



Al al-Bayt University  
Prince Hussein bin Abdullah College of Information Technology  
Computer Science Department

**An Algorithm for Finding Approximate Local Similarities in DNA  
Sequences**

By

Najah Methqal Ali ALshanableh

2009

# An Algorithm for Finding Approximate Local Similarities in DNA Sequences

By

Najah Methqal Ali ALshanableh

Supervisor: Dr. Mamoun Al-Rababaa

A Thesis Submitted to the  
Scientific Research and Graduate Faculty in partial fulfillment of the  
Requirements for the degree of Master of Science  
in Computer Science

Members of the Committee

Dr. Mamoun Al-Rababaa

Dr. Jehad Q. Alnihoud

Dr. Venus Samawi

Dr. Maryam Nuser

Approved

.....

.....

.....

.....

Al al-Bayt University

Mafraq, Jordan

2009

## Dedication

This thesis is dedicated to everyone who gave me love, friendship and support during my research.

## Acknowledgements

I would like to thank my supervisor Dr. Mamoun Al-Rababaa and Dr. Jehad Alkhalidy. They both gave me guidelines for doing a scientific research, and ensured the quality of my work. Also I would like to thank Dr. Wafa Elbjeirami, Director of Molecular Diagnostics and Immunogenetics in King Hussein Cancer Centre and all the staff who helped me in my research, thanks a lot.

I also want to thank all my friends who supported me over the period of my research. At last, I would like to thank my father, my mother, and all other members in my family for their generous and kind supports.

## List of contents

Subject	Page
Front Page	A
Dedication	B
Acknowledgment	C
List of contents	D
List of tables	F
List of figures	G
List of appendices	I
List of abbreviations	J
Abstract	K
<b>Chapter one : Introduction</b>	1
1.1 Scope of the Study	2
1.2. Aims and Objectives	2
1.3. Significance of the Study	2
1.4. Contributions	2
1.5 Thesis Outline	3
<b>Chapter two: Bioinformatics</b>	4
2.1 Bioinformatics	5
2.2 Deoxyribonucleic Acid (DNA)	6
2.3 Sequence Alignment	7
2.4 Approximate Local Similarities in DNA	10
2.4.1 History of the Problem	11
2.4.2 Formal Definition	11
2.4.3 Approximate String Matching	12
<b>Chapter Three: Literature Review</b>	16
3.1 Needleman-Wunsch algorithm	16
3.2 Smith-Waterman algorithm	17
3.3 FASTA algorithm	20
3.4 BLAST algorithm	21
3.5 PatternHunter	21

<b>Chapter Four : Methodology</b>	25
4.1 Heuristics	25
4.2 Scoring Matrices	26
4.3 Word length ( Seed )	27
4.4 String matching techniques	28
<b>Chapter Five: The proposed Algorithm</b>	34
5.1 Algorithm Description	34
<b>Chapter Six: AFALS-N Software</b>	41
6.1 Software Development Model	41
6.2 Implementation	45
6.3 User Interface Screens	48
<b>Chapter Seven : Results and Discussion</b>	53
7.1 Test environment	53
7.2 Sensitivity analysis	54
7.3 Execution Time Evaluation	56
7.3 Comparison with PatternHunter	57
<b>Chapter Eight: Conclusion and Future Work</b>	59
8.1 Conclusion	59
8.2 Future work	59
References	60
المخلص	63
Appendix A: Sample of Test Data	64
Appendix B: Implementation and Screens	70

## List of tables

<b>Table</b>	<b>Page</b>
Table 3.1: PatternHunter compared to Blastn	22
Table 7.1 :Organisms compared	53
Table 7.2 : Execution times for sequences of size ranging from 1 kBP to 3MBP	56
Table 7.3 : PatternHunter vs AFALS-N	57
Table 7.4 : PatternHunter vs AFALS-N(word size 11)	58

## List of figures

Figure	Page
Figure 2.1: Bioinformatics	5
Figure 2.2: DNA Components	6
Figure 2.3: DNA double helix	7
Figure 2.4: DNA sequencing	7
Figure 2.5: DNA sequencing	8
Figure 2.6: Bioinformatics	9
Figure 3.1 : PatternHunter compared to Megablast	22
Figure 3.2: PH alignment rank and score.	23
Figure 3.3: PH sensitivity	24
Figure 4.1: Alignment score	27
Figure 4.2: Borders $r, s$ of a string $x$	30
Figure 4.3: Extension of a border	30
Figure 4.4: Prefix of length $i$ of the pattern with border of width $b[i]$	30
Figure 4.5: Border of length $m$ of a prefix $x$ of $pt$	32
Figure 4.6: Shift of the pattern when a mismatch at position $j$ occurs	33
Figure 5.1 : AFALS Algorithm	35
Figure 5.2: AFALS Algorithm Flow chart	36
Figure 5.3 : Data partition example	37
Figure 5.4 : KMP output example	37
Figure 6.1: Classical Waterfall Model	42
Figure 6.2: Modified Waterfall Model	43
Figure 6.3: BlueJ Screen	45
Figure 6.4 : Classes in AFALS-N	46
Figure 6.5 : MainWindow Class	47
Figure 6.6 : Test Class	47
Figure 6.7 : AFALS-N screen	48
Figure 6.8 : File menu	49



Figure 6.9 : Edit menu	49
Figure 6.10 : View menu	49
Figure 6.11 : Help menu	49
Figure 6.12 : Alignment tap	50
Figure 6.13 : Result tap	51
Figure 6.14 : About tap	52
Figure 6.15 : Terminal Window	52
Figure 7.1 : Affected person mutation	54
Figure 7.2: Leukemia Mutations	55
Figure 7.3: Breast Cancer Mutation	55
Figure 7.4 : Execution times for sequences of size ranging from 1 kBP to 3MBP	57

## List of appendices

<b>Appendix</b>	<b>Page</b>
Appendix A: Sample of Test Data	64
Appendix B: Sample of Java Code	70

### List of abbreviations

Abbreviation	Meaning
AFALS-N	An Algorithm for Finding Approximate Local Similarities in DNA Sequences -Najah
NCBI	National Center for Biotechnology Information
DNA	Deoxyribonucleic Acid
RNA	Ribonucleic Acid
PH	PatternHunter
KMP	Knuth, Morris and Pratt algorithm
FLT3	Fms-related tyrosine kinase 3
AML	Acute myeloid leukemia
indel	Insertion or deletion mutation
URL	Uniform Resource Locator
BLAST	Basic Local Alignment Search Tool

**Abstract:**

Finding approximate local similarities in long DNA sequences is very important in bioinformatics. These local regions of approximated similarity may be a consequence of functional, structural, or evolutionary relationships between the sequences.

DNA sequences, which hold the codon of life for every living organism, can be abstractly viewed as very long strings over a four-letter alphabet of A, C, G, and T. Proteins which use an alphabet of 20 symbols, are translations from selected stretches of DNA, using a predefined translations table where each 3 letters of DNA translated to one amino-acid.

Many projects to sequence the genome of some species are well advanced or calculated. The very large number of species (and their genetic variations) that is of interest to man, suggest that many new sequences will be revealed as the improved sequencing techniques and analysis are deployed.

Consequently, we are at a technical threshold. Techniques that were capable of exploiting the smaller collections of genetic data, for example via serial search, may require radical revision.

Several techniques have been developed to address this problem. However this study focuses not only on developing an algorithm, we also suggest advanced way to find acceptable results with increased sensitivity and decreased computation time using heuristics.

The proposed algorithm (AFALS-N) has been presented as an approximate local similarities finder and as a pair wise alignment algorithm. It has been implemented using java and tested with real DNA sequences.

The experimental results have shown that AFALS-N performed better than PatternHunter. When Compared with PatternHunter the enhancement over execution time was 0.9%. Also AFALS-N has achieved 66% sensitivity.

**Keywords:** Bioinformatics, DNA Alignment, Approximate Similarities, Heuristics, seed.

## Chapter One

### Introduction

In bioinformatics, a sequence alignment is a way of arranging the primary sequences of DNA (Deoxyribonucleic Acid), RNA (Ribonucleic Acid), or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences [4].

There are many types of alignments, Local or global alignment, and multiple or pair-wise alignments. The most important of these alignments is the combination of local and pair-wise alignment which is a powerful tool in DNA analysis because it can uncover the homology relationship between two sequences. And also because the nature of small conserved regions in DNA that is conserved from mutations [32].

Finding a specific pattern in DNA is considered as a primary stage before many DNA processing procedures. Furthermore, approximate string matching has many applications in bioinformatics besides finding specific genes in DNA like finding similar parts of protein , RNA [26].

Approximate string matching has many applications including data retrieval, Uniform Resource Locator (URL) processing, language dictionaries. Therefore, the efficiency of approximate string matching has a great impact on the performance of these applications [9].

Approximate string matching is the technique of finding approximate matches to a pattern in a string. The closeness of a match is measured in terms of the number of primitive operations necessary to convert the string into an exact match. The usual primitive operations are insertion, deletion and substitution [9].

Many algorithms have been developed to gain the optimal local alignment. Previous algorithms use dynamic programming which always guarantee the optimal solution but with an increase in computational time. Current algorithms use heuristics which is faster than dynamic programming but sacrifices some of accuracy.

The running time of dynamic programming algorithms must be cut down in order to achieve practical run time , and the accuracy of algorithm that use heuristics must be increased to reach optimal . This is what we offer as thesis subject , an algorithm that finds the approximate local pair-wise alignment of DNA sequence within a reasonable computational time that is less than dynamic programming algorithms time , and more accurate than heuristic algorithms .We have developed this algorithm using heuristics.

### **1.1 Scope of the Study**

This study focuses on the pairwise local alignments in DNA sequences and developing an algorithm that falls in this scope.

### **1.2. Aims and Objectives**

Our aim in this research is to develop an algorithm that balance between the accuracy of dynamic programming algorithms such as smith-waterman algorithm and the speed of heuristic algorithms such as BLAST (Basic Local Alignment Search Tool).

In the proposed algorithm we tried to increase the sensitivity of finding the approximate local similarities between two pairs of DNA sequences without increasing in time at minimum.

### **1.3. Significance of the Study**

This thesis serves the biologists, physicians, lab technicians and researchers who are Interested in DNA processing.

### **1.4. Contributions**

The research contributions may be recorded as follows:

- Proposing new algorithm which decreases the time and space needed as compared to some of the currently used algorithms for solving the problem of local pair wise approximate string matching.

- Developing a tool for finding local similarities in DNA sequences.

## 1.5 Thesis Outline

The remaining of this thesis is organized as follows:

Chapter 2: Presents an overview of bioinformatics and approximate local similarities in DNA.

Chapter 3: Describes previous related work.

Chapter 4: Describes the methodologies that have been used.

Chapter 5: Describes the proposed algorithm.

Chapter 6: Describes the AFALS-N software.

Chapter 7: Discusses the results of Algorithm experiments.

Chapter 8: Presents conclusions and future work.

## Chapter Two

### Bioinformatics

In bioinformatics, a sequence alignment is a way of arranging the primary sequences of DNA, RNA, or protein to identify regions of similarity that may be a consequence of functional, structural, or evolutionary relationships between the sequences. So alignment is equivalent to finding approximate similarities. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that residues with identical or similar characters are aligned in successive columns [2].

There are some times unknown constraints on the sequences that cause the correct alignment to differ from the optimal alignment given by an algorithm. Hence, it is of some interest to produce all alignments with score within a specified distance of the optimum score, which is called near optimal alignment [26].

Current 'mainstream' alignment algorithms have optimization criteria based primarily on computational efficiency using parameters such as gap penalties, which are not biologically motivated. In addition, current alignment algorithms such as the Smith and Waterman technique provide a single alignment that could be sensitive to rather arbitrary choices in parameters such as gap penalties [2].

The heuristic algorithms such as BLAST is fast but have a weakness which is that there is a possibility of missing an alignment or giving inaccurate output [26].

The challenge in performing sequence alignments has been the tradeoff between accuracy and efficiency. Traditional algorithms which use dynamic programming tend to have a very high computational complexities, however manage to find the optimal alignment. Other algorithms which use heuristics sacrifice some of this accuracy to make the alignments faster; they find reasonably good alignments or find the optimal alignment reasonably often [26].

Before introducing sequence alignment, there are some concepts must be discussed. Starting with Bioinformatics, which is the broad discipline of sequence alignment, then



DNA (Deoxyribonucleic Acid), on which we do alignment and find approximate local similarities in DNA.

## 2.1 Bioinformatics

Bioinformatics is a discipline which originally arose for the utilitarian purpose of introducing order into the massive data sets produced by the new technologies of molecular biology [4].

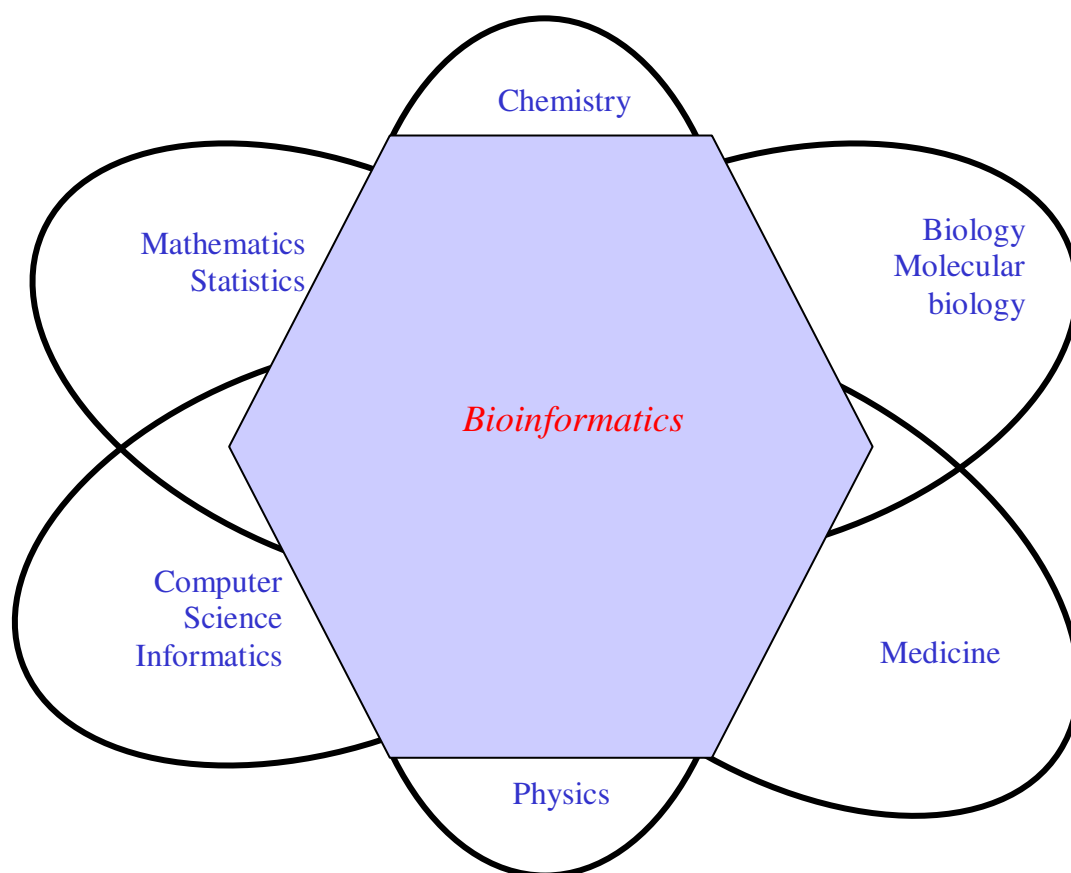


Figure 2.1: Bioinformatics [12].

Bioinformatics, or computational biology, refers to an emerging, interdisciplinary field in which computer technology, including software, hardware and algorithms are applied to solve problems arising in biology. One subject, of particular interest in the field, is to develop tools for processing biomolecular data. These data include DNA (deoxyribonucleic acid), RNA (ribonucleic acid), protein sequences, and their two-dimensional (2D) and three-dimensional (3D) structures [10].

Bioinformatics has been developed in the space, which was already occupied by a number of related disciplines. These include quantitative sciences such as [12]:

- Mathematical and computational biology,
- Biometry and biostatistics,
- Computer science,
- Cybernetics,

As well as biological sciences such as

- Molecular evolution,
- Genomics and proteomics,
- Genetics,
- Molecular and cell biology.

## 2.2 Deoxyribonucleic Acid (DNA)

Deoxyribonucleic Acid, DNA, is the molecule of life. DNA is a double helix comprising two DNA strands running anti parallel to each other and is made of many units of nucleotides, which each consist of sugar, a phosphate and a base [28].

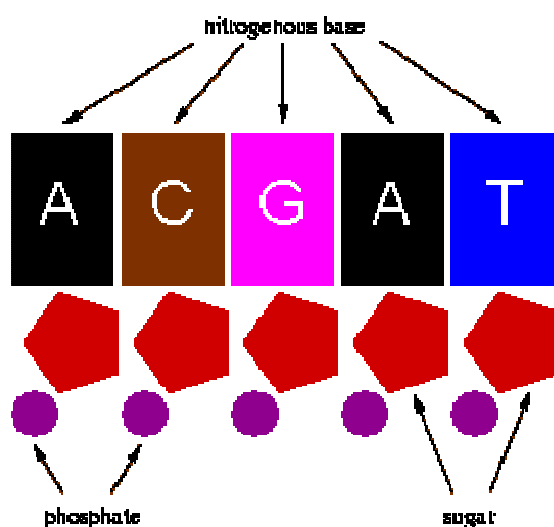


Figure 2.2:DNA Components [28].

Each strand of the DNA double helix is a polymer built from four components, called nucleotides: A, T, C, and G (the abbreviations for adenine, thymine, cytosine, and guanine). The two strands of DNA are complementary: whenever there is a T on one strand, there is an A in the corresponding position on the other strand; whenever there is

a G on one strand, there is a C in the corresponding position on the other. DNA can be represented by a sequence of these four letters, or bases [28].

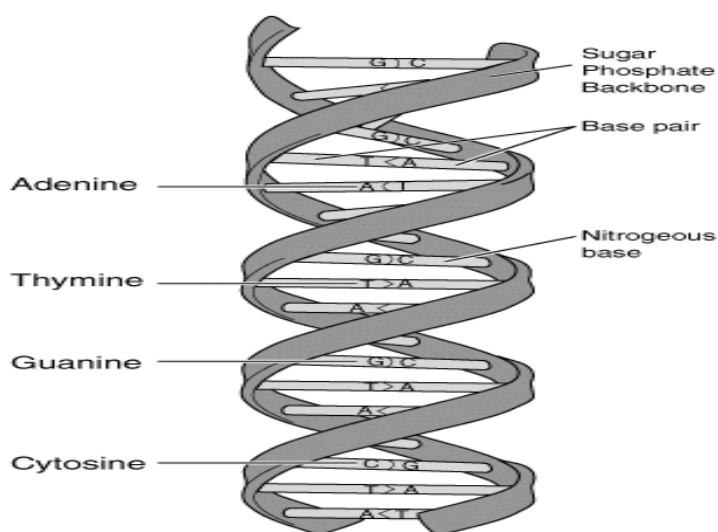


Figure 2.3:DNA double helix [28].

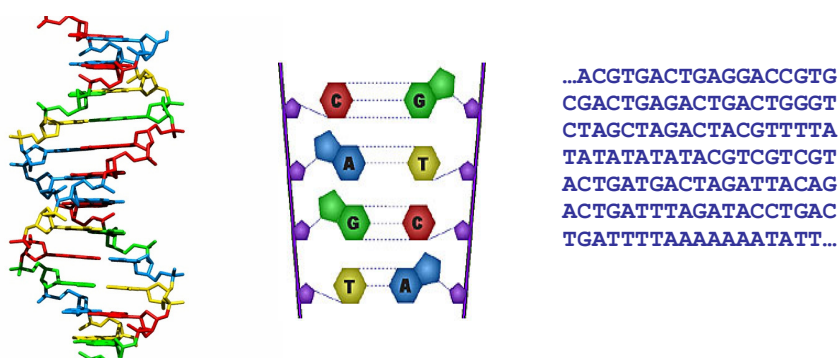


Figure2.4:DNA sequencing[28].

