Comparative Study to Extract Pattern Matching Approaches using Diverse DNA Sequences

Kavita[1]
Professor,
Kavitasaini_2000@yahoo.com
Galgotias University, Noida, India

Nidhi Raj Singh[2,] Adarsh kumar[3]
Undergraduate Student
Galgotias University, Noida, India
nnidhiraj123@gmail.com
adarshkumarjjr@gmail.com

***Abstract*— Exact DNA Sequence Matching is an important aspect of modern software for controlling and evaluating biological data. In the realm of biotechnology, the Knuth Morris Pratt (KMP) Algorithm is commonly used to process DNA sequences. The KMP method has linear time complexity in relation to the string pool length, but it also has linear space complexity in relation to the pattern length. The proposed methodology in this paper is the Rabin Karp Algorithm which has a linear time complexity of the summation of the size of the string pool and the pattern to be matched, that is overall a linear time complexity, but with this algorithm a constant space complexity can be achieved maintaining the overall same time complexity that the existing KMP algorithm has. The Rabin Karp method, on the other hand, does not perform well when all pattern and text characters are the same as the hash values of all substrings. This study compares and contrasts the Naive Method, KMP Algorithm, and Rabin Karp Algorithm to discover which algorithm is best suited for DNA Sequence Check.**

Index Terms: DNA Pattern Matching, Rabin Karp Algorithm, DNA Sequencing, KMP Algorithm, Boyer- Moore Algorithm, Pattern Matching Algorithm.

I. INTRODUCTION

Bioinformatics is the study, management, and assessment of biological problems using computer science. The volume of biological data has grown dramatically as the computing era has progressed. As a result of these large volumes of data, the need to review vast amounts of data in a reasonable amount of time and space has grown. Pattern matching across species is a tough problem to tackle because DNA sequences convey fundamental species information. There are various generic string matching algorithms and several particular DNA pattern matching tools in the literature. Fast and space-efficient pattern matching algorithms that take into account fresh hardware advancements are still required. In biotechnology, the Knuth Morris Pratt (KMP) Algorithm is commonly used to process DNA sequences.

The KMP method has linear time complexity in relation to the string pool length, but linear space complexity in relation to the pattern length. The Rabin Karp Algorithm is the methodology proposed in this paper, which has a linear time complexity of the summation of the string pool size and the pattern to be matched, as well as a linear time complexity overall, but can achieve a constant space complexity while maintaining the same overall time complexity as the existing KMP algorithm. When all pattern and text characters are the same as the hash values of all substrings, the Rabin Karp method is useless. To identify which algorithm is better for DNA Sequence Check, this study examines and contrasts the Naive Method, KMP Method, and Rabin Karp Method.

II. RELATED WORK

This section contains relevant papers on pattern matching. Brute force (BF) is a prominent approach in the pattern matching literature that does the text or the pattern. From left to right, BF one by one. The sliding window is relocated one placeto the right after a match or mismatch, and there the pattern. The fact that BF takes so long to complete is a major drawback. There are approaches that combine the dynamic programming paradigm and DFA and are based on Deterministic Finite Automata (DFA). These approaches are seldom scalable for extended sequences due to the usage of finite automata. Furthermore, because of the use of dynamic programming, the memory needs are significantly higher.

The KMP technique, created by Knuth et al., compares from the left side. When a mismatch occurs, KMP slides waiting the window as also in the to the right, preserving the largest landing on the overlap between the matched text's suffix and the pattern's prefix. The performance of this algorithm is linear. The KMP technique works well when thealphabet size is large, but it takes a long time to execute when the alphabet size is not very big or we can say small is small or the pattern length is short. The Boyer-Moore method and its variations look for a right-to-left pattern in the text. To put it another way, this algorithm favors the making of the hallucination pattern's last of the render machine character. At the end of the matching phase, itcalculates the shift increment. When a mismatch occurs, two helpful tips are used to limit the number of comparisons (bad character and good suffix). The disadvantage rattling feature of the Boyer Moore way of doing this pattern matching is that it takes a long time to prepare depending on the patterning version length and size.

Comparisons are the foundation of the DCPM method. At the start of the DCPM's preprocessing a result, in order to identify the windows, DCPM runs the text through two cycles and conducts a certain stage, the text is checked of the pattern's rightmost character. The discoveries index is found in the rightmost character table. The text is then re-examined to locate the pattern's leftmost character. The indices in the leftmost character table are retained in the event of similarity. The window limitations are determined by DCPM using these two tables. To put it another way, the pattern's length is used to probe the contents of the database. We'll identify a window when the distance between the leftmost and rightmost character (extracted earlier from the two tables) equals the pattern length. As a result, in order to identify the windows, DCPM runs the text through two cycles and conducts certain computations. During the matching stage, the algorithm checks the other characters in the windows.

