IGAUNA (IMPROVED GLOBAL SEQUENCE ALIGNMENT USING NON-EXACT ANCHORS)

by

Masoud Harati
B.S. Simon Fraser University, 2003

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Masoud Harati, 2007
SIMON FRASER UNIVERSITY
Spring 2007

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without the permission of the author.
APPROVAL

Name: Ilwoutl Hira
Degree: MASTER OF SCIENCE
Title of thesis: IGRUNA (Improved Global Sequence Alignment Using Non-exact Anchors)

Examinin Committee:
Dr. Gulay Tufeks
Professor, Computing Science,
Simon Fraser University,
Chair

Dr. Arvind Gupta,
Professor, Computing Science,
Simon Fraser University,
Senior Supervisor

Dr. Ranadee Krishna,
Professor, Computing Science,
Simon Fraser University,
Supervisor

Dr. Losikant Stolbo,
Assistant Professor of Mathematics,
Simon Fraser University,
SFU Examiner

Date Approved: 18072022
DECLARATION OF
PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection (currently available to the public at the “Institutional Repository” link of the SFU Library website <www.lib.sfu.ca> at: <http://ir.lib.sfu.ca/handle/1892/112> and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author’s written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada

Revised: Spring 2007
Abstract

With the sequencing of the entire genome for many species, bioinformaticians are increasingly relying on efficient whole genome alignment tools exhibiting both high sensitivity and specificity.

We introduce and analyze IGAUNA (Improved Global Alignment Using Non-exact Anchors), a new global alignment algorithm based on GAUNA, which in comparison with the other state-of-the-art algorithms, almost always produces as good or better alignments with high sensitivity and specificity using less time and space. While those tools either find exact or close-to-exact matches as anchors, IGAUNA makes use of suffix trees to find both types of anchors depending on the instance: Exact anchors for very similar sequences and otherwise non-exact anchors that are obtained from a complicated set of techniques. In particular, IGAUNA can rapidly align sequences containing millions of bases on a standard PC where other similar programs are currently incapable of accomplishing such a task.
I dedicate my work to my beloved parents for all their support.
"There are three kinds of people in this world:
Those who can count, and those who can't."
Acknowledgments

I am mostly grateful to my dear friend and colleague, Afroza Haj Khashbehshahi, for all his help and support on this mutual work; without his help this work would have not been possible. I deeply thank my supervisor Dr. Avraum Gupta for his support and invaluable remarks on my work. Great thanks to Chris Thelwall for his help in working with Emwebt genuine browser and GeneScan gene finder. I sincerely thank Jono Auldman and Deborah Joseph for their kindness and providing me with a copy of their paper. I thank my beloved parents for their unconditional love and continuous support. I should thank my dear laptop for being there for me and not letting me down. And last but not least, all my good friends whose support and encouragement has always been a big factor in my achievements, specially P., S., A., K. and M. (I have to use abbreviations to give me some level of freedom in assigning names to people and also avoid undesirable feelings like jealousy in others to some extent).
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Approval</td>
<td>ii</td>
</tr>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>iv</td>
</tr>
<tr>
<td>Quotation</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Contents</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
</tbody>
</table>

1 Introduction

1.1 Motivations
1.2 Our Contributions
1.3 Thesis Organization

2 Background

2.1 Biological Background
2.2 Biological Terms
2.3 Computer Science Background
3 Previous Work On Global Sequence Alignment ........................................ 15
  3.1 Scoring Methods ............................................................................. 15
  3.2 Dynamic Programming ..................................................................... 17
  3.3 Anchor-Based/Hit Methods ............................................................... 18
    3.3.1 FASTA .................................................................................. 18
    3.3.2 BLAST .................................................................................. 19
    3.3.3 BLAST .................................................................................. 21
    3.3.4 LAGAN .................................................................................. 21
    3.3.5 GLASS .................................................................................. 22
    3.3.6 MUMmer ................................................................................. 22
    3.3.7 CisatW .................................................................................. 23
    3.3.8 AVID ..................................................................................... 24
    3.3.9 MGA ...................................................................................... 24
  3.4 Genetic Algorithm ............................................................................. 25
  3.5 Hidden Markov Model ...................................................................... 25

4 General Algorithms For Anchor-Based Methods ................................... 27
  4.1 Building Suffix Trees ....................................................................... 27
    4.1.1 High-Level Ukkonen's Algorithm ............................................. 28
    4.1.2 Speedup Technique, Part 1 ...................................................... 29
    4.1.3 Speedup Technique, Part 2 ...................................................... 33
  4.2 Finding Maximum-Weight Anchor Set ............................................. 34

5 GAUNA ............................................................................................... 37
  5.1 Method description .......................................................................... 37
  5.2 Finding Maximal Unique Matches ................................................... 39
  5.3 Selecting Anchors ............................................................................ 44
    5.3.1 Finding Largest Total Weight Non-crossing Anchors ............... 44
  5.4 Closing The Gaps ........................................................................... 45
  5.5 GAUNA Parameters ......................................................................... 45
List of Tables

6.1 GAUNA Specificity ................................................................. 47
6.2 GAUNA Global Alignment Results ........................................ 49
8.1 K-value Effect ........................................................................ 61
8.2 Mouse Dog Alignment Results ................................................ 63
8.3 Mouse Chicken Alignment Results ........................................... 63
8.4 Human Dog Alignment Results ................................................ 64
8.5 Human Chicken Alignment Results .......................................... 64
8.6 Human Mouse Alignment Results ............................................ 65
8.7 Human Rat Alignment Results ................................................ 65
List of Figures

2.1 A suffix tree built on the sequence ATATAG. The terminal character S is added to ensure that each suffix ends in a leaf. 63

4.1 Figure (a) to (e) show the steps in using a suffix tree on the sequence ababa. 28

4.2 Figure (a) shows the implicit suffix tree for $S_1 = x$ node. Figure (b) shows the new implicit suffix tree for the new string $S_2 = x$ node which is obtained by adding character b to S. 30

4.3 Extension $j \geq 2$ of phase $k = 1$ [14]. 31

4.4 Skip/Count technique. In phase $i+1$, the substring $y$ has length 10. In order to find the end of a copy of $z$ starting from $x_i(y)$, we need to skip four nodes and count three characters. [14]. 32

5.1 To make the DP routine faster, only the area shown will be covered. This results in shorter sequences of indels in the matches. 42

5.2 In this figure $p_i$ is one of the locations in the suffix tree that is returned by Algorithm 6 for the suffix $S_2^7$ of $S_1$. $N$ is the set of occurrences of $Y_i$ in $S_1$ and $M_i$ is the set of exact matches corresponding to the location $p_i$. 44

5.3 A set of anchors is depicted. Rectangles represent the maximal matches and a set of good anchors is depicted in white rectangles connected by dashed lines. 44

8.1 Memory Usage in Megabytes for different sequences and tools. 66

8.2 Exon Coverage Length for Mouse-Dog and Mouse-Chicken. 66
8.2 Exon Coverage Length for Human-Rat, Human-Mouse, Human-Chicken, Human-Dog
8.4 Total Exon Coverage Length for Mouse-Dog and Mouse-Chicken
8.5 Total Exon Coverage Length for Human-Rat, Human-Mouse, Human-Chicken, Human-Dog
8.6 Improvements in Total Exon Coverage Length (compared to GAUNA)
8.7 Total Coverage Length for Mouse-Dog and Mouse-Chicken
8.8 Total Coverage Length for Human-Rat, Human-Mouse, Human-Chicken, Human-Dog
Introduction

After the discovery of DNA in 1953 [5], our knowledge of organisms and their building structures has vastly grown. This has in turn resulted in the creation of new branches of science dedicated to studying the main building blocks of life. Molecular biology, genetics and genomics are each new fields branching from biology and with the ever growing involvement of mathematics and computer science in these fields, disciplines such as computational biology have been created. Genetics is the area of biological study concerned with heredity and with the variations between organisms that result from it. Genomics is a recent scientific discipline with the aim of defining and characterizing the complete genetic makeup of an organism.

The inherent mathematical structure of DNA and the algorithmic processes used to express proteins has led to a close collaboration between molecular biology, computer science, and mathematics. As a result, computational biology has been created which is an interdisciplinary field that applies the techniques of computer science and applied mathematics to problems inspired by biology. As a discipline, computational biology is a relatively new field but there has been a virtual explosion of work in universities, government research labs and the private sector.

This field is relatively young and research in this field mainly started after the establishment of The Human Genome Project (HGP) in 1990. In 1994 the U.S. Department of Energy (DOE) established the Microbial Genome Program (MGP) as a companion to HGP and since then many new discoveries have been made in this field. Although from the computer science/mathematical point of view, a few algorithms with respect to sequence
alignment and sequence matching had already been developed prior to the start of these programs, the nature of DNA, proteins and genes requires much more efficient algorithms and in many cases there is a need for specialized algorithms to suit their specific characteristics. To date, algorithms based on dynamic and parallel programming, approximation methods, statistical and heuristic-based techniques have been developed, but there is still much work to be done.

1.1 Motivations

Genetics research in gene identification, phylogenetics, homology determination and a host of other problems requires tools for establishing the relationship between various biological sequences. One of the earliest problems identified was that of sequence alignment.

Sequence alignment problems are concerned with calculating the similarity of two sequences in terms of a scoring function. Alignment consists of pairing elements across the two sequences (or leaving some elements unpaired) while preserving the order of elements in both sequences. This definition of an alignment can be further extended by allowing changes in the order of elements to suit actual biological behavior. In its simplest form, the scoring function measurement is based on the edit distance (defined as the number of operations to change one sequence to the other) of the given sequences, however to give a more valid biological meaning to the final alignments, more sophisticated scoring methods are needed.

Using sequence global/local sequence alignment, we can identify similarities between DNA sequences which in turn result in identifying genes and proteins. Knowing more about proteins that are involved in nearly every aspect of the physiology and biochemistry of living organisms, can help us understand more about diseases and designing better drugs. Therefore we need sequence alignment methods that give us high quality alignments to be used to extract biologically valuable information about living organisms.

With the ever growing biological databases such as NCBI (National Center for Biotechnology Information) and PDB (Protein Data Bank), there is a huge amount of processed and unprocessed data available to scientists. Sometimes to find a good starting point to

\[\text{http://www.umass.edu/microbio/rasmol/pdblite.htm}\]
1.2 Our Contributions

Despite the existence of a few different programs and algorithms for global sequence alignment, there is still a need for better methods. Some of these programs produce alignments with high quality (such as LAGAN [32]) but require a lot of memory not available to users on a typical desktop/laptop computer, and their execution time is usually too long to be used frequently. The other ones that require less time to execute produce less reliable results with lower, and sometimes not acceptable qualities.

We introduce IGGAUNA, a new algorithm and program to find global pairwise alignments with very high quality results and in a very efficient manner, even executable on a typical laptop for large sequences.

We also introduce a new way of measuring the quality of sequences and introduce a so-called optimal global alignment between two sequences which can be used to measure the quality of a given alignment.

1.3 Thesis Organization

In Chapter 2, we will introduce some basic definitions and terms necessary to better understand the global sequence alignment problem. In Chapter 3, we will describe the different methods introduced in the literature to address this problem. In Chapter 4 we will describe the main algorithms used in anchor-based methods, specially since these algorithms are directly used in both GAUNA and IGGAUNA. In Chapter 5, we will describe GAUNA (on which IGGAUNA is based) in detail and show some results in Chapter 6. In Chapter 7 we will describe the improvements and enhancements done to the original GAUNA algorithm that resulted in IGGAUNA and will describe our method of building an optimal alignment as a means of benchmarking different alignments. Finally, in Chapter 8, we will describe our results in comparison with some other state-of-the-art programs and suggest possible future work and extensions.
Chapter 2

Background

In this chapter, we will introduce some basic terms, definitions and concepts, from both the Biology and Computer Science points of view, to help readers learn both disciplines comfortably follow the rest of the thesis.

As mentioned earlier, in today’s world, there is a close relationship between many forms of science previously considered unrelated. With the tight relationship between biology, chemistry, physics, computer science and mathematics, in order to correctly approach problems arising from biology and obtain meaningful results, one requires a decent knowledge of all the sciences involved in the particular problem being studied.

Interdisciplinary fields like bioinformatics, biochemistry and biophysics are the projections of biology in other fields of science. This collaboration has been bilaterally beneficial. Not only have chemical and physical explanations for biological phenomena been extremely useful, but at the same time they have become a source of inspiration in the original sciences involved. Computer science involvement in biology has been more recent, nevertheless the collaboration between them has proved to be very promising. Computer science can provide biologists the tools necessary to gather and interpret the enormous amount of data being produced. Tools created for analyzing data, modeling natural phenomena and providing theoretical models to explain biological phenomena are a few examples of how influential computer science has proved to be. On the other hand, influenced by biological concepts and nature, DNA computers and neural networks are good examples of how Biology has created new useful methods in Computer Science.

Bioinformatics is the descendant of what was (and sometimes still is) referred to as
Computational Biology. The work of earlier computational biologists is now widely used from the software for aligning DNA and chromosome sequences to that which predicts the secondary structure of RNA [11]. Although a lot of work has been done by these scientists, because of the complex nature of problems arising from Biology, there is still much to be done.

2.1 Biological Background

The problems in computer science arising from biology are different in nature from some other areas such as theory or graphics in that they originate from a practical and biologically science. Therefore it is extremely hard to model the problems correctly since many factors involved in the problems are not well known from the start. As Cohen points out, in biology, “No rules are without exceptions” [11]. Also, every problem is tied to other problems. For example, a simple change in a DNA sequence might lead to a completely distinct protein with different functionality.

Also, in many cases, the applications of results obtained from any such research will be applied to living entities and therefore before they are really put into practice, any result obtained from a theoretical point of view should be exhaustively tested in the labs to make sure it does not have any major side effects.

Having the above points in mind, the reasons why one should be familiar with the biological motivations behind any such problem becomes more clear. With a better understanding of the rules in the interactions between different living organisms and particles in the past decade, there has been an increased interest in research related to biology. Knowing the impact of such research can be a good motivation to further pursue activities in this field as well.

Some applications of biology are as follows: Diagnosis of disease by inspecting the DNA sequence and locating specific genes which are related to a certain disease or calculation of a disease risk in an individual; pharmacogenomics which is a fast-growing field in which different dosage and drugs are prescribed for different people according to their genome, i.e. the whole hereditary information of the organism encoded in DNA, which makes their responses to therapy dissimilar; identification of drug targets which are proteins whose functions can be modified selectively and help to cure a disease; and last but not least identification of missing or defective genes and replacement or supplying of its products
2.2 Biological Terms

Definition According to one of the most popular taxonomies, living things are divided into two major categories: **Prokaryota** or **Bacteria** kingdom consists of organisms that do not have a membrane-bound nucleus containing the genetic material of the cell. The genetic material for these can be found in strands (plasmids) within the cell's cytoplasm. The other group is called the **Eukarya** kingdom and consists of those that have a nucleus in which the genetic material is organized on chromosomes within a cellular nucleus. Eukaryotes are further divided into these four groups: Animalia, Plantae, Fungi and Protista kingdoms.\\(^{1}\)

Regardless of the originating organism or shape, size or behavior, there are common characteristics shared by all cells which resulted in the discovery of DNA. To understand what DNA is, we need some background and definitions.

Definition **Nucleic Acids** are organic compounds that make up the genetic material of living cells. They consist of long chains of **Nucleotides** which are categorized into A (Adenine), C (Cytosine), G (Guanine), T (Thymine) and U (Uracil).\\(^{1}\)

Living organisms contain two kinds of nucleic acids: deoxyribonucleic acid (DNA) and ribonucleic acid (RNA).

