NIPES

Journal of Science and Technology Research

Journal homepage: www.nipesjournals.org.ng

Analysis and Performance Evaluation of Selected Pattern Matching Algorithms

Princewill Aigbe[1], Emmanuel Nwelih[2]

[1]Department of mathematics & Computer Science, Western Delta University, Oghara, Delta State, Nigeria

[2]Department of Computer Science, University of Benin, Benin City, Edo State, Nigeria

Corresponding Author email: ***[1]aqbonx@yahoo.com, [2]emmanuel.nwelih@uniben.edu***

Article Info

Received 21 April 2021
Revised 19 May 2021
Accepted 22 May 2021
Available online 04 June 2021

***Keywords*:**
Pattern matching algorithms; DNA; Bioinformatics.

https://doi.org/10.37933/nipes/3.2.2021.7

https://nipesjournals.org.ng

Abstract

Pattern matching algorithms are used to find one or more patterns or sequence of patterns from huge text data or symbols. Pattern matching algorithms work for many purposes in engineering as well as in bioinformatics. Mostly, pattern matching algorithms are used for information retrieval, intrusion detection, data compression, content filtering, bioinformatics for analyzing DNA and other genome sequences. In this paper, selected pattern matching algorithms applied in the domain of document processing are discussed. The discussion of these algorithms is based on analysis of operations, execution time, and the number of comparisons required to locate or determine a pattern. The investigation culminated in the determination of the algorithm with the best performance characteristics.

1. Introduction

Pattern matching also known as string matching is the process of checking a perceived sequence of string for the presence of the constituents of some pattern. This means that in string matching pattern operation, strings are searched within a larger string or text. Assume that there is pattern string “p” and text string “S“, the problem of string matching deals by finding whether a pattern set “p“ occurs in “S“ or not. If “p“ occurs then the position of it should be reported in “S“ where “p“ occurs [1]. There are two types of string matching, and these are the exact string matching and approximate string matching. The search to be done on exact occurrence of the pattern comes underneath the category of exact string matching. Approximate string matching allows inaccurate searching acceptance founded on specific applications. Based on the number of patterns, string matching has two classifications: Single Pattern string matching and Multiple Pattern string matching. In Single Pattern string matching, a single pattern is to be searched in the text whereas in multiple pattern string matching, multiple patterns are searched in the text. Based on the order of searching string matching have four classifications i.e. left to right matching, right to left matching, specific order matching and no order matching [2].

There are many applications in which string matching plays important role. These applications are Spell Checkers [3], Spam Filters, Intrusion Detection System [4], Search Engines, Plagiarism Detection, Bioinformatics, Digital Forensics [5] and Information Retrieval Systems [6].

The very basic and conventional string matching strategy is Brute Force Algorithm which considers all possible cases and taking shifts only one place to right even match or mismatch condition occurs anywhere. This algorithm is known as Naïve approach. Avoiding numerous comparisons in brute force algorithm, Morris and Pratt algorithm was proposed which has a linear behaviour [7]. This algorithm is based on preprocessing of pattern and compares character from left to right and if mismatch occurs, it skips some character based on pre-processing phase. In 1977, Knuth, Morris and Pratt introduced an algorithm having a choice of improvements in Morris and Pratt algorithm [8]. KMP has same time complexity as Morris and Pratt algorithm but searching performance found to be much better than Morris and Pratt algorithm. In 1977, Boyer and Moore also proposed algorithm which compares character from right to left [9].
Different pattern or string matching algorithms has already been proposed in past decades in specific area of applications such as screen scrappers, parsers, digital libraries, web search engines, computational molecular biology, natural language processing, etc. and in this paper, the analysis and performance evaluation of characteristic features of some selected algorithms that have found applications in word-processors and document processing will be the focus point.

2. Methodology

The analysis of the selected pattern matching algorithms were carried out to establish their execution time and number of comparison required to locate or determine a pattern in a specified text. The selected pattern matching algorithms were programmed using Java programming language. The choice of this programming language is based on the availability of the required application programming interface (APIs). The culminated application programs for each pattern matching technique was executed using a computer specific features to generate the required information such as execution time, number of comparison, etc. The results generated from the evaluation process were further perused and shown using both 2D and 3D graphs. The graphs are plotted using MATLAB programming language.

