Developing an Approximate Heuristic Algorithm based on Existing Exact String Matching Algorithms

A Report

Submitted to

The School of Engineering and Computing

National University

In Partial Fulfillment of the Requirements

for the Degree of Master of Science in Computer Science

By

Emerald Trejo

August 2018

A report submitted to the School of Engineering and Computing, National University, in partial fulfillment of the requirements for the Degree of Master of Science in Computer Science by:

_____
Emerald Trejo

The report is hereby approved by:

_____
Pradip Peter Dey, Ph.D.
Professor, School of Engineering and Computing
National University

_____
Bhaskar Raj Sinha, Ph.D.
Professor, School of Engineering and Computing
National University

_____
Date

Table of Contents

Abstract

The human body is a unique work of art. Each individual that exists today consists of billions of genomic letters that make up their distinct patterns of DNA. Genomic sequencing is a dynamic and growing field of research that explores the current human genome patterns that allow for detection and prevention of diseases, cancer and other genetic mutations. Exact string matching algorithms allow researchers to find matching patterns in a long sequence of genetic data. This report presents an extensive overview of some well known exact string matching algorithms as well as a heuristic approximate algorithm. Some algorithms that will be used to compare and devise the heuristic algorithm are Boyer Moore, Brute Force, Knuth-Morris-Pratt (KMP) and Rabin-Karp algorithms. Approximate heuristic algorithms, also known as "fuzzy string" is a technique of finding strings that match a pattern approximately rather than exactly. By comparing each algorithms run time and efficiency, it allows for opportunities to unveil more pathways for developing an approximate heuristic algorithm. This report will be based on experiments conducted on genomic data are randomly generated on a local database platform.

Acknowledgment

I would first like to thank my thesis advisors Pradip Peter Dey, Ph.D. and Bhaskar Raj Sinha, Ph.D. of the School of Engineering and Computing at National University. Their constant support and direction of my thesis allowed me the opportunity to grow stronger as a graduate student.

I would also like to thank Alrieza Farahani, Ph.D. for having an open door and being my mentor.

I would also like to acknowledge John Cicero, Ph.D Vice Provost of National University and I am gratefully indebted to him for his valuable teachings towards this thesis.

Finally, I must express my very profound gratitude to my husband Hugo Trejo for providing me with unfailing support and continuous encouragement throughout my years of study and through the process of researching and writing this thesis.

This accomplishment would not have been possible without them. Thank you.

Emerald C. Trejo

1. Introduction:

Genomic sequence matching is a complex and dynamic field of research that is constantly evolving. A little under 150 years ago the presence of nuclein, now known as DNA, was published by Friedrich Miescher and not much later was Albrecht Kossel awarded the Nobel Peace prize for the discovery of the five nucleotide bases adenine, cytosine, guanine, thymine and uracil in 1910 (Wilgar 2016). In today's millenium we are examining genomic sequences to uncover patterns that lead to genetic mutations, diseases and illnesses.

Every human being possesses a unique set of genes which are made up of DNA. DNA is the carrier of that organism's genetic code. This "code" is what is used to define the construction of "a set of sequences which define what proteins to build within the organism" (Gope 2014). Simply stated, genes hold the DNA that are the instructions for making proteins and a genetic disease is defined as any disease that is caused by an abnormality in an individual's genome. There is a variant of genetic disorders. Some that are inherited from the parents while others being caused by mutations in a pre-existing gene or group of genes (Gope 2014).

In Bioinformatics the most well known application is DNA sequence detection (Gope 2014). The ultimate goal of bioinformatics is to "uncover the wealth of Biological information hidden in the mass of data and obtains a clearer insight into the fundamental biology of organisms" (Gope 2014). It initially starts with awareness that a sequence may be the cause for a disease and bioinformatics allows for opportunities in tracing sequences of the DNA to allow for further insight on the "number of occurrences of the sequence defines the intensity of the disease" (Gope 2014).

Considering BioInformatics is the study of sequences that can help discover mutations and diseases, the capability of how these patterns can be studied and discovered is unveiled through pattern matching of the DNA genome sequence. The most efficient way to unveil the sequence is through the use of various algorithms and the searching of a pattern held within the string sequence of DNA and testing whether or not the pattern exists or not. DNA is composed of billions of genomic letters. Today there are multiple types of algorithms to discover sequences in this large mass of data. Since DNA is so unique and comprises what makes a living organism function, punctuality is very critical in discovering the exact match of the sequence that places the living human beings life at risk whether is be for discovery or recovery. In this discovery of sequences, "There are two types of string matching Exact string matching and Approximate string matching" (Kapil 2014). Exact string matching is to have the search to be done on the "exact occurrence of the pattern" while approximate string matching "allows inaccurate searching acceptance" (Kapil 2014). The inaccurate searching acceptance allows for close to but not exact matches and opens opportunities to new discovery of mutated sequences. Classified within the string matching searches are 4 approaches for the execution of the algorithm: left to right, right to left, specific order matching and no order matching (Kapil 2014). These types will be seen in the explanation of the following algorithms Boyer Moore, Brute Force, Knuth-Morris-Pratt (KMP) and Rabin-Karp as seen in Section 4 of this paper.

The thesis that will be discussed in this paper will be featured around developing a heuristic approximate algorithm that uses pattern recognition at a smaller scale level. The heuristic approximate matching algorithm will be based off the already existing exact string

matching algorithm Boyer Moore and catering to a percent matched rates of the pattern being compared to the string that is randomly generated.

2. Requirements and Feasibility

2a. What is available today:

In San Diego, California there is a Biotech company by the name of Ilumina which is conducting research that would allow individuals to pay $1000 to have scientists piece together the complete sequence of an individual's genetic code. Since the code consists of 3 billion letters of DNA (adenine, cytosine, guanine, thymine) that make up the human genome, heavy machinery and storage is used to make the research possible (Fikes 2017). Illumina is making breakthroughs in research through their MovaSeq technology. To produce an insight of the importance of this testing, the technology of The NovaSeq 5000 and 6000 systems are priced at $850,000 and $985,000, respectively (Fikes 2017). The way that Ilumina processes the DNA is by taking the millions of tiny DNA fragments which only consist of about 250 letters in length, then “pieced together computationally to assemble the human genome jigsaw puzzle” (Watson 2017) The heuristic approximate algorithm that will be devised in this thesis will be based off this logic of having a “genome jigsaw puzzle”. The heuristic algorithm will be considered a more versatile puzzle piece where a match can be considered close but not an exact matching puzzle piece in the large genome jigsaw puzzle. Through this proposal there is hope that the more diversified search for patterns will open more opportunities for research and cures in the genomic community of research.. “Illumina’s high-volume, industrial-capacity sequencing machines are key to ever-growing research efforts to look for genetic patterns — and variants — among

millions of people that could unlock mysteries about the causes of various diseases" (Fikes 2017).

