AN APPLICATION OF PARALLEL AND DISTRIBUTED COMPUTING METHODS TO APPROXIMATE PATTERN MATCHING OF GENETIC REGULATORY MOTIFS

by

TUSHAR KUMAR JAYANTILAL

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF TEXAS AT ARLINGTON

August 2006

Copyright © by Tushar Kumar Jayantilal

All Rights Reserved

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Nikola Stojanovic, for his continual support and guidance, which helped me a lot to achieve my career goal. Under his supervision, I learned the various intricacies involved in conducting research and also ways to tackle them. I can strongly state that this work became possible only because of his constant motivation. I would also like to express my gratitude to Dr. Esther Betrán, who made this collaborative research work possible by providing the data set and also valuable Genetics information, in order to carry out this research. I would also like to thank David Levine, for providing support while working with high-speed computing environment. I am sincerely thankful to them for serving on my committee.

I thank Dr. Bahram Khalili, for his valuable advising through out my degree. I would also like to thank Dr. Gautam Das, for providing some algorithmic support. I would like to express my appreciation to all the members of Bioinformatics lab, my friends and other loved ones for their support.

I want to appreciate the continuous encouragement and love from my family, due to which this work got executed. Finally, I want to thank the Almighty, for making everything possible and helping me at every stage of my life.

This work was partially supported by the Department of Computer Science, at The University of Texas, Arlington.

July 19, 2006

ABSTRACT

AN APPLICATION OF PARALLEL AND DISTRIBUTED COMPUTING METHODS TO APPROXIMATE PATTERN MATCHING OF GENETIC REGULATORY MOTIFS

Publication No. ______

Tushar Kumar Jayantilal, M.S.

The University of Texas at Arlington, 2006

Supervising Professor: Nikola Stojanovic

Bioinformatics is a relatively new scientific field concerned with providing computational means and support to research in molecular biology and genetics. It draws from many different areas of computer science, including database theory, algorithm design and analysis, and artificial intelligence, to name just a few. In many applications, such as one described in this thesis, a biologist is interested in locating a particular pattern, or sequence motif, in a given string or set of strings over the four-letter DNA alphabet.

In this thesis we present an efficient approach to locating promoter and other regulatory sequences in entire genomes or in specific target areas. Promoters are short conserved sequences located upstream of the genes they regulate, and they have an essential role in driving the expression of the genes. However, in higher organisms promoter sequences are very diverse, and their motifs can feature substantial sequence variation, including character (chemical base) substitutions, insertions and deletions. We were thus interested in designing methods for the efficient search for approximate matches of the target sequences to putative promoter consensus. To achieve this goal we have used a combination of classic pattern-matching algorithms and high performance computing, parallelizing the search over a number of processors.

We have used these methods in the genome-wide search for a 14-base promoter element in the genome of the fruit fly *Drosophila melanogaster*, postulated to have a role in the testis-specific expression of the genes it controls. The list of testis-specific genes has been provided by our collaborator from the UTA Department of Biology, and we have obtained the raw sequence and other genetic data from Flybase, a public database maintained by a consortium of Drosophila researchers. Although the number of genes we were interested in was moderate, 791, the number of exact patterns approximately matching the 14 base consensus was very large, approximately 38,000. This problem has thus guided the design of the methods we describe in this thesis.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS... iii

ABSTRACT iv

LIST OF ILLUSTRATIONS.. viii

LIST OF TABLES.. x

Chapter

1. INTRODUCTION………………………………………………………… 1

1.1 Pattern matching and classical pattern matching algorithms 8

1.1.1 Naïve approach 8

1.1.2 Classical algorithms 9

1.1.3 Latest developments 15

1.2 Parallel and Distributed Computing (PDC).. 18

1.3 Applications of PDC in bioinformatics 20

1.4 PDC cluster at UTA ………………………………………………… 21

2. METHODS…………………………………………………………… 23

2.1 The problem statement ……………………………………………… 23

2.2 Approaches to the problem ………………………………………….. 24

2.3 Parallelization of the approaches ……………………………................ 25

2.4 Algorithms in detail ………………………………………………. 28

2.4.1 Technical details about the algorithms 28

2.4.2 Motif properties 37

3. RESULTS.. 52

3.1 Description of the problem 52

3.2 Data source…………………………………………………………… 56

3.3 Input data set ……………………………………………………………… 57

3.4 Results ………………………………………………………..................... 61

4. DISCUSSION………………………………………………………….... 70

4.1 Algorithmic improvements…………………………………………….. 70

4.2 Possible extensions of the described methods to
solve other problems ………………………………………………... 72

REFERENCES…………………… 74

BIOGRAPHICAL INFORMATION……………………………………………… 79

LIST OF ILLUSTRATIONS

Figure Page

1.1 Central Dogma of Molecular Biology.. 2

1.2 Typical promoter locations relative to genes 5

1.3 Construction of a lexicographic tree for 2 patterns
sharing a common prefix …………………………………………………. 14

2.1 Design of our distributed system for parallelization ………………………...26

2.2 Clustering approach ………………………………………………………36

2.3 Sample automata 1………………………………………………………... 40

2.4 Sample automata 2………………………………………………………... 41

2.5 Sample clustered tree 1…………………………………………………… 43

2.6 Finite State Machine (FSM) for the sample
clustered tree 1……………………………………………………………..44

2.7 Way of cutting FSM shown in Figure 2.6,
by making a cut at level 2 ..…………………………………………………... 45

2.8 Way of cutting FSM shown in Figure 2.6,
by making a cut at level1………………...……………………...………… 45

2.9 Sample clustered tree 2…………………………………………………… 47

2.10 Splitting of sample clustered tree 2………………………………………….. 47

2.11 After splitting the sample clustered tree 2…………………………………… 48

2.12 Cuts proving the Lemma 3.………………………………………………………... 49

2.13 Sample clustered tree 3…………………………………………………………..50

3.1 Inclusion of mismatches and gaps in the core
14-base promoter……………………………….………………………. 53

3.2 Positions of approximate matches to the putative
promoter………………………………………………………………….65

4.1 Common automaton construction
scenario……………………………….. 70

4.2 Reduction of size of the automaton while
constructing……………………………………………………………..... 71

4.3 Automaton after improvement ……………………………………………… 72

LIST OF TABLES

Table Page

2.1 Comparison of Naïve with Automaton Approach .... .27

2.2 Total Number of States ………………………...33

2.3 Significant Reduction in the Size of Automaton ……………………………33

2.4 Results Proving Lemma 2....…………………………………………………42

3.1 Information about the Genes Screen Shot 1 …………………………………58

3.2 Information about the Genes Screen Shot 2 …………………………………59

3.3 Classification of Genes Based on the Extracted Information ………………..60

3.4 Classification of Category 1 Genes... 61

3.5 Results for the Entire Genome and Genes of Interest ………………………62

3.6 Classification of Reduced Hits ………………………………………………63

3.7 List of Genes Containing an Approximate Match to
the Putative Promoter in the Target Region (-200, +50)……….……………64

3.8 Control Set of Genes …………………………………………………………67

3.9 Results for the Control Set of Genes ………………………………………...68

CHAPTER 1

INTRODUCTION

Every living species is composed of cells and it can be classified as a eukaryote or a prokaryote, depending on whether its cells contain nucleus or not. Each of these cells contain deoxyribonucleic acid (DNA), which specifies genetic instructions for the biological development. DNA is a long polymer of nucleotides (Adenine, Cytosine, Guanine, Thymine), some of which, the protein-coding genes, encode for amino acid sequences which form proteins [7]. DNA is often referred to as the molecule of heredity as it is responsible for the propagation of all inherited traits, such as hair color or disease susceptibility.

DNA is transcribed into RNA (ribonucleic acid) by enzymes called RNA polymerases. Similar to DNA, RNA is also a long polymer of nucleotide bases (Adenine, Cytosine, Guanine, Uracil) and serves as a template for the synthesis of proteins. This process of transformation from DNA into RNA is called *transcription*. RNA is a single-stranded molecule whereas DNA is a double-stranded molecule, coiled in a helix. There are various types of RNA such as messenger RNA (mRNA), transfer RNA (tRNA) and ribosomal RNA (rRNA), classified on the basis of their biological role. After DNA is transformed into RNA, splicing of RNA is performed (in eukaryotes), resulting in mRNA.

Finally, mRNAs are translated into proteins, which are responsible for most functions performed by the cell. Proteins are composed of long chains of amino acids, which are formed as a result of reading triplets or codons (three nucleotides together form a single codon) and the formation of peptide bonds between them. This process takes place in the ribosomes. There are 20 amino acids and the conversion process from mRNA to a chain of amino acids is called *translation*. Both processes together, transcription and translation, form the basis of the C*entral Dogma of Molecular Biology* which describes the flow of information from DNA to proteins.

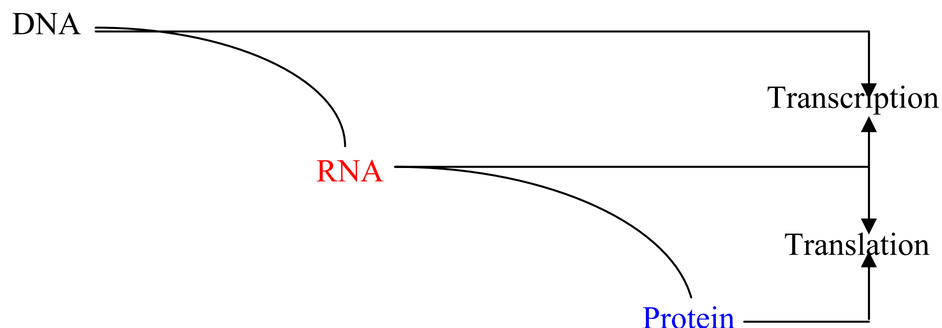


Figure 1.1 Central Dogma of Molecular Biology (Source: MIT, http://web.mit.edu/esgbio/www/dogma/dogma.html)

Bioinformatics is an inter-disciplinary field, between computer science, mathematics, statistics, biochemistry, molecular biology and genetics. It came into existence due to the rapid growth in the research of genetics, which led to huge amounts of data making it extremely difficult to perform further analysis manually. Hence the biologists needed computational support for interpreting their experiments. In computer science, bioinformatics incorporates concepts from various sub-fields, like algorithm

theory, artificial intelligence, data mining and database theory. Some of the typical bioinformatics tasks include analyzing the DNA sequence using a specific algorithm, finding out the similarity between two sequences, predicting genes, developing visualization tools in order to study protein folding, etc. For these purposes there have been many tools developed, such as BLAST [35], GENSCAN [10] or DALI [21]. Bioinformatics also plays a significant role in drugs discovery.

There are some significant motifs (short patterns), which play important role in regulating genes, such as promoters, enhancers, or silencers. A promoter is a short conserved motif, which is located upstream of the gene, responsible for driving its expression (i.e. controlling the amount and appearance of the functional products of a gene). Based on the classification of species, promoters can also be differentiated into prokaryotic and eukaryotic.

A prokaryotic promoter consists of two short sequences, located at positions -10 and -35 with respect to the start of the gene. The first one is responsible for starting transcription and the presence of the second one increases the transcription rate. Eukaryotic promoters are very difficult to characterize because they are variable, lie somewhere in the upstream region of their genes and there may be other regulatory elements near the transcription start site. These elements bind proteins called *transcription factors* [12] which are involved in the formation of the transcriptional complex. Promoters are important because many diseases like *asthma, cancer* and *heart disease* have been associated with the malfunction of promoter elements [http://en.wikipedia.org/wiki/promoter].

The process of translation of mRNA into proteins starts from the *start codon* (AUG), generally located downstream to the ribosome attachment site and extends until the *stop codon* (one of UGA, UAA, UAG) is encountered, usually before the end of mRNA. The regions which do not get translated are called *untranslated regions* (UTRs) and can be classified into 5' UTR and 3' UTR, depending upon their location, whether it is located near the 5' (upstream) end or 3' (downstream) of the gene transcript. While searching for a promoter, upstream of the gene, rather than just its protein-coding part, we have to make sure that we account for the 5' UTR region; otherwise we may still be looking for a promoter within the gene itself. However, studies [18] have concluded that it is possible for a promoter to be present within +50 bases into the gene. In consequence, we have assumed that a promoter is typically present anywhere within +50 bases with respect to the start of the gene to -200 bases [18] in the upstream region (in order to make sure that we do not miss any hit, in our study we actually considered up to -1000 bases). Figure 1.2 below shows a typical layout of a gene with respect to these features. However, a gene can have multiple transcription starts depending on its number of transcripts.

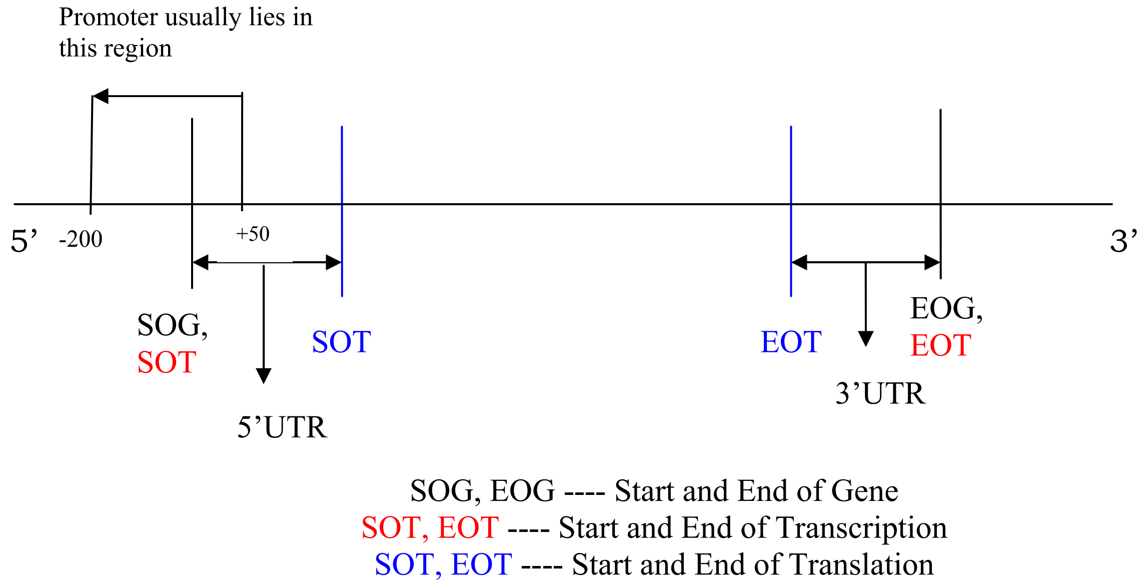


Figure 1.2 Typical promoter locations relative to genes

Tissue-specific promoters, as the name implies, are responsible for controlling the gene expression in a specific tissue. One example is a promoter determining the expression specifically in testis, thus active only in the male germ line. Significant attention is often given to these genes, because their analysis provides valuable information about development, general physiology and infertility.

Tissue-specific gene expression in higher eukaryotes can be accomplished by the recruitment of cell type-specific transcription factors to distinct promoter/enhancer sequences [2]. The binding of specific transcription factors allows for the interaction with the general transcription initiation machinery at the core promoter and leads to transcriptional activation. The tissue-specific expression of the Drosophila *β2 tubulin* gene is accomplished by the action of a 14-base activator element [18]. The *β2 tubulin* gene encodes a tissue-specific β2 tubulin variant which is exclusively used during

spermatogenesis [2]. The promoter sequence responsible for tissue-specific gene activation is confined to the 14-base activator element [18].

In the last three decades, the genetic approach has become one of the most powerful tools for elucidating biological mechanisms. It allows researchers to compare wild type with mutant phenotypes and to identify new genes involved in controlling a biological process, thus determining their functions in the organism [2]. Genes which control important functions are identified by mutations that cause defects in those functions [http://darwin.nap.edu/books/0309070864/html/151.html]. These genes are then mapped, cloned and identified at the molecular level so that the proteins they encode can be studied using methods of biochemistry and cell biology. This approach has proven to be useful, not only for basic research in model organisms but also for medical research on heritable human diseases.

There are several reasons behind carrying out the research on model organisms such as fruit flies or worms. Some of them are:

1. Research on humans and other primates is expensive and limited by ethical considerations. This is not true in other model organisms that share a lot of human genetic features because of their common evolutionary origin. The conservation of genetic structure and function has become the cornerstone of modern developmental biology, forming the basis for the usefulness of model organisms in understanding human developmental mechanisms.

