Multiple Anchor Staged Local Sequence Alignment Algorithm - MASAA

Bharath Govinda Reddy

B.E., Bangalore University, (India) 2001
M.B.A., University of Phoenix, (USA) 2007

Thesis Submitted In Partial Fulfillment Of

The Requirements For The Degree Of

Master Of Science

In

Mathematical, Computer, and Physical Sciences

(Computer Science)

The University Of Northern British Columbia

March 2009

Abstract

Technology advancements have helped biologists gather massive amount of biological data including genomic sequences of various species today. Sequence alignment techniques play a central role in investigating the adaptive significance of organism traits and revealing evolutionary relations among organisms by comparing these biological data.

This thesis presents an algorithm to perform pairwise local sequence alignment. Recent pairwise local sequence alignment algorithms are either slow and sensitive or fast and less sensitive. Our algorithm is faster and at the same time sensitive. The algorithm employs suffix tree data structure to accurately identify long common subsequences in the two given sequences quickly. Regions of high similarity are again identified between segments of long subsequences already found.

Several measures are taken into consideration to design the algorithm, such that the output is biologically meaningful. Data sets are carefully chosen and the output is compared with a well known algorithm, BLASTZ. Experiments conducted demonstrate that our algorithm performs better than BLASTZ in computation time, while either preserving or exceeding the accuracy of alignments at times.

To my beloved parents

Acknowledgements

It has been my good fortune to come across many fine people who have given me moral support, companionship, help, and above all their valuable time during the course of my time at UNBC.

First and foremost, I would like to express my sincere gratitude to my supervisors, Dr. Waqar Haque and Dr. Alex Aravind for their unconditional help, patience and guidance. This research work has certainly benefited from their advice, recommendations and suggestions. I would also like to take this opportunity to acknowledge and thank my graduate supervisory committee member Dr. Iliya Bluskov for his time, encouraging words and advice during my time at UNBC. I am grateful to my external examiner, Dr. Stephen Rader from Biochemistry and Molecular Biology department for his time, advice and suggestions to my thesis.

A special word of gratitude to all my co-workers, group members, friends and roommates that I have worked with over the years. A special word of gratitude, finally, to all the members of my research group present and past, Mr. Xiang Cui, Mr. Jai Prakash, Mr. Jonas Bambi, Mr. Hassan Tahir and Ms. Baldeep. I also take this opportunity to specially thank Mrs. Alex Aravind for her support and great food on various occasions.

I am particularly indebted to my parents, my brothers, and my sisters for their love, unwavering support and encouragement. They have truly backed me all these years, and without them none of this would have been remotely possible.

Prince George, *Bharath Govinda Reddy*

March 11, 2009

Contents

List of Figures

List of Tables

Publications

- Waqar Haque, Alex A. Aravind and Bharath Reddy; "An Efficient Algorithm for Local Sequence Alignment", *EMBC-IEEE Conference*, Personalized Health care through Technology, Vancouver, August, 2008.

- Waqar Haque, Alex A. Aravind and Bharath Reddy ; "Pairwise Sequence Alignment Algorithms A Survey", *INTERNATIONAL CONFERENCE on Information Science*, Technology and Applications (ISTA 2009), Kuwait University, March, 2009.

- Bharath Reddy, Waqar Haque and Alex A. Aravind; "Enhancing Parallelization of Sequence Alignment Using MPI", 4^{th} *Northern HPC Spring Conference*, held at UNBC, Prince George, Canada, May, 2006 (*oral presentation*).

Chapter 1

Introduction

All living creatures can be broadly classified into single cell organisms (for example, bacteria) and multi-cell organisms (for example, humans). Cells are the basic structural and functional units of life or sometimes called the building blocks of life. Multicellular creatures have many organs like tissue, bone, hair, etc. Cells are responsible for structure and function of these organs. Cells possess various molecules in them to perform these functions by chemical reactions. These molecules are typically proteins. To produce protein molecules, cells need a 'recipe book'. The recipe book is stored in a molecule called Deoxyribonucleic acid (DNA). Another molecule, ribonucleic acid (RNA) acts as an intermediary molecule between DNA and proteins.

Proteins are the necessary and vital product of a cell. Proteins are molecules that are responsible for development of all organism. Most functions in a cell are accomplished by proteins. For example, many proteins act as enzymes and catalyze chemical reactions. They carry signals in and out of the cell, and within the cell. They are also responsible for the transport of molecules[29]. The primary structure of a protein is a linear chain of amino acids. There are twenty amino acids, denoted by A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, and V [1]. Protein size is usually measured in terms of the number of amino acids that comprise it. A typical protein sequence contains 100-5000 amino acids. Each protein that an organism produces is encoded in a part of DNA called 'gene'. Humans are believed to have about 20000 different protein coding genes. The number of proteins that can be produced by humans far exceed the number of genes[29]. The genes and the non-coding sequences of the DNA where hereditary information is encoded are together called the genome of that organism.

DNA is a nucleic acid molecule that contains genetic instructions used in the development and functioning of all living organisms, to create new cells, to determine which protein to synthesize and the location of synthesis inside the cell. DNA consists of two long interwoven strands to form a double helix[29]. The chemical composition of a DNA would be a long polymer of simple units called nucleotides, which are held together by a backbone made of alternating sugars and phosphate groups. Attached to each sugar is one of four types of molecules called bases. It is the sequence of these four bases along the backbone that encodes information. This information is read using the genetic code, which specifies the sequence of the amino acids to produce proteins. These bases are adenine (A), guanine (G), cytosine (C) and thymine (T). In DNA, A, G, C and T bases can only form two different base pairs (bp): A-T and G-C. The length of a DNA molecules is expressed in these units (bp). The human

[1]Alanine{A}, Arginine{R}, Asparagine{N}, Aspartic acid{D}, Cysteine{C}, Glutamic{E}, Glutamine{Q}, Glycine{G}, Histidine{H}, Isoleucine{I}, Leucine{L}, Lysine{K}, Mitheionine{M}, Phenylalanine{F}, Proline{P}, Serine{S}, Threonine{T}, Tryptophan{W}, Tyrosine{Y}, Valine{V}

genome contains roughly 3 billion bp. An important property of a DNA molecules is its replication. Before a cell divides, the DNA is unwound into two strands which are both copied or replicated by the DNA polymerase enzyme[29]. During this process, errors or mutations may occur. A mutation is a change in the sequence of DNA bases. When a nucleotide is added to or lost from DNA, the mutation is an insertion or a deletion, respectively.

RNA is similar to DNA; they both are nucleic acids of nitrogen-containing bases joined by sugar-phosphate backbone. However structural and functional differences distinguish RNA from DNA. Structurally, RNA is single-stranded whereas DNA is double stranded. DNA has Thymine, whereas RNA has Uracil as its base. RNA nucleotides include sugar ribose, rather than the Deoxyribose that is part of DNA. Functionally, DNA maintains the protein-encoding information, whereas RNA copies the information from DNA to enable the cell to synthesize a particular protein. The four bases of RNA are adenine (A), guanine (G), cytosine (C) and uracil (U).

The DNA contains information needed by the cell to produce all its RNA and proteins. DNA and RNA molecules are collectively responsible for protein synthesis, which is a chemical reaction, and is responsible for various functions of the cell. Proteins are synthesized in two steps. First, RNA 'copy' of a portion of DNA is synthesized in a process called transcription. In the next step, this RNA sequence is read and interpreted to synthesize protein in a process called translation[29].

In general, RNA, proteins and DNA molecules can be abstracted as strings of letters from their respective alphabet set, given below :

- Alphabet set for DNA= { A, C, G, T }

- Alphabet set for RNA= { A, C, G, U }
- Alphabet set for Protein= { A, R, N, D, C, Q, E, G, H, I, L, K, M, F, P, S, T, W, Y, V }

1.1 Sequence Analysis

Computational biology involves the use of mathematical and computational techniques to solve biological problems ranging from identification of disease causing genes to drug development in both animals and plants. As indicated earlier, DNA, RNA and proteins are the primary components of a cell. Most of the above problems are studied in terms of analysing DNA, RNA and Protein sequences. Biological applications of sequence analysis are many, including:

- Identifying genes and predicting their functions in a new sequence.
- Medical applications. For example, sequence analysis is used to understand the cause and effects of diseases like multiple sclerosis, Alzheimer, etc.
- Identification of gene structures.
- Prediction of protein structures.
- Comparison of homologous sequences to construct an evolutionary tree or molecular phylogenetic tree.

Although, all living organisms have a common origin, which started with the same DNA sequence, they become distant due to mutations frequently occurring in DNA sequences. High sequence similarity usually implies strong functional or structural similarity. This observation namely that closely related organisms have a common ancestor, exploits a simple biological principle: certain regions of the genome (functional elements) tend to be conserved more strongly during evolution than other regions (non-functional). A statistical analysis is shown in Appendix A. Our research focuses on homology between two sequences, to discover similarity relationships among them.

1.2 Sequence Alignment

Sequence alignment is a way of arranging sequences of DNA, RNA and proteins with an objective to find regions of 'similarity', which may provide additional information on the functional, structural, evolutionary and other features of the sequences under study. Aligned sequences are typically represented in rows, one on top of the other. For example, given two sequences, ATATAGAGGACACG and ATAGGGGACATGG, one possible alignment is shown in Figure 1.1. The vertical lines indicate the match. Regions with many matches between the aligned sequences are called 'similar regions'.

```
A T A T A G A G G − A C A − C G
    | | | |   | |   | | |   | |
− − A T A G − G G G A C A T G G
```

Figure 1.1: ***An example of sequence alignment***

In some regions, special characters such as '-', also known as *indels* are added. This insertion of a special symbol represents a mutation (change) or could be looked at as deletion from the other sequence's perspective. When one looks from an evolutionary point of view, this deletion or insertion (change) represents a divergence of one sequence from the other. From a sequence alignment perspective, a similarity would be a rough estimate on how conserved a region is during the evolution. Regions that are perfectly matching represent structural and functional correlation[3].

Sequence alignments are broadly classified into pairwise sequence alignment and multiple sequence alignment. Pairwise sequence alignment is a fundamental technique used to find conserved regions in two sequences. Multiple sequence alignments are traditionally used to find common characteristics and conserved regions in more than two sequences and also to establish evolutionary linkage among the sequences within a family. It is also used to search top scoring hits of sequences in a sequence database. Often, multiple sequence alignment uses pairwise alignment as a component routine. Work presented in this thesis is focused on pairwise sequence alignment.

1.2.1 Pairwise Sequence Alignment

Pairwise biological sequence alignment is a fundamental problem in bioinformatics. Pairwise sequence alignment can be further classified into local sequence alignment and global sequence alignment.

Local sequence alignment finds the best approximate subsequence match within two given sequences. Local sequence alignments are designed basically to search for highly similar regions within the two given sequences. For finding similar (biologically conserved) regions, which may not be preserved in order or orientation, local sequence alignment is very useful. It is typically used to find similarity between two divergent sequences and for fast database searches for similar sequences. Local sequence alignment usually involves less computation compared to global sequence alignment. Some of the popular local sequence alignment are Smith-Waterman[27], FASTA[20], BLAST[2], Gapped BLAST[3], BLAT[18], BLASTZ[26], and PatternHunter[21]. These are discussed later.

Global sequence alignment is used to find the best alignment of both sequences in their entirety. Global sequence alignment looks for global mapping between entire sequences. The objective of global sequence alignment is to exhibit more information such as order and orientation of the similar regions in two given sequences. Global sequence alignments are useful where the sequences are from related organisms and highly likely to satisfy order and orientations of conserved regions in the sequences. However, because of their high computational cost, global sequence alignments are mostly used for the alignment of relatively small sequences[7]. Some of the popular global sequence alignment algorithms are Needleman-Wunch[23], MUMmer[1], GLASS[6], AVID[7], and LAGAN[11]. These are discussed later.

1.2.1.1 Characteristics of Pairwise Local Sequence Alignment Algorithm

Pairwise local sequence alignment algorithms could be characterized according to several parameters like, time and space efficiency, sensitivity and other characteristics including:

- *Molecule specific:* It is very likely that local alignment algorithm is developed for either a DNA, RNA or protein sequence.
- *Length of the sequences:* Certain local sequence alignment algorithms are designed to find alignment only for short sequences. Their efficiency in terms

of either speed or memory usage, drops considerably for long sequences[11]. For example, Smith-Waterman algorithm is very efficient in terms of time and space for short sequences and not for aligning genomic sequences, which are few hundred thousand bps in length.

- *Measure of accuracy:* The objective of local sequence alignment is to find conserved regions in both sequences, which serve as an evidence of structural and functional conservation, as well as an evolutionary relation between the two sequences to biologists. To quantify the similarity, an alignment is associated with a score, generally known as alignment score. Algorithms which compute the optimal score alignment are called 'optimal alignment algorithms'. Smith-Waterman algorithm is the only well known optimal local sequence alignment algorithm. In order to produce an alignment quicker, 'near optimal alignment' is sometimes also acceptable.

- *Speed of Alignment:* Alignment time is one of the important factor considered in the development of sequence alignment algorithms. From the literature, local sequence alignment algorithms developed earlier were slow but produced alignments which made good biological data[29]. Smith-Waterman algorithm[27] is an example which produced optimal local alignment but its speed deteriorated proportionally as the length of the sequences increased. Heuristic algorithms were developed later to make the alignment process faster and are designed to generate an approximate alignment rather than an optimal alignment. BLASTZ is an example of a heuristic algorithm. Majority of the local alignment algorithms developed recently are heuristic algorithms.

- *Memory Efficiency:* Algorithms may or may not be memory efficient. Smith-Waterman algorithm uses dynamic programming and has a serious disadvantage in terms of space utilization. Hirschberg[17] developed an algorithm similar to Smith-Waterman algorithm, that uses a divide-and-conquer approach to reduce the search space for a two-sequence alignment problem from O(n^2) to O(n).

1.3 Definitions and Terminology

In this section, we introduce several fundamental definitions that are necessary to understand the algorithms described later.

Definition 1 *Let 'L' be a set of characters called alphabet set. A sequence 'S' is an array of characters from 'L', such that they are all written contiguously from left to right and occupy a unique position in the sequence 'S'.*
Example : If 'S' is ACBCDB, then |S| =6 and S[3]=B.

Let |S| denote the length of 'S' and S[i] denote the ith character of S. Although alignment may visually indicate the closeness between the sequences, a quantified value of the alignment would be more convenient. In order to quantify an alignment, a scoring function is used. First, scoring function for the alignment of pairs of characters is defined and then scores of aligned pairs of characters are added to get the sequence alignment score. In Figure 1.1, if an exact match between two characters scores +2, and every mismatch or deletion (space) scores -1, then the alignment has a score of, 10.(2)+ 6.(-1)=14.

Based on the relationship between pair of characters in an alignment, the alignment score function σ can be classified into three types.

Definition 2 *A character alignment score σ is a real valued function on pairs of characters. The score function σ is called:*

1. Match alignment score if the characters are the same.
2. Mismatch alignment score if the characters are not indels and also different.
3. Gap alignment score if one character is indel and the other is not.

The value for gap alignment score is usually referred as gap penalty.

Definition 3 *Let S_1 and S_2 be two sequences of length n. The alignment score ρ of S_1 and S_2 is defined,*
$\rho(S_1,S_2) = \sum_{i=1}^{n}\sigma(S_1[i],S_2[i])$.

The sequence alignment scoring function ρ defined above is called sum-of-pair (SP).

1.3.1 Scoring Function

We now discuss some well known scoring functions. The simplest, is the constant function, where all matches are given the same value and all mismatches are penalized with a constant value [4]. For any given pair (x, y),

- $\sigma(x, x) = a,\ a \in R^+$
- $\sigma(x, y) = b,\ b \in R^-$
- $\sigma(x, -) = \sigma(x, -) = c,\ c \in R^-$

The constant values ‘a’ and ‘b’ are usually obtained from a scoring matrix. A scoring matrix is a $n \times n$ matrix[2] [3] in which each cell contains a score for the corresponding pair of bases[4]. The constant function described above would result in a matrix where matches have same value and mismatches have a different value, shown in Table 1.1. Unitary scoring matrix was used for sequence alignment[4] prior to popular scoring matrices. A unitary scoring matrix is shown in Table 1.2.

Table 1.1: ***Scoring matrix***

*	*A*	*C*	*G*	*T*
A	130	-36	-36	-36
C	-36	130	-36	-36
G	-36	-36	130	-36
T	-36	-36	-36	130

PAM (Percentage of Acceptable point Mutations per 10^8 years) series of matrices[28][13] and BLOSUM (BLOcks SUbstitution Matrix) series of matrices[16]. PAM matrix is based on mutations observed throughout a global alignment, focused mainly on highly conserved and highly mutable regions. The Blosum matrices also focuses on highly conserved regions but only in series of alignments which do not contain gaps. Both, PAM and BLOSUM matrices use different scores for each pair of bases unlike the constant function defined above. PAM and BLOSUM differ in the way replacements are counted, unlike the PAM matrix, the Blosum procedure uses groups of sequences

[2]In case of DNA, n = 4, the number of unique bases that is composed of

[3]In case of protein, n = 20

Table 1.2: *Unitary Scoring matrix*

*	A	C	G	T
A	1	0	0	0
C	0	1	0	0
G	0	0	1	0
T	0	0	0	1

within which not all mutations are counted the same[16].

1.3.2 Gap Alignment Scoring Function

The objective of sequence alignment algorithm is to pick an alignment which has the maximum alignment score. Since gap penalty scoring function contributes to overall alignment score, the gap alignment score and the number of the gaps in the alignment, for a scoring matrix selected, would affect the alignment that is finally picked. Hence, gap alignment scoring function used in the alignment score computation is very important. There are two types of gap alignment scoring functions namely, constant gap scoring function and affine gap scoring function. In a typical gap alignment scoring function, the gap penalty score is fixed irrespective of the location in the aligned sequences. This is called constant of fixed gap penalty score function. In practice, mutations usually occur as a block of contiguous columns with a gap in the same sequence. This observation supports the evolutionary model, that is, given 'S and

'T, an ancestral sequence, 'U, then there is a probability that a block of contiguous columns of gaps occurred in course of time, separating 'U from 'S and 'T. In order to score such events of long contiguous block of gaps, a score 'gap open penalty is given at the beginning of each gap, and for each subsequent gap in the same block, a 'gap extension penalty is given. Such scoring function, where the penalty is different for the first and subsequent gaps, is called affine gap penalty. It can be defined as follows.

Definition 4 *Affine gap score function has two components: open gap penalty 'd' and gap extension penalty 'e'. Now the affine gap penalty score can be computed as,* $d + l \times e$*, where 'l' is the length of the gap.*

In Figure 1.2, the rectangular blocks indicate the first gap position and subsequent gaps are shown in encircled area. Usually, higher penalty is given to first gap and lower penalty is given to subsequent gaps to encourage single large insertions or deletions[9] to help identify regions where large mutations have occurred. If a score of +2 is

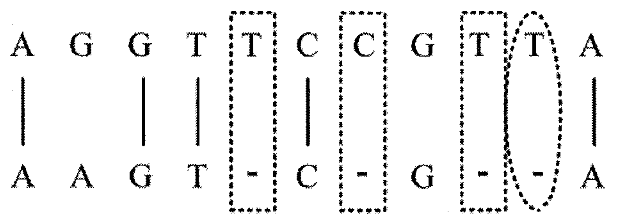


Figure 1.2: ***Affine gap score-indicating first gap and subsequent gap***

given to a match, -1 for a mismatch, -2 for gap opening penalty and -1 for each match following the first match, the alignment score would be 2. We next introduce reference sequence and annotated sequences.

Definition 5 *A reference sequence is a non-redundant sequence representing genomic data, and protein information. In other words, it is a comprehensive data representing the complete sequence information for any given species[25].*

Sequences are translated to include conserved regions starting and ending place in the sequence, different database cross-references for the sequence, and other features using a combined approach of collaboration and other input from the scientific community[25]. A conserved region could be a gene, protein, coding region (part of sequence responsible for protein production in the cell) or other information. An annotated sequence is the absolute or complete data gathered or constructed from different sources. Data includes the starting and ending positions of a gene in a genome, the exon starting and ending positions, specific protein synthesizing genes, length of the sequences and other data.

1.3.3 Performance metric

Let S and T be two sequences. The objective is to find an alignment of S and T, that has the maximum possible score for these two sequences. From the literature, algorithms vary depending on one or more of the following goals:

1. Space and time efficient.
2. Sensitive. The sensitivity is measured by four different scores:
    (a) Percent identity score: representing the percent of the alignment that involves identical base pairs(bp)[29].

(b) Total column score (TCS): the number of correctly aligned columns divided by the number of columns in the reference sequence[1].

(c) Percent similarity score: representing the percent of alignment that involves identical and similar matching aminoacids. This applies to protein sequence alignments, where similar residues are amino acids that have similar physiochemical properties[29].

(d) Maximum alignment score: which is scoring matrix dependent and gap penalty score function dependent. In alignment scores, there are different types of scores, depending on what part of the final alignment is taken into consideration:

   i. Total alignment score, that is the alignment of the whole sequence

   ii. Score of a filtered region of an alignment, this could be parts of the alignment which have score above a threshold (according to a suitable scoring matrix)[7]

   iii. Score of the filtered region, this could be a certain select region such as genes or its total coverage in volume[7].

1.3.4 Seed and Anchor

Consider the sequences S_1 and S_2 as shown in Figure 1.3. In this example, we see that there are regions in S_1 which are clearly aligned with regions in S_2. These common regions could be conserved regions that have not changed by evolution in either sequence.

In Figure 1.3, the conserved region X1 is identical to Y1. That is, every character

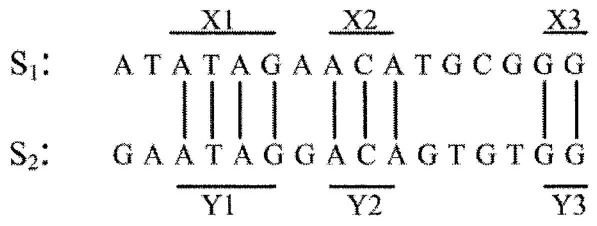

X1
X2
X3
S₁:
A T A T A G A A C A T G C G G G
S₂:
G A A T A G G A C A G T G T G G
Y1
Y2
Y3

Figure 1.3: ***Conserved regions in the two sequences***

in X1 matches the character in the corresponding position in Y1. Consider Figure 1.4 given below. In Figure 1.4, it is logical to consider that $X_1X^1X_2$ and $Y_1Y^1Y_2$ as

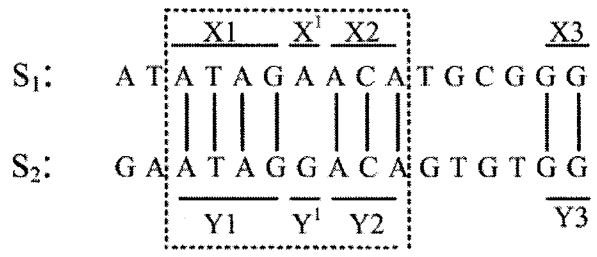

X1
X¹
X2
X3
S₁:
A T A T A G A A C A T G C G G G
S₂:
G A A T A G G A C A G T G T G G
Y1
Y¹
Y2
Y3

Figure 1.4: ***Highly similar regions in the two sequences***

conserved regions with a mutation at either X^1 or Y^1. If such highly similar regions could be identified, then they can be aligned with each other easily. Such pair of highly similar regions are referred to as 'seed' [18] [9] [8]. We formally define a seed below.

Definition 6 *X_1 and X_2 are considered as a seed, if there exists a contiguous region X_2 of m in S_2, such that X_1 is highly similar($\approx$) to X_2.*

High similarity between X_1 and X_2 could be

HS1: A perfect match, every character in X_1 matches the character in the corresponding position in Y_1

HS2: $\rho(X_1,X_2) \geq t$, for a given t and positions where base pairs (bps) have to match (care position) are not fixed.

HS3: X_1 matches closely with X_2, such that care (and do not care) positions are fixed unlike the other seeds already mentioned.

HS3:1 k mismatches in do not care positions and m-k matches in care positions (BLAT seeds)[18].

HS3:2 k mismatches in k do not care positions (Spaced seeds)[21].

HS3:3 The score of $\rho(X_1,X_2)$ in care position of $X_p, X_{p+1} \ldots X_{p+t-1}$ and $Y_p, Y_{p+1} \ldots Y_{p+s-1} \geq t$, for a given t (Vector seeds)[29].

The main goal of heuristic algorithms is to quickly find short regions of similarity and build the overall alignment around these short regions, commonly called 'seeds' as defined above. In global sequence alignment, to reduce the computational time, certain short regions are selected to be part of the final alignment. These short regions selected are commonly called 'anchors'[29].

1.4 Motivation and Contribution

In sequence comparison, the objective is to find local similar regions in the two sequences. Two regions can be either highly conserved or poorly conserved regions of the sequences. The research in this area of biological sequence comparison[27][20][2][20][3][24] has resulted in both optimal and heuristic algorithms. Optimal algorithm focus to produce the optimal alignment while heuristic algorithms produce near-optimal alignment. Optimal algorithms provide local alignments, but face serious constraints in terms of time required for the comparison procedure[27] or are less sensitive in identifying important local segments or even generate unrelated fragments, leading to problems in comparative gene prediction and establishing sequence functions. In this situation, achieving accurate local-alignment between regions of two long sequences within a reasonable time, pose a big challenge.

Our study has been inspired by the frequency of use and the application of a very efficient data structure called suffix tree (refer 2.2.2.1), which is used to find long similar regions in the two sequences efficiently. However, the suffix tree was used only for global sequence alignment algorithm and not for local sequence alignment algorithms because look-up table is faster than actually building the suffix tree and continues to be frequently used in recent algorithms. Also, 'high similarity region' defined by Kent[18], which not only adds more sensitivity but also improves the algorithm speed, has greatly influenced our work.

This thesis presents a new local pairwise sequence alignment algorithm that is not only fast, but also enables local alignment with a high degree of similarity between the two given sequences. The purpose is to present a way to improve speed and sensitivity over BLASTZ[26]. BLASTZ was selected for several reasons. First, it is the best algorithm among BLAST series of algorithms (BLASTN, BLASTP, BLAST2 and others), both in terms of speed and sensitivity. Second, BLASTZ is commonly used as a baseline algorithm to compare other algorithms in the literature. Third, BLASTZ is the only algorithm which is fast and sensitive at the same time. Other algorithms are either fast or very sensitive but not both. Our algorithm starts by finding all long conserved regions between the two sequences using suffix tree. Suffix tree is already used in global sequence alignment algorithms such as MUMmer[1] and AVID[7] to find similar long conserved regions. We employ suffix tree in the first phase of our local sequence alignment algorithm to find long similar regions of minimum length $'l'$.