3. Results and Discussion

3.1. Analysis of the selected Pattern Matching Algorithms

A pattern matching algorithm aims to find one or more occurrences of a desired substring pattern within another. The returns the position of the first character of the desired substring in the text. Given a text of length n, denoted as text[n] and pattern of length m, denoted as pattern[m], the occurrence(s) of the pattern[m] in the text[n] is often required in some forms of operations especially in document processing. The existence of the desired pattern[m] in text[n] can be determined using pattern matching algorithms with various properties or features such as the number of comparisons that eventually gives a measure of the time complexity of the algorithms [10]. The selected algorithms that can perform the stated operation are discussed as follows:

a. Naïve Pattern Matching Algorithm (NA)

Using this algorithm, the occurrences of the desired pattern[m] in text[n] can be determined, in which the first step is to place or set the pattern[m] to the left end of the text[n] and matching process starts. After a mismatch is found, pattern[m] is shifted one place to the right, a new matching process starts, and so on. The pattern and text are in the arrays pattern [1..m] and text [1..n] respectively. The operation pattern[i] = = text]j] in the algorithm is a comparison between characters, and measures the complexity of the naïve algorithm by the number of character comparisons. The rest of computing time is proportional to this measure. Hence in the naïve pattern matching algorithm, the time complexity is obviously O(nm) [11].

b. Rabin-Karp Pattern Matching Algorithm (RK)

The Naive Pattern Matching algorithm slides the pattern one by one. After each slide, it checks one character after another at the current shift and if all characters match then, displays a match is found. Like the Naive Algorithm, Rabin-Karp algorithm also slides the pattern one by one but, it determines a match by comparison of the hash value of the pattern with the hash value of current substring of text, and if the hash values match then only it starts matching individual characters [12]. So Rabin-Karp algorithm needs to calculate hash values for following strings:

i. Pattern itself of length n
ii. Text substrings of length m

Since it is required that hash values must be efficiently calculated for all the substrings of size m of text, a hash function which has following property: Hash at the next shift must be efficiently computable from the current hash value and next character in text must be efficiently computable from the following expression:

hash (text[s..s+m-1]) and text[s+m]. That is hash (text[s+1... s+m]) = rehash (text[s+m], hash (text[s.. s+m-1])). Rehashing is done using the following formula:
hash(txt[s+1.. s+m]) = (d (hash(txt[s .. s+m-1]) – txt[s]*h) + txt[s + m]) mod q.
The rehash operation has a constant time complexity.

The hash function suggested by Rabin and Karp calculates an integer value. The integer value for a string is numeric value of a string. For example, if all possible characters are from 1 to 10, the numeric value of "122" will be 122. The number of possible characters is higher than 10 (256 in general) and pattern length can be large. So the numeric values cannot be practically stored as an integer. Therefore, the numeric value is calculated using modular arithmetic to make sure that the hash values can be stored in an integer variable that can be accommodated in memory. In carrying out rehashing, the most significant digit is take off and add the new least significant digit in hash value [13].

The average and best case running time of the Rabin-Karp algorithm is O(n+m), but its worst-case time is O(nm). Worst case of Rabin-Karp algorithm occurs when all characters of pattern and text are same as the hash values of all the substrings of text length match with hash value of pattern length n.

c. Knuth-Morris-Pratt pattern matching algorithm (KMP)

This algorithm was developed to improve on the running time of the naïve pattern matching algorithm. Whenever there is a shift of the pattern[m] to the right after a mismatch is found at j on the pattern and i on the text, the matching history for the portion pattern [1..j] and text[i – j + 1 .. j] is not utilized in the matching operation.

The KMP pattern matching algorithm uses degenerating property of the pattern and improves the worst case complexity to O(n) [14]. The basic idea behind KMP's algorithm is that whenever a mismatch is detected after some matches, it is already known that some of the characters in the text of the next window, and the matching operation process takes advantage of this information to avoid matching the characters that the operation knows will anyway match. In this way the KMP algorithm makes use of the matching history information to determine number of shifts required in advance, and this guarantees the liner-time performance of the algorithm.

d. Boyer-Moore pattern matching algorithm (BM)

Boyer Moore is a combination of the following two approaches. These are:
i. Bad Character Heuristic
ii. Good Suffix Heuristic
The two stated heuristics approach can also be used independently to search a pattern in a text. The Boyer Moore algorithm does preprocess so that the pattern can be shifted by more than one. It processes the pattern and creates different arrays for both heuristics. At every step, it slides the pattern by the maximum number of the slides suggested by the two heuristics. So it uses best of the two heuristics at every step. Unlike the previous pattern searching algorithms, Boyer Moore algorithm starts matching from the last character of the pattern [15].