Definition **DNA** (Deoxyribonucleic Acid) is a nucleic acid that contains the genetic instructions for the biological development of all cellular forms of life (and many viruses).

DNA is often referred to as the molecule of heredity, as it is responsible for the genetic propagation of most inherited traits and it is replicated and transmitted to the offspring. The discovery of DNA happened in 1953 by James Watson and Francis Crick.\\(^{2}\)

Definition **RNA** molecules are much like DNA molecules, with the following basic compositional and structural differences.

---

\(^{1}\)http://www.microscopy-uk.org.uk/mag/artemys98/classif.html

\(^{2}\)http://www.mnh.si.edu/education/mindstorms/98/5.html
CHAPTER 2. BACKGROUND

1. RNA uses the nucleotide Uracil (U) where DNA would have used Thymine (T).
2. There is a slight difference in the nucleotide basic units that form the RNA.
3. RNA is single stranded where DNA is double stranded.

Definition: Chromosomes are the separate physical molecules which are arranged to form DNAs inside the cells.

Definition: The whole genetic information inside chromosomes is referred to as the genome.

When a cell is replicated, the entire genome inside the cell is copied into the new cells.

Definition: Amino Acids are organic compounds that link together to form proteins.

Proteins are involved in nearly every aspect of the physiology and biochemistry of living organisms. They carry out numerous functions such as growth and repair of tissue, allowing cells to detect and react to hormones and toxins in their surroundings, and are the chief ingredient in antibodies, which help resist infection. Enzymes are particular types of proteins that make possible a host of bodily processes [24].

Definition: Genes are specific sequences of bases that are subparts of DNAs.

Genes encode the recipes for making different proteins and then, the proteins determine our physical traits such as hair and eye color. Genes are estimated to comprise only 2% of the human genome and the remaining 98% is basically regions for which the functions are currently unknown. The existence of genes was postulated by Gregor Mendel in 1866 who devised a mathematical model for heredity of inherited characteristics [24].

Definition: An Exon (EXpressing region) is any region of DNA within a gene that is active in the coding of proteins and/or RNAs.

The term exon was coined by the American biochemist Walter Gilbert in 1978 [2]. The exons are typically multiples of three nucleotides (every triplet of bases called a codon is translated into a certain amino acid [26]). But not all the information inside the DNA is expressed as proteins or RNA, some regions of the DNA sequence are devoted to control mechanisms.

Definition. Exons of many eukaryotic genes interleave with segments of non-coding DNA called introns (INterVening seqUences), the functions of which are currently unknown to biologists.

With the above definitions in mind, DNA can be looked at as a template for building other DNAs and proteins. Based on these characteristics, the linguists tend to view a gene as a "word" while the genome can be thought of as a piece of "text". Structurally, DNA is organized as two complementary strands of bases with hydrogen bonds between them. Directions along each strand are named 3' end and 5' end and the correct direction of translation from DNA to protein is always from 5' end to 3' end [21].

Definition A phylogenetic tree, also called an evolutionary tree or also called a tree of life, is a tree showing the evolutionary interrelationships among various species that are believed to have a common ancestor.

In a phylogenetic tree, each node with descendants represents the most recent common ancestor of the descendants. Edge lengths sometimes correspond to time estimates showing the approximate time in which one species evolved to the other one [18].

2.3 Computer Science Background

Since DNA and protein sequences can be represented as sequences of nucleotides and amino acids, using computer science and mathematics has been the natural way to process these sequences. Sequence alignment provided a way to compare DNA and protein sequences with each other. Sequence alignment can be used to find similarities between different DNA/protein sequences in order to exploit their properties and give us a better understanding of the functionality of their different parts. This in turn results in a better understanding of the human body and making new drugs to cure diseases, build phylogenetic trees, etc. Nevertheless, the known algorithms were not efficient enough for the long sequences of DNA and therefore research in finding approximate algorithms resulted in many different approaches to solve this problem.

Exact string matching and sequence alignment are relatively old topics of computer science but their recent extensive applications in bioinformatics has resulted in renewed attention to these problems.
CHAPTER 2. BACKGROUND

Sequence alignment and finding the Longest Common Subsequence (LCS) of some given sequences date back to the 1970s where dynamic programming (DP) techniques were used to solve the problem. In [10] Knuth posed the question of whether finding the LCS of two strings can be done in less than quadratic time; Subsequently Manber and Myers proposed an algorithm that runs in $O(mn)$ where $m$ and $n$ are lengths of sequences and $n \leq m$ [31].

**Definition.** An Alignment consists of pairing elements across the two sequences (or leaving some elements unpaired) while preserving the order of elements in both sequences.

This definition of an alignment can be further extended by allowing changes in the order of elements to suit actual biological behavior.

**Definition.** The Global Sequence Alignment problem is defined as follows: Given two sequences A and B of lengths $m$ and $n$ respectively and a scoring function to give scores for matches/mismatches, what is the best way to align A to B; i.e., how should we align A to B to maximize/minimize (depending on what kind of scoring function we have) the total score.

Another issue to consider here is how to define a good alignment. Better scoring functions result in more meaningful alignments of sequences from a biological point of view.

**Definition.** The Optimal Global Alignment of two sequences is an alignment that maximizes a given measuring criteria.

In computer science and mathematics, this criteria is usually the scoring function and therefore, the optimal alignment is an alignment with the maximum score.

Global sequence alignment has many applications in biology. Finding similarities between DNA sequences can help us better understand how different species are related to each other (building better phylogenetic trees). Also, with the ever-growing databases of genes/proteins, there is more need to have an algorithm to search these databases efficiently.

Over the past decade, various methods have been developed for exact string matching and more importantly for approximate string matching which have resulted in software applications that are now widely used by scientists. Some of these methods are Dynamic Programming-based methods (Needleman-Wunsch), K-tuple methods (BLAST [3] and FASTA [28] and their improved versions in [2] and [29]), statistical methods (Hidden
2.4 Computer Science Terms

**Definition.** **Dynamic Programming (DP)** is a method of solving problems that takes much less time than naive methods. In order to solve a problem using DP, the problem should have two main properties:

1. Overlapping Subproblems
2. Optimal Substructure

Besides using these properties, the DP approach makes use of the so-called **Memorization**.

Having **Overlapping Subproblems** means that some problems of smaller sizes can be used to find the solution to a problem of the same type but bigger size. Sometimes finding the solution of a big problem is quite time/space-consuming, therefore instead of solving the big problem directly, we can solve smaller problems of the same type, save the solutions to the problems we have already solved and then, we can retrieve and reuse our already-computed solutions to find the solution to the bigger problem. This approach is called **memorization** (not to be mixed up with memorization, although this term also fits). In order to solve the big problem, we do not have to use all the solutions to the subproblems. If we are sure we will not need a particular solution anymore, we can throw it away to save space.

**Optimal Substructure** means that we can use the optimal solutions of subproblems to find the optimal solutions of the overall problem. Not all problems have this characteristic, and therefore DP is not always applicable to all problems.

In general, we can solve a problem with optimal substructure using a three-step process:

1. Break the problem into smaller subproblems.
2. Solve these problems optimally using this three-step process recursively (The subproblems are broken into smaller sub-subproblems and so on, until a case is reached for which we can solve it easily or the solution is trivial).

The German site [http://www.inf.uni-hamburg.de/jgk/groups-eng/siteID=48](http://www.inf.uni-hamburg.de/jgk/groups-eng/siteID=48)
3. Use these optimal solutions to construct an optimal solution for the original problem.

Definition: We denote the subsequence of the sequence $S$ starting at position $i$ and ending at position $j$ with $S[i, j]$.

Definition: Given two sequences $S_1$ and $S_2$, a quadruple $(i_1, i_2, l_1, l_2)$ is called a match if the optimal alignment score of the two subsequences $S_1[i_1, i_1+l_1-1]$ and $S_2[i_2, i_2+l_2-1]$ is greater than or equal to a certain threshold.

Definition: If for two matches we have $S_1[i_1, i_1+l_1-1] = S_2[i_2, i_2+l_2-1]$, the match is called an exact match; otherwise, it is called an inexact match.

Definition: Following the definition of Debeler et al., a match $(i_1, i_2, l_1, l_2)$ is called maximal if it cannot be extended at either endpoint [13]. For inexact matches we generalize this definition as follows: A match $(i_1, i_2, l_1, l_2)$ is maximal if there is no other match $(i'_1, i'_2, l'_1, l'_2)$ such that $S_1[i'_1, i'_1+l'_1-1]$ is a proper subsequence of $S_1[i_1, i_1+l_1-1]$ and $S_2[i'_2, i'_2+l'_2-1]$ is a proper subsequence of $S_2[i_2, i_2+l_2-1]$.

Definition: Consider two matches $(S_1, S_2)$ and $(S_3, S_4)$ in the two sequences and assume that $S_1$ starts before $S_3$. If $S_1$ and $S_3$ overlap or $S_2$ starts before $S_4$ ends, then we call these crossing matches. We refer to non-crossing matches as anchors. Figure 5.4 shows an example of maximal matches and anchors.

Definition: Given two sequences $S_1$ and $S_2$, a set of matches $M$ between $S_1$ and $S_2$ and a set of "good" regions $G$ that consists of subsequences of $S_1$ and $S_2$, we can define Sensitivity and Specificity. Sensitivity is the ratio of good regions of $G$ we have "hit" with our matches in $M$ to the total number of good regions. A match has "hit" a region if it overlaps with that region and we refer to such a match as a "good" match. Specificity is the ratio of good matches in $M$ to the total number of matches we have found.

From a biological point of view, we can define a good region to be an exon and define a match to be good if it overlaps with an exon.

Definition: A suffix of a string $T = t_1 \ldots t_n$ is a string $T' = t_m \ldots t_n$ where $m \leq n$. A proper suffix of a string is not equal to the string itself and not empty ($0 < m < n$). A suffix can be seen as a special case of a substring [24].
Definition A prefix of a string \( T = t_1 \ldots t_n \) is a string \( \hat{T} = t_1 \ldots t_m \), where \( m \leq n \). A proper prefix of a string is not equal to the string itself and not empty \( (0 < m < n) \) [14].

Definition A Binary Tree is a tree-like structure that is rooted and each vertex has at most two children, and each child of a vertex is designated as its left or right child [4].

Definition A Trie is a tree structure for storing strings in which there is one node for every common prefix.

In a Trie, all the descendants of any one node have a common prefix of the string associated with that node, and the root is associated with the empty string. Each edge is labeled with the characters whose addition to the end of the parent node gives the string stored in the child. Values are normally not associated with every node, only with leaves and some inner nodes that happen to correspond to keys of interest. The final strings are stored in extra leaf nodes. The name comes from retEek and is pronounced as "tree" or "try".

Definition A suffix tree (also called PAT tree or, in an earlier form, position tree) is a tree-like structure built based on a given string \( S \) and the degree of each vertex (node) can be at least the size of the alphabet that the string is built on. Each edge is labeled with a non-empty string such that each path from the root to a leaf corresponds to a unique suffix of the string \( S \).

When building a suffix tree for a string, sometimes a suffix of the string can be part of a longer string and therefore, its end position might be on an edge or an internal node of the tree. To ensure that each suffix actually ends at a leaf, a unique character which is not part of the alphabet, is added to the end of the string. This character is usually denoted by \( $\) and is called the terminal symbol. Hence, the suffix tree is actually built on \( SS \). Figure 2.1 shows a suffix tree built on the sequence \( ATTATG \).

A suffix tree on a string \( S \) of size \( n \) can be constructed in \( O(n) \) time and space. Once made, certain operations on \( S \) such as finding the location of a substring of \( S \) (and its possibly several occurrences) and finding the longest common substring between two strings can be accomplished in linear time [14].

[^1]: http://en.wikipedia.org/wiki/Trie
Figure 2.1: A suffix tree built on the sequence ATTATG. The terminal character $\delta$ is added to ensure that each suffix ends at a leaf.

**Definition** A Genetic Algorithm (GA) is a programming technique that mimics biological evolution as a problem-solving strategy. It is basically a search technique used in computing to find true or approximate solutions to optimization and search problems.

Genetic algorithms are categorized as global search heuristics and are a particular class of evolutionary algorithms. GAs were introduced by John Holland who began his work on genetic algorithms at the beginning of the 60s and his achievements were first published in a book titled *Adaptation in Natural and Artificial Systems* in 1975 [26].

Given a specific problem to solve, the input to the GA is a set of potential solutions to that problem, encoded in some fashion, and a metric called a *fitness function* that allows each candidate to be quantitatively evaluated. These candidates may be solutions already known to work, with the aim of the GA being to improve them, but more often they are generated at random.

The GA then evaluates each candidate according to the fitness function. In a pool of randomly generated candidates, of course, most will not get a reasonable score, and they will be deleted. However, a few of the instances might get a reasonable score (above activity) and these can be used toward further solving the problem. These candidates are kept, and multiple copies of them are produced, however some changes are introduced in the copying process. To make the changes, GA uses some operations inspired by evolutionary biology such as inheritance, mutation, selection, and crossover (also called recombination). Using the new generation produced, GA repeats the above process and the expectation is that the

[http://www.talkergina.org/64p/genalg/genalg.html#what](http://www.talkergina.org/64p/genalg/genalg.html#what)
average fitness of the population will increase each round, and so by repeating this process for hundreds or thousands of rounds, very good solutions to the problem can be discovered. One big advantage of GAs is their ability to be implemented using parallel processors to speed up the process.6

Definition A *Hidden Markov Model (HMM)* is a statistical model in which the system being modeled is assumed to be a Markov process with unknown parameters.

Based on observable parameters, the hidden parameters are deduced. The extracted model parameters can then be used to perform further analysis, for example for pattern recognition applications such as speech, handwriting, gesture recognition, musical score following and in bioinformatics. Viterbi and Forward/Backward algorithms are used to determine the hidden probabilities of an HMM based on observable parameters. Since we are not using HMMs in this thesis, we refer the interested reader to [37] for more details and information.7

6http://en.wikipedia.org/wiki/Genetic_Algorithms
7A concise description of HMMs and the related algorithms can also be found at http://www.cs.brown.edu/research/sdn/jamorita/tutorial/documents/HiddenMarkovModels.html
Previous Work On Global Sequence Alignment

In this section we will present the most influential literature in this field. There have been many tools introduced for local/global alignment based on different methods such as dynamic programming, anchor-based methods, genetic algorithms and Hidden Markov Models. We will describe the most influential ones briefly, however the focus of this chapter will be on anchor-based methods, firstly because most of the current programs being used are based on this method and secondly because IGAUNA is an anchor-based method.

In order to find similar regions between two sequences, there is a need for some kind of a measurement. Therefore we need to have a scoring function that assigns scores to matches and mismatches (deletions, insertions) and most algorithms try to maximize/minimize this score to find the best alignment. The choice of scoring function and definition of a good alignment are important topics that we will also address.

3.1 Scoring Methods

The simplest way to define the scoring function is the so-called **Hamming distance**.

Definition: Hamming distance is the cost of the cheapest set of changes needed to transform one sequence into the other.

---

1There is a list of alignment tools at [http://www.mavens.com/topic/sequence-alignment-software](http://www.mavens.com/topic/sequence-alignment-software)
We do not allow any gaps and therefore the two sequences should have the same length and usually every change (substitution) has a cost of 1. For example to convert ACCCGG to ACCGCA, we need two changes: change T to C and G to A. Therefore the hamming distance between these two sequences is 2.

The next step is to introduce gap penalties. A gap corresponds to an *inert*, i.e. a deletion in one sequence or an insertion in the other.

