International Journal on Computational Modelling Applications
Vol. 02, Iss. 03, S. No. 02, pp.9--20, Sept 2025
ISSN (Online): 3048-8516

Received: 26 June 2025, Accepted: 07 July 2025, Published: 02 August 2025
Digital Object Identifier: https://doi.org/10.63503/j.ijcma.2025.157

Review Article

Pattern Matching Algorithms for DNA Sequence Analysis and Disease Detection

Neelofar Sohi[1]*, Shaheena Sohi[2]

[1] Department of Computer Science & Engineering, Punjabi University, Patiala, Punjab, India

[2] Universal College of Pharmacy, Lalru, Dera Bassi, Punjab, India

neelofarsohi7@gmail.com[1], shaheenasohi@gmail.com[2]

*Corresponding author: Neelofar Sohi, neelofarsohi7@gmail.com

ABSTRACT

The focus of research in Genetics is upon analysis of massive amounts of information generated by various studies and research endeavours in the area. DNA sequence analysis paves way for understanding of factors responsible for diseases and leads to disease detection. Pattern matching algorithms have been explored for DNA sequence analysis and disease detection. In this study, some of the prominent pattern matching algorithms are comprehensively reviewed and presented as a potential technique for understanding and analysis of DNA sequences for disease detection.

Keywords: *Pattern Matching, DNA Sequence Analysis, Disease Detection.*

1. Introduction

Bioinformatics is an interdisciplinary area that combines biology, computer science and statistics [1-2]. It aims at developing tools and algorithms for the understanding and analysis of biological data. Currently, massive amounts of biological data are produced by various projects, and Bioinformatics aims at understanding and analysing this data to reveal valuable information and knowledge. This analysis impacts various areas such as molecular modelling, genome-wide analysis, comparative genomics, gene expression studies, genome sequencing, drug discovery, crop improvement, gene therapy, evolutionary studies, veterinary science, climate change studies, molecular medicine and so on. Hence, bioinformatics plays a significant role in healthcare.

1.1 Background & Motivation

For detection of diseases, the focus of research shifted from linkage analysis to association studies. The primary basis of association studies is to identify markers where markers are the sequence variations present on the DNA sequence which might cause diseases. Various pioneering association studies have been carried out such as Gambano et al., 2000; Cardon and Bell, 2001; Lohmueller et al, 2003; Manolio, 2010; Hollenbach et al, 2012; Alonso et al., 2021; Uffelmann et al., 2021; Shao et al., 2024; Yang et al., 2025 [3-11]. There are two different types of variations viz. Genetic variations and somatic variations. Genetic variations are inherited from parent to the offspring whereas somatic variations include mutations caused in a person's DNA during their lifetime. Hence, detecting the presence of markers on the DNA sequence (or gene) of a person enables to understand the likelihood or susceptibility of that person to develop that disease over their lifetime. Therefore, analysis of DNA sequences is highly important for detection of diseases. Pattern matching algorithms aim to find out

whether a particular pattern is present in a text and then finding its exact location in the text [12-13]. Genetic mapping is identification of genes underlying diseases. Linkage analysis is the most fundamental form of genetic mapping introduced by Sturtevant for fruit fly in 1913. It suggests that variants (or markers) having association with a particular trait must be lying nearby to each other on the genome. In humans, linkage analysis was introduced around 1980s. And the idea was employed for Huntington disease analysis in 1983. His approach was applicable to genetic and mendelian diseases where mendelian diseases are the ones where single gene is involved in causing the disease. A Genome wide approach to association studies was introduced in 1990s. This approach involves developing a catalog of human genetic variations and then testing the association of variations with diseases. Various pioneering studies were conducted based on Genome wide Association Approach since 2006 till now [14].

2. Pattern Matching Algorithms for DNA Sequence Analysis

In this study, some of the prominent pattern-matching algorithms are studied for their application in DNA sequence analysis. In this section, six string matching algorithms are discussed, giving their working principle, advantages, shortcomings, and complexities.