2b. Real Life Application:

The use of algorithms is mute as most individuals are unaware that their daily activities are being performed by algorithms. With daily text messages, emails that are written to clients or even web searches, the implementation of spell checks is being used. Spell checks are a form of an approximate string matching algorithm as "If any such pattern occurs then it shows the occurrence by reaching to its final states" (Kapil 2014). Another protective device from errors that are constantly used are spam filters. Spam filters and spam detection systems are another form of string matching algorithms as they serve as first line of defense for data based companies. Spam intrusion can cost companies thousands or even up to millions of dollars of loss data. Spam filters are constantly evolving and the filters need to be just as up to date as the hackers who create the viruses. The filters and detection systems "search suspected signature patterns in the contents of the email" (Kapil 2014) but are solely based on keywords. As hackers are becoming more clever, there needs to be a stronger development of algorithms to increase cyber security. Related to our spell check is the constant web searches that are performed on search engines. This string matching algorithm "organize[s] the required text" based on keywords (Kapil 2014). Algorithms play a crucial yet hidden role in our society as it not only protects against dangerous hackers, but from horrible and embarrassing spelling mistakes.

2c. Database Requirements

The database platform that will be used to run the heuristic approximate algorithm will be Microsoft Visual Studio. There will be no pre requirements for the project as the exact string matching algorithms Boyer Moore, Brute Force, Knuth-Morris-Pratt (KMP) and Rabin-Karp algorithms have been proven to run accurately. The requirements for the working heuristic algorithm will be based off an existing exact matching algorithm Boyer Moore algorithm. The heuristic algorithm must possess the same and equivalent code that already exists with a few added lines of code to perform the duties of the heuristic algorithm. The database requirements will be different than what exists for biotech companies today. After discussing with the vice provost of the department of my graduate school, he suggested that instead of using genetic strings that are available through medical schools and other sources, that instead the program use a random generator of strings that are of size 10,000 characters and allow a 5 character pattern to be read against the random generated string. A relational database, which is a collective set of multiple data sets organized by records and columns, will be useful for this thesis as the string that will be read will be only be 1000 characters in length compared to 3 billion that is what is usually read for large biotech companies.(Technopedia 2018). Once a working prototype has been established, the size of the string can start to expand to larger character lengths. As the processing of 3 billion characters takes immense computer power, and because actual genes represent only a small percentage of the DNA in a genome, many diagnostic tests keep costs down by focusing only on that part, which is called the exome (Fikes 2017). For real life application, traditional databases (row-based and even column-based) are not suitable for DNA sequencing as they are more suitable for storage of highly structured, fixed, limited sets and their

relations. A regular relational database is limited by the thousands rather than the 3 billion letters contained within a single genome. (Gavrilov 2015).

2d. Feasibility

The heuristic algorithm will include a search for the suffix and mid section of the code to match with the string sequence which will be the last 3 letters of the sequence. It will be marked as a match when there is a 50% or more match of the letters of the pattern to the string. The program will be running code analysis for the 5 letter pattern that will be entered by the user.. If there is a full match of the 3 last letters of the sequence, the code will continue to check towards the prefix if there is a full match. If there is no match past the 3 letters, the pattern will be shifted 5 characters to the right for analysis of the next set of letters. The requirements for the user will be prior knowledge of the pattern that they wished to be matched.

2e. Why is this important for today

Developing a heuristic approximate matching algorithm is important for today because it can allow for further developed research on increasing efficiency and speed for matching the pattern against the string sequence of genomic characters. Developing an approximate algorithm allows more insight on similar patterns that could have been overlooked with an exact matching string pattern. An exact matching string algorithm only finds exact matches from a pattern within a string where an approximate matching algorithm can have the capability to find similar matches or close to exact matching algorithms. This particular field of research is important as it has the capacity to save lives as “more and more patients stand to benefit from what is often

referred to as genomic or precision medicine, including cancer patients, critically ill newborns and children otherwise subjected to diagnostic odysseys"(Watson 2017). Speed is an important factor for this field of research as "being able to keep up with the mutations will allow doctors to better guide therapy, such as by discovering when cancer cells have become resistant to certain drugs (Fikes 2017). The proposed heuristic algorithm is a prototype to future endeavors of discovery in genetic mutations in the BioInformatics industry.

3. Design

3a. Operational Scenario Template

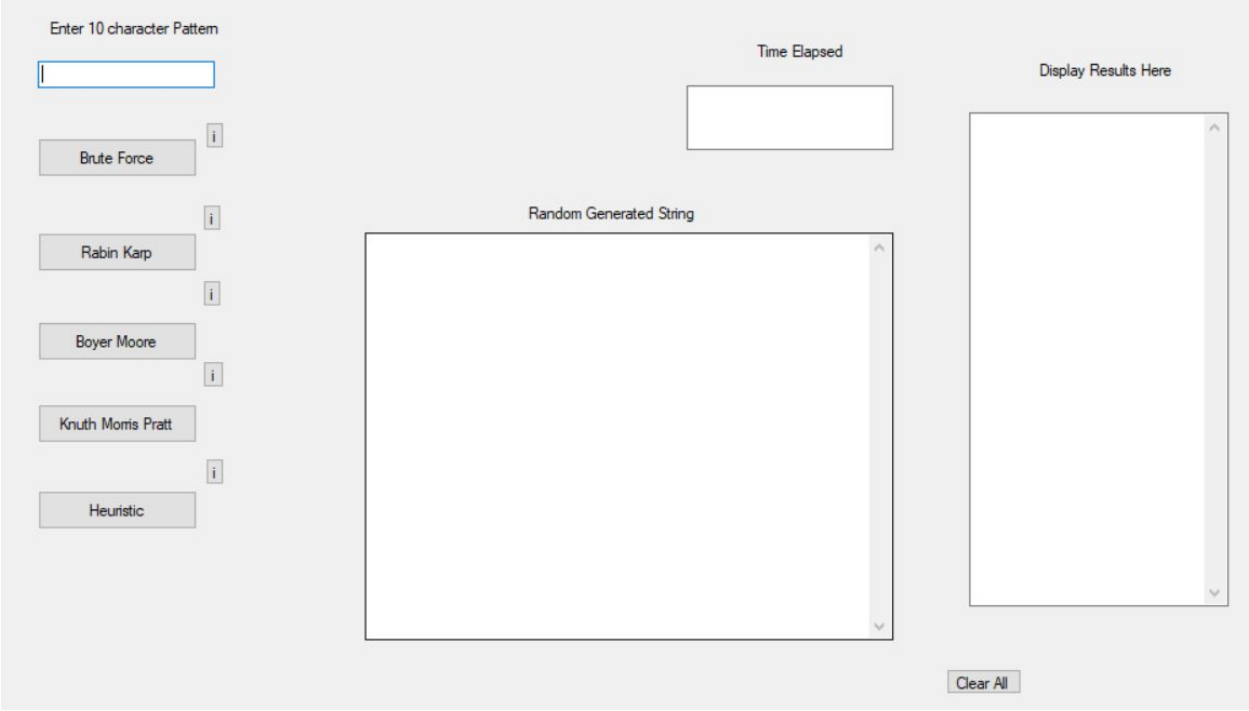
Student(s)	Emerald Trejo	Date	May 19, 2018
Program/Project	FileManager	Program #	CSC686-6
Instructor	Sinha, Dey	Language	C#