The idea of bad character heuristic is simple. The character of the text which doesn't match with the current character of the pattern is called the Bad Character. Upon mismatch, the pattern is shifted until:

i. The mismatch becomes a match;
Here, there will be a lookup of the position of last occurrence of mismatching character in pattern and if mismatching character exist in pattern then the pattern is shifted such that it get aligned to the mismatching character in text T, and

ii. Pattern P move past the mismatched character;
At this point, there is a lookup of the position of last occurrence of mismatching character in pattern and if character does not exist the pattern will be shifted past the mismatching character. Therefore, the bad character heuristic takes O(n/m) time in the best case, where n is the length of text and m is the pattern length.

e. Anagram substring pattern matching algorithm (AS)
Given a text denoted as txt[0..n-1] and a pattern as pat[0..m-1], all occurrences of pat[m] and its permutations or anagrams in txt[n] can be determined using anagram substring pattern matching algorithm. In this algorithm, it may be assumed that n > m and the simple idea in this application is the modification of Rabin-Karp algorithm [16]. The hash value as sum of ASCII values of all characters is kept under modulo of a big prime number. For every character of text, the current character is added to hash value and subtract the first character of previous window. This technique looks good, but like standard Rabin-Karp, the worst case time complexity of this approach is O(mn), and this worst case occurs when all hash values match and which is a one by one match of all characters.

It is possible to achieve O(n) time complexity under the assumption that alphabet size is fixed which is typically true as there are maximum 256 possible characters in ASCII. The idea is to use two count arrays:
i. The first count array stores frequencies of characters in pattern.
ii. The second count array stores frequencies of characters in current window of text.
The important thing to look at here is, time complexity to compare two count arrays is O(1) as the number of elements in them are fixed and that is independent of pattern and text sizes. The algorithm entails the following steps:
1. Store counts of frequencies of pattern in first count array pattern_count. Also store counts of frequencies of characters in first window of text in array text_count_window.
2. Iterates the loop from i = m to n-1. Execute the following steps in loop:
2.1 If the two count arrays are identical, an occurrence is found.
2.2. Increment count of current character of text in test_count_window
2.3 Decrement count of first character in previous window in count_window

3 The last window is explicitly checked but, not checked by above loop [17].
The characteristic features of the selected algorithms in terms of pattern searching techniques and performance behaviour with regard to time complexity is depicted in Table 1.

Table 1: Characteristic features of the selected algorithms

S/N	Pattern Matching Algorithms	Pattern Matching Technique	Time Complexity	
			Best Case	Worst Case
1.	Naïve (NA)	Continuous comparison until a match or mismatch	O(nm)	O(nm)
2.	Knuth-Morris-Pratt (KMP)	Use two indices to run through the text	O(n)	O(n)
3.	Rabin-Karp (RK)	Hashing	O(n+m)	O(nm)
4.	Boyer-Moore (BM)	Use both good suffix and bad character shift	O(n/m)	O(nm)
5.	Anagram Substring (AS)	Hashing with frequency counts	O(n)	O(nm)

Note: n = text length, and m = pattern length

3.2 Performance Evaluation of the selected Algorithms

The execution of the developed programs was carried out with different input sizes of text and a constant pattern size. The result generated from the execution process are shown in Table 2. The tabulated results from the evaluation of the five algorithms shows that each algorithm exhibits different running time and number of comparisons to determine or find a match of the required pattern. It is also shown from the results that pattern matching algorithms with higher running time have greater number of comparisons in locating the required pattern in a given text. The graph in Figure 1 is a plot of the pattern search time (i.e., running time) against the input size of the selected pattern matching algorithms. The graph shows the performance behaviour of the algorithms when subjected to different text input sizes.

Table 2: Performance Evaluation of the selected Algorithms

Test Cases and Input size in Megabytes	Running Time in Seconds (s) and number of Comparisons of the Algorithms:				
	NA	KMP	RK	BM	AS
10	0.09 **8**	0.04 **5**	0.06 **4**	0.01 **1**	0.05 **3**
20	0.09 **8**	0.08 **6**	0.06 **5**	0.01 **1**	0.03 **3**
30	0.07 **9**	0.05 **5**	0.07 **6**	0.01 **2**	0.04 **3**
40	0.29 **9**	0.12 **3**	0.17 **4**	0.10 **2**	0.12 **4**
50	0.35 **9**	0.18 **4**	0.14 **3**	0.12 **2**	0.17 **5**

Note: 1 MB of input size of patter was used for the evaluation, NA = Naïve Algorithm
KMP = Knuth-Morris-Pratt, RK = Rabin-karp, BM = Boyer-Moore
Number of Comparison values bolded