In the second phase of our algorithm, 'high similarity regions' defined by[18] are identified in the segments between long conserved regions found in the first phase. The advantage of finding 'high similarity regions' is to improve the sensitivity over BLASTZ without compromising computation time significantly. Two adjacent high similarity regions can be overlapping or crossing. Many algorithms including MUMmer[1], GLASS[6], AVID[7], and LAGAN[11] omit such overlapping similar regions. Our algorithm takes into account such conditions to improve sensitivity further. The algorithm's final phase involves extending similar regions already found to produce the final alignment. For this phase, we propose a new method which improves the computational time efficiency of our algorithm.

Chapter 2

Related Work

This section provides the background required to understand the context, functioning and contribution of the algorithm presented in this thesis. Our algorithm is inspired from previous work on local and global sequence alignments. We will review relevant algorithms, first for local sequence alignment and then for global sequence alignment.

2.1 Local Sequence Alignment Algorithms

Over the past several decades, many algorithms have been proposed for local sequence alignment. These algorithms fall under two distinct categories; optimal and heuristic. In the following sections, we begin by explaining a popular optimal alignment algorithm, and later introduce several heuristic algorithms.

2.1.1 Optimal Local Sequence Alignment Algorithm

A popular optimal local sequence alignment algorithm is the Smith-Waterman algorithm. Smith-Waterman algorithm[27] was proposed in 1981 and is based on a technique called, ‘dynamic programming’, a term coined by Richard Bellman in 1940’s [14] to describe problem solving, where one needs to find the best decisions one after another. The idea of dynamic programming is to decompose the problem into smaller problems and solve each subproblem using the same approach recursively. The subproblem solutions are saved and used later to find a solution to the whole problem. The local sequence alignment algorithm proposed by [27] has a computational time complexity of $O(mn)$, where m and n are the lengths of the two sequences.

Let S and T, be two sequences of length m and n, respectively. Let σ be character alignment score function. The dynamic programming algorithm builds up the optimal score of an alignment between S and T by computing the optimal scores of alignments between all characters of S and T. Let $V(i, j)$ be the value of the optimal alignment of strings *S[1]... S[i] and T[1]...T[j].* The algorithms finds all optimal values, $V(i, j)$ with $0 \leq i \leq n$ and $0 \leq j \leq m$, in increasing order of i and j. Each of the optimal values could be computed relatively easily provided optimal values for smaller i and j are computed already. To begin, we need a basis for $i{=}0$ and $j{=}0$[29],

$$\begin{aligned} V(0,0) &= 0 \\ V(i,0) &= V(i\text{-}1,\ 0) + \sigma(S[i], -), for\, i > 0 \\ V(0,j) &= V(0,\ j\text{-}1) + \sigma(-, T[j]), for\, j > 0 \end{aligned}$$

$V(i,\ 0)$ means that if i^{th} character of S is to be aligned with null character with T, they must be matched with an indel. $V(0,\ j)$ means that if null characters of S are to be aligned with the j^{th} character of T, then they must be matched with an indel. There are $n + 1$ characters of S and $m + 1$ characters of T including zeros in the first row and column. A sequence scoring table with $m + 1$ columns and $n + 1$ rows is created as shown below (Figure 2.1). The algorithm computes an optimal alignment

$$\begin{array}{cc|c|c|c|c|}
 & S_1 & & A & G & C \\
S_2 & & 0 & 1 & 2 & 3 \\
\hline
 & 0 & 0 & 0 & 0 & 0 \\
\hline
A & 1 & 0 & & & \\
\hline
A & 2 & 0 & & & \\
\hline
A & 3 & 0 & & & \\
\hline
G & 4 & 0 & & & \\
\hline
\end{array}$$

Figure 2.1: ***Initial scoring matrix***

between *S[1..i] and T[1..j]*, recursively using the following formula.

$$V[i,j] = max \begin{cases} V[i\text{-}1,\ j] + \sigma(S[i\text{-}1],\ \text{-}) & \text{for} \quad i > 0, j \geq 0 \\ V[i\text{-}1,\ j\text{-}1] + \sigma(S[i],\ T[j]) & \text{for} \quad i, j > 0 \\ V[i,\ j\text{-}1] + \sigma(\text{-},\ T[j\text{-}1]) & \text{for} \quad i \geq 0, j > 0 \end{cases} \tag{2.1}$$

The highest score in the matrix is the optimal score, *opt(S, T)*. During the matrix computation, arrows indicating how each sub-optimal score *V(i, j)* is obtained are saved, see Figure 2.2. On completion of the matrix, a procedure traces back the arrows from the highest score to the cell containing, *(0,0)*. The alignment is actually built during this, arrow tracing, stage. Each arrow represents a column in the sequence. A vertical arrow means, a column of *S[i]* matching with a space in *T*, a diagonal arrow

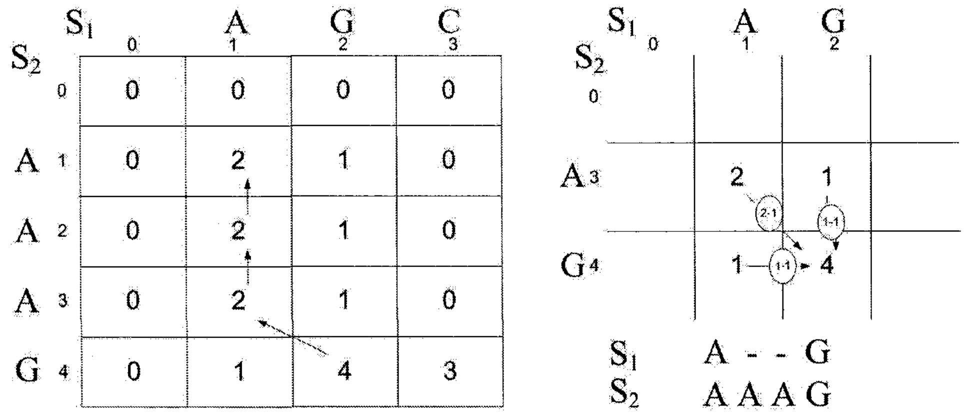


Figure 2.2: ***Matrix trace back to find the final alignment. Second figure clarifies the evaluation of entry matrix[4, 2]: it is obtained from matrix[3, 1], which is why we draw an arrow going from matrix[4,2] to [3,1]***

means, a column of *S[i]* matching a column of *T[j]*, a horizontal arrow means *T[j]* matched with space. Figure 2.2 illustrates an example of a filled matrix with drawn optimal paths and corresponding alignments. In the above figure, $\sigma(S[i], T[j]) = 2$, *if* $S[i]=T[j]$, $\sigma(S[i], T[j]) = -1$ *if* $S[i], \neq T[j]$ *and* $\sigma(S[i], T[j]) = -1$, *if* $S[i]$ *or* $T[j] = -$.

Smith-Waterman algorithm computes the optimal alignment between two sequences S and T of length m and n, respectively, in time and space equal to $O(mn)$. As the length of the sequences increase, Smith-Waterman algorithm becomes very demanding both in terms of time and memory resources. To overcome this, heuristics algorithms were developed.

2.1.2 Heuristic Local Sequence Alignment Algorithms

Heuristic algorithms differ from optimal alignments in that they do not find the optimal alignment but find near optimal alignment. Heuristic algorithms are based on an observation and a necessity to improve computational speed over optimal algorithms. The observation is that, there are seeds in both sequences which have high alignment score. If these seeds are identified in advance, a local alignment can be built around them. This brings us to a number of questions. What constitutes a seed? How are seeds identified? How are seeds chosen from a set of seeds found for the final alignment? In the following sections, we will examine different heuristic algorithms in detail and attempt to answer these questions.

2.1.2.1 FASTA

The FASTA standing for FAST-ALL, reflecting that it can be used for both fast protein comparison and nucleotide comparison was a heuristic algorithm developed by Lipman and Pearson[20]. Given two sequences, S_1 and S_2, the algorithm starts by finding perfect match seed of a given length 'l' using a look-up table. In order to understand a perfect match seed, consider an example shown below. In Figure 2.3,

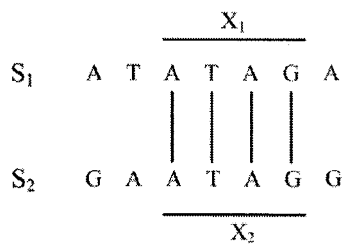


Figure 2.3: ***Perfect match seed from two sequences***

we see that all characters in X_1 perfectly match with all characters in X_2 at their respective positions. Such perfectly matching regions, X_1 and X_2 are called perfect match seeds.

A look-up table is a data structure, usually an array, used widely in heuristic algorithms usually for local sequence alignment problem. Given two sequences S_1 and S_2, seeds of a given length, *'l'*, made from character combination of the set T= $\{V_1, V_2, \ldots V_p\}$, are stored in an array or table shown below in Figure 2.4. The main idea

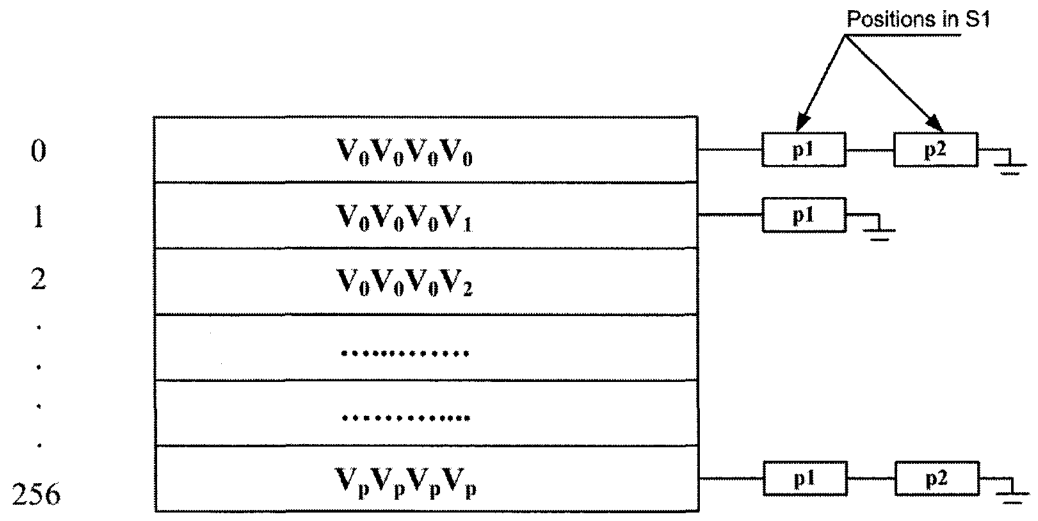
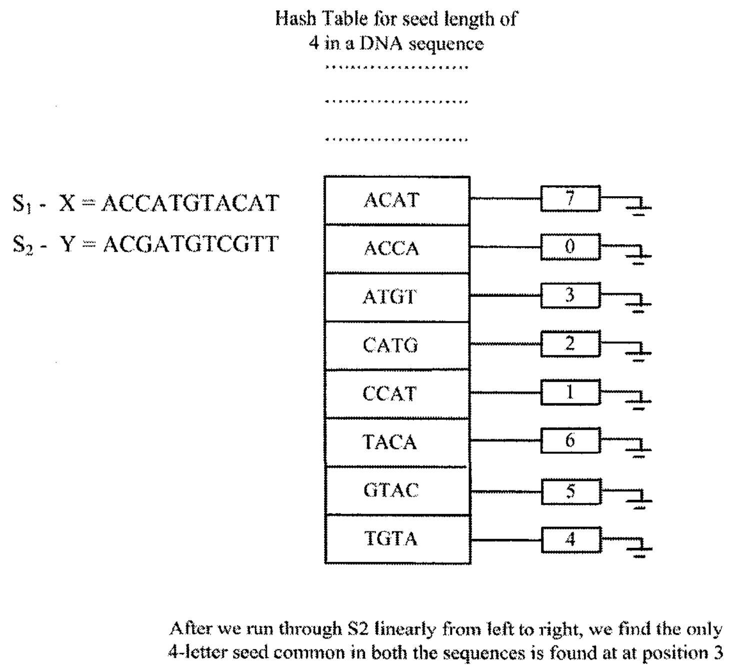


Figure 2.4: ***Look-up table***

of the look-up table is to have seeds occupy unique positions in the array, such that position of seed in the array is obtained from the value computed using a hashing function. Let us say, we are hashing seeds of length four in a DNA sequence and each character from the DNA sequence set, A, C, G, T is hashed individually. Assuming A=00, C=01, G=10, T=11 is the binary value for each DNA character of a hashing function, then a seed AAAA would have a binary hash value of 00000000. This value could be translated to a decimal value zero and used to point to that first position in the array; similarly, a seed TTTT would have a binary hash value of 11111111, pointing to the 256th position in the array. The data recorded at these positions in the array are the starting positions where seeds are found in sequence one. There are times when a seed is found multiple times in a sequence; in such cases, all seeds positions in the sequence are recorded using a linked list for each array element (each array element is a linked list). When one wants to find the seed position in S_1, valuable time is saved by directly going to the location in the array rather than linearly moving from top to bottom. Once the look-up table is established for S_1, we can linearly move through S_2 and find all seeds of length *'l'* in S_1 using look-up table. The positions of seeds in sequence S_2 is known when moving across it. In this way, we can find seeds in time *O(m)*, where *m* is the length of S_2. An example is shown in Figure 2.5 for seeds of length 4 and how they are recorded in the look-up table.



Hash Table for seed length of 4 in a DNA sequence
S1 - X = ACCATGTACAT
S2 - Y = ACGATGTCGTT
ACAT
7
ACCA
0
ATGT
3
CATG
2
CCAT
1
TACA
6
GTAC
5
TGTA
4
After we run through S2 linearly from left to right, we find the only 4-letter seed common in both the sequences is found at at position 3

Figure 2.5: ***Look-up table for seeds of length 4 in sequence S_1***

The FASTA algorithm was the first heuristic algorithm to employ the concept of seeds and look-up strategy to find the local sequence alignment[15]. The working of the FASTA algorithm, can be broken down into following steps.

Step 1. The algorithm starts by identifying perfect match seeds from the two sequences using the look-up table[5]. The FASTA achieves much of its speed in this step.

Step 2. In addition to the lookup table, FASTA uses a *'diagonal'* method to find all diagonal seeds between the two sequences. In other words, FASTA identifies all seeds along a diagonal path[5]. Since the final alignment for the two sequences is most likely to be found in the diagonal from the left-hand top corner to right-hand bottom corner, diagonal path is considered[5]. The diagonal path need not necessarily lie on the main diagonal. In this diagonal path, there are regions where there are seeds and regions of mismatches (seeds are absent). FASTA finds all seeds in a diagonal path using the same look-up table. For two seeds of a given length *'l'* in both the sequences, they are said to be diagonal to each other, if they are separated by exactly the same value in both sequences[5]. FASTA uses PAM[28] to score these seeds using sum of pairs scoring function. All seeds are given a positive value and the intermediate regions are given negative score, and the score decreases with increasing distance. Thus, groups of seeds with high similarity scores contribute more to the local diagonal score than to seeds with low similarity scores[5]. In the process, there could be *'n'* diagonal seeds, of which, FASTA saves the 10 best seeds, regardless of whether they are on the same side or on different diagonals (either left or right).

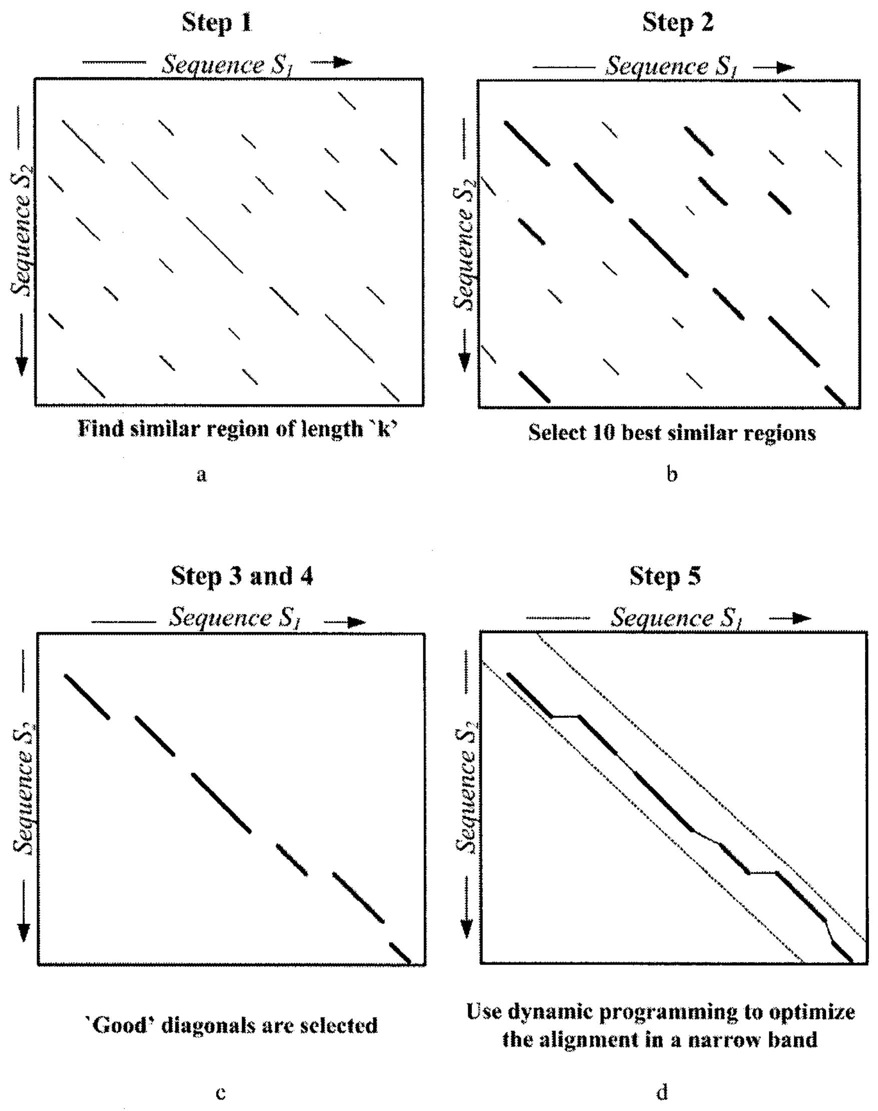

Step 1
Sequence S_1
Sequence S_2
Find similar region of length `k'
a
Step 2
Sequence S_1
Sequence S_2
Select 10 best similar regions
b
Step 3 and 4
Sequence S_1
Sequence S_2
`Good' diagonals are selected
c
Step 5
Sequence S_1
Sequence S_2
Use dynamic programming to optimize the alignment in a narrow band
d

Figure 2.6: *Four steps in FASTA algorithm*

Step 3 A diagonal containing seeds is composed of regions of perfect matches and mismatches in the intermediate seed regions. In this stage, FASTA identifies a diagonal which scores the highest value. This single diagonal which has the best score is called*init1.* Apart from this, other diagonals containing seeds above a threshold value, *'t'* are taken into consideration while those below the threshold are discarded.

Step 4 FASTA finds the 'good' diagonal from the diagonals found in the previous stage. A good diagonal is one which has a score above a threshold[5]. FASTA combines all such diagonals into a single high scoring alignment allowing spaces. This is done as follows. A directed weighted graph whose vertices are the seeds found in the previous step is constructed, and the weight in each vertex is calculated. The score is the combined score of all previous seeds, including the intermediate weighted graphs, Figure 2.7. FASTA then extends the edge from vertex u represented by seed u to vertex v, represented by seed v, if seed v starting address is lower than seed u starting address. Next, it extends an edge from vertex u to vertex v if the seed represented by v starts at a lower row than where the seed represented by u ends. The problem of overlapping seeds does not arise at all. The weighted graphs between the seeds is pictorially shown in Figure 2.7. The maximum weighted graph is then selected and the best alignment found is marked as *'initn'.* As in the previous stage, it discard alignments which have relatively lower score than *initn,* say 20 percent lower than *initn.*

Step 5 In this step FASTA computes an alternative local alignment score, in addition to *initn*[20]. FASTA builds a narrow band of width *'k'* centered along the

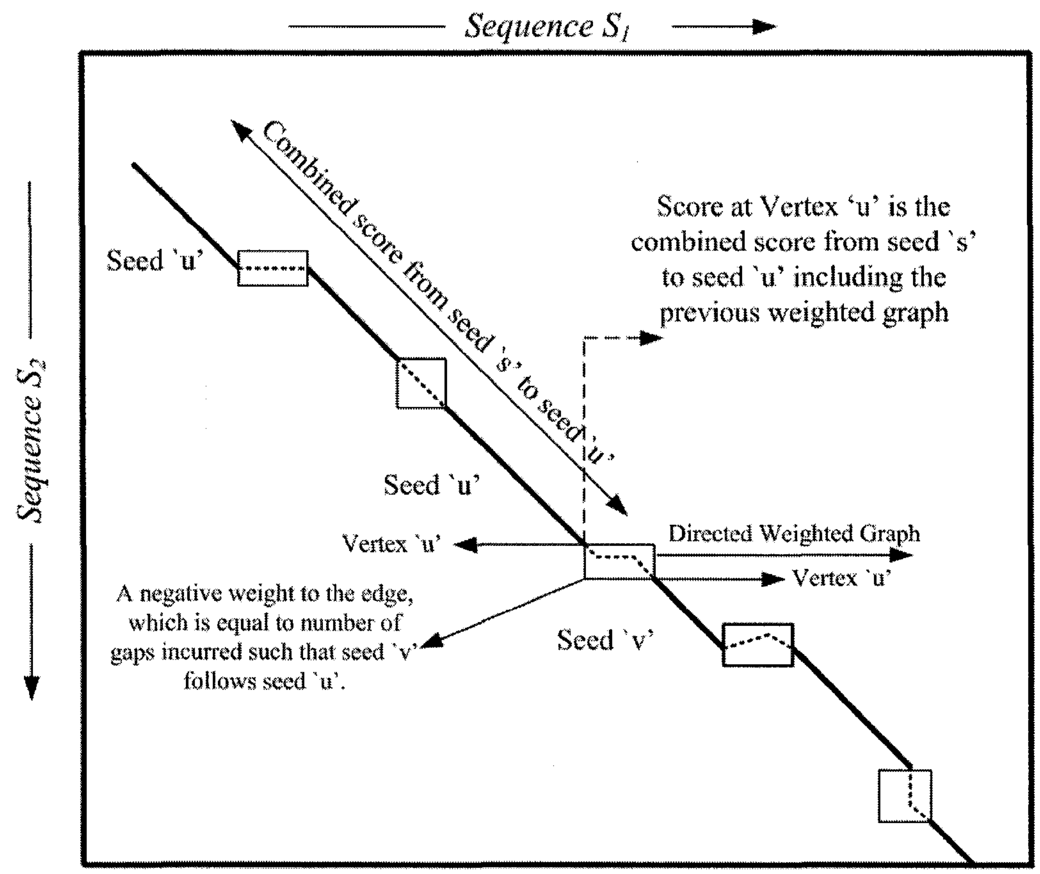

Sequence S_1
Sequence S_2
Combined score from seed 's' to seed 'u'
Score at Vertex 'u' is the combined score from seed 's' to seed 'u' including the previous weighted graph
Seed 'u'
Seed 'u'
Vertex 'u'
Directed Weighted Graph
Vertex 'u'
A negative weight to the edge, which is equal to number of gaps incurred such that seed 'v' follows seed 'u'.
Seed 'v'

Figure 2.7: ***Directed weighted graphs between seeds***

init1 [20] (high scoring diagonal). The idea behind this is that the optimal alignment would have *init1* in the final alignment. FASTA computes an optimal local alignment in this band by using Smith-Waterman algorithm assuming that the optimal alignment lies within this band. FASTA next finds that the best local alignment falls within the defined band, the local alignment algorithm essentially merges diagonal runs found in the previous stages to achieve a local alignment which may contain indels. The intuition is that best alignment would lie within this band. The best local alignment computed is the final local alignment between S_1 and S_2. Although FASTA is a heuristic algorithm, it was claimed by the authors that the resulting alignment scores compare well to the optimal alignment, while the FASTA algorithm is much faster than the optimal dynamic programming alignment algorithm[20].

Limitations: For sequences which are divergent, because the FASTA uses a k-tuple seed strategy, many smaller regions below k-tuple will be missed. Also, if the sequences under consideration have more than one region of homology (two optimal diagonals), only region around *init1* is found while the region which contributed to *initn* is discarded. The main advantage of FASTA over Smith-Waterman algorithm is the speed of the alignment process.

2.1.2.2 Basic Local Alignment Search Tool- BLAST

As in FASTA, BLAST[2] uses look-up table to identify seeds, but the rest of the algorithm is different. The main advantage of BLAST over FASTA is speed. BLAST considers seeds which have a score above a threshold *'t'* for protein sequence alignment and an exact match seed for DNA alignment. Since the algorithm considers seeds of a fixed length, the algorithm does not guarantee the optimal alignment, because some sequence hits may be missed. We will first define *'seeds above a threshold'.*

Definition 7 *Given a threshold value 't' and a scoring matrix, X_1 and X_2 are considered a seed, if there exists a contiguous region X_2 in S_2, such that, the alignment score ρ of X_1 and X_2 defined by,*

$$\rho(X_1,X_2) = \sum_{i=0}^{n}\sigma(X_1[i],X_2[i]) \geq t.$$

To illustrate this, consider the example shown in Figure 2.8; X_1 is "GSV" (size =

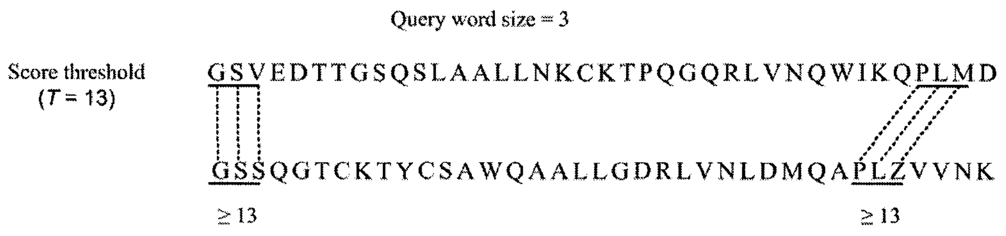

Query word size = 3
Score threshold (T = 13)
GSVEDTTGSQSLAALLNKCKTPQGQRLVNQWIKQPLMD
GSSQGTCKTYCSAWQAALLGDRLVNLDMQAPLZVVNK
≥ 13
≥ 13

Figure 2.8: ***Seeds above a threshold score***

3), we see one such possible combinations of X_2="GSS", when aligned with X_1, their alignment score exceeds or is equal to threshold value 13. All such combinations of X_1 and X_2 can be considered as seeds.