Perfect sameness is attained when all of the pattern's characters and text windows match. In this study, the first technique aids DCPM by recognising windows with only one text pass.

III. Problem Statement

Even if the amount of biological data has increased dramatically in recent years, these enormous amounts of data must nevertheless be analyzed in an acceptable period of time. This problem arises because amino acid or nucleotide sequences are commonly used to represent biological molecules in molecular biology. In order to discover any inconsistencies or flaws in a DNA sequence, DNA sequence analysis is frequently necessary. Pattern matching is also useful in domains like phylogenetics and evolutionary biology. To understand their relatedness, descent, and origin, these programmes extract particular DNA subsequences from the genomic data of organisms of various kinds. As a result, a descent, and origin pattern matching algorithm in this through datasets ranging from gigabytes to terabytes in size, as well as complete genomes, DNA sequences, on the other hand, are fairly lengthy. As a result, the time spent matching up with the pattern is the most important.

IV. Methodologies

Methodologies are divided into several parts, which are described in the below sub-sections.

1) Naive Pattern Matching

The Naive algorithm is a brute-force strategy. It's a straightforward method for locating any text string. It iterates across the text several times, finding the pattern's length by comparing it to the first few characters. If there isa mismatch, move the syndrome abderrahmane to the right one step and compare the following character of text with the first character of the pattern, compare the text and pattern characters after that. Carry on the above process as before; if a match is discovered throughout the whole length of the pattern, it means the pattern was spotted; as a result, return the match's position. The time complexity is O(m*n) in both the worst and best scenarios, where (m) is the length of string and (n) is the length of pattern.

Function Naive-Algorithm (Text, Pattern) :

Input: Pattern [1. m] Text [1. n]

Output: Position of the substring of text matching Pattern or -1 if not matched then its returned for j← 0 to n-m do i← 0

While i<m && Pattern [i]==Text [j+i]

do i←i+1

If i==m return j //match successful Return -1 // match unsuccessful

2) Knuth Morris Pratt Algorithm

DNA data matching, viral feature matching, data compression, and so on as the volume and complexity of data increases dramatically. String pattern matching has gotten a lot of attention in this industry for a long time [1]. One of the most fundamental operations in strings is pattern matching. If P is a given substring and T is a much longer string to be searched, all substrings in T that match P must be located. The pattern is denoted by the letter P, whereas the target is denoted by the letter T. In real-world situations, the initial step is to match the substring P with the string T using the brute force approach, starting with the first element in T. Despite the fact that this method complies to the notion, it is inefficient and time- consuming [2-3].

The number of letter matching operations necessary to retrieve P from T is $m^*(n-m+1)$ if f (T)=n and f (P)=m, and "length" is the total number of letters in a string. By minimizing superfluous letter comparisons, the KMP approach, an updated string matching strategy developed at the same time by Knuth, Morris, and Pratt, has become fairly fast and popular today.

The KMP method traverses a pattern string from start to finish, looking for the longest common components between each substring's prefix and suffix, and recording the length of the common section in a "Failure Table" that should be the same length as the pattern [4]. Each letter in the pattern correlates to a number in the Failure Table that must be calculated. The comparison then begins with the first letter of P and T, with N_1 signifying "the last matched character's matching number in the Failure Table" and updating with the comparison process, and N_2 denoting 'the last matched character's matching number in the Failure Table. If the not-match situation starts with one of P's letters, P must change the following numbers to the right: (N_1-N_2). To get the same result, use the array next.

Function KMP-Algorithm (Text, Pattern) :

n ← length [Text]

m ← length [Pattern]

pi ← Compute Prefix Table (Pattern) q ← 0

for i := 0 To n-1

while q > 0 and Pattern [q] !=Text [i] do

q ← Pattern[q]

if Pattern [q] = Text [i]

Then q ++ if q = m

Then return i – m + 1

return –1

3) Drawbacks of Knuth Morris Algorithm

Even though this method is rather good, there is still a lot of room for development. As the pattern progresses, there are still a few matches that aren't required. More comparisons will be required when the pattern first appears in the second section of the text. One of the drawbacks with the KMP Algorithm – data is that it does not scale well as the alphabets grow in size. As a result, errors are becoming more prevalent

[5]. Processing large DNA files needs additional resources in the form of processors, which may be a hurdle to implementing the KMP Algorithm-data for smaller enterprises.