### 2.3 Sequence Alignment

Alignment is one of the basic data mining and analysis methods in bioinformatics. Data mining and analysis aims at nontrivial extraction by computational means, of previously unknown and potentially useful information from data, or search for relationships and patterns that exist in databases [13].

Sequence alignment is defined as the process of lining up two or more sequences to achieve maximal levels of similarity and the possibility of homology (sequences that share a common ancestor) [12].

```

VLSPADKTNVKA AWAKVGAHAAGHG
| | | |   |   | | | | | | | |
VLSEAEWQLVLHVWAKVEADVAGHG

```

Figure 2.5: DNA sequencing [12].

The sequence alignment indicates the changes that could have occurred between two homologous sequences and a common ancestral sequence during evolution [27].

Alignments are commonly represented both graphically and in text format [16]. In almost all sequence alignment representations, sequences are written in rows arranged so that aligned residues appear in successive columns. In text formats, aligned columns containing identical or similar characters are indicated with a system of conservation symbols [22].

There are many ways to consider the sequence alignment :

#### 1- Pair-wise vs Multiple sequence alignment.

Pair wise sequence alignment methods are used to find the best-matching piecewise (local) or global alignments of two query sequences. Pair wise alignments can only be used between two sequences at a time, but they are efficient to calculate and are often used for methods that do not require extreme precision (such as searching a database for sequences with high homology to a query). The three primary methods of producing pair-wise alignments are dot-matrix methods, dynamic programming, and word methods (Heuristic methods) [2].

Multiple sequence alignment is an extension of pair-wise alignment to incorporate more than two sequences at a time. Multiple alignment methods try to align all of the sequences in a given query set. Multiple alignments are often used in identifying conserved sequence regions across a group of sequences hypothesized to be evolutionarily related. Such conserved sequence motifs can be used in conjunction with

structural and mechanistic information to locate the catalytic active sites of enzymes. Alignments are also used to aid in establishing evolutionary relationships by constructing phylogenetic trees [2].

## 2- Local vs global Alignment.

Global alignments, which attempt to align every residue in every sequence, are most useful when the sequences in the query set are similar and of roughly equal size[9]. A general global alignment technique is called the Needleman-Wunsch algorithm and is based on dynamic programming. Local alignments are more useful for dissimilar sequences that are suspected to contain regions of similarity or similar sequence motifs within their larger sequence context. The Smith-Waterman algorithm is a general local alignment method also based on dynamic programming. With sufficiently similar sequences, there is no difference between local and global alignments [16].

Local alignment identify regions of similarity within long sequences that are often widely divergent overall . The rationale for local similarity searching is that functional sites are localized to relatively short regions , which are conserved irrespective of deletions or mutations in intervening parts of the sequence . Thus , a search for local similarity may produce more biologically meaningful and sensitive results than a search attempting to optimize alignment over the entire sequence lengths ( global alignment ) [4] .

```

Global  FTFTALILLAVAV
        F--TAL-LLA-AV

Local   FTFTALILL-AVAV
        --FTAL-LLAAV--
  
```

Figure 2.6:Bioinformatics [16].

Local alignment are often preferable but can be more difficult to calculate because of the additional challenge of identifying the regions of similarity [27].

In [16] stated that local alignment are more suitable and meaningful for :

- 1- Aligning sequences that are similar along some of their lengths but dissimilar in others.
- 2- Sequences that share conserved regions or domains.
- 3- Sequences that differ in length.

Hybrid methods, known as semiglobal or "glocal" methods, attempt to find the best possible alignment that includes the start and end of one or the other sequence. This can be especially useful when the downstream part of one sequence overlaps with the upstream part of the other sequence. In this case, neither global nor local alignment is entirely appropriate: a global alignment would attempt to force the alignment to extend beyond the region of overlap, while a local alignment might not fully cover the region of overlap [32].

#### **2.4 Approximate Local Similarities in DNA**

Pattern matching occurs in various applications, ranging from simple text searching in word processors to identification of common motifs in DNA sequences in computational biology. The problem of exact pattern matching has been well studied and a number of efficient algorithms exist. However these exact pattern matching algorithms are of little help when they are applied to finding patterns in DNA sequences. The DNA sequence search is inheritably inexact in nature because there are acceptable equivalences of amino acids that made up of the sequence. Current inexact pattern matching algorithms are based on four approaches: (1) Dynamic Programming; (2) Automata; (3) Bit-Parallelism;(4) Filtering [26].

The problem of string matching is very simply stated. Given a body of text  $T[1\dots n]$  we try to find a pattern  $P[1\dots m]$  where  $m \leq n$ . This can be used to search bodies of text for specific patterns, or in biology, can be used to search strands of DNA for specific sequences of genes. Approximate string matching is a much more complicated problem to solve and has many more real world applications. Unfortunately, in real world applications the problem is not so cut and dry. This is where approximate string matching comes in. Instead of searching for the string exactly, approximate string matching searches for patterns that are close to  $P$ . In other words approximate string

matching allows for a certain amount of error between the two strings being compared. In this research we will define this more formally later [9].

One of the earliest applications of approximate string matching was in text searching. The approximate string matching algorithms can be applied to account for errors in typing. Internet searching is particularly difficult because there is so much information and much of it has errors in it. Also, since the internet spans many different languages, errors frequently arise in comparing words across language barriers. Also, text editors have to use approximate string matching when performing spell checks. Additionally, spell checkers have to generate a list of “suggested words” that are close in spelling to the misspelled word [9].

Another application of approximate string matching is in biology. As with text, ideally, exact string matching should be effective. But in reality, DNA searching is not an exact science. There are frequently mutations in DNA that a string matching algorithm must account for. In fact, oftentimes these mutations are sought out because they may indicate disease or other genetic problems [26].

#### **2.4.1 History of the Problem**

The problem of approximate string matching is obviously an offspring of the much simpler exact string matching problem. The simple brute force algorithm for exact string matching runs in  $O(nm)$  time where  $n$  is the length of the first sequence and  $m$  is the length of the second sequence. The first major advance in exact string matching algorithms came in 1965 when Levenshtien [4] developed a dynamic algorithm to compute distance in  $(n*m)$  time. That is still the premier algorithm used today. In 1970 Cooke [5] mathematically discovered that there was a possible algorithm to solve the problem in  $O(n+m)$  time. It was Knuth, Morris, and Pratt [6] that used Cooke’s theorem to produce an actual algorithm in 1976 [9].

#### **2.4.2 Formal Definition**

Consider two strings of text  $T[1..n]$  and  $P[1..m]$ , and a distance function  $d(x[i..j], y[a..b])$  where  $x[i..j]$  and  $y[a..b]$  denote substrings of  $x$  and  $y$ .  $d(x[i..j], y[a..b])$  computes

the minimal cost of converting  $x[i..j]$  into  $y[a..b]$ . There are three operations we can perform to convert  $x$  into  $y$ , each with a cost [9].

**Substitution:** To perform a substitution we simply take one character in  $x$  and change it to match a character in  $y$ .

**Insertion:** An insertion is when a character is simply inserted into  $x$  to match the character in  $y$  at the same position.

**Deletion:** This is the opposite of insertion. As the name suggests, it is the act of removing a character in  $x$  [9].

Obviously, conversions can very easily be made through a series of  $m$  insertions at the front of  $x$ , followed by  $n$  deletions. However, this is usually not optimal, except in the worst case. Intuitively it's easy to see when each of these operations would be used in the optimal way. However, it's much more difficult to define the optimal conversion in a specific form. The final input to the approximate string matching problem is  $k$ , the maximum allowable error. Then the problem is to calculate the set of  $P[i..j]$  such that  $d(T[x..y], P[i..j]) \leq k$ .

### 2.4.3 Approximate String Matching

The need to align inexact sequence data arises in various fields and applications such as computational biology, signal processing and text processing. In particular, in DNA sequence analysis, exact sequence matching is rare. Due to possible DNA mutation, the biological inference does not expect an identical match, but rather a high sequence similarity usually implies significant functional or structural similarity [9].

Inexact pattern matching is sometimes referred as “approximate pattern matching” or “matching with  $k$  mismatches/differences”. This problem, in the general form, can be stated as: Given a pattern  $P$  of length  $m$  and a string (or text)  $T$  of length  $n$  ( $m \leq n$ ), find all the occurrences of substrings  $X$  in  $T$  that are “similar” to  $P$ , allowing a limited number, say  $k$ , of “errors” in the “similarity” matches. The “errors” are the total cost of transforming the pattern  $P$  so that  $P$  and  $X$  are equal. The common allowable edit/transformation operations are insertion, deletion and substitution. The common error model is called “edit distance”. The edit distance is the minimal number of edit operations required to transform the first sequence into the second [9].

Inexact pattern matching algorithms can be classified into four main categories:



### 1. Dynamic Programming Approach

This is the oldest among the four approaches and the most commonly used approach, especially in the area of biological sequence analysis. Examples are the Needleman–Wunsch algorithm and Smith-Waterman algorithm. These algorithms are much more complex than the ones for exact pattern matching. It involved solving successive recurrence relations recursively. I.e. smaller problems are solved in succession to solve the main problem. The classical dynamic programming algorithm can also be thought of as a column-wise “parallelization” of the automaton [26].

The major advantage of dynamic programming approach is its flexibility in adapting to different edit distance functions. In general, the worst case complexity is  $O(mn)$ . Over the past two decades, a number of improved solutions have been proposed to lower the worst case complexity to  $O(kn)$  and average complexity of  $O(kn/\sqrt{|\Sigma|})$  [9].

### 2. Automata Approach

This approach is also rather old. Though automata approach doesn't offer time advantage over Boyre-Moore algorithm for exact pattern matching, this approach does offer better running time for inexact pattern matching. Both the average and worst case performance remain  $O(m+n)$  [9].

### 3. Bit-Parallelism

This approach is rather new (after 1990) and is based on exploiting the intrinsic parallelism of the bit operations inside a computer word. The basic idea is to “parallelize” another algorithm, using bits. In general, the number of operations that an algorithm performs can be cut down by a factor of at most  $w$ , where  $w$  is the number of bits in a computer word. Since in current computer architectures,  $w$  is 32 or 64, the speedup is very significant in practice. The results are especially significant when short patterns are involved. They may work effectively for any error level [3].

The first bit-parallel algorithm is known as “Shift-Or” which searches a pattern in a text (without errors) by parallelizing the operation of a nondeterministic finite automaton that looks for the pattern. This automaton has  $m+1$  states, and can be simulated in its

nondeterministic form in  $O(mn)$  time. For patterns longer than the computer word (i.e.  $m > w$ ), the algorithm uses  $(m/w)$  computer words for the simulation. The algorithm is  $O(n)$  on average. Bit-parallelism has become a general way to simulate simple nondeterministic automata instead of converting them to deterministic form. It has the advantage of being much simpler, in many cases faster, and easier to extend in handling complex patterns than its classical counterparts. Its main disadvantage is the limitation it is imposed by the size of the computer word. In many cases its adaptations for longer pattern search are not very efficient [9].

There are two main trends in bit-parallelism approach: (1) parallelize the work of the dynamic programming matrix; or (2) parallelize the work of the nondeterministic automaton [3].

#### 4. Filtering Algorithms

This approach started in 1990 and has been most very active since. Most of the new algorithms proposed in recent years belong to this class [3]. Filtering is based on the fact that it may be much easier to tell that a text position does not match than to tell that it matches. It is formed by algorithms that filter the text, quickly discarding text areas that do not match. Since the exact searching algorithms is much faster than approximate searching ones, most filtering algorithms take advantage of this fact by searching pieces of the pattern without errors [9].

Filtering algorithm, by itself, is normally unable to discover the matching text positions. Rather, it is used to discard large areas of the text that cannot contain a match. Filtering algorithms must couple with a process that verifies all those potential text matching positions. Any non-filtering algorithm can be used for this verification. The selection is normally independent, but the verification algorithm must behave well on short texts because it can be started at many different text positions to work on small text areas [9]. The major interest in this approach is the potential for algorithms that do not inspect all text characters. These filtering algorithms have a theoretical average running time  $O(n(k + \log m)/m)$ , which was proven optimal. In practice, filtering algorithms are among the fastest too [3].

The main drawback of this approach is that the performance of filtering algorithms is very sensitive to the error level. Most filters work very well on low error levels and very badly otherwise. This is related to the amount of text that the filter is able to discard

.When evaluating filtering algorithms, it is important not only to consider their time efficiency but also their tolerance for errors [3].

## Chapter Three

### Literature Review

Here, we shall look at the main algorithms: the dynamic programming algorithms by Needleman-Wunsch and Smith-Waterman, and the heuristic approximate alignment algorithms FASTA, BLAST and PatternHunter. We shall look at the algorithm itself and the computational and space complexity of each algorithm. From this, we can compare the efficiencies of the various algorithms and see what sacrifices the algorithms make in exchange for speed.

#### 3.1 Needleman-Wunsch algorithm

The Needleman-Wunsch algorithm [25], published in 1970, provides a method of finding the optimal global alignment of two sequences by maximizing the number of amino acid matches and minimizing the number of gaps necessary to align the two sequences. Because the Needleman-Wunsch algorithm finds the optimal alignment of the entire sequence of both sequences, it is a global alignment technique, and cannot be used to find local regions of high similarity [26].

In pairwise sequence alignment algorithms, a scoring function,  $F$ , must exist such that different scores can be assigned to different alignments of two proteins relative to the number of gaps and number of matches in the alignment. Thus, the alignment with the largest score must be the optimal alignment. In this scoring function, let  $m$  be the score for two residues matching,  $s$  is the penalty for mismatches, and  $g$  is the penalty for inserting a gap. The Needleman-Wunsch algorithm realizes that the score of aligning the entire proteins is the same as the sum of the scores of two subsequences of the proteins,  $F(x_{1:M}, y_{1:N}) = F(x_{1:i}, y_{1:j}) + F(x_{i+1:M}, y_{j+1:N})$  where  $M$  is the length of sequence  $x$ ,  $N$  is the length of sequence  $y$ , and  $1 < i < M$  and  $1 < j < N$ . From this, we can see that the optimal score of two partial sequences is the sum of score of residue  $i$  in sequence  $x$  and residue  $j$  in sequence  $y$ , and the maximum score aligning the rest of the sequences [25].

The overall time complexity of this algorithm is  $O(MN)$  and the total space complexity of this algorithm is  $O(MN)$  [24].

It is important to note here that the Needleman-Wunsch algorithm supports different scores for exact residue matches, similar residues, and gaps. A PAM or BLOSUM weight matrix can be used to weight residue matching scores[25]. These weighted scores can affect the final alignment of the two protein sequences and the biological relevance of the alignment, but will not affect the time or space complexity of the algorithm because the number of operations will not change. This alignment is limited, however, because it can only align entire proteins. A different algorithm was developed to create local alignments[26].

### 3.2 Smith-Waterman algorithm

The Smith-Waterman algorithm was published in 1981 [29] and is very similar to the Needleman-Wunsch algorithm. Yet, the Smith-Waterman algorithm is different in that it is a local sequence alignment algorithm. Instead of aligning the entire length of two DNA sequences, this algorithm finds the region of highest similarity between two DNAs. This is potentially more biologically relevant due to the fact that the ends of DNA tend to be less highly conserved than the middle portions, leading to higher mutation, deletion, and insertion rates at the ends of the sequence.

Only two things were changed in the Needleman-Wunsch algorithm to obtain the Smith-Waterman algorithm[29]. When filling the matrix, we do not let any of the matrix values become negative, and thus we consider 0 as potentially being the maximum value of the three other cases (where  $x_i = y_j$ , or there is a gap in  $x$  or a gap in  $y$ ). By not letting any of the values go below zero, we stop considering regions of high dissimilarity which have no good alignments. This allows the algorithm to focus on only those regions of the protein which are similar. The second change in the algorithm is in the traceback. Instead of starting at the n-terminus of both sequences, we start at the cell with the highest score in the entire matrix. This allows for the alignment of the similar subsequences of the proteins [26].

The complexity of the Smith-Waterman algorithm can also be computed. The time complexity of the initialization is  $O(M+N)$  because we need to initialize row 0 and column 0. In filling the matrix, we traverse each cell of the matrix and perform a constant number of operations in each cell, and thus the time complexity for this part is

$O(MN)$ . Thus far, the complexity of the Smith-Waterman algorithm is exactly the same as that for the Needleman-Wunsch algorithms. However, in the traceback, the algorithm requires the maximum cell be found, and this must be done by traversing the entire matrix, making the time complexity for the traceback  $O(MN)$  [26]. It is also possible to keep track of the largest cell during the matrix filling segment of the algorithm, although this will not change the overall complexity. Thus the total time complexity of the Smith-Waterman algorithm is

$$O(M+N) + O(MN) + O(MN) = O(MN)$$

which is identical to the complexity of the Needleman-Wunsch algorithm. The overall running time of this algorithm is actually slightly slower than the Needleman-Wunsch algorithm however, because more comparisons must be made when comparing the scores to 0, and when finding the largest cell during the traceback [24].

The space complexity of the Smith-Waterman algorithm is also unchanged from the Needleman-Wunsch algorithm. This is due to the fact that the same matrix is used and the same amount of space is needed for the traceback. Thus, there is no definite space or time advantage of one algorithm over the other. However, the Smith-Waterman algorithm tends to model protein homology better because it ignores misalignments at the ends of the proteins which are often not highly conserved. Thus, database searches are usually done with the Smith-Waterman algorithm over the Needleman-Wunsch algorithm which tends to model homology better in distantly related proteins. The Needleman-Wunsch algorithm will tend to be better for proteins which are closely related, with fewer mutations because the ends of the protein in closely related sequences will not be changed significantly [26].

The overall time complexity of this algorithm is  $O(MN)$  and the total space complexity of this algorithm is  $O(MN)$  [24].

### **Affine Gap Penalty**

In the Needleman-Wunsch and the Smith-Waterman algorithms, there existed a constant gap penalty,  $d$ , for a single missing or inserted residue. Thus, to insert a gap of size  $l$ , the total penalty would be  $d * l$ . However, in biological systems, a deletion or insertion of a

large number of residues may be significantly less rare than this, and thus, a different model of gap penalties must be used [26].

Realistically, gaps of different sizes would all have different penalties, but using this model increases the complexity of either algorithm from  $O(MN)$  to  $O(M_2N)$ . This is because when computing the score of each cell, instead of finding the maximum of three adjacent cells, we must find the number of cells to the right or down which also are included in the gap. Thus, we must look at  $i+j+1$  cells, which increases the time complexity to  $O(M_2N)$  [26].

To get around this increase in complexity, we can use affine gap penalties in which the initial gap opening penalty is set at a constant value,  $d$ , and extending the gap by a single residue is set at a constant, lower value,  $e$ . This linear gap penalty function is easier to deal with. In this case, we must keep track of two things for each cell in the matrix. We must keep track of the score of the aligned subsequences  $x_{1:i}$  and  $y_{1:j}$  plus the score of aligning  $x_i$  and  $y_j$ . We can store these values in matrix  $F(i,j)$ . We must also keep track of the score of the aligned subsequences  $x_{1:i}$  and  $y_{1:j}$  plus the score of inserting a gap at either  $x_i$  or  $y_j$ . We can store these values in  $G(i,j)$ . Here  $F(i,j)$  is the max score when  $x_i$  and  $y_j$  are aligned (either ending a gap at  $G(i-1,j-1)$ , or continuing an alignment in  $F(i-1,j-1)$ ).  $G(i,j)$  is the max score when either starting a gap in  $F$  with a penalty of  $d$  or extending a gap in  $G$  with an extension penalty of  $e$  [26].

The initialization, in this case, is also  $O(M+N)$  because row 0 and column 0 must be initialized to the linear gap penalty,  $d+(j-1)e$  or  $d+(i-1)e$  respectively. In the iterative phase, we now have two matrices to fill, but each cell of both matrices still only requires a constant number of operations. Each matrix has a time complexity of  $O(MN)$  yielding  $2O(MN) = O(MN)$  complexity. Finally, the traceback is still  $O(M+N)$  because it is unchanged. Thus, the total time complexity is  $O(MN)$  which is the same as the Needleman-Wunsch and Smith-Waterman complexities [26].