Step 1 BLAST begins by first identifying all seeds (above a threshold) in both the sequences. The default length of the seed is 3 for protein sequences and 11 for DNA sequences[2]. BLAST finds seeds for the entire sequence using the sliding window as shown in Figure 2.9. For each seed in the sequence, set of neighborhood words which exceed the threshold of *'t'*, is also generated dynamically. A neighborhood seed is a seed obtaining a score of at least *'t'* using a selected scoring matrix. There

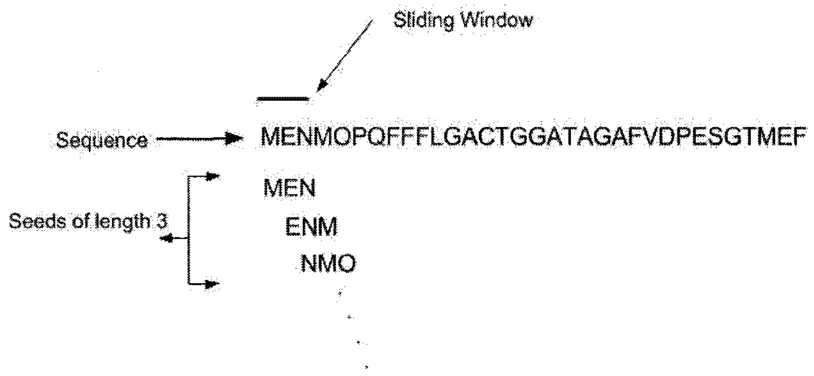


Figure 2.9: ***BLAST-seeding***

could be multiple neighboring seeds for a seed that exceeds a threshold value, '*t*', (see Figure 2.10). BLAST uses BLOSUM62[16] and PAM40[13][28] scoring matrices for proteins and DNA sequences, respectively. Set of neighboring words as well as the exact matches for the seed are then used to match against the second sequence.

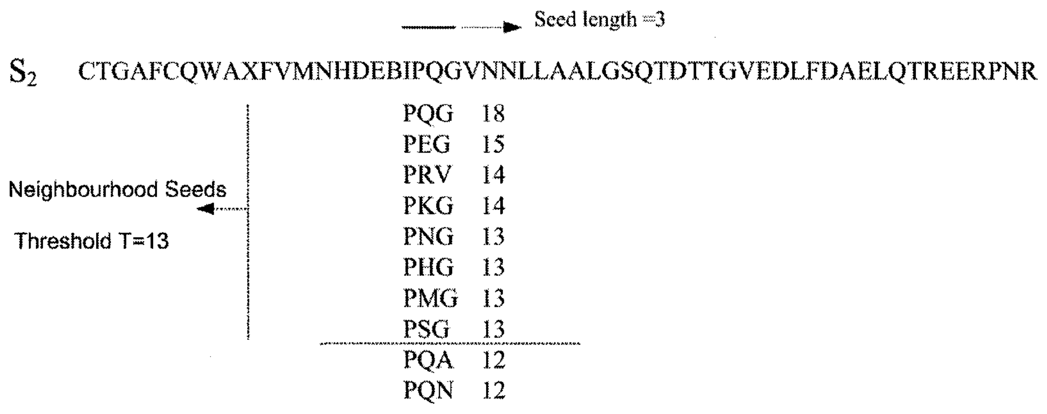


Figure 2.10: ***Indexing***

Step 2 After finding seeds, BLAST algorithm extends the seed alignment in both directions without introducing any gaps. When the alignment is extended on either side, an alignment score can increase or decrease. When the alignment score after extension on both sides drops below a predefined threshold *'S'*, the extension is stopped. S is determined empirically by examining a range of scores found by comparing random sequences and by choosing a value that is significantly greater than the range of score considered for random sequences[2]. S is chosen such that the segment has the highest score. After extending the seed alignment in both directions, if the score is above a certain threshold, then such seed segment pair is called high scoring segment pair (HSP). Many such HSP's are included in the final BLAST result. Figure 2.11 shows a seed, 'PQG' and 'PMG', with score 53 (using PAM matrix) which is greater than the assumed threshold seed score of 50, extended on both sides. The threshold *'S'* is assumed to be 49. The length of the extension to the left of the seed is lower than to the right. The reason of this unequal extension on either side is because, after 3 character alignment extension on both sides of the seed, the score is 49 (Score +2 for match, mismatch and indels = -2), if extended to the fourth character alignment position to left and right, the overall score does not change. Since there is a positive score on the fourth character alignment extension to the right of the seed, extension is encouraged in this direction thereafter. Once the score falls below *'S'* after the fifth character alignment to the right of seed, extension is stopped.

Step 3 Extend the high scoring pairs by performing restricted dynamic programming locally around HSP. By extending around the HSP, we mean, extending with gaps, using Smith-Waterman algorithm until the score falls below a threshold. This

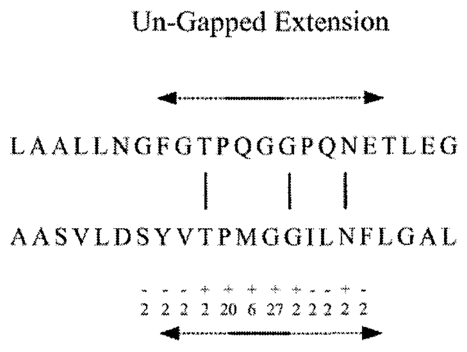


Figure 2.11: ***Extending the seeds***

step was not found in the original BLAST[2] but was added in GappedBLAST[3]. Using the high scoring pairs, BLAST was successful in fast database sequence searching.

Advantages and Disadvantages of BLAST The advantage of BLAST over FASTA is its speed due to heuristic extension of the seeds. The disadvantages are that it cannot find seeds smaller than the minimum length *'l'* considered for the exact match seed (DNA alignment) and reports only local alignments. It also finds too many seeds per sequence thus reducing speed (protein alignment) and does not allow for gaps in sequence. To overcome these disadvantages, BLAST2[3] was developed. The algorithm was changed by looking at two seeds at a distance 'd' which are then extended on both sides. The intuition behind this was that two smaller seeds are more likely than one longer one, therefore it is a more sensitive searching method.

2.1.2.3 BLAT-BLAST like Alignment tool

In this section, we explain how BLAT[18] aligns two sequences, S_1 and S_2, much faster than BLAST. The BLAT algorithm is similar to the BLAST and FASTA in that it first searches for seeds of fixed length *'l'*, and the final alignment is built around the seeds found. BLAT differs from BLAST in which sequence of the two sequences is indexed. BLAT builds an index of "non-overlapping seeds of S_2 database sequence and scans linearly through the S_1, whereas BLAST builds an index of S_1 and then scans linearly through the database"[18]. When aligning two sequences, the significance of BLAT is not very prominent. But, when the pairwise alignment solution is used for database searching, the significance of BLAT indexing is observed primarily because BLAT builds an index of non-overlapping seeds of S_2 database sequence and scans S_1 linearly. This implies that all non overlapping seeds from all the database sequences are preprocessed and only first sequence has to scanned, which saves considerable time. BLAT authors introduced a new seed which later came to be called as 'BLAT seed'. BLAT finds both perfect match seeds as well as BLAT seeds, and extends them in both direction similar to BLAST. After this extension, BLAT stitches them together to form a larger alignment[18]. In this section, we will explain BLAT seeds first and then explain the working of the algorithm.

Near Perfect Seeds: X_1 and X_2 of equal length m are considered as a near perfect match seed or BLAT seed if there exists a contiguous region X_2 in S_2 such that there are only r mismatches and the position where a mismatch is allowed is fixed. For example, in Figure 2.12, we see a seed with one mismatch allowed in the third position. If a seed exists in S_1 and S_2 such that the position where mismatch is allowed

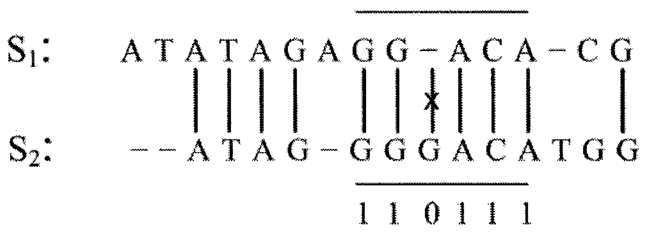


Figure 2.12: ***BLAT seed with one mismatch***

is actually a match, then it is also considered as a seed. The seed is characterized by ones and zeros. Ones represent the position where there should be a match and zero where a mismatch could be allowed.

Step 1 The BLAT algorithm first searches for perfect match seeds of length *'k'*, and considers perfect match seeds which are closer to each other, within a distance *'d'*.

Step 2 Indexing is done using the look-up table for S_2 unlike S_1 in BLAST. The seeds considered for indexing are all non-overlapping seeds. Sequence S_1 is searched linearly from left to right considering overlapping seeds.

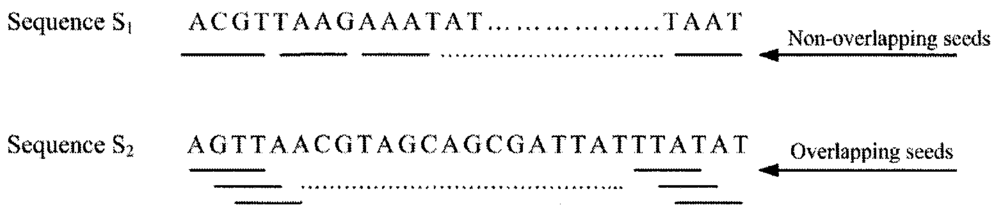

Sequence S1
ACGTTAAGAAATAT..................TAAT
Non-overlapping seeds
Sequence S2
AGTTAACGTAGCAGCGATTATTTATAT
Overlapping seeds

Figure 2.13: ***Indexing in BLAT***

Step 3 BLAT now searches for near perfect match seed defined above in both the sequences. These near perfect match seeds are again recorded by using a look-up table. When multiple seeds are within a distance *'d'*, they are extended to form a single seed[18].

Step 4 All seeds found in step 1 and step 3 are extended into high scoring segment pairs (HSPs), very similar to the HSPs found in the BLAST.

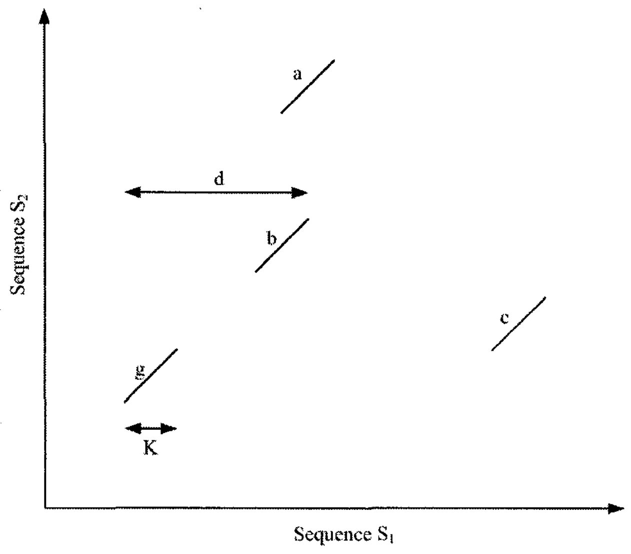


Figure 2.14: ***Multiple seeds in the same diagonal [18]***

Step 5 Stitching the HSP's is similar to the band created around the best diagonal discussed in FASTA algorithm with the only difference being there is no band in BLAT. In order to stitch the seeds, two adjoining seeds should be within a distance *'d'*. BLAT later considers the best HSP and reports the alignment.

From a database searching problem point of view, the main contribution of BLAT is its speed in searching. Indexing the database rather than the query sequence as in BLAST is the primary reason for the relatively high speed of BLAT.

Limitations: As in previous algorithms, BLAT is also limited in the sense that it cannot find small homologous regions because of the small seed length considered.

2.1.2.4 BLASTZ

BLASTZ[26] is the fastest algorithm in the BLAST series. In this section, we describe the working of the algorithm.

Step 1 In order to speed up the algorithm, all repeats in the sequences are removed. A repeat is a substring of same length repeated along the length of the sequence. The reason is that this algorithm primarily concentrates on aligning two long homologous DNA sequences as there are more likely to have more regions which match in both the sequences. Such regions are masked or ignored.

Step 2 Looks for all pairs of identical seed of length *'k'*, except for at most one transition. A transition is shift from one character to other. For example, in DNA sequences, transition from A-G, G-A, C-T or T-C. The earlier version of BLASTZ used a perfect matched seed of length 12. The look-up table is used to record the matches in both the sequences.

Step 3 Each seed is extended in both direction without gaps. The extension is stopped when the score drops below some threshold X, for example, X=3000. All the segments after the gapped alignment which score above, say 5000, are retained. Let us call all such segments as *'zones'*.

Step 4 For the regions between the zones aligned by the preceding steps, BLASTZ repeats these steps using a more sensitive seeding procedure (e.g., 7-mer exact matches) and lower score thresholds, say 2000 for gap free threshold and 2000 for gapped extension.

Step 5 BLASTZ finally adjusts the sequence positioning such that all masked segments could now also be included in the final alignment.

Limitations Although BLASTZ uses transitions seeds, the length of the seed being 12 is too long to find small regions of homology in divergent sequences. Hence BLASTZ fairs well with naturally evolving sequences but fairs relatively poorly with divergent sequences. Due to the seed length being 12, and only one transitions allowed in the entire seed length, BLASTZ finds too many seeds, and thus spends most of the time in calculating HSP's.

2.2 Global Sequence Alignment Algorithms

In this section, we will explain some of the global pairwise sequence alignment algorithms. We begin this section by explaining the optimal global sequence alignment algorithm, Needleman-Wunsch algorithm[23] and later explain popular heuristic global alignment algorithms.

2.2.0.5 PatternHunter

PatternHunter[21] introduces the concept of spaced seed to further improve the sensitivity and speed. PatternHunter uses a combination of different data structures including priority queues, a variation of data structure called red-black tree, queues, and hash tables to achieve its speed[9]. In this section, we first describe 'spaced seed' and later explain the working of the algorithm.

Definition 8 *X_1 and X_2 of equal length m are considered as a spaced seed, if there exist a contiguous region X_2 in S_2, such that there are minimum number of mismatches and the position where mismatches are allowed need NOT be fixed.*

$$(1\ 1\ 1\ 0\ 1\ 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 1\ 1\ 0\ 1\ 1\ 1\ ,\ 12)$$

Figure 2.15: ***Spaced seed model***

Figure 2.15 shows a spaced seed of length 19. The seed is represented by ones and zeros, ones representing the position where there should be a match and zeros where mismatches could be allowed. The last number 12 represents the seeds weight (number of ones in the seed). The difference between the BLAT seeds and Spaced seeds is that, in BLAT seeds, the mismatch position is fixed whereas in Spaced seed, the mismatch position is not fixed.

Step 1 The first step in this stage is building an index of the first sequence by moving the spaced seed window over the sequence from left to right, very similar to BLAST-sliding window. Using the look-up table, the first position where the model fits in the second sequence is recorded as a hit. Subsequent hits or positions where the model fits is recorded in another table, *'hit table'*[21]. For each hit, hits along its diagonal are considered and hits which are overlapping or to the right are ignored.

Step 2 The hits are extended to the right and left until the score of the segment falls below a threshold value *'t'*. This stage is very similar to the BLAST algorithm already described earlier. All such segment pairs, which have a score above the threshold value *'t'* are considered as High Scoring Segments Pair (HSP). The position of the last segment pair which reached the minimum threshold value *'t'* is stored, so that future HSP below it could be ignored.

Step 3 In this stage HSP's found in the previous stage are extended to the left. Before they are extended to the left, all the HSP's are diagonally sorted using a variation of red-black tree. Seeds of a smaller model 1101 in a limited length *'l'* from the left of the HSP are also stored along with other HSPs. "HSP's are inserted in the tree once an optimal gapped alignment to its left is found, and retired from the tree once newly generated HSP's are too far beyond its right end point to make use of it"[26]. This stage resembles the FASTA stage where many hits along the diagonal are found. The next step in this stage is to find the best diagonal. In order to connect the intermediate HSP's region with the HSP, a cost is computed using the affine gap penalty for the whole intermediate region. However, sometimes, two HSP's are diagonally overlapping, then the cost calculated is the affine gap penalty plus cost of shrinking the HSP in size to make a perfect fit[26] with the other HSP. In other words, overlapping HSP's are shrunk to make the best fit. This process is repeated to find the best HSP. In the end, the algorithm computes the optimal partial alignment score.

PatternHunter was implemented in JAVA, making it platform independent. In 2003, PatternHunter2 was introduced[19]. The objective of PatternHunter2 was to achieve 100 percent sensitivity and yet be faster than BLAST. PatternHunter2 uses multiple seed design or model instead of only one as in PatternHunter. In "two-hit mode, a gapped extension is performed only if two nearby hits are found on the same diagonal"[26]. Also, PatternHunter2 uses multiple hash tables for each of the seed considered.

The algorithm works better than the algorithms discussed earlier in terms of sensitivity. The speed of the algorithm is not better than BLAST as it is implemented in JAVA and incurs memory problems for long sequences.

2.2.1 Optimal Global Alignment Algorithm

Needleman-Wunsch algorithm[23] computes the optimal scores of alignments between all characters of sequences S and T similar to the Smith-Waterman algorithm using the same formula 2.1. Initially an empty matrix with first row and first column filled with zeros is constructed. The algorithm computes an optimal alignment between *S[1..i]* and *T[1..j]*, recursively. On completion of the matrix, the last row and last column cell of the matrix will have the optimal score, *opt(S, T)*. Using a similar trace back procedure used in Smith-Waterman algorithm, the final alignment is constructed.

Disadvantages: Needleman-Wunsch algorithm suffers from the same disadvantages as Smith-Waterman algorithm.

2.2.2 Heuristic Global Alignment Algorithm

Needleman-Wunsch algorithm is very slow and memory inefficient when comparing long sequences. To overcome these shortfalls, many heuristic algorithm have been proposed. Majority of the global sequence alignment algorithms explained in this section use a data structure called *'suffix tree'*. We first explain suffix tree, its advantages and its application before moving into global sequence alignment algorithms.

2.2.2.1 Suffix Tree Data structure

Suffix tree is a data structure that represents the "internal structure of a string in a comprehensive manner"[15]. The exact matching problem can be solved in linear time O(n), where n is the length of the string. Weiner[31] developed the first linear time suffix tree back in 1973. McCreight improved the Weiner's algorithm to achieve better space-complexity[22]. After two decades, Ukkonen built a linear time suffix-tree construction algorithm that incorporated all the benefits of McCrieghts algorithm and also offered a simpler implementation[30].

Definition 9 *A suffix tree τ for an m-character string S is a rooted tree with exactly m leaves numbered 1 to m. Each internal node, other than the root, has at least two children and each edge is labeled with a non-empty substring of S. No two edges out of a node can have edge labels beginning with the same character[15]. The key feature of the suffix tree is that for any leaf i, the concatenation of the edge-labels on the path* *from the root to leaf i exactly spells out the suffix of S that starts at position i, that is, it spells out S[i..m].*

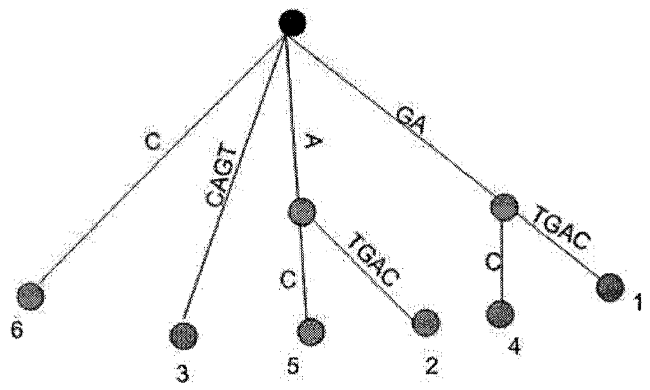


Figure 2.16: ***Example of suffix tree for GATGAC***

Label The label of a path from the root that ends at a node is the concatenation, in order, of the substrings labeling the edges of that path. The path-label of a node is the label of the path from the root to that node[15].

String Depth For any node *'i'* in a suffix tree, the string-depth of *'i'* is the number of characters in *i's* label.

Split A path that ends in the middle of an edge *(u, v)* splits that label on *(u, v)* at a designated point. A new node is introduced at the location of split[15].

2.2.2.2 Generalized Suffix Tree (GST)

Definition 10 *A generalized suffix tree is a suffix tree that combines the suffixes of a set of strings S_1, S_2 .., S_n.*

A generalized suffix tree can be constructed quickly for n strings. First, build the suffix tree for the first sequence, then starting at the root match the second sequence against a path in the tree until a mismatch occurs. At that point add the remaining characters of the suffix of second sequence to the suffix tree built for the first sequence. Let us take two strings S_1 = GATGA and S_2 = TATGTA. In figure 2.17, a leaf's

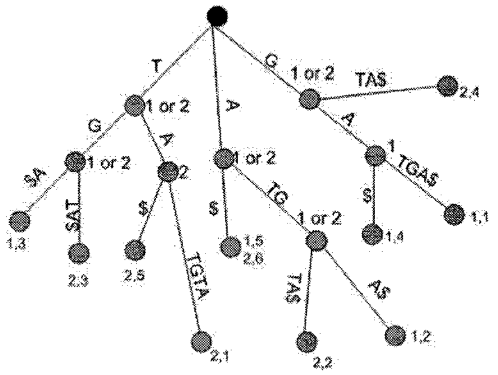


Figure 2.17: ***Example of generalized suffix tree for two string, S_1 and S_2***

label consists of two numbers. The first number represents the string number and the second number is the starting position of the suffix in that string.

2.2.2.3 Ukkonen Algorithm

Ukkonen's[30] linear time suffix tree construction algorithm brought many advantages over previous algorithms. The algorithm used suffix links to speed the building of the tree and is memory efficient. Our algorithm uses Ukkonen linear time suffix tree building technique. A detailed explanation of the Ukkonen algorithm is provided in the next chapter where we explain our algorithm.

Uniqueness in Ukkonen algorithm

- The algorithm begins at the root of the tree and constructs I_1 (the implicit suffix tree for just the first character) using the normal extension rules, that is, moving from left to right.
- The constructed suffix tree only has suffix links in the internal nodes of the tree (to save space).

2.2.2.4 Applications of Suffix Tree

The two popular applications of suffix tree are; solving longest common substring problem of two strings and all maximal repeats problem in a single sequence. We define these two applications below.

Definition 11 *Given two strings S_1 and S_2, the longest common substring is a substring that appears in both S_1 and S_2, and has the largest possible length.*

A generalized suffix tree (GST) is built for the two strings to obtain the largest common substring.

Definition 12 *A maximal repeat in a string S is a triple (i, j, l) such that S contains a repeat of length l starting at positions i and j, and this repeat cannot be extended further to the left or right.*

These repeats can occur either adjacent to each other (Tandem repeats) or apart, anywhere in the sequence. A GST can be used to find all maximal repeats in linear time. We now describe the working of some popular heuristic algorithms.

2.2.2.5 MUMmer

MUMmer stands for Maximal Unique Match-mer. A maximal unique match is a longest match that is found once in both the sequences. MUMmer was the first global alignment algorithm to align two long genomes. All previous algorithms could align genes and protein ranging up to few thousands in length[1], but either ran out of memory or were unacceptably slower when aligning genomic sequences. In addition, previous work on global alignment concentrated mainly to observe insertions, deletion and point mutations (change at a particular position in the sequences), but were not designed to look at large scale changes such as tandem repeats and large scale reversal. If there exists a substring SS_1 = 'ACGT', then a reversal of SS_1 is SV_1 = 'TGCA'. MUMmer makes use of suffix tree, longest increasing subsequence (LIS) and Smith-Waterman alignment for computation time and memory efficiency. A term 'SNP', meaning, a subsequence which appears in both the sequences but with a difference of only one base introduced. The algorithm has the following features:

1. Identify SNP's.

2. Identify regions of DNA where the two genomes differed by more than one SNP.

3. Identify regions where large segments of DNA were inserted in one genome.
4. Identify repeats, which are usually substring duplication.
5. Identify tandem repeats, substring repeats which had different number of copies in the genomes.

The working of the algorithm is as follows:

Step 1 Given two sequences, a suffix tree is built for the two sequences. Every unique matching sequence is then represented by an internal tree node with exactly two child nodes, whose child nodes are leaf nodes from different sequences. Finding maximal unique match will take one traversal from the root to all leaf nodes, or can be found in *O(n)*, where n is the length of the match. A maximal unique match is pictorially shown in Figure 2.18. The length of unique match MUMmer considerd is at least half of the length of the longest MUM found. For homologous regions, MUM is half the length of the longest MUM and for heterogeneous sequences, the length is varied. The rationale behind MUM being at least half the length of the longest is to remove small *'noisy'* matches (less than half) and to avoid potential fewer MUMs (more than half).

Step 2 Sort matches by their start locations in the first sequence from the suffix tree and extract the set of maximal unique matches (MUM) that occur in the same order in both sequences using a variation of the LIS (Longest-Increasing-Subsequence) algorithm[15]. This variation of the algorithm takes into account the lengths of the MUMs and allows them to overlap. It runs in O(K log K) time, where K is the number of MUMs. This step is pictorially shown in Figure 2.19.