Scenario # 1		User Objective: Select Algorithm to Process Pattern Matching Algorithm	
Scenario Objective: Output Results of Matches of the pattern against the genomic string			
Source:	Step	Action:	Comments
user	1	Start Program	
program	2	Display Main Menu Request Selection	
user	3	Enter values of pattern	Proper value
program	4	Awaiting response for designated action to perform	
user	5	Select action to be performed	Proper action
program	6	Perform action	

program	7	Display Results to screen	
user	8	Select Clear Button	
program	9	Results are cleared from screen, awaiting next pattern and action to be performed	

3b. User Interface

Screenshot below reflects the home page interface that the user will be prompted upon. The home page is subject to change in color, font, and any further added features that will be necessary.



3c. Test Specifications:

Step	Action	Expected System Response	Pass/Fail	Comment
1	Enter ACTGA	Hold Text until algorithm is selected		
2	Brute Force Algorithm is selected	System runs algorithm with text entered and displays results		
3	User presses Clear All Button	Information of the algorithm which includes how many matches has been cleared from the screen		
4	Boyer Moore Algorithm is Selected	System runs algorithm with text entered and displays results of current algorithm and summary of Brute Force Algorithm		
5	User presses Clear All Button	Information of the algorithm which includes how many matches has been cleared from the screen		
6	“i” is clicked	Displays information about algorithm in a new popup window		
7	Click ‘x’ on right hand corner	User is exited out of display of algorithm		
8	Heuristic Algorithm is clicked	System runs algorithm with text entered and displays results of current algorithm and summary of both Brute Force Algorithm and Boyer moore		
9	Click Exit	Program is closed		

4. Implementation of Algorithms

4a. Brute Force

The most basic and conventional string matching method is Brute Force also known as "Naive approach". This algorithm is done through initializing each part of the string with the pattern and a "Comparison [of] character by character of the text" (Gou 2014) is performed and returns all valid shifts found. The pattern is not further passed in comparison if the first index of the pattern does not match with the string sequence. For instance, if the second and third letter matched a 4 character pattern against the main string sequence but the first character of the pattern did not, the pattern would not make it the second character as the Brute Force algorithm distinctly looks for exact matchings..

If there is not a copy of the whole pattern in the first m characters of the text, we look if theres a copy of the pattern starting at the second character of the text (Benuskova 2017). If there is a match then we continue in the same way along the text and count the number of matches (Benuskova 2017).

The outer " for" loop calculates how many times the pattern can be shifted which can be 1 to max amount of times. While the inner "for" loop calculates the amount of times there are comparisons performed. When the line "if (pattern [j] != s[i +j])" is false, the checking of the pattern against the string will come out. If it remained true, the pattern will continue to check. With a full successful match, a pattern and location of pattern is returned and displayed. Below reflects the logic on how the Pattern, Text and shifts are performed.

P[0...m-1] Pattern
T[s...s+m-1] Text
S Σ { 0,..., n-m +1} shifts

Figure 1:

```
public void Search() // s is string sequence, pattern is what is inputted from user
{

    //brute force algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int i;
    int j;
    for (i = 0; i <= n - m; i++)
    {
        for (j = 0; j < m; j++)
        {
            if (pattern[j] != s[i + j])
                break;
        }
        if (j == m)
        {
            Console.WriteLine("Pattern found at:{0}", i);
            Console.WriteLine(pattern);
        }

    }

}
```

One problem with the Brute Force Algorithm is the effectiveness. The running time of Brute force is equal to its matching time since there is no pre processing (Gope 2014). Pre processing will be further discussed in other exact string matching algorithms that have a significant faster processing time.

4b. Boyer Moore Algorithm

In 1977 Robert Boyer and J Strother Moore proposed their algorithm (Kapil 2014) named the Boyer Moore Algorithm. What is distinguishable about their algorithm is that they devised

their algorithm against the norm of reading patterns from left to right like most western cultures read text. Through an innovative approach did they comprise an algorithm that read the pattern against the string from right to left. The pattern does however originate and shift from the left furthest position of the string like all algorithms do in string matching. This performance of reading right to left (Gou 2014) allows to skip more characters than the other algorithms as the last character of the pattern is checked last in other algorithms. “If the first character matched of the text is not contained in the pattern P[0...m-1] we can skip m characters immediately (Gou 2014). If there isn't an occurrence at all, the pattern is slided to the right by its own length without missing a match. There are two significant rules in Boyer Moore that make it one of the most efficient and quicker algorithms, the bad character shift rule and the good suffix rule. With the bad character shift rule, the shift can be pre calculated for every letter and stored in a table. This unique table is called a bad character shift rule (Benuskova 2017). The bad character rule also refers to upon mismatch there is a skip in alignment until mismatch becomes a match or moves past mismatched character. The good suffix rule refers to the pre computing of all the terminal substrings of the pattern (Benuskova 2017). It's speed derives from the fact that it can determine all occurrences of pattern within text without examining too many characters in text (Benuskova 2017). Alternatively the good suffix rule have a worst case running time if the pattern does not appear in the text at all (Gope 2014). Figure 2 displays the necessary code to reflect how the search is performed.