2.1 Brute Force Matching Algorithm

This is also termed as the Naive algorithm, known to be the simplest one for the string matching problem. The task is to match and find the pattern of length 'm' in the text of length 'n' [15]. The technique of this algorithm is discussed below:

- The first character of the pattern is matched against the first character of the text.
- If the characters match, then the second characters are compared. If the second character matches, then the third and so on until the entire pattern is found to occur at that location. Then starting location of the pattern inside the text is returned.
- If the characters at the first position do not match, then the first character of the pattern is moved to match it with the second character of the text, and the same procedure is repeated for matching.
-

❖ **Problems with Brute Force Matching Algorithm**

- Slow Execution Time: This algorithm follows linear search technique hence is very slow and consumes a lot of time. Its worst case complexity is O (m*n) as it performs character by character comparison.
- Poor performance for long sequences: This algorithm works well for short sequences but its complexity increases when the sequence becomes longer.
-

❖ **Advantages of Brute Force Matching Algorithm**

- Easy to understand & implement
- Simple technique
- Wide applicability

2.2 Boyer Moore Algorithm

This algorithm learns from the character comparisons it performs and then skips those alignments which cannot possibly be fruitful [16]. Hence, in comparison to Brute force matching algorithm, Boyer Moore performs less number of comparisons. This algorithm learns from the study of pattern that which

comparisons cannot yield matches and skips them. Alignment is performed from left to right whereas individual characters are compared from right to left.

❖ Approaches in Boyer Moore:

- Bad Character Rule
- Good Suffix Rule

Here, Idea is to apply the rule which skips more alignments hence best of the two heuristics (approaches) is applied at any given step.

- Bad Character Rule: Character of the text not matching with currently compared character of the pattern is called the Bad Character.This rule checks the following conditions: if there occurs a mismatch, we skip alignments or shift the pattern until
  - ➢ Mismatch becomes a match: We look for the mismatched character of text while moving towards left in the pattern or
  - ➢ P moves past the bad character if no match found while going towards left
- Good Suffix Rule: It keeps the matches and does not let them turn into mismatches. This rule checks the following conditions:
  - ➢ If a substring 't' is found in the text that occurs in the pattern while moving towards left, then move the pattern to match that substring of P with T or
  - ➢ Look for the next match of characters in P and T while moving left in the pattern

The working of Boyer Moore is explained with the help of an example in figure 1.

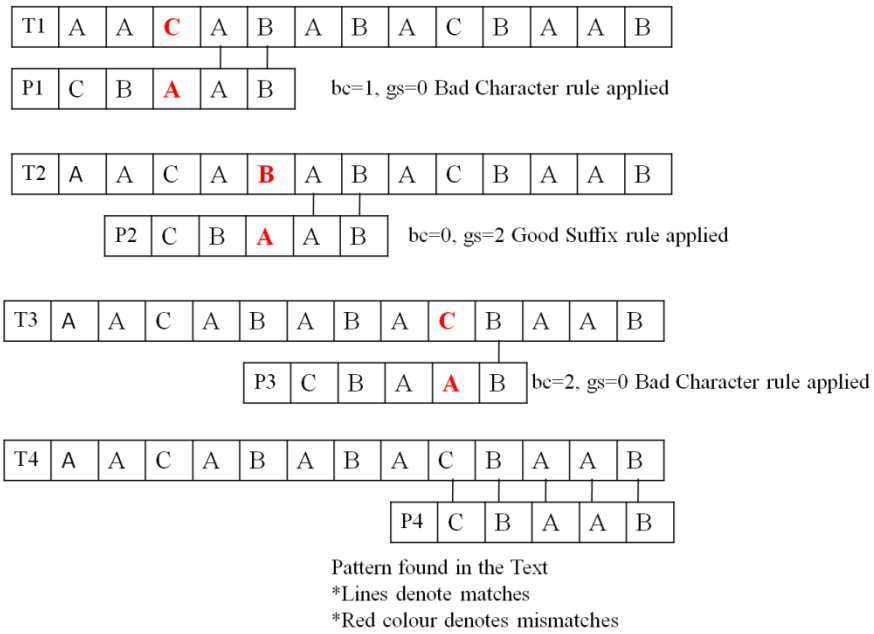

Fig. 1: Example of Boyer Moore Algorithm

❖ **Advantages of Boyer Moore Algorithm**

- Known to be the most efficient algorithm
- Works well for long pattern and moderate sized text

