COMPUTATIONAL INFERENCE TECHNIQUE FOR MINING STRUCTURED MOTIFS

BY

MAKOLO, ANGELA UCHE

(MATRIC NO.: 114866)

A Thesis in the Department of Computer Science

Submitted to the Faculty of Science

In partial fulfilment of the requirement for the Degree of

DOCTOR OF PHILOSOPHY

of the

UNIVERSITY OF IBADAN

SEPTEMBER 2012

CERTIFICATION

It is hereby certified that this work was carried out by MAKOLO, Angela, Uche in the Department of Computer Science, Faculty of Science, University of Ibadan.

……………….………………… ……………………...

Supervisor Date

Prof. Adenike Osofisan

B.Sc.(Hons) (Ife), M.Sc. (GIT), Ph.D. (Ife), MBA (Ibadan)

Computer Science Department

University of Ibadan.

[illegible]

30th Sept 2012

……………….……………… …………………… …

Co-Supervisor Date

Prof. Ezekiel Femi Adebiyi

B.Sc(Hons)(Ilorin),M.Sc.(Ilorin),Ph.D.(Tübingen).

CIS, CST, Covenant University, Otta.

Publications From this work

1. Conference papers

1. Makolo, Angela U, Osofisan Adenike, Adebiyi Ezekiel (2010): Computational Tool for Characterizing Malaria Parasite Transcription Factor and DNA Binding Site In proceeding of Research Consortium of the Nigerian Computer Society Pp 99 -106.

2. Makolo, Angela U, Osofisan Adenike, Adebiyi Ezekiel (2011): Mining Structured Motifs in Malaria Parasite with Gene Enrichment Motif Searching on Suffix tree. In proceeding of International Malaria Conference, Covenant University, Nigeria. Pp 15.

3. Makolo, Angela U, Osofisan Adenike, Adebiyi Ezekiel (2012): STGEMS: A Computational Inference Algorithm For Motif Discovery. In proceeding of ICT for Africa Conference, Uganda , Pp 800- 815

4. Makolo, Angela U, Osofisan Adenike, Adebiyi Ezekiel (2012): An inference Tool for identifying Regulatory Elements in Plasmodium falciparum. In proceeding of Louisiana Academy of Sciences Conference March 2012 Pp 34 -43.

5. Makolo, Angela U, Osofisan Adenike, Adebiyi Ezekiel (2012): Computational Tool for mining structured motifs. Abstract accepted for ISCB Conference July 2012 .

2. Journal Articles

1. Makolo, Angela, Ezekiel Adebiyi and Osofisan Adenike (2011). STGEMS: Mining Structured Motifs with Gene Enrichment Motif Searching on Suffix tree. Journal of Computer Science and its Applications 18(1) : 79-91.

2. Makolo, Angela, Ezekiel Adebiyi and Osofisan Adenike (2012). Comparative Analysis of Similarity Check Mechanism for Motif Extraction**.** Journal of Computer Science and its Applications . IEEE Afr J Comp & ICT 2012.http://www.ajocict.net/uploads/Final_Makolo-Similarity_Comparism-paper.pdf

3. Makolo, Angela, Ezekiel Adebiyi and Osofisan Adenike (2012) .A Survey of Computational Motif inference Algorithms. Oxford journal of Bioinformatics (In Press)

DEDICATION

This work is dedicated To Him in whom I Live and move and have my Being! The ultimate Good.

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Professor Adenike Osofisan, her candid encouragement, patience and tenacity in reading this work is note worthy. Working on this thesis has been a great learning experience for me and I am grateful to her for giving me the opportunity.

I equally want to thank Professor E.F Adebiyi for providing the desired technical assistance and research direction needed for this study; being the only Professor of Bioinformatics in the country at the moment, your contributions are invaluable.

My sincere gratitude to all the staff of Computer Science Department, University of Ibadan for their immense contributions to the success of this work. Especially to Dr Akinkumi for the technical assistance provided, Dr Onifade, for his patience in reading through the thesis, Dr Longe, Dr Robert, Dr Adeyemo, Dr Akinola, Dr Okike, Dr Osunade, Dr Folajimi, Dr Oladejo, Mrs Olamiti, Mrs Wumi. Adeyemo and Mrs Woods for their constant encouragement.

To my colleagues in the Bioinformatics research group at Covenant University, especially Dr S. Fatumo, Dr J. Oyelade, Dr G.Gbemi, Dr. Obi Eni, Itunu Isewon and Marion Adebiyi.

I certainly will remain indebted to my academic mentors: Prof. Chantal Epie and Dr Franca Ovadje of the Pan-African University Lagos, Prof. Helen Nwagwu, Prof. A.O. Olorunnisola, Prof. V.O Adetimirin, Prof. J.A.O. Woods, Prof. Bola Udegbe, Dr Chiaka Anumudu and Dr Awobode of the University of Ibadan, for their constant support.

My communication experts, Mr Ngozi Agbam and Ms Kemi Ogunyemi for taking time to read this thesis and proffer a constructive criticism from a literary perspective.

I appreciate the constant encouragement and support of my parents and siblings. I cannot thank you enough.

I cannot but acknowledge with deep appreciation, my immediate family, "Todos de la Casa" for the love, care and understanding you showed especially, during the very difficult moments of this work.

Table of Contents

List of figures

List of Tables

ABBREVIATIONS

GO	Gene Ontology
ATP	Adenosine Triphosphate
PSSM	Position Specific Scoring Matrix
PSWM	Position Specific Weight Matrix
PWM	Position Weight Matrix
NCBI	National Centre for Biotechnology Information

ABSTRACT

One of the major challenges in bioinformatics is the development of efficient computational tools for mining patterns. Structured motifs, like DNA binding sites in organisms with peculiarities in their genomic sequence like malaria parasite, *Plasmodium falciparum* have not been mined by existing structured motifs extraction tools. There is a need to develop faster computational tools to mine these DNA binding sites which are viable drug targets. This work was aimed at developing an algorithm for mining structured motifs in the genome of *P. falciparum.*

The Gene Enrichment Motif Searching (GEMS) method for mining simple motifs was modified by incorporating the time efficient implementation of the suffix tree data structure with suffix links. This enables an improved searching speed, while adding an optimized position-weight matrix computation using the hypergeometric-based scoring function. This algorithm, Suffix Tree Gene Enrichment Motif Searching (STGEMS) was implemented in C programming language on Linux platform. An empirical evaluation of the sensitivity of STGEMS was conducted by comparing the similarity check mechanism of the GEMS algorithm for mining simple motifs with that used in another popular algorithm for extracting structured motifs, a Multi-Objective Genetic Algorithm Motif Discovery (MOGAMOD). The output of STGEMS algorithm was validated by comparing the motifs discovered with those obtained using biological experiments. A further validation was done by applying the STGEMS and GEMS algorithm to selected metabolic pathways and the results were compared. The STGEMS algorithm was tested with four sets of genes from the intraerythrocytic development cycle of *P. falciparum.* The speed of execution was evaluated using three simple motif discovery tools: Expectation Maximization Motif Elicitation(*MEME)*, Gene Enrichment Motif Search (GEMS), and *WEEDER* as well as two structured motif discovery tools: *RISOTTO and EXMOTIF* on four different gene sizes.The high sensitivity of STGEMS in mining structured motifs from sequences in *P. falciparum* was proven empirically by its ability to identify 91% of the motifs in the sequences while MOGAMOD could not identify any motif. This validated the high sensitivity of the similarity check mechanism employed, in contrast with that used in MOGAMOD. The STGEMS algorithm identified 90% of the binding sites in *P. falciparum* which were similar to those obtained in biological experiments. On the selected metabolic pathways, STGEMS discovered all the simple motifs identified by GEMS, in addition to the structured motifs which GEMS could not identify. The empirical runtimes of STGEMS, *MEME*, *WEEDER, GEMS, RISOTTO and EXMOTIF* were respectively 20, 35, 26, 25, 28, 30 seconds for 20,000 base pair (bp), 32, 43, 44, 45, 42, 40 seconds for 40,000 bp, 41, 55, 56, 55, 52, 50 seconds for 60,000 bp and 54, 68, 69, 65, 67, 61 seconds for 80,000 bp respectively. The proposition resulted in a linear asymptotic runtime of O(N) at each iteration of the algorithm.