Genome A: tcgat*c***GACGATCGCAGTAGGATGGATAAGCATAA***c*gact
Genome B: gcatt*a***GACGATCGCAGTAGGATGGATAAGCATAA***t*cca

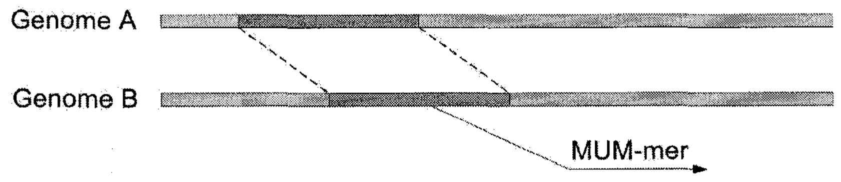


Figure 2.18: ***Maximal unique match in both the sequences***

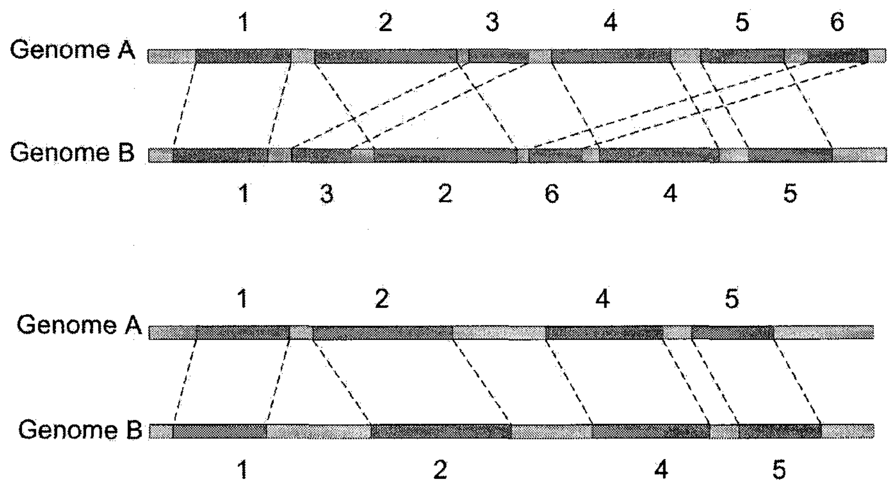


Figure 2.19: ***Consistent matches are selected which are in the same order in both the sequences***

Step 3 Generate Smith-Waterman alignments for all the regions between the MUMs. Once a global MUMmer is found, an overall global alignment is established. In these regions, MUMmer uses several algorithms for closing the local gaps and completing the final alignment. "A gap is defined as an interruption in the MUM-alignment which falls into one of four classes: (i) an SNP interruption, (ii) an insertion, (iii) a highly polymorphic region or (iv) a repeat"[1]. All four classes are pictorially shown in Figure 2.20. **SNP**: SNPs are found in two ways in the MUM alignment. In the simplest case, SNP is surrounded by MUMs. In some cases, however, an SNP is adjacent to sequences that are not unique. In such cases, the adjacent sequence and the SNP is captured and processed by the repeat processing procedure described below.
Insertions/ Deletions : Insertions are regions that appear in one sequence but does not appear in the other. These are large gaps in the alignment in one sequence and not in the other. The insertions or deletions are done without the use of Smith-Waterman algorithm or any other algorithm. Insertion are of two types, transpositions, that is, sub-sequence is deleted in one region of the sequence and inserted in other region of the sequence and simple insertions which appear in only one sequence. Simple insertions could be due to simple deleting, or other evolutionary process.
Polymorphic regions : Regions in between MUMs that do not align, but still should be aligned in the whole genome alignment. If such regions are small, MUMmer uses optimal algorithm to align such regions.
Repeats : MUMmer does not display substring repeats as the alignment is based on unique matches only. However, authors observed that repeat sequences were adjacent to unique sequence, and the MUM on either end of a tandem repeat extended into the repeat itself[1]. For example, Figure 2.21 shows there are two tandem repeats after

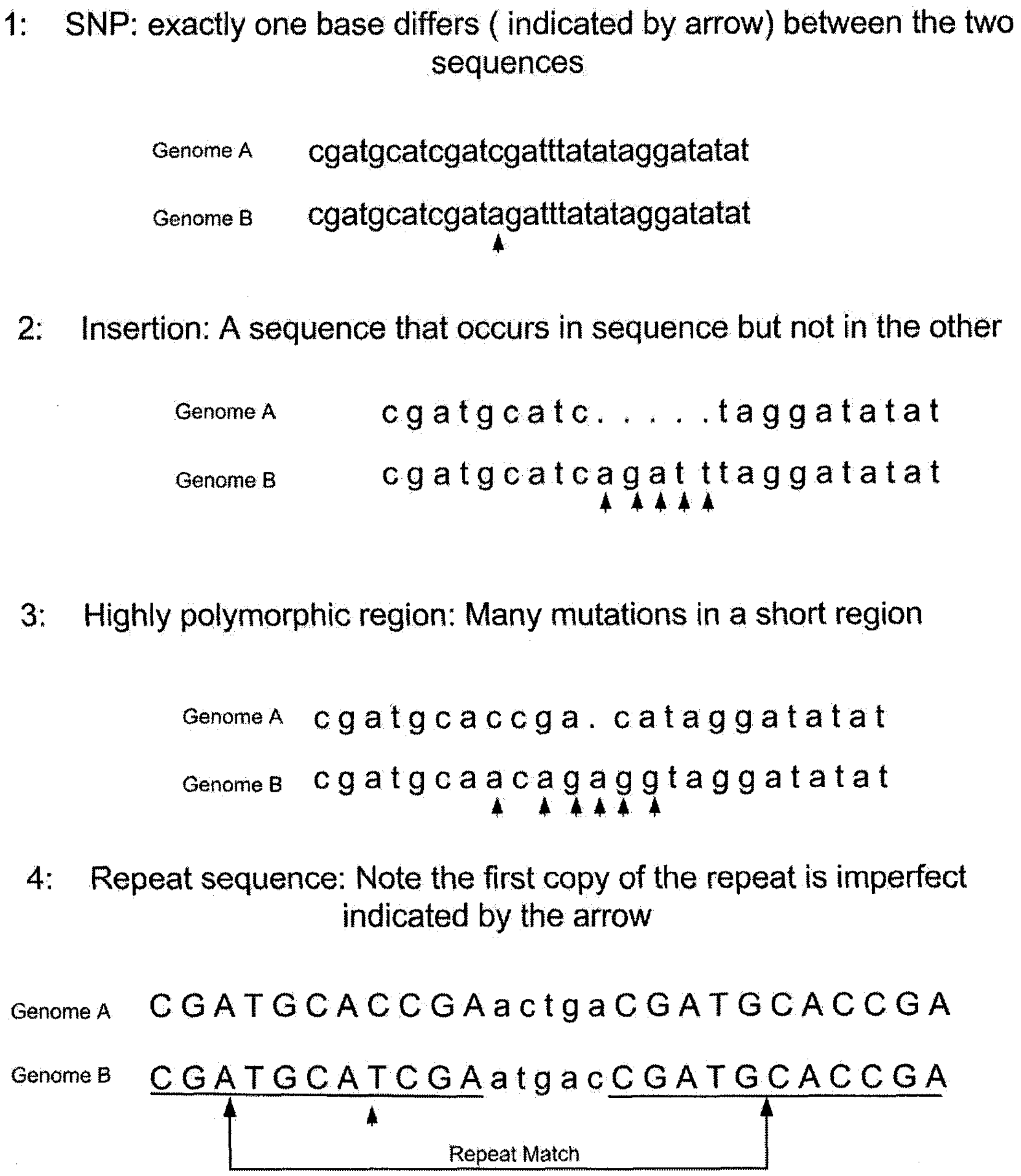

1: SNP: exactly one base differs (indicated by arrow) between the two sequences
Genome A cgatgcatcgatcgatttatataggatatat
Genome B cgatgcatcgatagatttatataggatatat
2: Insertion: A sequence that occurs in sequence but not in the other
Genome A c g a t g c a t c . . . . . t a g g a t a t a t
Genome B c g a t g c a t c a g a t t t a g g a t a t a t
3: Highly polymorphic region: Many mutations in a short region
Genome A c g a t g c a c c g a . c a t a g g a t a t a t
Genome B c g a t g c a a c a g a g g t a g g a t a t a t
4: Repeat sequence: Note the first copy of the repeat is imperfect indicated by the arrow
Genome A C G A T G C A C C G A a c t g a C G A T G C A C C G A
Genome B C G A T G C A T C G A a t g a c C G A T G C A C C G A
Repeat Match

Figure 2.20: *4 types of algorithms used in the inter MUM region*

the MUM: (i) uniqueAAGGAAGG and (ii) AAGGAAGGsequence are overlapping and are repeats. Four gaps could appear anywhere between positions 6 and 14 in the alignment. MUMmer always inserts the gaps in the rightmost position. The MUM in the final alignment would indicate that MUM (i) occupies positions 0...13, and MUM (ii) occupies positions 10...25 in Genome A. The fact that these two intervals overlap indicates a tandem repeat.

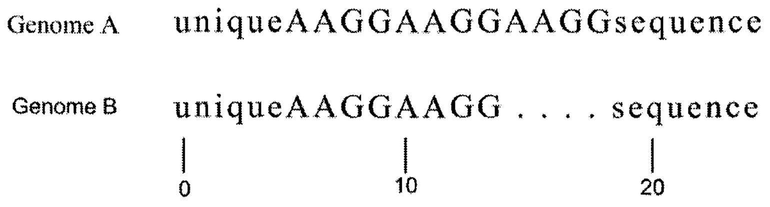


Figure 2.21: ***Tandem repeat in the inter MUM region[1]***

Step 4 Output the alignment, including all the matches in the MUM alignment as well as the detailed alignments of regions that do not match exactly.

Limitations MUMmer is heavily dependent on the unique matches it finds in the first step; if very few MUMs are found, the algorithm performs poorly. The requirement that two inputs sequences being homologous indicates that the algorithm is not flexible. In case of heterogeneous sequences, differing more than 30 percent, even if the minimum length of the MUM is reduced to 20 percent of the longest MUM, the inter MUM would more likely be aligned using Smith-Waterman algorithm, drastically increasing the computation time.

2.2.2.6 GLASS, GLobal Alignment SyStem

GLASS[6] was developed to align hundreds of kilobases of genomic sequence. It was primarily developed to overcome the limitations of standard dynamic programming (SDP) methods which had their running time scale in proportion to $O(nm)$ (where n and m are the lengths of the genomic sequences compared) and were not sensitive to finding short regions of good alignment between much longer regions of poor alignment. This program uses a hashing technique and computes a global alignment recursively by finding long segments that match exactly and whose flanking regions have high similarity. The working of the algorithm is as follows:

Step 1 Find all common K-mers of length 'K' that appear in both sequences.

Step 2 Use a hash technique, map each matching K-mer to a unique character and convert these matches in both sequences into strings of characters. The alphabets of these characters must be different from that of the letters in the other sequences. Let us say a symbol % such as inserted for a match in DNA sequences made of characters A, C, G, T. An example is shown in Figure 2.22.

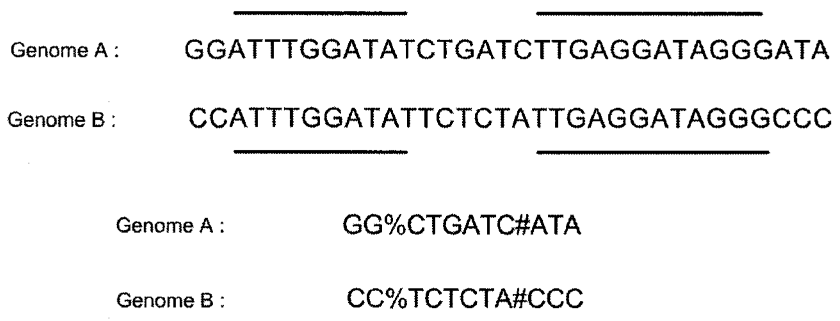
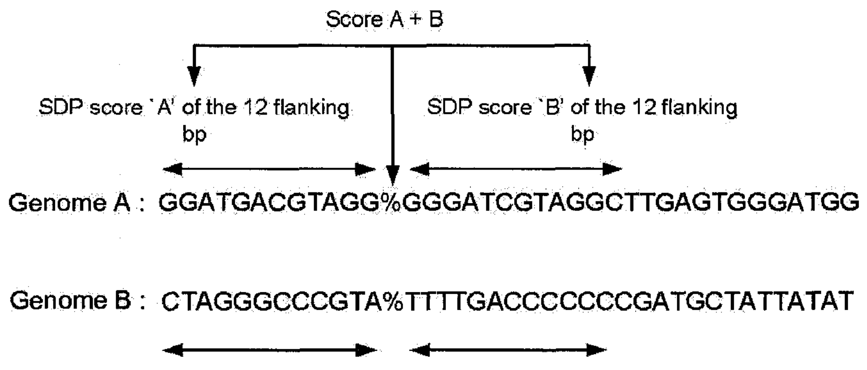

Genome A : GGATTTGGATATCTGATCTTGAGGATAGGGATA
Genome B : CCATTTGGATATTCTCTATTGAGGATAGGGCCC
Genome A : GG%CTGATC#ATA
Genome B : CC%TCTCTA#CCC

Figure 2.22: ***Converting a k-mer to a unique character after hashing the k-mer***

Step 3 Apply the standard dynamic programming algorithm to the short flanking regions (12 bps) on both ends of each matching K-mer and compute two scores. Each K-mer receives a score equal to the sum of these two scores. This step is pictorially represented in Figure 2.23.



Score A + B
SDP score 'A' of the 12 flanking bp
SDP score 'B' of the 12 flanking bp
Genome A : GGATGACGTAGG%GGGATCGTAGGCTTGAGTGGGATGG
Genome B : CTAGGGCCCGTA%TTTTGACCCCCCCGATGCTATTATAT

Figure 2.23: ***Apply SDP on 12 bps on either side of the match***

Step 4 Take only 'consistent' K-mers whose score exceeds a threshold *T*. Two K-mers are inconsistent if they are overlapping or criss-crossing.

Step 5 Recursively aligns the intervening regions using a smaller value of K. The value of K is 20, 15, 12, 9, 8, 7, 6 and 5. The value of K is decided empirically. Once recursive procedure is performed, GLASS "extend all pairs of aligned segments by short local alignments to the left and right by SDP"[6]. Finally, align the remaining (usually short) unaligned regions using SDP.

Limitations GLASS recursively align the two sequences hence, it is slower than MUMmer. However, as it considers short K-mers, it is more sensitive than MUMmer.

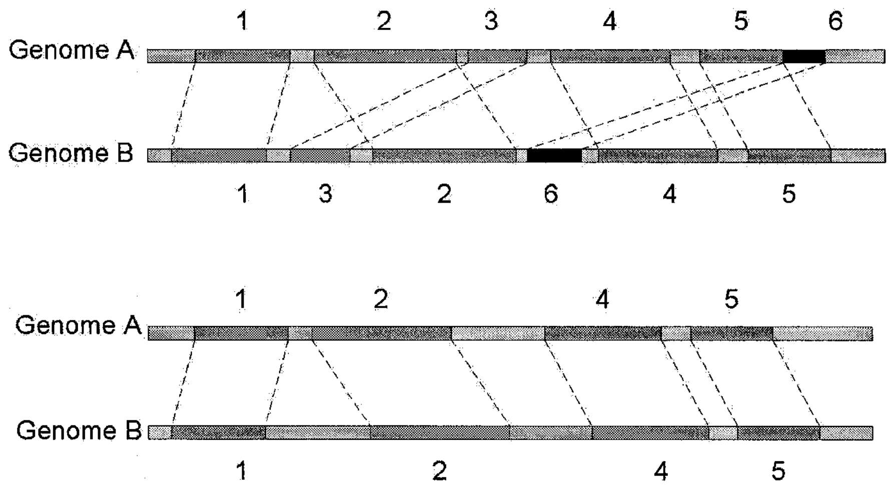


Figure 2.24: ***Shows seeds 2 and 3 criss-crossing and seed 6 (in blank) overlapping with 5***

GLASS neglects K-mers of size shorter than what is considered in the algorithm. The removal of overlapping and crossing seeds makes the algorithm less efficient for detecting trans-positions and reverse seeds.

2.2.2.7 AVID

AVID[7] attempts to balance both speed and sensitivity when aligning very long sequences. To achieve better computational efficiency, it uses suffix tree and considers overlapping anchors. To improve sensitivity, it uses a variant of Smith-Waterman algorithm, in the inter anchor region. The algorithm works as follows:

Step 1 The algorithm starts by concatenating two sequences by placing a special character N between them. A maximal repeat substring is a longest substring which is repeated in both the sequences. A maximal repeat in this string that crosses the boundary between the two sequences represents a maximal match between the two sequences. All such maximal repeats are found by browsing all nodes in the suffix tree only once. Thus, time taken is $O(n)$, where n is the length of the concatenated string.

Step 2 AVID removes matches that are less than half the length of the longest match found. AVID then sorts the matches by length with consistent matches appearing first.

Step 3 A variant of the Smith-Waterman algorithm is used to select anchors from the matches found in the previous stage. Every match is evaluated based on its length and alignment scores of its two flanking regions (10 bp on each side). This is similar to the idea first employed in the GLASS algorithm.

Step 4 From the anchors in the previous stage, AVID picks only those anchors which score above a threshold.

Step 5 AVID determines whether each match is entirely between two sets of anchors. Shorter matches removed in step 2 and repeat matches in step 4 are considered at this stage. Smaller inter-anchor regions are realigned using the anchor selection step recursively.

Step 6 For short regions, AVID uses the Needleman-Wunsch algorithm to get the final global alignment. The above 6 steps are shown in Figure 2.25.

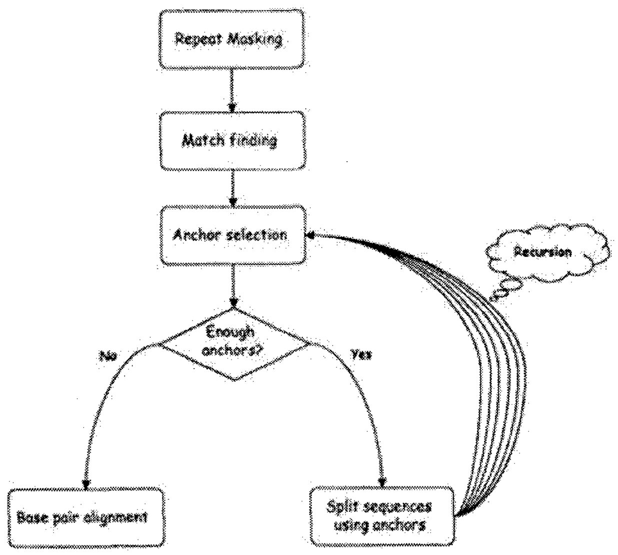


Figure 2.25: ***AVID Algorithm, courtesy:*** *[7]*

Limitations AVID is not sensitive when aligning distantly related sequences. This is partially due to the heavy dependence on maximal repeat substring. In case of divergent sequences, not many maximal repeat substrings are found, which makes the alignment bank on the local sequence alignment step in the inter anchor region for the final alignment. This in turn would be computationally expensive if two anchors are relatively closer or would be less sensitive, when two anchors are far away.

2.2.2.8 LAGAN-Limited Area Global Alignment of Nucleotides

LAGAN[10] is more sensitive than previous pair-wise global sequence alignment algorithms discussed in this thesis. LAGAN is an efficient and reliable pairwise aligner that is suitable for genomic comparison of distantly related organisms. LAGAN does that by finding small regions of local similarity first and then chaining them to produce the overall global alignment. The algorithm works as follows:

Step 1 CHAOS algorithm[11] is used to generate local alignments between the two sequences. The advantage of using CHAOS algorithm is that "it finds local alignments using multiple short inexact words instead of the longer exact words"[11] used by MUMmer, GLASS and AVID, Figure 2.27b. "Given a maximum distance d and maximum range s, two local alignments or anchor x and y in the two sequences, can be chained together if the indices of x (starting address) in both sequences are higher than the indices of y, and x and y are 'near' each other"[11] with 'near' defined by both a distance and a gap criteria as shown in Figure 2.26. The chaining is a global alignment between x and y. "The final score of a chain is the total number of matching bp in it. The default parameters used by CHAOS are words of length 10, a distance and gap criteria of 20 and 5 bp respectively"[11].

Step 2 Construct rough global map by maximizing the weight of a consistent chain of local alignments using the LIS algorithm which is also used by MUMmer, Figure 2.27c. A local alignment is chained to the previous one that produces the highest scoring chain among all chains that end with this alignment. Apply first two steps recursively between every pair of anchors that are separated by more bases than a threshold.

Step 3 Compute the optimal Needleman-Wunsch global alignment within the range 'r' from the anchors, to get the final alignment (Figure 2.27d).

Limitations LAGAN uses local sequence alignment to first find seeds of length 'k' using CHAOS algorithm and in the next step uses optimal global alignment to fill the inter seed regions; these two steps together are computationally expensive when

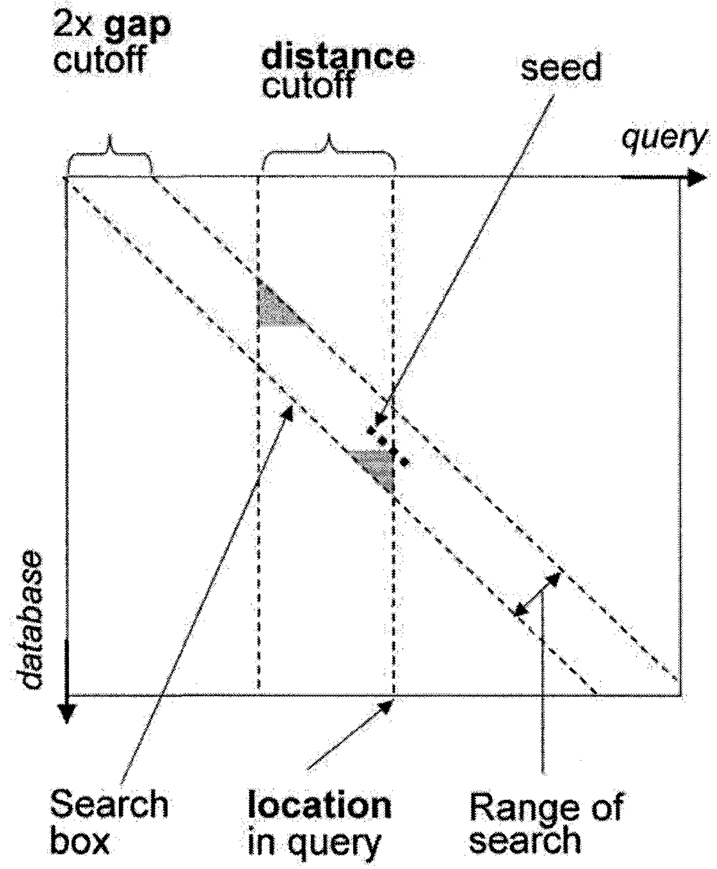


Figure 2.26: *LAGAN Algorithm [10]*

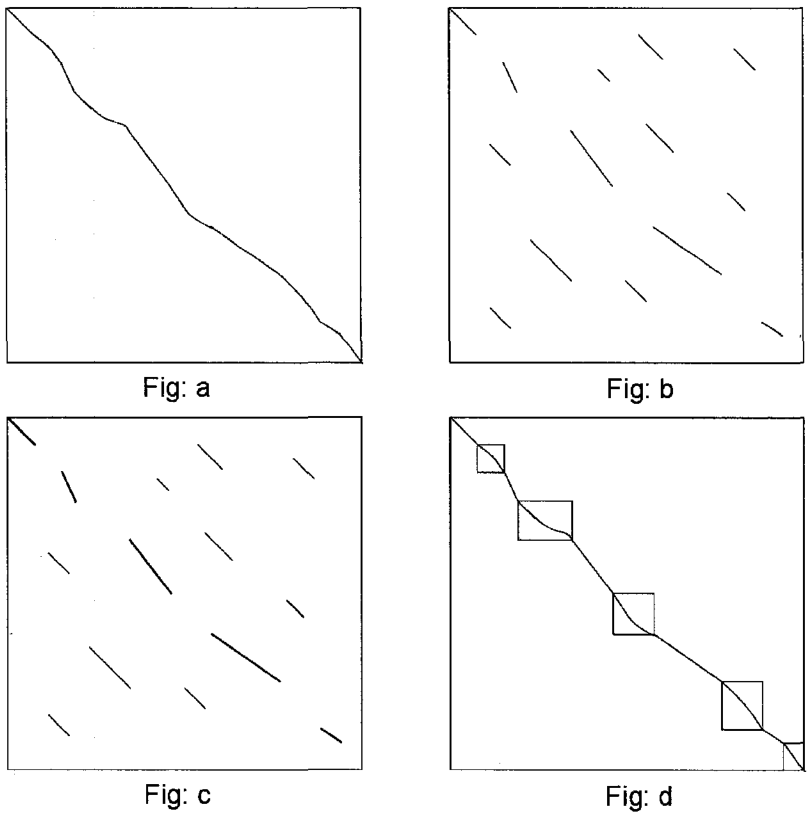


Figure 2.27: *The LAGAN algorithm. (A) A global alignment between two sequences is a path between the top-left and the bottom-right corner of their alignment matrix. (B) LAGAN first finds all local alignments between the two sequences. (C) LAGAN computes a maximal-scoring ordered subset of the alignments, the anchors, and puts together a rough global map. (D) LAGAN limits the search for an optimal alignment to the area included in the boxes and around the anchors, and computes the optimal Needleman-Wunsch alignment limited to that area*

compared to MUMmer or AVID. However, the sensitivity of this algorithm is good when compared to MUMmer, AVID or GLASS as it picks short subsequences and later stitches them for the final alignment.

Summary From the literature, we see local alignments algorithms first find regions of similarity (seeds or anchors), expand the seeds on both sides to get a substantial bigger similar region (high scoring pairs or HSP's) and then stitch these HSP's using either a variant of optimal algorithm or other heuristic algorithm to get the final alignment. Also, Hash table is the primary data structure used for seed searching for all local sequence alignment algorithms. Different seeds (BLAT, spaced seed, perfect match seed and seeds above a threshold) are proposed for local sequence alignment algorithms. Many local sequence alignment algorithms are not designed for pair wise sequence alignment but for searching similar database sequences. Considering speed to be the main objective of many local sequence alignment algorithms, the type of seed considered for the algorithm is one of the main contributor for varying sensitivity while the rest of the algorithm is quite common to most local sequence alignment algorithms.

Global sequence alignment algorithms on the other hand follow a similar technique as local sequence alignment algorithm in that, all global alignment algorithms start by first searching similar regions or seeds but some seeds are later selected to be part of the final alignment: anchors. These global alignment algorithms have used suffix trees wisely to improve their computational time. Recent algorithms use suffix tree to search similar regions, out of these regions some are selected as anchors, and the inter anchor regions are aligned using dynamic programming algorithm or other methods.

Some algorithms using the above technique are AVID, MUMmer and GLASS. LAGAN on the other hand local alignment between the two sequences. These local alignments are treated as seeds and the inter seed regions are stitched to get the final alignment.

In the next chapter, we explain our proposed algorithm for pairwise local sequence alignment.

Chapter 3

Multiple Anchor Staged Local Sequence Alignment Algorithm - MASAA

In this chapter, we explain in detail our proposed algorithm. Ukkonen online suffix tree construction algorithm[30], forms the first initial step of our algorithm. We begin this section by explaining the Ukkonen suffix tree building algorithm.