**Definition** The *edit distance* or *Levenshtein distance* between two sequences is the minimum number of operations needed to transform one into the other, where an operation is an insertion, deletion, or substitution of a single character. In edit distance, the cost of insertions, deletions and substitutions are all the same and equal to 1.

The next step is to have a linear scoring function in terms of number of deletions / insertions / substitutions. We can have a constant gap penalty (i.e. score = # of matches - # of mismatches - 2*deletions/insertions), or an affine gap penalty. Affine gap penalty specifies a penalty to start a gap and a different one to continue the gap. For example a scoring function that has constant scores for mismatches and assigns scores to gaps based on an equation of the form $a(x - 1) + b$, where $x$ is the number of gaps with $a$ and $b$ constants, is an affine function.

A scoring function can take a more complicated, non-linear form as a function of the number of deletions/insertions/substitutions, but since that increases the complexity of alignment algorithms, they are not widely used in current alignment methods.

A more general and still easy-to-calculate method of assigning scores is to use matrices this contains the score of pairwise alignments between every pair of the alphabet being used. Such examples are PAM [12] and BLOSUM [15] which are widely used in protein alignment algorithms. We use a similar matrix for pairwise nucleotide scores in IGMAUNA.

The choice of scoring function can have a great impact on the quality of the final alignment. Based on the algorithm used, the best alignment achievable can be dependant on the choice of the scoring function chosen. However there exist some commonly accepted scores based on experiments performed on actual biological sequences (PAM and BLOSUM mentioned above are such examples).
3.2 Dynamic Programming

The famous Needleman-Wunsch algorithm [34] is a dynamic-programming-based algorithm presented in [34] which was the first major step in the development of relatively fast global sequence alignment algorithms. Many global and local alignment algorithms are influenced by this algorithm. It requires quadratic time and space in the length of inputs and is based on the following observation:

For two sequences \( s[1..n+1] \) and \( t[1..m+1] \), one of the following cases must hold for the best alignment:

1. Last position is \((s(n+1), t(m+1))\)
2. Last position is \((s(n+1), t'_{m+1})\)
3. Last position is \((t'_{n+1}, t(m+1))\)

where \( ' \) represents aligning with a gap (i.e., insertion in one sequence or deletion in the other).

Define \( V[i+1, j+1] \) to be the score of optimal alignment of \( s[1..i+1] \) and \( t[1..j+1] \), then based on the above observations:

\[
V[i+1, j+1] = \max \left\{ \begin{array}{l}
V[i, j] + \sigma(s[i+1], t[j+1]) \\
V[i, j+1] + \sigma(s[i+1], t'_{j+1}) \\
V[i+1, j] + \sigma(s'_{i+1}, t[j+1])
\end{array} \right\}
\]

where \( \sigma(x, y) \) gives the score of aligning character \( x \) with character \( y \).

Using this recursive equation, we can dynamically build a table and use the values stored in the previous rows/columns to ultimately calculate \( V[n+1, m+1] \). Using this table, we can trace back and build the actual alignment.

Based on the Needleman-Wunsch algorithm, the authors in [33] improve the space complexity of Needleman-Wunsch algorithm by using a nice trick in the DP table. Instead of keeping the whole table, they only keep the last row and column and therefore they use linear space in the length of the inputs, and there is no change in the running time. If the actual optimal alignment is desired (instead of just the score of the optimal alignment), the running time will increase, but the magnitude will stay the same (i.e., quadratic).

With a slight change to the formula described above, the Smith-Waterman algorithm for local alignments can be obtained [36]. It has quadratic running time and space complexity,
but using a similar trick presented in [33], a linear space complexity can be achieved without changing the running time complexity (although if we need to reconstruct the actual alignment, we will again need more time, although with the same complexity order).

### 3.3 Anchor-Based/Hit Methods

All these algorithms are based on finding anchors, short non-crossing regions of very high similarity in the two sequences being aligned. Once identified, these are extended to regions of high similarity which are then claimed together to form a global alignment. In designing a global alignment algorithm, developers must contend with the tradeoff between anchor size, running time, and the quality of the alignment. Short candidate anchors will occur in relatively large numbers increasing the running time of the algorithm while making it more likely that biologically insignificant anchors are chosen. This decreases the specificity of the solution, because we find many matches of little significance. Conversely long candidate anchors are more likely to miss biologically significant regions, decreasing the sensitivity of the solution. Thus an important consideration in all of these algorithms is parameter setting of anchors length.

Many algorithms to date focus on finding exact anchors, that is anchors which are exactly the same. Exactness forces anchors to be short since even one point of difference will eliminate a candidate anchor, thereby decreasing specificity. Yet sensitivity is also compromised since anchors that are biologically less significant are more likely to eliminate regions of higher significance. These algorithms use suffix trees to find anchors; suffix trees are ideal for this purpose and ensure the algorithms are computationally efficient.

#### 3.3.1 FASTA

The first satisfactory attempt to use an approximate alignment algorithm in practice was FASTA [28] introduced in 1995. FASTA was designed to compare a query sequence to every sequence in a database. To do this, FASTA first finds all matching words of length $k$, which are called $k$-tuples. Using a DP table, so-called "hot spots" and "best diagonal runs" are found and then FASTA tries to extend the alignments. Up to the end of this stage no gaps are used. After that FASTA uses gapped alignments to join the ungapped regions.

*The online web application can be found at [http://www.ebi.ac.uk/ fasta3/](http://www.ebi.ac.uk/fasta3/)*
In [36] (Pearson and Lipman (working at NCBI) at the time) make some improvements to FASTA which in turn would be the basis for the development of BLAST described in Section 3.3.2. These improvements include finding better heuristics and using hash tables to speed up the look-up process in the database.

3.3.2 BLAST

BLAST (Basic Local Alignment Search Tool) was first introduced in 1990 [3] and is the most cited paper of the 90s. We will describe BLAST in more detail since it is very widely used and also IGAS/NA uses BLAST in building the so-called optimal alignment.

To describe BLAST, we begin by describing the concept of Maximal Segment Pair (MSP):

Definition. Maximal Segment Pair (MSP) is a pair of identical length segments (subsequences) chosen from two sequences with a sum of scores $S$ that cannot be extended in either direction without introducing gaps/mismatches.

Given two sequences of lengths $m$ and $n$, the number of MSPs with score at least $S$ is well approximated by

$$y = K \cdot n \cdot e^{-\lambda S}$$

where $K$ is a computable function of $S$ and $\lambda$ is the probability of finding an MSP with score greater or equal to $S$ [22].

This result gives meaningful semantics to $S$. Given the size of the database and a scoring system, the result determines what minimal scores we need to look for in order to not get random hits.

The general algorithm of BLAST can be summarized as follows:

**Step 1:**
Find all subsequences of length $W$, such that their score against the query $Q$ is at least $T(<\lambda S)$. $W$ is typically equal to 3-5 and 14-12 for proteins and DNA sequences respectively.

**Step 2:**

---

1National Center for Biotechnology Information \(\text{http://www.ncbi.nlm.nih.gov/}\)
2The online web application can be found at \(\text{http://www.ncbi.nlm.nih.gov/BLASTinfo/information.html}\)
3\(\text{http://en.wikipedia.org/wiki/BLAST}\)
Extend the subsequences along the diagonal (i.e., no gaps) to get MSP. Stop extending if the score falls $X$ bits below the max score associated with the given subsequence. Filter those MSPs whose score is $< Y$.

**Step 3:** Extend the MSPs with gaps until the score falls below the threshold given.

To efficiently perform Step 1, we do as follows:

- **1.1** Run a window of size $w$ through query $Q$ and create a list of words of size $w$ that match each query word of size $w$ with score $> T$. As an example, consider $Q = AHBTC$ and $w = 3$. We will have 3 windows AHB, BHT, RTC.
  - AHB might match words ABT and AHU.
  - BHT might match BRP and BHC.
  - RTC might match just RTC.
- Therefore the compiled list contains ABT, AHB, BHT, BRC and RTC.

- **1.2** Find occurrences of all words from the list in the database:
  - 1.2.1 Compile database into a lookup table from subsequences of length $w$ to positions in the database.
  - 1.2.2 Compile the list of words into a database for fast look-ups.

In order to compute the complexity of BLAST, we have to know "what is a chances $\phi$ of a $T$-scoring sequence not having a $T$-scoring word of size $W$". Experimental results show that given $T$ and $W$, one can find $w$ and $b$ such that $\phi = e^{-w+6b}$. Based on this, let $W$ be a number of words generated for an input query in Step 1 and $N$ be a number of residues in the database. Then complexity of BLAST is $O((W + br + \frac{W^2}{N})$.

After introduction of the original BLAST, many different versions aimed at different types of sequences (i.e. amino-acids, proteins, etc.) and for different platforms were developed. Some of these are BLASTN, TBLAST, BLASTX, PSI-BLAST, GAPPED-BLAST, MEGA-BLAST, PSA-BLAST, WU-BLAST, BLAT, etc.

**PSI-Blast** (Position-Specific Iterative BLAST) and **GAPPED BLAST** are introduced in [2]. The idea behind GAPPED-BLAST is as follows: The original BLAST finds a single word of length $w$ that scores at least $T$ against the query. But if we find two words of length $w$ and score $T$ that lie on the same diagonal within distance $A$ from each other, then if $T' = T$ satisfies certain criteria, we just extend $T'$ allowing gaps too, to reach $T$. Using this method, we end up with more hits, but since we cover more of the string (by connecting $T$
and $T$), the extensions are faster. PSI-BLAST is used to find distant relatives of a protein. PSI-BLAST relies on the fact that the search can be improved, if the "important" parts of the query are known. It will first run normal BLAST to find local alignments and based on that, will make a so-called position specific matrix. Then it will use this matrix to find better hits and iteratively repeats this procedure. This method results in better sensitivity.

3.3.3 CHAOS

CHAOS (Chain Of Scores) is presented in [8]. It is an anchor based tool for local alignments and is the basis for LAGAN and Dialign global alignment programs. CHAOS first finds a chain of local alignments of size $k$ (by using Trie data structure), uses as anchors, and extracts them using a global alignment algorithm such as Needleman-Wunsch [34]. The more dense a chain, the higher the gain in speed since the search space will be reduced. On the other hand, if too many anchors are found, the search space size and therefore the alignment quality will decrease. For these reasons, size $k$ should be chosen carefully.

3.3.4 LAGAN

LAGAN, which is currently one of the best global alignment tools is introduced in [32]. LAGAN is slightly different from the other anchor-based methods described in this section, because firstly it uses exact matches as anchors and secondly it does not plant the anchors directly into the final alignment. LAGAN uses the set of anchors as a tool to narrow down the search space for the optimal alignment in the standard dynamic programming table.

The three main steps of LAGAN are as follows:

1. Generate local alignments between the two sequences using a local alignment tool such as CHAOS [8] (which uses Trie) or PASI [22].
2. Construct a rough global map by chaining an ordered subset of local alignments obtained from step 1.
3. Compute the final global alignment by finding the best alignment that stays within the limited area around the rough global map.

MLAGAN is an extension of LAGAN used for multiple global alignment in which a multiple alignment of $r$ sequences is constructed in $r - 1$ pairwise alignment steps, whereas in each step two input sequences or intermediate sequences are aligned using LAGAN.
3.3.5 GLASS

GLASS (Global Alignment System) developed by a group at MIT is presented in [39] \(^2\).

GLASS works as follows:

1. For an initial \( k \), find all matching \( k \)-mers (\( k \)-long words).
2. Treat each matching \( k \)-mer as a unique \( "k \)-mer character", convert both sequences into strings of such characters.
3. Align the two sequences using non-standard DP as follows: Matching \( k \)-mers receive a score equal to the sum of the alignment score obtained by applying regular DP to the short region flanking the occurrence of the \( k \)-mer in the two sequences (i.e. 12 nucleotides to the left and 12 nucleotides to the right). Mismatches and gaps in the alignment of \( k \)-mers have a score of 0.
4. Identify those matching \( k \)-mers that have a score exceeding a threshold \( T \).
5. Remove those inconsistent matching \( k \)-mers. Two \( k \)-mers are inconsistent if they correspond to positions that overlap by at least 1 nucleotide in one sequence, but not in the other sequence.
6. For the remaining list of matching \( k \)-mers, recursively align the regions of underlying nucleotides, by repeating step 1-5 using smaller \( k \) (\( k = 20, 15, 12, 9, 7, 6, 5 \)).
7. Once the last recursive alignment is performed, extend all pairs of aligned segments by short local alignments to the left and right by regular DP.
8. Finally align the remaining unaligned region using regular DP.

3.3.6 MUMmer

MUMmer, a relatively fast global alignment algorithm presented in [12] \(^3\). It uses suffix trees to find matches between two strings.

MUMmer uses maximal unique exact matches called MUMs as anchors. The uniqueness of a match in the two sequences means that there has to be only one copy of the matching

\(^2\)The source code can be found at http://crescent.species.lcs.mit.edu/
\(^3\)The program itself can be found at http://www.tigr.org/software/mummer/
3.3.7 ClustalW

ClustalW is another alignment program introduced in [16] used mainly for protein alignments* . It is more sensitive than the other commonly-used global alignment methods by using the following method**: 

*It is available at http://www.ebi.ac.uk/clustalw/
**http://bimas.dcrt.nh.gov/clustalw/clustalw.html
• Firstly, individual weights are assigned to each sequence in a partial alignment in order to down-weight near-duplicate sequences and up-weight the most divergent ones.
• Secondly, amino acid substitution matrices are varied at different alignment stages according to the divergence of the sequences to be aligned.
• Thirdly, residue-specific gap penalties and locally reduced gap penalties in hydrophilic regions encourage new gaps in potential loop regions rather than regular secondary structure.
• Fourthly, positions in early alignments where gaps have been opened, receive locally reduced gap penalties to encourage the opening up of new gaps at these positions.

3.3.8 AVID

AVID [7] is another good anchor-based program used for global alignments. It uses suffix trees to find maximal matches, but unlike MUMmer, it uses maximal exact matches as anchors. After finding maximal matches, it removes the "noisy" matches. A match is considered "noisy" if its length is less than half the length of the longest match in the current step. AVID then chooses the anchors using a variant of Smith-Waterman algorithm [30] which become part of the final global alignment. This phase takes quadratic time in the number of matches and with a better variation it takes $O(n^2p)$ where $p$ is the number of matches. AVID will close the gaps, however, instead of finding matches again, AVID uses these initially found. Matches which overlap with the unaligned region under consideration are chosen and the inter-anchor region is aligned recursively using the anchor selection phase. Once the recursion is completed, AVID aligns the remaining unaligned regions using the Needleman-Wunsch algorithm [34] if they are sufficiently short and otherwise leaves these regions unaligned. MAVID [6] is a progressive multiple alignment tool that incorporates AVID.

3.3.9 MGA

MGA (Multiple Genome Aligner) [17] is a multiple alignment algorithm based on similar techniques as AVID but with more efficient algorithms and data structures to perform each phase. In phase one, MGA uses suffix arrays [18] instead of suffix trees to find exact matches.

A suffix array is a sorted list of the suffixes of a given string.
In the second phase, it computes the anchors in \(O(n \log n)\) time by using the algorithm described in [21]. Finally MGA closes the gaps between anchors by applying the same method recursively with smaller thresholds for matches. The remaining gaps are closed by the multiple sequence alignment program CLUSTALW when they are short, while long gaps remain unmatched.

3.4 Genetic Algorithm

In the past few years, Genetic Algorithms (GA) have found their way into different branches of Computer Science and Bioinformatics has been no exception. However tools based on this method are more experimental than others introduced earlier. SAGA (Sequence Alignment by Genetic Algorithm) [32] is one such tool that can be used both for pairwise and multiple sequence alignments and it can easily be implemented on parallel processors.