2. Most model organisms are relatively small and hence easy to maintain in large populations in the laboratory.

3. Many model organisms have short generation times, which allow for rapid analysis of breeding experiments.

One of the most widely used model organisms for conducting research is *Drosophila melanogaster*, which is the common fruit fly, found worldwide. It has a genome of size approximately 180 MB, containing 13,600 genes distributed on 4 chromosomes (X, 2, 3, and 4) [http://www.ornl.gov/sci/techresources/Human_genome/faq/compgen.shtml]. The study of the fly has been instrumental in many fundamental discoveries in eukaryotic genetics, such as linkage, gene mapping, recombination frequency, and chromosomal aberrations. These all made Drosophila a key organism for genetic analysis. One of the most useful outcomes of the genetic analysis of *D. melanogaster* development has been the identification of developmental pathways conserved in most organisms.

We were particularly interested in the 14-base activator element [18], responsible for deriving male specific gene expression in Drosophila. We started with a list of 791 target genes [23] and the task was to determine these which have this particular 14-base motif located in their upstream regions. We classified the set of genes into various categories based on the information extracted, such as the length of 5' UTR or genes having multiple mRNAs. Although we first looked at this as an exact pattern-matching problem, in our application we extended this to allowing 2 mismatches, 1 insertion and 1 deletion in the 14-base motif. This created a large list of exact motifs, which we needed to look for. For this purpose, we used the classical Aho-Corasick algorithm.

In the remaining sections of this chapter, we describe the pattern matching problem in general, some classical algorithms for solving it, introduce parallel and distributed computing (PDC), the applications of PDC in bioinformatics and finally the setup of the cluster at the University of Texas at Arlington.

1.1 Pattern matching and classical pattern matching algorithms

Pattern matching problem is one of the most common problems in computer science today. This is mainly due to its large list of applications, such as text-editing, security, cryptography, web search engines, and in many others. Due to its significance, there has been a rapid growth in developing algorithms in order to deal with the pattern matching problem and we now have algorithms which can solve this problem in a very short time. In some applications, there was a need to relax the exact-matching criteria and allow some mismatches. This type of matching procedure is called *approximate pattern matching*.

1.1.1 Naïve approach

Naïve approach is a straightforward method for locating a pattern within the string of text. It involves many comparisons and thus takes more time when compared to other approaches. It starts comparing each character of the pattern, left to right with the corresponding character in the text and then shifts the sliding window by one [39]. If the length of the pattern is m and the length of the text is n, then solving the pattern

matching problem using this approach will take time O(mn). It can be useful when the text is short so that the time for pattern matching is negligible.

1.1.2 Classical algorithms

The main drawback of the brute-force (naïve) algorithm is the number of character comparisons performed. Many algorithms have been developed in order to overcome this drawback, preprocessing either the text or the pattern before starting to match them. Some of the most commonly employed algorithms for solving the pattern matching problem are Rabin Karp algorithm [32], Knuth-Morris-Pratt algorithm [11], Boyer Moore algorithm [34], and Aho-Corasick algorithm [4]. The following sub-sections will describe the important features of these algorithms, including a few others which have been developed more recently.

1.1.2.1 Rabin-Karp algorithm

Rabin-Karp algorithm [32] employs hashing for matching a pattern within a text. It is not widely used for single pattern matching, but proves to be effective for multiple patterns. The reason is in that it often runs in worst-case time which is same as the running time of naïve algorithm (O(nm)). It speeds up the testing of the equality of the pattern to the substrings in the text by using a *hash* function, instead of skipping the characters in the text. In its preprocessing phase it computes a *hash* function for every pattern we are searching for and then looks for substrings in the text with the same hash value. The efficiency of this algorithm is largely dependent on the *hash* function,

because if the hash values computed for different patterns are the same and there is a match in the text then we have to verify which pattern has matched, thus adding to the total search time. The hash values of the substrings within the text itself should be computed efficiently. This can be done by the *rolling hash* function, which treats every substring as a number in some base. For example, if the substring is "AC" and the base is 4 (corresponding to 4 DNA letters), the hash value of the substring would be $0 * 4^1 + 1 * 4^0 = 1$ (assuming values of A as 0, C as 1, G as 2 and T as 3).

The advantage of using this representation is in that it is possible to compute the hash value of the next substring from the previous one by doing only a constant number of operations, independent of the substrings' lengths. For example, suppose the text is "ACCA" and we are searching for a pattern of length 2. We can compute the hash value of substring "CA" from the hash value of previous substring "AC". This is done by subtracting the number added for the first 'A' of "AC"i.e. $0 * 4^1$ (0 is the value of 'A' and 4 is the base), multiplying by the base and adding for the last 'A' of the substring "CA", which results in a new hash value of 4.

1.1.2.2 Knuth-Morris-Pratt algorithm

Knuth-Morris-Pratt algorithm [11] employs preprocessing of the pattern, hence resulting in the reduction of the number of comparisons required, which is a bottle-neck for the naïve approach. KMP is a linear-time algorithm. It executes in two stages – preprocessing and searching. In the first stage (preprocessing), a *failure* function is computed for every character position in the pattern. Whenever there is a failure while

matching the pattern, the value computed by the *failure* function for that character position is invoked. Hence shifting of the pattern is performed based on that value. The preprocessing stage takes time O(m).

The search stage starts scanning the text from left-to-right and aligns the pattern against a substring of text until either the entire pattern is exhausted (a match to the pattern is found in the text) or until a mismatch occurs at some position i of the pattern and k of text. In the latter case, the pattern is shifted right by *i-failure* function value for position i (longest possible prefix matching the text before the mismatch). The algorithm reduces the number of comparisons required by making shifts larger than one. The preprocessing stage of the KMP algorithm takes O(m) time. The matching itself takes O(n) where n is the size of the text. Hence the total time complexity of KMP is O(m), under the assumption that $m >> n$.

1.1.2.3 Boyer-Moore algorithm

Boyer-Moore algorithm [34] also employs preprocessing of the pattern. It is the most widely used algorithm for solving the pattern matching problem. The main reason for its popularity is in that it usually runs in sub-linear time.

BM employs three rules for matching the pattern: scanning the text from right-to-left, *bad character* shift rule and *good suffix* shift rule.

Similar to the KMP algorithm, Boyer-Moore also consists of two stages – preprocessing and searching. In the preprocessing stage it calculates the *failure* function, using the above two shift rules. If a mismatch occurs at some position with

character as y of the pattern and x of the text while aligning them from right-to-left, and we have computed the right-most position of x in the pattern, then we can shift the pattern to the right so that the rightmost x in the pattern is aligned with the mismatched x of the text. Since any shorter shift would result in an immediate mismatch, the longer shift is correct and it will not shift past an occurrence of the pattern in the text. Moreover, if x never occurs in the pattern, then we can shift pattern completely past the point of mismatch in text. In these cases, some characters of the text will never be examined and the method will run in sub-linear time [39]. However, the above rule is not useful when the mismatching character from the text occurs in the pattern to the right of the mismatch point. This situation is common when the alphabet is small and the text contains many similar, but not exact substrings, as it is typical in DNA sequences. Consequently, the *bad character* shift rule was further extended, so that it shifts the pattern by finding the closest matching character position to the left of current mismatch position in the pattern.

The *extended bad character* shift rule enables larger shifts and it is highly effective in practice, particularly for English text, but it proves to be less effective for small alphabets. Hence this brought a need to introduce another mechanism called the *good suffix* shift rule. The *good suffix* rule finds out the suffix of the pattern matched so far in the text whenever there is a mismatch such that the characters before both suffixes are not the same [13]. The *extended bad character* shift rule and the *good suffix* shift rule form the basis of the preprocessing stage, which runs in time $O(m)$, where m is the length of the pattern.

In the search stage Boyer-Moore starts aligning the pattern with the text from right-to-left. When a mismatch occurs, the shifting rules together compute the *failure* function for each character position of the pattern and the pattern is appropriately shifted to the left. This stage takes time O(n), where n is the length of the text. Typically, it runs in sub-linear time and the worst-case running time is O(m), under the assumption that $m >> n$. There were numerous extensions of the basic Boyer-Moore algorithm, such as these of Sunday [15] and Horspool [33].

1.1.2.4 Aho-Corasick algorithm

Aho-Corasick algorithm [4] was developed to facilitate bibliographic searches, simultaneously looking for several patterns in a possibly very long text. It also runs in linear time. It combines the ideas of KMP with finite state machines (FSM) [13] and it consists of two phases – constructing a finite state machine for the given set of patterns and then using the machine to process the string.

In the first phase, the algorithm constructs a finite state automaton using the set of patterns, based on a lexicographic tree. The number of its states depends on the length of the common prefix in the patterns. For example, suppose that the patterns are ACGT and ACGA. Then the algorithm constructs the following tree for the given patterns:

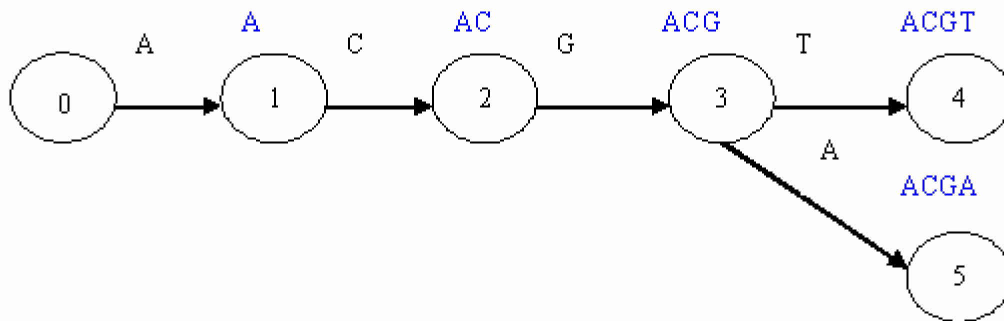


Figure 1.3 Construction of a lexicographic tree for 2 patterns sharing a common prefix.

It creates a node for every character in each pattern, as long as the character does not extend an already formed prefix. In Figure 1.3, the algorithm creates node for the initial state marked as 0 and then creates a node for every character in pattern1 (ACGT). In Figure 1.3, we have labeled each node with the characters of the pattern(s) read so far, although in practice only terminal nodes (these corresponding to the pattern ends) are labeled. While reading the characters of pattern2 (ACGA), it follows the same path up to node 3, because the first three letters of pattern2 are same as in pattern1 (ACG), and finally on reading 'T' it branches out and forms a new node (node 5). Here ACG is the longest prefix shared between the two patterns.

In an extension from the lexicographic tree to a finite state automaton, the Aho-Corasick algorithm computes *failure* functions, which return a link to a node or state for a particular input at some particular state. The *failure* links are determined for each state in the automaton. In the second phase, the text that is to be searched is parsed using the automaton constructed in phase1. A *goto* function returns a link to the next node on

reading an input character, when it is in some particular state. This function is called repeatedly during the parsing. When on reading some input the *goto* function fails to return a node, then the *failure* function is invoked. The algorithm also uses an *output* function for reporting matches and handling special cases of embedded patterns.

Typically, the Aho-Corasick algorithm runs in linear time $O(l + n)$, where l is the combined length of all patterns and n is the length of the text. This algorithm has wide applications in many fields due to its high speed and capability of matching multiple patterns.

1.1.3 Latest developments

Most of the recently developed algorithms for the pattern-matching problem are based on a combination of a few classical algorithms like KMP or Boyer-Moore. Since these algorithms combine their ideas, they make them more efficient and faster. In the following subsections, we describe some of the latest algorithms.

1.1.3.1 A simple fast hybrid pattern-matching algorithm

The simple fast hybrid pattern-matching algorithm developed by Franek, Jennings and Smyth [17] is a simple algorithm which combines the ideas of KMP, BM and Sunday algorithms. It proves to be very effective for alphabet sizes of 8 or more. The main ideas of the algorithm are to perform Boyer-Moore shifts, whenever there is a mismatch, and do KMP matching as long as there is no mismatch. This significantly reduces the required number of comparisons. Typically, this algorithm does $3n - 2m$

comparisons in most cases, which is fewer than the number of comparisons done by other algorithms. It is generally about 10% faster than algorithms like BM-Sunday (Boyer-Moore Sunday) [15] or Reverse Colussi [20].

1.1.3.2 String-pattern matching algorithm using partitioning and hashing

This algorithm, developed by Sun Kim [37], substantially differs from other algorithms used for solving the pattern-matching problem. It partitions the text to be searched into segments of input pattern length (further referred to as *l*) and then looks for the occurrence of the pattern (further referred to as *p*), using a hashing scheme. It does not take into account the shift length (i.e. shifting the pattern based on some preprocessing). The basic idea is to partition the text into segments s_i of length *l* from left to right and then assume that if the last character in s_i does not appear in *p*, then *p* cannot occur in s_i, so we can skip it. If the last character of s_i occurs in *p*, then it starts aligning *p* and s_i and verifies the occurrences of *p* according to the alignment. The alignments for each character of *p* are implemented efficiently using a hashing scheme. This algorithm proves to be efficient and significantly faster than Sunday's [15] and Horspool's [33] extension of Boyer-Moore algorithm [34].

1.1.3.3 SID_GT algorithm

SID_GT algorithm, developed by Oommen and Loke [6], received its name based on its capability to handle Substitutions, Insertions and Deletions (SID), and also Generalized Transpositions (GT; permits the transposed symbols to be subsequently

substituted). This algorithm takes into account the concept of edit distance (minimal number of transformations required to convert one string into other). It computes distances based on the properties of the two strings (i.e. prefixes of two strings) and calculates a set of five elementary edit distances defined using some elementary functions. This computation is performed recursively. This algorithm is significantly faster than other algorithms of this kind like Wagner & Fisher algorithm [30] and Lowrance & Wagner algorithm [28, 29] for handling errors in a string while matching.

1.1.3.4 Algorithm for string matching with *k* mismatches

Several algorithms have been developed to handle the problem of inexact matching with at most *k* mismatches. In particular, the algorithm developed by Amir, Lewenstein and Porat [1] finds all locations where the pattern has at most *k* errors and proves to be faster than other earlier algorithms. This algorithm performs in two stages – the marking stage and the verification stage. In the marking stage it identifies the potential starts of the pattern, and does a crude pruning of the potential candidates. It also uses a counting argument which shows the number of potential candidates left. In the verification stage it verifies which of the potential candidates are indeed a pattern occurrence. It runs in $O(n\sqrt{k \log k})$ time.

1.2 Parallel and Distributed Computing (PDC)

Parallel computing refers to the simultaneous execution of the same task (i.e. copying/duplicating a single task into many copies) on multiple processors with some coordination between them, in order to obtain the results much faster. A parallel computing system is a computer with more than one processor sharing memory. There are various ways to classify parallel computers, but the major one is based on the memory architectures [8]. Based on this criterion, they can be classified into shared memory parallel computers and distributed memory computers.

A shared memory parallel computer contains multiple processors sharing a global memory with all processors connected to a bus which acts as a medium for communications. A distributed memory parallel computer also contains multiple processors, but each processor has its own local memory. There is no global memory address space shared between them.

The processors in a parallel computer may communicate in a number of ways through a shared memory, a shared bus, a crossbar switch, or an interconnected network. Various architectures of parallel computers include multiprocessing, computer cluster, grid computing and distributed computing. However, a system of n processors is generally less efficient than one n-times faster processor. Parallel computing proves to be very effective for those tasks which require computation that can be easily divided into sub-tasks.

Distributed computing is one of the most common approaches to parallel computing. In distributed computing, two or more computers communicating over a

network are assigned a common task. The interaction between the computers in a distributed computing environment is very important and various types of communication protocols are used. Another important factor to be considered is the ability to exchange the application software between computers.

Some of the advantages of a distributed system are openness (each subsystem is continually open to interaction with other systems) and scalability (each subsystem can be scaled or altered according to the change in the number of users or resources) [8]. Disadvantages of using a distributed system are in that troubleshooting becomes very difficult, because one has to establish the connection to each remote node for analyzing the problem, and in that programming them can be quite difficult.

Distributed computing differs from a computer cluster, which consists of multiple stand-alone machines acting in parallel across a local high speed network. Unlike the computers in a distributed system, these machines are very tightly coupled [8]. A cluster exclusively runs group tasks whereas distributed computing facility does not. Very often, computers which are widely separated geographically can be a part of a distributed system, whereas in a cluster all the machines are physically close, often in the same room.