The suffix tree gene enrichment motif searching algorithm developed was time efficient and successful in mining structured motifs like DNA binding sites in *Plasmodium*

falciparum. This will aid a faster drug target discovery pipeline for the design of effective anti malaria drugs.

Keywords: Structured motifs, DNA Binding Site, Suffix tree.

Word Count: 500.

CHAPTER ONE

INTRODUCTION

1.1 BACKGROUND INFORMATION

Pattern discovery or identification remains a concern to biologists and computer scientists because of the challenges of developing efficient pattern mining tools for sequence motifs that consists of patterns on any combination of the characters that make up the DNA/RNA molecules, that is, {A,C,T,G) for DNA and {A,C,U,G}for RNA). These patterns appear repeatedly either in the same sequence string or over a set of sequences. Each of these patterns can be likened to a word in English language. Simple motifs are made up of single patterns or words while structured motifs are made up of several words with well defined gaps within a set of strings. For instance AATCGT is a simple DNA motif while

AATCGT-----AGTCCG is a structured motif consisting of two patterns, each of length six and five gaps. The need for efficient computational tools for mining these patterns has led to an increasing number of researchers developing new algorithms for the analysis of genomic data with the aim of extracting useful information from the patterns. Their identification is called motif inference or motif discovery which is an application area in data mining.

Data Mining is the process of automatically searching large volumes of data for hidden patterns with little or no knowledge of the existing patterns while pattern mining involves searching for patterns with a prior knowledge of the pattern of interest and applies to motif discovery problem where the interest is in searching for repeated patterns. (Daniels et al., 2011). They utilize computational techniques from statistics, information retrieval, machine learning and pattern recognition to extract these patterns. This area of pattern identification has applications in data compression, natural languages, databases, basically, any activity or research requiring text mining. The application of interest in this thesis is molecular biology and the motifs here may correspond to functional elements in DNA, RNA or protein molecules. Data mining algorithms have been widely used in molecular biology especially in protein structure prediction, gene classification and prediction, clustering of gene expression data, modeling of protein-protein interaction and motif discovery.

In biological applications, it is mandatory to allow for some mismatches between different occurrences of the same motif. This naturally makes the problem difficult from the computational point of view. In addition to this fundamental difficulty, this work also targeted the mining of structured motifs from the deadly organism- the malaria parasite, Plasmodium falciparum, with a peculiar genomic sequence. When the P. falciparum genome sequence was published in 2002, it was revealed that the genomic composition was unusually AT-rich, approximately 90% which is very high in comparison to other organisms. For example, the entire genomes of the two popular model organisms, yeast and fruit fly have AT contents of 62% and 65%, respectively (Gardner, 2002, Fatumo et al, 2010). This peculiarity necessitates the development of a suitable structured motifs inference algorithm that puts the malaria parasite genome in good perspective since none of the existing algorithms is capable of mining its structured motifs.

The importance of this work was highlighted at the 'Functional Genomics Workshop Group' meeting in Harvard, 2006, (Deitsch *et al*., (2007). The workshop identified the challenges of understanding of the biology of the deadly malaria parasite, *P.falciparum*. Key among these challenges is the identification of the proteins involved in its gene regulatory mechanisms. The fact that these proteins interact with the genomic DNA to bring the genome to life and that these interactions also define many functional features of the genome Iyer *et al*., (2001), make them viable drug targets. Thus, mining transcription associated proteins is a very important problem in malaria research.

Malaria is a tropical disease of great interest. It exacts a heavy toll of illness and death - especially amongst children and pregnant women. In Africa alone, malaria is estimated to kill a child, under the age of five every thirty seconds (Teklehaimanot, et al., 2005, Fatumo et al, 2009, Oyelade et al, 2010). It also poses a risk to travelers and immigrants, with imported cases increasing in non-endemic areas. The treatment and control of malaria has become more difficult with the spread of drug-resistant strains of the parasite and insecticide-resistant strains of mosquito vectors. Preventive measure such as health education, better case management, better control tools and concerted action are required to limit the scourge of the disease. (Bulashevska et al, 2007, Westenberger et al, 2009).

The dream of the global eradication of malaria, one of the millennium development goals, is beginning to fade with the growing number of cases, rapid spread of drug resistance in people and increasing insecticide resistance in mosquitoes. (Tuteja, 2007, Oyelade et al, 2010). This global malaria challenge makes the understanding of the biological mechanism of the malaria parasite very pertinent. The parasite inhabits two hosts- human beings and mosquitos, its characteristic genomic makeup makes it capable of adapting favourably in these two hosts, thereby making its eradication more challenging.

The malaria parasite exhibits a rapid growth and multiplication rate during many stages of its life cycle, this necessitates that the parasites, like all other organisms, acquire nutrients and metabolize these various biological molecules in order to survive and reproduce. It is expected that the parasite's metabolism will be intertwined with that of the host's because of the intimate relationship between the host and the parasite. (Bischo, E. and Vaquero, C.(2010)). These host-parasite interactions are further complicated by the complex life cycle of the parasite involving vertebrate and invertebrate hosts as well as different locations within each of these hosts. It is generally accepted that *P.falciparum* is entirely dependent on glycolysis for energy during the asexual stages, making this pathway an important drug target. One of the aims of this work is to discover motifs in this pathway and thus provide information on the functional modules of the glycolytic pathway genes. (Bozdech and Ginsburg, 2005).

The problem of understanding the transcription of genes is at the centre of interest in bioinformatics research. One of the reasons is that gene regulation is fundamental in determining the resulting functional protein. Transcription of genes serves as a substrate for evolutionary changes, because the control of the timing, location, and the amount of gene expression can have a profound effect on the functions of the genes in the organism. Transcription associated proteins are made up of transcription factors, that are master activators and inhibitors for sets of commonly regulated genes in each module. In this study, we examined their binding sites, which are expected on the DNA sequences of the genes that they regulate in the attempt to extract the binding sites. Since the malaria parasite *P. falciparum* is eukaryotic, that is, organisms that have a defined nucleus as opposed to prokaryotes which lack a nucleus, the binding sites are expected to be structured motifs (Flueck et al.,2010). Therefore, our computational inference technique mined these structured motifs as well as simple motifs in the challenging genomic sequence of the malaria parasite.

There are many computational methods for predicting transcription factors and their binding sites. These methods can be classified into three main categories, based on their operating principles. They are

- The pattern-driven or word-based approach
- The statistical based approach and
- The machine learning based approach.

A review of the motif discovery tools based on these three categories is provided in chapter two. Furthermore, there are experimental (wet lab) approaches for extracting binding sites, for example, DNA footprinting and Chromatin Immuno-Precipitation (chIP) methods; these approaches, however, are time consuming and very laborious. These weaknesses justify the need for computational methods to complement them. (Ponts et al., 2010). A repository of known transcription factors and their corresponding binding sites can be found in the TRANSFAC (Heinemeyer *et al.*, 1998) and JASPAR (Sandelin *et al.,* 2004) databases and these databases can be updated with novel transcription factors and binding sites after biological validation.

The computational inference technique, *Suffix Tree Gene Enrichment Motif Searching* (STGEMS) developed in this work, is a novel algorithm for mining motifs. The algorithm is specifically tailored to organism with peculiarities in their genomic sequence. STGEMS utilized the suffix tree, which has an inherent clustering technique that returns all repeated patterns at a remarkable speed and extracts optimal structured motifs by incorporating a highly sensitive similarity check mechanism using the hypergeometric scoring function and position weight matrix to rank the gene enrichment of the discovered motifs, thereby reporting only the optimal motifs. The suffix tree is a data structure that is useful in representing a string or set of strings, they are well suited to algorithms that require efficient access to substrings by content rather than by position. The suffix tree construction reorganizes data into a form that facilitates searching and exposes sections of the strings that are repeated. In view of the fact that the core aim of this study is in searching for repeated patterns in a set of DNA sequences, the choice of the suffix tree data structure in the framework of STGEMS algorithm is appropriate.

1.2 STATEMENT OF THE PROBLEM

Two fundamental challenges in malaria research are the identification and understanding of the complex proteins involved in the gene regulatory mechanism of *P.falciparum*. These challenges are greatly influenced by the extraction of transcription associated proteins (Transcription factors and DNA binding sites), which is cumbersome in the highly repetitive and specific alphabet bias sequence of P*.falciparum*. Therefore there is need for the development of computational tools for its effective mining, since existing tools had failed to mine these successfully.