SAGA works as follows:

1. Initialize: Shift each sequence randomly (\(\leq \) longest sequence length) to left and consider that an alignment
2. Modify the sequences randomly using an operator chosen based on heuristics
3. Assess Score
4. Keep or discard alignment
5. Repeat until satisfied

SAGA uses a set of 20 operators which are specifically designed for this problem. Since it is a GA algorithm, it can easily be modified to be used in parallel systems. The two-tournament model is an Island model (for which there are different independent population of species who develop separately from each other and are exchanged according to some rules), where in every 5 generations, the fittest individuals between evolving populations are exchanged, and the fittest ones replace the low-performing ones.

3.5 Hidden Markov Model

Another approach for sequence alignment is using Hidden Markov Models (HMM). First the topology of an HMM model should be designed which is highly dependent on the
problem, then transmission and emission probabilities should be decided by training using Forward/Backward algorithms. Then the Viterbi algorithm can be used to align sequences.

One good feature of HMMs is that they can be used to identify whether a sequence belongs to a particular family of sequences (i.e. proteins) [99]. However, this approach is not as popular as other methods, because the topology of the HMM model is highly dependent on the particular problem and the sequences being studied. We also need a large number of sequences in order to train the HMM and find the transmission/emission probabilities.
Chapter 4

General Algorithms For Anchor-Based Methods

As we mentioned in Chapter 3, many anchor-based methods use suffix trees to find matches. We will explain in detail how suffix trees can be built in $O(n)$ time and space using an algorithm called the Ukkonen algorithm as described in [14] by Dan Gusfield. We will also present an algorithm that can efficiently be used to choose anchors from a set of given matches developed by D. Joseph, J. Mehlhorn and P. Tarini as described in [21]. GAUNA and IAUNA make use of this algorithm to select anchors.

4.1 Building Suffix Tree

Using a naive approach to build a suffix tree on a string $S[1..n]$, takes $O(n^2)$ time and space. We can do that in an iterative way as follows: make the tree by making a root and an edge and label the edge with the longest suffix of $S$, i.e. $S$ itself. Then take the next suffix by eliminating the first character of the previous suffix and traverse the tree starting from the root. As long characters are found that match the current suffix on the tree, follow the edges and branches. When a character that does not match the next character on the tree is encountered, create a new branch and an edge and label the edge with the remaining characters of the current suffix. An alternative method using the same idea is to start from the shortest suffix and add longer suffixes in each iteration. Figure 4.1 shows this process. Each suffix of length $m$, can be added to the tree in $O(m)$ time and therefore the total
4.1.1 High Level Ukkonen's Algorithm

Definition An Implicit Suffix Tree on string $S$, is a tree obtained from the suffix tree for $S$ by removing every copy of the terminal symbol $\$ from the edge labels of the tree, then removing any edge that has no label, and then removing any node that does not have at least two children. We denote the implicit suffix tree of the string $S[1..i]$ by $I_i$.

An implicit suffix tree on $S$ includes all the suffixes of $S$, but some suffixes might not end at a leaf. Figure 4.2(a) shows an example of an implicit suffix tree.

Ukkonen's algorithm is divided into $n$ phases. In phase $i + 1$, tree $I_{i+1}$ is constructed from $I_i$. Each phase $i + 1$ is further divided into $i + 1$ extensions, one for each of the $i + 1$ suffixes of $S[1..i+1]$. In extension $j$ of phase $i + 1$, the algorithm first finds the end of the path from the root labeled with substring $S[i..j]$. It then extends the substring by adding the character $S[j+1]$ to its end, unless $S[j+1]$ already appears there. $I_i$ is just the single edge labeled by character $S[1]$.
Algorithm 1 High-level Ukkonen Algorithm

1: Construct $I$
2: for $i$ from 1 to $m-1$ do
3:  \[ \{ \text{performing phase } i+1 \} \]
4:  for $j$ from 1 to $i+1$ do
5:   \{ performing extension $j$ \}
6:   Find the end of the path from the root labeled $S[j..i]$ in the current tree. If needed, extend that path by adding character $S[i+1]$ to make sure that $S[j..i+1]$ is in the tree.
7: end for
8: end for

We have to specify what we mean by an extension in Algorithm 1. Let $S[j..i] = \beta$ be a suffix of $S[1..i]$. In extension $j$, when the algorithm finds the end of $\beta$ in the current tree, it extends $\beta$ to be sure the suffix $S[j..i+1]$ is in the tree, based on the following rules:

Rule 1: If path $\beta$ ends at a leaf in the current tree, add character $S[i+1]$ to the end of the label on that leaf edge.

Rule 2: If in the current tree, no path from the end of string $\beta$ starts with character $S[j..i]$, but at least one labeled path continues from the end of $\beta$, then create a new edge and leaf and label the edge with $S[j..i]$ and the leaf with number $j$. If $\beta$ ends inside an edge, then a new node should also be created where $\beta$ ends.

Rule 3: If there is some path starting from the end of $\beta$ with character $S[i+1]$, we do not need to do anything since $S[1..i+1]$ is already in the tree.

We can see an example in Figure 4.2. When we add character $\delta$ to the string $S_1 = \text{node}$, the first four suffixes get extended using Rule 1, the fifth suffix extends using rule 2 and the sixth suffix using rule 3.

Based on the above algorithm and naive implementations, Ukkonen algorithm takes $O(n^2)$ time. We need to find the end of a suffix $\beta$ of $S[1..j]$ ($\Omega(|\beta|)$), therefore extension $j$ of phase $i+1$ takes $O(i+1 - j)$ and $E_{i+1}$ is created in $O(j^2)$ and $I_k$ can be created in $O(n^2)$. We next show how this time can be reduced to $O(n)$.

4.1.2 Speedup Technique, Part 1

Definition Let $x$ denote an arbitrary string, where $x$ denotes a single character and $\alpha x$ denotes a (possibly empty) substring. For an internal node $v$ with path label $v_0x$, if there is another node $s(v)$ with path-label $\alpha$, then a pointer from $v$ to $s(v)$ is called a suffix link.
As a special case if α is empty, then the suffix link from an internal node with path-label α goes to the root.

Lemma 1: If a new internal node v with path-label α is added to the current tree in extension j of some phase i + 1, then either the path labeled α already ends at an internal node of the current tree or an internal node at the end of string α will be created in extension j = k in the same phase i + 1.

A corollary to this lemma states that any newly created internal node will have a suffix link from it by the end of the next extension. Also, in any implicit suffix tree T, if internal node v has path-label α, then there is a node s(α) of T with path-label α.

The end of the full string S[1..l] must end at a leaf of T, since $S[1..l]$ is the longest string represented in that tree. So as the trees are constructed, we keep a pointer to the leaf corresponding to the current full string $S[1..l]$ and it will be easy to find the end of the suffix $S[l..]$. Therefore extension 1 of phase i + 1 will be trivially done in constant time.

Having the above facts in mind, we can introduce the following algorithm for extension $j \geq 2$ of phase $i + 1$:

A simple note is worth mentioning that the first extension of phase $i + 1$, always applies suffix extension Rule 1. Figure 4.3 shows a general idea of how extension $j \geq 2$ of phase $i + 1$ works.

Now, we can introduce a technique that will reduce the worst case running time of the algorithm to $O(n^2)$.
Algorithm 2 Single Extension Algorithm (SEA)

1. Find the first node $v$ at or above the end of $S[j - 1, \gamma]$ that either has a suffix link from it or is in the root. This requires walking up at most one edge from the end of $S[j - 1, \gamma]$ in the current tree. Let $\gamma$ denote the (possibly empty) string between $v$ and the end of $S[j - 1, \gamma]$.
2. If $v$ is not the root, traverse the suffix link from $v$ to node $s(v)$ and then walk down from $s(v)$ following the path for string $\gamma$. If $v$ is the root, then follow the path for $S[j - 1, \gamma]$ from the root.
3. Use the extension rules and make sure the string $S[j + 1, \gamma]$ is in the tree.
4. If a new internal node $w$ was created in extension $j - 1$ (by extension rule 2), then by Lemma 1, string $\alpha$ must end at node $s(w)$. Create the suffix link from $w$ to $s(v)$.

Figure 4.3: Extension $j \geq 2$ of phase $i + 1$ [14].
In step 2 of extension \( j = 1 \), the algorithm walks down from node \( s(v) \) along a path labeled \( \gamma \). This should take \( O(|\gamma|) \), however using the following technique, we can reduce the traversal time to \( O(\text{number of nodes on the path}) \).

**Technique 1 (Skip/Count Technique):** Let \( g \) denote the length of \( \gamma \) and recall that the first character of \( \gamma \) must appear as the first character on exactly one edge out of \( s(v) \). Let \( g \) denote the number of characters on that edge. If \( g \) is less than \( g \), then the algorithm does not need to look at any more of the characters on that edge and it simply skips to the node at the end of the edge. Then it sets \( g \) to \( g \) and looks over the outgoing edges to find the correct next edge. When an edge is reached where \( g \leq g \), then the algorithm skips to character \( g \) on the edge and quits, making sure that the \( \gamma \) path from \( s(v) \) ends on that edge exactly \( g \) characters down its label.

![Diagram](image)

**Figure 4.4: Skip/Count technique.** In phase \( i + 1 \), the substring \( \gamma \) has length \( 10 \). In order to find the end of a copy of \( \gamma \) starting from \( s(v) \), we need to skip four nodes and count three characters. [14].

**Theorem 1:** Using Technique 1, any phase of Ukkonen’s algorithm takes \( O(n) \) time. Therefore Ukkonen’s algorithm runs in \( O(n^2) \).
4.1.3 Speedup Technique, Part 2

One problem to proceed further and reduce the running time of Ukkonen's algorithm to \(O(n^3)\) is the fact that we record all the characters on the edges of the tree, the algorithm will require \(O(n^2)\) space and therefore \(O(n)\) running time will not be achievable. To overcome this problem, instead of recording characters, we label the edges by a pair of indices identifying the start and end indices of the substring on that edge. This way, only two numbers are written on any edge and since the number of edges is at most \(2n^2 - 1\), the tree will only use \(O(n)\) space.

Observation 1: In any phase, if suffix extension rule 3 applies in extension \(j\), it will also apply in all further extensions until the end of that phase. The reason is that when rule 3 applies, the path labeled \(S[j..j]\) in the current tree must continue with character \(S[j+1]\) and so the path labeled \(S[j+1..j]\) does also, and rule 3 again applies in the next extensions. It is also beneficial to observe that a new suffix link needs to be added to the tree only after an extension in which extension rule 2 applies. Now we can state the next trick.

Technique 2: End any phase \(i+1\) the first time that extension rule 3 applies. If this happens in extension \(j\), then there is no need to explicitly find the end of any string \(S[k..j]\) for \(k > j\). We call the extensions in phase \(i+1\) that are done after the first execution of rule 3, implicit extensions.

Observation 2: If at some point in Ukkonen's algorithm a leaf is created and labeled \(j\) (for the suffix starting at position \(j\) of \(S\)), then that leaf will remain a leaf in all successive trees created during the algorithm. The reason is that there is no mechanism in the algorithm to extend a leaf edge beyond its current leaf, i.e. when a leaf is labeled \(j\), extension rule 1 will always apply to extension \(j\) in any successive phase.

Let \(j\) denote the last extension in this sequence. Now we can present the last trick.

Technique 3: In phase \(i+1\), if a leaf edge is first created and would normally be labeled with substring \(S[p..p+i]\), instead of writing indices \((p, i+1)\) on the edge, write \((p, e)\), where \(e\) is a symbol denoting "the current end". Symbol \(e\) is a global index that is set to \(i+1\) once in each phase. In phase \(i+1\), since the algorithm knows that rule 1 will apply in extensions 1 through \(j\), at least, it need no additional explicit work to implement those \(j\) extensions. Instead, it only does constant work to increment variable \(e\), and then does explicit work for (some) extensions starting with extension \(j+1\).

Using techniques 2 and 3, explicit extensions in phase \(i+1\) using 2 are only required
from extension $j+1$ until the first extension where rule 3 applies (or until extension $i+1$ is done). All other extensions (before and after those explicit extensions) are done implicitly. Hence, we can implement phase $i+1$ as follows:

Algorithm 3 Single Phase Algorithm (SPA)

1. Increment index $i$ to $i+1$ (By Technique 3, this correctly implements all implicit extensions 1 through $j$).
2. Explicitly compute successive extensions (using algorithm 2) starting at $j+1$ until reaching the first extension $j$ where rule 3 applies or until all extensions are done in this phase (By Trick 2, this correctly implements all the additional implicit extensions $j+1$ through $i+1$).
3. To prepare for the next step, set $j+1$ to $j-1$.

Theorem 2: Using suffix links and tricks 1, 2 and 3, Ukkonen's algorithm builds implicit suffix trees $T$ through $T_i$ in $O(n)$ time.

In order to make the real suffix tree, add the terminal character $\$ \$ to the end of $S$ and let Ukkonen's algorithm continue. Now, no suffix will be a prefix of any other suffix, so the algorithm builds an implicit suffix tree where each suffix ends as a leaf. We only need to replace each occurrence of index $e$ on every leaf edge with the number $e$. We can do that in $O(n)$ time by traversing each leaf edge. Now we have a true suffix tree.

4.2 Finding Maximum-Weight Anchor Set

Given a set of matches, we would like to select a set of non-crossing matches, also referred to as anchors, which we can use to build global/local alignments. Here we describe the method presented in [21] that can select a maximum weight set of matches in $O(q \log q)$ where $q$ is the number of matches.

To describe this method, we need some definitions:

Definition 1: For sequences $S_1$ and $S_2$, define character $i$ of $S_1$ by the point $(0, i)$ and character $j$ of $S_2$ by the point $(j, 0)$ in the cartesian plane.

Based on this definition, a given match $M = (i_1, i_2, j_1, j_2)$ corresponding to a match between $S_1[i_1..i_2]$ and $S_2[j_1..j_2]$, defines points of a rectangle $r$ in the cartesian plane with the bottom left corner being the point $(i_1, j_1)$ and the top right corner being $(i_2, j_2)$. Define the weight of $r$ to be its area. The problem of finding a set of maximal matches reduces to finding a set of non-crossing rectangles with maximum area coverage. More formally:
Definition Let

- $X_{min}(r)$ = minimum $x$ coordinate of any point in $r$
- $X_{max}(r)$ = maximum $x$ coordinate of any point in $r$
- $Y_{min}(r)$ = minimum $y$ coordinate of any point in $r$
- $Y_{max}(r)$ = maximum $y$ coordinate of any point in $r$

Definition For two rectangles $r$ and $s$, we say $r$ precedes $s$ (denoted by $r \prec s$) if $X_{max}(r) < X_{min}(s)$ and $Y_{max}(r) < Y_{min}(s)$.

Now we build a directed graph with nodes representing rectangles (matrices) and there is an edge between nodes $r$ and $s$ iff $r$ precedes $s$.

Definition A path in a rectangle graph is an ordered set $p = (r_1, ..., r_m)$ of rectangles such that $r_i \prec r_{i+1}$ for $0 \leq i \leq m - 1$. $Head(p) = r_1$ is called the head of $p$ if $p$ is non-empty.

Weight($p$) denotes the sum of the weights of all rectangles in path $p$.

Definition For two paths $p$ and $q$, we say $q$ dominates $p$ (denoted by $p \preceq q$) iff Weight($p$) $\leq$ Weight($q$) and either Head($p$) = Head($q$) or Head($p$) $\prec$ Head($q$).

The intuition behind this definition is that $q$ can subsume $p$ in any path without decreasing its weight.