Grid computing is an example of a very large distributed computing system. A grid consists of many isolated clusters connected by means of a large network (such as the Internet). A grid computing environment is typically used to solve problems which cannot be solved by any single installation in a reasonable time.

1.3 Applications of PDC in bioinformatics

Most of the work in bioinformatics involves substantial computation because of the large genome sizes and the complexity of biological structures, processes and interactions. Even though a single machine can usually handle these computations, it takes a long time to obtain the results. So there is always a need to find alternative means of achieving speed. For example, when one wants to find the occurrences of a large number of patterns in the complete genome of Drosophila, which is of size 180 MB, then it will take many hours using the naïve approach to pattern matching ($O(nm)$). One can easily argue that the naïve approach should be avoided. Even after the replacement of the naïve approach by a more efficient algorithm, it may still take unacceptably long to obtain the results. Hence this problem can be alleviated by using multiple machines.

With rapid developments in the field of computer science, there are many applications of PDC. Some of the distributed computing projects carried out in bioinformatics are:

1. Predictor@home - Initiated to predict the protein structures from the sequences using BOINC (Berkeley Open Infrastructure for Network Computing). The major goal of the project is testing and evaluating of new algorithms to predict protein structures [http://predictor.scripps.edu/].

2. Rosetta@home - This project is also aimed at protein structure prediction, which often results in the hang-up of a single machine [http://boinc.bakerlab.org/rosetta/].

3. PrimeGrid - initiated for testing an implementation of BOINC in Perl [http://www.primegrid.com/].

Using powerful parallel computing tools can lead to significant breakthroughs in deciphering genomes, understanding genetic disease, designing customized drug therapies, and understanding evolution. Generally, the data sets are large and the computation involved is complex, thus taking substantial time for producing the results. Therefore the need to change the nature of the program from sequential to parallel, in order to speed up the execution.

The application of PDC has quickened the biological discovery process to a great extent. One can easily argue that parallel and distributed computing have totally changed the view of bioinformatics.

1.4 PDC cluster at UTA

The Distributed and Parallel Computing Cluster (DPCC) at UTA was established in 2004, funded by NSF [http://www-hep.uta.edu/~mcguigan/dpcc]. The main goal was to facilitate collaborative research which requires large scale storage, high speed access and mega processing. It contains 194 processors and large shareable high speed storage (100's of Terabytes). Out of 97 machines, 81 are used as worker nodes, 2 are used as interactive nodes, 10 are used as IDE based RAID servers and the remaining 4 support Fibre Channel SAN. Each worker node consists of Dual Xeon Processors, either at 2.4 GHZ or 2.6 GHZ, 2 GB RAM, IDE storage of size either 60 GB or 80 GB and have Redhat Linux installed.

Some of the previous bioinformatics work performed on DPCC includes:

1. Pseudogene detection system based on a high-performance computing platforms such as cluster and grid. This system enables the user to detect pseudogenes by providing the parameters used in the process. The initial steps in the system involve substantial processing, which benefits from the use of cluster and grid [5].

2. REPCLASS, an automated tool for the classification of repeats, based on various characteristics exhibited by transposable elements. It utilizes the power of cluster computing to quickly classify repeats in entire genomes. It generates the final classification based on outputs of various stages, which may result in combined evidence for the classification [25].

CHAPTER 2

METHODS

In this chapter we describe the problem statement, the various approaches which we took to tackle the problem and our use of parallel and distributed computing to make these approaches more efficient.

2.1 The problem statement

With a very few exceptions, almost all motif finding problems in bioinformatics rely on approximate pattern matching. This is because of the nature of the chemical interactions taking place between DNAs, RNAs and proteins, where there is usually some variety permitted in the consensus sequences of the binding sites. In particular, individual gene promoter sequences can substantially vary from their consensus, allowing for a limited number of insertions, deletions and substitutions.

The primary problem we attempted to solve was that of finding the putative promoter sequences similar to one described in the literature, and matching these sequences to the position of genes known to be co-expressed with the gene driven by the original promoter. Promoter sites are targets for binding by RNA polymerase, and other elements of the transcription initiation complex. In eukaryotes they can have very diverse sequences and even when a promoter is a target to essentially same transcriptional complex its sequence can show a substantial variation from the consensus.

In our analysis we have limited the number of substitutions to no more than 2, and allowed up to 1 insertion or deletion in the consensus, but no combinations of insertions and deletions. The motivation for this limit was in that a greater variety would lead to too many possible targets, and consequently poor specificity. Although the consensus itself may not be known, one can use the experimentally confirmed promoter sequence to approximate it.

2.2 Approaches to the problem

One possible way of dealing with an approximate pattern-matching problem is to generate the list of all motifs which would have a match with the original pattern, and then attempt to match them exactly. An exact match with a member of this set would then automatically be an approximate match to the original pattern. When working with such set, a naïve approach would perform poorly. This is because the mismatches, insertion and deletion can occur at any position in the pattern and thus generate a large set to process. For example, assume the pattern is ACG. Including 1mismatch in it will create set ACG and CCG, GCG, TCG (mismatch at the first position), AAG, AGG, ATG (mismatch at the second position), ACA, ACC, ACT (mismatch at the third position). Similarly, the set can be extended for 2 mismatches, 1 insertion and 1 deletion.

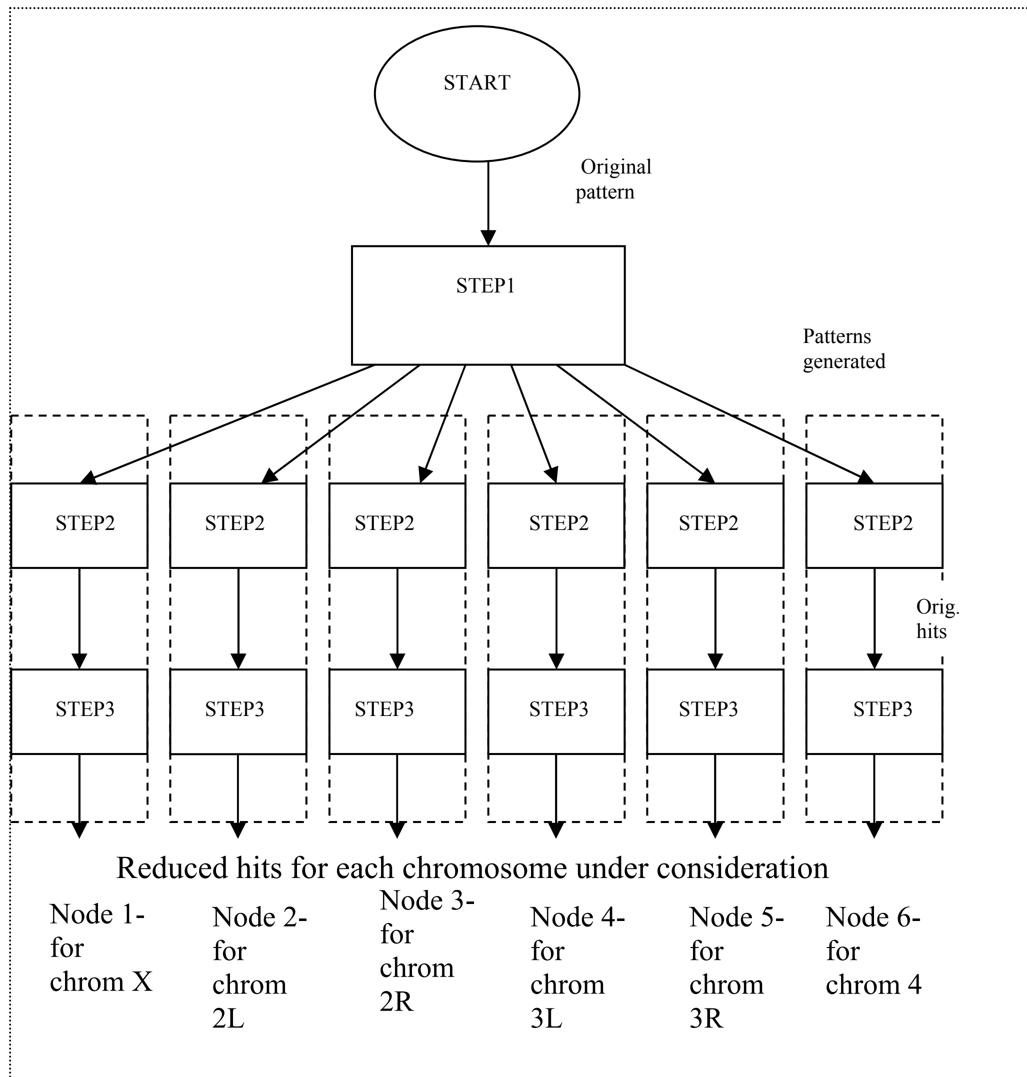
Our first concern was the time necessary for obtaining the results. We considered a number of classical algorithms for replacing the naïve approach to pattern matching and used Aho-Corasick method because we were looking for a list of patterns,

and because of its linear running time. Our work was directed by a specific search for near-occurrences of a 14 base long testis-specific promoter in the genome of *Drosophila melanogaster* [18]. We have thus generated a set of 38,783 patterns we could search for exactly, and constructed an automaton accommodating all these patterns. However, although the size of this automaton was manageable even on a desktop machine, we considered the explosion in the number of states to be a major problem. In the general case, one may work with a substantially larger consensus, possibly permitting for more substitutions and/or indels. In consequence, one of our goals was to devise methods for optimally dividing the full automaton into smaller parts, which could also be separately matched using several CPUs.

2.3 Parallelization of the approaches

In a parallel approach to pattern matching, one divides the system into many sub-systems (small problems), and then distribute each of these sub-problems to a processor, connected in a cluster. In our application, each of these sub-problems was submitted as a separate job on the cluster using PBS commands (Portable Batch System) and the output of each job was stored in a common directory. A shell script was written for the execution of each job and submitted using a *qsub* command. The first, and the most straightforward division of our problem into separate tasks involved the separation of the entire Drosophila genome into its 6 constituent chromosomal arms, and submitting the complete automaton (encapsulating all patterns) along with a chromosomal arm to a separate processor. The overall design and the flow of

information from one step to another in a distributed system using this approach are shown in Figure 2.1 below:



STEP2 referred to the hits obtained in the entire Drosophila genome. STEP3 referred to the hits obtained near the given list of genes, further described in the Results section

Figure 2.1 Design of our distributed system for parallelization.

As expected, the distributing of the jobs led to a substantial speedup in computation. The time taken was dependent on the longest job executing, among all jobs submitted to the cluster. However, after the distributing our implementation of Aho-Corasick algorithm was taking approximately1.5 hours compared to 4 taken by the naïve approach. So we divided the system into even smaller steps, which could run in parallel. Initially, we created only six different units, but latter on we continued dividing it, often coupled with the division of the complete automaton to smaller ones comprising subsets of all patterns. The smaller automata also led to a reduced number of hits, and thus reduced overhead in actions done for each hit (these actions being distributed in this case).

After all optimizations it took just minutes to produce the results. Hence the overall performance of the system was substantially improved (about 50-fold) by efficiently implementing Aho-Corasick algorithm and using cluster computing. Table 2.1 below shows the overall comparison between the times taken for each of the approaches, including the naïve method we used in order to measure the performance gain.

Table 2.1 Comparison of Naïve with Automaton Approach

Approach	Environment	Time Taken
Naïve	UNIX	18 hours, 45 minutes
Naïve	CLUSTER (6-WAY)	3 hours, 48 minutes
Finite State Automaton	CLUSTER (6-WAY)	1 hour, 30 minutes
Finite State Automaton	CLUSTER (30-WAY)	20 minutes

As we shall describe in more detail later, we have intersected the list of motif matches in the entire Drosophila genome with the list of target genes we were interested in, and this process was the final step in our computation.

2.4 Algorithms in detail

2.4.1. Technical details about the algorithms

Our system employs four algorithms for identifying all the occurrences of a large list of motifs in the complete genome of a species, generated from a given motif and then filtering out the hits obtained according to the given list of genes. The following sections describe the each of these algorithms in detail.

2.4.1.1 Algorithm for generating patterns

This algorithm is employed for generating all patterns, from a single motif, by taking into consideration substitutions, insertion and deletion of any character from the DNA alphabet, consisting of 4 letters (A, C, G, T).

The algorithm can generate patterns from a template of any length, for each of the following cases:

1. Up to 2 mismatches in the pattern,
2. 1 insertion in the pattern,
3. 1 deletion in the pattern,
4. Including both mismatches and insertion in the pattern and

5. Including both mismatches and deletion in the pattern.

Based on the user's choice, it can generate patterns accordingly. The core part is shown in Algorithm 2.1 below:

Algorithm 2.1: Generation of Patterns

Generate-patterns (*choice*, pattern *p*, *plength*)
If *choice* is 1 **then**
   If number of mismatch =1 **then**
      For *pos* = 1 to *plength* **do**
          Foreach *c* in [A, C, G, T] **do**
              Replace the character at *pos* by *c*;
   Else if number of mismatch =2 **then**
      For *pos* = 1 to *plength* **do**
          Foreach *c* in [A, C, G, T] **do**
              Replace the character at *pos* by *c*;
                  For *pos1* = *pos* + 1 to *plength* **do**
                      Foreach *d* in [A, C, G, T] **do**
                          Replace the character at *pos1* by *d*;
Else if *choice* is 2 **then**
   For *pos* = 1 to *plength* **do**
        Foreach *c* in [A, C, G, T] **do**
            Insert *c* at *pos*;
Else if *choice* is 3 **then**
   For *pos* = 1 to *plength* **do**
        Delete the character at *pos*;
Else If *choice* is 4 **then**
   If number of mismatch =1 **then**
      For *pos* = 1 to *plength* **do**
          Foreach *c* in [A, C, G, T] **do**
              Replace the character at *pos* by *c*;
   Else if number of mismatch =2 **then**
      For *pos* = 1 to *plength* **do**
          Foreach *c* in [A, C, G, T] **do**
              Replace the character at *pos* by *c*;
                  For *pos1* = *pos* + 1 to *plength* **do**
                      Foreach *d* in [A, C, G, T] **do**
                          Replace the character at *pos1* by *d*;
   Foreach p_i in [p_1, p_2, p_3……p_n] **do**
        For *pos* = 1 to p_i*length* **do**

```
            Foreach c in [A, C, G, T] do
                Insert c at pos;
Else If choice is 5 then
    If number of mismatch =1 then
        For pos = 1 to plength do
            Foreach c in [A, C, G, T] do
                Replace the character at pos by c;
    Else if number of mismatch =2 then
        For pos = 1 to plength do
            Foreach c in [A, C, G, T] do
                Replace the character at pos by c;
                    For pos1 = pos + 1 to plength do
                        Foreach d in [A, C, G, T] do
                            Replace the character at pos1 by d;
    Foreach p_i in [p_1, p_2, p_3…….p_n] do
        For pos = 1 to p_ilength do
            Foreach c in [A, C, G, T] do
                Delete the character at pos;
```

In cases 4 and 5, this method first generates all patterns including mismatches and then includes insertion or deletion in these patterns.

2.4.1.2 Algorithm for matching patterns

The core of our approach is an implementation of the Aho-Corasick algorithm for matching a list of motifs against the genome.

The Aho-Corasick algorithm has already been described in section 1.1.2.4. It is a dictionary-matching algorithm which locates elements of a finite set of patterns (the "dictionary") in an input text. It matches all patterns at once, so the complexity of the algorithm is linear in the size of the patterns plus the size of the search string. As mentioned earlier, the algorithm consists of two phases – constructing a keyword tree (i.e. an automaton from the set of patterns and parsing the string through the automaton

constructed in the previous phase. The detailed description of these phases is given in Algorithm 2.2 below:

Algorithm 2.2: Aho-Corasick Algorithm

Phase 1 – Construction of automaton from the list of patterns
Construct-automaton ([$p_1, p_2 \ldots\ldots\ldots p_n$])
Begin with a root node;
Foreach character *c* in p_1 **do**
Add an edge and a node;
Foreach p_i in [$p_1, p_2 \ldots\ldots\ldots p_n$] **do**
Starts at the root; follow the path labeled by characters of p_i;
If the path ends before p_i **then**
Continue it by adding new edges and nodes for all the remaining characters of p_i;
Store identifier *i* of p_i at the terminal node of the path;
Foreach node *v*, except the root, with a label *l (v)* **do**
Find a node *v'*, such that *l (v')* = longest proper suffix of *l (v)*, computed by trying nodes labeled by shorter and shorter suffixes of *l(u)*, where *u* is the parent of *v* such that *l(v)* = *l(u) c*;

Phase 2 – Lookup of the automaton constructed
Lookup ( )
Starts from the root;
Foreach character *c* in the given string *s* **do**
Follow the path labeled by characters of any p_i as long as possible; if not makes a transition based on the failure link for that node;
If the path leads to a node with an identifier *i* **then**
p_i is found in *s*;

2.4.1.3 Algorithm for arranging patterns

Using the Algorithm 2.2 we found the hits in the entire genome of Drosophila. Even though the size of the automaton constructed was large, a single machine was able to handle our problem. However, one can easily argue a scenario where a single

machine would fail, due to large automaton size. In that case we have to divide the single automaton into a number of smaller ones which can be handled by a computer, even though this would result in introducing additional states.