- Best case complexity= O(n)

2.3 Rabin Karp Algorithm

Rabin Karp algorithm is used for searching and matching of patterns in a text. Originally, it is a string matching algorithm developed by Michael O. Rabin and Richard M. Karp in 1987 [17]. It is used in string matching to find a particular pattern in the text. It calculates a hash function. For Text $T=\{t1,t2\ldots\ldots t_n\}$ and Pattern $P= \{p1,p2\ldots..p_m\}$, goal is to find out occurrence and location of the pattern in the text. It is achieved by first comparing whole pattern with substring of the text with length equal to length of the pattern. It is stated that pattern is found in the text with shift 's' where shift is the number of characters after which pattern is found.

❖ **Two Components of Rabin Karp Algorithm**

There are two main components of Rabin Karp algorithm:

- Hashing: It is the basic technique used in Rabin Karp algorithm to reduce the number of comparisons required. Also, it converts the string into a numeric value. Hash function is calculated and hash value for the pattern and the sub-sequence of the text can be compared for finding the match.
- Rolling Hash Function: In order to calculate the Hash Function of next sub-string, previous character's hash value is used. This is termed as 'Rolling Hash Function'.

❖ **Selection of Hash function in Rabin Karp**

It is basically a tool to convert a large value to a small value. Here, in this case, it converts a string into a numeric value. The output value is termed as hash value [18].

For Text $T=\{t1,t2\ldots\ldots t_n\}$

& Pattern $P= \{p1,p2\ldots..p_m\}$

Hash function for Rabin Karp can be defined as:

- P [1]+ P [2]+ P [3] or
- $P[1] * d^{m-1} + P[2] *d^{m-2} + P[3] *d^{m-3}$ or
- $h \leftarrow (P[i]* d^{m-1}) \bmod K$

  where d is radix base (d is chosen as 4 when there four elements in the alphabet set); K is prime number

❖ **Working of Rabin Karp Algorithm**

The step by step procedure of working of Rabin Karp algorithm is discussed below:

- Take the Reference nucleotide sequence, $T=\{t1,t2\ldots\ldots t_n\}$ with length 'n' to be matched
- Convert each character in the Alphabet set {A,G,C,T} into a numeric value by having codes assigned to each of these characters viz. 0 for A, 1 for G, 2 for C and 3 for T
- Take the Query nucleotide sequence, $P= \{p1,p2\ldots..p_m\}$ with length 'm' to be matched against reference sequence
- For this Query string, Convert each character in the Alphabet set {A,G,C,T} into a numeric value by having codes assigned to each of these characters viz. 0 for A, 1 for G, 2 for C and 3 for T
- Calculate the Hash Value for the chosen Reference as well as Query sequence using this formula: $h \leftarrow (\text{Sequence } [i]* d^{m-1}) \bmod K$

- Choose a prime number ; K. We choose K=29 because if we look at ASCII codes for A, G, C & T, highest code is for T and lowest for A. The difference between the two is 32. Hence we choose a number nearest to that.
- 'd' is the radix base. Since we have four characters {A, G, C, T} in the alphabet set, 'd' is taken equal to 4.

- Now, calculated hash values for Reference and Query sequences are compared to each other.
  - If the two values match that means this part of sequence has no SNP then next subsequence in the main reference & Query sequence is compared using Rabin Karp.
  - If the values do not match that means this part of compared sequences is the 'Candidate part' for SNPs. Candidate part is taken and character by character comparison is done using naive Brute Force String matching algorithm.

❖ **Advantages of Rabin Karp Algorithm**

- Reduced number of comparisons: In case of Rabin Karp Algorithm, the two sequences do not have to be compared on character by character basis as in case of Brute Force Matching algorithm. The hash values are calculated for both reference and query sequences and are compared to check for match. If the two values match that means this part of sequence has no SNP then next subsequence in the main reference & Query sequence is compared using Rabin Karp. If the values do not match that means this part of compared sequences is the 'Candidate part' for SNPs. Candidate part is taken and character by character comparison is done using naive Brute Force String matching algorithm. So, individual characters are compared only when needed.
- Constant Pre-processing Time: In this algorithm, there is one subtraction, one addition and one multiplication involved at one location for computing hash function therefore the pre-processing time is constant i.e. O(1).
- There is a role of intuition in selection of prime number 'k'