Definition If neither $p \preceq q$ nor $q \preceq p$ is true, we say $p$ and $q$ are compatible.

In order to find the maximum weight set of non-crossing rectangles, we sort the rectangles based on their $X_{max}$ and we sweep them from right to left. At each stage, when we are processing rectangle $r$, we want to link it to the maximum weight path in the interval $[X_{max}(r), +\infty) \times [Y_{min}(r), +\infty)$ and store it in a set $D$. We use Next($i$) to denote the next rectangle in the best path containing rectangle $i$ (the path with maximum weight).

The operations needed to build and manipulate set $D$ are Update and Best. The operation $Best(D, i)$ returns Head($i$) of a rectangle with minimum $Y_{min}(i) \geq y$ and returns null if no such rectangle exists. $Update(D, i)$ updates the set $D$ as follows. It adds the best path starting at rectangle $i$ to $D$, but it preserves the compatibility among members of $D$. In other words, for any two paths $p$ and $q$ such that $p \preceq q$ in set $D$, we would like to remove $p$. 
Algorithm 4 Finding Maximum Weight Anchors

1. Sort all $X_{min}$ and $X_{max}$ and put them in the array $Sort(1, 2n)$
2. $TotalWeight[0] = 0$
3. $D = \emptyset$
4. for $j := 2n$ down to 1 do
5. if $Sort[j] = X_{max}[i]$ for some $i$ then
6. $Next[i] := \text{Best}(D, Y_{max}(i))$
7. $TotalWeight[i] := TotalWeight(Next[i]) + Weight(i)$
8. else
9. $Next[i] = X_{min}[j]$ for some $i$
10. $Update(D, i)$
11. end if
12. end for

Now we can formally describe this process in Algorithm 4.

In order to achieve a running time of $O(\log n)$, we need to efficiently implement set $D$ so that the Best and Update operations take $O(\log n)$ time. That means we need to be able to search, insert, join and split in $O(\log n)$ time. We can accomplish that using a balanced search tree. Since each path $i$ can be uniquely identified by its $Head(i)$ (and the rest of the path can be constructed using $Next[i]$ pointers), each element in $D$ can be represented by a number which is the rectangle number that it corresponds to. Each element of $D$ has a $TotalWeight$ and a $Ymin$ associated with it. However, since all elements of $D$ are mutually compatible and become in the forest, when we scan the elements based on their $X$-coordinates from right to left, the order of accessing $TotalWeight$ is the same as the order of descending $Ymin$. Therefore, although we have two keys associated with each element (i.e. $TotalWeight$ and $Ymin$), if we sort the elements in the tree on one key, they will be sorted on the other key in the opposite order.
In this chapter we will describe GAUNA (Global Alignment Using Non-exact Anchors) (developed at Simon Fraser University by Alireza Khosravi-Shirazi al. [4]) and show its performance compared to other state-of-the-art algorithms. In order to do this, we will first give a high-level overview of how GAUNA works and then we will explain every part of the algorithm in more detail. GAUNA is the basis for IGAUNA and therefore understanding how it works is essential to understanding IGAUNA.

5.1 Method description

GAUNA is a recursive algorithm based on the following three main steps:

- Finding maximal matches;
- Selecting anchors;
- Closing the inter-anchor regions

To determine whether exact or inexact anchors should be used, GAUNA finds exact anchors of length at least 30 in a preprocessing step. If these anchors cover more than 50% of the input sequences then exact anchors are used in the recursions otherwise inexact anchors are used. Algorithm 5 gives a high-level description of the alignment algorithm. In the following sections, we will describe the main steps in more detail (the sub-continuous-FinalMaximal is explained in Section 5.2).
**Algorithm 5 GAUSA High Level**

1. GAUSA(sequence $S_1$, sequence $S_2$, array $K[1..l]$, index $i$, matchSet $M$, node suffix-Tree$\text{Root}$)

   - {$K[1..l]$}: a list of length thresholds for anchors in different recursions; $K[i] > K[i-1] > ... > K[1]$}
   - {i: the current index for $K$}

2. if $(|S_1|, |S_2|)$ is sufficiently small then

3. Align $S_1$ and $S_2$ using Needleman-Wunsch algorithm

4. return

5. end if

6. if $i > 0$ then

7. return

   - {Leave $S_1$ and $S_2$ unaligned}

8. end if

9. Call FindMaximal ($S_1$, $S_2$, $K[i]$, $M$, suffix$\text{Tree}\text{Root}$) to find maximal matches between $S_1$ and $S_2$

10. Select a subset of anchors with maximum total weight and put them in the final alignment of $S_1$ and $S_2$

11. for each pair of inter-anchor sequences, $S_1'$ and $S_2'$ do

12. Call GAUSA($S_1'$, $S_2'$, $K$, $i + 1$, $M$, suffix$\text{Tree}\text{Root}$) to align $S_1'$ and $S_2'$

13. end for

14. return
5.2 Finding Maximal Inexact Matches

Given two sequences \( S_1 \) and \( S_2 \), a quadruple \((i_1, i_2, j_1, j_2)\) is called a match if the optimal alignment score of the two subsequence \( S_1[i_1:i_1+|i_1-1]\) and \( S_2[j_1:j_2+|j_2-1]\) is greater than or equal to a certain threshold. Note that in the sequel \( S[i:j] \) denotes the subsequence of the sequence \( S \) starting at position \( i \) and ending at position \( j \). If \( S_1[i_1:i_1+|i_1-1]=S_2[j_1:j_2+|j_2-1]\) the match is called an exact match, otherwise it is called an inexact match.

Following the definition of Delcher et al., a match \((i_1, i_2, j_1, j_2)\) is called maximal if it cannot be extended at either endpoint [13]. For inexact anchors we generalize this definition as follows: An exact match \((i_1, i_2, j_1, j_2)\) is maximal if there is no other match \((l_1', l_2', r_1', r_2')\) such that \( S_1[i_1:i_1+|i_1-1]=S_2[j_1:j_1+|j_1-1]\) and \( S_2[r_1':r_1'+|r_1'-1]=S_2[l_2':l_2'+|l_2'-1]\). We will only consider inexact matches for which \( l_2 = r_2 \) and therefore our matches will be represented by a triple \((i_1, i_2, r)\).

As in other anchor based methods, GAUINA uses suffix trees to find anchors. In Section 4.4 we have described in detail how to build a suffix tree in linear time and space.

For a sequence \( S \), the salient feature we need of a suffix tree for \( S \) is that the concatenation of edge-labels on the path from the root to an internal node is a repeat subsequence in \( S \) where the number of repeats corresponding to an internal node is equal to the number of leaves of the subtree rooted at that internal node.

Let \( S_1 \) and \( S_2 \) be the two input sequences to our algorithm of length \( m \) and \( n \) respectively. We build a suffix tree for \( S_1 \) and then search for submatches of \( S_2 \) over this suffix tree. We wish to find all maximal matches of \( S_2 \) and \( S_1 \) that have alignment score above a threshold \( s \). The naive method is to find, for each subsequence \( S_2' \) of \( S_2 \), those paths of the suffix tree starting at the root whose label, matched with \( S_2' \), has an alignment score greater than \( s \). Notice there can be a large number of such paths and even the most efficient algorithms known for this problem have very high running time and space requirements making them impractical [30].

To overcome this problem, we must consider the structure of input sequences. When the input sequences are very similar, there can be up to \( O(nm) \) maximal matches (when we have short matches in one sequence repeated many times in the other sequence) while the number of the anchors is at most \( O(\min\{m, n\}) \) (since anchors are non-crossing matches, the number of anchors is at most equal to the number of characters in the shortest sequence).

In this case most of the maximal matches are discarded during the anchor selection phase.
Thus it suffices to find a small number of maximal matches; we choose these with highest score. On the other hand if the input sequences are dissimilar, the number of maximal matches is usually close to the number of final anchors in which case our algorithm will find most of the maximal matches.

Define the similarity value between two subsequence X and Y, \( s(X, Y) \), as the score of the optimal alignment divided by the length of the subsequence (we only consider matches with subsequence of the same length).

Our suffix tree search algorithm works as follows. Let \( T \) be the suffix tree of \( S_1 \). A location in \( T \) is either a node of \( T \) or a point on an edge of \( T \) that splits the label of the edge into two subsequences. For each suffix \( S_t^r \) of \( S_2 \), we find all locations \( p \) in \( T \) such that the label of the path from the root of \( T \) to \( p \) has a high similarity value with some prefix of \( S_t^r \). Algorithm 6 depicts our suffix tree search method for a suffix \( S_t^r \) of \( S_2 \).

Let \( S_t^r = S_2[r, n] \) be a suffix of \( S_2 \) and let \( P \) be the set of locations in \( T \) returned by algorithm 2 for \( S_t^r \). We now find the set of exact matches between the prefixes of \( S_t^r \) and subsequences of \( S_1 \). For each location \( p \) in \( P \), let \( Y \) be the label of the path from root of \( T \) to \( p \). Let \( R_i \) be the set of occurrences of \( Y \) in \( S_i \). Notice that occurrences of \( Y \) in \( S_i \) correspond to the labels of the paths from the root to \( p \) to a leaf of the subtree rooted at \( p \). Therefore \( R_i \) can be computed efficiently by traversing the subtree of \( T \) rooted at \( p \). Once the set \( R_i \) is computed, for each subsequence \( S_i[r_j, r_j + |Y| - 1] \) in \( R \), the algorithm outputs the \( (r_j, r_j, |Y|) \) as an exact match (see Figure 5.2).

This procedure significantly reduces search time by pruning the search space. However, computing the optimal alignment between the \( Y \) and \( S_t^r \) using Needleman-Wunsch requires quadratic time and too slow for our purposes. To overcome this, the dynamic programming is limited to a band of width \( d \) around the main diagonal (Figure 5.1) thus reducing running time to \( O(d|Y|) \). Notice that matches found this way do not contain long sequences of indels; long sequences of indels in the conserved regions will only be detected when we close gaps between the anchors. Algorithm 7 describes the maximal match-finding method.

After finding exact matches, the algorithm next identifies maximal matches. To do so, we sort the matches with respect to their locations in one of the sequences, detect the non-maximal matches, and remove them from the set of matches.
Algorithm 6 Searching On The Suffix Tree

1: SearchTree(node v of T, rectangle $S_p$, set $P$, table $H$, int $k$)
   
   \{ $P$: set of locations in \( T \) \}
   \{ $H$: length threshold for matches \}
2: Let \( Z \) be the label of the path from root of \( T \) to \( v \).
3: Let \( H \) be the dynamic programming table for the alignment between \( Z \) and \( S_p \), where
   \( S_p \) is the prefix of \( S \) of length \( |Z| \).
4: Let \( C \) be a set for candidate matches
5: for nodes \( u_1, u_2, \ldots, u_l \) children of \( v \) do
6:   Let \( Y \) be the label of the path from root to \( u \).
7:   Use \( H \) to compute the alignment of prefixes of \( S_p \) and \( Y \).
8:   Let \( Y \) be the longest prefix of \( Y \) such that for \( S_p \) the prefix of \( S \) of length \( |Y| \),
   \( d(Y, S_p) \geq d \).
9:   Let \( p \) be the position in the tree with label \( Y \).
10: if \( |Y| > \sqrt{d} \) and \( |v| \geq \sqrt{k} \) then
11:   Add \( p \) to \( C \).
12: end if
13: end for
14: if \( C \neq \emptyset \) then
15:   \{ Meaning node \( v \) was not extended \}
16:   Add \( v \) to set \( P \) (i.e. so that matches corresponding to \( v \) can be extended later) \}
17: else
18:   break.
19: end if
20: Let \( Y \) be the longest sequence amongst \( Y_1, Y_2, \ldots, Y_n \).
21: if \( Y \) \( = \) \( Y_n \) then
22: \{ Meaning that when we extended, we reached the next node \}
23: Remove \( v \) from \( T \) and add \( T \) to \( P \) for later match extensions
24: Let \( H' \) be the dynamic programming table for the alignment between \( S_p \) and \( Y_n \).
25: Call SearchTree(\( u_n, S_p', P, H', k \))
26: else
27: \{ We did not reach a next node when extending \}
28: Add \( T \) to \( P \)
29: end if
30: end if
Figure 5.1: To make the DP routine faster, only the area shown will be covered. This results in shorter sequences of indels in the matches.

Algorithm 7 Finding Maximal Matches

1. FindMaximalMatches (sequence $S_1$, sequence $S_2$, int $k$, set $M$, node suffixTreeRoot)
   
   \{ $M$ will hold matches \}

2. $P = \emptyset$

3. for ∀ suffixes $S_2$ of $S_2$ do

4. Call SearchTree (suffixTreeRoot, $S_2$, $P$, $\emptyset$, $k$)

5. for ∀ $p \in P$ do

6. Find occurrences of $p$ in $S_1$ and add them to $M$

7. end for

8. end for

9. Remove redundant matches
Figure 5.2: In this figure $p_i$ is one of the locations in the suffix tree that is returned by Algorithm 6 for the suffix $p_i$ of $S_i$. $R_i$ is the set of occurrences of $Y_i$ in $S_i$ and $M_i$ is the set of maximal matches corresponding to the location $p_i$. 
5.3 Selecting Anchors

Let $M = \{m_1, m_2, \ldots, m_q\}$ be the set of maximal matches. For each $m_i$ in $M$, we define its weight $w(m_i)$ to be the product of its length and similarity value. Moreover, for two distinct $m_i = (r_1, r_2, l)$ and $m_j = (r_1', r_2', l')$ in the set $M$, we define $m_i < m_j$ if and only if the two inequalities $r_1 + l < r_1'$ and $r_2 + l < r_2'$ hold.

5.3.1 Finding Largest Total Weight Non-crossing Anchors

The next step is to select a set of anchors that has the largest total weight. An anchor set is a collection of non-crossing and non-overlapping maximal matches (see Figure 5.3).

One way to solve the problem of finding a set of anchors with largest total weight is to model it as a graph theoretic problem [17]. Construct a weighted directed graph $G = (V, E)$ from the set $M$ as follows: for each $m_i$ in $M$ there is a vertex $v_i$ in $V$. Moreover, $V$ includes two special vertices start and stop. The set of edges $E$ is defined as follows: For each $j \in [1, q]$ there is an edge $\text{start} \rightarrow v_j$ with weight 0, for $m_i < m_j$ there is an edge $v_i \rightarrow v_j$ with weight $w(m_i)$, and for each $i \in [1, q]$ there is an edge $v_i \rightarrow \text{stop}$ with weight $w(m_i)$. It takes $O(q^2)$ time to construct the graph. A maximum weight set of anchors corresponds to a path with maximum weight from start to stop in $G$. Since $G$ is acyclic, such a path can be computed in $O(|V| + |E|)$. Hence a set of anchors with maximum weight can be computed in $O(q^2)$ time since $O(|E|) = O(|V|^2)$ and $|V| + |E| = O(q^2)$.

However, in our implementation we used a more complicated method in terms of implementation, but more efficient with time complexity $O(\log q)$ [21]. We have described this algorithm in detail in Section 4.2.

Figure 5.3: A set of anchors is depicted. Rectangles represent the maximal matches and a set of good anchors is depicted in white rectangles connected by dashed lines.
5.4 Closing The Gaps

The last step in our algorithm is to close the gaps between the anchors. We align each gap as follows: If the gap is sufficiently small (less than a user-specified threshold), we align it using the Needleman-Wunsch method. Otherwise there are two cases: If the minimum length threshold $k$ for the anchors is larger than a certain threshold, then we align the gap by recursively calling the algorithm with a decreased value for $k$. However if $k$ is smaller than the threshold, then we have failed to find any match of significant size and there are no significant similarities to be discovered in such a gap. In this case, we divide the gap into short pieces of equal length, align the pieces separately using the Needleman-Wunsch algorithm, and concatenate them to get an alignment of the entire gap.