In one of our tests we have divided our automaton constructed from 38,783 patterns into 5 smaller automata and distributed them to separate processors. We were still concerned about the size of each of the newly formed automata, due to the additional states. We have therefore developed an algorithm which reduces this additional number of states. Basically, the algorithm arranges all patters in such a way that similar ones appear in a group, based on the length of their common prefix. This strategy tends to reduce the total size of the automata. The core method is shown in Algorithm 2.3 below:

Algorithm 2.3: Arranging a List of Patterns

Arrange-patterns ([$p_1, p_2 \ldots\ldots\ldots p_n$], *pat-size*)
Copy p_1 to the output file;
l = *pat-size*;
Pre = p_1 of l;
While ($l > 0$) **do**
  Foreach p_i in [$p_1, p_2 \ldots\ldots\ldots p_n$] **do**
    Pre1 = p_i of l;
    j = compare (*pre*, *pre1*);
    If j = match **then**
      Copy p_i to the output file;
$l = l$-1;

One needs to be careful in the initial determination of a common prefix between the patterns. In our particular case, we knew that all patterns were derived from a 14-base pattern (ATCGTAGTAGCCTA) [18], so we looked at the variants of this 14-base

motif as a common prefix, each of length 13. Table 2.2 below shows the total number of states obtained after adding the states from each of the automata, before and after arranging the patterns.

Table 2.2 Total Number of States

Scenario	No. of states in the automata
Before the arrangement of patterns	153395
After the arrangement of patterns	140693

Table 2.2 does not show significant reduction in the size of the automata. The reason behind this is in that most of the patterns generated were already partially arranged. In order to confirm this, we have randomly shuffled all 38,783 patterns in a file and constructed the automata using this order. We then arranged the patterns using the Algorithm 2.3 and constructed the automata after arranging and suitably dividing the set. Table 2.3 below shows the significant difference between the two, in terms of the size of the constructed automata.

Table 2.3 Significant Reduction in the Size of Automata

Scenario	No. of states in the automata
Random Placement of patterns	256786
After Arrangement of patterns	140693

2.4.1.4 Algorithm for filtering out the hits

Using the previously described algorithms we have obtained the hits for the patterns in the complete Drosophila genome. But the problem was to find out the hits of these patterns in the upstream regions of genes from the given list. So we now incorporated an algorithm which discards all hits not found in the considered regions, taken from the upstream (-1000 bases) to the downstream flanks (+1000 bases) of all the genes of interest. The core method is shown in the Algorithm 2.4 below:

Algorithm 2.4: Filtering the Hits Obtained in the Entire Genome

Foreach hit *h* in the hits file **do**
    Foreach gene *g* in the genes file **do**
        regionstart = *genestart* - 1000;
        regionend = *geneend* + 1000;
        If (*h* > = *regionstart*) and (*h* <= *regionend*) **then**
            Copy *g* to the output file;
            Copy *h* to the output file;

Since we have divided the entire Drosophila genome to individual chromosomes to be separately matched, we have also divided the given list of genes based on their chromosomal locations and used these sub-lists in Algorithm 2.4.

2.4.1.5 Clustering approach

By using a composite automaton we have improved the efficiency of the overall system. Most modern computers have large memory, so the size of constructed automaton should rarely be an issue. However, in a few exceptional cases we have to divide the automaton so that its size can be efficiently handled.

Now let us assume that the memory of a system is not sufficient to handle the current size of the automaton constructed from our 38,783 patterns and that we know the size *s* of an automaton which can be handled in memory. We thus have to divide this automaton into a number of small automata, each of size less than or equal to *s*. We also need to perform this division efficiently, so that it results in a minimal number of additional states. This means that we have to arrange the patterns by some method. We describe an approach which can solve this problem below.

This approach uses hierarchical clustering, with a slight difference. We start with clustering the patterns based on the computed distances between them. The distance metric is computed as the difference between the pattern lengths and the length of its longest common prefix. For example, the distance between "ATAGATCATG" and "ATACATCATG" would be 7, but the distance between "ATAGATCATG" and "ATAGATCCAT" would be only 3. We then form a matrix representing the distance of each motif from every other motif in the list. Like in the hierarchical clustering, we can group these patterns based on the distance. But unlike hierarchical clustering where a node can have only two children, in this case we can have any number of children branching from a single node, and the level at which a node is formed represents the difference in prefix sizes between the motifs. For example, suppose we have the following motifs – AGATC, AGTAC, ATTAC, ATTCC and ATTGC. Figure 2.2 below shows the clustering of these motifs:

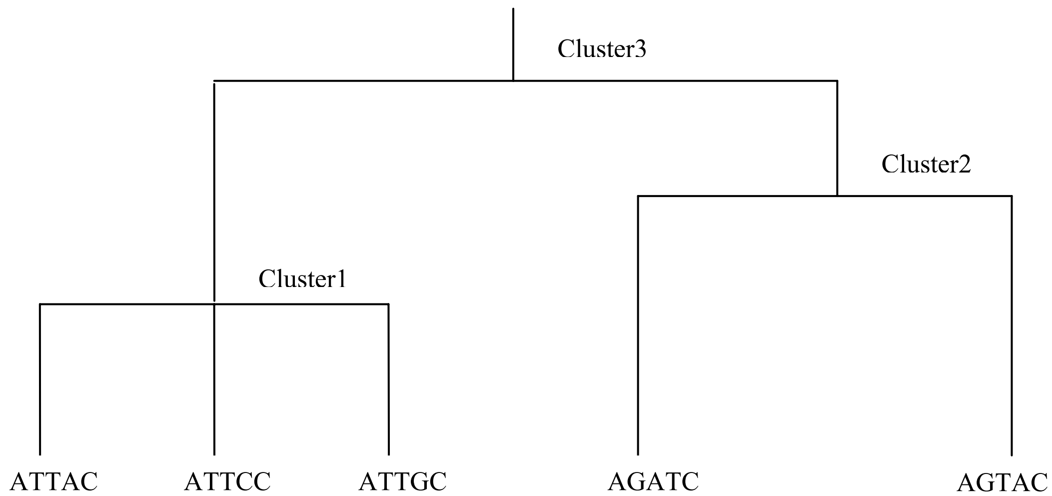


Figure 2.2 Clustering approach

In Figure 2.2, Cluster1 indicates distance of 2 between the motifs, Cluster2 indicates distance of 3 between the motifs and Cluster3 represents distance of 4.

While constructing the clustered tree, we keep track of the maximal number of states at each of the formed nodes. So when we divide the tree by cutting off a branch, in order to obtain smaller size automata, it provides information about the size of the result. This provides an efficient way of dividing the automaton when the overall size is too large and we know the size that can be handled by the memory.

There are some issues which need to be addressed before this method can be implemented. One of them is that a distance can be calculated only for strings of the same length, but in our case all motifs are not of same length (they are of lengths 13, 14 or 15). So we need to make some changes in the core distance metric to suit our needs,

by substituting the length of the motifs with their maximum, even if it may lead to artifacts, such as having identical strings at a distance greater than 0.

2.4.2 Motif properties

Lemma 1: Suppose we have a motif of length m and we allow up to 2 mismatches, 1 insertion and 1 deletion in it. The maximal total number of motifs which can be derived in this way can be obtained by adding the counts from the cases below.

Case 1: Original pattern

Maximal number of patterns = 1, without any substitutions, insertion or deletion.

Case 2: Allowing 1 mismatch in the original pattern

Maximal number of patterns generated = $3m$.

Case 3: Allowing 2 mismatches in the original pattern

Maximal number of patterns generated = $9/2 * m (m-1)$.

Case 4: Allowing 1 insertion in the original pattern

Maximal number of patterns generated = $4(m-1)$.

Case 5: Allowing 1 insertion and 1 mismatch in the original pattern

Maximal number of patterns generated = $12m (m -1)$

Case 6: Allowing 1 insertion and 2 mismatches in the original pattern

Maximal number of patterns generated = $18m ((m-1)^2)$

Case 7: Allowing 1 deletion in the original pattern

Maximal number of patterns generated = m

Case 8: Allowing 1 deletion and 1 mismatch in the original pattern

Maximal number of patterns generated = $3(m^2)$

Case 9: Allowing 1 deletion and 2 mismatches in the original pattern

Maximal number of patterns generated = $9/2 * (m^2)(m-1)$

Proof: Case 1 is straightforward. In Case 2, we can allow 1 mismatch, which can occur at any position within the motif and this mismatch can be one of the three remaining letters. Thus the maximal number of patterns generated is $3m$.

In Case 3, we can allow 2 mismatches and they can occur at any positions in the motif. Again, the mismatches can be any one of the three remaining letters. Therefore, we have to consider the combinations of the positions at which these two mismatches can occur and for each combination, we can have 9 different motifs. Thus the maximal number of patterns generated is ${}^{m}C_{2} * 9$, where ${}^{m}C_{2} = (m(m-1))/2$ is the number of possible combinations of positions.

In Case 4, we can allow 1 insertion at any position in the original pattern and the insertion can be one of the four letters [A, C, G or T]. There are m-1 possible insertion positions, so the maximal number of patterns is $4(m-1)$. In practice some of these patterns would be identical, depending on the exact motif composition; however we are here concerned with the maximal possible number only.

In Case 5, we can allow 1 mismatch and 1 insertion in the pattern. The count is thus resulting from the multiplication of Case 2 and Case 4.

In Case 6, we can allow 2 mismatches and 1 insertion in the pattern. Similar to Case 5, we first generate all the patterns by allowing only mismatches and then include

insertion in those patterns. Hence the maximal number of patterns generated is 9/2 m (m-1) * 4 (m-1) = 18m ((m-1) ^{2}).

In Case 7, we can allow 1 deletion, which can occur at any position in the given motif. Hence maximal number of patterns generated is m.

In Case 8, we can allow 1 mismatch and 1 deletion in the pattern. Here we first generate all the patterns by allowing only 1 mismatch and then include 1 deletion in all the previously generated patterns. Hence the maximal number of patterns can be generated by multiplying the counts from cases 2 and 7.

In Case 9, we can allow 2 mismatches and 1deletion in the pattern. Similar to Case 8, we first generate all patterns by allowing mismatches and then include deletion in all those patterns. Hence the maximal number of patterns can be generated by multiplying the counts from cases 3 and 7.

□

Lemma 2: When an automaton A of size N is divided into k smaller automata, $A_1, A_2, \ldots, A_k$, each of size $N_1, N_2, \ldots, N_k$, respectively, then $\sum_{i=1}^{k} N_i > N$.

Proof: The sum of the sizes of A_1 through A_k cannot be smaller than N, since they cumulatively need to have all states necessary to parse all patterns processed by A. Each pattern starts from the root node, so each of A_i needs to incorporate a sub-tree starting from the root, and all sub-trees must be incorporated in some A_i. It follows that $\sum_{i=1}^{k} N_i >= N$.

We now show that it cannot be that $\sum_{i=1}^{k} N_i = N$. If any A_i and A_j share a path from the root (all their patterns having the same prefix) then they must repeat the states along that path, and other automata A_l, l not equal to i, j, must still contain all other paths. If no A_i and A_j share a path from the root (i.e. no two automata track patterns with same prefixes) then each of A_l must replicate the root itself, leading to $\sum_{i=1}^{k} N_i = N + k - 1$.

□

We illustrate Lemma 2 with the following example. Suppose we have two patterns, ACTA and ACTG, then the automaton constructed from these two patterns looks like

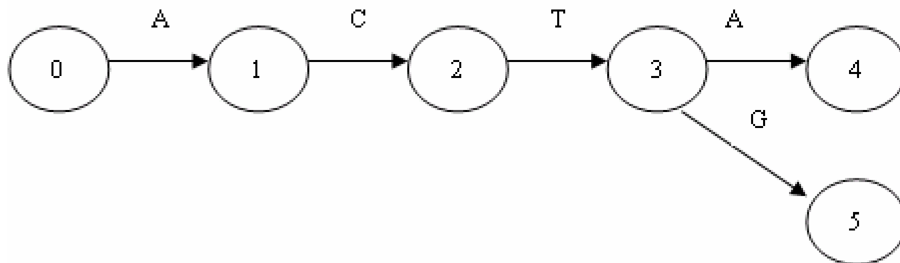


Figure 2.3 Sample automata 1

The size of the automata shown in Figure 2.3, N = 6. Now if we build two separate automata, one for each motif, as shown in Figure 2.4 below:

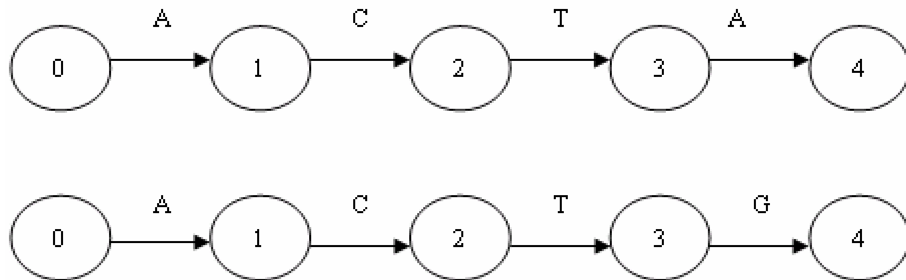


Figure 2.4 Sample automata 2

The size of the first automaton is $N_1 = 5$ and of the second one is $N_2 = 5$, so $N_1 + N_2 = 5 + 5 = 10 > N$. When splitting an automaton into 2, the number of states is increased by 1 + *length of the common prefix* (between the motifs tracked by the automaton)

In this example, the common prefix between the two motifs is ACT, hence its length is 3, so 1 + 3 = 4, which is the difference between $(N_1 + N_2)$ and N.

Considering the worst case, when both the patterns in the above example are different, still the number of states increases by 1 (length of the common prefix is 0). Another important observation is, when we have an automaton made from patterns sharing long prefixes, then splitting it into small automata results in a higher number of new states compared to the automata constructed from patterns sharing shorter prefixes.

In order to further illustrate Lemma 2, we took one of the finite state machines obtained after arranging the patterns and divided into five equal sets, then constructed a machine for each set of patterns. The number of states in the original finite state

machine was 16981. Table 2.4 below shows the results after dividing this finite state machine into 5 smaller machines.

Table 2.4 Results Proving Lemma 2

Finite State Machine no.	No. of states
1	1972
2	3150
3	3877
4	5029
5	4855
Total	18883

Corollary: Cuts at the higher levels in a clustered tree always result in minimal number of additional states in the newly formed automata.

Proof: The higher levels in our clustered tree correspond to shorter common prefixes and from Lemma 2 it follows that shorter common prefixes would lead to a smaller number of additional states.