❖ **Problems in Rabin Karp Algorithm**

- Improper selection of Modulus function and prime number creates problem
- In some situations, same hash value is generated by two equal strings. So, H(P) i.e. hash value of the pattern and $H(T_i)$ i.e. hash value of the text subsequence with length equal to pattern (m) is calculated. These two hash values are compared to establish whether there is a match or not. Sometimes, hash value comes out to be equal but actually pattern and text subsequence are not equal. This situation is called 'spurious hit' [19]. Then characters are to be individually compared to ascertain that it is a 'valid hit' and not' spurious hit'. And if character by character comparison is done then complexity increases.
- Larger number of comparisons
- Increased complexity
- Algorithm is slow
- Takes up extra space O(P)

2.4 Knuth-Morris-Pratt (KMP) Algorithm

Knuth-Morris-Pratt (KMP) algorithm is used to search for a pattern with length 'm' from a text of length 'n'. The most salient feature of this algorithm is that it reduces number of comparisons and hence the time complexity by utilising the information gained from previous comparisons. In comparison to brute

force matching (naive) algorithm, it performs a reduced number of comparisons [20]. The average case complexity of Brute Force Matching algorithm is O(mn) whereas that of KMP algorithm is O(m+n).

- It utilizes the information obtained from previously carried out comparisons.
- The failure function (f) is computed, which describes how much of the previous comparison can be reused.

❖ **Pre-processing in KMP algorithm**

There is a need to pre-process the pattern and construct an integer array lps[ ]. The array lps[] gives the number of characters which can be skipped as they have been already compared. It tries to find which characters are repeating in the given pattern to reduce number of comparisons required.

- In KMP algorithm, pre-processing of pattern is done and auxiliary integer array lps[ ] is constructed of size 'm' i.e. size of the pattern.
- The term 'lps' stands for longest proper prefix. For example, possible prefixes of 'abc' are 'a', 'ab' and 'abc' taken from left to right and suffixes are 'c', 'bc' and 'abc' which are taken from right to left.
- For every sub-pattern pat [0.....i] where i=0 to m-1, lps[i] is an array storing length of the maximum matching prefix that is also a suffix of this sub-pattern.
  The working of KMP algorithm is explained below with the help of an example (figure 2).

Pattern with length; m=5

Index (j)	1	2	3	4	5
Pattern	a	b	a	b	d
lps	0	0	1	2	0

Text with length; n=15

Index (i)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Text	a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

Fig. 2: Example of KMP Algorithm

The step-by-step procedure of working of the KMP algorithm is presented in Table 1 with the help of the given example.

Table 1: Example of Step by step working of KMP Algorithm

Step 1. Begin at index; i=1 for text & j=0; do j+1=1 At i=1, T(i)=a At j+1=1, P(j)= a, hence Match	Step 2. Move to i=2, T(i)=b and j=1; j+1=2, P(j)= b, hence Match
Step 3. Move to i=3, T(i)=a and At j=2; j+1=3, P(j)= a, hence Match	Step 4. Move to i=4, T(i)=b and At j=3; j+1=4, P(j)= b, hence Match