5.5 GAUNA Parameters

The main parameters of GAUNA are as follows:

1. A list of positive integers $K = \{k_1, k_2, \ldots, k_l\}$, which represents the minimum length thresholds for the maximal matches in different recursive steps of the algorithm.

2. An inter-anchor length threshold $\varepsilon$. If the product of the lengths of the subsequences corresponding to a region between two anchors is no more than $\varepsilon$ then the region is aligned using the Needleman-Wunsch method.

3. A similarity threshold $s$ for anchors. GAUNA uses anchors with similarity value of $s$ or higher.

4. Width $d$ of the band around the main diagonal of the dynamic programming table. To align the subsequence of $S_y$ with the path labels of the suffix tree, the suffix tree search algorithm reuses the dynamic programming table to a band of width $d$ around the main diagonal.
Chapter 6

GAUNA Results

In this chapter we will show the results obtained from GAUNA and will compare them to some of the best other tools available.

6.1 Exact vs. Inexact Anchors, Specificity Evaluation

To justify the use of inexact matches over exact matches, in this section we investigate the effect of using inexact matches on the specificity of anchors. The specificity of the anchors is defined as the percentage of the total length of anchors that hit the coding regions. An anchor is said to hit a coding region if more than half of its length overlaps with the coding region.

We ran GAUNA with both exact and inexact anchors with $K = \{20, 10, 5\}$ for exact and $K = \{25, 10, 7\}$ for inexact anchors. Our results in each case are summarized in Table 6.1. Notice that the specificity is increased by 40% on average when inexact anchors are used.

6.2 GAUNA Parameter Settings and Results

We evaluate the performance of GAUNA in two ways. First, we compare GAUNA with the state-of-the-art global alignment algorithms mentioned earlier. The performance is measured by the ability of each program to correctly align the coding regions in the sequence. Second, we evaluate the effect of using inexact anchors in terms of specificity.
Table 6.1: GAUNA Specificity

<table>
<thead>
<tr>
<th>Sequences</th>
<th>Authors</th>
<th>Specificity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse/Dog</td>
<td>exact</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>14</td>
</tr>
<tr>
<td>Mouse/Chicken</td>
<td>exact</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>13</td>
</tr>
<tr>
<td>Human/Chicken</td>
<td>exact</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>30</td>
</tr>
<tr>
<td>Human/Mouse</td>
<td>exact</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>15</td>
</tr>
<tr>
<td>Human/Rat</td>
<td>exact</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>7</td>
</tr>
<tr>
<td>Human/Chimp</td>
<td>exact</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>2</td>
</tr>
<tr>
<td>Human/Dog</td>
<td>exact</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>inexact</td>
<td>8</td>
</tr>
</tbody>
</table>

Specificity comparisons for exact versus inexact anchors.

Our test set includes the syntenic sequences from the human, chicken, chimp, pig, and rat. Syntenic sequences are those that share some similar genes with each other. The sequences have been obtained from Ensembl genome browser.

The sequences along with the corresponding alignments produced by all the programs considered in our test process are available at [http://www.plms.math.cn/gauna/](http://www.plms.math.cn/gauna/).

For comparison purposes all programs were run on a Linux machine with a 3.4 GHz intel(R) Xeon(TM) processor and 2 GB of RAM.

We note that for alignment of human and chimp, GAUNA chose exact anchors as those covered more than 50% of the sequences. In the other cases the parameter settings were: 
\[ K = (25, 10, 7), \quad \varepsilon = 1500^2, \quad d = 0.8, \text{ and } \omega = 7 \] (a description of each parameter can be found in Section 3.5).
6.2.1 Experimental Results

Our comparisons of GAUNA, LAGAN, AVID, MUMmer, and MGA are summarized in Table 6.2. We tested the other tools (where possible), to find the parameters that maximize their performance.

To measure the quality of the alignments, we find the alignment regions that have a high alignment score and cover more than 10% of an exon. The total length of such regions determines the quality of an alignment and is shown under the Coverage column in Table 6.2.

To handle regions of different length, we define the normalized score for an alignment region to be $s$, where $s$ is the score of the alignment, $l$ is the length of the alignment, and $e$ is the maximum value in the scoring matrix (note that normalized score is always less than 1). A region that has a normalized score above 0.8 is considered as a high alignment score region.

Table 6.2 also shows the Sensitivity of the alignments. The sensitivity of an alignment is defined as the percentage of the coding regions that is covered by high score alignment regions. Our results indicate a dramatic improvement over existing aligning algorithms. In particular, when time, space, and coverage are considered in aggregate, it is clear that GAUNA is the best performing of the current whole genome sequence alignment algorithms. For coverage GAUNA is always within 1% of the best known coverage taking all other algorithms into account (in three cases it actually exhibits better coverage than the others). Even when it has slightly worse coverage, time and space considerations more than compensate. In the worst case for example, when human and mouse were compared, GAUNA used about half the time and space used by LAGAN.
### Table 6.2: GAUNA Global Alignment Results

<table>
<thead>
<tr>
<th>Sequences</th>
<th>Length</th>
<th>Program</th>
<th>Coverage</th>
<th>Sensitivity</th>
<th>Time</th>
<th>Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mouse</td>
<td>4772141</td>
<td>GAUNA</td>
<td>12532</td>
<td>26</td>
<td>103</td>
<td>89</td>
</tr>
<tr>
<td>Dog</td>
<td>2399566</td>
<td>LAGAN</td>
<td>12139</td>
<td>20</td>
<td>189</td>
<td>223</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>11780</td>
<td>24</td>
<td>54</td>
<td>560</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>5922</td>
<td>12</td>
<td>103</td>
<td>191</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>10729</td>
<td>22</td>
<td>182</td>
<td>41</td>
</tr>
<tr>
<td>Mouse</td>
<td>3230581</td>
<td>GAUNA</td>
<td>9125</td>
<td>7</td>
<td>76</td>
<td>130</td>
</tr>
<tr>
<td>Chicken</td>
<td>1214554</td>
<td>LAGAN</td>
<td>8910</td>
<td>7</td>
<td>275</td>
<td>259</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>8505</td>
<td>7</td>
<td>80</td>
<td>642</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>2621</td>
<td>2</td>
<td>38</td>
<td>182</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>5376</td>
<td>4</td>
<td>231</td>
<td>41</td>
</tr>
<tr>
<td>Human</td>
<td>4688090</td>
<td>GAUNA</td>
<td>57451</td>
<td>14</td>
<td>419</td>
<td>254</td>
</tr>
<tr>
<td>Dog</td>
<td>6124516</td>
<td>LAGAN</td>
<td>57045</td>
<td>14</td>
<td>509</td>
<td>261</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>57658</td>
<td>14</td>
<td>188</td>
<td>1822</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>39628</td>
<td>9</td>
<td>459</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>41163</td>
<td>10</td>
<td>765</td>
<td>1151</td>
</tr>
<tr>
<td>Human</td>
<td>1779863</td>
<td>GAUNA</td>
<td>22280</td>
<td>31</td>
<td>70</td>
<td>85</td>
</tr>
<tr>
<td>Mouse</td>
<td>1775382</td>
<td>LAGAN</td>
<td>22086</td>
<td>32</td>
<td>127</td>
<td>208</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>16627</td>
<td>27</td>
<td>53</td>
<td>498</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>17472</td>
<td>24</td>
<td>433</td>
<td>887</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>69118</td>
<td>24</td>
<td>205</td>
<td>32</td>
</tr>
<tr>
<td>Human</td>
<td>6273849</td>
<td>GAUNA</td>
<td>22185</td>
<td>21</td>
<td>368</td>
<td>271</td>
</tr>
<tr>
<td>Rat</td>
<td>7336917</td>
<td>LAGAN</td>
<td>22173</td>
<td>21</td>
<td>647</td>
<td>704</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>10777</td>
<td>10</td>
<td>210</td>
<td>1907</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>13703</td>
<td>12</td>
<td>291</td>
<td>416</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>14166</td>
<td>13</td>
<td>780</td>
<td>139</td>
</tr>
<tr>
<td>Human</td>
<td>4887765</td>
<td>GAUNA</td>
<td>129896</td>
<td>83</td>
<td>42</td>
<td>186</td>
</tr>
<tr>
<td>Chicken</td>
<td>4944871</td>
<td>LAGAN</td>
<td>128641</td>
<td>82</td>
<td>900</td>
<td>322</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>124345</td>
<td>82</td>
<td>78</td>
<td>1146</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>61252</td>
<td>39</td>
<td>168</td>
<td>407</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>10292</td>
<td>64</td>
<td>55</td>
<td>186</td>
</tr>
<tr>
<td>Human</td>
<td>2933747</td>
<td>GAUNA</td>
<td>31503</td>
<td>14</td>
<td>55</td>
<td>124</td>
</tr>
<tr>
<td>Chicken</td>
<td>1162765</td>
<td>LAGAN</td>
<td>32009</td>
<td>15</td>
<td>206</td>
<td>272</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AVID</td>
<td>31301</td>
<td>14</td>
<td>55</td>
<td>587</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MUMmer</td>
<td>5557</td>
<td>3</td>
<td>72</td>
<td>186</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MGA</td>
<td>24172</td>
<td>10</td>
<td>294</td>
<td>37</td>
</tr>
</tbody>
</table>

The results of global alignments for different programs. Time is in seconds and memory is in megabytes.
Chapter 7

IGAUNA

In this chapter we will introduce the improvements to GAUNA which resulted in IGAUNA (Improved Global Alignment Using Noireceur Anchors). However in order to properly justify some of the changes made, we will discuss methods to identify a good alignment and later will introduce the concept of an optimal alignment. Then we will show the results obtained from IGAUNA and will compare these with results from other global alignment tools. At the end we will have a conclusion and suggest a direction for future work.

7.1 Measuring An Alignment

As described in Section 3.1, there are many different ways to design a scoring function and these have a direct impact on the output of an alignment algorithm. The methods used in most of today’s tools are based on score matrices that give a pairwise score for each of the possible allowable pairwise matches/mismatches. Determining how good a scoring method or an alignment algorithm is, depends on the quality of the output and similar to what we described above, there are different methods for determining how good a given alignment is.

The simplest way is to only look at the scoring function and base our judgement on the final score of the alignment. Using this method, the best alignment is an alignment obtained from the Needleman-Wunsch DP method. Although this method is simple (but as we know not efficient) to apply, for the case of biological sequences, it does not provide us with much information about the alignment. So we need more sophisticated ways of
judging an alignment. As mentioned in Section 6.2.1, for GAUNA, we took a look at the so-called conserved regions and measure how much they cover the final alignment. We defined a conserved region to be an alignment region that has a high alignment score and covers more than 10% of an exon. The reason why exons are important, is that from the knowledge we have about DNA sequences so far, exons are the preserved (or conserved) regions of DNA that perform a similar function in a species. So we would like to identify them because the high-scoring regions that have an overlap with an already-known exon, show us that our alignment has identified exons correctly. In order to incorporate this idea, we have introduced a weight adjustment parameter for exons in GAUNA that we will describe more in Section 7.2.1. We also treat exons a bit more favorably as described in Section 7.2.2.

Having said that, we think exons are not the only important segments to be considered. There is still not much known about the introns and therefore if high scoring regions are found that do not overlap with any exon, we think they might still be of some importance. The biological motivation behind the importance of conserved regions (and therefore exons) is that if a region has changed little over time, it must be resistant to mutations and so there is a good chance that it has been of some importance for the life of the species.

Therefore, in GAUNA besides looking at the conserved regions overlapping exons, we also look at the overall coverage of conserved regions and our measurements are based on both these factors.

7.2 Improvements to GAUNA

The improvements made to GAUNA that have resulted in GAUNA can be categorized into three groups:

- Exon weight adjustment
- Branching
- Parameter optimization

We will explain them briefly here and then in more detail in the upcoming sections.

Since exons are of special interest to biologists and a measurement of a good alignment is how exons are aligned. We will try to give more weight to potential exons when finding matches.
7.2.2 Branching

Branching is an improvement we made when traversing the suffix tree and as the results will show, has a direct influence on the quality of matches and therefore the quality of the final alignment.

As mentioned earlier in Section 5.5, GAUNA has some parameters that should experimentally be set. With the introduction of Branching into GAUNA, there are more parameters introduced and there is a need to optimize these parameters. We have done extensive testing to find the best parameters suitable for the sequences we had access to, to ensure that we get the maximum performance quality while considering the running time.

7.2.1 Exon Weight Adjustment

Some alignment tools (such as LAGAN), use heuristics to better align biologically related regions. With a similar motivation in mind, in order to emphasize the importance of exons when building the alignment, we incorporated a weight adjustment method into GAUNA. Whenever we find a match that has an acceptable coverage with an exon, we increase its weight, so that it will be more desirable to be picked in the final anchor set when applying the maximum match finding routine described in Section 5.2.

However, we do not know where exons are located. To overcome this problem, we use an exon finding program called GeneScan (developed at Stanford University) [9, 10] that does a relatively good job in finding potential exons. GeneScan runs relatively fast and it does not require much memory. We introduce a new boolean parameter called USE_EXONS that when set true, causes GAUNA to run GeneScan before finding the matches/anchors. Then, when GAUNA finds a match that has an overlap with a potential exon found by GeneScan, it increases its weight by a factor proportional to how much it overlaps with that exon. We introduce a parameter called AMPLIFYING_RATIO that determines how much this weight adjustment should be.

7.2.2 Branching

As described in Section 5.2, as GAUNA tries to search the suffix tree to find matches, it traverses the tree along a path. When it reaches a node, there are different paths to consider, so GAUNA goes along each of these paths for a short distance and based on a score drop off threshold, chooses the best path and discards the others. Obviously, this greedy choice might not always give the best path. Every different branch of a tree that is traversed,
gives different matches in different locations on the search sequence. Therefore, choosing a different, non-optimal branch might result in losing some score-wise and quality-wise significant matches.

To deal with this problem we introduced branching. When branching is enabled, IGANNA does not merely take the best path candidate and extend along it. Instead, it will keep a set of all candidates and extends a predefined number of them. This way, we can be sure that we do not miss as many matches as before. We can also be sure that even if the greedy choice chosen before was not the optimal choice, other choices are tried as well. In order to control the behavior of IGANNA and avoid an exponential running time due to branching, we only use branching when we encounter a potential error (for the reasons described in Section 7.1). Otherwise, if branching is used on the whole sequence, IGANNA will take so much time to execute that it will be practically infeasible. We incorporated two other methods for controlling the behavior of branching as follows:

- The first method is to put a limit on the number of good candidates to consider. A good candidate is a candidate considered in line 9 of Algorithm 6. We have two bounds: one is the number of candidates to consider at every node called \textsc{MaxChildren}, \textsc{Extend}, the other is the number of total candidates that should be considered called \textsc{MaxPaths}. \textsc{Traversed} (every candidate branch selected to be traversed at a node adds one to the number of paths considered so far). When IGANNA reaches \textsc{MaxPaths.Traversed}, it continues working as \textsc{Gauna}, i.e. it only considers the best greedy choice at each branching node.

- The second method is to change the way branching is executed. When considering paths, one way is to take each path being considered and go as far as possible, i.e. a depth-first search, and the other way is to consider all candidates at each node and take them to the next node and then consider the new paths again. If we use depth-first search and we reach \textsc{MaxPaths.Traversed}, we will end up with fewer paths considered, but they will have potentially longer lengths, which translates to longer matches. However, if we use breadth-first search, in theory (depending on the settings), we reach \textsc{MaxPaths.Traversed} order and based on what was described about the behavior of \textsc{Gauna} after reaching \textsc{MaxPaths.Traversed}, we will end up with potentially now shorter matches and fewer longer ones.