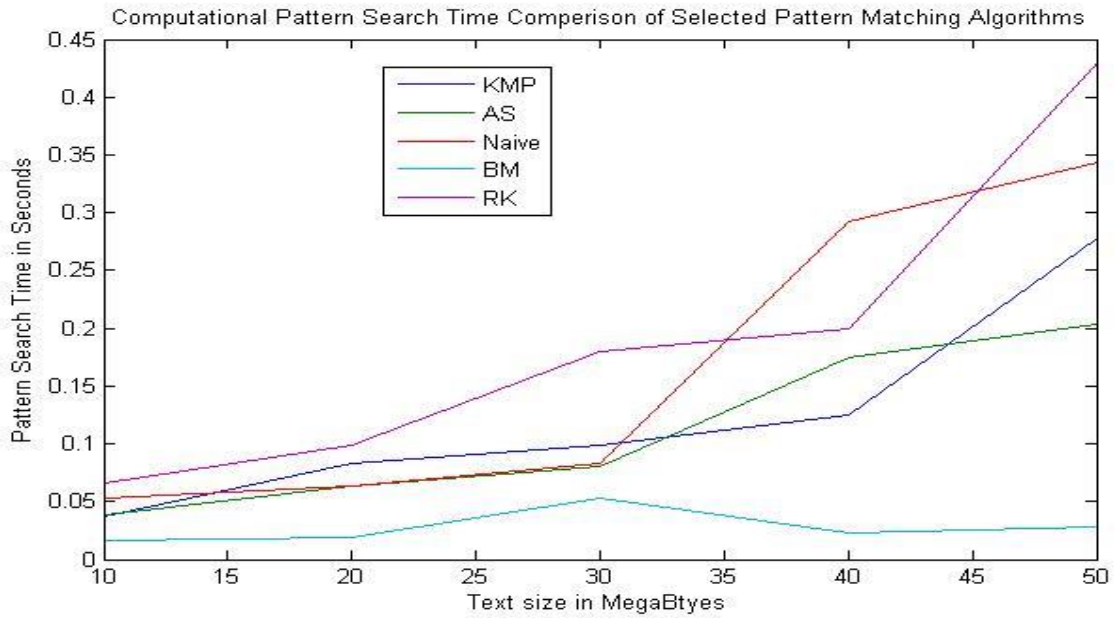

Figure 1: Execution time of the selected pattern matching algorithms.

Similarly, the graph in Figure 2 is a 3D plot of the pattern search time (i.e., running time), text input size and number of comparisons for the specified pattern search generated from the execution of the selected pattern matching algorithms.

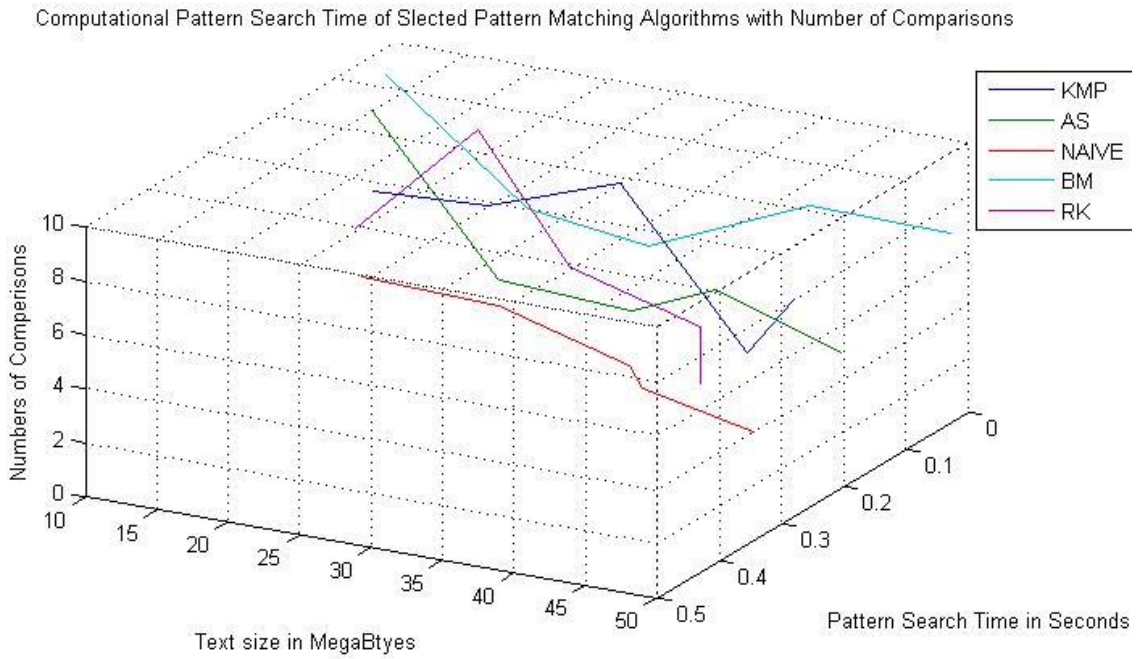

Figure 2: Number of comparisons and execution time of the algorithms.

