*Abstract and Introduction summary*

* To Process biological data we need many resources and a lot of time, which are costly.
* The problem is biological data is very big and complex, we need efficient way to analyze it.
* We will analyze it with pattern matching, we will search for a specific pattern for disease or mutation.
* The paper will study different kinds of algorithms for pattern matching, complexity and performance.
* Keyword search and matching are techniques to discover patterns inside specific strings. Algorithms for matching, are used to discover matches between patterns and input strings.
* We will consider the pattern is p and text is t, pattern is shorter than t, and we need to find all occurrence of p in t.
* There are two main approaches for string matching, first is exact matching, for instance: Smith-Waterman (SW); Needleman Wunsch (NW); Boyer Moore Horspool (BMH); Dynamic Programming; Knuth Morris Pratt (KMP). Second approach, is approximate matching, also known by Fuzzy string searching, for instance: Rabin Karp; Brute Force.
* In this paper is analyzed the similarity measures on Protein, DNA and RNA, using for that effect, different types of string matching algorithms.

***Related works***

* In pattern recognition problems it is essential to measure distance or similarity. Given the following example :

String T : A C C T C G A G T

Pattern p: \_ \_ \_ \_ C G A ­\_ \_

Pattern P can be matched in String T by adding four empty spaces before the pattern and two after.

* Authors in (Yeh and Cheng 2008), use Levenshtein distance applied to images and videos to determine feature vectors.
* In (Amir et al. 2004), the authors propose a new distance for string matching, similar to Levenshtein distance, with K-Mismatches on the given string. This proposed approach was implemented with Message Passing Interface (MPI), and proved to be useful to establish similarity between strings.
* Authors in (Knuth et al. 1977), proposed an algorithm for pattern matching in strings, with running time proportional to the sum of the length of the strings. This traditional algorithm is now known as KMP string matching algorithm.
* Other classical string pattern matching algorithm was proposed in (Hussain et al. 2013), named Bidirectional Exact Pattern Matching (BDEPM). This algorithm introduces the idea to compare strings using pointers in simultaneous, one from the left other from the right.
* In (Alsmadi and Nuser 2012), they evaluated two algorithms for DNA string comparison in terms of accuracy and performance. The Longest Common Substring (LCS) algorithm, and Longest Common Sub-Sequence (LCSS) algorithms. In the following example, the highlighted letters, CTCT, in the sequences is LCSS of the specified sequences.

Aho-Corasick and CommentZ-Walter algorithms (Vidanagamachchi et al. 2012) are two types of multiple patterns matching algorithms, authors in (Vidanagamachchi et al. 2012) implemented these two algorithms and worked with peptide sequences to study their accuracy and execution time. Results show that Aho-Corasick performs better than the CommentZ-Walter algorithm.

*In this section, we analyze the proposed some matching algorithms*

1. Rabin Karp Algorithm

Rabin Karp (RK) algorithm is used to look for similarities in two sequences, proteins, i.e. a high sequence similarity means significant structural or functional similarity. RK utilizes a hash function to make the string searching faster. The hash value for the pattern is calculated, used then to compare with sub-sequences of the text. When the hash values are different, RK algorithm estimates the hash for the next matching sequence of characters. If hash values are equal, RK algorithm uses brute-force to compare sequence pattern with the text. RK algorithm efficiency is based on the computation of hash values of text sub-strings.

Rolling hash functions analyze sub-strings as a number, the base of this functions, usually is a large prime number. For instance, considering the sub-string “AC” and the base 1011, the hash would be 65×10111+67×10110=6578265×10111+67×10110=65782, where “A” is 65 and “C” is 67 in the ASCII table.

Lets consider a *M-character* sequence with *M-digit* number, base *b*, where *b* is the number of letters in the alphabet, the text mapped into sub-sequence t[1...i+M−1]t[1...i+M−1].

X(i)=t[i]×bM−1+t[i+1]×bM−2+...+t[i+M−1]X(i)=t[i]×bM−1+t[i+1]×bM−2+...+t[i+M−1]

(8)

Equation [8](https://link.springer.com/chapter/10.1007/978-3-030-45385-5_41#Equ8) is used to compute sub-sequences of specific sequences.

Given x(i)x(i) it is possible to compute x(i+1)x(i+1) for the next sub-sequence t[i+1...i+M]t[i+1...i+M] as in Eq. [9](https://link.springer.com/chapter/10.1007/978-3-030-45385-5_41#Equ9)

X(i+1)=t [i+1]×bM−1+t[i+2]×bM−2+...+t[i+M]X(i+1)=t [i+1]×bM−1+t[i+2]×bM−2+...+t[i+M]

(9)

Equation [9](https://link.springer.com/chapter/10.1007/978-3-030-45385-5_41#Equ9) shows how to discover the next sub-sequence for the predecessor.

h(i)=((t[i]×bM−1modq)+(t[i+1]×bM−2modq)+…+(t [i+M−1]modq))h(i)=((t[i]×bM−1modq)+(t[i+1]×bM−2modq)+…+(t [i+M−1]modq))

(10)

Equation [10](https://link.springer.com/chapter/10.1007/978-3-030-45385-5_41#Equ10) is used to calculate hash values for the sub sequences. Where:

* x(i)x(i) represents the text sub-sequence t[i]t[i];
* h(i)h(i) represents the hash function;
* *b* represents the base of the string;
* *q* is the prime number;

2. Levenshtein Distance

Levenshtein distance between two sequences is defined as the minimum number of substitutions required to transform one sequence into another. It allowed to compare sequences with different lengths, by considering the operations insertion, deletion and substituting characters.

Definition 2

*(Levenshtein Distance).* Consider two sequences A=(a1,…,an1)A=(a1,…,an1) and B=(b1,…, bn2)B=(b1,…, bn2) over the alphabet ΣΣ. The Levenshtein distance between two sequences *A* and *B* of length n1n1 and n2n2, respectively, is given by δ lev (n1,n2) δlev(n1,n2) where

δlev(i,j)={ max (i,j)Δ(i,j) if  min (i,j)=0, otherwise δlev(i,j)={ max (i,j) if  min (i,j)=0,Δ(i,j) otherwise

(2)

where

Δ(i,j)= min ⎧⎩⎨⎪⎪δlev(i−1,j)+1δlev(i,j−1+1δlev(i−1,j−1)+s(ai,bj)Δ(i,j)= min {δlev(i−1,j)+1δlev(i,j−1+1δlev(i−1,j−1)+s(ai,bj)

and s(ai,bj)=1s(ai,bj)=1 when ai≠bjai≠bj and 0 otherwise. The first element in the minimum function corresponds to deletion from *A* to *B*, the second to insertion and the third to match or mismatch (substitution).

For instance, assuming that the Levenshtein distance between “CCGTCG” and “CGGTTGA” is three, then it is not possible to transform one into the other with less than three edits:

1. [[https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Fige_HTML.gif](https://static-content.springer.com/image/chp:10.1007/978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Fige_HTML.gif)](https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Fige_HTML.gif) (replace ‘C’ for ‘G’);

1. [[https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figf_HTML.gif](https://static-content.springer.com/image/chp:10.1007/978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figf_HTML.gif)](https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figf_HTML.gif) (replace ‘T’ for ‘C’);

3. [[https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figg_HTML.gif](https://static-content.springer.com/image/chp:10.1007/978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figg_HTML.gif)](https://static-content.springer.com/image/chp%3A10.1007%2F978-3-030-45385-5_41/MediaObjects/492924_1_En_41_Figg_HTML.gif) (remove ‘A’ from the end).