3.1 Ukkonen Online Suffix tree Algorithm

To aid the understanding of our proposed algorithm, we first present some terminology related to suffix trees. Let *S[0..N]* be the string indexed by the tree *T*. The leaf node corresponding to the *i-th* suffix, *S[i..N]*, is represented as l_i. An internal node, *v*, has an associated length *L(v)*, which is the sum of edge lengths on the path from root to *v*. We represent by *σ(v)*, the string at *v* to represent the substring *S[1..i + L(v)]* where l_i is any leaf under *v*. The suffix tree for an example, S = MISSISSIPPI is shown in Figure 3.1. The numbers at the bottom of leaf nodes represent the start of the suffix *S[i..N]* that they represent. From definition 12, we know that a suffix tree

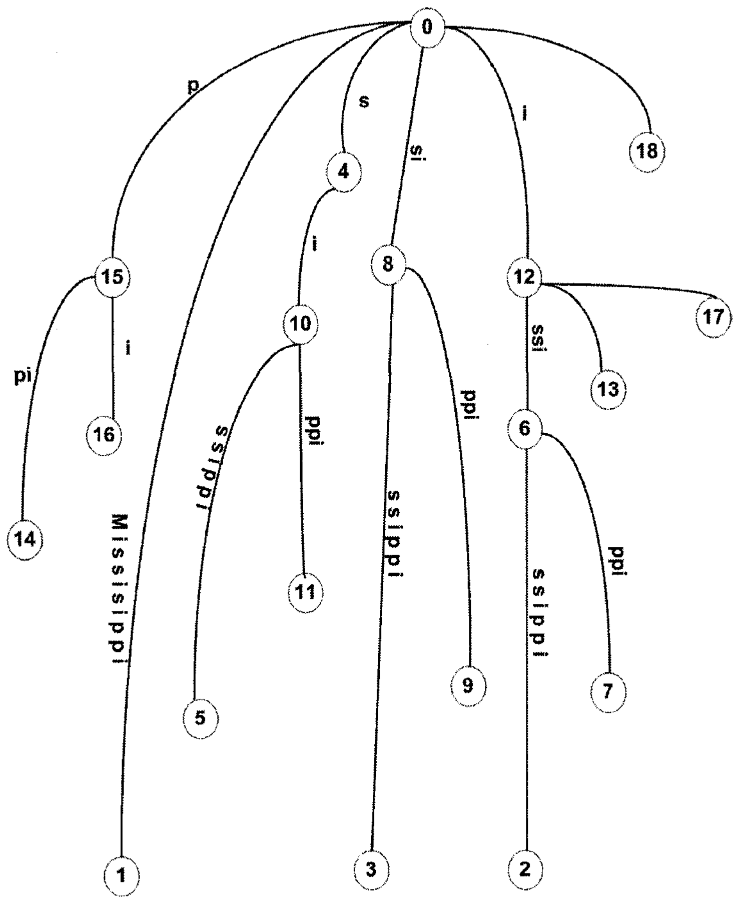


Figure 3.1: ***Suffix tree for 'MISSISSIPPI'***

T for an *m*-character string *S* is a rooted tree with exactly *m* leaves numbered 1 to *m*. The suffix tree is constructed incrementally by scanning the string from left to right, one character at a time. That is, suffix tree is built in m phases, one for each character. At the end of phase i, we will have tree T_i, which is the tree representing the prefix *S[1..i]*. In each phase i, we have i extensions, one for each character in the current prefix. At the end of extension j, we will have ensured that *S[j..i]* is in the tree T_i. There are four possible ways to extend *S[j..i]* with character *i+1*.

1. *S[j..i]* ends at a leaf. Add the character *i+1* to the end of the leaf edge.

2. There is a path through *S[j..i]*, but no match for the *i+1* character. Split the edge and create a new node if necessary, then add a new leaf with character *i+1*.

3. There is already a path through *S[j..i+1]*.

4. Do nothing.

The algorithm can be viewed to consist of two phases, Locate phase and Extension phase, for each character in the sequence.

Definition 13 *"Let $a\alpha$ denote an arbitrary string, where a denotes a single character and α denotes a (possibly empty) substring. For an internal node v with path-label $a\alpha$, if there is another node sl(v) with path-label α, then a pointer from v to s(v) is called a suffix link. A suffix link sl(v) = w exists for every node v in the suffix tree such that if $\sigma(v) = a\alpha$, then $\sigma(w) = \alpha$, where a is a single character of the alphabet and α is a substring (possibly null) of the string. Note that sl(v) is defined for every node in the suffix tree. And, more importantly, sl(.) - the entire set of suffix links, forms a tree rooted at the root of T, with the depth of any node v in this sl(.) tree being L(v)"[15].*

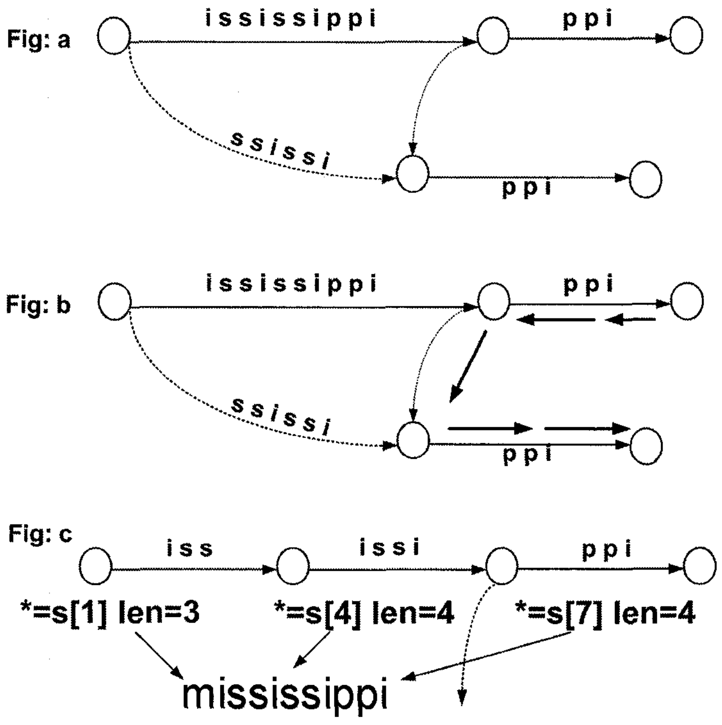


Figure 3.2: *Speeding up steps to build the Suffix tree*

The suffix tree for S = MISSISSIPPI with dashed edges between internal nodes representing suffix links is shown in Figure 3.3. In order to be fast and memory efficient, Ukkonen algorithm employs the following:

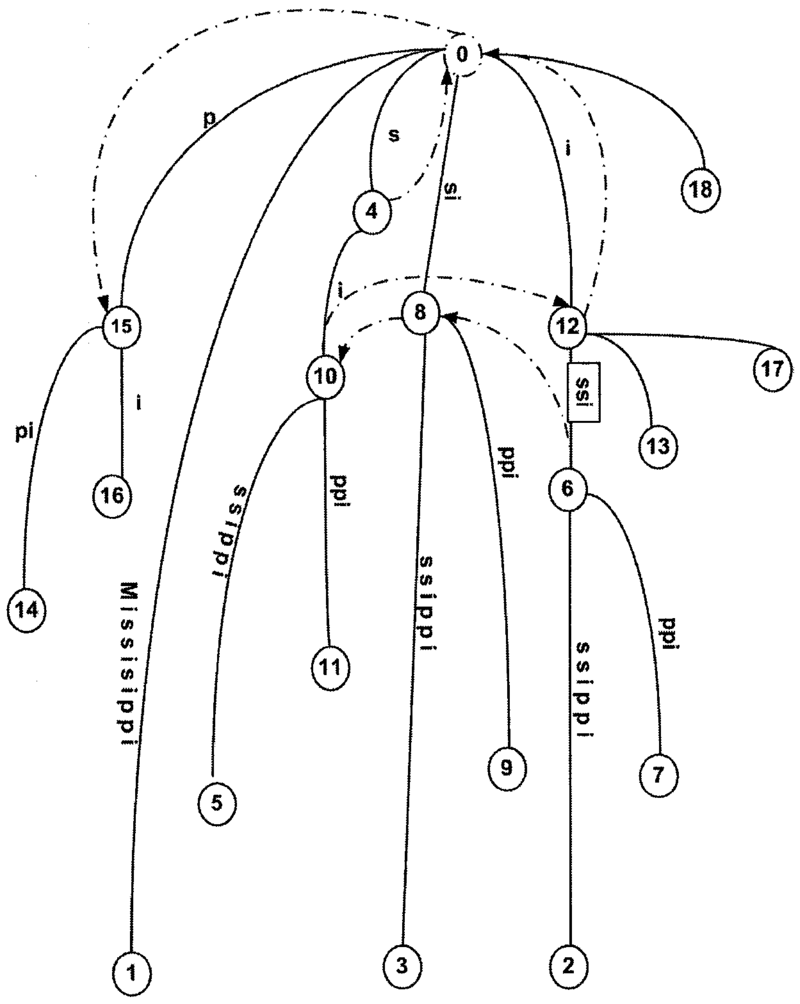


Figure 3.3: ***Suffix links in the suffix tree***

Step 1: The tree is augmented with additional edges, called suffix links, that provide shortcuts to move across the tree quickly. These suffix links play a crucial role in reducing the running time of the algorithm.

Step 2: Skip/Count Trick as it is called: instead of stepping through each character, we know that we can just jump, as long as the tree has common substrings. In other words, there are two branches having common substrings at different places, one can jump from one branch to the other branch, as shown in Figure 3.2b.

Step 3: Edge-Label Compression, since we have a copy of the string, we do not need to store copies of the substrings for each edge as shown in Figure 3.2c.

Step 4: A match is a 'show stopper', meaning, If we find a match to our next character, we do not have to do anything as the substring now is already part of the built tree.

Step 5: Once a leaf, always a leaf. We do not need to update each leaf, since it will always be the end of the current string[15].

A pseudo code for the Ukkonen algorithm is shown below[15].

```
• input $S[0..m]$ : string to be indexed
• $I_0$ - Implicit suffix tree for $S[0 ... 0]$
• for $i = 0$ to $m$ do
    – for $j = 0$ to $i + 1$ do
        ∗ LOCATE PHASE
        ∗ Locate $\beta = S[j ... i]$ in $I_i$
        ∗ EXTENSION PHASE
        * if $\beta$ ends at a leaf then
          $I_{i+1}$, add $S_{i+1}$ to $I_i$
          else
          $\beta$ ends at an internal node, or the middle of the edge
            · if from the end of $\beta$ there is no path labeled $S[i + 1]$ then
            · $I_{i+1}$, split edge in $I_i$ and add a new leaf else
            · $I_{i+1} < I_i$, $\beta$ already exists in $I_i$
            · end if
        * end if
    – end for
• end for
```

3.2 Contribution

All the pairwise alignment algorithms discussed in the previous chapter essentially use either some form of seeds (for example, repeat match, unique match, contiguous seed, spaced seed, vector seed, etc.) or maximum match subsequences (MMSS) as the base for the alignment. Although these two approaches (seed based and MMSS based) seems to be similar, they are not the same. A seed could be an MMSS, but the converse need not be true. It is quite possible that a seed based approach could fail to extract an MMSS as a seed in the alignment. For example, the underlying MMSS, shown below, is not found using BLAST alignment algorithm.

AAATACATACTTAGGCTCAAAACGCACTGTTTAATAAAA

GTTAGGCCCCTGTTTAATTAGGCTCCCCCCCGGGGGGCC

Based on our observation and from the literature, we feel that MMSSs are more likely to be conserved regions and therefore must appear in the final alignment.

Among the global sequence alignment algorithms, AVID uses the MMSS as its base elements for alignment. However, no local sequence alignment algorithm uses the longest common substring (LCSS) as its base element for alignment. It is our objective to use MMSSs as the primary base element and, in the regions between MMSSs, use BLAT seed[18] as the secondary base elements for the alignment. In this way, local sequence alignment can be strengthened to detect even weakly conserved regions. This is the main motivation for our local sequence alignment algorithm.

3.3 MASAA - Multiple Anchor Staged Alignment Algorithm

The objective of local alignment algorithm is to find similar subregions of significant sizes within given two sequences and align them. Among the similar subsequences, a subset is identified and anchored for possible extensions. Some of these anchors are expected to be a part of the final alignment. Once a set of subsequences are anchored, then the anchors of size greater than a threshold value are extended to form the final alignment.

Subsequences are identified and anchored in two rounds. In the first round, all the MMSSs of size greater than or equal to a threshold value are identified and then a subset of them are anchored. This process is relatively fast and that improves the overall computational time of the algorithm. To improve sensitivity, in the second round, mismatch seeds are identified and a subset of them are anchored.

The algorithm is implemented in five logical steps: (i) finding MMSSs; (ii) selecting MMSS anchors; (iii) finding mismatch seeds; (iv) selecting mismatch seed anchors; and (v) extending anchors. We elaborate these five steps below.

3.3.1 Finding MMSSs

To find MMSSs, we use a suffix tree similar to the one used in AVID[7]. Initially, two strings are concatenated by placing a character N in between them. Now, the problem of finding all matching substrings between two sequences is transformed into a problem of finding maximal repeated substrings in the concatenated string. Such maximal repeated substrings (i.e., MMSS of original strings) of lengths greater than or equal to a threshold value δ are found using suffix tree of the concatenated string. For the current implementation, we have fixed $\delta = \frac{l}{2}$, where l is the length of the longest MMSS. $\delta = \frac{l}{2}$, strikes a balance between short 'noisy' MMSS's ($\delta < \frac{l}{2}$) and long MMSSs ($\delta > \frac{l}{2}$) [7].

3.3.2 MMSS Anchors Selection

In this step, the algorithm starts selecting anchors from the MMSS set formed in the previous step. The algorithm uses a simple technique to select such anchors. Let us label the MMSS's as $M_1, M_2, ...M_n$, starting from left to right. The algorithm starts with the first pair (M_1, M_2). If there is no crossing and overlapping, M_1 is included in the anchor set and (M_2, M_3) is selected as the next pair to be examined. Otherwise, the pair (M_1, M_2) is ignored and (M_3, M_4) is selected as the next pair to be examined. The process continues, from left to right, until the last pair (M_{n-1}, M_n) is examined. This simpler technique is relatively faster and seems to capture most important MMSS anchors.

3.3.3 Finding Mismatch Seeds

Once MMSS anchors are selected, the focus shifts to the inter MMSS anchor regions. For an initial value of k, we find all matching k-mers in this region. The match is defined by a BLAT seed[18], with $k = 12$, tolerating 4 mismatches. The mismatches is set at 4 as it increases the number of the anchors found. This process serves two purposes:

1. The matches which were lost in the MMSSs anchor selection step due to overlapping and crossing are found again as seeds, and

2. The mismatch seeds can find smaller regions of similarity.

A mismatch seed is found in two steps.

1. Find smaller seeds of size up to k

2. Extend each of them on both sides to become k sized mismatch seed if the size of the seed is less than k

Consider the following example region between two MMSS anchors.

$MMSS_{11}$...GGGCC$\underline{\text{TACTTAGC}}$GCTAAAACGCAAAAA...$MMSS_{12}$

$MMSS_{21}$...GTTA$\underline{\text{TACTTAGC}}$TCCCAAAACGCCTTAGG...$MMSS_{22}$

In this example, TACTTAGC and TACTTAGC is a match in the region between two MMSS anchors. They are extended to form CCTACTTAGCGC and TATACTTAGCTC mismatch seed pair. Similarly, other mismatch seeds are found in the remaining part of the current inter MMSS anchor region, for example, AAAACGC. The same process is repeated for all other inter MMSS anchor regions.

3.3.4 Mismatch Seed Anchors Selection

In the literature, many algorithms use different heuristics to choose a subset of matches to anchor. We identify anchors from non-overlapping, overlapping, non-crossing, and crossing matches. Identifying anchors from non-overlapping and non-crossing matching is relatively straightforward. To identify anchors from overlapping and crossing matches, we use the heuristic of "closeness". In overlapping mismatches, if the length of the overlaps on both matches is same then they are merged into a single non overlapping match and therefore included as an anchor. Crossing mismatches brings us four cases.

Case 1: When crossing mismatches are at different distance and have same number of matching bps, mismatch seed closer to each other is selected.

Case 2: When crossing mismatches are at same distance and have same number of matching bps, either one is selected. We choose the left most seed in sequence one.

Case 3: When crossing mismatches are at same distance and have different number of matching bps, mismatch seed with maximum matching bps is selected.

Case 4: When crossing mismatches are at different distance and have different number of matching bps, mismatch seed with maximum matches is selected.

Eventually, all selected anchors are ordered from left to right.

3.3.5 Extending Anchors

The MMSSs anchors found in step 3.3.2 form part of the final alignment. These anchors are extended on both sides without any involvement of Smith-Waterman algorithm. The extension starts with the longest MMSS. The mismatch anchors on both sides of this MMSS anchor facilitate the extension process. The extension is done as follows. If the neighboring MMSS anchor is within the distance d bp, then it is extended up to the neighboring MMSS. Otherwise, it is extended up to d bp. This is done on both sides. Then the next longest MMSS is chosen and extended. The algorithm terminates when all MMSS are extended. A pictorial representation of this stage is shown in Figure 3.4 where extension starts from MMSS (i). The algorithm ensures that longest common substring or the longest MMSS will be a part of the final alignment. BLAT seeds and the MMSS are determined linearly from left to right in both the sequences. BLAT seeds with 4 mismatches are used, primarily to remove the assumption that amino acid substitutions at neighboring sites are uncorrelated to a degree. BLAT seed consideration in-between MMSS regions imposes additional computation but captures information that could increase remote homology detection. However, our algorithm is not designed to catch homologies which are very distant

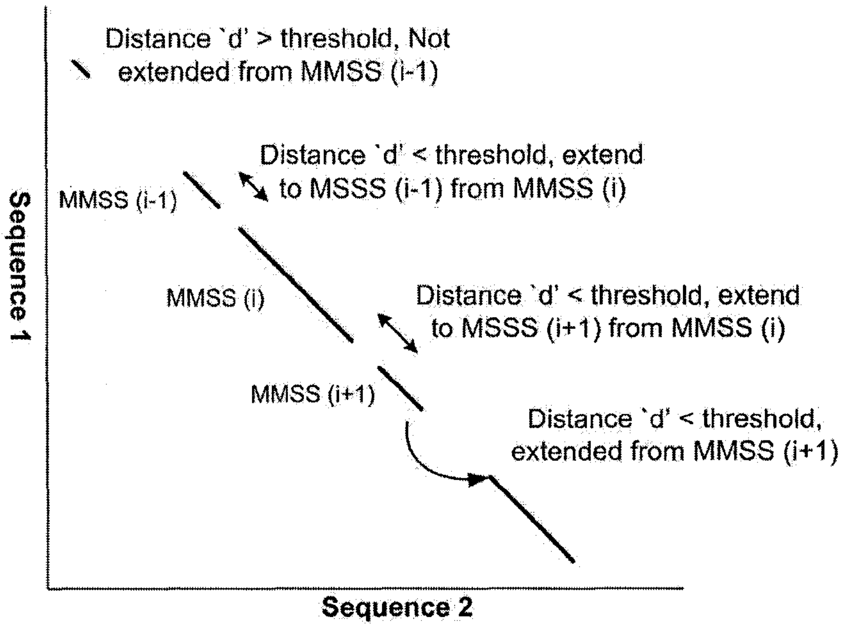

Distance 'd' > threshold, Not extended from MMSS (i-1)
Distance 'd' < threshold, extend to MSSS (i-1) from MMSS (i)
MMSS (i-1)
MMSS (i)
Distance 'd' < threshold, extend to MSSS (i+1) from MMSS (i)
MMSS (i+1)
Distance 'd' < threshold, extended from MMSS (i+1)
Sequence 1
Sequence 2

Figure 3.4: ***MMSS's extension***

to each other in terms of their position in the sequences. When two crossing seeds are considered in both the sequences, nearest seeds are chosen, as there is a greater probability that the mutation occurs at a relatively same region in both the sequences.

3.3.6 Implementation

The algorithm was implemented in C language. C was primarily chosen for speed over other languages like JAVA. JAVA due to the inherent virtual machine, could bring down the computation efficiency in terms of speed. We next explain the implementation of our algorithm.

Consider two sequences, S1 = TATAA and S2 = AACGA. The objective is to align these two sequences. A special character λ is added to the first sequence and the two sequences are then concatenated. S1 = TATAAλAACGA. We then build online suffix tree to the concatenated string and find longest matching substrings or maximal match substring (MMSS). This is done by finding internal nodes which have at least two children, one child branch's substring having λ and other without λ. We then pick the starting address of the two respective branches and add them to a list. The structure of the node is built in a way that it holds the starting address and ending address of edge. Both the addresses are the indexes with respect to the sequence S1, that is, the concatenated string. Once the MMSS's are found, they are sorted according to MMSS position in the first sequence, that is, from left to right. For example, if MMSS1 position in the sequence is x, then the next MMSS2 position in sequence would be y, where $y>x$. The sorting algorithm used is quicksort. The inter anchor regions are scanned linearly from left to right to find smaller anchors. The final alignment is then recorded after all smaller anchors are found.

The main component of the algorithm is the Ukkonen suffix tree, which is fast and memory efficient. In the next section, we will review the complexity of the algorithm.

3.3.7 Complexity of the Algorithm

In the first stage of the algorithm, where only MMSS's are found by reaching all nodes once, the time complexity is O(n). Next, all MMSSs are sorted using quicksort algorithm which has an average time complexity of O (n log n), where n is the number of MMSSs found. In the next stage, we find smaller anchors in the inter MMSS region. This stage is computationally expensive, as overlapping and criss-crossing anchors are first identified, removed if necessary and sorted using quick sort algorithm. We find all linearly increasing anchors in both sequences, very similar to longest increasing subsequence problem (LIS), which has a time complexity of O (n log n) [15]. We then move to the last stage of extension, the time complexity of this stage is O (n), where n is the number of MMSSs. So the time complexity of our algorithm is O (n log n).

3.3.8 Hypothesis

MASAA would not only be faster for long sequences but would also be as sensitive as BLASTZ on sequences which have varying homology similarity.

Chapter 4

Experimental Model, Results & Analysis

We present the performance of MASAA and compare the results with BLASTZ which is the most recent version of the most popular and widely used pairwise sequence alignment algorithm, BLAST. This chapter describes the experimental setup, assumptions, results and analysis.

4.1 Experimental setup

4.1.1 Data sets

We compare MASAA and BLASTZ with four different data sets. The first data set is called 'ROSETTA' data set which consists of a set of 117 sequences of human and mouse genes. The second data set is a subset of 'Homophila' data set[12]. The 'Homophila' data set is a database consisting of more than 700 human disease causing genes and corresponding fruit fly genes cognates called 'Homophila'. The term 'cognate' implies that there is a functional similarity between genes, but not necessarily the homology similarity. 'Homophila' data set does not show the percentage of homology similarity or conserved region present between the human and fruit fly gene but presents only the gene name. In order to find the percentage of homology, we first aligned 400 sequences taken randomly from 'Homophila' data set using a global sequence algorithm, 'LAGAN'. For experimental purposes, 67 genes were selected from this data set as the remaining sequences had the homology similar to ROSETTA dataset. Genes which had a homology similarity close to the first data set were excluded. Genes in the second data set have conserved region ranging from 0 to 70 percent. Second data set sequences with different percentage of conserved region is shown in table A.1 of the Appendix. Genes in the first data set have conserved region ranging from 85 to 96 percent.

Due to the size of the SECOND data set, we show only a few human and fruit fly gene alignment comparison in this chapter. The third data set, consists of twenty gene sequences from human, mouse, pig and horse, whose length range from 120,000 to 800,000 bp's sourced from NCBI website[25]. Twenty sequences were sourced mainly to match the range of the randomly generated sequences. Finally, we have the fourth data set consisting of randomly generated sequences ranging from 100,000 bp's to nearly half a million bps in length. Randomly generated sequences were created because of the difficulty of finding uniformly increasing real sequences. Combining all data sets, we have a total of 350 sequences. For reasons of simplicity first, second, third and fourth data set are referred as ROSETTA, SECOND, REAL and RANDOM data set, respectively, in this chapter.

4.1.2 Performance Metrics

For our experiments, we consider four performance metrics, they are:

- Time (in seconds), total time taken to the alignment
- Exon coverage, the percentage of exon covered in the final alignment
- Alignment score, the total alignment score of the alignment
- Bp coverage, Is the total number of bps aligned in the final alignment

We use different performance metrics for different data sets. Exon coverage is used for ROSETTA data set for two reasons: (1)This commonly used metric is used in the literature as it determines the ability of the algorithm to detect and align conserved regions [7] and (2) ROSETTA data set is also the only data set which contains gene sequences with exon annotation. Alignment score and bp coverage are used for SECOND data set. This is because, exon is absent in either human or fruit fly or both of the human-fruit fly sequence pair for some genes making exon coverage irrelevant. For REAL and RANDOM data set, time and bp coverage are used. The alignment score is not used for these data sets because alignment score is used to show the degree of evolutionary closeness and is not a good fit for randomly generated sequences. The main objective of the experiments on RANDOM data set is to observe time taken to complete the alignment using MASAA and BLASTZ. RANDOM dataset contains sequences whose length are increasing linearly from 100000 to half a million with an interval of 2000 bps. Since randomly generated sequences do not clearly show the practical effectiveness of the algorithm, we later used REAL data set whose sequence length are in the same range as that of RANDOM data set.

4.1.3 Assumptions

For experimental purposes, we have made several assumptions as described below:

- The MMSS anchors selected are 'good' when aligning two real sequences, S_1 and S_2. We call an anchor *good* if either the beginning position or the ending position associates two related positions of the alignments. The two related positions could be starting and ending position of exon region, untranslated region, complete gene or others. All anchors that are not good are called *bad* anchors. *Bad* anchors can result in serious errors or drop in MASAA sensitivity.

- There is at least one MMSS between the sequences, S_1 and S_2 under study. The longest MMSS found is a part of the final alignment and is always a *good* anchor.

- We are also interested in the overall accuracy of the alignment algorithm. Alignments are scored by sum of pairs described earlier and accuracies given by the number of columns that are correct (bp coverage). We consider a column to be correct if pair from both sequence at a given position i, j in S_1 and S_2 match in a region. This region could be again, exon region, untranslated region, whole gene, whole sequence in some cases and others. For RANDOM and REAL data set we consider the whole sequence. This condition might look too strict when aligning short real sequences, as there are often short substrings and MASAA might miss these substrings. Since we use long real and random sequences, we feel this sensitive criterion is reasonable.

- We believe that mutations (changes in the child sequences from the parent sequence) in the sequences would have occurred at different places in the sequence as a long stretch rather than short stretches or single mutations. Block insertion and deletion are assumed to be true from the literature.

