S., & Moore, J. S. (1977). A fast string searching algorithm. *Communications of the ACM*, *20*(10), 762–772. https://doi.org/10.1145/359842.359859
– Catania, L. (2018). *A fast string matching algorithm with moderately long patterns and small alphabets.* https://doi.org/10.13140/RG.2.2.13345.51040

- Chandraseta, R. (2017). *Optimization of Boyer-Moore-Horspool-Sunday Algorithm*. Retrieved from https://www.semanticscholar.org/paper/Optimization-of-Boyer-Moore-Horspool-Sunday-Chandraseta/d740cce2b915860a3ac6b237ea818655f9f2a5fd
- Charras, C., & Lecroq, T. (2004). *Handbook of Exact String Matching Algorithms*.
- Choudhary, R., Rasool, A., & Khare, N. (2012). Variation of Boyer-Moore String Matching Algorithm: A Comparative Analysis. *International Journal of Computer Science and Information Security*, *10*(2), 95–101.
- De V. Smit, G. (1982). A comparison of three string matching algorithms. *Software: Practice and Experience*, *12*(1), 57–66. https://doi.org/10.1002/spe.4380120106
- Ersin, A. K., Carus, A., & Mesut, A. (2007). *The Efficiency Of String Matching Algorithms On Natural Languages*.
- Gurung, D., Chakraborty, U. K., & Sharma, P. (2016). Intelligent Predictive String Search Algorithm. *Procedia Computer Science*, *79*, 161–169. https://doi.org/10.1016/j.procs.2016.03.116
- Knuth, D. E., Morris, Jr., J. H., & Pratt, V. R. (1977). Fast Pattern Matching in Strings. *SIAM Journal on Computing*, *6*(2), 323–350. https://doi.org/10.1137/0206024
- Oracle. (2019). Code Tools: jmh. Retrieved June 1, 2019, from https://openjdk.java.net/projects/code-tools/jmh/
- Rahim, R., Zulkarnain, I., & Jaya, H. (2017). A review: search visualization with Knuth Morris Pratt algorithm. *{IOP} Conference Series: Materials Science and Engineering*, *237*, 12026. https://doi.org/10.1088/1757-899x/237/1/012026
- Sarhan, Q. I., & Gawdan, I. S. (2017). Java Message Service Based Performance Comparison of Apache ActiveMQ and Apache Apollo Brokers. *Science Journal of University of Zakho*, *5*(4), 307–312. https://doi.org/10.25271/2017.5.4.376
- Sunday, D. M. (1990). A Very Fast Substring Search Algorithm. *Commun. ACM*, *33*(8), 132–142. https://doi.org/10.1145/79173.79184
- Tsarev, R., Chernigovskiy, A., A Tsareva, E., V Brezitskaya, V., Yu Nikiforov, A., & A Smirnov, N. (2016). Combined string searching algorithm based on knuth-morris-pratt and boyer-moore algorithms. *IOP Conference Series: Materials Science and Engineering*, *122*, 12034. https://doi.org/10.1088/1757-899X/122/1/012034
- Wahlström, S. (2013). *Evaluation of String Searching Algorithms*. Retrieved from https://www.semanticscholar.org/paper/Evaluation-of-String-Searching-Algorithms-Wahlström/8afc6c601aa4ae2e0878c943735e75935e995b58