The work of Young et al., (2008) and Kaya (2009) provide a motivation for this research. Young et al (2008) developed GEMS algorithm which extracted simple motifs in *P.falciparum*, but it failed to extract structured motifs, which form the core of regulatory element in eukaryotes. Kaya (2009) developed MOGAMOD algorithm, a multi-objective genetic algorithm, which extracted structured motifs successfully in yeast and other model organism but it failed when applied to the extraction of structured motifs in P.*falciparum* due to the peculiarity in its genomic sequence resulting in AT-rich sequence.

Our model, STGEMS, overcomes these limitations by extracting simple and structured motifs from the challenging sequence of the malaria parasite. This was achieved by implementing the similarity check mechanism used in MOGAMOD and GEMS, comparing them in terms of sensitivity level and incorporating that of GEMS into the framework of STGEMS while adding the suffix tree for improved speed.

1.3 AIM AND OBJECTIVES OF THE STUDY

The aim of this work is to develop a computational inference technique for mining structured motif that results in improved runtime with a view of elucidating the DNA binding site of transcription factors of the malaria parasite, P.*falciparum*. The specific objectives of the study are :

i. To develop a computational tool for mining structured motifs with improved running time with a high sensitivity or accuracy.

ii. To discover novel DNA binding sites, that is, viable drug targets to combat resistant malaria strain of *P.falciparum,* on a large scale using the developed tool.

iii. To compare the performance of the tool with the five popular motif discovery tools namely : MEME, WEEDER, RISOTTO, EXMOTIF and GEMS

iv. To compare the similarity check used in GEMS, that is hypergeometric scoring function based, with that used in MOGAMOD, that is dominance nucleotide value based.

v. To apply the novel algorithm to scan the glycolysis metabolic pathway genes of *P.falciparum*. .

1.4 RESEARCH QUESTIONS

To accomplish the above research objectives, the following research questions are addressed:

RQ1: How can an efficient motif discovery algorithm that would work for the challenging repeat alphabet sequence of *P.falciparum.* be developed?

RQ2: To what extent would this computational technique work for extracting DNA binding sites having a high correlation with those biologically extracted?

RQ3: To what extent would using a similarity check mechanism that employed a hypergeometric based scoring approach achieve a high sensitivity over the similarity check that does not use this approach?

RQ4: How would this new algorithm perform when compared to five standard motif discovery algorithms already in existence?

RQ5: How would this algorithm behave when applied to the glycolysis metabolic pathway genes?

1.5 METHODOLOGY

The overview of the research methodology is encapsulated in the figure 1.1:

The processes involved in STGEMS algorithm is depicted in steps

- STGEMS receives a list of DNA sequences as input, which contains unknown motifs that needs to be identified. A generalized suffix tree is constructed with the DNA sequences.
- The trees are traversed to output unique patterns or candidate motifs. In any suffix tree construction, each traversal from the root node to a leaf node is a unique pattern.

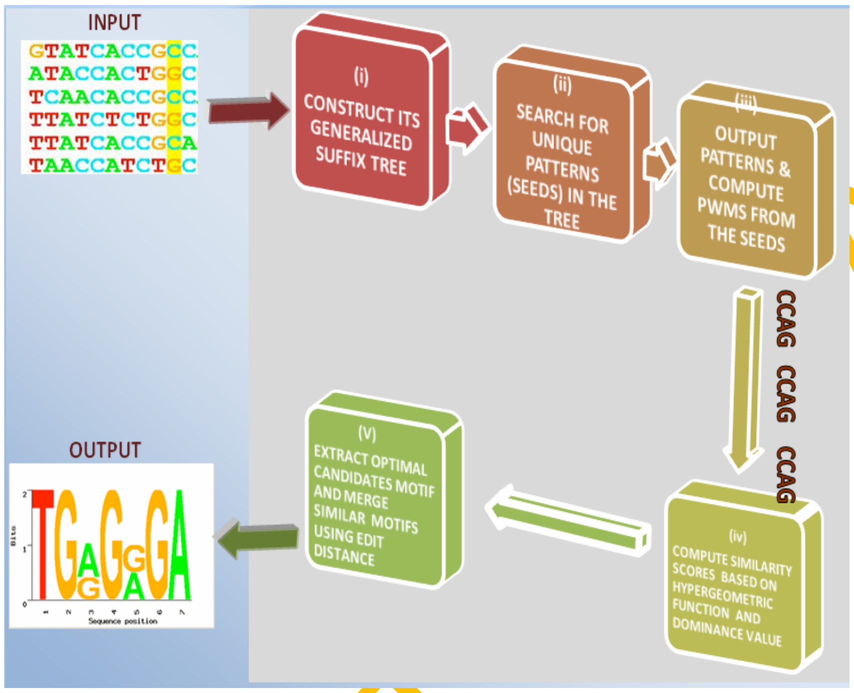

INPUT
GTATCACCGCC
ATACCACTGCC
TCAACACCGCC
TTATCTCTGGC
TTATCACCGCA
TAACCATCTGC
(i) CONSTRUCT ITS GENERALIZED SUFFIX TREE
(ii) SEARCH FOR UNIQUE PATTERNS (SEEDS) IN THE TREE
(iii) OUTPUT PATTERNS & COMPUTE PWMS FROM THE SEEDS
CCAG CCAG CCAG
(iv) COMPUTE SIMILARITY SCORES BASED ON HYPERGEOMETRIC FUNCTION AND DOMINANCE VALUE
(V) EXTRACT OPTIMAL CANDIDATES MOTIF AND MERGE SIMILAR MOTIFS USING EDIT DISTANCE
OUTPUT
Bits
Sequence position

Figure 1.1 Computational Framework for STGEMS algorithm

- The next step is the computation of position weight matrix (PWM) for the extracted unique patterns. The PWM is a matrix that shows the information content of the motifs which depends on the frequency of occurrence of each of the characters in the identified unique pattern.
- The biological significance of the candidate motif is computed. This is determined by computing the similarity scores. The motifs with low similarity scores are reported as better optimal motifs.
- The last step is the merging of similar motifs i.e. those with one or two variations in the character that make up the motifs. These are merged using edit distance, before returning them as optimal.

STGEMS's methodology also involves the implementation of the similarity check mechanisms based on the hypergeometric scoring function and the dominance nucleotide value of the extracted pattern from the suffix tree. The two methods were compared; the result of this comparison influenced the incorporation of the similarity check based on the hypergeometric scoring function into STGEMS framework.

The asymptotic runtime analysis of our novel computational inference technique, STGEMS was carried out by analyzing the time complexity of its various modules and procedures. A performance evaluation was conducted, based on the empirical runtime of STGEMS with three simple motif discovery tools (*MEME*, *WEEDER and GEMS)* and two structured motif discovery tools *(RISOTTO and EXMOTIF)*.

STGEMS was tested by running four experiments using different gene sets of *P. falciparum*. The first experiment used the set of genes in the work of Flueck *et al.* (2010) which experimentally extracted regulatory elements for P.*falciparum,* that is 1000 base pairs upstream of gene start codons. The second experiment used the set of genes used by Yuda *et al.* (2009) which identified transcription factors in the mosquito-invasive stage of malaria parasite. The two experiments aimed at inferring the ability and sensitivity of STGEMS in predicting correctly, the motifs already extracted by some known biological experiments. The third experiment targeted predicting novel binding sites by using the 3D7 gene clones of *P. falciparum*, this contains about 3,000 genes from the Intraerythrocytic developmental cycle (which represents all the stages in the development of P.*falciparum*, responsible for the symptom of malaria and is the target for the vast majority of antimalaria drugs strategy). The fourth experiment utilized the genes in the glycolytic pathway of *P.falciparum*. The twenty six genes that are known to be involved at the glycolytic pathway of *P.falciparum* were harvested from www.plasmodb.org. This experiment aimed at scanning the upstream regions for conserved sequence motifs using our computational technique. They were then compared with those extracted using the popular GEMS algorithm.