Our other simple change that we made, was to sort the candidates by their score rather
than their similarity. This way, if we have two matches with the same similarity, the longer one is given priority. As we see in Section 8.2, this small change results in a higher performance.

Based on the above descriptions and what was mentioned in Section 7.2.1, Algorithms 5, 7 and 6 will change to Algorithms 8, 9 and 10 respectively:

Algorithm 8. IGAUNA High Level

1. exonList := GeneScan(S2) \{finding the potential set of exons in S2\}
2. IGAUNA(sequence S1, sequence S2, array K[1 \ldots \ell], index i, matchSet M, node suffix-TreeRoot, exon exonList) \{(K[1 \ldots \ell]: a list of length thresholds for anchors in different recursions: K[1] > K[2] > \ldots > K[i]\}\}
3. if (\|S1\|\|S2\|) is sufficiently small then
4. \{exonList is the set of potential exons\}
5. Align S1 and S2 using Needleman-Wunsch algorithm.
6. return
7. end if
8. if i > j then
9. return \{Leave S1 and S2 unaligned\}
10. end if
11. Call FindMaximal(S1, S2, K[i], M, suffixTreeRoot, exonList) .
12. Adjust match weights of matches in M based on exons in exonList and the parameter \textsc{AmplifyingRatio}.
13. Select a subset of anchors with maximum total weight and put them in the final alignment of S1 and S2.
14. for each pair of inter-anchor sequences, Sj and S\ell do
15. Call IGAUNA(Sj, S\ell, K, i + 1, M, suffixTreeRoot, exonList) to align Sj and S\ell.
16. end for
17. return

7.2.3 Parameter Optimization

As discussed in Section 5.5, the main parameters used in GAUNA (which are kept the same in IGAUNA as well) are K-values, \(z\), similarity threshold \(z\) and the diagonal width \(d\) in the DP table.

The inter-anchor length threshold \(z\) is a threshold that determines when to use Needleman-Wunsch algorithm in a region instead of finding matches. It is set such that the \(O(n^2)\)
Algorithm 9 Finding Maximal Matches

1: InitialMaximalMatches (sequence $S_1$, sequence $S_2$, list $M$, node suffixTreeRoot, 
   exon exonsList) \{ $M$ will hold matches \}
2: $P = \emptyset$
3: for each suffix $S_1$ of $S_2$ do
4:  
5:  
6:  if $S_1$ has an overlap with an exon in exonsList then
7:     Call SearchTreeBranching (suffixTreeRoot, $S_2$, $P$, $\emptyset$, $k$)
8:  else
9:     Call SearchTree (suffixTreeRoot, $S_2$, $P$, $\emptyset$, $k$)
10:  end if
11: for each $p \in P$ do
12:     Find occurrences of $p$ in $S_1$ and add them to $M$
13: end for
14: end for
15: Remove redundant matches

running time of Needleman-Wunsch does not affect the performance of IGAUNA and at the
same it should be big enough to justify the use of DP over the overhead required to use
suffix tree matches.

The similarity threshold $\kappa$ determines how similar the matches should be so that they
are considered as potential anchors. If it is set too low, the number of anchors found will
increase, however their significance will decrease.

Width $w$ determines how much of the dynamic table should be covered when extending
matches and helps us control the running time by limiting the space that DP covers. As
a side-effect, it also limits the alignment to be of a special type that does not allow more
than $w$ consecutive gaps in the matches. This is acceptable, because for matches containing
long consecutive gap intervals, we can view these as two separate matches and identify these
independently.

Based on our experiments, $K$-values greatly affect the speed and quality of the alignment
and can make a major difference in the quality of the solution. We performed extensive test-
ing to determine the quality of the alignment based on different sets of $K$-values. Intuitively,
the more levels there are, the longer IGAUNA takes to run. The bigger the $K$-value, the
longer the matches we find should be and therefore if we choose too large a $K$-value, we
might find just a few number of matches that do not cover much of the two sequences and
Algorithm 10 Searching On The Suffix Trie

1. Search($text$, $trie$, $pattern$, $suffix Trie$, $suffix text$, $pat$, $text$, $trie$, $trie$)

2. If $text$ is empty, return $true$.

3. Let $pat$ be the prefix of $pat$ and $text$ that matches $text$.

4. Call Algorithm 1 for $pat$ and $text$.

5. If $text$ is empty, return $true$.

6. If $text$ is empty, return $false$.

7. Call Algorithm 2 for $text$ and $trie$.

8. If $trie$ is empty, return $false$.

9. If $trie$ is empty, return $true$.

10. Call Algorithm 3 for $text$ and $trie$.

11. If $trie$ is empty, return $false$.

12. If $trie$ is empty, return $true$.

13. End if

14. End if

15. If $trie$ is empty, return $false$.

16. If $trie$ is empty, return $true$.

17. End if

18. End if

19. End if

20. End if

21. End if

22. End if

23. End if

24. End if

25. End if

26. End if

27. End if

28. End if

29. End if

30. End if

31. End if

32. End if

33. End if

34. End if

35. End if

36. End if

37. End if

38. End if

39. End if

40. End if

41. End if

42. End if

43. End if

44. End if
therefore it will be a waste of time. However, if the K-value is too small, there will be too many matches found and each match will have less meaningful significance.

7.3 Optimal Alignment

Based on the accepted methods used for measuring the quality of an alignment, it is possible to compare the output of different programs with each other. However, one question that comes up is the scope for improvement. To answer this question and also to have another benchmark to compare different alignments with each other, we propose a way to make an optimal alignment. We use IGAUNA but with some biased input information in order to accomplish this.

Since proper alignment of exons is something that many researchers consider to be important, similar to what we have done in IGAUNA, we would like to prioritize exons when building the alignment. However, we will use the exact positions for known exons and give an extra weight for matches that are found in these regions. We also use BLAST to find matches in exons, mostly because BLAST is a widely accepted tool in finding local alignments.

The remaining regions between exons, are aligned using regular IGAUNA routines, the way we have described them in the previous sections (as opposed to using BLAST). The final result is an alignment with emphasis on the alignment of exon regions only as we will see has a higher score when we consider conserved regions overlapping exons.

The advantage of having the so-called optimal alignment is that, it gives us an estimate of how much a regular alignment has the potential to be improved in terms of exon alignments. If the optimal alignment has a score much higher than a given alignment, we know that there is still much room for improvement in the tool that produced the alignment. However, if the given alignment scores very close to the optimal alignment, we know that the quality of the alignment cannot be improved much. In this case, our focus will be on improving the speed of the tool and how much space it uses. So basically if two different methods produce two alignments with scores very close to the optimal alignment, then the one which produces the alignment faster and with less memory, has the advantage over the other one.
7.4 IGAUNA Parameters

Using different parameters gives IGAUNA the flexibility to be used for different cases easily. By properly setting these parameters, IGAUNA can actually run exactly like GAUNA or it can be executed to find the optimal alignment. Besides the ones we have already described in the previous sections, some of the other important parameters and their meanings are described below:

- **BRANCHING** determines whether branching should be used or not.
- **USE.EXONS** determines whether matches found in exon regions should be treated differently or not (i.e., weight adjustment and branching should be performed or not), should be adjusted or not.
- **USE.BLAST** determines whether BLAST should be used to find matches in exon regions.
- **USE.GENESCAN** determines whether GenScan is used to find exons (otherwise real exon coordinates will be used). If only valid if USE.EXONS is set to true.
- **MAX.PATHS.TRAVERSED** sets the limit of new paths to be taken when branching. If the number of paths we have traversed reaches this limit, from that point on, only the best candidate at each branching node is considered and other choices are discarded.
- **MAX.CHILDREN.EXTEND** determines the maximum number of children at each node to be considered for branching.
- **GOOD.CHILD.BREADTH** is set to true if we would like to search the suffix tree based on a breadth-first search (to find matches), otherwise matches are found on a depth-first search basis. It is only valid if BRANCHING is true.
IGAUNA Results and Conclusion

In this final chapter, we will show results obtained from IGAUNA in comparison to other programs and the optimal alignment and will suggest future directions for further improvements.

8.1 Experimental Settings

As was the case for GAUNA, our test was based on systemic sequences from the human, chicken, chimp, pig, and rat obtained from Ensembl genome browser.

For comparison purposes all programs were run on a Linux machine with a 3.4 GHz intel(R) Xeon(TM) processor and 2 GB of RAM.

8.2 Parameter Settings

In this section, we will discuss how the parameters of IGAUNA have been chosen and then we will show the results obtained from IGAUNA based on these settings.

As described in Section 7.4, IGAUNA has a few parameters that enable it to run under different "modes"; it can run as IGAUNA, GAUNA, Optimal Finder, etc. Having these parameters in mind, to use IGAUNA in its normal mode, we set BRANCHING, USE.EXONS, USE.BLAST and USE.GENESCAN to true. To run IGAUNA as GAUNA, we simply set BRANCHING and USE.EXONS to false. In order to find the optimal alignment, we set BRANCHING, USE.EXONS and USE.BLAST to true and set USE.GENESCAN to false.
Since GeneScan is time consuming, if we are using repetitive sequences (which is usually the case, since we are aligning for example a human gene to different species to study them), we can run GeneScan once for every species that we are studying and simply use the exon coordinates reported by GeneScan when we run IGAUNA. In order to do this, we simply set USE.EXONS to true and USE.GENSCAN to false but will set the E-values in a way that the output of GeneScan is used to extract potential exon coordinates.

Based on our own judgement and similar settings in tools such as LAGAN, we tried a range of threshold values and found that a value of 0.8 for a similarity threshold \( x \) (described in Section 5.5) produces the best alignment when considering both the quality of the alignment and the running time.

Trying different values for AMPLIFYING.RATIO, we found out that the ratio 8 results in the best quality of alignments in terms of exon coverage. However, while it seems intuitive that increasing the AMPLIFYING.RATIO will result in more conserved region length, that is not the case. By setting AMPLIFYING.RATIO to a number greater than 1, we are emphasizing on the exons, however it might be the case that there are some conserved regions in the introns (maybe potential exons not discovered yet), and when we emphasize on exons, they will not be picked up as anchors. Therefore the overall length of conserved regions might decrease. As we will see, this is actually the case many of the times.

As described in Section 7.2.2, in IGAUNA we use scores rather than similarities (which was used in GAUNA) to prioritize good candidates at nodes in order to follow paths. This simple change, resulted in about 2% increase in the quality of alignments. By using scores rather than similarities, we are choosing longer matches over shorter ones with the same similarity. So the result we obtained, shows that having longer matches but slightly less similar than shorter matches result in better alignments.

For the scoring matrix, we used the same matrix used in LAGAN (both for aligning and for evaluation of the alignments). This matrix produced good results for IGAUNA, but another reason why we chose this matrix was that LAGAN had the closest results to GAUNA (as can be seen in Table 6.2) and we wanted to see how IGAUNA performs against LAGAN using the same scoring system that LAGAN uses.

### 8.2.1 K-value Sets

We tried different K-values to systematically study the effect of minimum match length in speed and quality of the program. For each of the 7 pairs of species being studies, we
Table 8.1. K-value Effect

<table>
<thead>
<tr>
<th>K-value Set</th>
<th>Sensitivity</th>
<th>Specificity</th>
<th>Conserved Region Cover</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(20, 7)</td>
<td>0.24183</td>
<td>0.121267</td>
<td>0.248913</td>
<td>764</td>
</tr>
<tr>
<td>(20, 10)</td>
<td>0.24183</td>
<td>0.127838</td>
<td>0.249415</td>
<td>276</td>
</tr>
<tr>
<td>(25, 10)</td>
<td>0.24183</td>
<td>0.127778</td>
<td>0.249468</td>
<td>291</td>
</tr>
<tr>
<td>(35, 15)</td>
<td>0.248590</td>
<td>0.128763</td>
<td>0.247123</td>
<td>274</td>
</tr>
<tr>
<td>(20, 10, 7)</td>
<td>0.24183</td>
<td>0.127766</td>
<td>0.249418</td>
<td>290</td>
</tr>
<tr>
<td>(25, 10, 7)</td>
<td>0.24183</td>
<td>0.128145</td>
<td>0.248945</td>
<td>290</td>
</tr>
<tr>
<td>(35, 20, 7)</td>
<td>0.241236</td>
<td>0.130655</td>
<td>0.247774</td>
<td>316</td>
</tr>
<tr>
<td>(50, 30, 12)</td>
<td>0.240004</td>
<td>0.128949</td>
<td>0.247734</td>
<td>329</td>
</tr>
<tr>
<td>(40, 20, 10, 7)</td>
<td>0.24183</td>
<td>0.128149</td>
<td>0.248946</td>
<td>348</td>
</tr>
<tr>
<td>(45, 25, 10, 7)</td>
<td>0.24183</td>
<td>0.128181</td>
<td>0.248945</td>
<td>365</td>
</tr>
<tr>
<td>(50, 30, 10, 7)</td>
<td>0.24183</td>
<td>0.128161</td>
<td>0.248645</td>
<td>354</td>
</tr>
</tbody>
</table>

K-value effect on IGAUNA:

Sensitivity = Number Of Exon Hits / Number Of Exons,
Specificity = Total Conserved Regions Length / Total Length
Conserved Region Cover = Conserved Region Hit / Total Conserved Region Number.

Conserved Region Hit increments by one for every time a conserved region is actually picked in the final calculations of conserved regions. Sometimes a conserved region might not be picked because it does not have enough overlap with an exon.

As can be seen from Table 8.1, changing K-value sets (within reasonable values) does not affect sensitivity and specificity significantly. However, the running time can jump at some points. The same pattern of jumps in time applies to other species as well, but with varying intensities from 1.5 to 4 times increase in time.

Table 8.1 reveals that since the running time does not change much (except for the jump points), to get the maximum sensitivity and specificity, we should choose the K-value set (25, 10, 7).

considered the following K-value sets: (20, 7), (20, 10), (25, 10), (25, 7), (30, 14), (30, 10, 7), (40, 10, 7), (45, 10, 7), (50, 30, 12), (40, 20, 10, 7), (45, 25, 10, 7), (50, 30, 10, 7). With every change in the set of K-values, IGAUNA’s performance consistently changed in all the species, therefore to show the changes, we will only show the results for one pair of species (Human-Dog alignment).

As can be seen from Table 8.1, changing K-value sets (within reasonable values) does not affect sensitivity and specificity significantly. However, the running time can jump at some points. The same pattern of jumps in time applies to other species as well, but with varying intensities from 1.5 to 4 times increase in time.

Looking at Table 8.1 reveals that since the running time does not change much (except for the jump points), to get the maximum sensitivity and specificity, we should choose the K-value set (25, 10, 7).
8.3 Alignment Results

In the following tables we show results of pairwise alignments for different species. Note that we do not include Human-Chimp, since IGAUNA uses exact matches for this alignment and the results are the same as for GAUNA shown earlier in Section 6.2.1. Also, unlike the tests for GAUNA, we are not including test results for MGA and MUMmer, since the results of these two programs either in terms of quality or memory/reusing time were not comparable to IGAUNA. Therefore we are only including the comparisons between Optimal, GAUNA, GAUNA, LAGAN and AVID. We have tested LAGAN for different parameters and used the ones that maximised its performance, however, AVID did not provide us the option to change its parameters.

In order to measure the quality of alignments, we use a similar idea described in Section 6.2.1, the concept of conserved region. A conserved region is a high-scoring region of an alignment with a similarity above 80%.