4.1.4 Considerations

- When aligning sequences S_1 and S_2, both the sequences are of similar length. The reason is, if the difference between two sequence length is large, then the chances of finding many MMSSs are low. For example, if S_1 and S_2, are 10,000 and 1,000 bp in length respectively, there could be only one MMSS of length 400 and other MMSS's might be very small to be considered. In this condition, the algorithm would end up with only one large MMSS. In order to minimize this condition, sequences which are more or less of the same length are considered.
- In the sequence, there could be substrings which are repeatedly found at different places along the length of the sequence. These substrings are called, 'repeat strings'. In the literature, we found many algorithms which removed these substrings to enhance the speed of their respective algorithm. In our case, for both BLASTZ and MASAA, we have not removed or masked any repeat substrings. Thus, the reported time reflects the actual time taken for the alignment.

4.2 Analysis of Results

Experimental results are collected for four performance metrics described in 4.1.2. We investigate these four performance metrics by: (1) varying the size of MMSS, (2) varying the size of the inter MMSS anchor, and (3) varying the minimum distance of the MMSS to be considered for the final alignment. Results obtained by varying these parameters are averaged over 10 simulation runs. The gene number in the horizontal axis in the experimental graphs correspond to the gene number in the data set shown in appendix A. For baseline configuration, the default parameters chosen are: (1) The length of MMSS considered in the first stage of the algorithm is fixed at 50 percent of the longest MMSS found, (2) The length of the inter MMSS anchors considered is 12 with 4 mismatches and (3) The minimum distance δ, between MMSS in the final stage is kept at 10000 bp.

4.2.1 Baseline configuration

4.2.1.1 RANDOM data set

Experiment 1 (Total alignment time with baseline configuration): In this experiment, we observe the time taken to align sequences from RANDOM data set by MASAA and BLASTZ. The results are summarized in Figure 4.1.

Observation 1: We found that MASAA consistently outperformed BLASTZ for long sequences because while BLASTZ spends more time finding the high scoring segment pairs, MASAA quickly finds MMSSs and inter MMSS anchor seeds using the suffix tree. MASAA spends most of its time in finding inter MMSS anchors, and criss-crossing and overlapping seeds in the inter MMSS region. BLASTZ spends additional time in identifying and expanding high scoring segment pairs. From the experimental results, it is clear that MASAA's overall time is lower than BLASTZ.

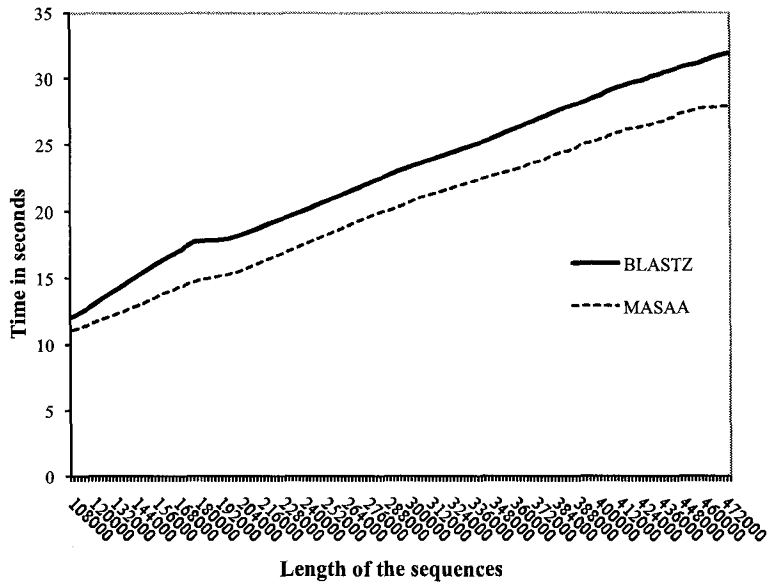


Figure 4.1: ***Total alignment time on RANDOM data set with baseline configuration***

Experiment 2 (Bp coverage with baseline configuration): This experiment shows the trend in the bp coverage using sequences from RANDOM data set. The bp coverage for MASAA and BLASTZ is the cumulative bp's of all the HSPs found in the alignment. The results are shown in Figure 4.2.

Observation 2: Figure 4.2 shows that, as the length of the sequence increases, MASAA has better bp coverage, which can be attributed to many MMSS's and inter anchors identified by MASAA. BLASTZ bp coverage is at 100 percent as we are comparing BLASTZ bp coverage with MASAA. MASAA also considers overlapping and criss-crossing anchors in the region between the inter MMSS anchors, thus increasing the overall bp coverage.

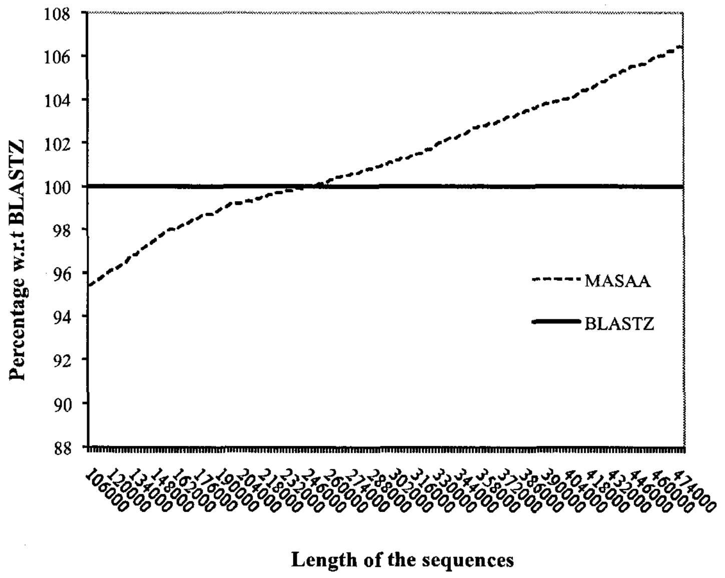


Figure 4.2: ***Bp coverage on RANDOM data set with baseline configuration***

4.2.1.2 REAL data set

Experiment 3 (Total alignment time with baseline configuration): In this experiment, we observe the time taken to align long sequences by MASAA and BLASTZ. The results are summarized in Figure 4.3.

Observation 3: For real sequences, we observe the same trend as randomly generated sequences in experiment one. Figure 4.3 shows the time gap between MASAA and BLASTZ on real sequences gets higher as the length of the sequence increases. This is because MASAA spends little time in selecting the MMSS anchors thus establishing itself the range within which the final local alignment could be produced. However, BLASTZ spends considerable amount of time in detecting and expanding high scoring segment pairs. Also, real sequences with high homology similarity con-

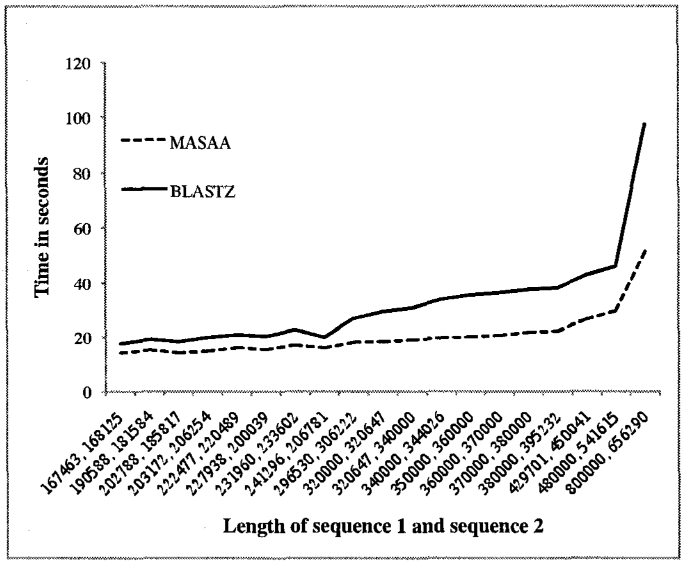


Figure 4.3: ***Total alignment time on REAL data set with baseline configuration***

tributes to MASAA's poor performance as it is more likely that BLASTZ detects many high scoring pairs which do not contribute to the final alignment.

Experiment 4 (Bp coverage with baseline configuration) This experiment shows the bp coverage using sequences from REAL data set. The results are shown in Figure 4.4.

Observation 4: Figure 4.4 shows that the MASAA bp coverage is much better than BLASTZ when the length of the sequences increases. We attribute this behavior to two factors: (1) in case of real sequences, the chances of finding MMSS and inter MMSS anchors are more because the sequences have greater homology similarity, (2) as a result, in the last phase of our algorithm, chances of more MMSS being considered for final alignment increases. In case of small sequences, the MMSS and inter MMSS

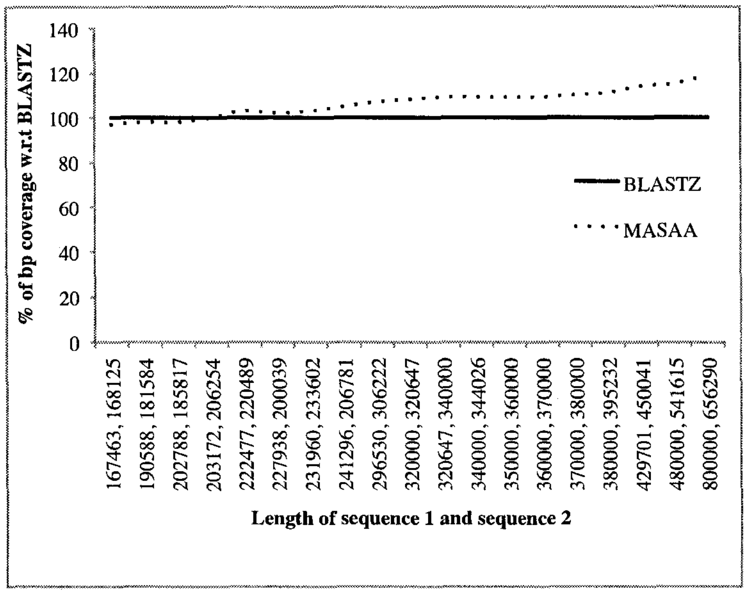


Figure 4.4: ***Bp coverage on REAL data set with baseline configuration***

region are also small. In the small inter MMSS region, MASAA removes unnecessary overlapping and criss-crossing anchors reducing the overall inter MMSS anchor set size. Hence, bp coverage is low for smaller sequences.

4.2.1.3 ROSETTA data set

Experiment 5 (Exon coverage with baseline configuration): To test the sensitivity of our algorithm, we used ROSETTA data set[6]. The results are shown in Table4.1. The table shows the percentage of ROSETTA sequences which covered different percentages of exon region.

Observation 5: Table4.1 shows that BLASTZ is good for aligning naturally evolving sequences because BLASTZ was designed for homologous sequences. BLASTZ

Table 4.1: ***Exon coverage on ROSETTA data set with baseline configuration***

Aligner	100 exon	90 exon	70 exon
BLASTZ	94	97	98
MASAA	94	94	96

picks substrings effectively along the length of the sequences easily using the spaced seed design explained earlier. There are two reasons why BLASTZ is not able to outperform in case of homologous sequences; (1) 'wrong' seeds are considered in the inter MMSS region because many seeds whose position in both sequences are more or less the same, are eliminated as they are criss-crossing or overlapping with longer seeds, see Figure 4.5, and (2) the first and the last MMSS's are not extended to left and right as BLASTZ does on high scoring pairs.

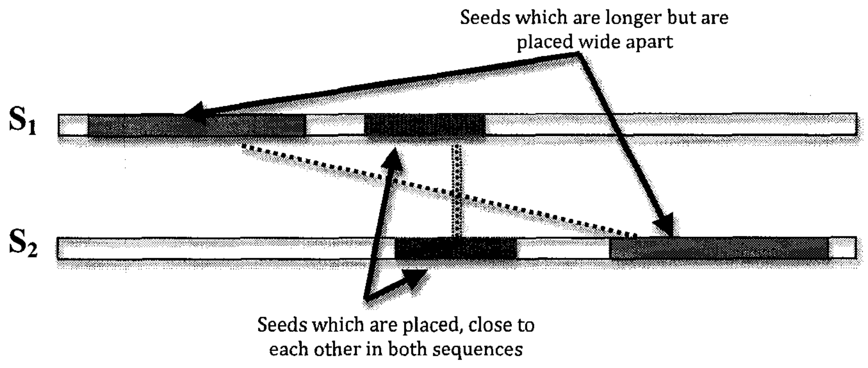

Seeds which are longer but are placed wide apart
S_1
S_2
Seeds which are placed, close to each other in both sequences

Figure 4.5: ***Longer seeds are given preference over smaller seeds***

4.2.1.4 SECOND data set

Since SECOND data set contains sequence set which have smaller percentage of conserved region, exon coverage as a performance metric is not relevant. Thus, alignment score and maximum bp coverage are used as the performance metric.

Experiment 6 (Alignment score with baseline configuration): In this experiment, we observe the alignment score for BLASTZ and MASAA. To observe the alignment score, we used the scoring matrix used in BLASTZ. The experimental observation is shown in Figure 4.6 and Table 4.2.

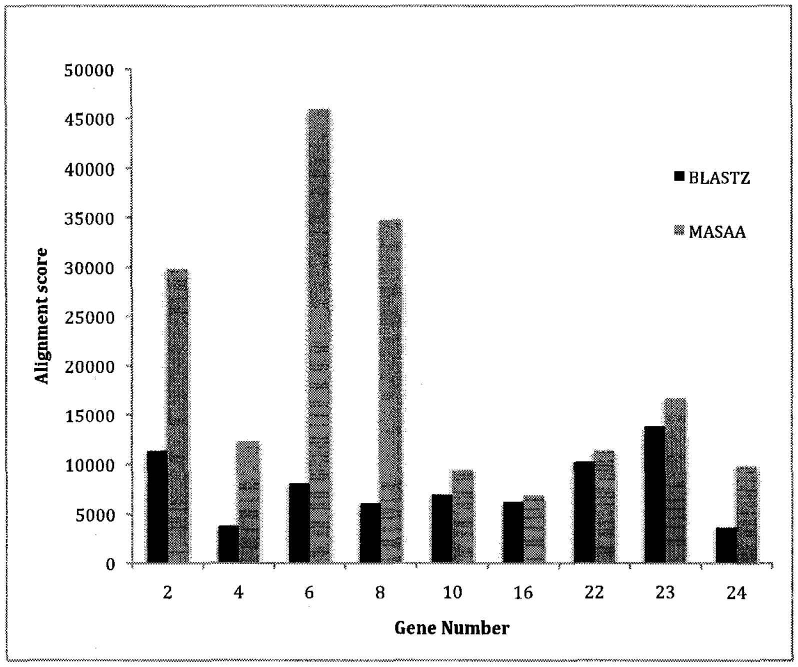


Figure 4.6: ***Alignment score on SECOND data set with baseline configuration***

Table 4.2: *Number of genes MASAA's alignment score better than BLASTZ*

Aligner	No of genes
BLASTZ	8
MASAA	59

Observation 6: Figure 4.6 shows MASAA performing better than BLASTZ for majority of the genes (10/13) in its default parameter configuration. The observation also infers that suffix based maximal match substring design as the base for our algorithm suits well than a look-up table based seed design algorithm for aligning sequences which have a homology similarity in the range of 0 to 70 percent. Table 4.2 shows the number of genes each aligner is outperforming the other, MASAA clearly is outperforming BLASTZ for the reasons outlined earlier.

Experiment 7 (Bp coverage with baseline configuration) In this experiment, we are interested in observing the bp coverage on SECOND data set. The observations are shown in Figure 4.7 and Table 4.3. An example of MASAA alignment for a gene ACE is shown in Figure 4.8.

Observation 7: Figure 4.7 and Table 4.3 shows MASAA performing better than BLASTZ. We attribute this bp coverage to MMSS and, more importantly, the anchors found in between the MMSSs. MASAA considers both overlapping and criss-crossing anchors seeds in the regions between MMSS's. As a result, many inter MMSS anchors are still considered after unnecessary anchors are eliminated, which in turn contributes

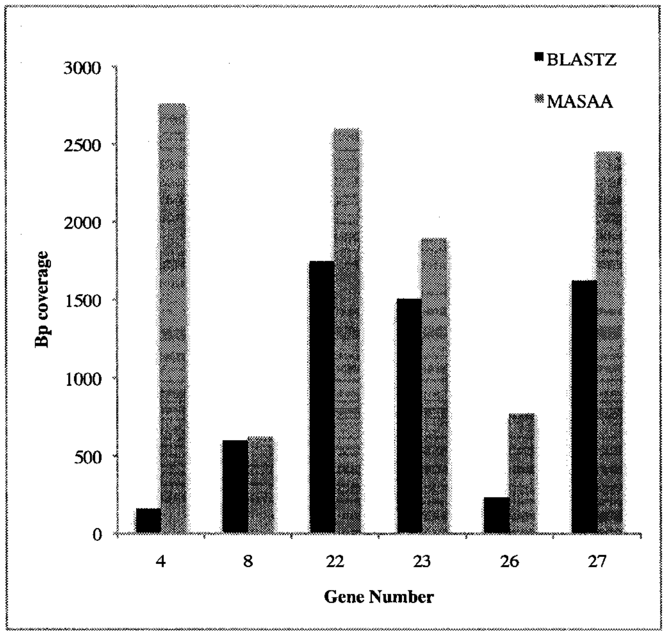
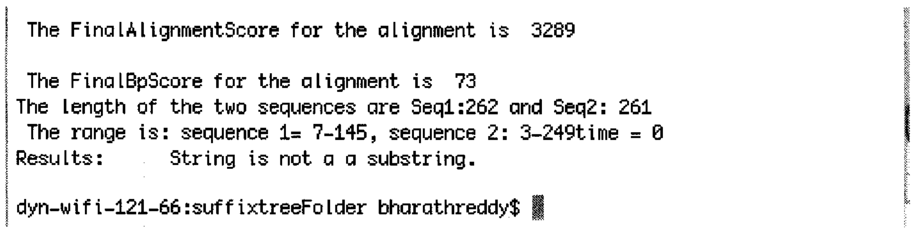


Figure 4.7: *Bp coverage on SECOND data set with baseline configuration*

Table 4.3: *Number of genes MASAA's bp coverage better than BLASTZ*

Aligner	No of genes
BLASTZ	8
MASAA	59



```
 The FinalAlignmentScore for the alignment is  3289

 The FinalBpScore for the alignment is  73
The length of the two sequences are Seq1:262 and Seq2: 261
 The range is: sequence 1= 7-145, sequence 2: 3-249time = 0
Results:       String is not a a substring.

dyn-wifi-121-66:suffixtreeFolder bharathreddy$ 
```

Figure 4.8: *An example of MASAA alignment for gene ACE*

to the overall bp coverage.

Conclusion: In this section, we compared BLASTZ and MASAA in its baseline configuration. From the experiments, we conclude that as the length of the sequences increases, MASAA is faster and sensitive than BLASTZ . For ROSETTA set, MASAA is comparable to BLASTZ. For the SECOND data set, MASAA outperformed BLASTZ.

4.2.2 Varying the MMSS Length Parameter

One of the most important parameter in our algorithm is the length of the MMSS. The length of the MMSS is critical because, if the length of MMSS considered is of a smaller percentage than the longest found in the suffix tree, there would be many MMSSs. This results in many inter MMSS anchors and would in turn be computational expensive. In the following experiments, we vary the MMSS length and observe variations in terms of sensitivity and speed.

4.2.2.1 RANDOM data set

Experiment 8 (Total alignment time by varying MMSS length): In this experiment, we vary MMSS anchor length from 35% to 60% of the longest MMSS found in the tree for randomly generated sequences. The observations are shown in Figure 4.9.

Observation 8: Figure 4.9 shows MASAA's performance increases as the length of the MMSS increases from 50% to 60% of the longest MMSS in the tree, however, the

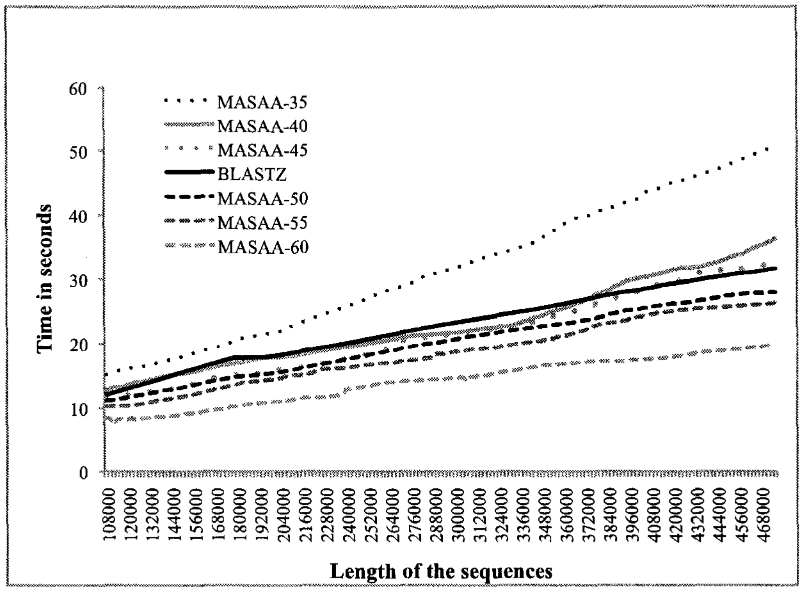


Figure 4.9: ***Total alignment time on RANDOM data set by varying MMSS length***

speed decreases as the percentage is reduced from 50% to 35% of the longest MMSS in the tree. The reason for this trend is that, when the MMSS is shorter than the longest found in the suffix tree, many MMSS are found. There are many inter MMSS anchors regions now, and more seeds are found in the inter anchor region, this directly increases the computational time of the algorithm.

Experiment 9 (Bp coverage comparison by varying MMSS length): In this experiment, we want to observe bp coverage when the MMSS length is varied. The experimental observation is shown in Figure 4.10.

Observation 9: Figure 4.10 shows that when MMSS length is lower than 60, the bp coverage exceeds that of BLASTZ. When the MMSS length is 60% of the longest

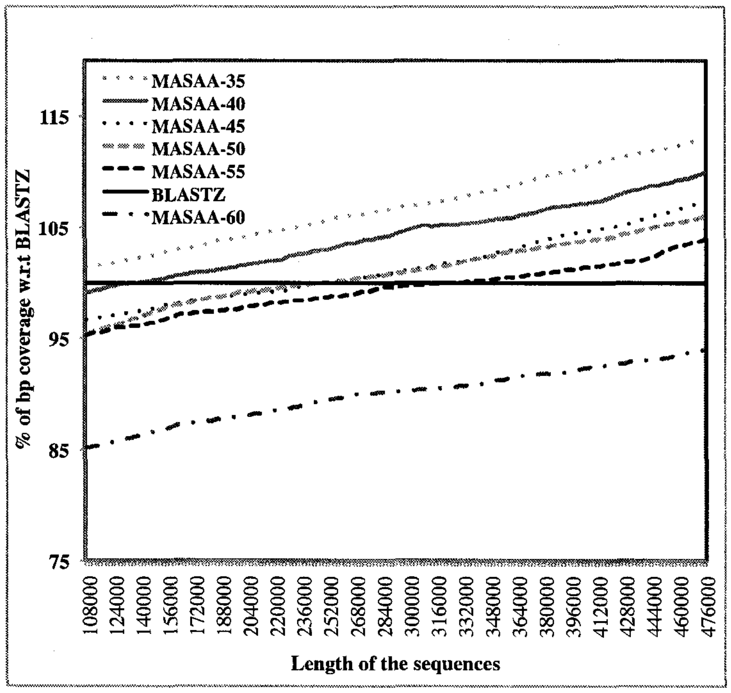


Figure 4.10: ***Bp coverage on RANDOM data set by varying MMSS length***

MMSS found in the tree, MASAA fails to have better bp coverage than BLASTZ. The reason is that, there are not many MMSS's found. When MMSS length is lower than 60% of the longest MMSS found, MASAA bp coverage exceeds BLASTZ bp coverage for longer sequences.

4.2.2.2 REAL data set

Experiment 10 (Total alignment time by varying MMSS length): In this experiment, we vary MMSS length from 35% to 60% of the longest MMSS found in the tree for real sequences. The observations are shown in Figure 4.11.

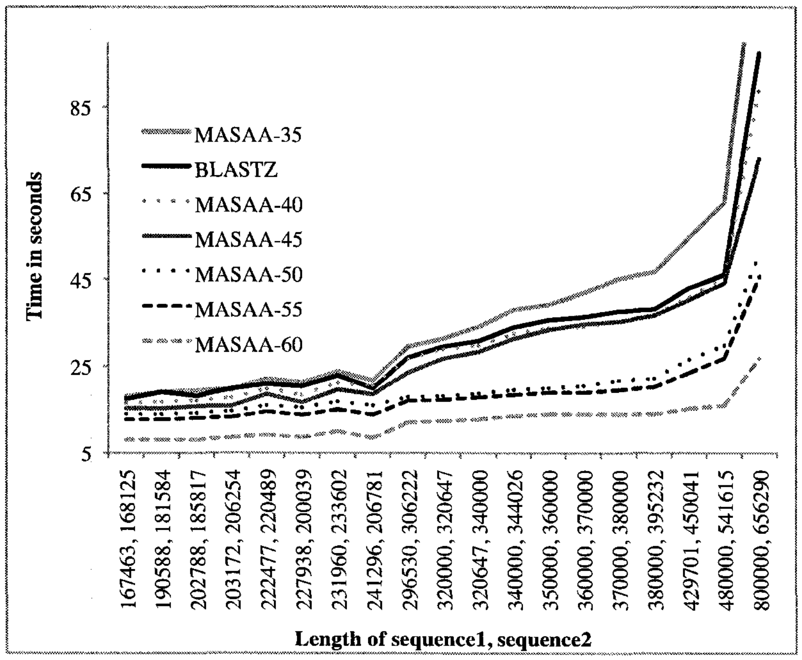


Figure 4.11: ***Total alignment time on REAL data set by varying MMSS length***

Observation 10: Figure 4.11 shows that when the MMSS length is shorter than 45% of the longest found in the tree, the time taken by MASAA is larger than BLASTZ. The numbers in the horizontal axis of the graph are the actual length of two sequences. For percentages greater than 45, the time taken by MASAA is less than BLASTZ. When MMSS length is 35 or 40 most of the MMSS's in the tree will be included. Hence, the time taken is larger than when the MMSS length is set higher.

Experiment 11 (Bp coverage by varying MMSS length): In this experiment, we want to observe the bp coverage when the MMSS length is varied for real sequences. The experimental observations is shown in Figure 4.12.