Step 5. Move to i=5, T(i)=c and At j=4; j+1=5, P(j)= d, hence Mismatch	Step 6: Do not backtrack "i"; only "j" backtracked • Value of lps at j=4 is 2 So, move j=2; j+1=3, P(j)=a • Continue with i=5 T(i)=c Hence Mismatch
Step 7: Do not backtrack "i"; only "j" backtracked • Value of lps at j=2 is 0 So, move j=0; j+1=1, P(j)=a • Continue with i=5 T(i)=c Hence Mismatch	Step 8: "j" already at 0; cannot be backtracked; hence move i to 6, T(6)=a and j=0; j+1=1, P(j)= a, hence Match
Step 9: Move to i=7, T(i)=b and j=1; j+1=2, P(j)= b, hence Match	Step 10: Move to i=8, T(i)=c and j=2; j+1=3, P(j)= a, hence Mismatch
Step 11: Do not backtrack "i"; only "j" backtracked • Value of lps at j=2 is 0 So, move j=0; j+1=1, P(j)=a • Continue with i=8 T(8)=c Hence Mismatch	Step 12: "j" already at 0; cannot be backtracked; hence move i to 9, T(9)=a and j=0; j+1=1, P(j)= a, hence Match
Step 13: Move to i=10, T(i)=b and j=1; j+1=2, P(j)= b, hence Match	Step 14: Move to i=11, T(i)=a and j=2; j+1=3, P(j)= a, hence Match
Step 15: Move to i=12, T(i)=b and j=3; j+1=4, P(j)= b, hence Match	Step 16: Move to i=13, T(i)=a and j=4; j+1=5, P(j)= d, hence Mismatch
Step 17: Do not backtrack "i"; only "j" backtracked • Value of lps at j=4 is 2 So, move j=2; j+1=3, P(3)=a • Continue with i=13 T(13)=a Hence Match	Step 18: Move to i=14, T(i)=b and j=3; j+1=4, P(j)= b, hence Match
Step 18: Move to i=15, T(i)=d and j=4; j+1=5, P(j)= d, hence Match	Therefore, pattern found at index: 11 to 15 in the text

2.5 Trie String Matching Algorithm

Trie is a tree-based data structure, also called a Digital tree or a Prefix tree. It is for storing of strings to enable efficient retrieval. Tries enable prefix queries for the retrieval of information. Prefix queries look for the longest prefix of a given string that matches a prefix of some string in the trie [21].

If S is the set of strings, a Trie for S is an ordered tree T where:

- Every edge in T is labeled with a character belonging to $\sum$.
- Order of edges coming out from some internal node is decided by $\sum$.

- A path from root node to any node in T denotes a prefix in ∑ which is equal to concatenation of characters while traversing the path
- A character associated with an edge can be represented at a child node under it

❖ **Operations in Trie**

- Insertion operation: This operation is to insert a string 'X' into a set of strings 'S' as described in an example in figure 3.

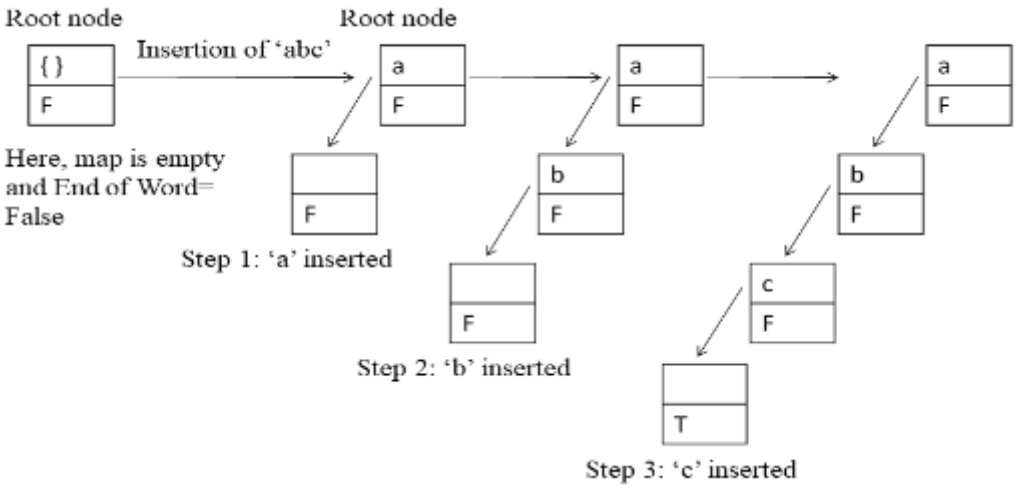

Fig 3: Example of Insertion operation in Trie

- Deletion operation: This operation is to remove a string 'X' from a set of strings 'S'.
- Search operation: This operation returns all those strings in set 'S' which have the longest prefix of 'X'. Searching in the given set of strings for a particular string can be of two types:

  ➢ Prefix based search: This search operation just requires the prefix to be present on the trie structure as shown in figure 4.