□

The results stated above form a theoretical basis for our division of a large automaton into smaller ones. In order to divide a large automaton we should cluster all the motifs first and then use the clustering for the division. By Lemma 2, the division of a single automaton into multiple automata would result in additional states. Suppose

we have four different motifs, AC, AG, GT and GA. We cluster all the motifs. The clustered tree is shown in Figure 2.5 below:

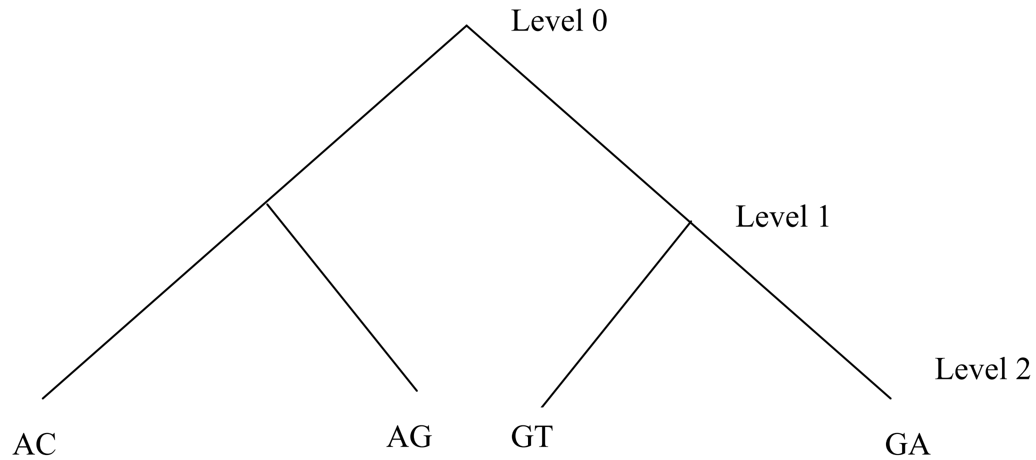


Figure 2.5 Sample clustered tree 1

A finite state machine built from the clustered tree in Figure 2.5 is shown in Figure 2.6 below:

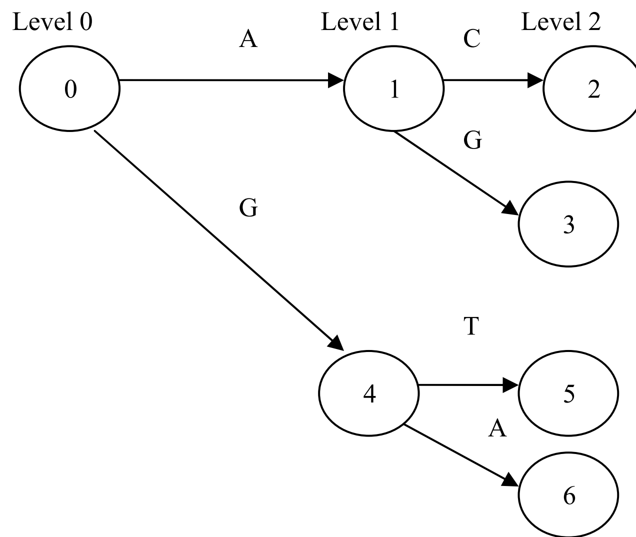


Figure 2.6 Finite State Machine (FSM) for the sample clustered tree 1

Now if we want to divide the above automaton into two, then one can cut the automaton in various ways depending upon the combinations of the edges. Some of the ways in which the cuts can be made in the FSM shown in Figure 2.6 are shown in figures 2.7 and 2.8. In these figures, if a cut is made on the edge connecting level1 and level2, then we say that it is made at level2 and when a cut is made on the edge connecting level0 and leve1, then it is a level1 cut.

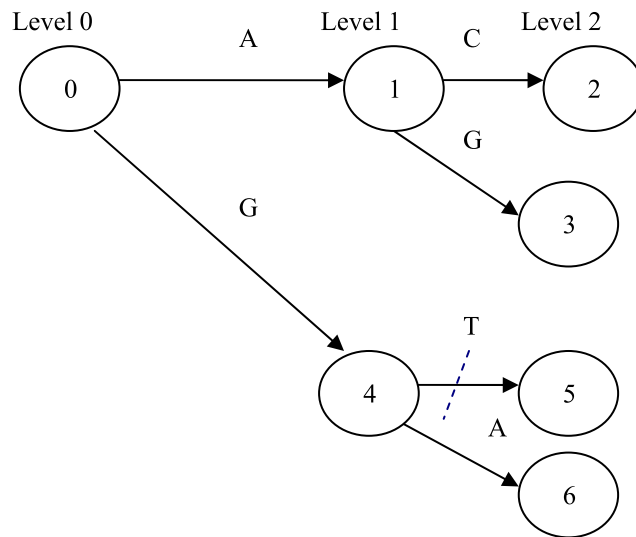


Figure 2.7 Way of cutting FSM shown in Figure 2.6, by making a cut at level 2

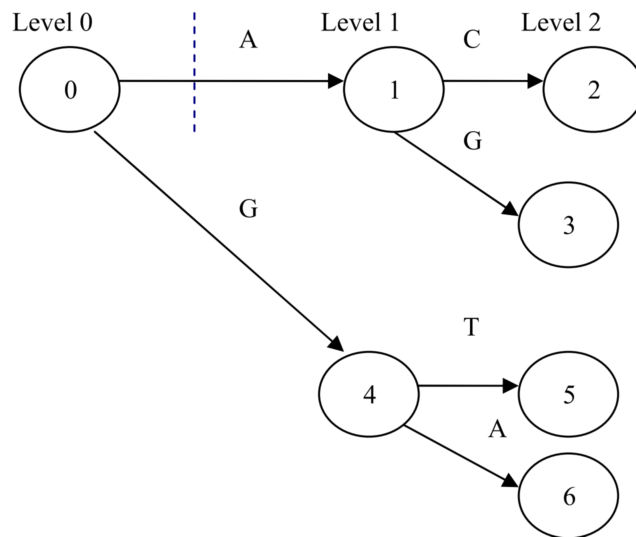


Figure 2.8 Way of cutting FSM shown in Figure 2.6, by making a cut at level 1

If we denote the number of cuts by n and the level at which cut i has been made by L_i, then we can generalize a formula for finding the number of extra states.

No. of extra states = $\sum_{i=1}^{n} L_i$ (2.1)

Using (2.1), we can compute the number of extra states for each of the cases described above.

For Figure 2.7, No. of extra states = $\sum_{i=1}^{n} L_i = 2$ (Level 2 cut).

For Figure 2.8, No. of extra states = 1 (Level 1 cut).

Hence a cut at higher levels always results in the minimal number of additional states.

Lemma 3: The number of cuts in a clustered tree which will result in automata with the minimal number of additional states depends on the number of children the root node possesses.

Proof: Based on Lemma 2 and its corollary, the cuts should be made at the higher levels in a clustered tree inorder to obtain minimum number of additional states. Suppose we have a clustered tree shown in Figure 2.9 below:

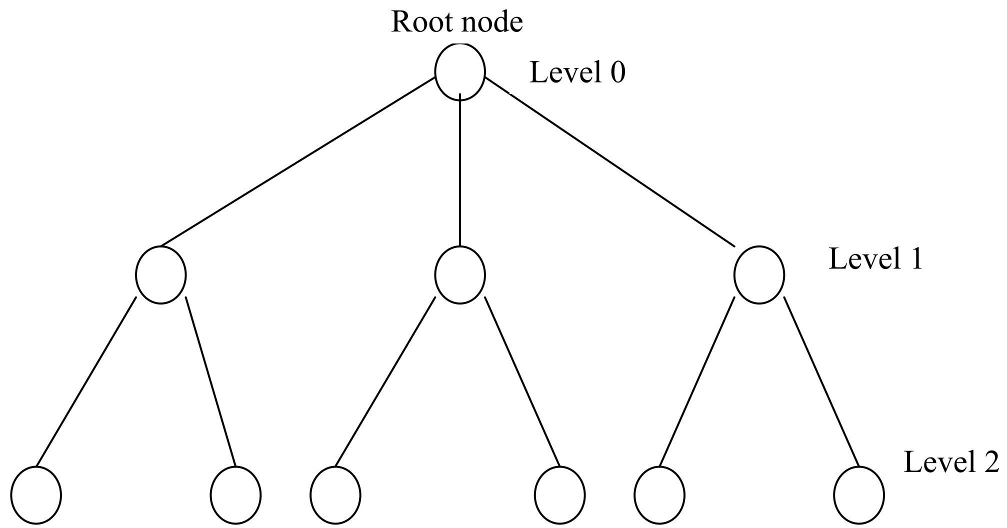


Figure 2.9 Sample clustered tree 2

Now if we divide it, as shown in Figure 2.10 below:

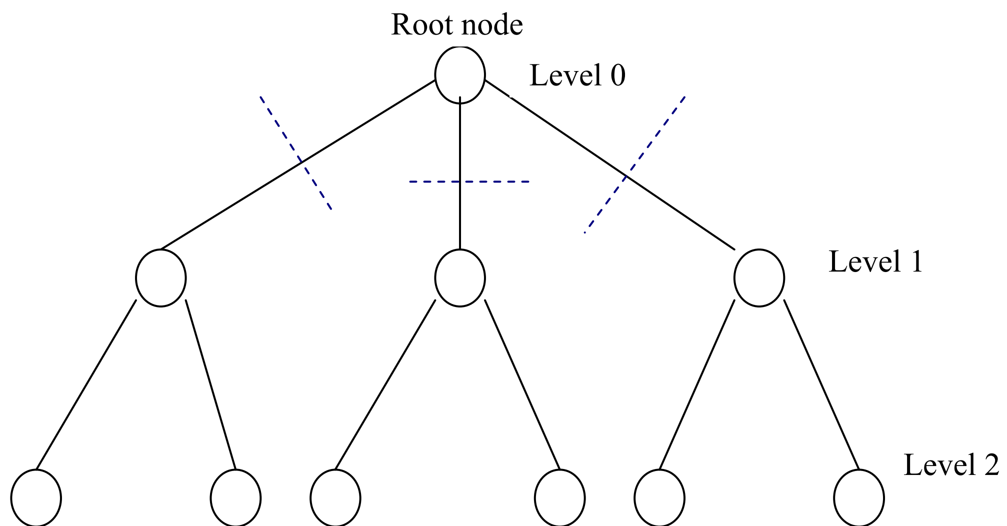


Figure 2.10 Splitting of sample clustered tree 2

Then we have three small automata (the number of automata is equal to the number of children of root node) as shown in Figure 2.11 below:

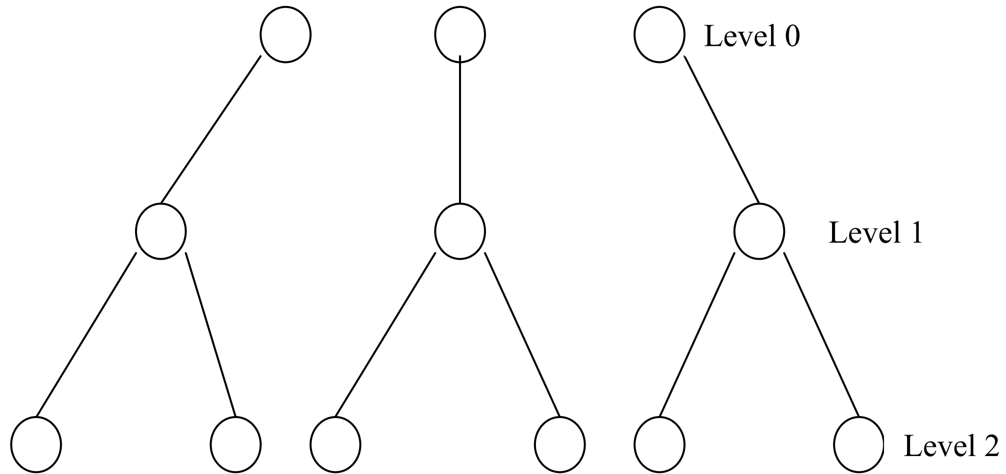


Figure 2.11 After splitting the sample clustered tree 2

Now using (2.1), no. of additional states is given as

$1 + 1 + 1 = 3$

All other attempts of cutting the tree in more than 3 would only result in more additional states as shown in Figure 2.12 below:

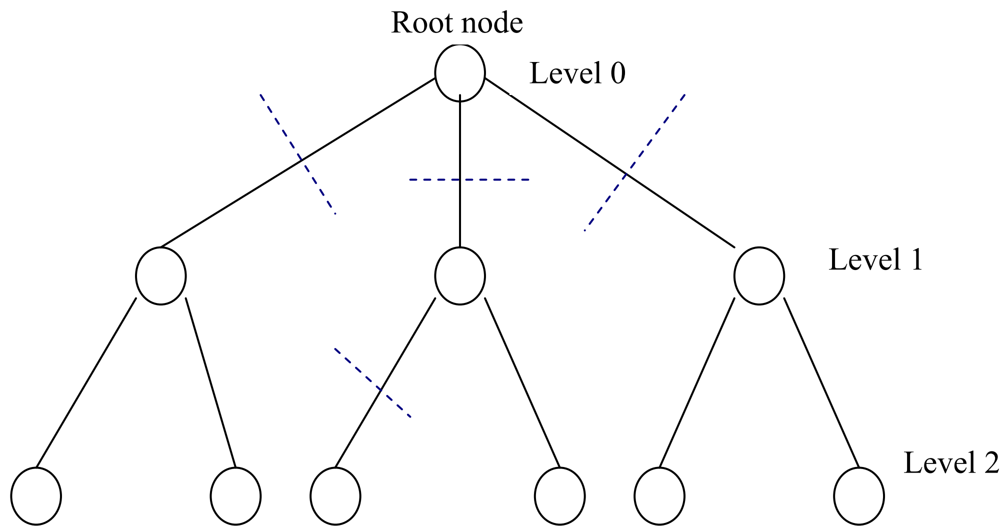


Figure 2.12 Cuts proving the Lemma 3

Using (2.1), no. of extra states after performing the cuts shown in Figure 2.12 is given as 1 + 1 + 1 + 2 = 5.

Hence once can conclude that the number of cuts should be equal to the number of children the root node possess, in order to obtain automata with the minimal number of additional states.

□

Given the size *k* of an automaton which can be handled by a computer, the problem is to divide a single large automaton in such a way that all smaller automata contain no more than *k* states with a minimal number of cuts made.

This problem is essentially the *bin packing problem*, which is NP-hard [24]. In the bin packing problem, objects of different volumes must be packed into a finite number of bins of capacity V in a way that minimizes the number of bins used. There is no

algorithm up-to-date which can provide an optimal solution for this problem in every case. There are only approximation algorithms available.

We need to divide the automaton in such way that all the new automata are of size no more than *k*, containing patterns of different lengths and this should be done with a minimal number of cuts. For example, one such tree is shown in Figure 2.13 below:

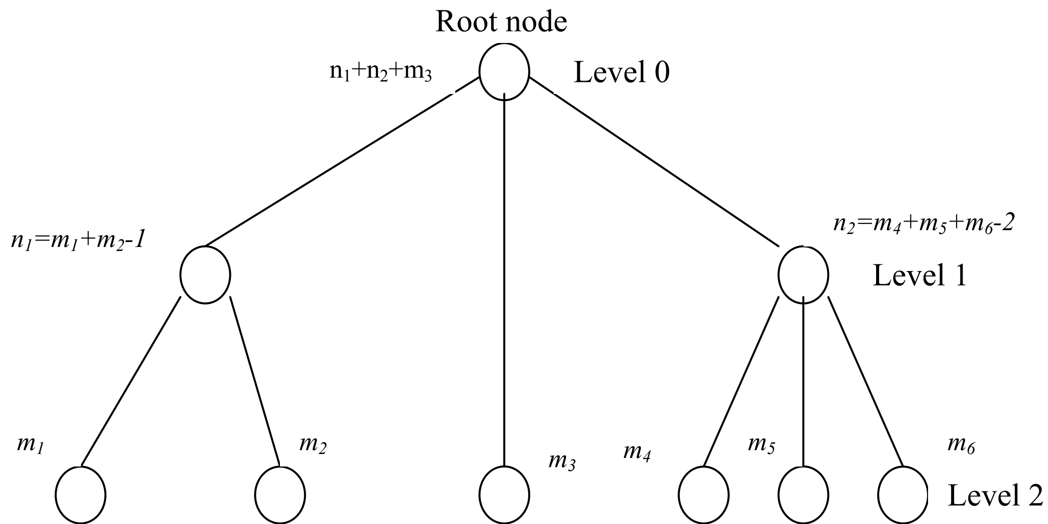


Figure 2.13 Sample clustered tree 3

In Figure 2.13, the leaf nodes (level 2) are the motifs and the nodes at higher levels represent the clusters. In this example, we assumed that the motifs which form clusters at level 1 share a common prefix of length 1 and at level 0 they share a common prefix of length 0. Each node in the tree shown in Figure 2.13 is labeled with the number of states based on the length of motifs and common prefixes between them. This approach

of labeling the nodes provides information such as the number of states which will be resulting, if a cut is made at that instant in the tree. Based on this clustered tree, we can form an automaton and its size will be $n_1+n_2+m_3$, with variables as in Figure 2.13. Now let us assume that a computer cannot handle an automaton of this size. We need to divide the tree in order to form smaller automata, so that each one can be handled and for that we need to make many cuts. So, based on the Lemma 2, we should try to make cuts at level1 (the edges connecting level 0 with level 1). If we cannot achieve the minimal sizes doing this, then we need to keep dividing, and this leads to an NP-hard combinatorial problem. One can argue that shifting a sub-tree from one branch to another can solve this, but this is not true. This is because whenever we perform the shifting of branches, it would result in a sub-optimal assignment.