The terminology we have used in the tables and their meanings are as follows:

- **Total Conserved Length (TCL)** is the total length of conserved regions in the alignment.

- **Exon Coverage Length (ECL)** is the length of exons covered by conserved regions. For this calculation, we do not count every part of the exon that falls in a conserved region, rather if the summation of the regions within an exon that fall in conserved regions add up to more than 50% of the exon length, then we count those conserved regions in our summation.

- **Total Conserved Length in Exons (TCECL)** takes into account all parts of an exon covered by a conserved region.

- **Number of Exons Covered (NEC)** shows how many exons were covered more than 50% by the conserved regions.

We will now compare IGAUNA with other tools and with GAUNA.

8.3.1 Memory Usage

In terms of memory usage, IGAUNA works very similarly to GAUNA. Although GeneScan takes a lot of memory compared to IGAUNA (or GAUNA) on its own, by breaking up
Table 8.2: Mouse Dog Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE</th>
<th>Time (s)</th>
<th>Mem (mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>64011</td>
<td>2480</td>
<td>21</td>
<td>6232</td>
<td>266</td>
<td>91</td>
</tr>
<tr>
<td>IGAUNA</td>
<td>83415</td>
<td>2128</td>
<td>19</td>
<td>5999</td>
<td>178</td>
<td>91</td>
</tr>
<tr>
<td>GAUNA</td>
<td>71009</td>
<td>2308</td>
<td>19</td>
<td>5739</td>
<td>103</td>
<td>89</td>
</tr>
<tr>
<td>LAGAN</td>
<td>61930</td>
<td>2228</td>
<td>18</td>
<td>6390</td>
<td>189</td>
<td>223</td>
</tr>
<tr>
<td>AVID</td>
<td>72206</td>
<td>2291</td>
<td>19</td>
<td>4862</td>
<td>51</td>
<td>569</td>
</tr>
</tbody>
</table>

TCL: Total Conserved Length, ECL: Exon Coverage Length, NEC: Number of Exons Covered, TCLE: Total Conserved Length in Exons. The results of global alignment of Mouse (chr19, length=17421490)-Dog (chr19, length=2190565). Total Mouse Exon Length 31076, Total Dog Exon Length 17281.

Table 8.3: Mouse-Chicken Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE</th>
<th>Time (s)</th>
<th>Mem (mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>74824</td>
<td>7914</td>
<td>55</td>
<td>16730</td>
<td>235</td>
<td>150</td>
</tr>
<tr>
<td>IGAUNA</td>
<td>77465</td>
<td>6891</td>
<td>38</td>
<td>9770</td>
<td>259</td>
<td>142</td>
</tr>
<tr>
<td>GAUNA</td>
<td>73244</td>
<td>6623</td>
<td>36</td>
<td>9290</td>
<td>79</td>
<td>130</td>
</tr>
<tr>
<td>LAGAN</td>
<td>57639</td>
<td>6997</td>
<td>40</td>
<td>9227</td>
<td>285</td>
<td>259</td>
</tr>
<tr>
<td>AVID</td>
<td>104343</td>
<td>6313</td>
<td>36</td>
<td>8783</td>
<td>89</td>
<td>612</td>
</tr>
</tbody>
</table>

TCL: Total Conserved Length, ECL: Exon Coverage Length, NEC: Number of Exons Covered, TCLE: Total Conserved Length in Exons. The results of global alignment of Mouse (chr19, length=3226881)-Chicken (chr23, length=1248453). Total Mouse Exon Length=76866, Total Chicken Exon Length=93194.

The sequences and feeding them to GeneScan, we can solve that problem. Figure 8.1 shows memory usage of different tools for different sequences in a more precise form.

As we can see from the results, IGAUNA is the best tool in terms of memory usage after GeneScan and the gap between these and other tools is significant. This is one of the major advantages of IGAUNA over other programs since it can be executed on an ordinary computer for very long sequences whereas other tools are simply incapable of performing such a task (or if they can run by using secondary memory, the speed will be greatly impacted).
In terms of speed, the overhead that GeneScan causes, is the biggest factor in reducing IGAUNA’s speed compared to GAUNA. However, it is possible to extract the names by GeneScan as a pre-processing step and use those to feed IGAUNA (the parameters of IGAUNA allow this conveniently). Branching has an influence on the speed as well, but depending on the case, we can limit it by choosing suitable values for the corresponding parameters. Overall, although branching does reduce the speed, it does not reduce it much. As we can see from Tables 8.2 to 8.7, IGAUNA works slower than GAUNA and AVID, but is faster than LAGAN (and from MUMmer as shown in Table 6.2). However with the

### Table 8.4: Human Dog Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE</th>
<th>Time(s)</th>
<th>Mem (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>921101</td>
<td>39874</td>
<td>260</td>
<td>52198</td>
<td>580</td>
<td>274</td>
</tr>
<tr>
<td>IGAUNA</td>
<td>1023041</td>
<td>38067</td>
<td>260</td>
<td>51191</td>
<td>1995 (584s by GeneScan)</td>
<td>270</td>
</tr>
<tr>
<td>GAUNA</td>
<td>981271</td>
<td>39974</td>
<td>260</td>
<td>50117</td>
<td></td>
<td>319</td>
</tr>
<tr>
<td>LAGAN</td>
<td>1084915</td>
<td>47242</td>
<td>328</td>
<td>63174</td>
<td></td>
<td>790</td>
</tr>
<tr>
<td>AVID</td>
<td>1106773</td>
<td>39638</td>
<td>260</td>
<td>51001</td>
<td></td>
<td>1898</td>
</tr>
</tbody>
</table>

| TCEL: Total Conserved Length | ECL: Exon Coverage Length | NEC: Number of Exons Covered | TCLE: Total Conserved Length in Exons |

The results of global alignment of Human (chromosome1, length=6490900)-Dog (chromosome5, length=62246515). Total Human Exon Length=329877, Total Dog Exon Length=74268.

### Table 8.5: Human Chicken Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE</th>
<th>Time(s)</th>
<th>Mem (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>100972</td>
<td>24500</td>
<td>192</td>
<td>35508</td>
<td>101</td>
<td>134</td>
</tr>
<tr>
<td>IGAUNA</td>
<td>1084221</td>
<td>23200</td>
<td>199</td>
<td>34100</td>
<td>121</td>
<td>130</td>
</tr>
<tr>
<td>GAUNA</td>
<td>302227</td>
<td>21900</td>
<td>184</td>
<td>32001</td>
<td>70</td>
<td>123</td>
</tr>
<tr>
<td>LAGAN</td>
<td>102219</td>
<td>23061</td>
<td>196</td>
<td>33109</td>
<td>220</td>
<td>272</td>
</tr>
<tr>
<td>AVID</td>
<td>130129</td>
<td>19337</td>
<td>169</td>
<td>26792</td>
<td>73</td>
<td>587</td>
</tr>
</tbody>
</table>

| TCEL: Total Conserved Length | ECL: Exon Coverage Length | NEC: Number of Exons Covered | TCLE: Total Conserved Length in Exons |

The results of global alignment of Human (chromosome1, length=2933746)-Chicken (chromosome21, length=1162785). Total Human Exon Length=120322, Total Chicken Exon Length=90767.

#### 8.3.2 Speed

In terms of speed, the overhead that GeneScan causes, is the biggest factor in reducing IGAUNA’s speed compared to GAUNA. However, it is possible to extract the names by GeneScan as a pre-processing step and use those to feed IGAUNA (the parameters of IGAUNA allow this conveniently). Branching has an influence on the speed as well, but depending on the case, we can limit it by choosing suitable values for the corresponding parameters. Overall, although branching does reduce the speed, it does not reduce it much. As we can see from Tables 8.2 to 8.7, IGAUNA works slower than GAUNA and AVID, but is faster than LAGAN (and from MUMmer as shown in Table 6.2). However with the
Table 8.6: Mouse Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE Time (s)</th>
<th>Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>110875</td>
<td>5957</td>
<td>63</td>
<td>7568</td>
<td>97</td>
</tr>
<tr>
<td>IGAGNA</td>
<td>141853</td>
<td>5928</td>
<td>62</td>
<td>7481</td>
<td>130</td>
</tr>
<tr>
<td>GAUNA</td>
<td>126716</td>
<td>5922</td>
<td>62</td>
<td>7481</td>
<td>82</td>
</tr>
<tr>
<td>LAGAN</td>
<td>123452</td>
<td>5862</td>
<td>61</td>
<td>7351</td>
<td>141</td>
</tr>
<tr>
<td>AVID</td>
<td>121918</td>
<td>5777</td>
<td>60</td>
<td>6981</td>
<td>60</td>
</tr>
</tbody>
</table>

TCL: Total Conserved Length, ECL: Exon Coverage Length, NEC: Number of Exons Covered, TCLE: Total Conserved Length in Exons.

The results of global alignment of Human (chromosome 9, length=1713862)-Mouse (chromosome 13, length=1775532). Total Human Exon Length 31912, Total Mouse Exon Length 36579.

Table 8.7: Rat Alignment Results

<table>
<thead>
<tr>
<th>Program</th>
<th>TCL</th>
<th>ECL</th>
<th>NEC</th>
<th>TCLE Time (s)</th>
<th>Mem (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal</td>
<td>381350</td>
<td>11217</td>
<td>60</td>
<td>17505</td>
<td>394</td>
</tr>
<tr>
<td>IGAGNA</td>
<td>405607</td>
<td>10891</td>
<td>56</td>
<td>16691</td>
<td>410</td>
</tr>
<tr>
<td>GAUNA</td>
<td>397121</td>
<td>10865</td>
<td>55</td>
<td>16554</td>
<td>382</td>
</tr>
<tr>
<td>LAGAN</td>
<td>397457</td>
<td>10469</td>
<td>56</td>
<td>15824</td>
<td>659</td>
</tr>
<tr>
<td>AVID</td>
<td>231496</td>
<td>10462</td>
<td>58</td>
<td>1929</td>
<td>385</td>
</tr>
</tbody>
</table>

TCL: Total Conserved Length, ECL: Exon Coverage Length, NEC: Number of Exons Covered. TCLE: Total Conserved Length in Exons.

The results of global alignment of Human (chromosome 2, length=271489)-Rat (chromosome 1, length=7380917). Total Human Exon Length 66289, Total Rat Exon Length 36510.

8.3.3 Quality Of Alignments

In order to better compare IGAGNA results with the other tools mentioned, it will be helpful to consider graphs shown in Figures 8.2 to 8.5.

As we can see in Figures 8.2 and 8.3, IGAGNA performs quite well when it comes to exon coverage. When considering exon coverage (only considering exons that have been covered more than 30°) and also total exon coverage (considering all the conserved regions that fall inside exons), IGAGNA performs better than all the other tools; well, except for preprocessing of exons, the speed can be significantly improved.
Figure 8.1: Memory Usage in Megabytes for different sequences and tools

Figure 8.2: Exon Coverage Length for Mouse-Dog and Mouse-Chicken
CHAPTER 8. IGAUNA RESULTS AND CONCLUSION

Figure 8.3: Exon Coverage Length for Human-Rat, Human-Mouse, Human-Chicken, Human-Dog.

Figure 8.4: Total Exon Coverage Length for Mouse-Dog and Mouse-Chicken.
the Human-Dog case. We consider this case in detail:

Surprisingly, LAGAN scored even higher than our optimal alignment, and we investigated explanations for this "incident". After aligning the Human-Dog sequences, we observed that LAGAN is not using the complete sequences in its final alignment: We extracted the original sequences from the aligned sequences and found that the length of the sequences used by LAGAN were 6406900 of the original and 5033645 for sequence two (as opposed to 6424515 of the original). In essence, LAGAN is "throwing away" parts of the sequence that could not be aligned properly (i.e. 1% of the first sequence and 16% of the second sequence). This results in a higher density of conserved regions (either in total or just in the exons, depending on where the thrown-away segments have been) and therefore LAGAN will score higher when measuring the conserved regions. That explains the huge difference in our Exon-Coverage-Length and Total Conserved-Length-in-Exons estimates between LAGAN and even the optimal alignment.

8.3.4 IGAUNA Improvements Compared To GAUNA

The following figures help better understand how IGAUNA improves over GAUNA. As mentioned before, we created the optimal alignment to help us better measure exon coverage. In a way, the optimal alignment shows what can be achieved in terms of exon coverage. So
to see how IGANA has improved, we have our comparisons on GAUNA and comparing to the optimal alignment, we see how much room there is for improvement. Then we see how much IGANA has been able to achieve. The results are shown in Figure 8.6.

We call sec that IGANA has improved without. In comparison to GAUNA, meaning that it has covered about 50% of the potentially coverable regions not previously covered by GAUNA. The only exception is Mouse-Chicken alignment. Although IGANA has improved in this case as well, it seems that there is still much more room for improvement (and all the other tools are falling behind in this case as well).

In terms of total conserved regions, as mentioned earlier, we cannot use the optimal alignment for comparison, because it biases exons too much and that might interfere with the alignment of other potentially good regions. Therefore we compare IGANA only to GAUNA and as we can see from Figures 8.7 and 8.8 and the tables in Section 8.3, IGANA has improved from 5 to 12 percent. The only exception is Human-Rat alignment where we see only about 3% improvement (which is the longest sequence in our test set).

Figure 8.6: Improvements in Total Exon Coverage Length (compared to GAUNA)
Figure 8.7: Total Coverage Length for Mouse-Dog and Mouse-Chicken

Figure 8.8: Total Coverage Length for Human-Bat, Human-Mouse, Human-Chicken, Human-Dog.
8.3.5 Summary of Results

As we can see from the results in the above tables and figures, all these programs are state of the art and in some cases have close results to each other. The original GAUNA is faster that all the other programs and although IGAUNA is slower than GAUNA, it is still faster than LAGAN with usually a higher quality result. AVID is relatively fast, however its exon coverage is not satisfactory and it requires significant memory. In terms of memory usage, GAUNA and IGAUNA are the best among others and one major advantage they have over the other tools is their ability to execute on a personal computer. We actually experimented on aligning a whole chromosome (about fifty million base pairs in length) and GAUNA and IGAUNA were the only tools capable of successfully accomplishing this task in less that two hours; other tools either took a very long time or simply could not execute in our test environment.

The optimal alignments that we created always have better exon coverage, so they are actually optimal in this sense. However as predicted earlier, the conserved region lengths usually drop in the optimal alignments because we bias the "real" exons significantly when producing the optimal alignment and this results in the "mis-alignment" of some other conserved regions.

These tables show once again how much the quality of an alignment depends on what is being measured. Depending on the type of measurement, sometimes the so-called optimal alignment performs better, but if we change our comparison basis, IGAUNA shows very good results. In terms of speed, GAUNA and AVID outperform others.

8.4 Conclusion and Future Work

IGAUNA has proved to be a great tool for building the global alignment of long sequences: it is efficient both in time and space. Depending on whether speed or quality needs to be optimised, IGAUNA’s parameters can be flexibly set to suit each individual case. However there is still much room to incorporate more biological heuristics into it in order to achieve better results. In the same way that GAUNA is capable of performing multiple sequence alignment, we can further extend the capabilities of IGAUNA to be used on multiple sequences as well. Scientists are constantly doing research on scoring functions and the basic definition of an alignment to cover biological phenomena such as repetitions and reversals and IGAUNA can benefit from incorporating these new ideas. There is also much
room for refining the definition and usage of an optimal alignment which requires a better understanding of biological sequences and scoring methods. There is also a potential to incorporate parallelism into IGAUNA by dividing the query string and feeding each part into a separate processor. We are also working on developing a user-friendly interface for IGAUNA and make it available as a stand-alone program and also as a web application.
Bibliography