Figure 2:

```
Program                                                    Search()
public void Search() // s is string sequence, pattern is what is inputted from user
{
    var watch = System.Diagnostics.Stopwatch.StartNew();
    List<int> retVal = new List<int>();

    // boyer moore algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int[] badChar = new int[256];

    BadCharHeuristic(pattern, m, ref badChar);

    int i = 0;

    while (i <= (n - m))
    {
        int j = m - 1;

        while (j >= 0 && pattern[j] == s[i + j])
            --j;

        if (j < 0)
        {
            retVal.Add(i);
            i += (i + m < n) ? m - badChar[s[i + m]] : 1;
        }

        else
        {
            i += Math.Max(1, j - badChar[s[i + j]]);
        }

    }
```

Boyer Moore algorithm is considered “sublinear” in the sense that it looks at fewer characters than it passes (Benuskova 2017). Boyer Moore algorithm is considered the benchmark for the practical string searching (Benuskova 2017).

4c. Knuth Morris Pratt Algorithm

The Knuth Morris Pratt (KMP) Algorithm was conceived in 1970 by Donald Knuth and Vaughan Pratt and independently by James Morris and the three published it jointly in 1977 (Wikipedia 2018). Just as the other algorithms initially start from the far left of the string, KMP also reads from left to right. When a mismatch occurs the pattern itself is used to determine where to jump to the next meaningful position to continue. (Benuskova 2017). When a mismatch occurs, the algorithm will use the information given by a specific table, obtained by a preprocessing pattern, to avoid re-examine of the characters that have been previously checked thus limiting the number of comparison required (Gou 2014). KMP has two parts, a searching part which consists to find the valid shifts in the text, and a preprocessing part which consists to preprocess the pattern (Gou 2014). The success link will take us to the next node in the chain, and the failure link will take us back to a previous node based on the word pattern (Cao 2014). See figure 3 for the searching and preprocessing sections of the code. Figure 3:

```
public void preKMP(string pattern, ref int[] lpsArray)
{
    int M = pattern.Length;
    int len = 0;
    lpsArray[0] = 0;
    int i = 1;

    while (i < M)
    {
        if (pattern[i] == pattern[len])
        {
            len++;
            lpsArray[i] = len;
            i++;
        }
        else
        {
            if (len == 0)
            {
                lpsArray[i] = 0;
                i++;
            }
            else
            {
                len = lpsArray[len - 1];
            }
        }
    }
}
```

```
public List<int> KMP(string s, string pattern) // s is string sequence, pattern is what i
{

    //kmp algorithm is executed here

    int m = pattern.Length;
    int n = s.Length;
    int[] lpsArray = new int[m];
    List<int> matchedIndex = new List<int>();

    if (n < m) return matchedIndex;
    if (n == m && s == pattern) return matchedIndex;
    if (m == 0) return matchedIndex;

    preKMP(pattern, ref lpsArray);

    int i = 0, j = 0;
    while (i < n)
    {
        if (s[i] == pattern[j])
        {
            i++;
            j++;
        }

        // match found at i-j
        if (j == m)
        {
            matchedIndex.Add(i - j);

            j = lpsArray[j - 1];

            screenValue = ("Pattern found at:" + i).ToString() + " " + pattern;

            mydisplay1.Text += screenValue + Environment.NewLine;
        }
        else if (i < n && s[i] != pattern[j])


        {
            if (j != 0)
            {
                j = lpsArray[j - 1];
            }
            else
            {
                i++;
            }
        }
    }

    return matchedIndex;

}
```

To calculate the positions for the pattern as to how much a pattern need to shift itself so that the corresponding characters of text match with it. The table is called the next table or sometimes the “faiture function ” (Gope 2014). Failure function is defined as the length of the

longest prefix of P that is a suffix of P[i...j] (Mohammed 2018). Failure links do not get a new character but reuse the last character fetched (Cao 2014). In order to build the KMP automaton (or so called KMP "failure function") we have to initialize an integer array lpsArray[]. The indexes (from 0 to m - the length of the pattern) represent the numbers under which the consecutive prefixes of the pattern are listed in our "list of prefixes" above. Under each index is a "pointer" that identifies the index of the longest proper suffix which is at the same time a prefix of the given string (or in other words [lpsArrayi], is the index of the next best partial match for the string under index i). (Top Coder 2016). The automaton consists of the initialized array lpsArray[] ("internal rules) and a pointer to the index of the prefix of the pattern that is the best (largest) partial match that ends at the current position in the text ("current state). (Top Coder 2016). Goal of the table is to allow the algorithm not to match any character of String more than once (Wikipedia 2018).

Searching for encoded patterns in encoded text raises the problems of false matches: finding an occurrence of the encoded pattern in the encoded text which does not correspond to an occurrence of the pattern in the original text due to crossing codeword boundaries (Shapira 2006).

4d. Rabin Karp Algorithm

Richard Karp and Michael Rabin developed the Rabin Karp algorithm in 1987 which "computes a hash function for the pattern and then look[s] for a match by using the same hash function for each possible substring of the same length in the text" (Gou 2014). Similar to the Knuth Morris Pratt algorithm, Rabin Karp also uses preprocessing technique before the search

operation (Gou 2014). The pre processing comprises of discovering the hash that is associated with the part of the string that the pattern is attempting to compare to. To better explain what a "hash function" is, it is a conversion of every string into a numerical value using a set table of numerical values or for instance the ASCII value of characters. This algorithms success is based off the fact that if two strings are equal their hash values are also equal (Benuskova 2017). The key to performance is the efficient computation of hash values of the successive substrings of the text (Benuskova 2017).

In Figure 5 below, the code displays the computation of the hash function h(x) for the pattern P[0...m-1] and then look for a match by using the same hash function for each substring of length m-1 of the text (Gou 2014). It first divides the pattern with a predefined prime number 'q' to calculate the modular of the pattern P. Then it tests the first m characters (m=[P]) from text T to compute remainder of m characters from Text T. (Gope 2014).