The space complexity must take into account both matrices and the space needed for traceback on both matrices. Since the space complexity of a single matrix is  $O(MN)$ , the space complexity for two matrices is  $2O(MN)=O(MN)$ . Thus, the space complexity is also unchanged. However, the actual space used is two times the space used for

Needleman-Wunsch and Smith-Waterman, and the running time is also about two times as long for the affine gap model. Thus, we see that increasing biological accuracy involves a sacrifice in efficiency [29].

### 3.3 FASTA algorithm

The FASTA algorithm was developed in 1985 by Lipman and Pearson [18]. Unlike the Needleman-Wunsch and Smith-Waterman algorithms, FASTA approximates the optimal alignment by searching and matching *k-tuples*, or subsequences of length *k*. The algorithm assumes that related proteins will have regions of identity, and by searching with *k-tuples*, the FASTA algorithm allows small regions of local identity to be found quickly. For proteins, these *k-tuples* tend to be of length two. FASTA creates a hash table of all possible *k-tuples* and goes through the entire query protein of length *N* and inputs the location of all the *k-tuples* into the table. Each *k-tuple* in the database sequence can be looked-up in the hash table, and any matches will allow the algorithm to mark the matching cells in the matrix. This results in a matrix in which all points of local identity of length *k* are marked [18].

The FASTA algorithm then identifies the ten highest scoring diagonal runs by identifying each marked point on the matrix, and adding a positive score for every other marked cell along a diagonal, and subtracting a penalty for unmarked cells between marked cells along the diagonal. These ten highest scoring segments are kept, and all other segments of local alignment are discarded. The ten diagonals are scored once again using an amino acid weight matrix (PAM or BLOSUM matrix) and any diagonals with scores below a threshold are discarded again. The highest scoring diagonal is termed *init1*. Thus, we are left with ten or fewer regions in which the two proteins align with no gaps (although mismatches are allowed in the form of missing marked cells along the diagonal). The FASTA algorithm assumes that the optimal alignment will include or be near the *init1* diagonal [26].

The FASTA algorithm is substantially faster than the Needleman-Wunsch or Smith-Waterman alignments and thus can be more easily used in database queries [26].



In the worst case, the time complexity of FASTA is  $O(MN)$  and the space complexity of this algorithm is also  $O(MN)$ . But the average-case complexity would be about  $O(MN/20^k)$ . Thus, the complexity of the FASTA algorithm depends on the size of the  $k$ -tuples, and the larger the  $k$ -tuples, the faster the algorithm. Although the FASTA algorithm is faster than any of the previous algorithms, it is not guaranteed to find the optimal alignment between two proteins [24].

### 3.4 BLAST algorithm

The BLAST (Basic Local Alignment Search Tool) algorithm was developed by Altschul et al. in 1990 [1] and similar to the FASTA algorithm, is also a heuristic pairwise sequence aligner. However, the basis of the BLAST algorithm is the use of words and High-scoring Segment Pairs (HSPs) instead of  $k$ -tuples. BLAST begins by finding all words, or subpeptides of length  $w$  (typically 3), which exist in the protein sequence. Using a substitution matrix, a list of other words, called a neighborhood, is created for each word found in the protein sequence; these words must be related to the original word and must have a substitution matrix score higher than  $T$ , else they are not considered. For fast access to these data, the word positions are entered into a hash table. Each word in the database sequence can be compared to the hash table, and only those matches which are deemed statistically significant by a statistical method will be kept. This significantly reduces the number of hits which must be analyzed. Every match of a word in the database sequence with one of the neighbor words is called a High-scoring Sequence Pair (HSP) and these act as “seeds” to start a local sequence alignment [26]. The time complexity of BLAST is  $O(20^W)$  and the space complexity is  $O(20^W + MN)$  [24].

### 3.5 PatternHunter

Ma, Tromp and Li had a quite different observation. Drawing upon ideas from the pattern matching literature, they noted that one can find seeds in more alignments if one requires an exact match in  $k$  positions, but does not require them to be consecutive. Their program, PatternHunter [19] and its sequels [17], allow one to find local alignments of either nucleotide or protein sequences, using this approach [15].

When a comparison made of PatternHunter with Blastn and MegaBlast which are an enhanced versions of BLAST via BL2SEQ, using the most favorable parameters for Blastn and MegaBlast and standard parameters for PatternHunter. On a computer: PIII 700Mhz, 1G main memory here are the results that shows that the PH is much faster than Blastn [19] .

Table 3.1: PatternHunter compared to Blastn [19]

Sequence Length	Blastn	PatternHunter
816k vs 580k	47 sec	9 sec
4639k vs 1830k	716 sec	44 sec
20M vs 18M	out of memory	13 min

The next figures shows a comparison of PH with Megablast on long sequences and the time and memory results .

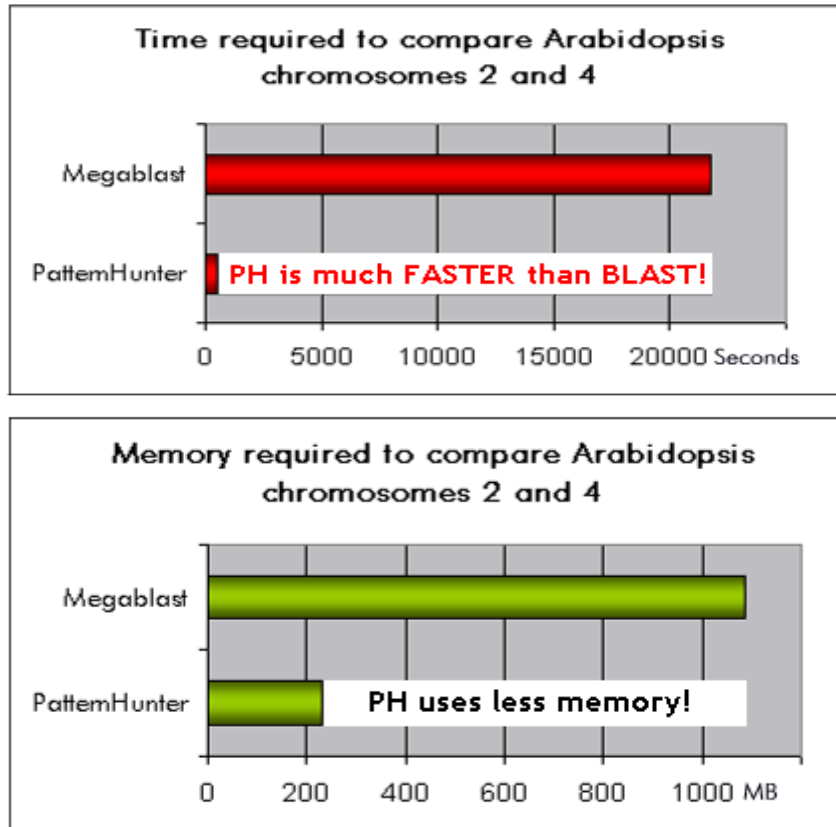


Figure 3.1 : PatternHunter compared to Megablast [19]

The output quality is also on par with the default Blastn and much superior to MegaBlast; the next figure shows a typical comparison of how alignment scores fall off (from best to worst) [19].

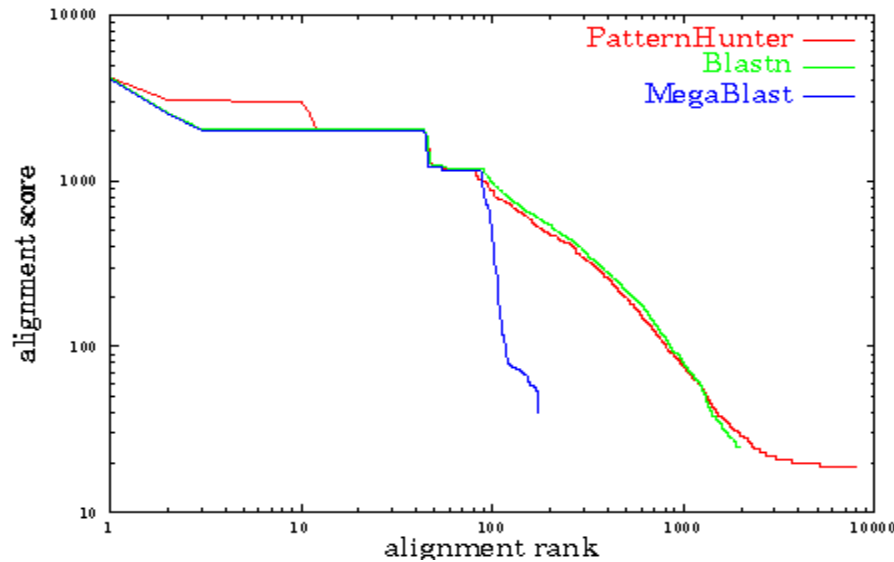


Figure3.2: PH alignment rank and score.

At default Blastn sensitivity, PatternHunter runs at MegaBlast speed, using only 1/4 of the memory used by either program. For a genome of length  $N$ , PatternHunter requires about  $8N$  bytes of internal memory. When given two inputs of lengths  $M$  and  $N$ , PatternHunter requires  $M+8N$  internal memory. Memory usage can be reduced with PatternHunter's automatic database partitioning feature [19]

There is also a comparison of the time and sensitivity of different configurations of PatternHunter with BLAST. In the following figure, Smith-Waterman algorithm's sensitivity is set to be 100%. And the sensitivity curves of PatternHunter and BLAST indicate how many of the homologies found by Smith-Waterman can be found by PatternHunter and BLAST, respectively. The data we used in this comparison are approximately 30k mouse EST sequences (25Mb) and 4k human EST sequences (3Mb). According to the figure, PatternHunter with 4 seeds run at the same speed of BLAST but with sensitivity close to Smith-Waterman [19]. PatternHunter finds a lot of alignments not found by MegaBlast [19].

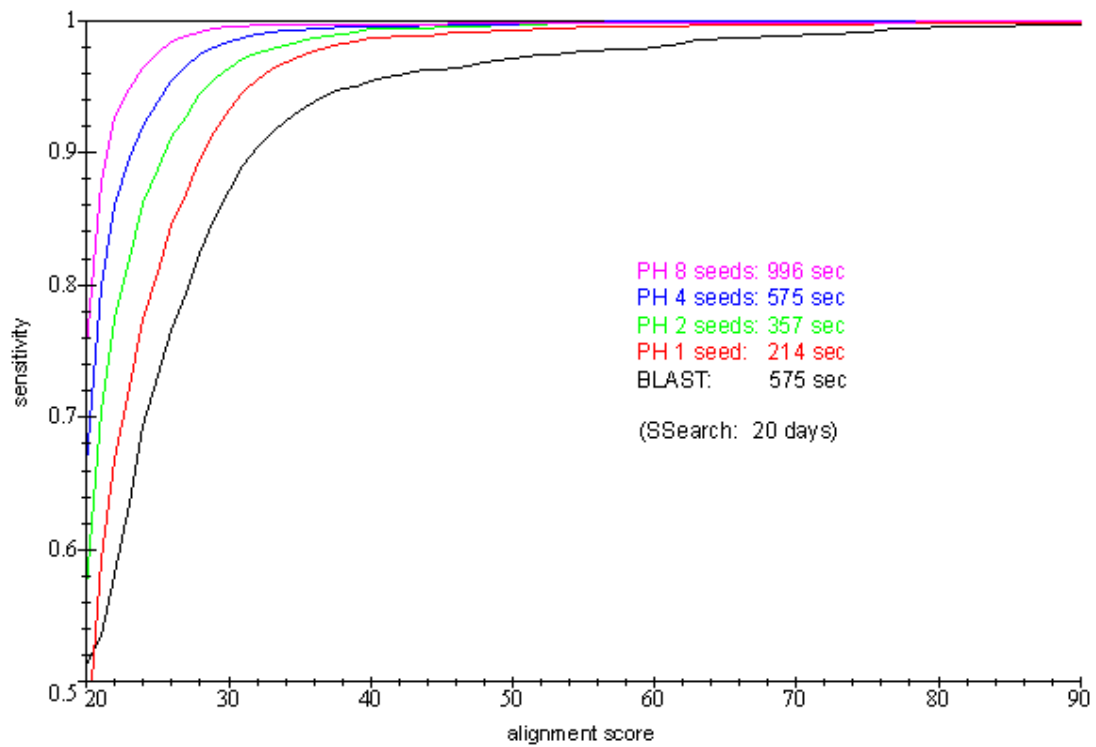


Figure 3.3: PH sensitivity [19].

We depended on patternHunter for our research results and discussion because it's the closest one to our work , and approved to be the best among related work in results .

## Chapter Four

### Methodology

In this thesis we have tried to increase the sensitivity of finding the approximate local similarities between two pairs of DNA sequences without decreasing in time at minimum .

The sensitivity of the alignment algorithm is the key to the success of such methods . The sensitivity of a search algorithm, however can have a crucial effect on the quality of the annotation; different algorithms will find( and miss ) different potential homologues under different circumstances [26].

In order to achieve our objectives we have used Heuristics , Scoring Matrices , Word length ( Seed ) , and String matching techniques.

#### 4.1 Heuristics

Because of the large search space in alignment problem which may grow in an exponential fashion we have used heuristics to reduce this search complexity by pursuing the most promising paths in the state space. In state space search , a heuristic is formalized as a rule for choosing those branches in a state space that are most likely to lead to an acceptable problem solution [7].

A heuristic is a "rule of thumb," a guideline that wasn't proven mathematically but our intuition /experience tells us is correct. When working under heuristic assumptions we can not guarantee that we will get the best answer, but we will get a correct answer, and in most cases it will be a good answer. Heuristics are usually used to improve run time [31].

The aim of heuristic is to eliminate unpromising states and their descendants from consideration by the heuristic algorithm in order to find a solution in a feasible computational time .Filtration is based on the observation that a good alignment usually includes short identical or highly similar fragments . Thus we search for short exact matches and use these short matches as seeds for further analysis.

When working with local alignments it is of interest to have an alignment with the highest score . We eliminated alignments with negative scores and zero score.

## 4.2 Scoring Matrices

A two-dimensional matrix containing all possible pair-wise nucleotides scores is called a *scoring matrix*. Scoring matrices are also called substitution matrices because the scores represent relative rates of evolutionary substitutions. Scores are real numbers but are usually represented as integers in text files and computer programs [27].

A sequence can be described in terms of the number of bits needed to specify its message .The correspondence between two aligned sequences can be expressed in terms of similarity/identity score [13]. Scoring penalties are introduced to minimize the number of gaps, the total alignment score is then a function of the identity between aligned residues and the gap penalties incurred [13].

Such matrices are constructed for:

- 1-Evaluating match/mismatch between any two characters.
- 2-A score for insertion / deletion.
- 3-Optimization of total score.
- 4-Evaluating the significance of the alignment .

The scoring scheme that we have used consists of residue *substitution scores* (i.e. score for each possible residue alignment) plus penalties for gaps which is the same scheme used by PatternHunter [19]. The *alignment score* is the sum of substitution scores and gap penalties. The alignment score thus reflects goodness of alignment. An example of a simple scoring scheme for DNA: Use '+1' as a reward for match, and '-1' as the penalty for mismatch, and ignore gaps. Thus, for DNA we can construct the following *substitution matrix* N x N for this simple scoring scheme:

```
- C T A G
C +1 -1 -1 -1
T -1 +1 -1 -1
A -1 -1 +1 -1
G -1 -1 -1 +1
```

A Substitution Score is chosen for each aligned pair of letters. The matrix scores highly identical matches of bases, and also gives 'better' scores to alignments of non-identical bases that are similar in some way, and a 'worse' score to pairs that are very dissimilar. The alignment score is the sum of the scores specified for each of the aligned pairs of letters, and letters with nulls, in the alignment. The higher the alignment score, the better the alignment.

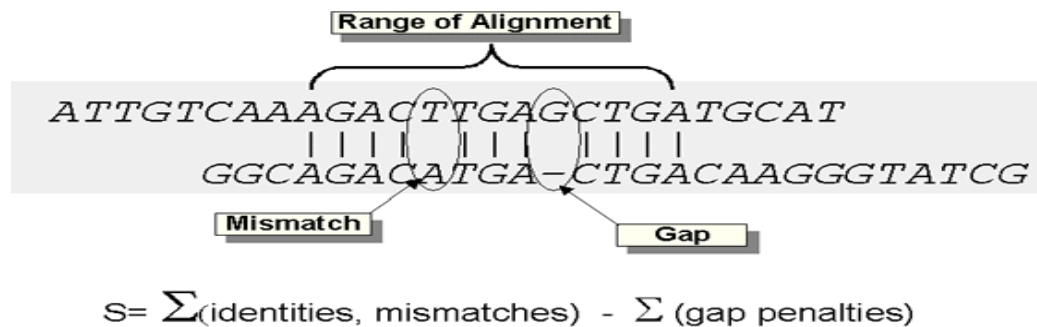


Figure 4.1: Alignment score [32]

The scoring scheme that we have used :

- 1 for mismatch
- 5 for gap opening
- 1 for gap extension
- +1 for matched

### 4.3 Word length ( Seed )

If the word length is too small the computational time increase and the sensitivity will also increase .And if the word length is large the computational time will decrease and the sensitivity decrease. Large seeds lose distant homology while small ones creates too many random hits which slow down the computation [19] .

In this research we tried to balance between word length and sensitivity in order to achieve good computational time with good result. We have used the word length 9 in the AFALS-N(An Algorithm for Finding Approximate Local Similarities in DNA Sequences–Najah) algorithm and word length 11 as a second version of it , that to compare it with PatternHunter which uses both word length .

#### 4.4 String Matching Techniques

Sequence alignment is a string-matching procedure. We have get benefit of using fast string matching algorithm besides alignments technique .We have used a KMP algorithm as a filtration mechanism to eliminate unpromising words [26].

The algorithm of Knuth, Morris and Pratt makes use of the information gained by previous symbol comparisons. It never re-compares a text symbol that has matched a pattern symbol. As a result, the complexity of the searching phase of the Knuth-Morris-Pratt algorithm is in  $O(n)$  [26].

However, a preprocessing of the pattern is necessary in order to analyze its structure. The preprocessing phase has a complexity of  $O(m)$ . Since  $m \leq n$ , the overall complexity of the Knuth-Morris-Pratt algorithm is in  $O(n)$  [26].

Definition: Let  $A$  be an alphabet and  $x = x_0 \dots x_{k-1}$ ,  $k \in \mathbb{N}$  a string of length  $k$  over  $A$ .

A prefix of  $x$  is a substring  $u$  with

$$u = x_0 \dots x_{b-1} \quad \text{where } b \in \{0, \dots, k\}$$

i.e.  $x$  starts with  $u$ .

A suffix of  $x$  is a substring  $u$  with

$$u = x_{k-b} \dots x_{k-1} \quad \text{where } b \in \{0, \dots, k\}$$

i.e.  $x$  ends with  $u$ .

A prefix  $u$  of  $x$  or a suffix  $u$  of  $x$  is called a proper prefix or suffix, respectively, if  $u \neq x$ , i.e. if its length  $b$  is less than  $k$ .

A border of  $x$  is a substring  $r$  with

$$r = x_0 \dots x_{b-1} \quad \text{and} \quad r = x_{k-b} \dots x_{k-1} \quad \text{where } b \in \{0, \dots, k-1\}$$



A border of  $x$  is a substring that is both proper prefix and proper suffix of  $x$ . We call its length  $b$  the width of the border.

Example: Let  $x = abacab$ . The proper prefixes of  $x$  are

$\epsilon$ , a, ab, aba, abac, abaca

The proper suffixes of  $x$  are

$\epsilon$ , b, ab, cab, acab, bacab

The borders of  $x$  are

$\epsilon$ , ab

The border  $\epsilon$  has width 0, the border ab has width 2.

The empty string  $\epsilon$  is always a border of  $x$ , for all  $x \in A^+$ . The empty string  $\epsilon$  itself has no border.

The following example illustrates how the shift distance in the Knuth-Morris-Pratt algorithm is determined using the notion of the border of a string .

Example:

**0 1 2 3 4 5 6 7 8 9 ...**

a b c a b c a b d

a b c a b d

a b c a b d

The symbols at positions 0, ..., 4 have matched. Comparison c-d at position 5 yields a mismatch. The pattern can be shifted by 3 positions, and comparisons are resumed at position 5.