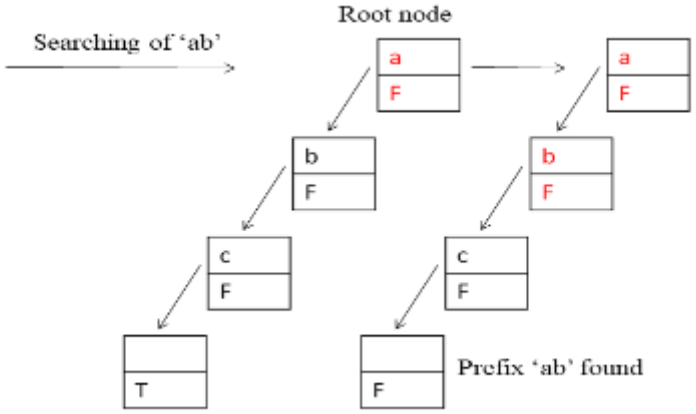

Fig. 4: Example of Prefix based search in Trie

- Whole word based search: This search operation requires the whole word to be present on the trie structure while traversing down from the root node. It must have its end of word=True (as shown in figure 5).

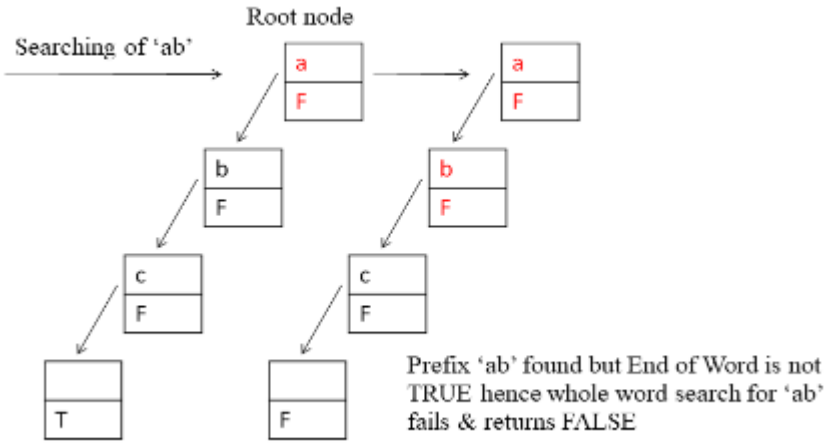

Fig. 5: Example of Word-based search in Trie

❖ **Advantages of Trie**

- Less Memory: If there are repeating characters 'ab' and 'abcd', they will be represented on same path hence less memory is used.
- Reduced Complexity: O(m) where m is length of string to be inserted or searched for.

2.6 Finite Automata

The idea is to construct finite automata (finite state automation) for searching a pattern from the given text. The approach followed consists of three steps. The working of this technique is described below in detail with the help of an example. Pattern and the text are given in figure 6.

Pattern	a	b	a	b	a	c	a				
Index	1	2	3	4	5	6	7	8	9	10	11
Text	a	b	a	b	a	b	a	c	a	b	a

Fig. 6: Pattern and text taken in the example

Step I. Build Finite Automata for the given pattern as described in step by step manner in table 2:

Take Alphabet set; ∑= {a, b, c}

Q: Finite set of states

$q_o \in Q$: Initial State

F Ç Q: Final State

δ : Transition from one state to another

Tuple: $(Q, \sum, \delta, q_o, F)$ defines the FA

Now, the question is how many states are to be formed. As there are 7 characters in pattern (P), we take States in Finite Automata (FA) from 0 to 7.