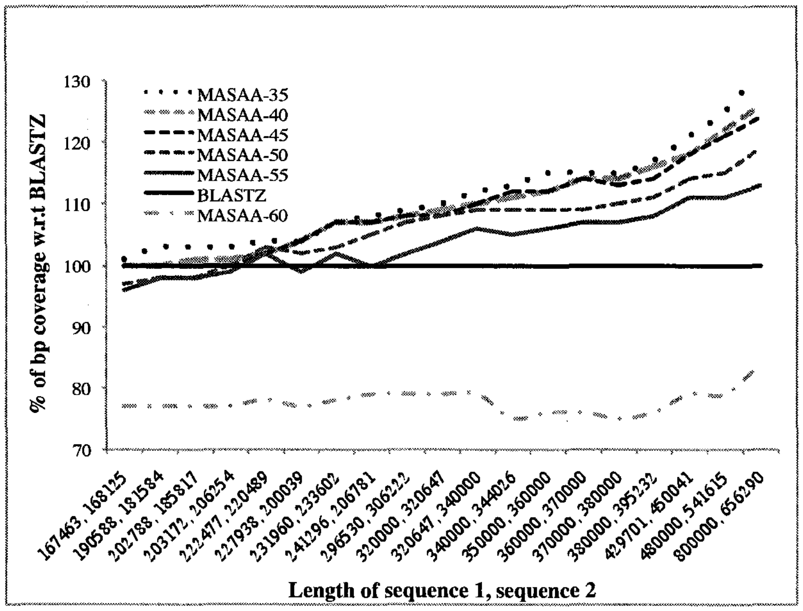


Figure 4.12: *Bp coverage on REAL data set by varying MMSS length*

Observation 11: Figure 4.12 shows that MASAA is able to perform better than BLASTZ when MMSS length is varied from 35% to 55% of the longest MMSS found in the tree. When the MMSS length is 60%, MASAA fails to have a better coverage as there are fewer MMSS found. Similarly, when the MMSS length is of a smaller percentage to that of the longest MMSS found in the tree, MASAA is able to perform better than BLASTZ in terms of bp coverage becuase they are many MMSS found.

4.2.2.3 ROSETTA data set

Experiment 12 (Exon coverage by varying MMSS length from 35 to 60% of the longest MMSS): In this experiment, we vary the MMSS length from 35 percent to 60 percent and observe variations in terms of sensitivity and speed. The experimental observations are shown in Table 4.4.

Observation 12: From the Table 4.4, it is clear that the performance gradually increases as the length of the MMSS is reduced. We also observe that lowering the length of the MMSS to 35% of the longest MMSS is acceptable as many MMSS anchors are now considered for final alignment. We conclude that reducing the size of MMSS anchor does have a direct impact on the sensitivity of the algorithm.

4.2.2.4 SECOND data set

In this section, we vary the length of the MMSS and observe the sensitivity on SECOND data set.

Experiment 13 (Alignment Score by varying MMSS length): In this experiment, we vary the length of the MMSS from 35 to 60 percent of the longest MMSS found and

Table 4.4: ***Exon coverage on ROSETTA data set by varying MMSS length***

Aligner	100 exon	90 exon	70 exon
BLASTZ	94	97	98
MASAA-35%	95	97	94
MASAA-40%	94	95	96
MASAA-45%	94	95	96
MASAA-50%	94	94	96
MASAA-55%	94	94	96
MASAA-60%	92	97	94

observe the alignment score on SECOND data set. The experimental observations is shown in the Figure 4.13 and Table 4.5.

Observation 13: Figure 4.13 and Table 4.5 shows that as the length of the MMSS decreases, the alignment score varies but the score is still higher than BLASTZ. We also observe that for few genes, as we increase the MMSS length threshold to 60%, the alignment score drops. The exact reason for this drop is difficult to predict. There could be many reasons for this low alignment score, some of them could be, smaller MMSSs, smaller inter MMSS anchors, MMSS's distributed far away from each other, and many overlapping and criss-crossing anchors eliminated in the inter MMSS regions.

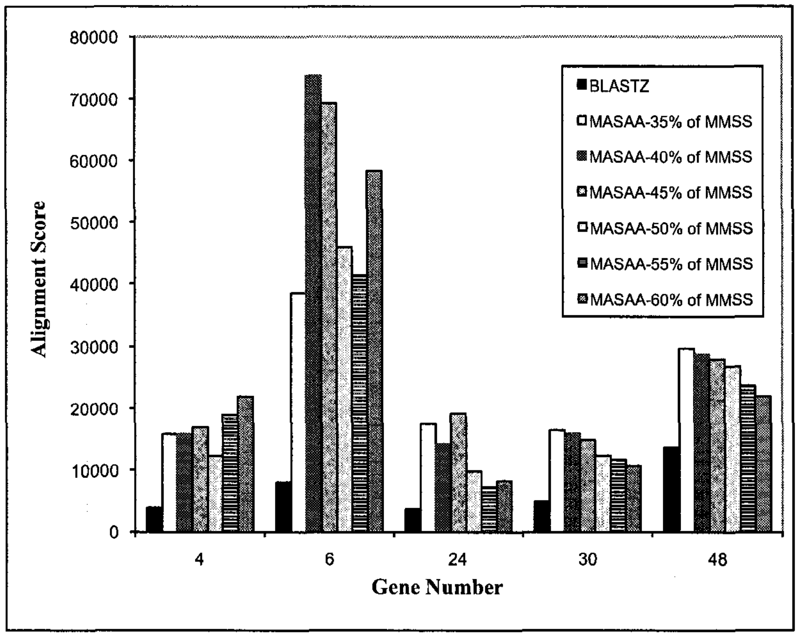


Figure 4.13: *Alignment Score on SECOND data set by varying MMSS length*

Table 4.5: *Number of genes MASAA's alignment score better than BLASTZ*

Aligner	No of genes
MASAA-%35 of MMSS	65
MASAA-%40 of MMSS	64
MASAA-%45 of MMSS	61
MASAA-%50 of MMSS	59
MASAA-%55 of MMSS	55
MASAA-%60 of MMSS	44

Experiment 14 (Bp coverage by varying MMSS length): In this experiment, we vary the length of the MMSS from 35 to 60 percent of the longest MMSS found in the tree and observe the bp coverage. The experimental observation is shown in Figure 4.14 and Table 4.6.

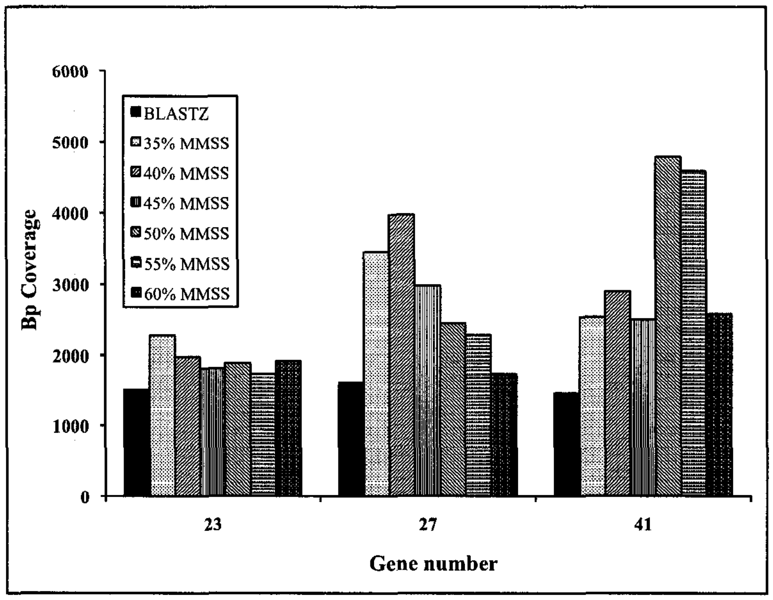


Figure 4.14: ***Bp comparison on SECOND data set by varying MMSS length***

Observation 14: Figure 4.14 and Table 4.6 shows that, as the length of the MMSS is varied, the bp coverage is unpredictable. This is due to two reasons: (1) The number of MMSSs found varies when the length of MMSS parameter is changed (2) The number of inter MMSS anchors also varies when the MMSS parameter is varied. If the MMSS and inter MMSS anchors are more, we see better bp coverage.

Table 4.6: *Number of genes MASAA's bp coverage better than BLASTZ*

Aligner	No of genes
MASAA-%35 of MMSS	66
MASAA-%40 of MMSS	63
MASAA-%45 of MMSS	61
MASAA-%50 of MMSS	59
MASAA-%55 of MMSS	56
MASAA-%60 of MMSS	42

4.2.3 Varying the inter MMSS anchor Length Parameter

The inter anchor region plays an important role in the sensitivity of our algorithm. The length of the inter MMSS anchor is critical because it can affect the speed and sensitivity of MASAA. In the following experiments, we vary the inter MMSS anchor length and observe any variations in terms of sensitivity and speed. First we look at RANDOM data set.

Conclusion: In this section, we compared BLASTZ and MASAA by varying the length of MMSS. From the experiments, we conclude that as the length of the MMSS increases, MASAA is faster and less sensitive than BLASTZ . If the length of MMSS is lowered then MASAA is slower but it sensitive than BLASTZ. We conclude by saying that, the length of the MMSS considered in the initial stage of the algorithm is vital in determining the speed and sensitivity of the algorithm.

4.2.3.1 RANDOM data set

Experiment 15 (Total alignment time by varying inter MMSS anchor length): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp for randomly generated sequences. The observations are shown pictorially in Figure 4.15.

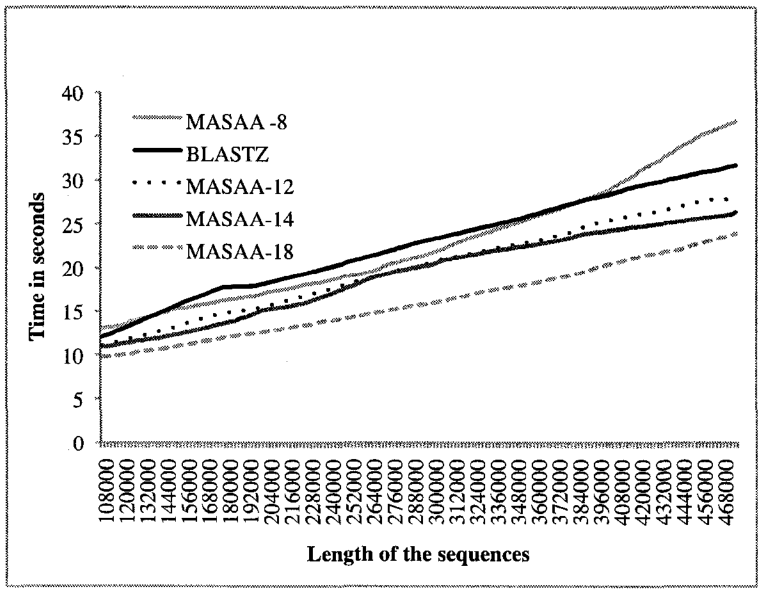


Figure 4.15: ***Total alignment time by varying the inter anchor size***

Observation 15: Figure 4.15 shows that, when the inter MMSS length is 8 bp, the MASAA performs poorly than BLASTZ. When the MMSS length is 12, 14 and 18 bp in length, MASAA performs better than BLASTZ. When the inter anchor size is low, the number of anchors selected are many as the length of the sequence increases. As a result, MASAA would be slower than BLASTZ.

Experiment 16 (Bp coverage by varying inter MMSS anchor length): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp for real sequences and observe the percentage of bp coverage. The experimental observations are shown in Figure 4.16.

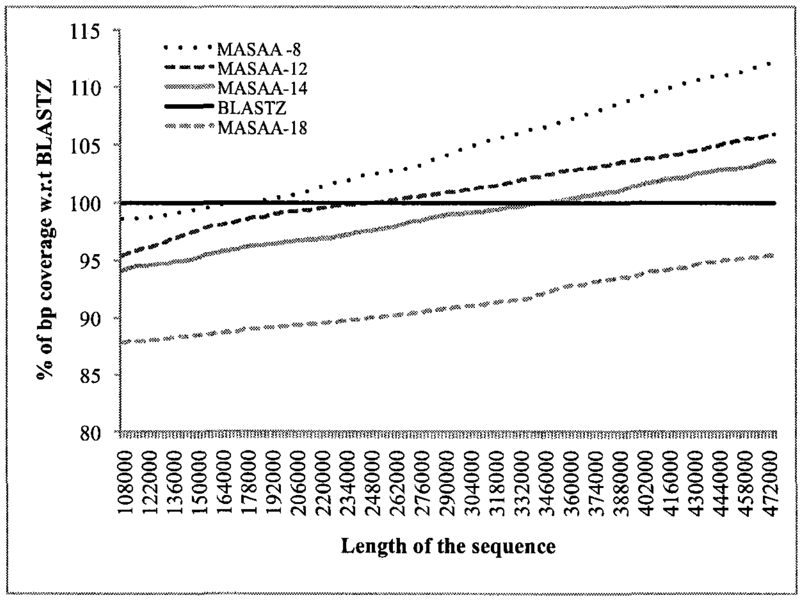


Figure 4.16: ***Bp coverage by varying the inter anchor length***

Observation 16: Figure 4.16 shows that when the inter MMSS anchor length is 12 and 14, the bp coverage is very close to each other. However, when the inter MMSS length is 8, the bp coverage is better because there are many seeds found in the inter MMSS anchor. Similarly when the inter MMSS anchor is 18, there is more bp coverage than BLASTZ even when the sequence length is increased to approximately half a million. The reason is that the number of seeds found in the inter MMSS region are fewer with the inter anchor length of 18.

4.2.3.2 REAL data set

Experiment 17 (Total alignment time by varying inter MMSS anchor): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp for real sequences. The experimental observations are shown in Figure 4.17.

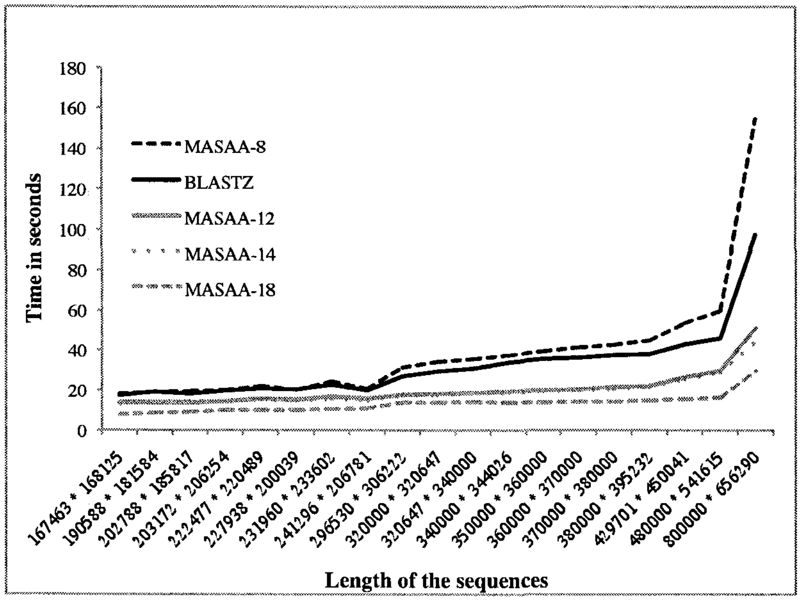


Figure 4.17: ***Total alignment time by varying the inter anchor length***

Observation 17: Figure 4.17 shows that when the length of the inter MMSS anchor is 12 and 14, there is no significant difference in the time taken by the two algorithms. When the inter MMSS anchor is 8 and 18, we see a significant difference in the time taken by MASAA. When the inter MMSS anchor is 8 bp, the number of inter MMSS anchors found are many and MASAA spends more time in identifying, selecting anchors from overlapping and criss-crossing anchors. Hence MASAA is slower than BLASTZ for inter MMSS anchor.

Experiment 18 (Bp coverage by varying inter MMSS anchor): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp for real sequences and observe the percentage of bp coverage. The experimental observations are shown in Figure 4.18.

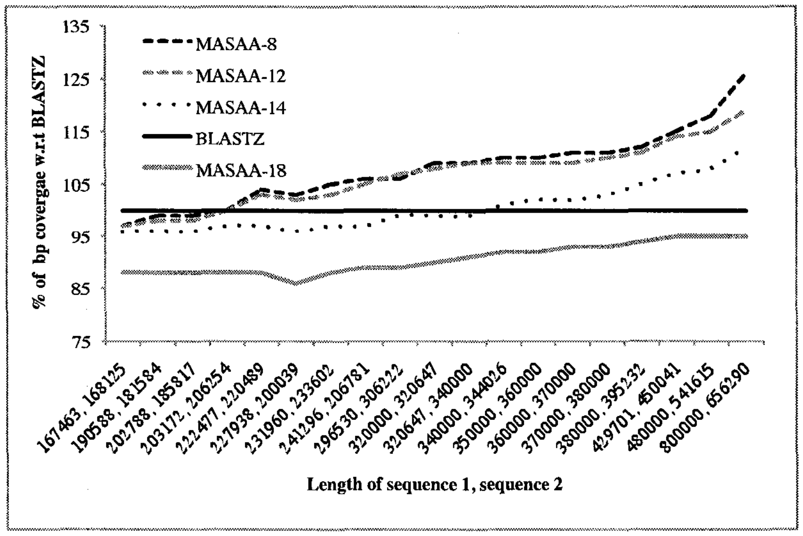


Figure 4.18: ***Bp coverage by varying the inter anchor length***

Observation 18: Figure 4.18 shows that when the inter MMSS anchor length is 8 and 12, the bp coverage is close to each other until the length of the sequence is less than a million. When the inter MMSS anchor is 18, the bp coverage remains inferior to BLASTZ due to smaller number of anchors being picked at this phase of MASAA. We conclude that inter MMSS anchor length should be less than 18.

4.2.3.3 ROSETTA data set

Experiment 19 (Exon coverage by varying inter MMSS anchor length): In this experiment, we consider ROSETTA data set and observe the percentage of exon coverage in the final alignment. The experimental observations are shown in the Table 4.7.

Table 4.7: ***Exon coverage on ROSETTA data set when inter anchor length is 8, 12, 14 and 18***

Aligner	100 exon	90 exon	70 exon
BLASTZ	94	97	98
MASAA-8	95	97	98
MASAA-12	94	94	96
MASAA-14	94	94	96
MASAA-18	91	87	97

Observation 19: We observe that for lower inter MMSS anchors, sensitivity does increase. We also observe that larger the inter anchor region, lower the exon coverage. This is due to variation in the number of the inter MMSS anchors picked by MASAA.

4.2.3.4 SECOND data set

In the following experiments, we vary the inter MMSS anchor length and observe variations in terms of sensitivity and speed on SECOND data set.

Experiment 20 (Alignment score by varying inter MMSS anchor length): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp in length and observe alignment score of the final alignment. The experimental results are shown in the Figure 4.19 and Table 4.8.

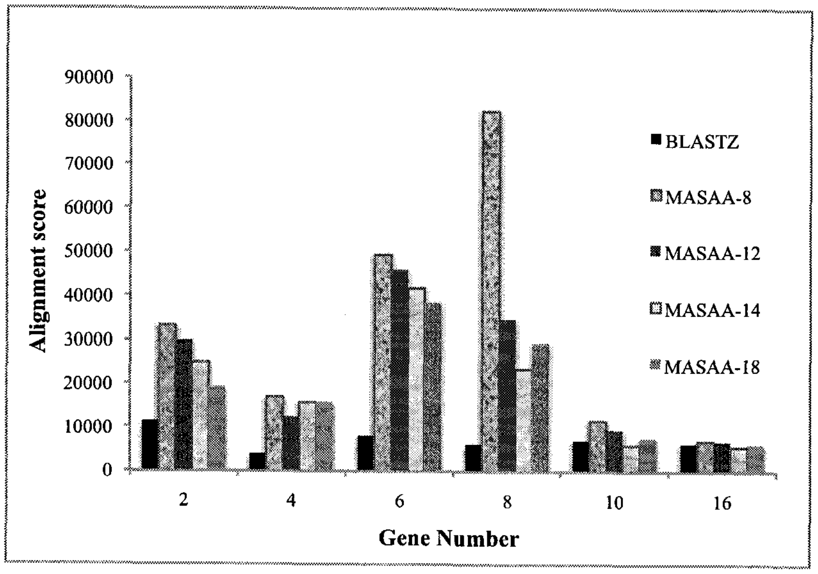


Figure 4.19: ***Alignment score on SECOND data set by varying inter MMSS anchor length***

Observation 20: Figure 4.19 and Table 4.8 shows that, as the length of the inter MMSS anchor length is reduced, sensitivity increases. We also observe that, as the

Table 4.8: ***Number of genes MASAA's alignment score better than BLASTZ by varying inter MMSS anchor length***

Aligner	No of genes
MASAA-%8 bp	64
MASAA-%12 bp	59
MASAA-%14 bp	50
MASAA-%18 bp	45

inter MMSS anchor decreases, the alignment score increases and as the inter MMSS anchor length increases, the alignment score decreases. This is due to two reasons; (1) when the inter MMSS anchor length increases, the sensitivity decreases as many overlapping, criss-crossing seeds found close to each other are canceled out by the algorithm described in chapter three or there are now fewer anchors and (2) when the inter anchor length decreases, the sensitivity increases, as there are many inter MMSS anchors which can cover the entire range between the MMSS's.

Experiment 21 (Bp coverage by varying inter MMSS anchor length): In this experiment, we vary inter MMSS anchor length from 8 to 18 bp in length and observe bp coverage of the final alignment. The experimental results are shown in the Figure 4.20 and Table 4.9.

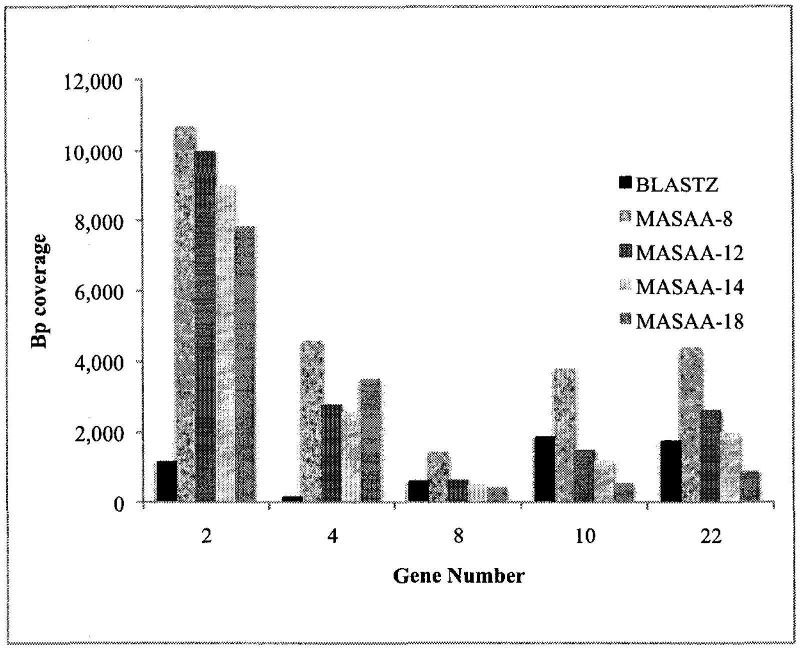


Figure 4.20: ***Bp coverage on SECOND data set by varying inter MMSS anchor length***

Table 4.9: ***Number of genes MASAA's bp coverage better than BLASTZ by varying inter MMSS anchor length***

Aligner	No of genes
MASAA-%8 bp	63
MASAA-%12 bp	59
MASAA-%14 bp	52
MASAA-%18 bp	45

Observation 21: Figure 4.20 and Table 4.9 shows the same phenomenon as in alignment score for the bp coverage. We observe as the length of the inter MMSS anchor decreases the bp coverage increases. The Table 4.9 clearly shows theat MASAA outperforming BLASTZ. This indicates that smaller seeds can align more bps than larger anchors.

Conclusion: In this section, we compared BLASTZ and MASAA by varying the inter MMSS anchor length. From the experiments, we conclude that as the inter MMSS anchor increases, MASAA is faster and less sensitive than BLASTZ . If the length of inter MMSS anchor is lowered then MASAA is slower but sensitive than BLASTZ. We conclude by saying that, the length of the inter MMSS anchor considered in the algorithm is vital in determining the speed and sensitivity of the algorithm.

4.2.4 Varying the Minimum Distance to Extend Anchors

The minimum distance, 'd', to extend anchors, also plays an important role in the sensitivity of our algorithm. We compare BLASTZ with MASAA by varying the minimum distance, 'd', in the following experiments.

4.2.4.1 RANDOM data set

Experiment 22 (Total alignment time by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we observe any variations in speed on ROSETTA data set. The observations are shown in Figure 4.21.

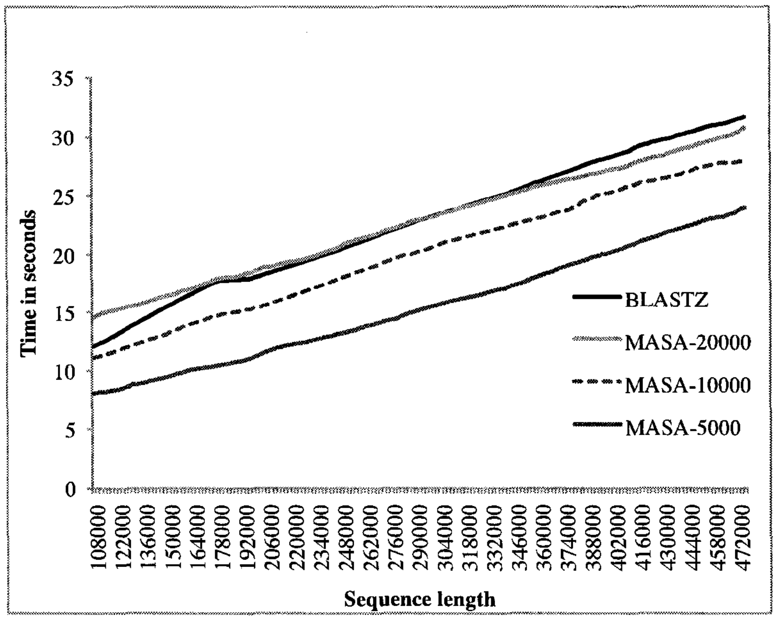


Figure 4.21: ***Total alignment time on RANDOM data set by varying the minimum distance, 'd' between MMSS***

Observation 22: Figure 4.21 shows that when minimum distance between MMSS, 'd', is increased, the algorithm is slower than what it is when 'd' is decreased. The reason is when the distance 'd' is large, majority of the MMSS anchors are extended, as a result, time is spent on the inter MMSS anchors and the anchors in between these inter MMSS anchors.