The shift distance is determined by the widest border of the matching prefix of  $p$ . In this example, the matching prefix is abcab, its length is  $j = 5$ . Its widest border is ab of width  $b = 2$ . The shift distance is  $j - b = 5 - 2 = 3$ .

In the preprocessing phase, the width of the widest border of each prefix of the pattern is determined. Then in the search phase, the shift distance can be computed according to the prefix that has matched[26].

Theorem [26] : Let  $r, s$  be borders of a string  $x$ , where  $|r| < |s|$ . Then  $r$  is a border of  $s$ .  
 Proof: Figure 1 shows a string  $x$  with borders  $r$  and  $s$ . Since  $r$  is a prefix of  $x$ , it is also a proper prefix of  $s$ , because it is shorter than  $s$ . But  $r$  is also a suffix of  $x$  and, therefore, proper suffix of  $s$ . Thus  $r$  is a border of  $s$ .



Figure 4.2: Borders  $r, s$  of a string  $x$

Definition: Let  $x$  be a string and  $a \in A$  a symbol. A border  $r$  of  $x$  can be extended by  $a$ , if  $ra$  is a border of  $xa$ .

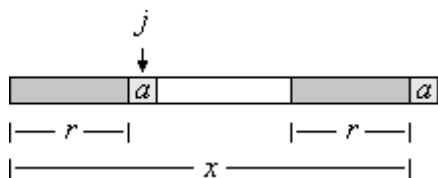


Figure 4.3: Extension of a border

Figure 3 shows that a border  $r$  of width  $j$  of  $x$  can be extended by  $a$ , if  $x_j = a$ .

In the preprocessing phase an array  $b$  of length  $m+1$  is computed. Each entry  $b[i]$  contains the width of the widest border of the prefix of length  $i$  of the pattern ( $i = 0, \dots, m$ ). Since the prefix  $\epsilon$  of length  $i = 0$  has no border, we set  $b[0] = -1$ .

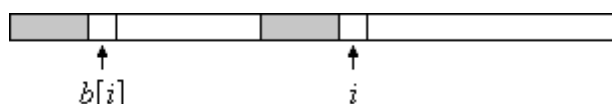


Figure 4.4: Prefix of length  $i$  of the pattern with border of width  $b[i]$

Provided that the values  $b[0], \dots, b[i]$  are already known, the value of  $b[i+1]$  is computed by checking if a border of the prefix  $p_0 \dots p_{i-1}$  can be extended by symbol  $p_i$ . This is the case if  $p_{b[i]} = p_i$  (Figure 3). The borders to be examined are obtained in decreasing order from the values  $b[i], b[b[i]]$  etc.

The preprocessing algorithm comprises a loop with a variable  $j$  assuming these values. A border of width  $j$  can be extended by  $p_i$ , if  $p_j = p_i$ . If not, the next-widest border is examined by setting  $j = b[j]$ . The loop terminates at the latest if no border can be extended ( $j = -1$ ).

After increasing  $j$  by the statement  $j++$  in each case  $j$  is the width of the widest border of  $p_0 \dots p_i$ . This value is written to  $b[i+1]$  (to  $b[i]$  after increasing  $i$  by the statement  $i++$ ) [26].

#### Algorithm 4.1: KMP Preprocessing algorithm :

Let  $m$  = size of the pattern,  $b$ = the border,  $p$ =the pattern,

```
void kmpPreprocess ()
{
    int i=0, j=-1;
    b[i]=j;
    while (i<m)
    {
        while (j>=0 && p[i]!=p[j]) j=b[j];
        i++; j++;
        b[i]=j;
    }
}
```

Example: For pattern  $p = ababaa$  the widths of the borders in array  $b$  have the following values. For instance we have  $b[5] = 3$ , since the prefix  $ababa$  of length 5 has a border of width 3.

$j$ :	0	1	2	3	4	5	6
$p[j]$ :	a	b	a	b	a	a	
$b[j]$ :	-1	0	0	1	2	3	

Conceptually, the above preprocessing algorithm could be applied to the string  $pt$  instead of  $p$ . If borders up to a width of  $m$  are computed only, then a border of width  $m$  of some prefix  $x$  of  $pt$  corresponds to a match of the pattern in  $t$  (provided that the border is not self-overlapping) (Figure 4.5)[26].



Figure 4.5: *Border of length  $m$  of a prefix  $x$  of  $pt$*

This explains the similarity between the preprocessing algorithm and the following searching algorithm.

#### Algorithm 4.2: KMP Searching algorithm :

Let  $n$ = size of the text,  $m$ = size of the pattern,  $b$ = the border,  $p$ =the pattern

```
void kmpSearch()
{
    int i=0, j=0;
    while (i<n)
    {
        while (j>=0 && t[i]!=p[j]) j=b[j];
        i++; j++;
        if (j==m)
        {
            report(i-j);
            j=b[j];
        }
    }
}
```

When in the inner while loop a mismatch at position  $j$  occurs, the widest border of the matching prefix of length  $j$  of the pattern is considered (Figure 4.5). Resuming comparisons at position  $b[j]$ , the width of the border, yields a shift of the pattern such that the border matches. If again a mismatch occurs, the next-widest border is considered, and so on, until there is no border left ( $j = -1$ ) or the next symbol matches. Then we have a new matching prefix of the pattern and continue with the outer while loop.

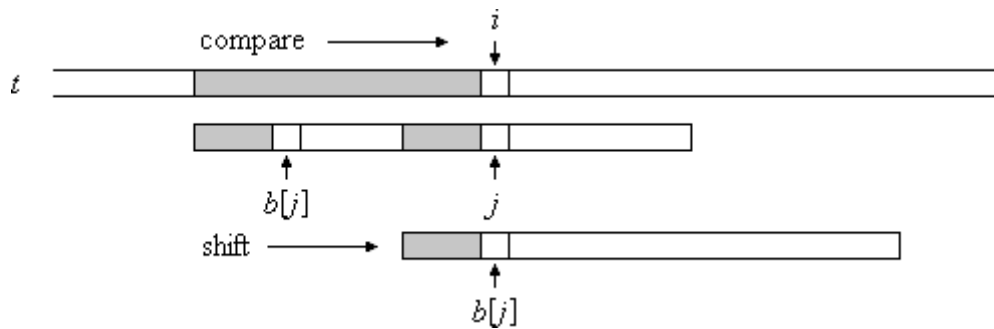


Figure 4.6: Shift of the pattern when a mismatch at position  $j$  occurs

If all  $m$  symbols of the pattern have matched the corresponding text window ( $j = m$ ), a function *report* is called for reporting the match at position  $i-j$ . Afterwards, the pattern is shifted as far as its widest border allows.

In the following example the comparisons performed by the searching algorithm are shown.

Example:

```

0 1 2 3 4 5 6 7 8 9 ...
a b a b b a b a a
a b a b a c
  a b a b a c
    a b a b a c
      a b a b a c
        a b a b a c

```

The inner while loop of the preprocessing algorithm decreases the value of  $j$  by at least 1, since  $b[j] < j$ . The loop terminates at the latest when  $j = -1$ , therefore it can decrease the value of  $j$  at most as often as it has been increased previously by  $j++$ . Since  $j++$  is executed in the outer loop exactly  $m$  times, the overall number of executions of the inner while loop is limited to  $m$ . The preprocessing algorithm therefore requires  $O(m)$  steps [26]. From similar arguments it follows that the searching algorithm requires  $O(n)$  steps. The above example illustrates this. The whole staircase is at most as wide as it is high; therefore at most  $2n$  comparisons are performed [26]. Since  $m \leq n$  the overall complexity of the Knuth-Morris-Pratt algorithm is in  $O(n)$  [9].

## Chapter Five

### The proposed Algorithm

#### 5.1 Algorithm Description

The algorithm (AFALS-N) finds the regions of highest similarity between two sequences, thus generating one or more islands of matches or sub-alignments in the aligned sequences.

#### Steps of alignment algorithm:

1-Build a complete list of all words in one sequence and make this into a table.

2-For each word in the second sequence a simple lookup in the table shows every match in the first sequence.

3-A negative score/weight is given to mismatches. Therefore, score drops (from initial zero value) as more and more mismatches are added .Hence the score will rise in a region of high similarity and then fall outside this region.

Scoring function for gapped alignment:

$$f = \Sigma \text{ match score} - (\text{mismatch score} + \text{gap score}) \dots \dots \dots (5.1)$$

4-The alignments are produced by starting at the highest scoring positions in the scoring matrix and trace the path from those positions up to a box that scores zero.

The next figure shows the input and output of AFALS-N algorithm .It has two inputs which are a DNA sequences and 3 approximate local similarity strings.

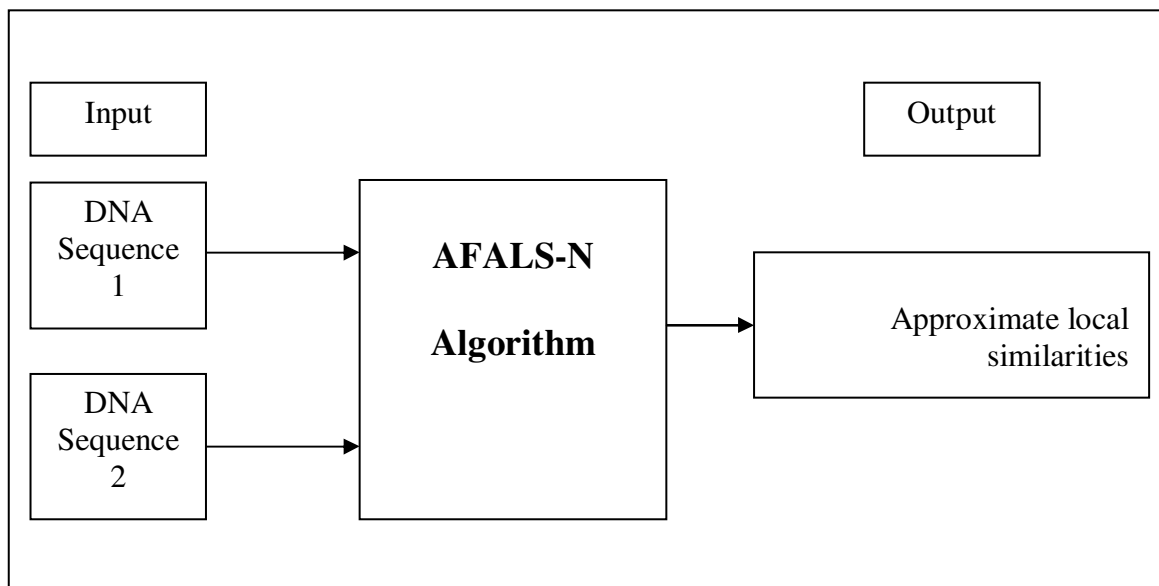


Figure 5.1: AFALS-N Algorithm

Description of input and output

**1-Input:**

DNA Sequence 1: S with size n.

$$S = \{ i_1, i_2, \dots, i_n \}$$

DNA Sequence 2: T with size m.

$$T = \{ j_1, j_2, \dots, j_m \}$$

**2-Output:**

Three alignment or approximated substrings

Word size =  $w = 9 \text{ \& } 11$

$$\text{Possible words} = \lfloor m/w \rfloor \dots\dots\dots(5.2)$$

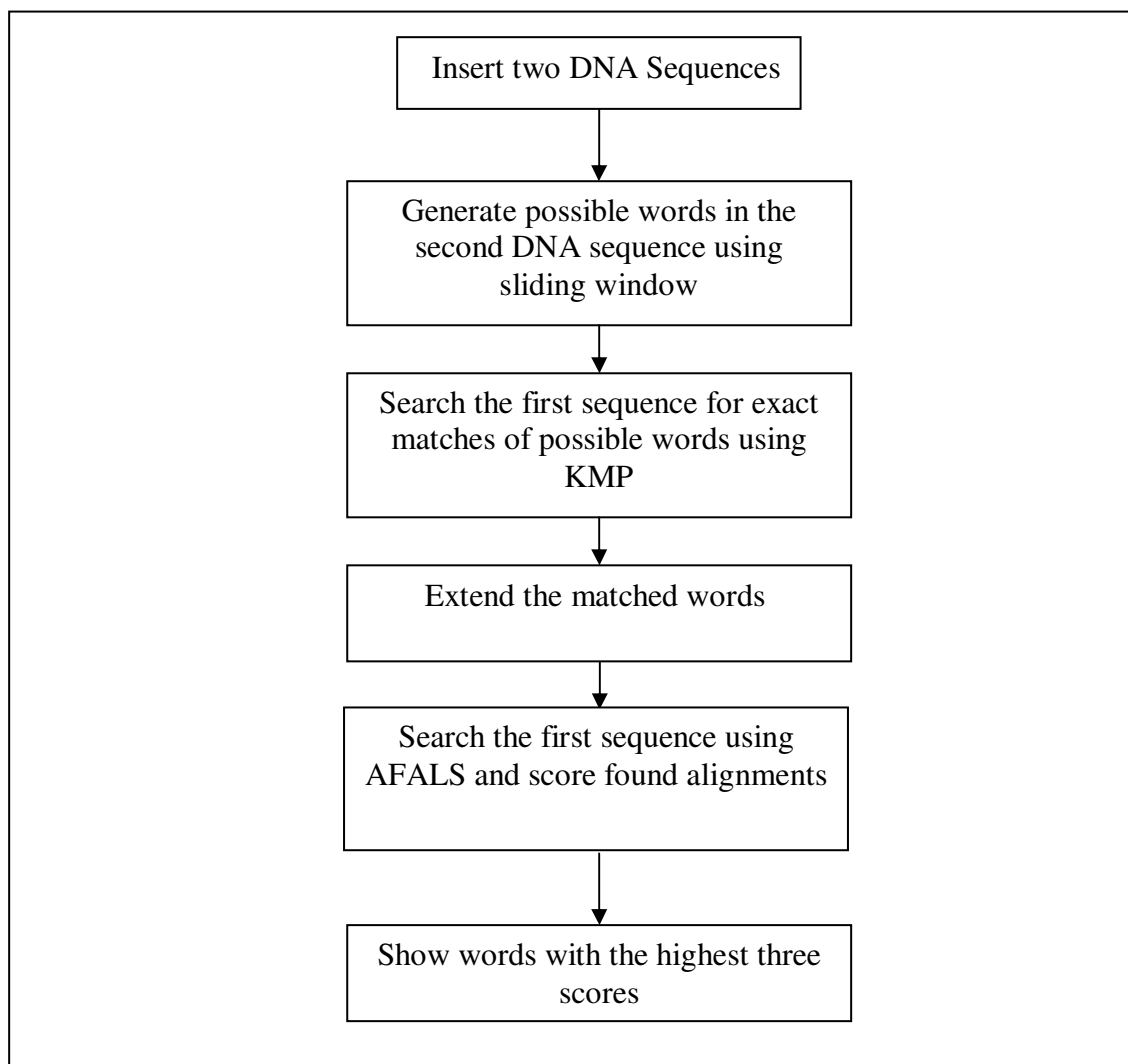


Figure 5.2:AFALS Algorithm Flow chart.

AFALS-N produce local alignments in four phases .In the first phase , the sequence to be compared is partitioned .The second phase KMP is used to find exact matches . In the third phase the candidate words are extended using gaps . Finally in the fourth phase the maximum three alignments are selected and shown as an output of the algorithm.

### Phase 1: Data Partitioning

We partition the second DNA sequence (T with size m) to Z substrings depending on the next formula :  $Z = m/11$  where 11 is the word size .

Next is an example of data partition where a sequence of size 44 is partitioned to 4 strings with size = 11 .



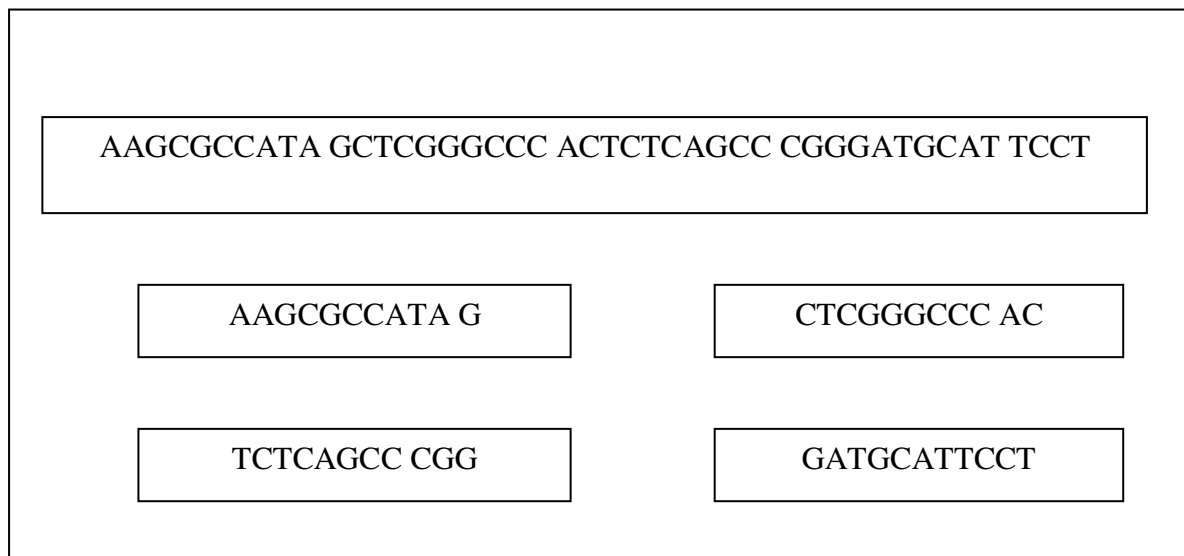


Figure5.3 : Data partition example .

### Phase 2: KMP

The inputs to the KMP algorithm are the substrings generated by phase1. And the outputs are the candidate seeds with their indexes (the index were the KMP start to find the seed).

The output of this phase looks like as shown in the following figure .

<i>Candidate seeds</i>	<i>indexes</i>
AAGCGCCATA G	1
TCTCAGCC CGG	22

Figure5.4 : KMP output example .

### Phase 3: Gap Extension

A gap is a maximal consecutive run of spaces in a single string of a given alignment. It corresponds to an atomic insertion or deletion of a substring [26].Gap extension is the process of inserting gaps wherever there is a mismatch.

The penalty of gap is -5 and for gap extension it is -1 . When there is a mismatch gap is inserted and score is decremented .If the score falls than K the extension will stop and the word is discarded from candidates .

K is the similarity score which must not be less than 90% .The allowable number of mismatches is 10 % .That because the mean number of wrongly inferred indels and gap character states increases with substitution rate for closely related sequences , the error segments are short and frequently result from a single indel being erroneously positioned . As the two sequences farther diverge , the errors multiply .At the same time, neighboring indels in the true alignment being inferred with one another produce several segments where several indels are simultaneously misplaced . At the higher divergence rates , the error segments get longer and longer , with relatively short intervening correct segments , until almost the whole reconstructed alignment consist of error segments [26].

If the KMP finds the word it saves it in a table with the specified index. At phase 3 gap extension will start .It will use the indexes that have been saved in the phase 2.

Here is the pseudo code for this phase

```

Algorithm 5.1: AFALS-N Gap extension algorithm:
Let w := 11 //word size
Let score is the alignment score //initialized to word size
Let k := 0 //number of allowed mismatch
Get List of candidate words index from Kmp result table
For index :=1 to K>10% of the score //number of matches
  If character[index+w]:=character[seedIndex+w]
    score+=1
    gap_start := false
  If character[index+w]!:=character[seedIndex+w]
    score +=-1
    If gap_start := false
    score+=-5
    gap_start :=true
  Else Score+= -1
Next index

```

## Phase 4: Output Selection

The aligned substrings that have the maximum three score will be shown in the result screen.

### Example :

The input of the AFALS-N algorithm is the next two DNA sequences:

Sequence 1: aaacctggagcacgaacctgccacccccccccgggttcag

Sequence 2: aaacctggagcaaaacctgcc

#### *Phase 1 output:*

In phase 1 the second sequence is partitioned to seeds of size 11 as shown next.