Table 2: Step I of building the Finite Automata

Step 1: Check for a,b,c • 'a' is in the P so Make transition for 'a' from 0 to 1	Step 2: • Now from 'a' check aa,ab,ac aa: p=(a) & s=(a); p=s; len=1; so make transition from 1 to 1 for next 'a' • Check ab: p≠s hence no transition • Check ac: p≠s hence no transition
Step 3: Check : aba, abb, abc • aba: Found in P so make transition from 1 to 2 • abb: p≠s hence no transition • abc: p≠s hence no transition	Step 4: check abaa,abab,abac • check abaa: p=(a,ab,aba) & s=(a,aa,baa) p=s; len=1; hence transition from 3 to 1 • abab: Found in P so make transition from 3 to 4 • abac: p≠s hence no transition
Step 5: check ababa, ababb, ababc • ababa: Found in P so make transition from 4 to 5 • ababb: p≠s hence no transition • ababc: p≠s hence no transition	Step 6: check ababaa, ababab, ababac • ababaa: p=(a,ab,aba,abab) & s=(a,aa,baa,abaa) p=s; len=1; hence transition from 5 to 1 • ababab: p=(a,ab,aba,abab,ababa) & s=(b,ab,bab,abab,babab) p=s for 'ab' and 'abab' take len=4 hence transition from 5 to 4 • ababac: Found in P so make transition from 5 to 6
Step 7: check ababaca, ababacb, ababacc • ababaca: Found in P so make transition from 6 to 7 • ababacb: p≠s hence no transition • ababacc: p≠s hence no transition	Step 8: check ababacaa, ababacab, ababacac • ababacaa: p=(a,ab,aba,abab,ababa,ababac,ababaca) & s=(a,aa,caa,acaa,bacaa,abacaa,babacaa) p=s for 'a' with len=1 hence transition from 7 to 1 • ababacab: p=(a,ab,aba,abab,ababa,ababac,ababaca) & s=(b,ab,cab,acab,bacab,abacab,babacab) p=s for 'ab' with len=2 hence transition from 7 to 2 • ababacab: p≠s hence no transition

Step II. The first step gives us the Finite Automata for given pattern. Second step is to convert this FA into transition table given in table 3. Here, 0 denotes 'no transition'.

Table 3: Transition Table constructed from FA in Step II

State/Alphabet Set Elements	a	b	c	Pattern (P)
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

Step III. Third step is to match the pattern with text using transition table. Here, we read the elements of text one by one as described in table 4.

Table 4: Step III of matching Pattern with Text using Finite Automata

Step 1: Start checking from initial state= 0 (0,a) → 1 States encountered=0	Step 2: (0,a) → 1 States encountered=0,1
Step 3: (1,b) → 2 States encountered=0,1,2	Step 4: (2,a) → 3 States encountered=0,1,2,3
Step 5: (3,b) → 4 States encountered=0,1,2,3,4	Step 6: (4,a) → 5 States encountered=0,1,2,3,4,5
Step 7: (5,b) → 4 States encountered=0,1,2,3,4,5,4	Step 8: (4,a) → 5 States encountered=0,1,2,3,4,5,4,5
Step 9: (5,c) → 6 States encountered=0,1,2,3,4,5,4,5,6	Step 10: (6,a) → 7 States encountered=0,1,2,3,4,5,4,5,6,7

Step IV. The condition used in the code is "if (q==m) then "pattern occurs with a shift of (i-m)". Here, q=7 is reached and the value of P; Pattern is initially set to 7. Hence, the condition is met, which implies that the pattern occurs with a shift of (i-m). Here, the value of index i in Text where 7 occurs is 9. Hence, (i-m) gives 2, which means the pattern occurs with a shift of 2, i.e., the pattern occurs at index 3 of the Text. The complexity of Finite Automata for pattern matching is O (n), where n is the length of the text. Here, every character is processed only once.

3. Conclusions and Future Scope

In this study, several prominent pattern-matching algorithms, such as Brute Force, Boyer-Moore, Rabin-Karp, Knuth-Morris-Pratt (KMP), Trie and Finite Automata, have been explored for DNA sequence analysis and disease detection. Strengths and shortcomings of these algorithms are discussed along with their working principle in detail. The paradigm shift of research in genetics from linkage studies to Genome Wide Association Studies is discussed. Also, the importance of understanding sequence variations and their association with diseases is discussed. It is established that analysis of sequence variations can lead to disease detection. Pattern-matching algorithms hold the potential for understanding and analysis of DNA sequences for disease detection. They can be applied for the detection of SNPs and other sequence variations. There is a scope to improve the performance of these algorithms for such applications.

Acknowledgements

We are thankful to the Department of Computer Science & Engineering, Punjabi University, Patiala, Punjab and Universal College of Pharmacy, Lalru, Dera Bassi, Punjab for providing the necessary labs and other infrastructure for carrying out this research work.

Funding source

"None."

Conflict of Interest

The authors declare no conflict of interest.