CHAPTER 3

RESULTS

In this chapter, we shall describe the applications of our algorithms in detail, the source (Flybase) from which we extracted additional information, the input data set of genes and our results.

3.1 Description of the problem

It is common that a biologist is interested in a particular region of a genome and conducts various experiments in the laboratory, in order to analyze the region and determine its role. One of our collaborators, a biologist from UTA, is conducting research on male-specific genes of *Drosophila melanogaster* and she was interested in knowing how often these genes use a 14-base motif [18] that was determined to play an essential role in expressing those genes in the male germ line.

Our collaborator also prepared a list of genes which were the potential candidates for being driven by the 14-base pattern in the upstream region. This list was obtained from [23]. The list also contained FlyBase [40, 41, 31] IDs for each of these genes, which helped us to extract additional information required for processing. The significance of this motif is in that it has been experimentally shown to drive the

expression of testis-specific genes in the male germ line. The exact sequence of this motif is ATCGTAGTAGCCTA.

Our task was to develop computational tools, described in Chapter 2, which efficiently find all occurrences of this 14-base motif in the upstream regions of the genes present in the list. Various studies [18, 14] have also shown that the slight variants of this 14-base motif drive the expression of testis-specific genes in different Drosophila species. Figure 3.1 below explains this:

```
A
D. hyd.:   -53 TGATCGCAGTAGTCTAACTAGGGATATT-26..... + 71
D. mel.:   -53 AAATCGTAGTAGCCTATTTGTGAACATT-26..... +156
```

(i) (Source: F.Michiels et al., 1989 [18])

```
β(2)Tu  ATCGTAGTA.GCCTATTTGTGAACATTC -26
        ********  **** *  * *** ****
Sdic    ATCGTAGTGTGCCTTTGGGGGAA.ATTC -42
                       TA   A     CG
```

(ii) (Source: Nurminsky et al., Nature, 1998)

Figure 3.1 Inclusion of mismatches and gaps in the core 14-base promoter

In Figure 3.1 (i), the motif which drives the expression of testis-specific genes in *Drosophila melanogaster*, also drives the expression in *Drosophila hydei*, but with two mismatches at positions -47 and -41. Hence the study concluded that this 14-base motif can have up to two mismatches without loosing its promoter capacity. Another study conducted at Harvard University [14] showed that this motif can also feature a gap. In

Figure 3.1 (ii) variants of this motif which drive the expression of β2 tubulin and Sdic genes are shown. The '*' symbol denotes the same character in both motifs and a dot in the motif for β2 tubulin gene denotes a gap. The motif which drives the expression of Sdic gene includes two mismatches and an insertion. These studies encouraged us to look for the 14-base motif, with 2 mismatches, 1 insertion (gap) and 1 deletion in it. Since we can have 1-base insertion, we also allowed 1-base deletion.

We illustrate this with another example. Suppose ACTG is the given pattern of interest, and we allow up to 2 mismatches, 1 insertion and 1 deletion in this pattern. Then the following patterns can be derived:

1. 1 mismatch in ACTG – CCTG, GCTG, TCTG (mismatch at position 1), AATG, AGTG, ATTG (mismatch at position 2), ACAG, ACCG, ACGG (mismatch at position 3), ACTA, ACTC, ACTT (mismatch at position 4)
2. 2 mismatches in ACTG – AATG, AGTG, ATTG, CATG, CCTG, CGTG, CTTG, GATG, GCTG, GGTG, GTTG, TATG, TCTG, TGTG, TTTG (1 mismatch at position 1 fixed and the other occur at position 2), ACAG, ACCG, ACGG, CCAG, CCCG, CCGG, CCTG, GCAG, GCCG, GCGG, GCTG, TCAG, TCCG, TCGG, TCTG (1 mismatch at position 1 fixed and the other occur at position 3), ACTA, ACTC, ACTT, CCTA, CCTC, CCTG, CCTT, GCTA, GCTC, GCTG, GCTT, TCTA, TCTC, TCTG, TCTT (1 mismatch at position 1 fixed and the other at position 4), AAAG, AACG, AAGG, AATG, ACAG, ACCG, ACGG, AGAG, AGCG, AGGG, AGTG, ATAG, ATCG, ATGG, ATTG (1 mismatch at position 2 fixed and the other occur at position

3), AATA, AATC, AATG, AATT, ACTA, ACTC, ACTT, AGTA, AGTC, AGTG, AGTT, ATTA, ATTC, ATTG, ATTT (1 mismatch at position 2 fixed and the other occur at position 4), ACAA, ACAC, ACAG, ACAT, ACCA, ACCC, ACCG, ACCT, ACGA, ACGC, ACGG, ACGT, ACTA, ACTC, ACTT (1 mismatch as position 3 fixed and the other occur at position 4).

3. 1 insertion in ACTG – AACTG, CACTG, GACTG, TACTG (insertion at position 1), AACTG, ACCTG, AGCTG, ATCTG (insertion at position 2), ACATG, ACCTG, ACGTG, ACTTG (insertion at position 3), ACTAG, ACTCG, ACTGG, ACTTG (insertion at position 4).
4. 1 deletion in ACTG – CTG (deletion at position 1), ATG (deletion at position 2), ACG (deletion at position 3), ACT (deletion at position 4).
5. 1 mismatch and 1 insertion in ACTG – first generate all patterns shown in 1 and then include 1 insertion (similar to 3).
6. 2 mismatches and 1 insertion in ACTG – first generate all the patterns shown in 2 and then include 1 insertion (similar to 3).
7. 1 mismatch and 1 deletion in ACTG – generate all patterns shown in 1 and then include 1 deletion (similar to 4).
8. 2 mismatches and 1 deletion in ACTG – generate all patterns shown in 2 and then include 1 deletion (similar to 4).

Many of these patterns (such as CCTG, GCTG or TCTG) are repeated. Even after eliminating the duplicates, which were necessary in case of naïve approach to pattern matching (for sanity check), the number of patterns was still very large. The

task was to find an efficient algorithm, which can easily handle them. This algorithm has already been described in Chapter 2.

3.2 Data source

In order to perform the matching, we needed to obtain additional information about each of the genes in the list [23] provided by our collaborator. In particular, we had to extract the upstream regions of these genes, as precisely as possible. Since our initial list contained Flybase IDs for each gene, we have decided to use FlyBase [http://www.flybase.org/] as our source of additional information. Flybase is one of the most popular Drosophila repositories, and it contains sequences deposited by scientists from all over the world. It is maintained by a consortium of Drosophila researchers located at Harvard University, Cambridge University (UK), Indiana University and the National Center for Biotechnology Information (NCBI), USA. It contains the complete genome sequence of Drosophila and it is updated whenever new genes are found and annotated.

Flybase contains information about the genes, the expression and properties of transcripts and proteins, functions of gene products, chromosomal aberrations, genomic clones, etc. It incorporates various visualization tools, which provides different views of the genomic regions. One can access the information stored in Flybase by providing a Flybase ID, gene name, chromosomal location, or other keywords associated with the item of interest. One can easily retrieve a particular region of the genome by providing its start and end, for instance.

In our study, we needed the upstream regions to check for the occurrence of a 14-base motif we were interested in. As mentioned earlier, in order to extract the upstream region of a gene, we need to know its start and end, transcription and translation data, number of mRNAs, length of its 5’ UTR and its orientation (either original or complementary strand). The upstream (regulatory) region of a gene lies immediately before its 5’ UTR. Using Flybase, we could obtain all this information.

3.3 Input data set

Our collaborator has provided a list of putative target genes for our analysis, but we still needed to extend it. Two screen shots of our extended list are shown in Table 3.1 and Table 3.2 below (Table 3.2 is the continuation of Table 3.1).

Table 3.1 Information about the Genes Screen Shot 1

Gene Name	GPL20 ID (1st)	GPL20 ID (2nd)	LocusLink ID	FLYBASE ID	Status	No. of mRNA	Length of 5'UTR	Reason For not locating
Acp76A	9758	10434	40078	FBgn0015586	No	1	0bp	No 5'UTR
Actn3	7986	8662	44964	FBgn0015008	Yes	1	41bp	
ACXC	12499	13159	34689	FBgn0040508	Yes	1	59bp	
Adgf-B	608	1264	39975	FBgn0036751	Yes	1	95bp	
Adgf-E	3090	3744	36627	FBgn0033952	Yes	1	90bp	
alpha-Est3	9455	10131	40907	FBgn0015571	Yes	1	118bp	
BcDNA:GH02220	16259	16915	39222	FBgn0027615	Yes	2	276bp	
BcDNA:GH06032	8277	8953	34546	FBgn0027582	Yes	1	143bp	
BcDNA:GH07774	16291	16947	34403	FBgn0027568	Yes	1	196bp	
BcDNA:GH10229	25228	25904	38886	FBgn0027554	Yes	2	107bp	
BcDNA:GH11023	19902	20558	38897	FBgn0027549	Yes	1	236bp	
BcDNA:GH24095	17304	17960	41376	FBgn0046225	no	2	315bp, 124bp	Two 5'UTR
BcDNA:LD23587	5841	6517	42806	FBgn0028475	Yes	1	500bp	
BcDNA:LD28247	21676	22352	32042	FBgn0027498	no	3	287bp, 226bp, 113bp	Three 5'UTR
BEST:GH09435	25524	26200	50456	FBgn0045827	Yes	1	444bp	
BEST:GH11908	8118	8794	50130	FBgn0046297	Yes	2	143bp	
BEST:LD13681	12130	12790	47729	FBgn0026147	no	3	594bp, 549bp, 534bp	Three 5'UTR
betaTub85D	25515	26191	41124	FBgn0003889	Yes	1	230bp	
BG:BACR44L22.8	11286	11946	34914	FBgn0028944	Yes	1	9bp	
BG:DS00365.1	12450	13110	34935	FBgn0028935	no	1	0bp	No 5'UTR
BG:DS00464.1	15010	15666	40837	FBgn0026563	Yes	1	92bp	
BG:DS01068.1	9503	10179	34833	FBgn0028918	Yes	1	40bp	
BG:DS01068.11	9511	10187	34834	FBgn0028531	Yes	1	161bp	
BG:DS02252.1	17308	17964	34933	FBgn0028903	Yes	1	24bp	
BG:DS02740.5	7004	7680	34958	FBgn0028896	no	1	0bp	No 5'UTR
BG:DS03023.2	19164	19820	34907	FBgn0028892	Yes	1	93bp	
BG:DS04095.1	10799	11459	34948	FBgn0028520	Yes	1	27bp	
BG:DS04095.3	21255	21931	50457	FBgn0028883	Yes	1	0bp	No 5'UTR
BG:DS04095.3	30778	31246	50457	FBgn0028883	no			same reason,whether sameas29
BG:DS04641.8	23146	23822	34848	FBgn0028881	Yes	1	122bp	
BG:DS06874.2	13296	13954	34857	FBgn0028869	no			Data not found,FBgn0028838 in fly
BG:DS06874.3	5521	6197	34858	FBgn0028868	Yes	1	99bp	
BG:DS07486.2	6892	7568	34944	FBgn0028858	Yes	1	12bp	
BG:DS07486.4	17318	17974	34942	FBgn0028857	no	2	0bp	No 5'UTR

In Table 3.1, the "Status" column provides information whether the gene was found in Flybase or not, and if the upstream region could not have been uniquely located, the reason for that.

Table 3.2 Information about the Genes Screen Shot 2

Information about Location	Gene Starts	Gene Ends	Orientation	Chromosome	Length of the Gene
Same mRNA, CDS positions on - strand	19016050	19017285	Comp. Strand	3L	1236bp
Diffe. mRNA, CDS positions on - strand	12801620	12803382	Comp. Strand	3R	1763bp
Diff. mRNA, CDS positions on + strand	12912530	12916774	Orign. Strand	2L	4245bp
Diffe. mRNA, CDS positions on - strand	17714092	17715847	Comp. Strand	3L	1756bp
Diffe. mRNA, CDS positions on - strand	10171661	10173618	Comp. Strand	2R	1958bp
Diffe. mRNA, CDS positions on - strand	3364694	3367349	Comp. Strand	3R	2656bp
Diffe. mRNA, CDS positions on - strand	10858208	10860418	Comp. Strand	3L	2211bp
Diffe. mRNA, CDS positions on - strand	11115636	11120314	Comp. Strand	2L	4679bp
Diffe. mRNA, CDS positions on - strand	10358162	10362703	Comp. Strand	2L	4542bp
Diffe. mRNA, CDS positions on - strand	7946396	7948955	Comp. Strand	3L	2560bp
Diff. mRNA, CDS positions on + strand	8084210	8087274	Orign. Strand	3L	3065bp
Diff. mRNA, CDS positions on + strand	7263507	7291573	Orign. Strand	3R	28067bp
Diffe. mRNA, CDS positions on - strand	19580343	19583860	Comp. Strand	3R	3518bp
Diff. mRNA, CDS positions on + strand	10937711	10939882	Orign. Strand	X	2172bp
Diff. mRNA, CDS positions on + strand	16000501	16001396	Orign. Strand	2L	896bp
Diff. mRNA, CDS positions on + strand	17311006	17311923	Orign. Strand	2R	918bp
Diff. mRNA, CDS positions on + strand	10989305	10995024	Orign. Strand	2L	5720bp
Diffe. mRNA, CDS positions on - strand	5234075	5236004	Comp. Strand	3R	1930bp
Diffe. mRNA, CDS positions on - strand	15608419	15609264	Comp. Strand	2L	846bp
Same mRNA, CDS positions on - strand	15873544	15875994	Comp. Strand	2L	2451bp
Diff. mRNA, CDS positions on + strand	2880288	2884393	Orign. Strand	3R	4106bp
Diff. mRNA, CDS positions on + strand	14328429	14333867	Orign. Strand	2L	5439bp
Diff. mRNA, CDS positions on + strand	14338240	14340080	Orign. Strand	2L	1841bp
Diff. mRNA, CDS positions on + strand	15848602	15849993	Orign. Strand	2L	1392bp
Diff. mRNA, CDS positions on + strand	16287213	16287885	Orign. Strand	2L	673bp
Diff. mRNA, CDS positions on + strand	15429291	15431179	Orign. Strand	2L	1889bp
Diff. mRNA, CDS positions on + strand	16090938	16091873	Orign. Strand	2L	936bp
Same mRNA, CDS positions on - strand	16130636	16133442	Comp. Strand	2L	2807bp
Diffe. mRNA, CDS positions on - strand	14547520	14549075	Comp. Strand	2L	1556bp
conflicting flybase ids.(Ensembl,flybase)					
Diff. mRNA, CDS positions on + strand	14761837	14763153	Orign. Strand	2L	1317bp
Diff. mRNA, CDS positions on + strand	15975489	15976049	Orign. Strand	2L	561bp
Diff. mRNA, CDS positions on + strand	15937435	15940577	Orign. Strand	2L	3143bp

In Table 3.2, the column "Information about Location" describes the location of the gene along with positions of mRNA and CDS (Coding Sequence, or the translated sequence).