Rabin Karp is an inferior algorithm for single pattern searching compared to Boyer Moore algorithm because of its slow worst case behavior and is the best choice algorithm for multiple pattern search (Benuskova).

```
public void Search() // s is string sequence, pattern is what is inputted from user
{

    //Rabin karp algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int i;
    int j;

    int p = 0;
    int h = 1;
    int t = 0;

    for (i = 0; i < m - 1; i++)
        h = (h * d) % Q;
    for(i=0; i < m; i ++)
    {
        p = (d * p + pattern[i]) % Q;
        t = (d * t + s[i]) % Q;
    }
    for (i = 0; i <= n-m; i++)
    {
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (s[i + j] != pattern[j])
                    break;
```

Figure 5:

```
            }
            if (j == m)
            {
                Console.WriteLine("Pattern found at:{0}", i);
                Console.WriteLine(pattern);
            }
        }
        if (i < n-m)
        {
            t = (d * (t - s[i] * h) + s[i + m]) % Q;
            if (t < 0)
                t = (t + Q);
        }
    }


}
```

4d.i Related Article:

With further research on Rabin Karp, an interesting paper appeared that related to the use of Rabin Karp in searching for matched sequences but in the light of palm printing research. Biometric palm printing image pattern matching uses Rabin-Karp Palm-Printing Pattern Matching (RPPM) method which is constructed with the help of double hashing method with different angle position (Kanchana 2015). In order to accurately match, the concept of "bit pattern matching" includes features such as principle lines, ridges, minutiae points, and texture is performed simultaneously in RPPM method. (Kanchana 2015). These minutiae points are based on position, path and direction of ridges which are unique to each individual (Kanchana 2015). Double hashing technique is applied to RPPM method with the objective of reducing the collision rate (Kanchana 2015). "If the palm print features are not matched with the single hash value, then the other hashing is carried out with different hash key" (Kanchana 2015).

4e. Heuristic Algorithm

In the development of a heuristic algorithm for this particular report, the approximate heuristic algorithm is derived from the existing exact string matching algorithm of Boyer Moore. The use of the Boyer Moore as a platform for the Approximate Heuristic algorithm is due to the efficiency of the reading the pattern against the string from right to left rather than left to right. Boyer Moore algorithm is able to save time and memory by eliminating patterns that do not match at the end of the algorithm rather than reading through the entire algorithm and failing to match at the end, such as Brute Force which has the worst time complexity of all the existing exact string matching algorithms. In computer science, fuzzy logic can be viewed as a form of

logic used in some expert systems and (other artificial intelligence) applications in which variables can have degrees of truth or falsehood (Cork 2002). The approximate heuristic algorithm produces results that are 50% or great matched with the string from the pattern. The importance of this break down of matching is that it can present alternative views of the genetic data should there exist other mutations that would have been blindly overlooked because they were not an exact match. The outcome of an event is expressed in term of probabilistic confidence rather than in terms of certainty (Cork 2002). The algorithm is to provide a gateway to other alternative data for genomic sequence processing. Therefore in addition to examining the exact occurrence of an event, we can also focus on the probability of that event to occur or not occur. Figure 6:

```
public void Heuristic() // s is string sequence, pattern is what is inputted from user
{

    List<int> retVal = new List<int>();
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int[] badChar = new int[256];

    BadCharHeuristic(pattern, m, ref badChar);

    int i = 0;

    while (i <= (n - m))
    {
        int j = m - 1;

        while (j >= 0 && pattern[j] == s[i + j])
            --j;

        if (j < 0)
        {
            retVal.Add(i);
            i += (i + m < n) ? m - badChar[s[i + m]] : 1;
        }
        else
        {
            i += Math.Max(1, j - badChar[s[i + j]]);
        }

    }
```

```
private void heuristic_Click(object sender, EventArgs e)
{
    pattern = enter1.Text.ToUpper();
    RandomGenerator();

    // 50% match
    pattern1 = pattern.Substring(0, (int)(pattern.Length * 0.5));
    screenValue = ("50% Match: ").ToString() + " " + pattern1;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    // 60% match from start
    pattern2 = pattern.Substring(0, (int)(pattern.Length * .6));
    screenValue = ("60% Match: ").ToString() + " " + pattern2;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    // 70% match from start
    pattern3 = pattern.Substring(0, (int)(pattern.Length * .7));
    screenValue = ("70% Match: ").ToString() + " " + pattern3;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    // 80% match from start
    pattern4 = pattern.Substring(0, (int)(pattern.Length * .8));
    screenValue = ("80% Match: ").ToString() + " " + pattern4;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();


    // 90% match from start
    pattern5 = pattern.Substring(0, (int)(pattern.Length * .9));
    screenValue = ("90% Match: ").ToString() + " " + pattern5;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    //100%match from start
    screenValue = ("100% Match: ").ToString() + " " + pattern;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    BoyerMoore();

    DisplayTime();
}
```

4f. Big O Analysis

The "Big O analysis" compares how quickly the runtime of algorithm grows with respect to size of input. Big O analysis describes the efficiency of algorithms as the size of the input increases. The relation of Big O analysis is discussed further in relation to the algorithms that were used for the genomic sequence search process. The first algorithm is the Brute Force which has the time complexity of $O(M \times N)$, where generally M is a very small compared to N, (Gou 2014). A second algorithm that was mentioned is the Boyer Moore algorithm that has a runtime of O(N/M) and the runtime gets faster as the pattern gets longer (Benuskova 2017). The worst case running time for Boyer Moore is O(NxM). This would be exemplified where the pattern and the text are strings composed by sequences of one same character (Gou 2014). Taking into consideration the good suffix rule in has a worst case running time of O(m) provided that the pattern does not appear in the text (Gope 2014). A third string matching algorithm mentioned is Knuth Morris Pratt (KMP) which is considered a linear time algorithm (Gou 2014) which has the runtime of $O(N + M)$.