The implementation and testing of STGEMS was done in C, on a Dell computer, INTEL® CORE™ DUO CPU T2300 @1.66GHz, 512 RAM, 80GB HDD running on Suse Linux 11.2 operating system. This platform was adopted because it is an efficient platform for development of Bioinformatics algorithms. Moreover, the five popular motif discovery algorithms that STGEMS was compared with were implemented in **C** on Linux operating system thus ensuring a standardized comparative metric.

1.6 SIGNIFICANCE OF THE STUDY

Generally, the knowledge of the biology of *P.falciparum,* that consists of *:*

- How its cells function
- How proteins organize into modules such as metabolic pathways
- And the DNA binding sites present in the genome,

provides an invaluable resource for characterizing the complex roles of the individual genes and ultimately the identification and validation of new drug and/or vaccine targets for anti malaria strategies. The ability to discover these drugs or vaccine targets can only be enhanced from the deep understanding of the detailed biology of the parasite, which motivated this research.

This thesis addresses a significant need in our understanding of mining structured motifs especially in organisms with peculiarities in their genomic sequence. In particular, this work in developing a new motif discovery algorithm which efficiently mined biological patterns in the malaria parasite genome, provides information that will enhance understanding of malaria and development of drugs for its cure.

1.7 LIMITATION OF THE STUDY

This work focused on developing a motif discovery algorithm applied principally to mining structured motifs from *P.falciparum.* This is to provide an in-depth understanding of the organism. Despite the fact that other species of *Plasmodium* and other parasitic organisms can cause disease of importance, this study is limited to human malaria caused by *P. Falciparum,* because it is the most fatal and is considered to be of much economic importance.

1.8 ORGANIZATION OF WORK

This write-up is structured in five chapters. Chapter one gives a brief introduction to the work. It also includes statement of the problems, research questions, methodology, aims and objectives, as well as scope and limitation of work.

Chapter two explains the theoretical background of the research, the principles of sequence analysis algorithms, a review of related works in motif discovery algorithms and the gaps identified.

Chapter three describes the research methodology proposed for solving the identified gaps. The techniques used and model implementations were discussed. In addition, the structure of the data (malaria parasite genomic data) used in validating the algorithm was also expounded upon. Chapter four discusses the various results generated from running the algorithm and the comparative analysis carried out while chapter five gives the conclusion of the study and discusses future research directions in this area.

CHAPTER TWO

LITERATURE REVIEW

This chapter contains the theoretical background of the study, a review of the related concepts, principles and techniques used in this thesis. A review of related work is provided based on a chronicle of some popular Motif Discovery tools- examining their building paradigm, strengths and limitations. The basic data structures used in these motif discovery tools were discussed. Finally, the application domain of this research is highlighted, that is, mining patterns in molecular biology using malaria parasite, *Plasmodium falciparum* as a case study.

2.1 THEORETICAL BACKGROUND

In the post-genomic era, the ability to predict the behavior, the function, or the structure of biological entities (such as genes and proteins), as well as interactions among them, play a fundamental role in the discovery of information to help biologists explain biological mechanisms. (Pizzi, 2011).

Several functional and structural properties, and also evolutionary mechanisms, can be predicted either by the comparison of new elements with already classified elements, or by the comparison of elements with a similar structure or function and using it to infer the common mechanism that is at the basis of the observed similar behavior. Such elements are commonly called *motifs*.

Comparison-based methods for sequence analysis find their application in several biological contexts, such as extraction of transcription factors, DNA binding sites, identification of structural and functional similarities in proteins, and phylogeny reconstruction[1]. Therefore, the development of adequate methodologies for genomic sequence analysis is of paramount interest in computational biology. In other words sequence analysis refers to the process of subjecting a DNA, RNA or protein sequence to any of a wide range of analytical methods to understand its features, function, structure, or evolution. Sequence analysis algorithms are basically classified into three:

- **Gene finding Algorithms:** These algorithms are used to predict gene structure. Gene prediction or gene finding refers to the process of identifying the regions of genomic DNA that encode genes. This includes protein-coding genes as well as RNA genes, but may also include prediction of other functional elements such as regulatory regions. Gene finding is the first step in sequence analysis procedure. This is because the genes in the genome of any specie that had just been sequenced had to be annotated before any further processing can take place. The operating principle of gene finding algorithms is relatively simple; it is basically based on an inference system that can decode the twenty amino acids using the genetic code. Some popular gene finding tools are GENESCAN, GENEMAK, GENIE, HMMGENE, PHAT etc. (Cawley et al, 2001).
- **Sequence Alignment Algorithms**: These are algorithms that align genomic sequences to detect similarity. Sequence Alignment is a way of arranging the sequences of DNA, RNA, or protein to identify regions of similarity that may be

[1] Phylogenetic Reconstruction is a biological concept used in examining evolutionary relationship between organisms.

a consequence of functional, structural, or evolutionary relationships between the sequences. Aligned sequences of nucleotide or amino acid residues are typically represented as rows within a matrix. Gaps are inserted between the residues so that identical or similar characters are aligned in successive columns. The building paradigm of Sequence Alignment algorithms is usually more complex than gene finding algorithms. Popular sequence alignment tools include BLAST, ClustalW, T-coffee, FASTA3x among others. **(**Ortet & Bastien, 2010**).**

- **Motif Discovery Algorithm**s: These are algorithms that predict patterns from the sequence data hypothesised to have biological functions such as gene regulation. This class of algorithm is the most complicated of the three categories of sequence analysis algorithm available. Primarily due to the complicated makeup of the motifs been sought and therefore require exquisite methodologies to effectively predict them. Some popular tools in this class include MEME, WEEDER, MUSA among others.

Motif discovery algorithms are based on the biological theory of high conservation which states that patterns repeated in a sequence data with high frequency is a potential motif or pattern of interest and needs to be mined effectively (Gardener, 2002). The goal of motif discovery algorithms is to enumerate these patterns repeated with high frequency, since they have been established experimentally to have biological significance. The task is to eliminate those randomly occurring patterns which could result in false positive prediction and report only the best motifs.

The development of adequate methodologies for motif discovery is of unquestionable interest for several different fields in computational biology.

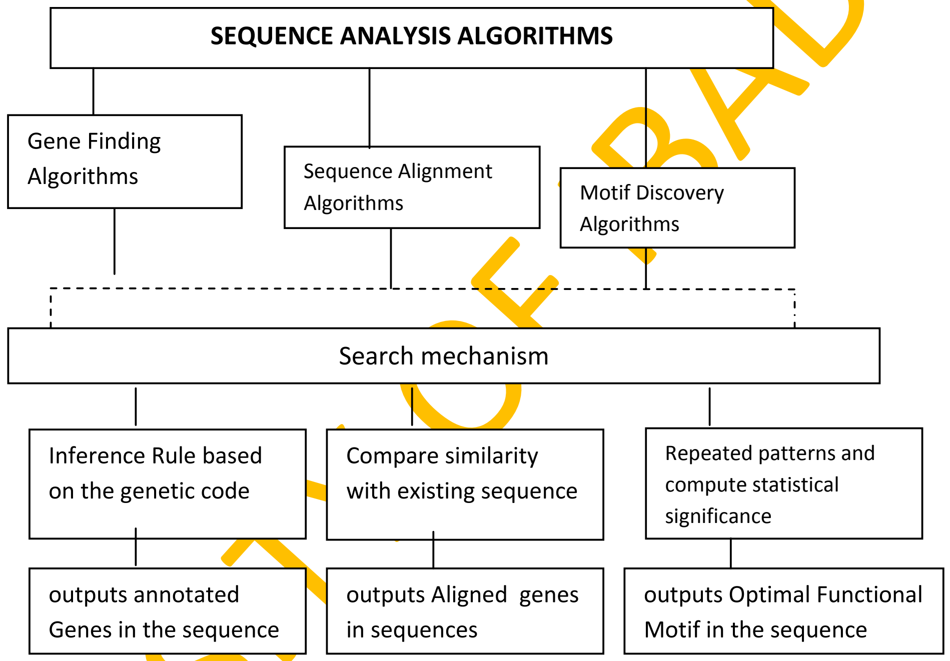

SEQUENCE ANALYSIS ALGORITHMS
Gene Finding Algorithms
Sequence Alignment Algorithms
Motif Discovery Algorithms
Search mechanism
Inference Rule based on the genetic code
Compare similarity with existing sequence
Repeated patterns and compute statistical significance
outputs annotated Genes in the sequence
outputs Aligned genes in sequences
outputs Optimal Functional Motif in the sequence

Fig 2.1 Hierachical View of Sequence Analysis Algorithms

Different researchers have adopted several approaches to extract these patterns such as word-driven or pattern-driven approach, statistical based approach and machine learning based approach. All known motif discovery algorithms are based on one or a combination of two or three of these approaches. Among the most popular methods are those based on the pattern driven approach methods which uses several heuristics to extract candidate motifs and thereafter performs a validation check using statistical methods to extract candidate motifs with optimal features based on the statistical significance analysis.