4) Rabin-Karp Algorithm

The Rabin-Karp algorithm is a hashing-based search strategy for finding a substring pattern in a text. Plagiarism detection is one of the practical uses of the Rabin approach. The proportion of plagiarism is calculated using a hash algorithm using Karp's Rabin-Karp. You may adjust the accuracy level using this function. The hash function is used to calculate the feature value of a certain syllable fraction. Each text is converted into a number known as a hash value

[6]. The hash value is calculated using the same phrase in the Rabin-Karp method. In order to determine the hash value, you must overcome two obstacles. For starters, a single sentence can include a large number of distinct strings [8]. This problem may be handled by assigning the same hash value to several strings. The second problem is that, although if each string is allocated to the brute- force approach, the string with the same hash value match does not have to be overcome. To avoid hash outputs that are identical to various words, Rabin-Karp requires a big prime integer. The scenario becomes rather difficult when fraudulent hits appear many times across all windows.

This is the case because a good hashing algorithm seldom produces collisions. As a result, comparing two substrings is rarely necessary if they do not match. In addition, if all we want to do is observe if the pattern appears, the complexity will be O(n+k), because we could break after the first time [7].

```
Data: s: String for which we want to calculate hash values
prefix: Prefix sum of ASCII codes in s
Function Init(s):
n ← Length(s); prefix[0] ← int(s[0]) for i ← 1 to n-1
do prefix[i] ← prefix[i-1] + int(s[i]); end

end

Function getHash(L,R):
if L = 0 then
return prefix[R];

      end

end

return prefix[R] - prefix[L-1];

Rabin-Karp String Matching

Data: P: The Pattern to look for T: The text to look in

hashT, hashP: Hash structures for strings T,

P Result: Returns occurrences of P in T

Answer ← 0; n ← length(T); k ← length(P);

for i ← 0 to n-k do

textHash ← hashT.getHash(i, i+k-1); patternHash ←
hashP.getHash(0, k-1); if textHash = patternHash then
valid ← true;

for j←0 to k-1 do

if T[i+j] != P[j] then

valid ← false;

break;

            end

   end


         end


if valid = true then

answer ← answer + 1;

end end

        return answer;
```

5) Boyer-Moore Algorithm

Like the nave technique, the Boyer-Moore method aligns P and T consecutively before assessing if P matches T's opposing characters. After the check is done, P is shifted right relative to T, just like in the prior step [8-10]. These methods are used to develop a methodology for analyzing less than m
+ n characters in linear worst-case time (a sublinear-time method) (with a specific extension) [11].

6) Relevance of Boyer - Moore Algorithm in DNA Pattern Matching

The bad character rule helps, but has no impact. When the alphabet is small and the text has numerous substrings that are similar but not identical, this is a regular occurrence. This is true of DNA, which has a four-letter alphabet, and even protein, which has a twenty-letter alphabet and usually has distinct parts that are quite identical.

```
{Preprocessing stage}

Given P as a pattern,

Calculate L' I and l'(i) for each I of P's position, and R(x) for
each character x.

{Search stage}

k := n;

while k ≤ m do

begin i := n;

h := k;

while i > 0 and P(i) = T(h) do

begin

i := i − 1; h

:= h − 1;

end; if i = 0 then

begin

P in T ending at position k should be reported.

k := k + n − l` (2);

  end

                else

  end;
```

Where, the maximum amount defined by the (extended) bad character rule and the good suffix rule is used to shift P (raise k).

7) Finding Best Suited Algorithms for DNA Pattern Matching By Comparative Analysis

From the above facts we can clearly derive that the size of a human DNA is of order 10^6. On implementation we have developed an artificial string of human genome or the same order and performed time analysis on the algorithms discussed above[12, 13, 14]. Figure 1 shows the algorithm time graphs of all the four algorithms that we have discussed for pattern size ranging from 10^1 to 10^5 characters. Fig. 1 shows the algorithm time graphs of all the four algorithms that we have discussed for pattern size ranging from 10^1 to 10^5 characters.

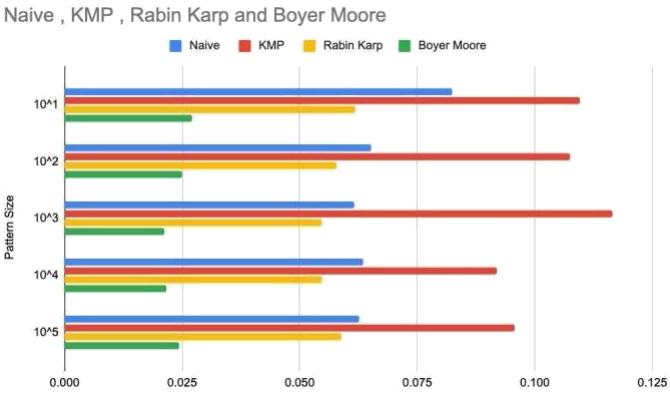
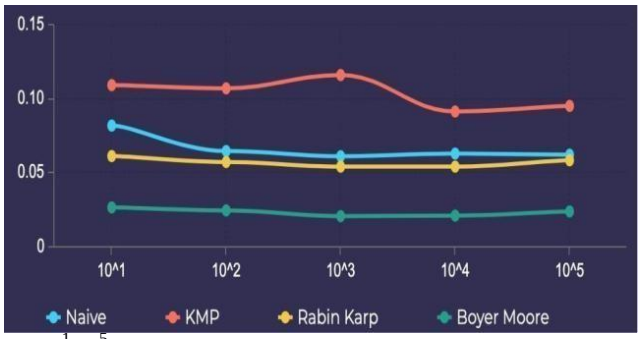