Based on the above information for each gene, we classified them into different categories such as these having more than one mRNA, genes with no 5' UTR recorded,

or genes not found in Flybase. A screen shot of the classification summary is shown in Table 3.3 below:

Table 3.3 Classification of Genes Based on the Extracted Information

CATEGORIES	NAME	NO. OF GENES
1	SEQUENCE EXTRACTED	543
2	GENES WITH NO 5` UTR	82
3	GENES WITH MORE THAN ONE MRNA	118
4	GENES WITH CONFLICTING FLYBASE IDS.	18
5	GENES IMMEDIATELY BEFORE/AFTER ANOTHER GENE	14
6	GENES NOT FOUND IN THE DATABASE	2
7	GENES WITH TWO ENTRIES IN THE ORIGINAL SHEET	12
8	GENES PRESENT WITHOUT EXACT SEQENCE DEFINITION	2
	SHORTEST 5' UTR REGION	2bp
	LONGEST 5' UTR REGION	2219bp
	HIGHEST NUMBER OF MRNAS	7
	TOTAL NUMBER OF GENES	791

However, the gene-by-gene approach worked poorly. Not only that it was labor-intensive, but it was also problematic. For example, the problem with the second category in Figure 3.3 was in that since there was no 5’ UTR recorded we could not correctly locate the upstream region. All the genes in category 3 had more than one mRNA and hence more than one recorded 5’ UTR. This led to an ambiguous situation about which 5’ UTR should be taken into consideration for determining the upstream region. Genes in category 4 had conflicting Flybase IDs. Genes in category 5 immediately start after another gene, due to which it was not possible to extract their upstream regions. Genes in category 6 were not found in Flybase. Category 7 represented genes with multiple entries. Genes in category 8 were found, but had no exact sequence definition.

Looking at the category 1 genes we could divide them into three types based on the length of the upstream region which could have been extracted (without coliding

with another gene). These categories were (1.3) less than or equal to 100bp, (1.2) between 100 -1000bp and (1.1) 1000bp. This classification of category 1 genes is shown in Table 3.4 below.

Table 3.4 Classification of Category 1 Genes

Category	Name	Number of genes
1	Upstream sequence extracted	543
1.1	Genes having clear upstream region of length 1000bases	363
1.2	Genes having clear upstream region of length between 100-1000bases	150
1.3	Genes having clear upstream region of length up to 100bases	30

3.4 Results

In order to avoid extracting the upstream region for every gene in the given list and avoid the problems described in the previous section, we looked at the complete genome of Drosophila. We divided the file containing the genome into 6 smaller files, one for each chromosomal arm of Drosophila. We also classified the given list of genes into six categories based upon the chromosomes on which they were located. Table 3.5 below shows the results obtained for X chromosomal arm of Drosophila (original hits) and also for the genes from our list which are located on the X chromosomal arm

(reduced hits were obtained from the original hits, by considering the region from -1000 bases upstream to +1000 bases downstream).

Table 3.5 Results for the Entire Genome and Genes of Interest

	Total No.of Genes found in FLYBASE	755						
	Total seq. length of genes found	2039731 bp						
	Total seq. region considered	3549731 bp	(Total seq. length +upstream (1000) +down stream (1000))					
	Genes stored in the file	Genes.txt						
SNO	Case under consideration	Chromosome	Input file	Original hits	Output file	Reduced hits	Genes	Genes repeated
1	1 mismatch	X	op1.txt	0	geneop1.txt	0		
2	2 mismatches	X	op2.txt	31	geneop2.txt	1	Tob	Yes
3	1 insertion	X	op3.txt	0	geneop3.txt	0		
4	1 mismatch and 1 insertion	X	op4.txt	5	geneop4.txt	0		
5	2 mismatches and 1 insertion	X	op5.txt	216	geneop5.txt	3	Tob	1 time
							CG1835	
							CG1503	
6	1 deletion	X	op6.txt	1	geneop6.txt	0		
7	1 mismatch and 1 deletion	X	op7.txt	67	geneop7.txt	2	CG32548	Yes
							CG1722	Yes
8	2 mismatches and 1 deletion	X	op8.txt	1162	geneop8.txt	24	ptr	
							EG:155E2.5	
							CG1958	
							CG15035	
							CG11369	
							CG12118	
							CG15306	Yes
							CG15306	1 time
							CG15196	
							CG15728	
							CG15742	
							CG32601	
							Pp1-13C	
							Tob	2 time
							Tob	3 time
							Tob	4 time
							CG12992	
							OdsH	
							CG32548	1 time
							CG32548	2 time
							CG32548	3 time
							CG15040	
							CG17003	
							CG1722	1 time

We have obtained the results for the remaining 5 chromosomal arms of Drosophila in the same way. The number of patterns under consideration was 38,783 and the number of genes involved was 755. The total number of hits in the entire genome was 7957 (original hits). The number of hits found in the vicinity of 755 genes was 277 (reduced hits).

We further classified these 277 hits into four categories, depending on their location. These categories were:

1. Category 1 – hits which were found in the broader upstream region (-1000 to -200 bases, with respect to the start of the gene).
2. Category 2 – hits which were found in the putative promoter region (-200 to +50 bases).
3. Category 3 – hits which were found within the gene (gene start – gene end).
4. Category 4 – hits which were found in the downstream region of the gene (between the gene end and +1000 bases downstream).

A screen shot of this categorization is shown in Table 3.6 below.

Table 3.6 Classification of Reduced Hits

Genes	Gene Starts	Gene Ends	Category1	Category2	Category3	Category4	Trancript. Start	Translat. start
Adgf-E	10173618	10171661			1		10173618	10173529
BcDNA:GH02220	10860418	10858208		1			10860238	10859932
BcDNA:GH24095	7263507	7291573	1				7263507	7289784
betaTub85D	5236004	5234075	3				5236004	5235774
BG:DS00365.1	15875994	15873544			3		15875994	15875994
BG:DS01068.1	14328429	14333867	1				14328429	14328469
BG:DS04095.3	16133442	16130636			1		16133442	16133442
BG:DS04641.8	14549075	14547520			1		14549075	14548953
BG:DS07486.4	15937435	15940577			1		15937435	15937435
BG:DS07660.1	13597186	13595502			1		13597186	13596997
blue	14015151	14008945			1		14015151	14014928
CdGAPr	19787829	19771021			1		19787829	19778721
CG10014	8048109	8049770	1				8048109	8048205
CG10191	13363935	13362240			1		13363935	13363819
CG10383	18691351	18686831			1	2	18691351	18691053
CG10407	11871693	11870511			1		11871693	11871631
CG10694	19933113	19934068			2		19933113	19933171
CG10947	20386987	20373649				1	20386987	20375124
CG10984	12724066	12727349			1		12724455	12724554
CG11369	7419834	7421837			1		7419834	7419834
CG11526	3312403	3305806			2		3312403	3312401
CG11634	21903954	21905041				1	21903954	21903998
CG11896	12975129	12971174				1	12975129	12974616
CG11983	4834157	4835714	1				4834157	4834308
CG12034	3202656	3204460	1				3202656	3202836
CG12118	9041682	9039963		1			9041682	9041323
CG12169	564869	566342		2			564869	565095
CG12229	17401690	17399731			2		17401690	17401506
CG12963	11071447	11077309				2	11071447	11076605
CG12992	17424108	17425485				1	17424108	17424139
CG13039	16292021	16291511			1		16292021	16291940
CG13088	8488985	8490548			1		8488985	8489031
CG13280	16817022	16797950			3		16817022	16816935
CG13322	8452434	8450134			1		8452434	8451860

We were primarily interested in the category 2 hits (-200bases to +50bases relative to the gene start), because a promoter should be very close to the Transcription Start Site (TSS). Moreover, studies have shown that the promoter becomes less active with the increase in its distance from the TSS. The list of genes which belong to category 2 hits is shown in Table 3.7 below, and the exact positions of the matches we have found in this category are shown in Figure 3.2.

Table 3.7 List of Genes Containing an Approximate Match to the Putative Promoter in the Target Region (-200, +50).

Category2 Hits	Case Under Consideration	Chromosome
BcDNA:GH02220	2 mismatches and 1 deletion	3L
CG12118	2 mismatches and 1 deletion	X
CG12169	2 mismatches and 1 insertion	3L
CG12169	2 mismatches and 1 deletion	3L
CG14034	2 mismatches and 1 deletion	2L
CG14305	2 mismatches and 1 deletion	3R
CG15040	2 mismatches and 1 deletion	X
CG15510	2 mismatches and 1 deletion	3R
CG1722	1 mismatch and 1 deletion	X
CG1722	2 mismatches and 1 deletion	X
CG17302	2 mismatches and 1 deletion	2L
CG1835	2 mismatches and 1 insertion	X
CG18810	2 mismatches and 1 deletion	2L
CG31773	2 mismatches and 1 deletion	2L
CG31960	2 mismatches and 1 deletion	2L
CG5048	2 mismatches and 1 insertion	3L
CG5048	1 mismatch and 1 deletion	3L
CG5048	2 mismatches and 1 deletion	3L
CG5048	2 mismatches and 1 deletion	3L
CG5207	2 mismatches and 1 deletion	3R
CG5458	2 mismatches and 1 insertion	2L
CG6130	2 mismatches and 1 deletion	3R
CG8298	2 mismatches and 1 deletion	2R
EG:155E2.5	2 mismatches and 1 deletion	X
Glut3	2 mismatches	3R
Glut3	2 mismatches and 1 insertion	3R
Glut3	2 mismatches and 1 deletion	3R
Glut3	2 mismatches and 1 deletion	3R
Mst84Db	2 mismatches and 1 deletion	3R
robl62A	2 mismatches and 1 deletion	3L
Total no. of hits	30	

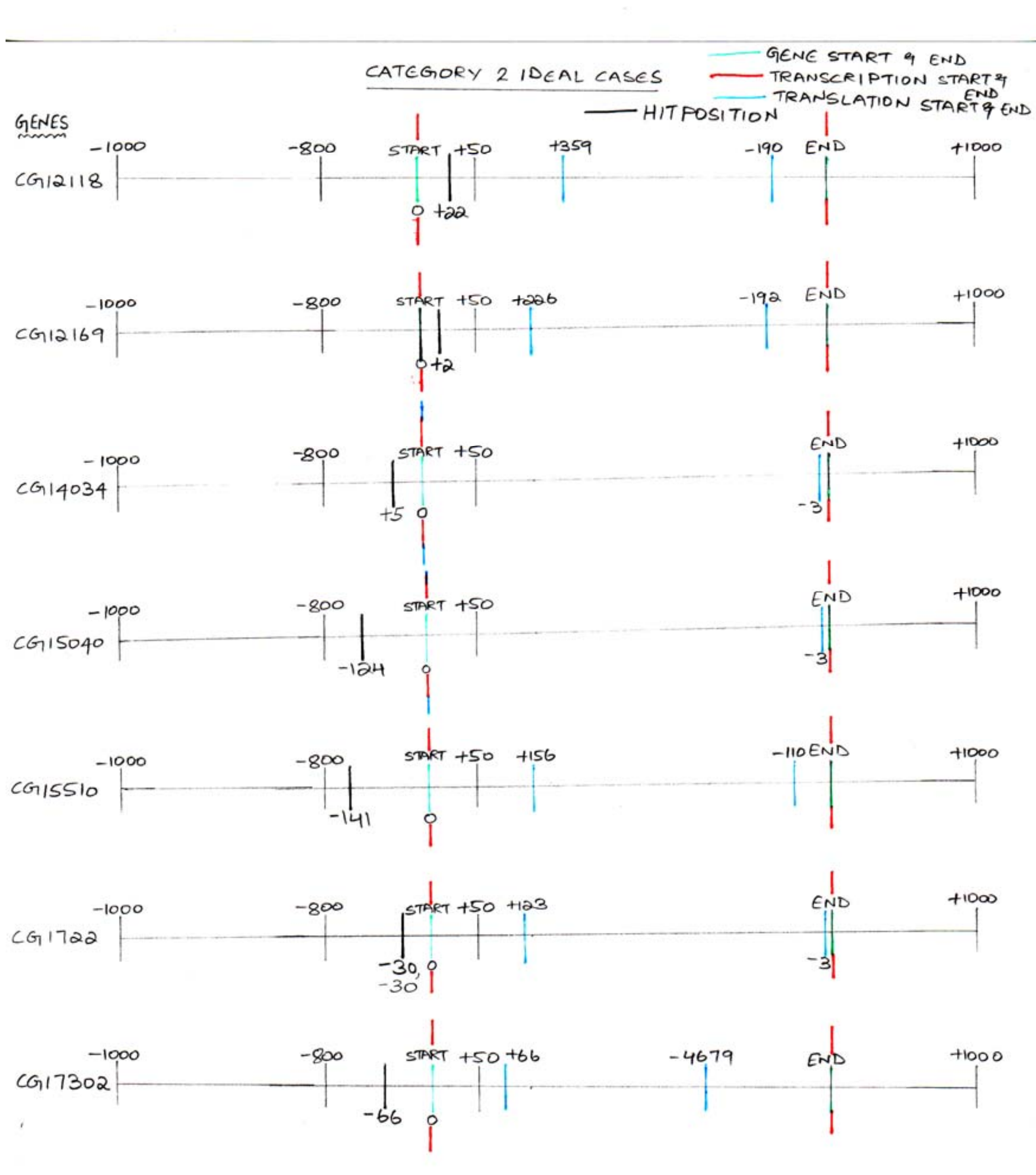


Figure 3.2 Positions of approximate matches to the putative promoter.

In order to check whether these hits were significant we have calculated the expected number of hits in the entire genome and near the genes of interest for our study. The rough estimates, based on the naïve model of equal base probabilities, are as follows. Total number of patterns generated = 38,783. These patterns can be divided into three categories based on their lengths. They are:

Number of patterns of length 13 = 7952

Number of patterns of length 14 = 905

Number of patterns of length 15 = 29926

The expected number of hits in the entire genome can be obtained by adding the expected number of hits obtained in each of these categories. The total amount of sequence considered (a large fraction of the entire Drosophila genome) was 118,357,599 bases. The total expected number of hits for all patterns of length 13 was thus $7952 * (1 / 4^{13}) * 118357599 = 14025$; for patterns of length 14 it was $905 * (1 / 4^{14}) * 118357599 = 399$; and for patterns of length 15 it was $29926 * (1 / 4^{15}) * 118357599 = 3299$. Hence, the total number of hits in the entire considered part of Drosophila genome was 17723. However, the number of hits which we have observed for all patterns in the entire sequence was only 7957. Although this may appear paradoxical, the genome of Drosophila is biased towards some patterns, due to which the total number of observed hits is heavily skewed depending on the A-T content of the patterns. Because of this phenomenon, we have used our actual number of hits in order to estimate the likelihood of having a chance one near the genes of interest. Denoting the total length of all neighborhoods of genes from our list by *N*, and the expected

number of hits in this region by X, we get 118357599/7957 = N/X, with N was approximately 3549731. It follows that X should be around 239. The number of hits we observed near our genes of interest was 277. Using the same formula, we calculated the expected the number of hits within the upstream regions of 22 genes where we had a category 2 hit, and it was around 15. However, we have observed 30 hits, which is twice the expected number. This provides a strong indication that the 14-base motif of interest has a functional (i.e. non-random) association with our genes of interest, due to which it has been seen more often than expected. In order to confirm this, we have generated a control set of genes. It consisted of 250 genes, which we selected randomly from the *Drosophila melanogaster* genome. A screen shot showing these genes is shown in Table 3.8 below:

Table 3.8 Control Set of Genes

Gene Name	Gene Start	Gene End	Orientation	Chromosomal arm	Length of the gene
arc	17644571	17680423	Orig. Strand	2R	35853bp
abrupt	11210681	11261081	Orig. Strand	2L	50401bp
achaete	226853	227813	Orig. Strand	X	961bp
adenosine2	6045970	6041178	Comp. Strand	2L	4793bp
aristaless	378116	387439	Orig. Strand	2L	9324bp
Aldolase	22087313	22080404	Comp. Strand	3R	6910bp
arrow	8999125	8969540	Comp. Strand	2R	29586bp
Arrestin1	18074195	18076352	Orig. Strand	2L	2158bp
Arrestin2	8622883	8621214	Comp. Strand	3L	1670bp
asense	317322	320008	Orig. Strand	X	2687bp
black	13821203	13823894	Orig. Strand	2L	2692bp
Distal-less	20322430	20342763	Orig. Strand	2R	20334bp
bagofmarbles	21070884	21068761	Comp. Strand	3R	2124bp
bazooka	16994876	17029254	Orig. Strand	X	34379bp
Blackcells	13404835	13402076	Comp. Strand	2R	2760bp
bicoid	2585199	2581581	Comp. Strand	3R	3619bp
belle	4486483	4481292	Comp. Strand	3R	5192bp
bendless	13830291	13833745	Orig. Strand	X	3455bp
bifid	4258093	4329701	Orig. Strand	X	71609bp
bigbrain	9984647	9995545	Orig. Strand	2L	10899bp
bicaudal	8386144	8388010	Orig. Strand	2R	1867bp
basket	10250479	10247485	Comp. Strand	2L	2995bp
buttonhead	9539409	9542794	Orig. Strand	X	3386bp
burgundy	21167290	21161976	Comp. Strand	2L	5315bp
brown	19046093	19035405	Comp. Strand	2R	10689bp
Beadex	18368602	18405653	Orig. Strand	X	37052bp
cactus	16322030	16308958	Comp. Strand	2L	13073bp
caudal	20758328	20770735	Orig. Strand	2L	12408bp
Calmodulin	7775315	7790532	Orig. Strand	2R	15218bp

Using this information, we have considered the control set of genes instead of the original list. We filtered out the hits which did not fall in the region, 1000 bases (upstream) to +1000 (downstream) with respect to the orientation of each of the genes in our control set. The number of hits observed now was 264, which was again more than the expected number. However, we were interested only in category 2 hits (+50 to -200 bases), so we further filtered the hits which did not fall in this category. Table 3.9 below shows the genes, based on the hits in this category.