4. Conclusion

Pattern matching algorithms are very useful for pattern search operations in document processing. Each of the pattern matching algorithm investigated has its own characteristic features. In terms of complexity of each algorithm, Knuth–Morris–Pratt algorithm has less time complexity and Boyer-Morris algorithms with least preprocessing time complexity while the naïve algorithm has the largest running time as well as the highest number of comparisons to locate or determine a pattern

in search operation. The Boyer Morris and Knuth–Morris–Pratt algorithm are more effective for searching with fewer number of comparisons when compared to the other pattern matching algorithms investigated.

References

[1] Haraee, H., Shokoufeh, S., and Nima, M. (2014). A survey of pattern matching algorithm in intrusion detection system. 7th International Symposium Telecommunications (IST) on. IEEE).
[2] Alberto, A. and ZviGalil, D. (2007). Pattern Matching Algorithms. Oxford University Press, USA, 1st edition.
[3] Ali, P. (2010). Application of string matching in Internet Security and Reliability. Marsland Press Journal of American Science, Volume 6, Issue 1, pp. 25 - 33.
[4] Ching-Tung, W., Kwang-Ting, C., Qiang, Z. and Yi-Leh, W. (2005). Using Visual Feature for Anti-Spam Filtering. In the proc. of IEEE International Conference on Image Processing (ICIP2005), pp. 509 – 512.
[5] Jooyoung, L., Sungkyung, U., and Dowon, H. (2009). Improving Performance in Digital Forensics: A Case using pattern matching board. In the Proc. of International Conference on Availability, Reliability and Security (ARES), pp. 1001 – 1005.
[6] Lin, U. and Cheng-Hung, Z. (2013). Accelerating pattern matching using a novel parallel algorithm on gpus. Computers, IEEE Transactions, Volume 62, Issue 10, pp. 1906 – 1916.
[7] Morris, J. H. and Pratt, V. R. (1970). A linear pattern-matching algorithm. Technical Report 40, University of California, Berkeley, pp. 78 - 94.
[8] Nimisha, S. and Deepak, G. (2012). String Matching Algorithms and their Applicability in various Applications. International Journal of Soft Computing and Engineering (IJSCE), Volume 1, Issue 6, pp. 56 – 64.
[9] Boyer, R. S. and Moore, J. S. (1977). A fast string searching algorithm. Communication of ACM 20, Volume 10, pp. 762–772.
[10] Hyunjin, K., Hong-Sik, K. and Sungho, K. (2011). A Memory- Efficient Bit-Split Parallel String Matching Using Pattern Dividing for Intrusion Detection Systems. IEEE Transactions on Parallel and Distributed Systems, Volume 22, Issue 11, pp. 1904 - 1911.
[11] Knuth, D., Morris J., and Pratt, V. (1977). Fast pattern matching in strings. In the procd. Of SIAM Journal of Computing. Volume 6, Issue 1, pp. 323 – 350.
[12] Jingbo, Y., Jisen, Z., and Shunli, D. (2010). An Improved Pattern Matching Algorithm. In the Proc. of Third International Symposium Intelligent Information Technology and Security Informatics, pp. 599 - 603.
[13] Sanchez, D., Martin-Bautista, M., Blanco, I. and Torre, C. (2008). Text Knowledge Mining: An Alternative to Text Data Mining. In the proc. of IEEE International Conference on Data Mining Workshops, ICDMW, pp. 664 - 672.
[14] Mei-Chen, Y. and Kwang-Ting, C. (2011). Fast Visual Retrieval Using Accelerated Sequence Matching. IEEE Transactions on Multimedia, Volume 13, Issue 2, pp. 56 -68.
[15] Lok-Lam, C., David, W., and Siu-Ming, Y. (2003). Approximate String Matching in DNA Sequences. In Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA), pp. 303 – 310.
[16] Bhukya, R. and Somayajulu, D. (2011). Exact multiple pattern matching algorithm using DNA sequence and pattern pair. International Journal of Computer Applications, Volume 17, Issue 8, pp. 32 – 38
[17] Cormen, T, Leiserson, E., Rivest, R., L., and Stein, C. (2001). The Rabin–Karp algorithm. Introduction to Algorithms (2nd ed.), Cambridge, Massachusetts, MIT Press, pp. 911 – 916.