Motif discovery is an application area in the field of data mining in computer science. It is concerned with identifying and extracting relevant patterns hypothesised to have biological significance. Usually, a large data set is provided, then the data mining task involves the use of efficient techniques to mine the relevant patterns contained in the data set. A brief overview of data mining and its clustering techniques is provided in the next session.

2.1.1 OVERVIEW OF CLUSTERING IN BIOINFORMATICS

Motif discovery is basically a pattern identification process which involves the use of clustering techniques to extract biologically significant patterns from a given sequence data.

Clustering is partitioning of data into groups of similar objects. Each group, called a cluster, consists of objects that are similar to each other and dissimilar to objects of other groups. The representation of data using fewer clusters necessarily results in loss of some fine details while achieving simplification. Clustering is one of the most commonly used computational methods for analyzing microarray[2] and gene expression data of various organisms. The results obtained from clustering have been used to support the classification of genes into functional modules, such as regulatory elements, metabolisms and metabolic pathways.

Clustering differs from the supervised classification since in clustering; there is no information about the number of classes that are present before clustering begins.

The most typical example of clustering in bioinformatics is the clustering of genes in expression data. In microarray essays, the expression value for thousands of genes is obtained and interesting information that can be extracted from these data includes for instance, genes that are co expressed in the different samples. This is a clustering problem because here genes with similar expression level in all samples are grouped into a cluster. Cluster analysis, also called data segmentation, has a variety of goals. All relate to grouping or segmenting a collection of objects into subsets or 'clusters', such that those within each cluster are more closely related to one another than objects assigned to different clusters. Usually, the goal is to arrange the clusters into a natural hierarchy. This involves successively grouping the clusters themselves so that, at each level of the hierarchy, clusters within the same group are more similar to each other than those in different groups. Fundamental to all of the goals of cluster analysis is the notion of the degree of similarity (or dissimilarity) between the individual objects being clustered. A clustering method attempts to group the objects based on the definition of similarity supplied to it. Suffix tree clustering technique is widely used in motif discovery algorithms due to its ability to return common strings using their order and proximity of

[2] DNA Microarray is a collection of microscopic DNA spots attached to a solid surface. In molecular biology, DNA microarrays are used to measure the expression levels of large numbers of genes simultaneously or to genotype multiple regions of a genome.

reference in linear time. Other clustering methods used are partitioning based methods, hierarchy based methods, density-based, grid-based, and model-based methods. (Daniel et al., 2011, Naresh and Shrish , 2011). It is important to note that clustering in motif discovery algorithms is actually a pattern mining problem since there is a preconception about the patterns of interest (strings repeated with a high frequency) unlike clustering in data mining where the focus is searching for patterns of interest arising from the data without prior knowledge of the type or nature of the pattern. (Naresh and Shrish , 2011).

2.1.2 CHALLENGES OF GENE CLUSTERING

Gene-based clustering present many new challenges and problems that are still open due to the unique features of gene expression data and the particular requirements from the biological domain. Gene expression is the process by which information from a gene is used in the synthesis of a functional gene product. The challenges of gene clustering include the following:

- Cluster analysis is typically the first step in data mining since the purpose of clustering gene expression data is to reveal the natural data structures and obtain some initial insights regarding data distribution, a good clustering algorithm should therefore, as little as possible, depend on prior knowledge that is not usually available before cluster analysis.
- Due to the complex procedures of microarray experiments, gene expression data is often plagued with a lot of noise. Therefore, clustering algorithms for gene expression data must be capable of extracting useful information from a high level of background noise.
- Empirical studies have demonstrated that gene expression data are often "highly connected" (Jiang, et al, 2003) and clusters may be highly embedded in one another.

Therefore, algorithms for gene-based clustering should be able to effectively handle this situation.

- The users of microarray data are not only interested in the clusters of genes, but also in the relationship between the clusters and the genes within the same cluster. A clustering algorithm, which can partition the data set and also provide some graphical representations of the cluster structure (intra- and inter- relationship wise) is more useful to the biologists.

All Motif discovery algorithms incorporate a specific type of clustering mechanism in its design framework. The challenges of gene clustering discussed above are taking into considerations while developing different methodologies for motif discovery. This section has examined the main challenges that should be bore in mind while considering a particular clustering technique for motif discovery tool. We shall hereafter, .provide a definition of the motif discovery problem.

2.1.3 THE CONCEPT OF MOTIF DISCOVERY

Motif discovery in DNA sequences is a fundamental problem in computational biology with important applications in understanding gene regulation. Biological approaches for this problem are tedious and time-consuming. The availability of large amounts of genome sequence data and gene expression micro-array data make it possible to solve this problem computationally. However, most computer science problems of this sort are NP-complete.

A motif is a pattern in a sequence, that is, a word and a representation of a set of such words. In a more formal way, a motif can be defined as follows:

Let Σ represent the alphabet $\{A,C,G,T\}$ of nucleotides.

An element $u \in \Sigma^{+}$ is said to be a word in a sequence $s \in \Sigma^{+}$

if $s=xuy$ *for* $x, y \in \Sigma^{*}$.

An element $m \in \Sigma^{+}$, called a motif is said to have an *e-occurrence in s* for a non negative integer if there is at least one word u in s

such that the minimum number of substitution between u and m is no more than e. Given N sequences s_1 , .. $s_N \in \Sigma^{*}$ and an integer $1 \leq q \leq N$, an element $m \in \Sigma^{+}$ is said to be a valid motif if it has at least an occurrence in at least q distinct sequences of the set(q is called the quorum).

Motif discovery has been applied to discover many types of patterns in DNA and amino acid sequences. For example, motif discovery has been used extensively to identify transcription factor binding sites and to discover protein-protein interaction domains. In most cases, motif discovery algorithms accept as input a set of sequences hypothesized to contain a biologically important sequence pattern, and search for patterns that are repeated with high frequency, that is, patterns unlikely to occur by chance. However, since motifs are usually short and can be highly variable sequence patterns, a challenging problem for motif discovery algorithms is to distinguish functional motifs from random patterns that are over-represented by chance. (MacIsaac *et al.,* 2006). This problem is addressed in some motif extraction algorithms by using the information content or relative entropy of the motif. The relative entropy of a motif is defined as follows: Suppose that a motif of length L, has approximate occurrences in a subset S of N input sequences. Then the relative entropy or information content of this motif is defined to be

$$\sum_{j=1}^{L} \sum P_{r,j} \log_2 P_{r,j/b_r}$$

r$\in$ {A,C,G,T}

where $P_{r,j}$ is the frequency of occurrence of the nucleotide or residue r in position j among the motif occurrences in S, and br is the background frequency of the residue .

Relative entropy provides a measure of frequency of occurrence and how unlikely a motif is with respect to the background distribution. In particular, the more dissimilar the distribution $P_{r,j}$ from the background distribution br, the higher the relative entropy of position j.

Relative entropy is a good criterion to use when comparing two motifs with the same number of occurrences; however, it does not suffice if the two motifs occur in a vastly different number of sequences. This is due to the fact that relative entropy does not take into account the absolute number of occurrences; rather it depends on the relative frequency of occurrence of each of the nucleotides. (Tompa M , 2005). The nucleotides in a DNA sequence are made up of alphabets 'A','C','T','G' while in RNA sequence, the alphabet 'T' is replaced by 'U'. .

Motif discovery can also be seen as the problem of discovering promoter sequences and binding sites for transcription factors, usually referred to as consensus sequences or motifs, without any prior knowledge of their characteristics. These motifs can be sought by analyzing regulatory regions taken from genes of the same organism or from related genes of different organisms. Bock et al.,(2006).

Motif discovery can be formally defined as follows:

Assuming that G ={g1, ….gT } is a set of DNA sequences.