References

[1] N.M. Luscombe, D. Greenbaum D and M. Gerstein, "What is bioinformatics? An Introduction and overview", *Yearbook of Medical Informatics*, pp. 83-100.

[2] J. Pevsner, *Bioinformatics and functional genomics*, 3rd ed. John Wiley & Sons Inc, Chichester, 2015.
[3] G. Gambano, F. Anglani and A. D'Angelo, "Association studies of genetic polymorphisms and complex disease", *The Lancet (British edition)*, vol. 355, no. 9200, pp. 308-311, 2000.
[4] L.R. Cardon and J.I. Bell, "Association Study Designs for Complex Diseases", *Nature Reviews (Genetics)*, vol. 2, pp. 91-99, 2001.
[5] K. Lohmueller, C. Pearce, M. Pike et al., "Meta-analysis of genetic association studies supports a contribution of common variants to susceptibility to common disease", *Nature Genetics,* vol. 33, pp. 177–182, 2003.
[6] T.A. Manolio, "Genomewide Association Studies and Assessment of the Risk of Disease", *The New England Journal of Medicine,* vol. 363, no. 2, pp.166-176, 2010.
[7] J.A. Hollenbach, S.J. Mack, G. Thomson and P.A. Gourraud, "Analytical methods for disease association studies with immunogenetic data", *Methods in Molecular Biology*, vol. 882, pp. 245-266, 2012.
[8] L. Alonso, I. Moran, C. Salvaro and D. Torrents D, "In Search of Complex Disease Risk through Genome Wide Association Studies", *Mathematics,* vol. *9*, no. 23, pp. 3083, 2021.
[9] E. Uffelmann, Q.Q. Huang, N.S. Munung et al., "Genome-wide association studies", *Nature Review Methods Primers,* vol. 1, 59 , 2021.
[10] M. Shao, K. Chen, S. Zhang, M. Tian, Y. Shen, C. Cao and N. Gu, "Multiome-wide Association Studies: Novel Approaches for Understanding Diseases", *Genomics, Proteomics & Bioinformatics* , vol. 22, qzae077 , 2024.
[11] L. Yang, M.C. Sadler and R.B. Altman, "Genetic association studies using disease liabilities from deep neural networks", *The American Journal of Human Genetics*, vol. 112, no. 3, pp. 675-692, 2025.
[12] A.P. Gope and R.N. Behera, "A Novel Pattern Matching Algorithm in Genome Sequence Analysis", *International Journal of Computer Science and Information Technologies,* vol. 5, no. 4, pp. 5450-5457, 2014.
[13] P.Neamatollahi, M. Hadi and M. Naghibzadeh, "Simple and Efficient Pattern Matching Algorithms for Biological Sequences", *IEEE Access*, vol. 8, pp. 23838-2384, 2020.
[14] D. Altschuler, M.J. Daly and E.S. Lander, "Genetic Mapping in Human Disease", *Science*, vol. 322, no.5903, pp. 881-888, 2008.
[15] E. D'Souza, B. Shalini Pai and S. Vijayakumar, "Comparative Analysis on Efficiency of Single String Pattern Matching Algorithms", *International Journal of Latest Trends in Engineering and Technology SACAIM*, vol. 2016, pp.221-225, 2016.
[16] R.S. Boyer and J.S. Moore, "A Fast String Searching Algorithm", *Communications of the ACM* , vol. 20, no.10, pp. 762-772,1977.
[17] R.M. Karp and M.O. Rabin, "Efficient randomized pattern-matching algorithms", *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249-260, 1987.
[18] M. Shabaz and N. Kumari, "Advance-Rabin Karp Algorithm for String Matching", *International Journal of Current Research,* vol. 9, no. 9, pp. 57572-57574, 2017.
[19] Sunita, R. Malik and M. Gulia M, "Rabin-Karp Algorithm with Hashing a String Matching tool", *International Journal of Advanced Research in Computer Science and Software,* 2014.
[20] D.E. Knuth, J.H. Morris and V.R. Pratt, "Fast Pattern Matching in Strings", *SIAM Journal on Computing*, vol. 6, no. 2, pp. 323-350, 1977.
[21] Ahmed, "The Role of Trie Data Structure in String Processing", Research Project, Benha University, Shoubra, 2019.