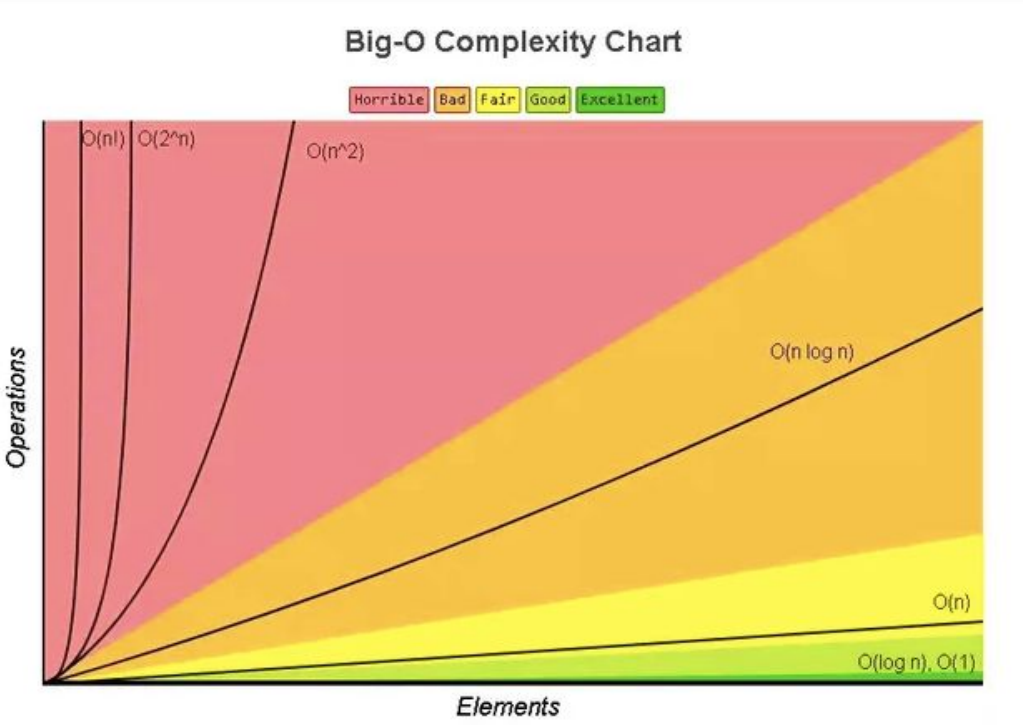
KMP has 2 parts, a searching part which consists to find the valid shifts in the text where the time complexity is O(N) obtained by comparison of the pattern and the shifts of the text, and a preprocessing part which consists to preprocess the pattern (Gou 2014). KMP complexity of the preprocessing part is O(M). The worst case running time of KMP is O(m+n) (Mohammed 2018). The Rabin Karp algorithm has the run time of O(M x (N-M +1)). Hashing of the pattern in the algorithm has the time complexity of O(M). Rabin Karp is the most efficient for multiple pattern matching (in addition to single pattern matching) because it has the unique advantage of

being able to find any one of k strings in O(n) time on average regardless of the magnitude of k. (Benuskova)

Discussed below are examples of how the Big O expresses an algorithm in terms of a function O(n). As the size of n increases, the running time increases at a predictable rate which can modeled as a function (Shu 2017):

1. **O(1) —** A constant function that always executes in the same amount of time regardless of the size of the input. (Shu 2017)
2. **O(logn) —** A logarithmic function that constantly reduces the input size by a fraction (usually by half as in binary search). (Shu 2017)
3. **O(n) —** A linear function where the runtime increases in proportion to the size of the input. (Shu 2017)
4. **O(nlogn) —** This function grows a little more rapidly than the linear function but a lot less rapidly than the quadratic function. In general, these algorithms combine an O(n) operation with an O(logn) operation. (Shu 2017)
5. **O(n^2) —** A quadratic function where the runtime increases exponentially by a factor of two as the size of the input increases. This function is most commonly found in nested loops. (Shu 2017)
6. **O(n^3) —** A cubic function where the runtime increases exponentially by a factor of three as the size of the input increases. A simple example is an algorithm that requires triple nested loops. (Shu 2017)

7. **O(2^n) —** An exponential function where the runtime increases exponentially by a factor of n as the size of the input increases. An example is using brute force to crack a password by generating all possible combinations. (Shu 2017).



5. Improvements of Project

Had time and resources permitted, this project could be a constant development. The strings that were generated for this project were minimal compared to the real genome of human beings which consists of billions of characters. Keeping in mind that the machines that perform these tests are millions of dollars to purchase and Microsoft Visual Studio is a free platform. Had time permitted, the project could have taken DNA information that is made public, and used to tests the efficiency and speed of the multiple algorithms including the proposed approximate

heuristic algorithm. Considering that this report was drafted with the sole focus on the evaluation and analysis of multiple algorithms in the permitted time, the extension of this report could have evaluated real life mutation patterns as well. A more interactive interface could be implemented with more time and resources. An updated version of the interface can include saved results of each algorithm that was performed rather than to have to clear the system of results.

6. Conclusion

String matching has a great involvement in everyday tasks whether this includes searching the web or discovering life changing genetic mutations and will continue to play a large role in real world problems. The use of string matching algorithms will continue to grow as they become more utilized in this technology base world. String matching algorithms have helped the field of BioInformatics grow more cost effective for cures and this will change millions of lives in the future and is creating for a greater tomorrow. The results that were provided through this dynamic processing is that Boyer Moore is the most time efficient and best performing algorithm. Brute Force is the worst algorithm as it has the worst execution time complexity. The approximate heuristic algorithm presented in this report is based off the best performing algorithm so that it can present an additional presentation of alternative data for research purposes in BioInformatics.

7. References

Allmer, J. (2016). Exact pattern matching: Adapting the Boyer-Moore algorithm for DNA searches. *PeerJ PrePrints,* PeerJ PrePrints, Feb 19, 2016.

Benuskova, Lubica. (2017). COSC 348: Computing for Bioinformatics Lecture 4: Exact string searching algorithms. August 27, 2017 https://pdfs.semanticscholar.org/presentation/3c89/56c70ec642b10468139769fa0438a394cfa8.pdf

Cao, Z., & Liu, L. (2014). A Fast String Matching Algorithm Based on Lowlight Characters in the Pattern.

Cork, D., & Toguem, A. (2002). Using Fuzzy Logic to Confirm the Integrity of a Pattern Recognition Algorithm for Long Genomic Sequences. Annals of the New York Academy of Sciences, 980(1), 32-40.

Gavrilov, Vasili. "What Is the Best Database System for Comparing DNA Data (DNA Sequencing)?" *Quota*, 9 Dec. 2015, www.quora.com/What-is-the-best-database-system-for-comparing-DNA-data-DNA-sequencing.

Gawrychowski, Paweł, Jeż, Artur, & Jeż, Łukasz. (2014). Validating the Knuth-Morris-Pratt Failure Function, Fast and Online. Theory of Computing Systems, 54(2), 337-372.

Gope, Ashish & Behera, Rabi. (2014). A Novel Pattern Matching Algorithm in Genome Sequence Analysis. International Journal of Computer Science and Information Technologies. Vol. 5 (4), 2014, 540-5457

Gou, Marc. (2014) Algorithms for String Matching. July 30, 2014 http://www.student.montefiore.ulg.ac.be/~s091678/files/OHJ2906_Project.pdf

Fikes, Bradley J. "New Machines Can Sequence Human Genome in One Hour, Illumina Announces." *Sandiegouniontribune.com*, 10 Jan. 2017, www.sandiegouniontribune.com/business/biotech/sd-me-illumina-novaseq-20170109-story.html.