Let M be a motif of length L. M a motif that can occur in a sequence with mutations up to F instances in the sequence. Assuming m1, m2…mT to be instances of occurrence of M.

The motif finding problem of (L, F) is to find M such that $P^* \leq P$. Usually, we assume $P^* < 3$ to avoid discovering motifs with less significance called trivial motifs.

It is important to note that typically several target (L, F) motifs may exist, depending on P*.

In general, a computational algorithm cannot determine a priori the biological significance of a motif with certainty, thus the aim will be to find all these motifs based on their frequency of occurrence and to statistically validate the biological relevance of the extracted motifs. This statistical validation is usually accomplished using a scoring matrix. The most common scoring matrix model used for motif discovery tools is the Position Specific Scoring Matrix (PSSM). (Pizzi et al. 2011).

2.1.6 OPEN CHALLENGES IN MOTIF DISCOVERY

There are many open challenges in motif discovery, one that is often overlooked, involves the partitioning of the input set of sequences into target and background sets. The target set is the cluster of interest suspected to have the motif, while the background set is the remaining sequence set.

Many methods rely on the user to provide these two sets and search for motifs that are overabundant in the target set when compared with the background set. The question of how to partition the data into target and background sets is left to the user. However, the boundary between the sets is often unclear and the exact choice of sequences in each set arbitrary. For example, suppose that one wishes to identify motifs within promoter sequences that constitute transcription factor binding site, a strategy that can be adopted would be to partition the set of promoter sequences into target and background sets according to the transcription factor binding signal (as measured by ChIP–chip experiments (Keich and Pezner, 2002). The two sets would contain the sequences to which the transcription factor binds ‘‘strongly’’ and ‘‘weakly,’’ respectively. A motif detection algorithm could then be applied to find motifs that are overabundant in the target set compared with the background set. In this scenario, the positioning of the cutoff between the strong and weak binding signal is somewhat arbitrary. Obviously, the final outcome of the motif identification process can be highly dependent on this choice of cutoff. A stringent cutoff will result in the exclusion of informative sequences from the target set while a loose cutoff will cause inclusion of irrelevant sequences. It is obvious that both extremes hinder the accuracy of motif prediction. This example demonstrates a fundamental difficulty in partitioning most types of data.

There are five major challenges in motif discovery which require consideration, they are:

- The cutoff used to partition data into a target set and background set of sequences is often chosen arbitrarily;
- The lack of an exact statistical score and p-value for motif enrichment. Current methods typically use arbitrarily set thresholds or simulations, which are inherently limited in precision and costly in terms of running time.
- The need for an appropriate framework that accounts for multiple motif occurrences in a single promoter region. For example, how should one quantify the significance of a single motif occurrence in a promoter against two motif occurrences in a promoter? Linear models (Bussemaker et al, 2001) assume that the weight of the latter is double that of the former. However, it is difficult to justify this approach since biological systems do not necessarily operate in such a linear form
- Another issue that is related to motif multiplicity is low complexity or repetitive regions. These regions often contain multiple copies of degenerate motifs. Since the nucleotide frequency underlying these regions substantially deviates from the standard background frequency, they often cause false-motif discoveries.

Consequently, most methods mask these regions in the preprocessing stage and thereby lose vital information that might reside therein.

- Some criticism has been made over the fact that motif discovery methods tend to report presumably significant motifs even when applied on randomly generated data (Habison et al, 2004). These motifs are clearly, cases of false positives and thus should be avoided.
- The challenge of the multiple occurrence of motif in a single sequence is further compounded by the fact that the there might not be an absolute optimal motif. It usually occurs in multiples.

Several motif discovery tools tried to take this challenge into consideration when building the motif extraction model.

A typical example is the motif discovery tool by Kaya, 2009 who developed MOGAMOD(Multi) using multi-objective genetic algorithm approach to discover optimal motifs in sequential data. The main advantage of this approach is that a large number of tradeoff (i.e., nondominated) motifs can be obtained by a single run with respect to conflicting objectives: similarity, motif length and support maximization. Thereby reporting all the multiple optimal motifs present in the sequence data. A discussion of multi-objective genetic algorithm and its implementation is discussed in the sections below.

2.1.7 MULTI-OBJECTIVE GENETIC ALGORITHM (GA)

Multi-objective genetic algorithms are evolutionary algorithms which have become the method of choice for optimization problems that are too complex to be solved using deterministic techniques such as linear programming or gradient methods. It uses Multi-objective optimization. Multi-objective optimization is the process of simultaneously optimizing two or more conflicting objectives subject to certain constraints. For multiple-objective problems, the objectives are generally conflicting, preventing simultaneous optimization of each objective. Many, or even most, real engineering problems actually do have multiple objectives, that is, minimize cost, maximize performance, and maximize reliability, among others. These are difficult but realistic problems. GA are a popular meta-heuristic that is particularly well-suited for this class of problems. Traditional GA is customized to accommodate multi-objective problems by using specialized fitness functions and introducing methods to promote solution diversity. There are two general approaches to multiple-objective optimization. One is to combine the individual objective functions into a single composite function and the other involves moving all but one objective to the constraint set. In the former case, determination of a single objective is possible with methods such as utility theory, weighted sum method among others, but the problem lies in the proper selection of the weights or utility functions to characterize the decision-maker's preferences.

In practice, it can be very difficult to precisely and accurately select these weights, even for someone familiar with the problem domain. Compounding this drawback is that scaling amongst objectives is needed in order to be able to choose the best objective criteria since a few alterations in the weights could result in different solutions. In the latter case, the problem is that in moving objectives to the constraint set, a constraining value must be established for each of these former objectives. This movement of objectives could be arbitrary and thus the problem of consistency ensued. In both cases, an optimization method would return a single solution rather than a set of solutions that can be examined for trade-offs. For this reason, decision-makers often prefer a set of good solutions considering the multiple objectives.

The second general approach is to determine an entire Pareto optimal solution set or a representative subset. A Pareto optimal set is a set of solutions that are non dominated with respect to each other. While moving from one Pareto solution to another, there is always a certain amount of sacrifice in one objective to achieve a certain amount of gain in the other. Pareto optimal solution sets are often preferred to single solutions because they can be practical when considering real-life problems since the final solution of the decision-maker is always a trade-off. Pareto optimal sets can be of varied sizes, but the size of the Pareto set usually increases with the increase in the number of objectives.

A general minimization problem of N objectives can be stated mathematically as

Minimize $f(x) = [fi(x), \text{i} = 1,..,\text{N}]$

Subject to:

$g_j(x) \leq j = 1.2\ldots,J$

$h_k(\text{h}) = 0 \quad \text{k} = 1.2\ldots,\text{K}$

Where $f_{\text{i}}(x)$ is the ith –objective function, $g_j(x)$ is the jth inequality constraint. The multi objective optimization problem is then reduced to finding x such that $f(x)$ is optimized. The ultimate goal of a multi-objective optimization algorithm is to identify solutions in the Pareto optimal set. However, identifying the entire Pareto optimal set, for many multi-objective problems, is practically impossible due to its size. In addition, for many problems, especially for combinatorial optimization problems, proof of solution optimality is computationally infeasible because of the search operations involved. Therefore, a practical approach to multi-objective optimization is to investigate a set of solutions (the best-known Pareto set) that represent the Pareto optimal set as much as possible. With these concerns in mind, a multi-objective optimization approach should achieve the following three conflicting goals (Zitzler et al, 2000)

- The best-known Pareto front should be as close as possible to the true Pareto front. Ideally, the best-known Pareto set should be a subset of the Pareto optimal set.
- Solutions in the best-known Pareto set should be uniformly distributed and diverse over of the Pareto front in order to provide the decision-maker a true picture of trade-offs.
- The best-known Pareto front should capture the whole spectrum of the Pareto front. This requires investigating solutions at the extreme ends of the objective function space.

For a given computational time limit, the first goal is best served by focusing the search on a particular region of the Pareto front. On the contrary, the second goal demands the search effort to be uniformly distributed over the Pareto front. The third goal aims at extending the Pareto front at both ends, exploring new extreme solutions. (Konak et al., 2006).