Experiment 23 (Bp coverage by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we vary minimum distance, 'd', to extend MMSS anchors for randomly generated sequences and observe the percentage of bp coverage with respect to BLASTZ. The observations are shown in Figure 4.22.

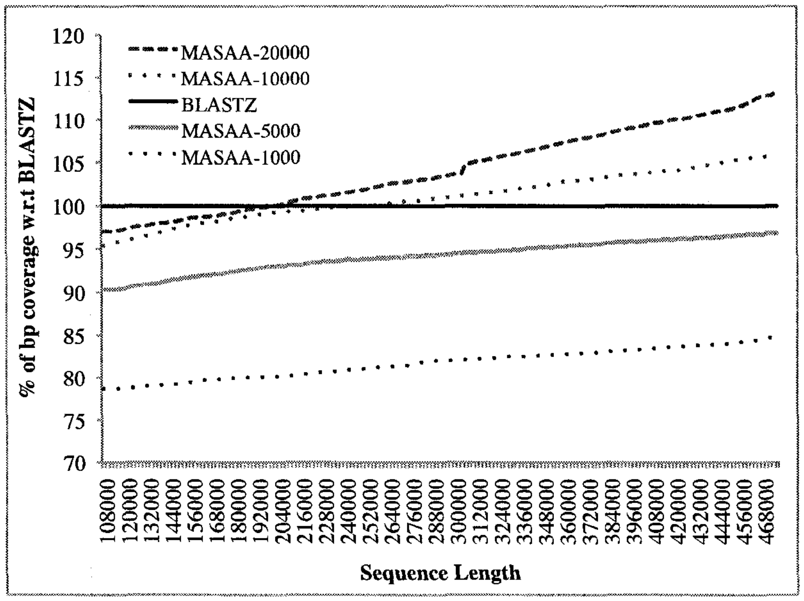


Figure 4.22: ***Bp coverage on RANDOM data set by varying the minimum distance, 'd' between MMSS***

Observation 23: Figure 4.22 shows that when the minimum distance between MMSS anchors is 5000 and 1000, the bp coverage is lower than BLASTZ. This is because of fewer MMSS anchors left after MASAA removes unnecessary MMSS anchors in the first stage of the algorithm. Out of the anchors selected for the second stage of the algorithm, there are only a few anchors which are within 5000 and 1000 bp from the longest MMSS in the last stage of the algorithm. Fewer MMSS anchors mean fewer inter MMSS anchors, hence we see a low bp coverage. On the contrary, when the minimum distance, 'd' is large, there are many MMSSs selected by MASAA and we see the bp coverage better than BLASTZ.

4.2.4.2 REAL data set

Experiment 24 (Total alignment time by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we vary minimum distance, 'd', to extend MMSS anchors for real sequences. The observations are shown in Figure 4.23.

Observation 24: Figure 4.23 shows that when the minimum distance 'd' between MMSS anchors is large, the time taken to align the sequences is also large. When the 'd' is 1000, we see an irregular, fluctuating line. This is mainly due to the absence of MMSS within 1000 bp from the previous largest MMSS during the extension phase of MASAA.

Experiment 25 (Bp coverage by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we vary minimum distance, 'd', to extend MMSS anchors for randomly generated sequences and observe the percentage of bp coverage. The

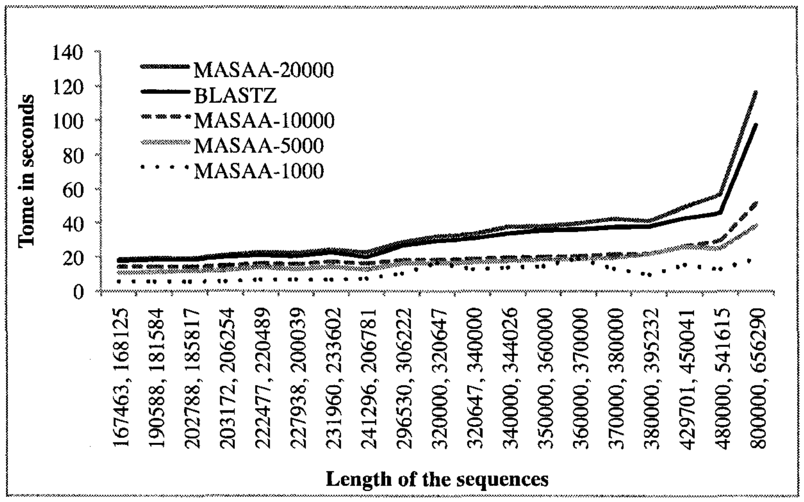


Figure 4.23: ***Total alignment time on REAL data set by varying the minimum distance, 'd' between MMSS***

observations are shown in Figure 4.24.

Observation 25: Figure 4.24 shows that bp coverage increases as the minimum distance 'd' is larger. When the minimum distance, 'd' is 1000, we see a curve which is fluctuating. This is due to fewer MMSS seeds in the final phase of the algorithm, resulting in poor bp coverage.

4.2.4.3 ROSETTA data set

Experiment 26 (Exon coverage by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we consider ROSETTA data set of 117 sequences and observe the percentage of exon coverage in the final alignment. The experimental observations are shown in Table 4.10. The numbers next to MASAA in the Table 4.10 refers to the bp distance within which MASAA searches for another

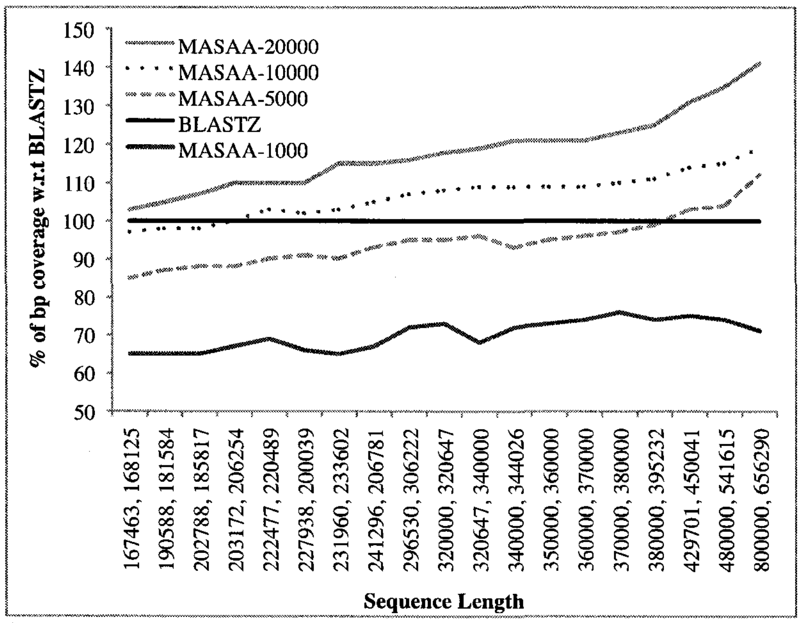


Figure 4.24: ***Bp coverage on REAL data set by varying the minimum distance, 'd' between MMSS***

MMSS in the last stage.

Table 4.10: ***Exon coverage on ROSETTA data set when minimum inter anchor distance is varied***

Aligner	100 exon	90 exon	70 exon
BLASTZ	94	97	98
MASAA-20000	97	97	98
MASAA-10000	94	94	96
MASAA-5000	81	87	94
MASAA-1000	68	61	82

Observation 26: From the Table 4.10, it is clear that the performance gradually increases as the length of the minimum distance 'd' is increased. We also observe that lowering to minimum length 'd' makes the algorithm perform poorly versus BLASTZ. This is due to smaller number of MMSS within the distance 'd' which in turn imply, smaller number of inter anchor and mismatch seeds.

4.2.4.4 SECOND data set

In the following experiments, we vary the minimum MMSS anchor distance, 'd' and observe variations in terms of sensitivity on SECOND data set.

Experiment 27 (Alignment score by varying minimum distance, 'd', to extend MMSS's anchor): In this experiment, we observe score of the final alignment. The experimental results are shown in Figure 4.25 and Table 4.11.

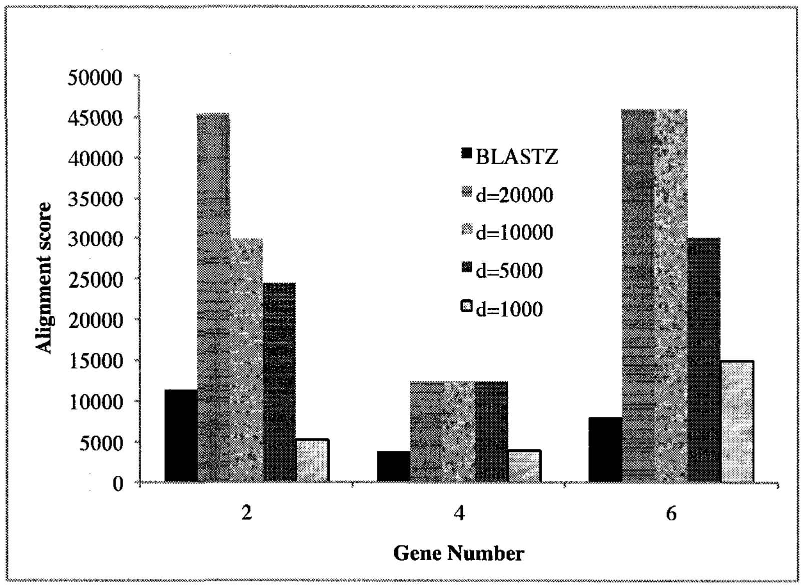


Figure 4.25: ***Alignment score on SECOND data set by varying the minimum distance, 'd' between MMSS***

Observation 27: Figure 4.25 and Table 4.11 shows that when the minimum distance, 'd', is increased from 10000 bp to 20000 bp, the alignment score increases. This is due to many MMSS anchors present and within minimum distance. Large number of MMSS anchors would also increase inter MMSS anchors, which in turn increases the alignment score. When the minimum distance, 'd' is decreased from 5000 to 1000, the alignment score drops down because of fewer MMSS's present within the minimum distance. With fewer MMSS within the minimum distance, MASAA would

Table 4.11: ***Number of genes MASAA's alignment score better than BLASTZ by varying minimum distance, 'd', to extend MMSS's anchor***

Aligner	No of genes
MASAA-%20000 bp	66
MASAA-%10000 bp	59
MASAA-%5000 bp	50
MASAA-%1000 bp	39

stop expanding from the largest MMSS as described in chapter three. This means fewer inter MMSS anchors, which in turn decreases alignment score.

Experiment 28 (Bp coverage by varying minimum distance, 'd', to extend MMSS's anchor) In this experiment, we observe bp coverage of the final alignment. The experimental results are shown in the Figure 4.26 and Table 4.12.

Observation 28: Figure 4.26 and Table 4.12 shows the bp coverage is better when the minimum distance 'd' between MMSS anchors is high. When the minimum distance between MMSS anchors is 1000, MASAA performs poorly. This is again due to fewer MMSS anchors within 1000 bp range.

Conclusion: In this section, we compared BLASTZ and MASAA by varying the varying minimum distance, 'd', to extend MMSS's anchor. From the experiments, we conclude that when the minimum distance, 'd', to extend MMSS's anchor is increased,

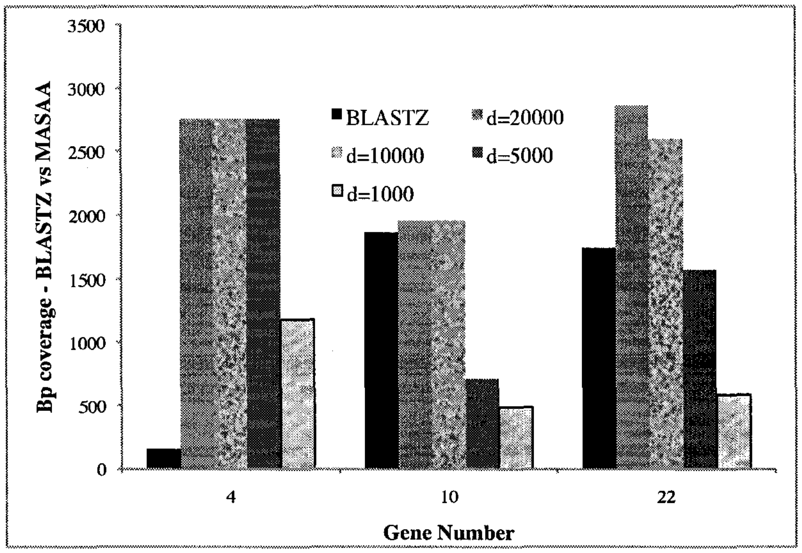


Figure 4.26: ***Bp coverage on SECOND data set by varying the minimum distance, 'd' between MMSS***

Table 4.12: ***Number of genes MASAA's bp coverage better than BLASTZ by varying minimum distance, 'd', to extend MMSS's anchor***

Aligner	No of genes
MASAA-%20000 bp	66
MASAA-%10000 bp	59
MASAA-%5000 bp	52
MASAA-%1000 bp	41

MASAA is slower and more sensitive than BLASTZ . If the minimum distance, ‘d’, to extend MMSS’s anchor is lowered then MASAA is faster but less sensitive than BLASTZ. We conclude by saying that, the minimum distance, ‘d’, to extend MMSS’s anchor considered in the algorithm is vital in determining the speed and sensitivity of the algorithm.

Chapter 5

Conclusion and Future Directions

Bioinformatics has lately become an active research area. There are many applications to be realized from the research in this field. Pairwise sequence alignment is a fundamental problem in bioinformatics and forms a vital step in solving other bioinformatics problems such as multiple sequence alignment, predicting ancestral sequence from two sequences and many others. Pairwise sequence alignment algorithms are of two types: local sequence alignment and global sequence alignment algorithms. Local sequence alignment algorithms focus on identifying a subregion which is most similar in both the sequences. Global sequence alignment algorithms concentrate on the whole sequence to detect regions of similarity. A considerable attention has been paid to solve pairwise sequence alignment problem lately, and various approaches have been proposed to solve both global and local pairwise sequence alignment problem.

Most of the algorithms developed lately are heuristic to overcome the disadvantages of optimal algorithms. Heuristic local sequence alignment algorithms use look-up table and a seed model to quickly ascertain a subregion of similarity while scanning the sequence from left to right. These subregions are then expanded both sides until the score of the subregion falls below a threshold. The process is completed when the best subregion is found. Depending on the sequence being scanned of the two and on the seed model used, the proposed algorithms differ.

Most heuristic global sequence alignment algorithms use suffix tree and anchors to align both the sequences. Certain variations in the data structure and anchor composition has given rise to many algorithms. The basic idea for both local and global alignment algorithms remain the same. First quickly identify regions of similarity and expand on these region to either find a subregion which aligns best with the other sequence (local sequence alignment algorithm) or use a collection of these subregions to align the whole sequence (global sequence alignment algorithm).

Both heuristic local and global sequence alignment algorithms fall into three categories: (1) fast algorithms, (2) sensitive algorithms, and (3) fast and sensitive algorithms. Our objective was to come up with a fast and yet sensitive heuristic local sequence alignment algorithm. From the literature we found that most of the local sequence alignment algorithms do not capture shorter subregions in their final alignment. This is because, they use longer seeds (subregion) to align sequences. As a result when aligning sequences of very low homology similarity, most algorithms would not align the sequences efficiently. We further investigated the causes for this short falls in BLASTZ which is the latest version of the popular algorithm BLAST.

On the other hand, most global sequence alignment algorithms lately have used suffix tree data structure to quickly align long sequences. The suffix tree data structure is capable of capturing short and long common subregions quickly. From our initial observation, we found that long common subregions were a part of the final alignment of most algorithms and this is the basis for our proposed algorithm. To overcome local sequence alignment algorithm shortcomings and take advantage of suffix tree, we propose Multiple Anchor Staged Alignment Algorithm - MASAA in this thesis.

MASAA borrows suffix tree and anchors from global sequence alignment algorithms and a seed model from local sequence alignment algorithms. MASAA quickly finds long subregions (maximal match substring-MMSS's). Further, it finds anchors in the regions between the MMSS's. Later, it finds small seeds between these anchors to finally align the two given sequences. MASAA is designed such that longer subregions are not lost in the final alignment and takes into consideration both overlapping and criss-crossing anchors and seeds. MASAA is not only faster than BLASTZ but outperforms BLASTZ in terms of sensitivity. To test the speed on longer sequences, we had two set of sequences, one which were randomly generated and the other which is a set created from the sequences taken from the Gene Bank. In order to test the sensitivity of the algorithm, we had two data sets, first data set was ROSETTA data set, which was the standard set used in the literature and second data set was a set of sequences from Human, Fruit fly, Fish, Worm, Fungi, and Bacteria.

From the simulations, we observed that for very long sequences, our algorithm, MASAA, performs better than BLASTZ in terms of speed. Also, as the length of the sequences increased, the bp (base pair) coverage of MASAA is better than BLASTZ. For small sequences, we did not find any significant differences in terms of speed. In terms of sensitivity, on ROSETTA data set, BLASTZ performed slightly better than MASAA. However, on sequences which had lower homology similarity than ROSETTA data set, MASAA outperformed BLASTZ. This clearly shows that

MASAA is comparable to BLASTZ for sequence which have high homology similarity and outperforms BLASTZ when the homology similarity is between 0 and 70 percent. MASAA's performance in terms of speed is closely related to the suffix tree and sensitivity, due to MMSS's, inter MMSS anchor and inter anchor seed design.

5.1 Future Direction

There are many directions in which the work proposed in this thesis can be expanded further. These include,

- The pairwise sequence alignment can be further extended to solve multiple sequence alignment.
- There are many variations of our current anchor model than can be explored. For example, the number of mismatches and a new seed model can be incorporated to improve sensitivity.
- An interesting step further is to solve the problem of global sequence alignment and expand further into protein sequence alignment.

Appendix A

A.1 Percentage of conserved region in SECOND dataset

The '?' in the table below indicates that there is no gene existing for that organism. All the genes are referred in numerical terms, that is first gene as '1', second gene as '2' and so on in the main chapters of this thesis. Fish, worm and fungi genes were collected for theoretical purposes and experiments in this thesis were conducted for fruit fly genes only.

Table A.1: ***SECOND data set genes conserved region in %***

Number	**Genes**	**Fruit fly**	**Fish**	**Worm**	**Fungi**
1	ABCD1;ALD	0	70	?	?
2	ABL1	74	?	70.2	?
3	ACE	0	0	?	0
4	ACOX	0	74.1	?	0

Continued on next page

Table A.1 – continued from previous page

Number	Genes	Fruit fly	Fish	Worm	Fungi
5	ACTN3	0	78	?	?
6	ADCAD2	0	78.7	?	?
7	AGL-GDE	0	73	0	0
8	AHCY	69.7	77.6	?	?
9	AMPD-AMP	0	73.1	?	0
10	ANk2	73	72	?	0
11	ARA	0	?	?	0
12	ATP7A-GRK	0	73.8	?	0
13	ATR	0	73.3	?	71.6
14	BRIC	0	0	0	74.4
15	CAR	0	71	?	?
16	CAT	72.4	75.1	0	0
17	CBP	0	73.1	0	?
18	CCO	0	84	0	0
19	CCT	73.6	76.2	72.5	70.8
20	DAR	0	?	?	?
21	DRP1	0	69.9	0	0
22	ERCC2	79.1	74.6	70	69.9
23	ERCC3,	72.9	77	?	70.5
24	Ext1	70	0	0	0

Continued on next page

Table A.1 – continued from previous page

Number	Genes	Fruit fly	Fish	Worm	Fungi
25	EXT2	74	75.6	?	?
26	FH,FHC	0	0	?	?
27	G6PD	75	?	?	?
28	GBE	0	?	?	?
29	GCE	69	?	?	?
30	GCLC	77	74	?	?
31	GPI	75.6	74.6	70.4	0
32	GS1	0	?	?	?
33	HERG;LQT2	73.1	71	77	?
34	HL1	0	?	?	?
35	HMG-CoA	0	?	0	?
36	HSP67B	0	72.4	0	0
37	HTT-OCD1	74.9	0	?	?
38	INSR	76.3	?	?	?
39	IVD	71.1	73.2	72.9	74.5
40	KIF1B	74.7	75.2	71.2	0
41	KIF5A	72.1	77.4	?	?
42	LAMB2	0	?	?	?
43	MSH2	0	72.8	71.8	75.3
44	MCM6	72.1	75.5	75.5	73.6

Continued on next page

Table A.1 – continued from previous page

Number	Genes	Fruit fly	Fish	Worm	Fungi
45	MTM1	0	73.3	0	0
46	MYH9	76.7	77.6	?	?
47	NAT1	0	?	?	0
48	NF1	70.5	?	0	0
49	NOS2A	0	73.2	0	0
50	NPC	0	?	?	?
51	OPA1	0	0	?	0
52	P300	0	?	0	?
53	PC	73	74.5	?	?
54	PHK	0	?	?	?
55	POMT2	75.9	76.3	?	?
56	POR	0	72.7	?	?
57	PTD	0	71.9	0	?
58	SCAD;ACAD	0	0	0	0
59	SCP2	0	75.5	?	0
60	SDH2	0	?	?	0
61	SEC	73.2	66.9	72.4	65.5
62	SLO	73.1	0	0	?
63	SMC3	71.3	76.3	70.3	72.2
64	SRC	77.1	75.3	71.9	?

Continued on next page

Table A.1 – continued from previous page

Number	Genes	Fruit fly	Fish	Worm	Fungi
65	SUR	0	75	0	0
66	RDP	0	73.3	?	0
67	RDX	0	74.1	?	?

A.2 Statistics for a mutation

Assume that there are mutations happening for 10^9 years. If there are 10^{18} organisms then, in 10^9 years, if there is 1 replication every hour for every day all year.

$= 10^9$ years$\times$1replication/hr$\times$24hr/day$\times$365days/yr

$= 10^9$ years$\times$ 10^4replications/year

$= 10^{13}$ replications.

If there are 10^{18} organisms, then $10^{13}\times 10^{13} = 10^{31}$ organismic replications

If there are 10^9 bp/organism $= 10^{40}$ bp-replications.

Assuming the fidelity to be 10^{-3}, then there are 10^{37} mutations, that is 10^{13} sequences.

If a DNA strand is 300 nt (nick translation), then there are 4^{300} sequences, that is approximately 10^{150} mutations.

Bibliography

[1] S. K. A. L. Delcher et al. Alignment of whole genomes. *Nucl. Acids. Res.*, 27(11):2369–2376, 1999.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *J Mol Biol*, 215(3):403–410, October 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schfferl, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402, September 1997.

[4] Anonymous. Sequence analysis: Which scoring method should i use?, July 2007. http://www.psc.edu/research/biomed/homologous/scoring_primer.html.

[5] Anonymous. 2can support portal - bioinformatics-fasta similarity search - introduction, 2008. http://www.ebi.ac.uk/2can/tutorials/nucleotide/fasta.html.

[6] S. Batzoglou, L. Pachter, J. Mesirov, B. Berger, and E. S. Lander. Human and mouse gene structure: comparative analysis and application to exon prediction. In *RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology*, pages 46–53, New York, NY, USA, 2000. ACM.

[7] N. Bray, I. Dubchak, and L. Pachter. Avid: A global alignment program. *Genome Res.*, 13(1):97–102, January 2003.

[8] B. Brejova, D. G. Brown, and T. Vinar. Vector seeds: an extension to spaced seeds. *Journal of Computer and System Sciences*, 70(3):364—-380, 2005. Early version appeared in WABI 2003.

[9] D. G. Brown. A survey of seeding for sequence alignments. In I. Mandoiu and A. Zelikovsky, editors, *Bioinformatics Algorithms: Techniques and Applications.* J. Wiley and Sons, 2007. To appear.

[10] M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, E. D. Green, A. Sidow, and S. a. Batzoglou. Lagan and multi-lagan: efficient tools for large-scale multiple alignment of genomic dna. *Genome Res*, 13(4):721–731, April 2003.

[11] M. Brudno and B. Morgenstern. Fast and sensitive alignment of large genomic sequences. *Proc IEEE Comput Soc Bioinform Conf*, 1:138–147, 2002.

[12] S. Chien, L. T. Reiter, E. Bier, and M. Gribskov. Homophila: human disease gene cognates in drosophila. *Nucleic Acids Res*, 30(1):149–151, January 2002.

[13] O. B. Dayhoff M, Schwartz R. A model of evolutionary change in proteins. in dayhoff, m. o. (ed.), atlas of protein sequence structure. *Natl. Biomedical Res.*, vol. 5, suppl. 3(1):345–352, 1978.

[14] S. Dreyfus. Richard bellman on the birth of dynamic programming. *Oper. Res.*, 50(1):48–51, 2002.

[15] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, January 1997.

[16] S. Henikoff and J. G. Henikoff. Amino acid substitution matrices from protein blocks. *Proc Natl Acad Sci U S A.*, 89(22):10915–9, 1992.

[17] D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.

[18] W. J. Kent. Blat–the blast-like alignment tool. *Genome Res*, 12(4):656–664, April 2002.

[19] M. Li, B. Ma, D. Kisman, and J. Tromp. Patternhunter II: highly sensitive and fast homology search. *J Bioinform Comput Biol*, 2:417–439, 2004.

[20] D. J. Lipman and W. R. Pearson. Rapid and Sensitive Protein Similarity Searches. *Science*, 227:1435–1441, Mar. 1985.

[21] B. Ma, J. Tromp, and M. Li. PatternHunter: faster and more sensitive homology search. *Bioinformatics*, 18:440–445, 2002.

[22] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[23] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443–453, March 1970.

[24] W. R. Pearson and D. J. Lipman. Improved tools for biological sequence comparison. *Proc Natl Acad Sci U S A*, 85(8):2444–2448, April 1988.

[25] K. D. Pruitt, T. Tatusova, and D. R. Maglott. Ncbi reference sequences (refseq): a curated non-redundant sequence database of genomes, transcripts and proteins. *Nucleic Acids Res*, 35(Database issue), January 2007.

[26] S. Schwartz, J. W. Kent, A. Smit, Z. Zhang, R. Baertsch, R. C. Hardison, D. Haussler, and W. Miller. Human-mouse alignments with blastz. *Genome Res*, 13(1):103–107, 2003.

[27] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[28] D. J. States, W. Gish, and S. F. Altschul. Improved sensitivity of nucleic acid database search using application-specific scoring matrices. *Methods: A companion to Methods in Enzymology*, 3(1):66–70, 1991.

[29] M. Tompa. Lecture notes on biological sequence analysis, January 2009. www.informatik.uni-kiel.de/fileadmin/arbeitsgruppen/technical_cs/Files-Jan/Tompa_lecture_notes.pdf.

[30] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[31] P. Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.