Idury, R., & Schaffer, Alejandro A. (1994). Dynamic multiple pattern matching. ProQuest Dissertations Publishing.

Kanchana, S., & Balakrishnan, G. (2015). Palm-Print Pattern Matching Based on Features Using Rabin-Karp for Person Identification. The Scientific World Journal, 2015, 8.

Mohammed Farik, & ABM Shawkat Ali. (2015). Algorithm To Ensure And Enforce Brute-Force Attack-Resilient Password In Routers. International Journal of Scientific & Technology Research, 4(8), 184-188

Mohammed, Rashid. (2018). Knuth-Morris-Pratt Algorithm. http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/kuthMP.htm

Ni, B., & Leung, Kwong-Sak. (2011). Generalized Pattern Matching Applied to Genetic Analysis. ProQuest Dissertations Publishing.

Salmela, Leena, Tarhio, Jorma, & Kalsi, Petri. (2010). Approximate Boyer-Moore String Matching for Small Alphabets.(Report). *Algorithmica, 58*(3), 591.

Shapira, Dana, & Daptardar, Ajay. (2006). Adapting the Knuth-Morris-Pratt algorithm for pattern matching in Huffman encoded texts. Information Processing & Management, 42(2), 429.

Sharma, J., & Singh, M. (2015). CUDA based Rabin-Karp Pattern Matching for Deep Packet Inspection on a Multicore GPU. International Journal of Computer Network and Information Security, 7(10), 70-77.

Shu, R. (2017). “I am learning algorithms, what is the usage of Big O notation?” October 2, 2017 Retrieved from https://www.quora.com/I-am-learning-algorithms-what-is-the-usage-of-Big-O-notation

Siti Ramadhani. (2017). Sistem Pencegahan Plagiarism Tugas Akhir Menggunakan Algoritma Rabin-Karp (Studi Kasus: Sekolah Tinggi Teknik Payakumbuh). Digital Zone: Jurnal Teknologi Informasi Dan Komunikasi, 6(1), 44-52.

Soni, Kapil, Vyas, Rohit & Sinhal & Amit. (2014). Importance of String Matching in Real World Problems. International Journal of Engineering and Computer Science. Volume 3 Issue 6 June 6, 2014.

Techopedia.com. (2018). *What is a Relational Database (RDB)? - Definition from Techopedia*. [online] Available at: https://www.techopedia.com/definition/1234/relational-database-rdb [Accessed 21 May 2018].

TopCoder.com. (2016). Introduction to String Searching Algorithms. https://www.topcoder.com/community/data-science/data-science-tutorials/introduction-to-string-searching-algorithms

Waga, M., Akazaki, T., & Hasuo, I. (2016). A Boyer-Moore Type Algorithm for Timed Pattern Matching.

Watson, James D, et al. “The Future of DNA Sequencing Is in the Palm of Your Hand.” *Time*, Time, 12 Oct. 2017, time.com/4971220/future-dna-sequencing/.

Wilgar, Hannah. “Timeline: History of Genomics.” *Facts*, The Public Engagement Team at the Wellcome Genome Campus, 5 Feb. 2016, www.yourgenome.org/facts/timeline-history-of-genomics

Wikipedia.com. (2018) “Knuth–Morris–Pratt algorithm”. June 12, 2018. Retrieved from https://en.wikipedia.org/wiki/Knuth–Morris–Pratt_algorithm

Appendix A

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Final_Algo_Project
{
   public partial class Form1 : Form
   {

      private String s;
      private String pattern;
      private String  pattern1, pattern2, pattern3, pattern4, pattern5;
      private String screenValue;
      int Q = 100007;

      public const int d = 256;

      int[] f = new int[100];

      public Form1()
      {
         InitializeComponent();
      }

      private void Bruteforce_Click(object sender, EventArgs e)
      {
         pattern = enter1.Text.ToUpper(); ;
         RandomGenerator();
         Bruteforce();
         DisplayTime();
      }

      private void rabinkarpbutton_Click(object sender, EventArgs e)
      {
         pattern = enter1.Text.ToUpper(); ;
         RandomGenerator();
         RabinKarp();
         DisplayTime();
```

```
}

private void boyermoore_Click(object sender, EventArgs e)
{
   pattern = enter1.Text.ToUpper(); ;
   RandomGenerator();
   BoyerMoore();
   DisplayTime();
}

private void knuthmorrispratt_Click(object sender, EventArgs e)
{
   pattern = enter1.Text.ToUpper(); ;
   RandomGenerator();
   preKMP();
   KMP();
   DisplayTime();
}

private void heuristic_Click(object sender, EventArgs e)
{
   pattern = enter1.Text.ToUpper();
   RandomGenerator();

   // 50% match
   pattern1 = pattern.Substring(0, (int)(pattern.Length * 0.5));
   screenValue = ("50% Match: ").ToString() + " " + pattern1;
   mydisplay1.Text += screenValue + Environment.NewLine;
   RandomGenerator();
   Heuristic();

   // 60% match from start
   pattern2 = pattern.Substring(0, (int)(pattern.Length * .6));
   screenValue = ("60% Match: ").ToString() + " " + pattern2;
   mydisplay1.Text += screenValue + Environment.NewLine;
   RandomGenerator();
   Heuristic();

   // 70% match from start
   pattern3 = pattern.Substring(0, (int)(pattern.Length * .7));
   screenValue = ("70% Match: ").ToString() + " " + pattern3;
   mydisplay1.Text += screenValue + Environment.NewLine;
   RandomGenerator();
   Heuristic();

   // 80% match from start
   pattern4 = pattern.Substring(0, (int)(pattern.Length * .8));
   screenValue = ("80% Match: ").ToString() + " " + pattern4;
```

```
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    // 90% match from start
    pattern5 = pattern.Substring(0, (int)(pattern.Length * .9));
    screenValue = ("90% Match: ").ToString() + " " + pattern5;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    Heuristic();

    //100%match from start
    screenValue = ("100% Match: ").ToString() + " " + pattern;
    mydisplay1.Text += screenValue + Environment.NewLine;
    RandomGenerator();
    BoyerMoore();

    DisplayTime();
}

//information for brute force "i" icon
private void button1_Click(object sender, EventArgs e)
{
    bruteforceinfo information = new bruteforceinfo();
    information.Show();