Usually, Genetic algorithms are well suited to solve multi-objective optimization problems. A generic single-objective GA can be modified to find a set of multiple non-dominated solutions in a single run. The ability of GA to simultaneously search different regions of a solution space makes it possible to find a diverse set of solutions for difficult problems with non-convex, discontinuous, and multi-modal solutions spaces. The crossover operator of GA is capable of exploiting structures of good solutions with respect to different objectives to create new non-dominated solutions in unexplored parts of the Pareto front. In addition, most multi-objective GA does not require the user to prioritize, scale, or weigh objectives. Therefore, GA has been the most popular heuristic approach to multi-objective design and optimization problems. A survey by Jones *et al.*, (2002) reported that 90% of the approaches to multi objective optimization are aimed at approximating the true Pareto front for the underlying problem, majority of these used a meta-heuristic technique, and 70% of all meta-heuristics approaches were based on evolutionary approaches.

The first multi-objective GA, called vector evaluated GA, VEGA, was proposed by Schaffer (1985). This was followed by Several Multi-objective Genetic Algorithm(MOGA) such Niched Pareto Genetic Algorithm (NPGA), Weight-based Genetic Algorithm (WBGA), Random Weighted Genetic Algorithm (RWGA), Non dominated Sorting Genetic Algorithm (NSGA), Strength Pareto Evolutionary Algorithm (SPEA), improved SPEA called SPEA2, Pareto-Archived Evolution Strategy (PAES), Pareto Envelope-based Selection Algorithm(PESA), Region-based Selection in Evolutionary Multiobjective Optimization, PESA-II, a fast Non dominated Sorting Genetic Algorithm, (NSGA-II), Multi-objective Evolutionary Algorithm, Micro-GA, Rank-Density Based Genetic Algorithm(RDGA), and Dynamic Multi-objective Evolutionary Algorithm(DMOEA). It is important to note that although there are many variations of multi-objective GA available, these cited GA are well-known and credible algorithms that have been used in many applications and their performances were tested in several comparative studies. Konak et al, (2006). A list of some popular multi-objective GA is contained in table 2.6 below indicating their strengths and weaknesses.

Table 2.5 A List of popular Multi-objective Genetic Algorithms (Konak et al, 2006)

Algorithm	Fitness assignment	Diversity mechanism	Elitism	External population	Advantages	Disadvantages	Author
VEGA	Each subpopulation is evaluated with respect to a different objective	No	No	No	First MOGA Straightforward implementation	Trend converges to the extreme of each objective	Schaffer J.D (1985)
MOGA	Pareto ranking	Fitness sharing by niching	No	No	Simple extension of single objective GA	Usually slow convergence	Fonseca wt al. (1993)
WBGA	Weighted average of normalized objectives	Niching. Predefined weights	No	No	Simple extension of single objective GA	Difficulties in nonconvex objective function space	Hajela P.,Lin C-y. (1992)
NPGA	No fitness assignment, tournament selection	Niche count as tie-breaker in tournament selection	No	No	Very simple selection process with tournament selection	Problems related to niche size parameter. Extra parameter for tournament selection	Horn J., Nafplio N. Goldberg D.E. (1994)
RWGA	Weighted average of normalized objectives	Randomly assigned weights	Yes	Yes	Efficient and easy implement	Difficulties in nonconvex objective function space	Murata et al (1995)
PESA	No fitness assignment	Cell-based density	Pure elitist	Yes	Easy to implement. Computational efficient	Performance depends on cell sizes. Prior information needed about objective space	Corne D.W., Knowles J.D., Oates M.J. (2000)
PAES	Pareto dominance is used to replace a parent if offspring dominates	Cell-based density as tie breaker between offspring and parent	Yes	Yes	Random mutation hill-climbing strategy. Easy to implement. Computationally efficient	Not a population based approach. Performance depends on cell sizes	Knowles J. Corne D. (1999)
NSGA	Ranking based on non-domination sorting	Fitness sharing by niching	No	No	Fast convergence	Problem related to niche size parameter	Srinivas N. Deb K. (1994)
NSGA-II	Ranking based on non-domination sorting	Crowding distance	Yes	No	Single parameter (N). Well tested. Efficient	Crowding distance works in objective space only	Deb K et al. (2000)
SPEA	Ranking based on the external archive of non-domination solutions	Clustering to truncate external population	Yes	Yes	Well tested. No parameter for clustering	Complex clustering algorithm	Zitzler E. Thiele L.(1999)
SPEA-2	Strength of dominators	Density based on the k-th nearest neighbor	Yes	Yes	Improved SPEA. Make sure extreme points are preserved	Computationally extensive fitness and density calculation	Zitzler E. Laumanns M. Thiele L.(2001)
RDGA	The problem reduced to bi-objective problem with solution rank	Forbidden region cell-based density	Yes	Yes	Dynamic cell update. Robust with respect to the number of objectives	More difficult to implement than others	Lu H. Yen G.G. (2003)
DMOEA	Cell-based ranking	Adaptive cell-based density	Yes (implicitly)	No	efficient techniques to update cell density. Adaptive approaches to set GA parameters	More difficult to implement than others	Yen G.G. Lu H. (2003)

2.1.7.1 THE NON DOMINATED SORTING GENETIC ALGORITHM II (NSGA II)

The multi-objective genetic algorithm adopted in STGEMS implementation is the Non dominated Sorting Genetic Algorithm NSGA II. This is an improvement on the non dominated sorting genetic algorithm (NSGA) proposed by Srinivas and Deb (1995). The main criticisms of the NSGA approach are as follows:

- High computational complexity of non dominated sorting: The currently-used non dominated sorting algorithm has a computational complexity of $O(MN^3)$ where M is the number of objectives and N is the population size. This makes NSGA computationally expensive for large population sizes. This large complexity is due to the complexity involved in the non dominated sorting procedure in every generation.
- Lack of elitism: *The* reports of Fonseca and Fleming(1998) and Coello and Pulido(2001) showed that elitism can speed up the performance of GA significantly, and at the same time prevent the loss of good solutions once they are found.
- Need for specifying the sharing parameter: The traditional mechanisms of ensuring diversity in a population so as to get a wide variety of equivalent solutions have relied mostly on the concept of sharing. The main problem with sharing is that it requires the specification of a sharing parameter and usually, parameter-free diversity preservation is always preferred.

NSGA II procedure basically consists of creating a random parent population at first; then the population is sorted based on the non domination. Each solution is assigned a fitness (or rank) equal to its non domination level. Thus, minimization of fitness is assumed.

In general, motif discovery algorithms employ a search algorithm to obtain the initial cluster, since the first step in any clustering technique is the search for the items to be clustered. However, the search scheme in motif discovery algorithms differ considerably from other search schemes especially the web search where the search item is specified before the search begins but not the same with the search scheme of motif discovery tools since the items are not known before the search begins. An overview of search algorithms is discussed below.

2.1.8 OVERVIEW OF SEARCH ALGORITHMS

A search algorithm is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure.

There are basically two classes of search algorithms. The informed search and the uninformed search algorithm. An informed search algorithm looks for a specific answer to a specific problem in a data, that is, the particular item being search for is known before the search commences. A typical example of this is searching for items on the web. While for uninformed search, the item is not known before the search begins. For example, the search involved in motif discovery, since the patterns are not known before hand and therefore, they usually search for high occurring patterns.

The different types of search algorithms are:

List search – A list search algorithm searches through specified data looking for a single key. The data is searched in a very linear, list-style method. The result of a list search is usually a single element.

Graph Search

Graph searches can be either depth-first or breadth-first. These two kinds of searches visit the nodes in different orders. A depth-first search is most often implemented with a recursive algorithm. A depth-first search follows one path of the graph until it can no longer proceed. It then backs up until it finds a path that has not been visited and proceeds down that path until it comes to the end. The process continues until all reachable nodes have been visited. The breadth-first search algorithm is very different. A breadth-first search visits all nodes that are one node away from the starting node first. It then visits all nodes that are two nodes away from the starting node and continues until all reachable nodes have been visited.

To search a graph (directed or not) in breadth first; this is done by using a queue where the vertices found are stored. Unlike binary trees, graphs do not have a root, so the search can begin at any node. Also, because graphs can contain cycles, it is necessary to mark each node as it is visited to prevent an endless loop around the cycle.

SQL search - One of the difficulties with a tree search is that it is conducted in a hierarchical manner, which implies that the search is conducted from one point to another, according to the ranking of the data being searched. A SQL search allows data to be searched in a non-hierarchical manner, which gives the advantage of searching for data from any subset of data.