Fig. 1 Algorithm-Time horizontal histogram graph for pattern size



(m~10^1-10^5)

Fig. 2. Algorithm-Time Line graph for pattern size (m ~ 10^1-10^5)

Algorithm	**Time in sec m ~ 10^1**	**Time in sec m ~ 10^2**	**Time in sec m ~ 10^3**	**Time in sec m ~ 10^4**	**Time in sec m ~ 10^5**
Naive	0.082317	0.065112707	0.061608827	0.063417745	0.06263312
KMP	0.109626234	0.107455507	0.116449436	0.091818921	0.095603884
Rabin-Karp	0.061723139	0.05773003	0.054595293	0.054593009	0.058861485
Boyer-Moore	0.027069688	0.025041379	0.021167181	0.021548849	0.024340384

V. Conclusion And Future Work

We need to make sure that the algorithms we're employing presently are as efficient as possible. This is only achievable if you focus on keeping complexity and execution time to a minimum. The Advance Rabin Karp method enhances the performance of the Rabin Karp algorithm by reducing its execution time, resulting in a betterresult in less time. As our hypothesis, the DNA pattern is comparatively so small in size hence all the algorithms are tending to linear time complexities. We can see that asymptotically all the algorithms are performing in linear time. Among all the algorithms Boyer-Moore performed outstandingly well. That is because Boyer-Moore performs better in case of small alphabet size. In DNA sequences the alphabet size is 4. Hence we can conclude that the best algorithm for DNA pattern matching is Boyer-Moore Algorithm. In Future, we can perform analysis in space and according to theoretical analysis the Rabin-Karp would perform quite well in that case.

References

[1] Knuth, D.E., Morris, J.H., Pratt, V.R. (1977) Fast pattern matching in strings. SIAM Journal on Computing, 6(2) 323-350.

[2] Chang, C., Wang H., (2012). Comparison of two-dimensional string matching algorithms. In: 2012 International Conference on Computer Science and Electronics Engineering. IEEE. pp. 608-611.

[3] Song Y.U., Zheng J., Hu W.X. (2009) Improved KMP algorithm, Journal of East China Normal University, 32(4): 92-97.

[4] Morris Jr, J.H., Pratt, V.R. (1970) A linear pattern-matching algorithm, University of California, Berkeley.

[5] Knuth, D.E., Pratt, V.R. Automata theory can be useful, unpublished manuscripts.

[6] Altschul, Stephen F., Gish, Warren, Miller, Webb, Myers, Eugene W. and Lipman, David J. 1990. Basic Local Alignment Search Tool. Journal of Molecular Biology 215(3), pp. 403-410.

[7] Jason Coit, Stuart Staniford and Joseph McAlerney, "Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort".

[8] Kurtz. S, Approximate string searching under weighted edit distance. In proceedings of the 3rd South American workshop on string processing. Carleton Univ Press, pp. 156-170, 1996.

[9] A. P. Gope and R. N. Behera, "A Novel Pattern Matching Algorithm in Genome," International Journal of Computer Science and Information Technologies, vol. 5, no. 4, pp. 5450-5457, 2014.

[10] A. P. U. Siahaan, Mesran, R. Rahim and D. Siregar, "K-Gram As A Determinant Of Plagiarism Level In Rabin-Karp Algorithm," International Journal of Scientific & Technology Research, vol. 6, no. 7, pp. 350-353, 2017.

[11] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, vol.20, Session10, Oct.1977, pp.761– 772.
© 2013 Global Journals Inc. (US) 47Year 013 2 Global Journal of Computer Science and Technology Volume XIII Issue I Version I ( ) A Analysis of Parallel Boyer-Moore String Search Algorithm.

[12] P. J. C, and K. S. Panicker, "Single Pattern Search Implementations in a Cluster Computing Environment", in 4th IEEE International Conference on Digital Ecosystems and Technologies, 2010.

[13] Holland MM, Parsons TJ. Mitochondrial DNA sequence analysis – validation and use for forensic casework. Forensic Sci Rev 1999; 11:21-50.

[14] Budowle B, Wilson MR, DiZinno JA, Stauffer C, Fasano MA, Holland MM, et al. Mitochondrial DNA regions HVI and HVII population data. Forensic Sci Int 1999; 103:23-35.