Seed1: aaacctggagc size=11

Seed2: aaaaacctgcc size=11

#### *Phase 2 output:*

In phase 2 kmp searches for seeds in the first sequence and save the index where the match starts. In this example it only found one match for the first seed and save the index for the next phase use.

aaacctggagc index : 1

#### *Phase 3 output:*

*In gap extension phase wherever there is a mismatch a gap is inserted and the penalty of a mismatch and a gap is added to the score.*

```
aaacctggagcacgaacctgccacccccccccgggttcag
aaacctggagca- -aacctgcc
```

score = 11

score+= 1 match

score+=-1 mismatch

score+=-5 gap open

score+=-1 mismatch

score+=-1 gap extension

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

score+= 1 match

Score = 12

The output of this phase is:  
aaacctggagca- -aacctgcc  
and its score (12)

*Phase 4 output :*

Since there is one alignment it is the only output with its score .

### **Algorithm Time and space complexities**

In the phase1 the algorithm needs  $m/w$  space , same in the phace2 and phace3 , so the space complexity for AFALS-N algorithm is  $O(m)$  since  $w$  is a constant.

Time complexity for KMP is  $O(n)$ , and for phase3 is  $O(zm)$  w here  $z$  is the number of candidate words . So the overall AFALS-N complexity is  $O(n +z m)$ .

## Chapter Six

### AFALS-N Software

AFALS-N software is a demonstration for the AFALS-N algorithm . It was built using java .In the next pages we presented to the development model , the implementation of this software , and the interface screen shots .

#### 6.1 Software Development Model

A software process is a framework of activities that are required to develop software. A software process model is a development strategy that encompasses the process, methods, tools and generic phases used during the development of software In other words, a software process model is an abstraction, which is used to describe the steps involved in a software process [21] .

We have chosen to use a modified Waterfall Model as a standard software process model that we can follow for the development of this project.

The waterfall model is the classic model of software engineering. It has deficiencies, but it serves as a baseline for many other lifecycle models.The pure waterfall lifecycle consists of several non-overlapping stages, as shown in Figure 6.1. It begins with the software concept and continues through requirements analysis, architectural design, detailed design, coding, testing, and maintenance [30].

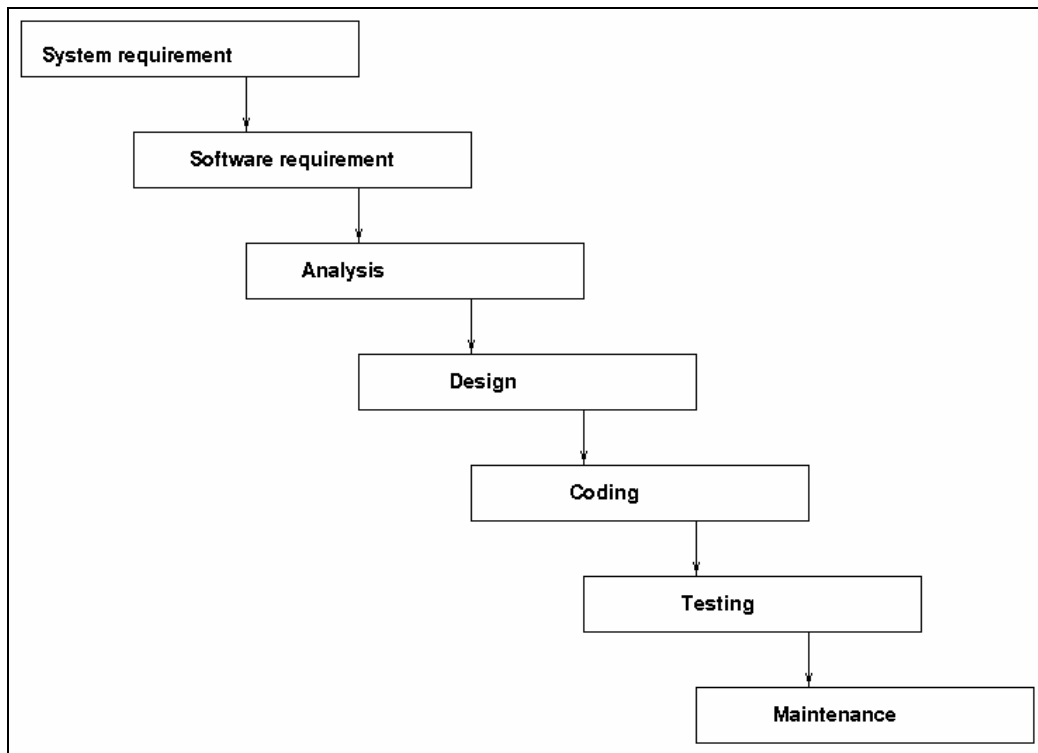


Figure 6.1: Classical Waterfall Model [30].

The waterfall model does not prohibit returning to an earlier phase, for example, from the design phase to the requirements phase. This leads to many versions of modified waterfall model [30].

These modifications tend to focus on allowing some of the stages to overlap, reducing the documentation requirements, and reducing the cost of returning to earlier stages to revise them. Another common modification is to incorporate prototyping into the requirements phases [21].

Overlapping stages such as requirements and design make it possible to feed information from the design phase back into the requirements.

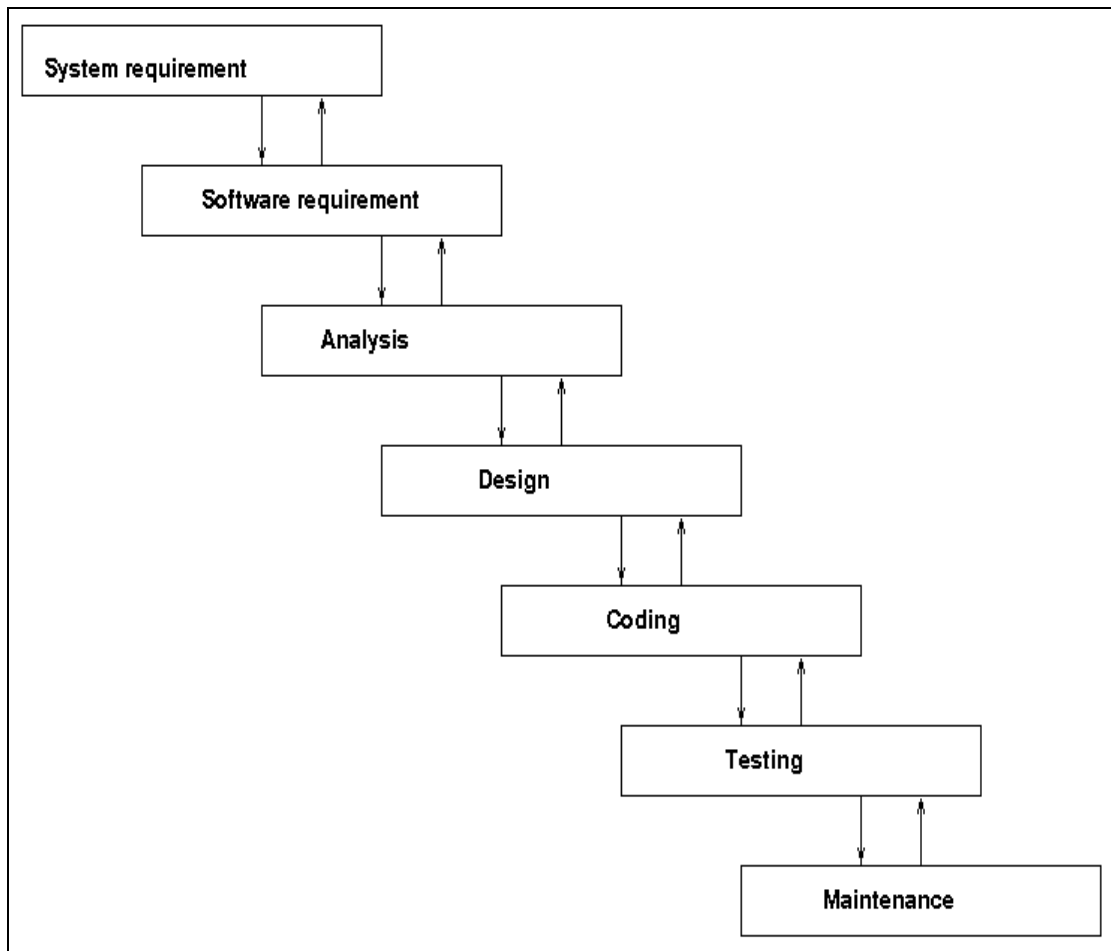


Figure 6.2: Modified Waterfall Model [30].

- System requirements—Establishes the components for building the system. This includes the hardware requirements (number of channels, acquisition speed, and so on), software tools, and other necessary components.
- Software requirements—Concentrates on the expectations for software functionality. You identify which of the system requirements the software affects. Requirements analysis might include determining interaction needed with other applications and databases, performance requirements, user interface requirements, and so on.
- Architectural design—determines the software framework of a system to meet the specified requirements. The design defines the major components and the interaction of those components, but it does not define the structure of each

component. Also determine the external interfaces and tools to use in the project.

- Detailed design—examines the software components defined in the architectural design stage and produces a specification for how each component is implemented.
- Coding—Implements the detailed design specification.
- Testing—determines whether the software meets the specified requirements and finds any errors present in the code.

We have used black box testing. Black Box Testing is not a type of testing; it instead is a testing strategy, which does not need any knowledge of internal design or code etc. As the name "black box" suggests, no knowledge of internal logic or code structure is required. The types of testing under this strategy are totally based/focused on the testing for requirements and functionality of the work product/software application. Black box testing is sometimes also called as "Opaque Testing", "Functional/ Behavioral Testing" and "Closed Box Testing" [30].

The base of the Black box testing strategy lies in the selection of appropriate data as per functionality and testing it against the functional specifications in order to check for normal and abnormal behavior of the system. Nowadays, it is becoming common to route the Testing work to a third party as the developer of the system knows too much of the internal logic and coding of the system, which makes it unfit to test the application by the developer [21]. In order to implement Black Box Testing Strategy, the tester is needed to be thorough with the requirement specifications of the system and as a user, should know, how the system should behave in response to the particular action.

- Maintenance—Perform as needed to deal with problems and enhancement requests after the software is released.



## 6.2 Implementation

The algorithm implementation has been done using Java language, because of the built in String classes and methods and its platform independent . Using a cross-platform and object oriented ease of development of the Java programming language we have built a simulation to AFALS-N algorithm .

We choose the BlueJ version 1.3.5 as a coding environment . BlueJ is an integrated Java environment specifically designed for introductory teaching.

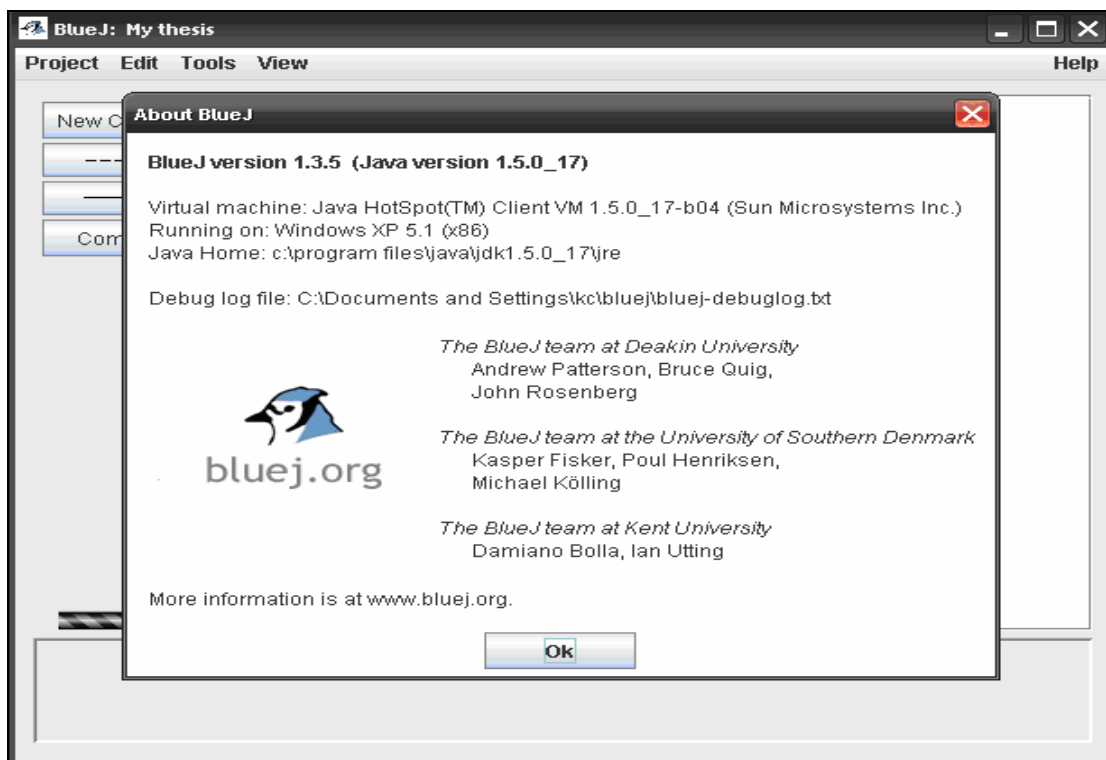


Figure 6.3: BlueJ Screen .

BlueJ supports:

- fully integrated environment
- graphical class structure display
- graphical and textual editing
- built-in editor, compiler, virtual machine, debugger, etc.
- easy-to-use interface, ideal for beginners

- interactive object creation
- interactive object calls
- interactive testing
- incremental application development

The AFALS-N software has two classes ; The MainWindow class which has the main functionality and components , and the Test class which has been used to create objects from MainWindow class .

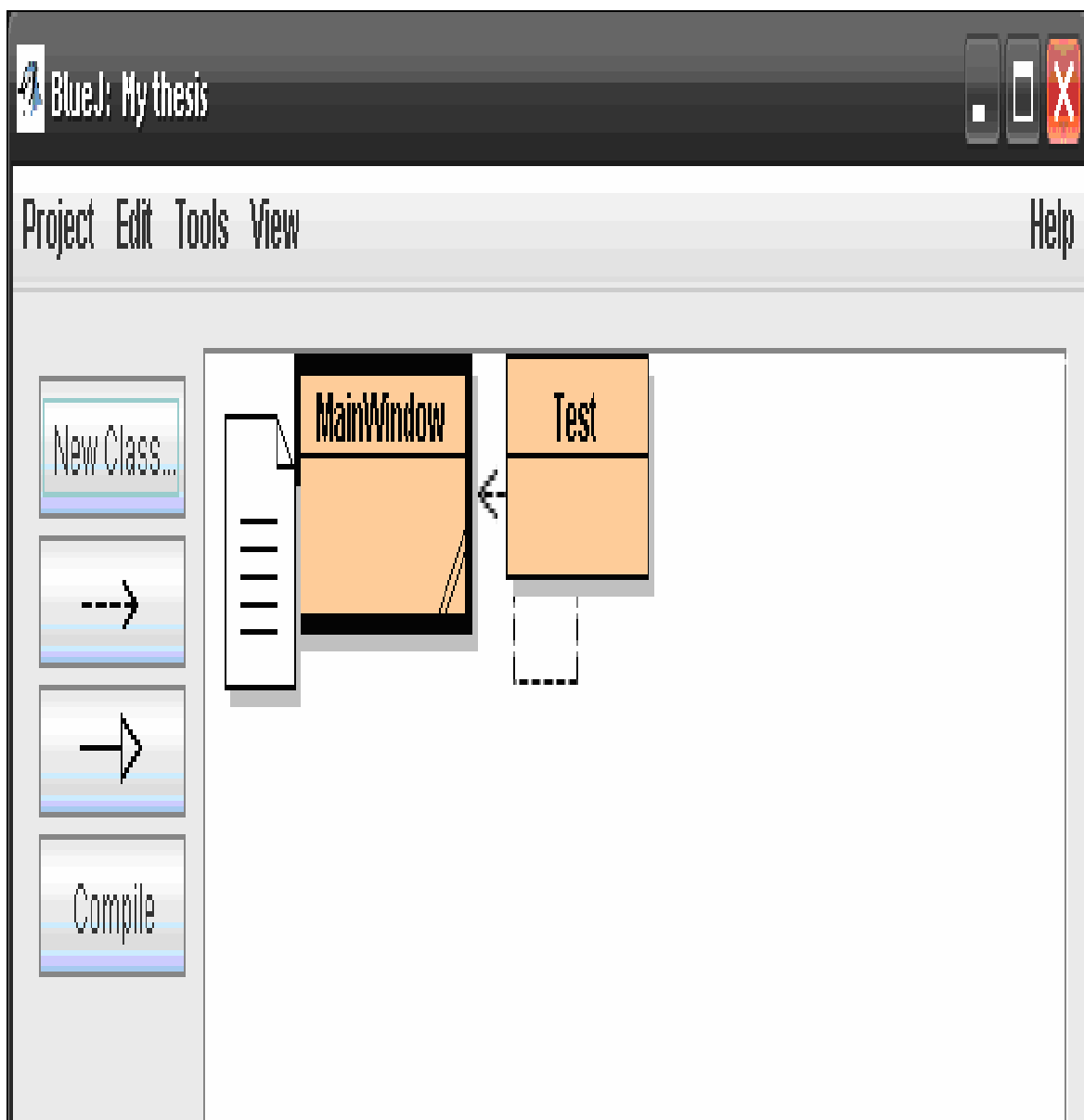
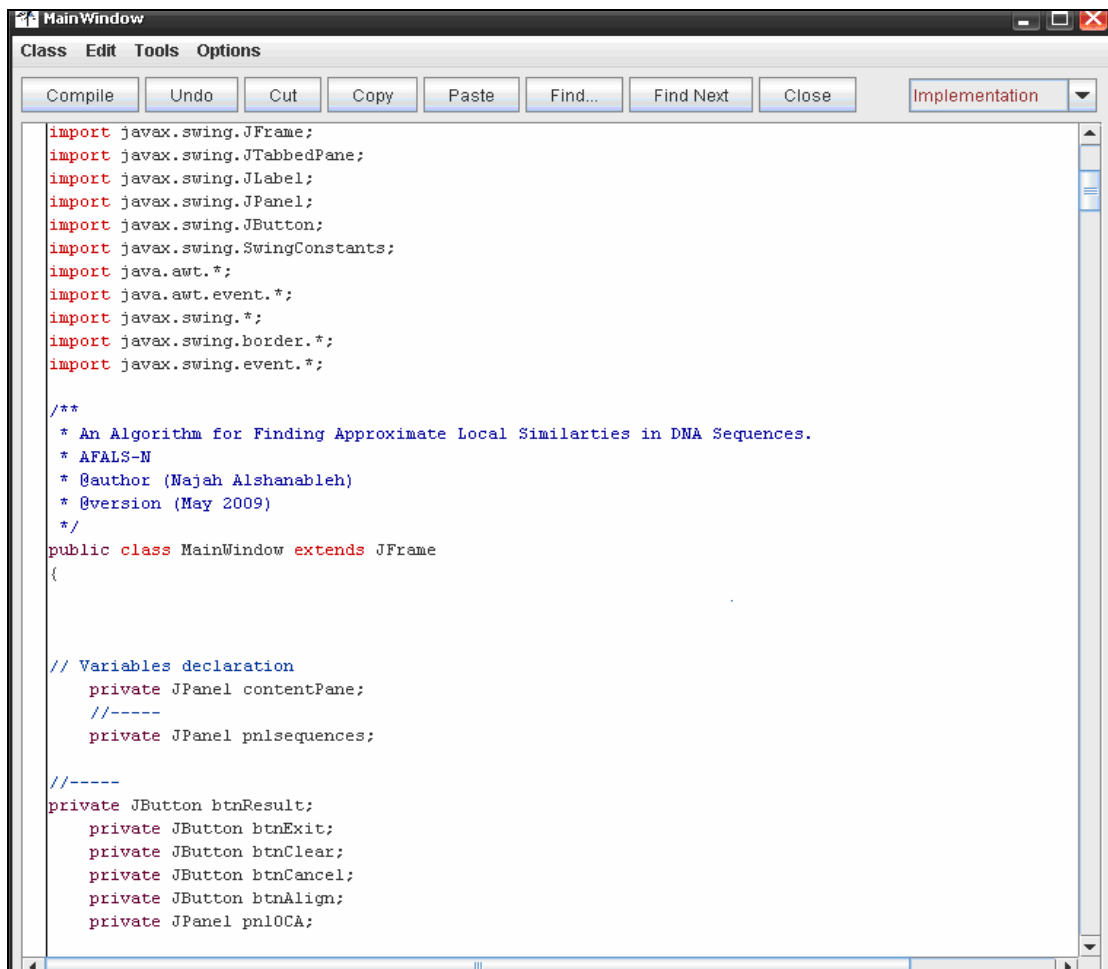