}

private void rabinkarpinfo_Click(object sender, EventArgs e)
{
    rabinkarpinfo information = new rabinkarpinfo();
    information.Show();
}

private void boyermooreinfo_Click(object sender, EventArgs e)
{
    boyermoore information = new boyermoore();
    information.Show();
}

private void kmpinfo_Click(object sender, EventArgs e)
{
    knuthmorrispratt information = new knuthmorrispratt();
    information.Show();
}

private void heuristicinfo_Click(object sender, EventArgs e)
{
```

```
    heuristic information = new heuristic();
    information.Show();
}

public void RandomGenerator()
{
    Random geneSequence = new Random();  // The random number sequence


    for (int i = 0; i < 1000; i++)
    {
        int x = geneSequence.Next(1, 4);


        switch (x)
        {
            case 1:
                s = s + 'C';
                break;

            case 2:
                s = s + 'T';
                break;

            case 3:
                s = s + 'A';
                break;

            case 4:
                s = s + 'G';
                break;
        }

    }
    screenValue = s;

    randomstring.Text += screenValue + Environment.NewLine;
}

public void Bruteforce() // s is string sequence, pattern is what is inputted from user
{

    //brute force algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;
```

```
    int i;
    int j;
    for (i = 0; i <= n - m; i++)
    {
        for (j = 0; j < m; j++)
        {
            if (pattern[j] != s[i + j])
                break;
        }
        if (j == m)
        {
            //i want these results to be sent back to the main screen

            screenValue = ("Pattern found at:" + i).ToString() + " " + pattern;

            mydisplay1.Text += screenValue + Environment.NewLine;
        }

    }
}

public void RabinKarp() // s is string sequence, pattern is what is inputted from user
{

    //Rabin karp algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int i;
    int j;

    int p = 0;
    int h = 1;
    int t = 0;

    for (i = 0; i < m - 1; i++)
        h = (h * d) % Q;
    for (i = 0; i < m; i++)
    {
        p = (d * p + pattern[i]) % Q;
        t = (d * t + s[i]) % Q;
    }
    for (i = 0; i <= n - m; i++)
    {
```

```
        if (p == t)
        {
            for (j = 0; j < m; j++)
            {
                if (s[i + j] != pattern[j])
                    break;
            }
            if (j == m)
            {
                screenValue = ("Pattern found at:" + i).ToString() + " " + pattern;

                mydisplay1.Text += screenValue + Environment.NewLine;
            }
        }
        if (i < n - m)
        {
            t = (d * (t - s[i] * h) + s[i + m]) % Q;
            if (t < 0)
                t = (t + Q);
        }
    }


}

public void BoyerMoore() // s is string sequence, pattern is what is inputted from user
{

    List<int> retVal = new List<int>();

    // boyer moore algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int[] badChar = new int[256];

    BadCharHeuristic(pattern, m, ref badChar);

    int i = 0;

    while (i <= (n - m))
    {
        int j = m - 1;

        while (j >= 0 && pattern[j] == s[i + j])
```

```
            --j;

        if (j < 0)
        {
            retVal.Add(i);
            i += (i + m < n) ? m - badChar[s[i + m]] : 1;
        }

        else
        {
            i += Math.Max(1, j - badChar[s[i + j]]);
        }

    }


    foreach (var x in retVal)
    {
        screenValue = ("Pattern found at:" + x).ToString() + " " + pattern;
        mydisplay1.Text += screenValue + Environment.NewLine;
    }

}

private static void BadCharHeuristic(string str, int size, ref int[] badChar)
{
    int i;

    for (i = 0; i < 256; i++)
        badChar[i] = -1;

    for (i = 0; i < size; i++)
        badChar[(int)str[i]] = i;
}

private void clear_Click(object sender, EventArgs e)
{
    mydisplay1.Clear();
    randomstring.Clear();
    timebox.Clear();
}



public void preKMP()
{
    int m = pattern.Length;
    int k;
```

```
    f[0] = -1;
    for (int i = 1; i < m; i++)
    {
        k = f[i - 1];
        while (k >= 0)
        {
            if (pattern[k] == pattern[i - 1])
                break;
            else
                k = f[k];
        }
        f[i] = k + 1;
    }
}


public int KMP() // s is string sequence, pattern is what is inputted from user
{

    //kmp algorithm is executed here

    int m = pattern.Length;
    int n = s.Length;
    int[] f = new int[m];

    preKMP();
    int i = 0;
    int k = 0;

    while (i < n)
    {
        if (k == -1)
        {
            ++i;
            k = 0;
        }
        else if (s[i] == pattern[k])
        {
            i++;
            k++;
            if (k == m)
            {
                screenValue = ("Pattern found at:" + i).ToString() + " " + pattern;

                mydisplay1.Text += screenValue + Environment.NewLine;

                return 1;
            }
```

```
        }
        else
            k = f[k];

    }
    return 0;

}

public void Heuristic() // s is string sequence, pattern is what is inputted from user
{

    List<int> retVal = new List<int>();

    // boyer moore algorithm is executed here
    int n;
    n = s.Length;

    int m;
    m = pattern.Length;

    int[] badChar = new int[256];

    BadCharHeuristic(pattern, m, ref badChar);

    int i = 0;

    while (i <= (n - m))
    {
        int j = m - 1;

        while (j >= 0 && pattern[j] == s[i + j])
            --j;

        if (j < 0)
        {
            retVal.Add(i);
            i += (i + m < n) ? m - badChar[s[i + m]] : 1;
        }

        else
        {
            i += Math.Max(1, j - badChar[s[i + j]]);
        }

    }
```

```
        foreach (var x in retVal)
        {
           screenValue = ("Pattern found at:" + x).ToString() + " " + pattern;
           mydisplay1.Text += screenValue + Environment.NewLine;
        }


      }

      public void DisplayTime()

      {

         //creating a stopwatch instance

         Stopwatch stopWatch = new Stopwatch();

         //starting the stopwatch

         stopWatch.Start();

         //getting ellapsed ticks

         var ellapsedTicks = stopWatch.ElapsedTicks;

         //converting ellapsed ticks into microseconds by multiplying with 1000000 and dividing

         //with the frequency of the stopwatch

         var microSeconds = ellapsedTicks * (1000000L) / Stopwatch.Frequency;

        // Console.WriteLine("Time processed: {0} micro seconds", microSeconds);
         screenValue = ("Time processed: " + microSeconds + " micro seconds" ).ToString();
         timebox.Text += screenValue + Environment.NewLine;
      }

   }
}
```