Table 3.9 Results for the Control Set of Genes

Category2 Genes	No. of Hits
shakingB	1
Syntaxin5	1
shadow	1
Total no. of hits	**3**

The observed number was 3, and it was much smaller than expected. This indicates that there indeed exists a non-random bias within our genes of interest.

One can argue that if a gene in the control set overlaps or has some influence on a gene in the original list, it would not provide a good measure. In order to address this issue, we calculated the percentage of expected overlap below, as

(Number of genes in the given list / Total number of genes in Drosophila genome) * 100 = (791 / 13600) * 100 = 5.81%

Hence there is a relatively small possibility that a gene from the control set overlaps or has an influence on a gene from the original list.

In order to confirm that the Drosophila genome is biased towards some patterns, we have done a cross check by considering seven sample patterns, each of length 14

bases and repeated the whole process of determining the hits in the entire genome. The results are shown below:

1. ATTTAAGATATTAC (AT rich pattern) has been found 54576 times.

2. CCCAGGGCCTGGGC (GC rich pattern) has been found 7857 times.

3. ATGCGGTACCATTC (A, C, G, T evenly distributed) has been found 9079 times.

4. CGATTCGTTACGTT (3 CpG islands) has been found 15030 times.

5. ATCGTTAAGGCCTA (1 CpG island) has been found 7796 times.

6. TATTCATCGTAAAG (5 A's, 5 T's and remaining C, G) has been found 18229 times.

7. TATTTAATAAATTA (No C, G in the pattern) has been found = 166537 times.

The goal of this experimentation was to confirm the bias in the number of hits, genome-wide, depending on the pattern's A and T content.

From the above results it is clear that the genome of *Drosophila melanogaster* is more biased to A's and T's, compared to G's and C's. This comes as no surprise, as it is well known that the genome of *Drosophila melanogaster* consists of 60% AT and 40% GC [9]. In the test cases above most sequence was intergenic, which is especially AT rich.

CHAPTER 4

DISCUSSION

In this chapter, we discuss some of the possible improvements which can be made to our algorithms, possible extensions of the described methods to solve other similar problems in bioinformatics or computer science in general.

4.1 Algorithmic improvements

The overall performance of the existing algorithm is good, but it can still be improved. Most of the times while constructing the automaton we come across situations like this

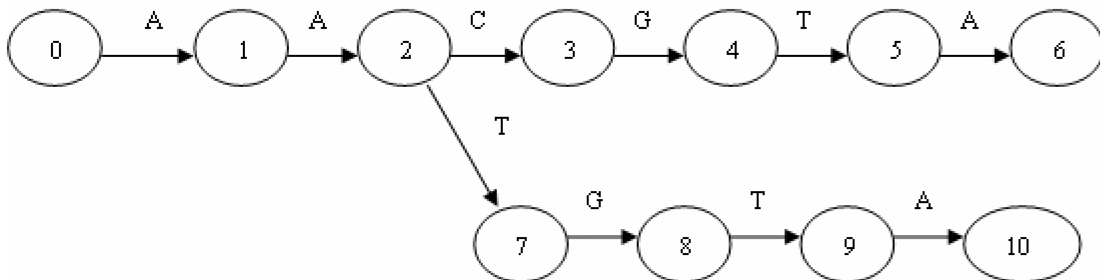


Figure 4.1 Common automaton construction scenario

Where the patterns share common prefix (AA in the above example) and a common suffix (GTA in the above example). In this case, a single character difference leads to many new states. In terms of bioinformatics, one can relate this difference to an SNP. Instead of making a transition to state 8 on reading G we can make a transition to state 4 again and whenever a hit is found at state 6 (which contains both patterns as labels), we backtrack to see whether it included state 7 in the pathway. If it does, then we can say that pattern 2 is found or else pattern 1 is found. A schematic representation of this is shown in Figure 4.2.

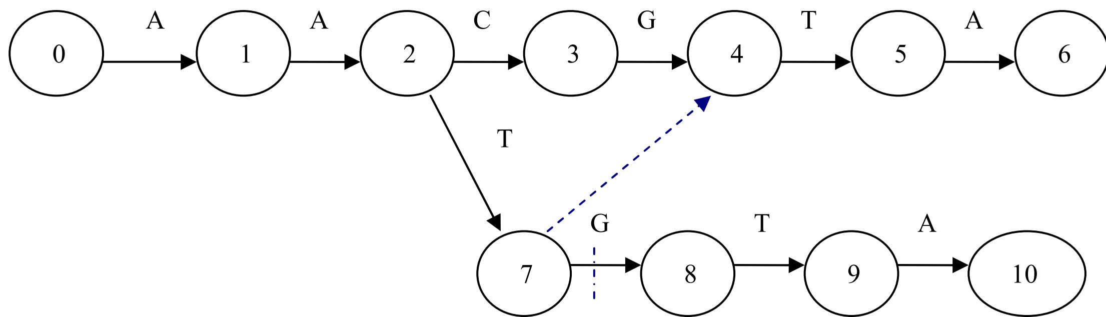


Figure 4.2 Reduction of size of the automaton while constructing

The above concept changes the automaton and makes it smaller when compared to the one constructed previously (in Figure 4.1). The searching stage of the Aho-Corasick algorithm would remain similar. After the application of this concept the automaton would look like one shown in Figure 4.3.

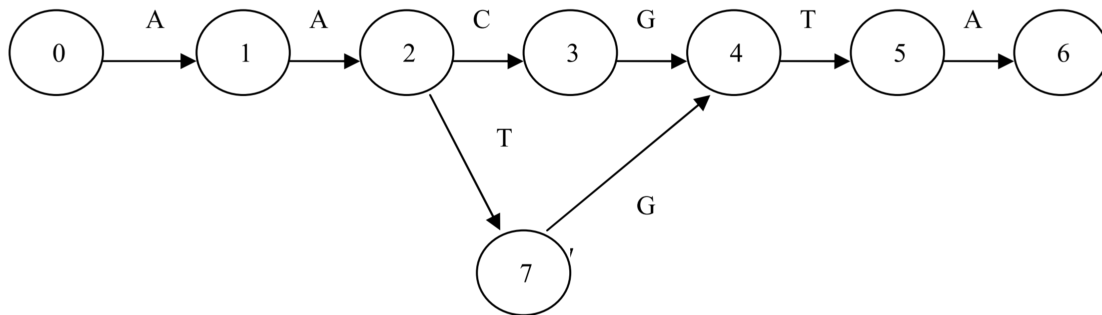


Figure 4.3 Automaton after improvement

The number of states in this example got reduced from 11 to 8. Even though it looks small in this short example, we are sure that if we would be able to apply this concept in our case the number of states would get reduced significantly.

Another possible improvement to the algorithm can be made by replacing the clustering scheme, in order to locate patterns with common long prefixes more efficiently.

4.2 Possible extensions of the described methods to solve other problems

The methods described in this thesis can be applied to many problems in bioinformatics and also to other problems in general computing. Generating all possible patterns from a given consensus can be applied to any pattern of interest and it can be of any length. Further, it can also be extended to handle more mismatches, insertions and deletions. Since substitutions, insertions and deletions are common in DNA sequences, the method described in this thesis can be of interest.

The method of reducing duplicates is non-trivial, but it can solve many problems where a duplication needs to be avoided. The Aho-Corasick algorithm can be applied for matching other patterns of interest, like the enhancers, silencers or other regulatory motifs. It can also be applied to extending the *find* feature of Microsoft Word, Notepad, etc. so that multiple patterns can be simultaneously located in the opened document, instead of just one at a time.

Our system on the whole can also be applied to amino acid sequences, by replacing the four letter DNA alphabet with the 20 letters of amino acids alphabet and incorporating the additional complexities of handling mismatches, such as these introduced by PAM [22] and BLOSUM [36] matrices. If one has more free processors, the distributed system described in this thesis can easily be extended to any number of them. This can be achieved by further division of the original jobs. One just has to write a few more shell scripts, depending on the number of available processors.

A GUI (Graphical User Interface) can be added to our system, possibly web-based, which would take a motif as user input and perform all other operations in the background. The hits can be redisplayed as shown in Figure 3.2.

With many more applications and extensions of these methods possible, we can conclude that the overall system can be useful for various types of research, not only in bioinformatics, but also in the general scientific world.

REFERENCES

1. A. Amir, M. Lewenstein and E. Porat, *Faster algorithms for string matching with k mismatches*, In proceedings of 11th Association for Computing Machinery – SIAM Symposium on Discrete Algorithms (SODA), 794-803, 2000.
2. A. Santel, J. Kaufmann, R. Hyland and R. Renkawitz-Pohl, *The initiator element of the Drosophila ß2 tubulin gene core promoter contributes to gene expression in vivo but is not required for male germ-cell specific expression*, Nucleic Acids Research, 28(6): 1439-1446, 2000.
3. A.K. Jain, R.P.W. Duin, and J. Mao, *Statistical pattern recognition: A review*, IEEE Transactions on Pattern Analysis and machine Intelligence, 22(1): 4-37, 2000.
4. A.V. Aho and M.J. Corasick, *Efficient String Matching: An Aid to Bibliographic Search,* Journal of the Association for Computing Machinery, 18(6): 333-340, 1975.
5. A. Vaidya, *A Pseudogene Detection System Based on a High-Performance Computing Platform*, Master's Thesis, UT Arlington, 2005.
6. B.J. Oommen and R.k.S.Loke, *Pattern recognition of strings containing traditional and generalized transposition errors*, In Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, 2: 1154-1159, 1995.
7. B. Lewin, Genes VIII, Prentice-Hall, Inc., 2004.
8. B. Parhmi, Introduction to Parallel Processing, Plenum Press, 1992.

9. B.T. Wakimoto, *Doubling the Rewards: Testis ESTs for Drosophila Gene Discovery and Spermatogenesis Expression Profile Analysis*, Genome Research, 10(12): 1841-1842, 2000.

10. C. Burge and S. Karlin, *Prediction of complete gene structures in human genomic DNA,* Journal of Molecular Biology, 268: 78-94, 1997.

11. D.E. Knuth, J.H. Morris, and V.R. Pratt, *Fast pattern matching in strings*. SIAM Journal of Computing, 6: 323-350, 1977.

12. D. Casey, Primer on Molecular Genetics, DOE Human Genome program, 1992

13. D. Gusfield, Algorithms on strings, trees, and sequences: computer science and computational biology, Cambridge University Press, 1997.

14. DI Nurminsky, MV Nurminskaya, De Aguiar, DL Hartl, *Selective sweep of a newly evolved sperm-specific gene in Drosophila,* Nature, 396(6711): 572-575, 1998.

15. D.M. Sunday, *A very fast substring search algorithm*, Journal of the Association for Computing Machinery, 33(8): 132-142, 1990.

16. D.W. Mount, Bioinformatics: Sequence and Genome Analysis, Cold Spring Harbor Laboratory Press, 2001.

17. F. Franek, C.G. Jennings and B. Smyth. *A Simple Fast Hybrid Pattern-Matching Algorithm*, In the proceedings of 16th Annual Symposium on Combinatorial Pattern Matching, LNCS 3537, Springer-Verlag, 288-297, 2005.

18. F. Michiels, A. Gasch, B. Kaltschmidt and R. Renkawitz-Pohl, *A 14 bp promoter element directs the testis specificity of the Drosophila β2 tubulin gene,* Journal of European Molecular Biology (EMBO), 8(5): 1559-1565, 1989.

19. G. Navarro. *A Guided Tour to Approximate String Matching*, Journal of the Association for Computing Machinery Computing Surveys, 33(1): 31-88, 2001.

20. L. Colussi, *Fastest pattern matching in strings*, Journal of Algorithms, 16(2): 163-189, 1994.

21. L. Holm and C.C. Sander, *Touring protein fold space with Dali/FSSP,* Nucleic Acids Research, 26: 316-319, 1998.

22. M.O. Dayhoff and R.V. Eck (Eds.), Atlas of Protein Sequence and Structure, Natl. Biomed. Res. Found., 3: 33.

23. M. Parisi, R. Nuttall, P. Edwards, J. Minor, D. Naiman, J. Lu, M. Doctolero, M. Vainer, C. Chan, J. Malley, S. Eastman and B. Oliver, *A survey of ovary-,testis-, and soma-biased gene expression in Drosophila melanogaster adults*, Genome Biology, 5: R40, 2004.

24. M.R. Garey and D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W.H. Freeman, 1979.

25. N. Ranganathan, *REPCLASS: Cluster and Grid Enabled Automatic Classification of Transposable Elements Identified DE NOVO in Genome Sequences*, Master's Thesis, UT Arlington, 2005.

26. P.A. Pevzner, Computational Molecular Biology: An Algorithmic Approach, The MIT Press, 2000.

27. R. Durbin, S. Eddy, A. Krogh and G. Mitchison, *Biological sequence analysis*, Cambridge University Press, 1998.

28. R. Lowrance and R.A. Wagner, *An extension of the string-to-string correction problem*, Journal of Association for Computing Machinery, 177-183, 1975.

29. R.A. Wagner, *On the complexity of the extended string-to-string correction problem*, In proceedings of 7th Association for Computing Machinery Symposium on Theory of Computing, 218-223, 1975.

30. R.A. Wagner and M.J. Fisher, *The string to string correction problem*, Journal of Association for Computing Machinery, 21: 168-173, 1974.

31. R.A. Drysdale, M.A. Crosby and The FlyBase Consortium, FlyBase: genes and gene models, Nucleic Acids Research, 33: D390-D395, 2005.

32. R.M. Karp and M.O. Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development, 31(2): 249 – 260, 1987.

33. R.N. Horspool, *Practical fast searching in strings*, Software – Practice & Experience, 10(6): 501-506, 1980.

34. R.S. Boyer and J.S. Moore, *A fast string searching algorithm,* Communications of the ACM, 20(10): 762–772, 1977.

35. SF Altschul, W Gish, W Miller, EW Myers, DJ Lipman, *Basic local alignment search tool,* Journal of Molecular Biology, 215: 403-410, 1990.

36. S Henikoff and JG Henikoff, Amino acid substitution matrices from protein blocks, In Proceedings of the National Academy of Science USA, 89(22): 10915-10919, 1992.

37. S. Kim. *A New String Pattern-Matching Algorithm Using Partitioning and Hashing Efficiently*, Association for Computing Machinery Journal of Experimental Algorithms, 4: 2, 1999.

38. S. Wu and U. Manber, *A Fast Algorithm For Multi-Pattern Searching*, Technical Report, TR-94-17, University of Arizona, 1993.

39. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, Second Edition, MIT press and McGraw-Hill, 2001.

40. The FlyBase Consortium, The FlyBase database of the Drosophila genome projects and community literature, Nucleic Acids Research, 27: 85-88, 1999.

41. The FlyBase Consortium, The FlyBase database of the Drosophila genome projects and community literature, Nucleic Acids Research, 31: 172-175, 2003.

BIOGRAPHICAL INFORMATION

Tushar Kumar Jayantilal joined The University of Texas at Arlington in the spring of 2004, for pursuing his M.S. He started his research career in the field of bioinformatics from fall of 2004. His research areas of interest are bioinformatics, algorithms, pattern recognition, data mining and high performance computing. He received his Bachelor's degree in Computer Science and Engineering from M.N.M Jain Engineering College, Chennai, India, affiliated to the University of Madras, Chennai, India. He received his Master of Science degree in Computer Science and Engineering, in August, 2006. His future plan is to start an industrial career, as a researcher for one of the nation's leading companies, in the field of bioinformatics.