Figure 6.4 : Classes in AFALS-N



```

import javax.swing.JFrame;
import javax.swing.JTabbedPane;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JButton;
import javax.swing.SwingConstants;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;

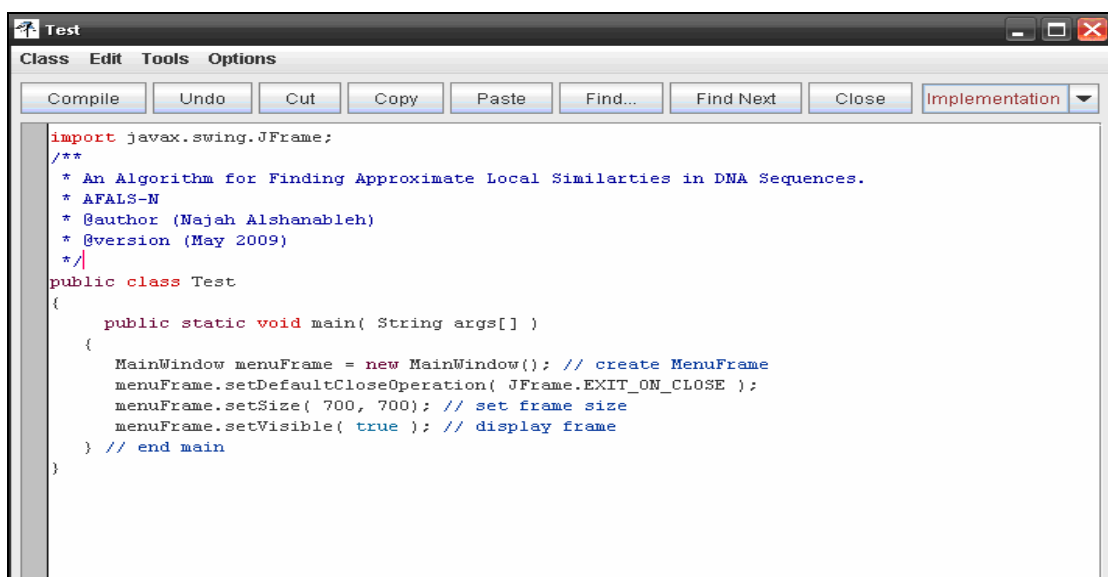
/**
 * An Algorithm for Finding Approximate Local Similarities in DNA Sequences.
 * AFALS-N
 * @author (Najah AlshanaBLEH)
 * @version (May 2009)
 */
public class MainWindow extends JFrame
{

    // Variables declaration
    private JPanel contentPane;
    //-----
    private JPanel pnlsequences;

    //-----
    private JButton btnResult;
    private JButton btnExit;
    private JButton btnClear;
    private JButton btnCancel;
    private JButton btnAlign;
    private JPanel pnlOCA;

```

Figure 6.5 : MainWindow Class .



```

import javax.swing.JFrame;
/**
 * An Algorithm for Finding Approximate Local Similarities in DNA Sequences.
 * AFALS-N
 * @author (Najah AlshanaBLEH)
 * @version (May 2009)
 */
public class Test
{
    public static void main( String args[] )
    {
        MainWindow menuFrame = new MainWindow(); // create MenuFrame
        menuFrame.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
        menuFrame.setSize( 700, 700); // set frame size
        menuFrame.setVisible( true ); // display frame
    } // end main
}

```

Figure 6.6 : Test Class .

### 6.3 User Interface Screens

The user interface of the system has been designed based on the description of the AFALS-N algorithm . A description of the user interface design and coding is shown below and in the next few pages .

Building the user interface in java is difficult since every thing need programming .The basic user interface component is the “form”. It contains all the controls that form the shape of each screen. During the development phase, built in controls (that are provided by the programming language itself) have been used to form up the final shape of the system screens. Those include text boxes, buttons, labels, panels and tapped pane.

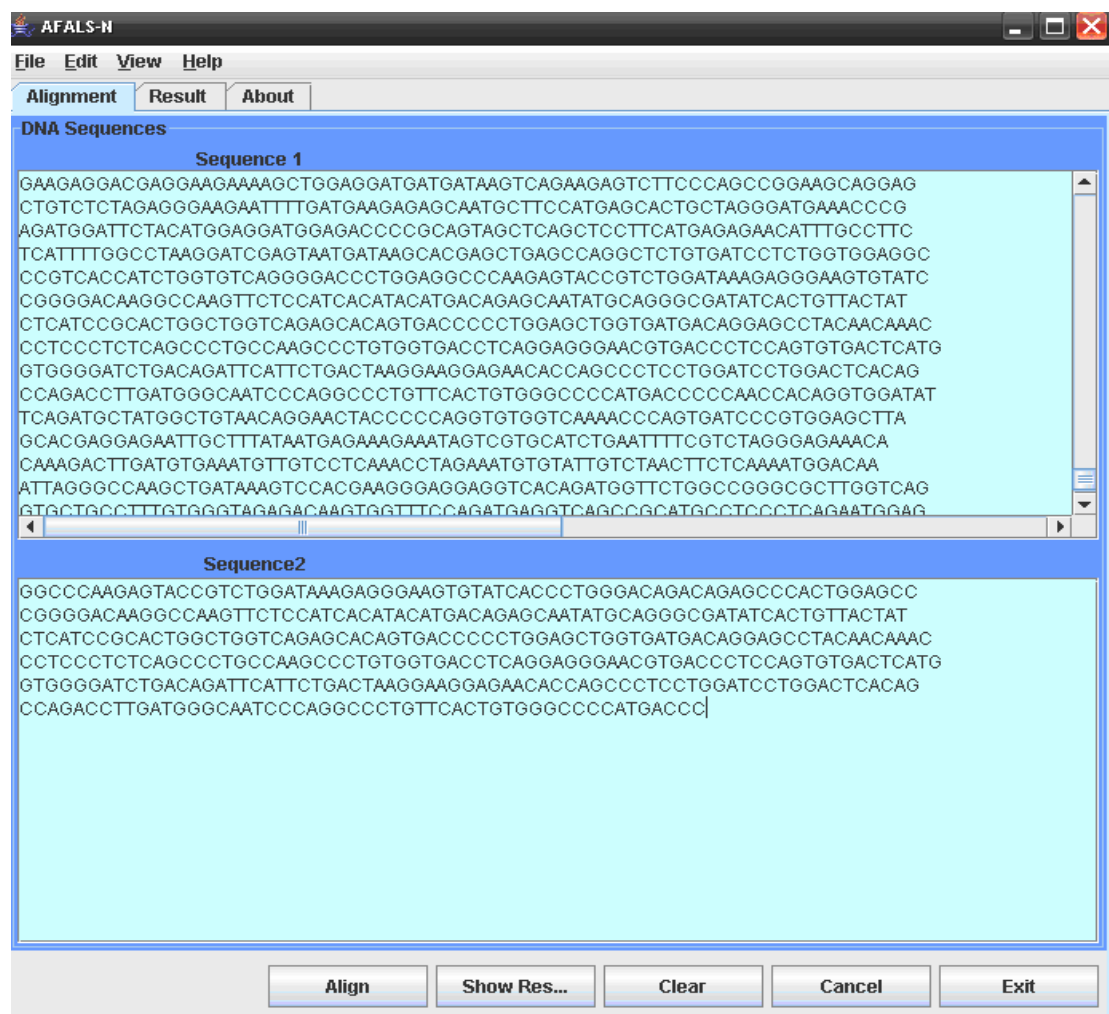


Figure 6.7 : AFALS-N screen .

User Interface Parts :

A-Menu bar

The menu bar contains four menus :

1-File Menu

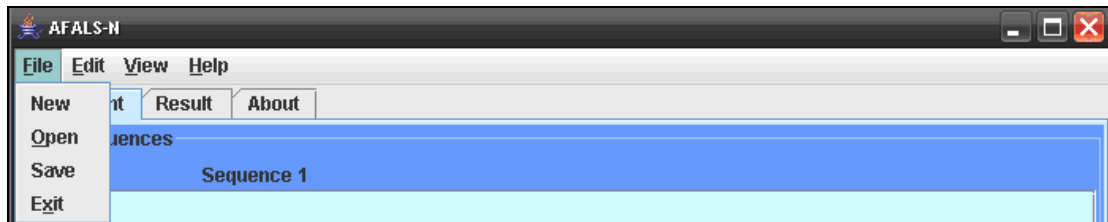


Figure 6.8 : File menu

2-Edit Menu

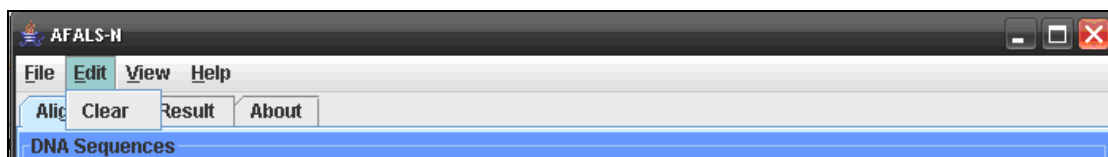


Figure 6.9 : Edit menu

3-View Menu

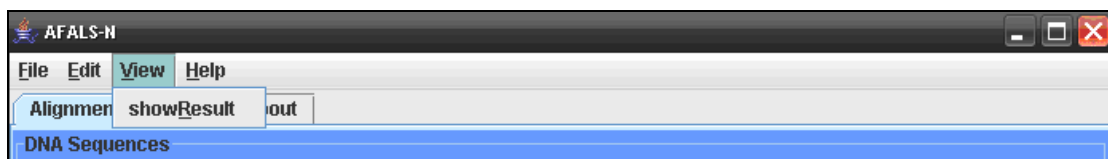


Figure 6.10 : View menu

4-Help Menu

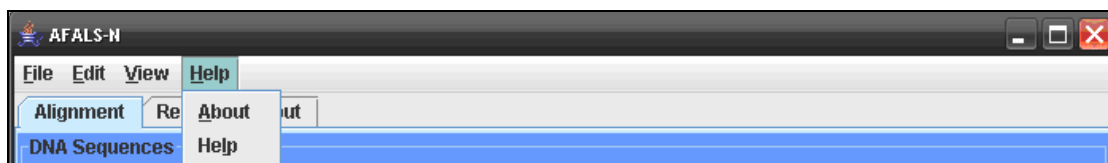


Figure 6.11 : Help menu

## B-Alignment Tap

This tap is the default tap shown once we run the AFALS-N software . It consists of two text area for the two DNA inputs .It also has five buttons for the basic operations .

- Align Button : Its start the alignment operation once its clicked.
- Show result Button: Shows the result .
- Clear Button : Clear the text areas.
- Cancel Button: Stop the alignment operation before it finished.
- Exit Button : Close the software screen .

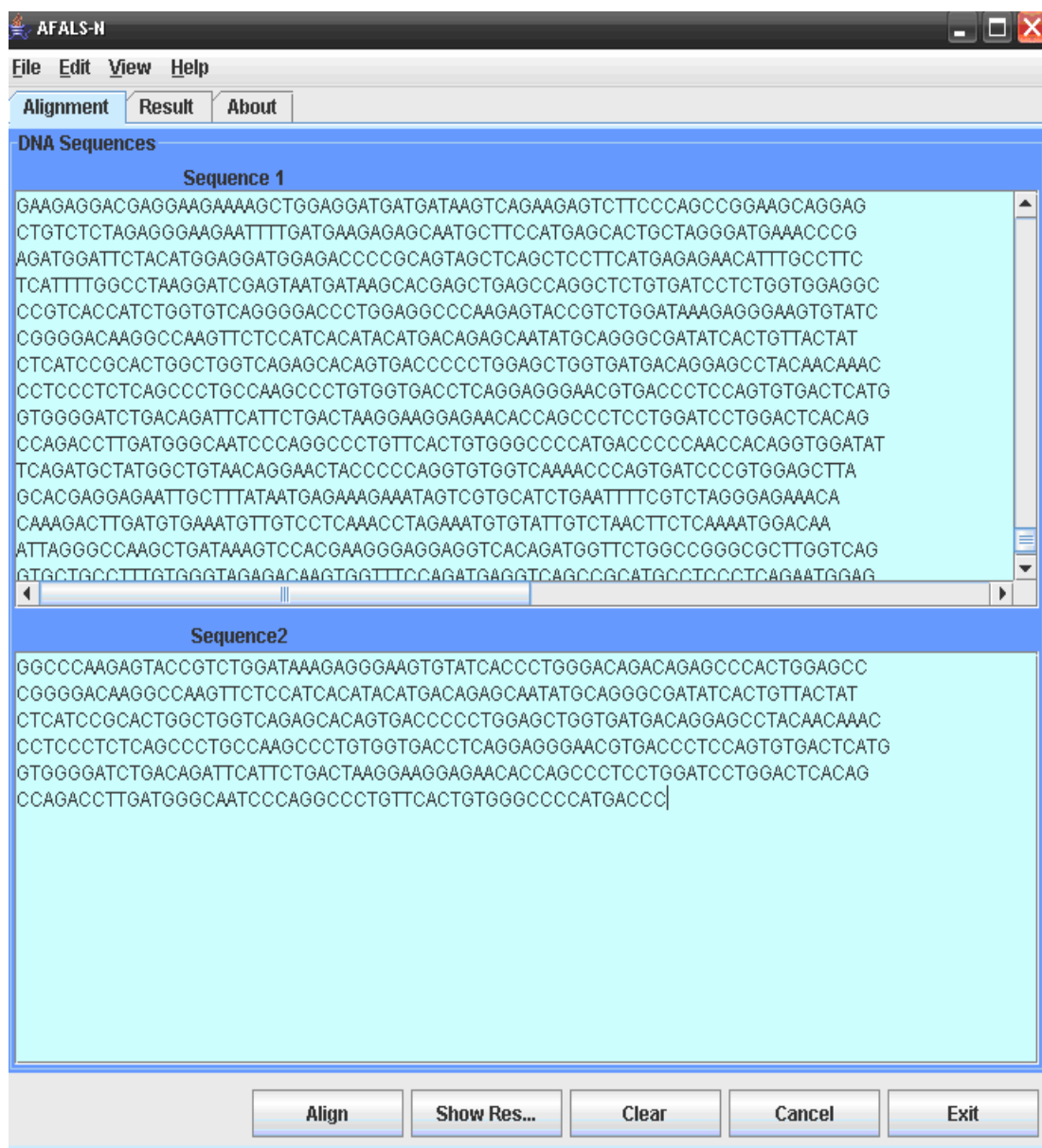


Figure 6.12 : Alignment tap

## B-Result tap

This is the second tap that shows the resulted alignment . It consist of three text boxes with its aligned score .

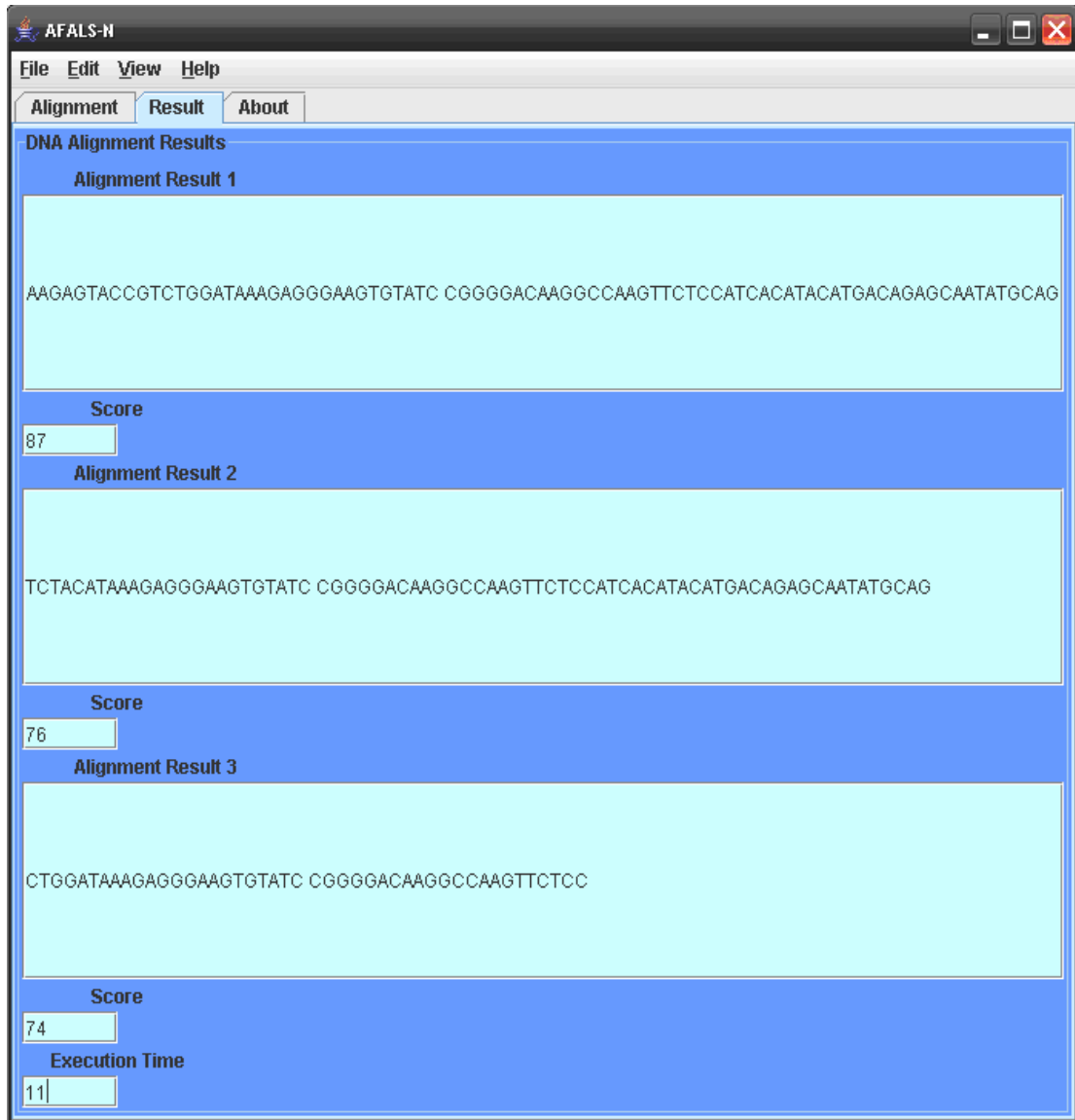


Figure 6.13 : Result tap

## C-About Tap

It's the final tap of AFALS-N software that shows general info about the software .

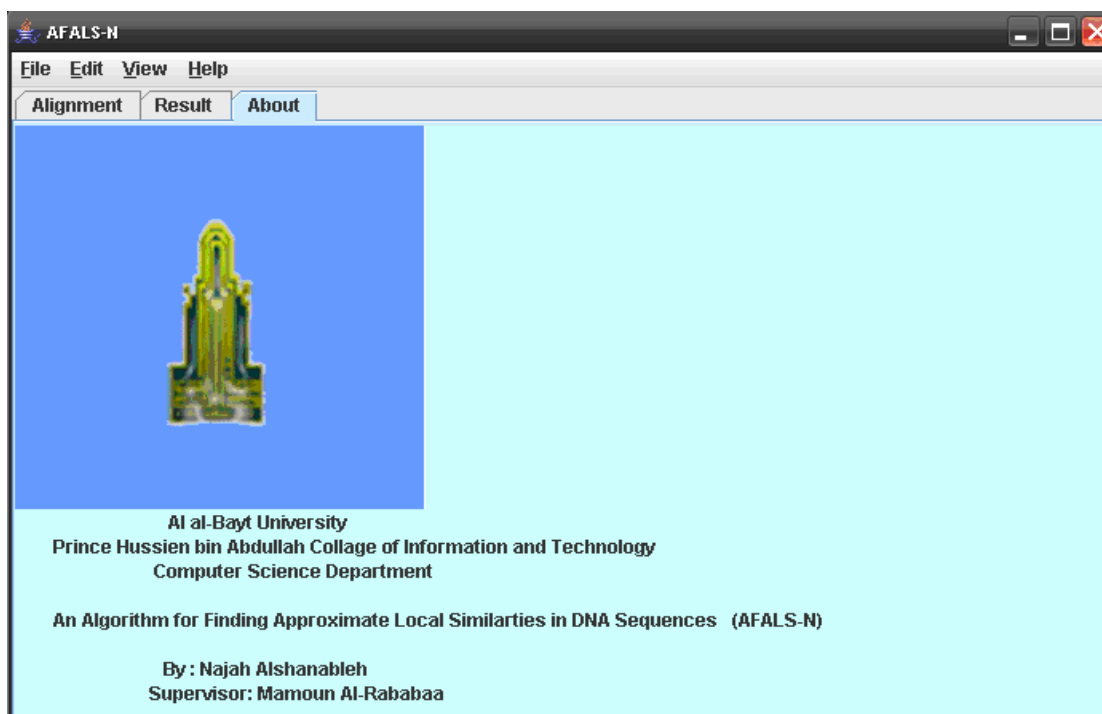


Figure 6.14 : About tap .

### D-Terminal Window

This terminal window shows the elapsed time for the alignment process .

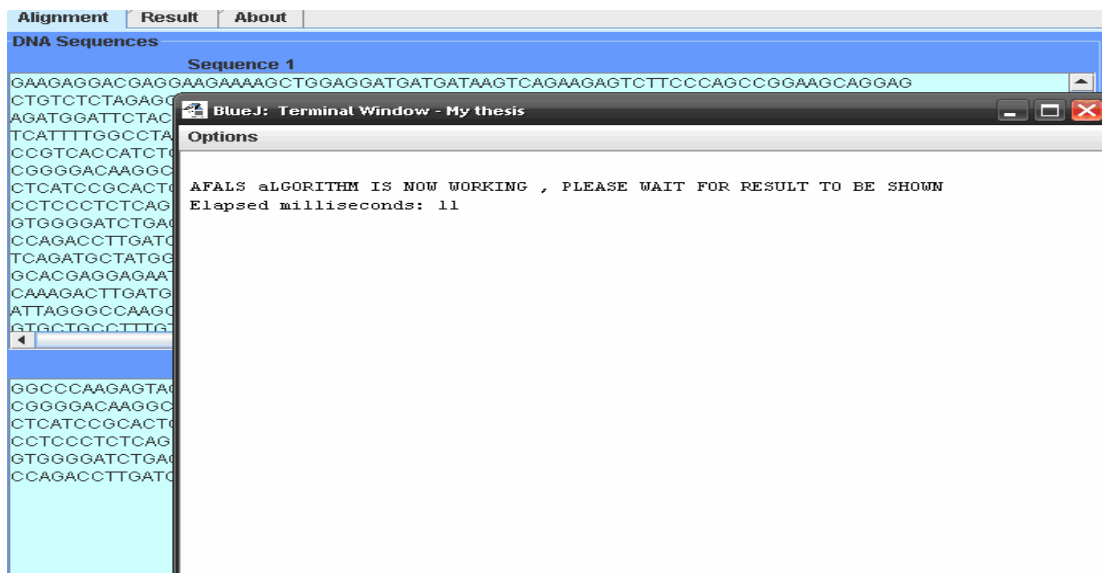


Figure 6.15 : Terminal Window .



## Chapter Seven

### Results and discussion

In this chapter we analyze the simulation results that we obtained for different sequences that we consider so as to measure the performance of the proposed algorithm. We have tested the algorithm for different DNA sequences and compare it with PatternHunter results.

#### 7.1 Test environment

The proposed algorithm was implemented in Java .Real DNA sequences obtained from the NCBI (National Center for Biotechnology Information), web page (<http://www.ncbi.nlm.nih.gov/>) were used in the tests . Sample of the sequences are presented in table 7.1 below.

Table 7.1: Organisms compared

Approx size	Real size	Seq number	Name
1 kBP	1440 BP	NC_004991	Acetobacter Pasteurians
1 kBP	1743 BP	NC_005026	Bacteroides Fragilis
10 kBP	10,035 BP	AF133821	HIV-1 isolate MB2059 from Kenya
10 kBP	10,280 BP	AY352275	HIV-1 isolate SF33 from USA
50 kBP	56,574 BP	AF494279	Chaetospheridium globosum
50 kBP	57,473 BP	NC_001715	Allomyces Macogynus
150 kBP	162,114 BP	NC_000898	Human Herpesvirus 6B
150 kBP	171,823 BP	NC_007605	Human Herpesvirus 4
500 kBP	542,869 BP	NC_003064	Agrobacterium tumefaciens
500 kBP	563,165 BP	NC_000914	Rhizobium sp.
1MBP	1,044,459 BP	CP000051	Chlamydia trachomatis
1MBP	1,072,950 BP	AE002160	Chlamydia muridarum
3MBP	3,147,090 BP	BA000035	Corynebacterium efficiens
3MBP	3,282,708 BP	BX927147	Corynebacterium glutamicum

## 7.2 Sensitivity analysis

Sensitivity analysis is the study of how the variation in the output of an algorithm can be apportioned, qualitatively or quantitatively, to different sources of variation in the input of that algorithm.

In order to analyze AFALS-N sensitivity we have selected two known mutations with known DNA as an input to the algorithm. These mutations are FLT3 and BRCA.

Mutation detection is increasingly undertaken as a tool for a wide spectrum of research especially in cancer diseases, disease association and clinical diagnostics. The pharmaceutical industry spends billions of dollars to locate the mutated genes associated with particular diseases.[13]



Figure 7.1 Affected person mutation [13].

An example of such mutations is the FLT3 (Fms-related tyrosine kinase 3) mutation which responsible for leukemia disease. FLT3 is the most commonly mutated gene in human acute myeloid leukemia (AML) and has been implicated in its pathogenesis [23].

The clinical identification of *FLT3* mutations in a prospective manner will yield important information about the incidence and natural history of *FLT3* mutations in AML [14].

In addition, identification of *FLT3* mutations is likely to become important for optimization of patient care. Because *FLT3* ITD mutations portend a worse prognosis, it

has been proposed that patients testing positive for a *FLT3* mutation may benefit from aggressive up-front treatment regimens such as an allogeneic bone marrow transplantation. On-going clinical trials will determine whether AML patients with *FLT3* mutations will also benefit from novel therapeutic strategies that target and inhibit *FLT3* tyrosine kinase activity [14].



Figure 7.2: Leukemia Mutations [20].

Germline mutations in breast cancer susceptibility genes, *BRCA1* and *BRCA2*, are responsible for a substantial proportion of high-risk breast and breast/ovarian cancer families [6].

Breast cancer is the most commonly diagnosed cancer in women in world today. A family history of the disease in a first degree relative significantly increases the risk of disease. A segregation analysis demonstrated the existence of an autosomal dominant pattern of inheritance accounting for 5-10% of breast cancer cases [6].

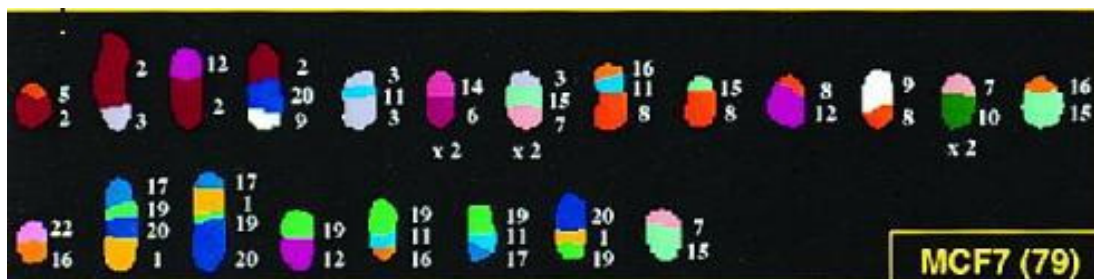


Figure 7.3: Breast Cancer Mutation [6].

We have used 50 DNA sample of infected people who has a leukemia or a breast cancer. AFALS-N was able to catch the mutation for 33 person .That means that the sensitivity of this algorithm is 0.66 .

$$\text{Sensitivity} = a/b \dots\dots\dots( 7.1)$$

where a is the number of cases that the algorithm catch and b is the number of the whole cases considered in the testing .

### 7.3 Execution Time Evaluation

Behavior of algorithm with inputs of arbitrary length is shown in the following table. The execution times according to the size of the sequences are presented in the table 7.2 below . AFALS-N has shown an acceptable execution time over different sequence length .

Table 7.2 : Execution times for sequences of size ranging from 1 kBP to 3MBP

Sequence Size	Execution time(Milliseconds)
1 kBP	225
10 kBP	543
50 kBP	878
150 kBP	1180
500 kBP	1809
1MBP	2050
3MBP	8701

Figure 7.1 shows the ratio between execution time and sequence size. We can notice from the next chart that the behavior of AFALS-N algorithm under input size increase tend to be stable even when the sequences size increased dramatically .

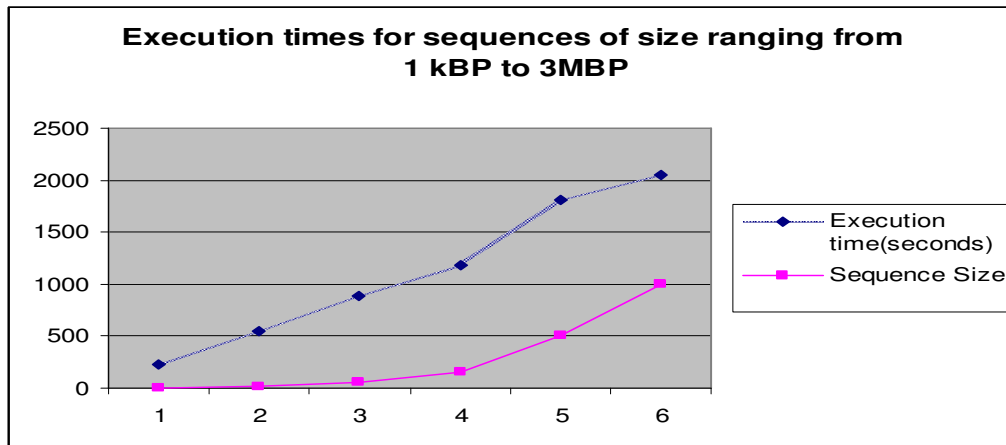


Figure7.4 :Execution times for sequences of size ranging from 1 kBP to 3MBP

### 7.3 Comparison with PatternHunter

In order to verify the quality of the results produced by AFALS-N , we have compared it with PatternHunter .

Comparing to PatternHunter , AFALS-N with word size 9 achieved a better time as shown in the table7.3 . The enhancement ratio is around 0.9 % .

Enhancement in execution time (F) computed by the equation 7.2 which is shown next and sample execution time result is shown in table 7.3 .

$$F = \text{average (AFALS-N execution time / PatternHunter execution time )} \dots\dots\dots(7.2)$$

Table 7.3 : PatternHunter vs AFALS-N

Sequence Length	PatternHunter	AFALS-N
<b>816k vs 580k</b>	9 sec	7.5 sec
<b>4639k vs 1830k</b>	44 sec	38.6 sec
<b>20M vs 18M</b>	13 min	10.3 min

Also when we changed the word length from 9 to 11 AFALS-N performs better than PatternHunter as shown in the next table with enhancement ratio = 0.85 . Table 7.4 shows sample of the comparison made between PatternHunter and AFALS-N.

Table 7.4 : PatternHunter vs AFALS-N(word size 11)

Sequence Length	PatternHunter	AFALS-N
<b>816k vs 580k</b>	7 sec	6 sec
<b>4639k vs 1830k</b>	39 sec	34.6 sec
<b>20M vs 18M</b>	11 min	9 min

## Chapter Eight

### Conclusion and Future Work

#### 8.1 Conclusion

In this thesis we have suggested an Algorithm for Finding Approximate Local Similarities in DNA Sequences (AFALS-N) and it was presented as an approximate local similarities finder and as a pairwise alignment algorithm. It has been implemented using java and tested with real DNA sequences.

The experiments have shown that the performance of AFALS-N was better than the other algorithms mentioned in this study .When Compared with Pattern Hunter the enhancement over execution time was 0.9%.

AFALS-N can also be used for non DNA data comparisons, like protein or amino acids comparisons. Besides that it can be used for non biological string data approximate matching.

A windows application for AFALS-N algorithm has been built using java , and it will be applied in King Hussein Cancer Center in the Molecular Diagnostics and Immunogenetics section .

#### 8.2 Future work

As a future work we will consider different techniques to enhance AFALS-N performance and usability.

We may consider a better candidate selection or verification techniques to reduce the number of candidates or the verification time. Nonconsecutive models for words may considered in order enhancing AFALS-N sensitivity.

The AFALS-N algorithm can be extended to be used in the biological database search or connected to a server and modified to a web version.

## References

- 1- Altschul, S. F., Gish, W., Miller, W., Myers, E. W. and Lipman, D. J. "**A Basic Local Alignment Search Tool**", J.Mol.Biol.215,403-410 , 1990.
- 2-Andreej Poloski and Mark Kimmel , **Bioinformatics** , Book , Springer ,2007 .
- 3- Costas S. Iliopoulos and Thierry Lecroq , **String Algorithmics** , Book ,King's College London Publications, 2004.
- 4-Dan E.Knan and Michael I.Raymor , **Fundamental Concept of Bioinformatics** ,Book ,Benjamin Cummings ,2003.
- 5- Dimitris Papamichail , "**Improved algorithms for approximate string matching**" , BMC Bioinformatics , vol 10 , Suppl1 , 2009 .
- 6- Eva Machackova, Jiri Damborsky, Dalibor Valik, and Lenka Foretova , "**Novel Germline BRCA1 and BRCA2 Mutations in Breast and Breast/Ovarian Cancer Families**" , HUMAN MUTATION Mutation in Brief #459 ,WILEY-LISS, INC, 2001.
- 7-George F.Luger , **Artificial Intelligence** , Book , Adison Wesley , 2002 .
- 8- Giddy Landan and Dan Graur , "**Characterization of pairwise and multiple sequence alignment errors**" , Gene , doi:10.1016 , 2008 .
- 9- Hyyro Heikki , **Practical Methods for Approximate String Matching** , Book ,Tampereen yliopiston laitosten julkaisut , 2003 .
- 10-Jason wang and Katherine Herbert , "**Software engineering and knowledge engineering in bioinformatics**", Bioinformatics ,vol 20 , no 3 , 2000.
- 11- Jean Claverie and Cedric Notredame , **Bioinformatics for dummies** , Book , Wiley Publishing ,2007 .



12- Jeff Augen , **Bioinformatics in the post genomic era** , Book , Addison-Wesley , 2007 .

13-John Micheal , **Computational Biology** , Book ,Chapman and Hall , 2005.

14- Kathleen M. Murphy, Mark Levis, Michael J. Hafez, Tanya Geiger, Lisa C. Cooper, and B. Douglas Smith, " **Detection of *FLT3* Internal Tandem Duplication and D835 Mutations by a Multiplex Polymerase Chain Reaction and Capillary Electrophoresis Assay** " , Journal of Molecular Diagnostics, Vol. 5, No. 2, May 2003.

15- Kisman, D., M. Li, B. Ma, and L. Wang. "**tPatternHunter: gapped, fast and sensitive translated homology search**", Bioinformatics, vol 12 , page 321-325 ,2005

16-Lesy Vinh and Ward C. Wheeler , "**Pairwise Alignment with Rearrangements**" , Genome Informatics , vol 45 , 2006.

17- Li, M., B. Ma, D. Kisman, and J. Tromp , "**PatternHunter II: Highly Sensitive and Fast Homology Search** " , Journal of Bioinformatics and Computational Biology , vol 14 , 164 -175 , 2003 .

18- Lipman, D.J. and Pearson, W.R. " **Rapid and Sensitive Protein Similarity Searches**" .Science . vol 11 , page 446-448 ,1985.

19- Ma, B., J. Tromp, and M. Li . "**Patternhunter: Faster and more sensitive homology search** " , Bioinformatics , vol 18 , no 3 , page 440-445 ,2002 .

20- Maxime Crochemore , Christophe Hancart , and Thierry Lecroq , **Algorithms on Strings** , Book , Springer ,1992 .

21- Mazza and Nelson, **Software Engineering Standards**, book ,Prentice-Hall, 1994.

22-Michael S.Waterman , **Introduction to computational Biology** , Book ,Chapman and Hall , 2003.

- 23- Michele Malagola , Michela Rondoni , Costanza Bosi , Michele Baccarani , and Giovanni Martinelli , " **Rapid Detection of *Flt3* Mutations in Acute Myeloid Leukemia Patients by Denaturing HPLC** " , *Clinical Chemistry* , vol 49:10 1642–1650 , 2003 .
- 24-Nadia Essoussi and Sandes Fayeche , " **A comparison of four pair-wise sequence alignment method** " . *Bioinformation* , vol 2(3): 166-168 ,2007 .
- 25- Needleman, S. B. and Wunsch, C. D. " **A general method applicable to the search for similarities in the amino acid sequence of two proteins** " , *J. Mol. Biol.* 48, 443-453, 1970.
- 26- Neil C. Jones and Pavel A. Pevzner , **An introduction to bioinformatics algorithms** ,Book , Massachusetts Institute of Technology ,2004.
- 27-P.Narayanan , **Bioinformatics a primer** , Book , New age international publishers , 2006.
- 28-Pamela C.Chompe and Richard A.Harvey , **Biochemistry** , Book , Lippincott Williams and Wilkins ,1994.
- 29- Smith, T. F. and Waterman, M. " **Identification of common molecular subsequences** " , *J. Mol. Biol.* 147, 195-197,1981.
- 30- Sommerville I , **Software Engineering**, Sixth Edition, Addison Wesley, 2004.
- 31-Stuart J.Russel and Peter Norving , **Artificial Intelligence a modern approach** , Book, Prentice Hall , 2004 .
- 32-TK Attwood and DJ Parry smith , **Introduction to bioinformatics** , Book , Addison-Wesley ,2005.

## الملخص :

يعتبر إيجاد مناطق التشابه في الحمض النووي (DNA) احد أهم العمليات عند تحليل الحمض النووي . وهو يعد مؤشر على وجود علاقات القرى بين الأحماض أو يستخدم للبحث عن الطفرات الوراثية ذات الدلالات المرضية. إن إيجاد الطفرات الوراثية مهم جدا لتحديد الطرق العلاجية الملائمة لبعض المرضى مما قد يؤثر على حياتهم . يستخدم التشابه التقريبي أيضا في تحديد درجة تشابه السلالات الجينية للكائنات الحية مما يساعد على التعرف على الوظائف الحيوية للجينات المكتشفة حديثا .

أهم طرق البحث عن مناطق التشابه في سلاسل الحمض النووي الريبوزي تنقسم بشكل رئيسي إلى نوعين البرمجة الديناميكية و البرمجة المعتمدة على تنقية النتائج المحتملة . لكل منهما ما يميزها عن الأخرى فالبرمجة الديناميكية تضمن أفضل النتائج بينما التنقية للنتائج تقلل من الزمن اللازم للبحث .

تم في هذه الدراسة اقتراح خوارزمية (AFALS-N) لإيجاد مناطق التشابه التقريبي في سلاسل الحمض النووي (DNA) ، ويتمثل مبدأ عمل الخوارزمية على تنقية النتائج المحتملة و تقليلها لتقليل عمليات البحث .

تم بناء برمجية تطبق الخوارزمية المقترحة و تم اختبار الخوارزمية المقترحة باستخدام عينات حمض نووي حقيقية . وقد أظهرت النتائج تحسنا ملموسا من ناحية وقت البحث و الدقة .

و لقد قورنت الخوارزمية المقترحة مع خوارزمية (PatternHunter) حيث كان أدائها افضل و بلغت نسبة التحسين نحو ٠,٩% .

## Appendix A : Sample of test data

### 1- GAPDH Mutation

LOCUS NG\_007073 3880 bp DNA linear PRI 22-MAR-2009  
 DEFINITION Homo sapiens glyceraldehyde-3-phosphate dehydrogenase (GAPDH)  
 on chromosome 12.

ACCESSION [NG\\_007073](#) REGION: 5001..8880

VERSION NG\_007073.2 GI:163954974

SOURCE Homo sapiens (human)

ORGANISM [Homo sapiens](#)

COMMENT REVIEWED [REFSEQ](#):

This record has been curated by NCBI staff. The  
 reference sequence was derived from [AC006064.10](#).

On Dec 28, 2007 this sequence version replaced gi:[160358353](#).

### ORIGIN

1 aaattgagcc cgcagcctcc cgcttcgctc tctgctcctc ctgttcgaca gtcagccgca  
 61 tcttcttttg cgtcgccagg tgaagacggg cggagagaaa cccgggaggg tagggacggc  
 121 ctgaaggcgg cagggggcggg cgcaggccgg atgtgttcgc gccgctgcgg ggtgggcccc  
 181 ggcggcctcc gcattgcagg ggcggggcggg ggacgtgatg cggcgcggggc tgggcatgga  
 241 ggcttggtgg gggaggggag gggaggcgtg tgtgtcggcc gggggcacta ggcgctcact  
 301 gttctctccc tccgcgcage cgagccacat cgctcagaca ccatggggaa ggtgaagtc  
 361 ggagtcaacg ggtgagttcg cgggtggctg gggggccctg ggctgcgacc gccccgaac  
 421 cgcgtctacg agccttgcgg gctccgggtc ttgcagtcg tatgggggca ggtagctgt  
 481 tccccgaag gagagctcaa ggtcagcgtc cggacctggc ggagccccgc acccaggctg  
 541 tggcgccttg tgcagctccg cccttgcggc gccatctgcc cggagcctcc tcccctagt  
 601 cccagaaac aggaggtccc tactcccgcc cgagateccg acccggacct ctaggtggg  
 661 gacgctttct ttctttcgc gctctgcggg gtcacgtgic gcagaggagc ccctcccca  
 721 cggcctccgg caccgcaggc cccgggatgc tagtgcgcag cgggtgcatc cctgtccgga  
 781 tgctgcgctc gcggtagagc ggccgccatg ttgcaaccgg gaaggaaatg aatgggcage  
 841 cgtaggaaa gcctgccggg gactaacctc gcgctcctgc ctgatgggt ggagtcgct  
 901 gtggcgggga agtcaggtgg agcagggcta gctggcccga tttctctcc gggtgatgct

961 tttcctagat tattctctgg taaatcaaag aagtgggttt atggaggtcc tcttgtgtcc  
 1021 cctccccgca gaggtgtggt ggctgtggca tgggtccaag ccgggagaag ctgagtcag  
 1081 ggtagttgga aaaggacatt tccaccgcaa aatggcccct ctggtggtgg ccccttctg  
 1141 cagcggcggc tcacctcag gccccgcct tcccctgcca gcctagcgtt gacccgacc  
 1201 caaagccag gctgtaaatg tcaccgggag gattgggtgt ctgggcgcct cggggaacct  
 1261 gccttctcc ccatcctc tccggaaac cagatctcc accgcacct ggtctgaggt  
 1321 taaatatagc tgctgacctt tctgtagctg ggggcctggg ctggggctct ctccatccc  
 1381 ttctccccac acacatgcac ttacctgtc tccactcct gatttctgga aaagagctag  
 1441 gaaggacagg caacttgca aatcaaagcc ctgggactag ggggttaaaa tacagctcc  
 1501 cctcttccca cccgccccag tctctgtccc tttgtagga gggactaga gaaggggtg  
 1561 gcttgccctg tccagtaat ttctgacct tactcctgcc cttgagttt gatgatgctg  
 1621 agtgacaag cgtttctcc ctaaagggtg cagctgagct aggcagcagc aagcattct  
 1681 ggggtggcat agtgggggtg tgaataccat gtacaaagct tgtcccaga ctgtgggtg  
 1741 cagtgcacca catggccgt tctctggaa gggcttcgta tgactggggg tgtgggcag  
 1801 cctggagcc ttcagttgca gccatgcctt aagccaggcc agcctggcag ggaagctcaa  
 1861 gggagataaa attcaacctc ttgggccctc ctgggggtaa ggagatgctg cctcgcct  
 1921 cttaatgggg agtggccta gggctgctca catattctgg aggagcctcc cctctcatg  
 1981 ccttctgcc tctgtctct tagatttggc cgtattgggc gcctggtcac cagggtgct  
 2041 tttactctg gtaaagtga tattgttgc atcaatgacc ccttcattga cctcaactac  
 2101 atggtgagtg ctacatggtg agcccaaag ctggtgtggg aggagccacc tggtgatgg  
 2161 gcagccctt catacctca cgtattccc caggtttaca tgtccaata tgattcacc  
 2221 catggcaat tccatggcac cgtcaaggct gagaacggga agcttgcatt caatgaaat  
 2281 cccatcaca tctccagga gtgagtggaa gacagaatgg aagaaatgtg cttggggag  
 2341 gcaactagga tgggtggct ccttgggta tatgtaacc ttgttcctt caatatgctc  
 2401 ctgtcccat ctccccca ccccatagg cgagatcct ccaaatcaa gtggggcgat  
 2461 gctggcgctg agtacgtct ggagtccact ggcgtctca ccaccatgga gaaggctggg  
 2521 gtgagtgcag gagggccgc gggaggggaa gctgactcag cctgcaaag gcaggaccg  
 2581 ggtcataac tgtctctc tctgtctg gctcattgc aggggggagc caaagggtc  
 2641 atcatctctg cccctctc tgatgcccc atgtctgca tgggtgtgaa ccatgagaag  
 2701 tatgacaaca gcctcaagat catcaggtga ggaaggcagg gcccgtggag aagcggccag  
 2761 cctggcacc tatggacag ctcccctgac ttgcgcccc ctccctctt cttgcagca  
 2821 atgcctctg caccaccaac tgcttagcac cctggccaa ggtcatccat gacaacttg  
 2881 gtatctgga aggactcatg gtatgagagc tggggaatgg gactgaggt cccaccttc  
 2941 tcatcaaga ctggctctc cctgccggg ctgctgcaa cctggggtt gggggtctg

3001 gggactggct ttccataat ttctttcaa ggtggggagg gaggtagagg ggtgatgtgg  
 3061 ggagtacgt gcagggcctc actcctttg cagaccacag tccatgcat cactgccacc  
 3121 cagaagactg tggatggccc ctccgggaaa ctgtggcgtg atggccgagg ggctctccag  
 3181 aacatcatcc ctgccttac tggcgtgcc aaggctgtgg gcaaggtcat cctgagctg  
 3241 aacgggaagc tcaatggcat ggcttccgt gtccccactg ccaacgtgc agtgggtggac  
 3301 ctgacctgcc gtctagaaaa acctgcaaaa tatgatgaca tcaagaagg ggtgaagcag  
 3361 gcgtcggagg gccccctcaa gggcatcctg ggctacactg agcaccaggt ggtctcctct  
 3421 gacttcaaca ggcacacca ctctccacc ttgacgctg gggctggcat tgcctcaac  
 3481 gaccactttg tcaagctcat ttctggtat gtggctgggg ccagagactg gctcttaaaa  
 3541 agtgcagggt ctggcgcct ctggtggctg gctcagaaaa agggcctga caacttttt  
 3601 catcttctag gtatgacaac gaatttggt acagcaacag ggtgggtggac ctcatggccc  
 3661 acatggcctc caaggagtaa gaccctgga ccaccagccc cagcaagagc acaagaggaa  
 3721 gagagagacc ctactgctg gggagtcct gccacactca gtccccacc aactgaatc  
 3781 tccctctc acagttgcca ttagacccc tgaagaggg gaggggccta gggagccgca  
 3841 cctgtcatg taccatcaat aaagtacct gtgctcaacc

//

## 2- FLT3 Mutation

LOCUS AC\_000145 97423 bp DNA linear CON 03-MAR-2008  
 DEFINITION Homo sapiens chromosome 13, alternate assembly (based on HuRef), whole genome shotgun sequence.  
 ACCESSION [AC\\_000145](#) REGION: 9398612..9496034  
 VERSION AC\_000145.1 GI:157704454  
 PROJECT GenomeProject:[20837](#)  
 DBLINK Project:[20837](#)  
 SOURCE Homo sapiens (human)  
 ORGANISM [Homo sapiens](#)  
 REFERENCE 1 (bases 1 to 97423)  
 COMMENT The DNA sequence is from the whole genome assembly released by the J Craig Venter Institute as HuRef in May 2007 (see <http://www.jcvi.org/research/huref/>). It is included in the NCBI RefSeq collection as an alternative assembly to the one produced by the Human Genome Sequencing Consortium. The original whole genome shotgun project has the project accession ABBA00000000.1. The HuRef assembly represents a composite haploid version of the diploid genome sequence from a single individual. The highest scoring allele contained is represented in the consensus sequence. DNA Donor Name: J. Craig Venter | Date of Birth: October 14, 1946 | Sex: Male | Ethnicity: Caucasian | Descent: European - England.

ORIGIN (only part of DNA sequence is presented here)

```

1  gtggggacaa gagtaacttt attgaaaata ctaatcctcc atgttacttc tgactggccc
61  tgagtctggg aaggccgcca aagtgtctag gtgatgtatt actctttatg gtagaacacc
121 tattcattat aaactcccc caatacaacc cctgttggtg cagaaatctt aggctgtgac
181 aaccatagct gcctacacat tcctgtatc ttggggtaaa agcacacgtg ctctggaagg
241 aatgtgtagg tggctatggg tgcacaattt caggggtttc gtgaactcca gttaagactt
301 gcctaatta taccatgtaa ataattcaat aatgggcaat tctgtagtag aaattttatt
361 cccaccata aaatatatca ctaaatagct gaaaaattta catattattt taaaacatag
421 acttaaaaaa tcatattagc ttctccttag caaaatgctt ttgtttatg tatttacaag
481 aatatactgt acttcaggta cacaattcac tcaagccagc ctgagaaggc cttggatgca

```

541 gatcaatgct ccaataaagt tcattatcag ctctctctgc cttgtgacag gatgatttga  
 601 ttttcaaaaa gtccctttga aaacaagagt aaacgcagac agcttctaga gaaaagtctg  
 661 gtgaagcagc agttgataat agattttctt ttagtgatga aattaatctt gttttggtaa  
 721 tctacagcct gttaggata ggtggaggga tgaagtcctt aaaactaaat tgttctctta  
 781 cgaatcttcg acctgagcct gcggagagag tagcccaaaa tccatctctc tgctgaaagg  
 841 tcgctgttt tggtaggtgt gaggacattc cgaaacacgg ccatccacat tctgatacat  
 901 ctgaatgtgg gaaagagaca gaacactgat taccatctga ttagatgca catgttatgc  
 961 gccatatta caaattattt aaataaaaac agttgttcta tatagacaat tactttttg  
 1021 tttgtttgt gttgtttgt ttatttttg agacagagtc tcgctctgtt gccagactg  
 1081 gagtgcagtg gtacaacat agttaccctt ggccttgatg ttctgggtt gagcaatcct  
 1141 cccaccttaa cctcctgagt agctgggacc acaagcaggc tccaccacac cctgctaatt  
 1201 tttttattt ttgaaagac aaagtctcac tatgtgtcc aggggtgtct caaactcctg  
 1261 ggctcaagtg atcccacacc accccggcct cccaaagtgc tgcgattaca ggtgtgagcc  
 1321 actacgcccg gcctagacat cacttttaaa atgtttaaac tgatatataa tagatgtaca  
 1381 tattttcagg aaacgtgtag acaagtactt ttattatgca taggtctcag aggatattct  
 1441 atataactaa aaaagcaatt ttggtccttt tattaatgga gaaatcaaat catagtcaaa  
 1501 tattttattt cattattgag tctactctca gatataaaat gtcactctag aaatcctaaa  
 1561 accatgcaga aaaatcataa aagagaaagg ccacaaaagg aaatctgttc attatggagt  
 1621 taatacaagg gactgattct tgagttttcc cttggagttt cagactttt aaatattttt  
 1681 ttctgaaatg aagagattta ctttccttcc ccaaatatga agttaacatg cattcatata  
 1741 gataatttga gaaatacaga aagagacgta gaaggccggg cgcagtggtc catgcctgta  
 1801 atcccagcac tttgggatgc cgaggcgggc ggatcacctg gggttgggag ttcaggacca  
 1861 gcctagccaa cgtggagaaa cctgtctct actaaaaata caaaactagc cgggcatggc  
 1921 ggcgcgcgcc tgcagtcca gctactggg aggctaaggc aggagaattg cttgaacctg  
 1981 ggaggtggag gctgcagtga gcctagattg tgccactgca ctccagcctg ggtgacagag  
 2041 caagactcca tctcaaaaaa aagaaaaaag gctacaagtc atgacaagta cccgccatta  
 2101 tagacagctt gctatgcaca cacaattttg tgtctgtggg ctcaggctat atattctatt  
 2161 ttggaacctt atttgaatt atcaatata tgttattata ctgctcatat gttgcctgtg  
 2221 tcacatattt acattattca ctaaggatgg ccacatattt tttttcacg gcagcctaga  
 2281 gttccatagt actgatgcat cataattaac cttttgacca actggttata cgaacaaac  
 2341 tgaaaagtgc aactcaatc tagtctgacg ttgggataag cagaagtgga attgctggat  
 2401 caaaaaggat gcacactga actgtgatac acaaggccag gctgccctgc agaaaggttg  
 2461 tattttatc tactctatc aacgggtcct ggaaactata tttccagag cttctgaac  
 2521 aatgggtatg tcagctccc acatctctc tcttttctg agcaagaagt tctatctct



2581 taactatata tgttcaacta tctgcaaagt tgggcctttt tectatactt tatgtccatt  
 2641 tgttttcatt ttgaatagcc tgcctatttc ctttgccttt tatgtatata aaaggctata  
 2701 caggctgggt gtactggctc aactctgtaa tcttagcact ttgggaggcc ggggcgggag  
 2761 gattgcttga ggccaagagt tcaagaccaa actaaccaac atagcaagat cctgtcteta  
 2821 aaagaataa gtttttaaaa ggtgatacat ttttattatt attgtttat aagaacttta  
 2881 tgatttagga atcatgcatt ctatttaaat tttatagatt tgttccgttc ccacagcct  
 2941 tgttactgtc tacttctttt tctttgtttt tgacagtttt aatgtgaact gtccagactg  
 3001 cctectacac ttcacttttc tttctagaaa attgtgtgtg tgtgtgtgtg tgtgtgtgtg  
 3061 tgtgtgtatc ctttgacca aatatatcc attgtgaacc aggtgttgca caatgacagc  
 3121 tttgcttga cctgaagga tgaacagtaa ctactcatgc gtgcctttg tgaagtagac  
 3181 atagcagtta gttagcattt gttgaacctg ttgaatccaa atgtacatct ctaccactga  
 3241 atttcaacc acctcatgaa gtttgtgtag cacaaatacc aataacactt ccaatctcc  
 3301 acctgaatta actaacatgt gctcttcate cagtctcact gtctagaagt ttctagaacc  
 3361 atctctgaca atctctctcc actcccatag ctagtactg ggtatagtg taatacatca  
 3421 ctcttttcca tttcttaag tgacctttct ttcctttttt tttttttt ttgttgtgc  
 3481 tgttctgtt gtgacagagt ctactctgt tgcccaggct ggagtgtgt ggcatgatct  
 3541 cagctcacag caacctctgc ctctcagggt caagcaattc ttgtgcctca agtagctggg  
 3601 actacagggt gtaccacca cacctggcta attttatat ttttttagta gagacagggt  
 3661 ttcaccatgt tgaccaggct ggtcttgca cctggcctc aagtatcca cccacctagg  
 3721 cctcccaaag tgctgcgatt acaggcgtga gccaccacce tcagccactg ttgttttaa  
 3781 caggctcaca gataacatca taaaagtgac cttaaatga ctttttaaat acattctct  
 3841 tatgaaattg tgaacaaaac cctaggtttt caaatgtatc attataaaga agtacataaa  
 3901 tttcttata ctttaaaaaa tggttctttt ttccttagtt atcgttctct tttcatctga  
 3961 atgttattta ttgtgcttt tttcttttc aagacagggt ctgctctgt cactcaggct  
 4021 ggagtacagt ggtgcaatca cagctcactg cagcctcaac ctctaggca gaagtgatc  
 4081 tctgtctca gctctctgag taactgggac tactgggtgt cgccactaca cctggttaat  
 4141 ttttaattt ttgtagaga tgggggccca ttatgttccc cagtctggtc tcaaactct  
 4201 gagccaagt gatccttca ccttggcctc tcaatgtct gggattacag gcgtgagcca  
 4261 ccacaccga cttcttttt tcttaattgt tcaactcaaa gatagtagct ttagtagtat  
 4261 ccacaccga cttcttttt tcttaattgt tcaactcaaa gatagtagct ttagtagtat  
 4321 attgtatac ttgttgata tacaatatta atatacagta tagactacac tcagactgct  
 4381 gtattcaaat cctaagtctg acattacca tgttacctta ggcaaattac ttaacctctc  
 4441 tgtgcctcaa tttactagtc tgctaaaggg ataataatag aacctacttc aggagattga  
 4501 ggtgaggatt aagagttatt aattttgtgg ctaatacatt agtaaactct atgattaat

## Appendix B: Sample Java Code

```
//btnAlign
//
btnAlign.setText("Align");

btnAlign.setPreferredSize(new Dimension(100, 29));
btnAlign.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        alignMethod();
    }
});

//btnResult
//
btnResult.setText("Show Results");
btnResult.setPreferredSize(new Dimension(100, 29));
btnResult.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e)
    {
        //btnResult_actionPerformed(e);
    }
});

// btnExit
//
btnExit.setText("Exit");
btnExit.setPreferredSize(new Dimension(100, 29));
```

```

        btnExit.addActionListener)

new ActionListener() // anonymous inner class
}
//      terminate application when user clicks exitItem
public void actionPerformed( ActionEvent event(
}
        System.exit( 0 ); // exit application
// {      end method actionPerformed
// {      end anonymous inner class
// {(    end call to addActionListener

        //btnCancel
//
btnCancel.setText("Cancel");
btnCancel.setPreferredSize(new Dimension(100, 29);
btnCancel.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e(
    }
        //btnCancel_actionPerformed(e(
    {

    {(
//
//btnClear
//
btnClear.setText("Clear");

btnClear.setPreferredSize(new Dimension(100, 29);

btnClear.addActionListener)

new ActionListener() // anonymous inner class
}

```

```

//      display message dialog when user selects About...
public void actionPerformed( ActionEvent event(
}

    seq1.setText(" ")
    seq2.setText(" ")
// {      end method actionPerformed
// {      end anonymous inner class
// :(    end call to addActionListener

        //pnIOCA
        //
        pnIOCA.setLayout(new FlowLayout(FlowLayout.RIGHT, 5, 5)((

            pnIOCA.add(btnClear, 0)((
            pnIOCA.add(btnCancel, 1)((
            pnIOCA.add(btnExit, 2)((
            pnIOCA.add(btnResult, 0)((
pnIOCA.add(btnAlign,0)((

//layout = new FlowLayout(FlowLayout.Right)((
    JMenu fileMenu = new JMenu( "File" ); // create file menu
    fileMenu.setMnemonic( 'F' ); // set mnemonic to F

//      create new... menu item
    JMenuItem newItem = new JMenuItem( "New)(( "
    newItem.setMnemonic( 'n' ); // set mnemonic to A
    fileMenu.add( newItem ); // add about item to file menu
    newItem.addActionListener(

        new ActionListener() // anonymous inner class
    }
//      display message dialog when user selects About...
public void actionPerformed( ActionEvent event(

```

```
}  
  
// {      end method actionPerformed  
// {      end anonymous inner class
```