Adversarial search - An adversarial search algorithm looks for all possible solutions to a problem. It is similar to finding all the possible solutions in a game. This algorithm is found useful in most genomic data set, since we want all the possible patterns or combinations to be reported, however, its implementation is involving.

Constraint satisfaction search - In this type of search algorithm, the solution is discovered by meeting a set of constraints, and the data set can be searched in a variety of different ways that do not have to be linear.

Tree search – A tree search algorithm works by searching a data set from the broadest to the narrowest, or from the narrowest to the broadest. Data sets are like trees; a single piece of data can branch to many other pieces of data. Tree searches are more useful when conducting searches on a large data sets and speed rather than space is of paramount importance. There are different types of trees with various search capabilities for instance the binary tree, prefix tree, and suffix tree among other.

The performance of algorithms are measured based on the computational metrics of time and space. The section below provides a brief description of these metrics.

2.1.9 COMPUTATIONAL COMPLEXITY METRICS FOR MOTIF DISCOVERY ALGORITHMS

The efficiency of algorithms is a measure of the time and space complexity, these metrics are important when comparing algorithms. Estimation of complexity metric for motif discovery algorithms is a necessary benchmark for the efficiency of the methodology adopted in the design of motif discovery tool. For tasks, such as searching, that are repeated frequently, the choice among alternative algorithms becomes important because they differ in complexity. The complexity of an algorithm can be measured in three different ways: best-case complexity, average-case complexity and worse-case complexity. Best-case complexity is the measure of the complexity of solving the problem for the best size of input. Average-case complexity measure is the complexity of solving the problem on an average input size while the worst-case complexity is that of solving the problem for the worst size of input.

The time complexity of an algorithm is a measure of the amount of time taken by the algorithm to run as a function of the size of the input to the problem. It measures the running time of the algorithm which is the number of machine instructions it executes when the algorithm is run on a particular instance. The time complexity of an algorithm is commonly expressed using big O notation, which suppresses a multiplicative constants and lower order terms. When it is expressed this way, the time complexity is said to be described asymptotically, that is, as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $8n^3 + 5n$, the asymptotic time complexity is O (n^3). (Hopcroft et. al., 2007).

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm may take a different amount of time even on inputs of the same size, the most commonly used measure of time complexity, is the worst-case time complexity of an algorithm, denoted as $T(n)$. It is the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm. It is said to take logarithmic time if $T(n) = O(\log n)$. The better the time complexity of an algorithm is, the faster the algorithm will carry out its work in practice. (Flum et al, 2006).Time complexity measurement is crucial for motif discovery algorithms which aim at extracting patterns from very large data sets. It is very essential that the algorithm is adequately tuned to speed up the search process and thereby achieve a favourable runtime.

The space complexity of a program is the number of elementary objects that the program needs to store during its execution. It is essentially the number of memory cells which an algorithm needs and a good algorithm keeps this number as minimal as possible. This number is computed with respect to the size n of the input data.

In space and time complexity management, there is usually a trade-off between storage space and computing time, consequently, there is need for a compromise between the choice of computing time and memory consumption . This however, depends on the choice of algorithm and the parameters specified. (Naresh and Shrish , 2011).

The performance of an algorithm is also influenced by the choice of data structures used in the implementation of the algorithm. Researchers involved in the development of motif discovery tools realising the importance of these data structures have employed the different structures in an attempt to improve the performance of their algorithm. Some popular data structures used include suffix tree, suffix array, hash tables and link lists.

2.2 RELATED WORKS

There are several approaches proposed in literature for the discovering of simple and structured motifs. This review of related work is based on the three main approaches used in the literature namely the Pattern Driven Approach, Statistical Based Approach and Machine Learning Approach.

2.2.1 MOTIF DISCOVERY TOOLS BASED ON MACHINE LEARNING APPROACH

Several motif discovery algorithms used different machine learning techniques as their operating principle. The most common machine learning technique used in motif inference tool is the genetic algorithm. The advantage of such genetic algorithm based methods is that they are likely to locate the global optimum in a typically difficult search space. On the other hand, they are stochastic and so they may fail to report consistent results in different runs. They also require a large population of solutions and the computation time is usually high. The other machine learning techniques that have been used in motif discovery are the Artificial neural Network and Support Vector machine . In general, machine learning techniques requires a large data set to adequately train the learning tool in order to achieve relevant predictions and that can limit its use when the required data size is not available.

A survey of motif discovery tools by Kaya, 2009 reported that MOGAMOD (Multi-Objective Genetic Algorithm Motif Discovery), outperformed the existing motif discovery tools that used pattern driven or statistical approach in terms of accuracy of predicted motifs. However, this success was limited to model organisms with a generic genomic structure, since it failed to identify any motif from the challenging sequence of the malaria parasite genome.

MOGAMOD used the multi-objective genetic algorithm to discover optimal simple motifs in sequential data. Multi-objective optimization involves having a solution which is a family of pareto-optimal set or non dominated solutions. The optimal motif discovery problem was converted into three conflicting optimization problems of maximizing similarity, increase motif length and support for candidate motifs. The implementation of MOGAMOD was based on a well known high performance multi-objective Genetic Algorithm called NSGA II(Non- dominated Sorting Genetic Algorithm) by Deb et al (2002).

The sensitivity of MOGAMOD was enhanced by its flexibility in choice of similarity measures for finding motifs. The user can analyze the obtained optimal motifs, and can make a decision on the tradeoff between the different objectives.

In 2009, the same author extended MOGAMOD to extract structured motif. Kaya (2009). The objective function optimizations used were similarity maximization, total gap minimization and support maximization. The performance of the algorithm was compared with two well known structured motif extraction methods: ExMOTIF and RISOTTO and it showed a good performance with respect to speed and accuracy. Nevertheless, it could not identify structured motifs from malaria parasite genome (Makolo et al., 2012).

Another popular tools based on genetic algorithm is FMGA (Finding Motif with Genetic Algorithm) by Liu *et al*. (2004). In FMGA, the mutation in GA is performed by using position weight matrices to reserve the completely conserved positions. The crossover is implemented with specially designed gap penalties to produce the optimal child pattern. This algorithm also uses a rearrangement method based on position weight matrices to avoid the presence of a very stable local minimum, which may make it quite difficult for the other operators to generate the optimal pattern. The authors reported that FMGA performs better in comparison to MEME and Gibbs sampler algorithms.

GAME (Genetic Algorithm Motif Elicitation) by Wei and Jensen (2006) is also a common motif discovery tool based on genetic algorithm. GAME is an optimization algorithm for motif discovery capable of an exhaustive search of the space of possible motifs. It used position weight matrix based optimization to find an optimal motif in the search space. GAME outperformed MEME and BioProspector when used on both simulation and real-data sets.

2.2.2 PATTERN-DRIVEN BASED MOTIF DISCOVERY TOOLS

Pattern-driven method enumerates all the patterns in order to determine those appearing with a high frequency in the input sequence. It also considers the number of possible substitutions and thereafter provides a ranking for the extracted patterns according to some statistical measure of significance. The drawback in this approach is that they can have many false predictions, since they are not good at discriminating the relevant extracted motifs from the potentially numerous false hits. In addition, this method requires a large number of parameters to be specified. (Apostolico et al, 2008)

The techniques used in Pattern driven include enumeration (listing items in an order), suffix tree, graph, hash table and link list. The review of motif discovery tools based on the tools that utilized suffix tree and Enumeration techniques that are more relevant to this study is the subject next section. The review is grouped under simple and structured motif discovery tools.

As stated earlier, simple motifs are made up of a single pattern that is made up of different characters while Structured motif consists of a subsequence of intermittent characters interspersed with spaces and appearing frequently in an input sequence. (Apostolico et al, 2008).

2.2.2.1 Simple Motif Discovery Tools using Pattern Driven-based Approach

The first simple motif discovery algorithms developed using enumeration technique - a pattern-driven approach was by Van Helden *et al*. (1998). The algorithm – Oligo-Analysis was conceptually simple and could not detect motifs with mutation. This problem was resolved by Tompa (1999) using the same enumeration method. The algorithm identified motifs with mutations by considering both the absolute number of occurrences and the background distribution, while creating a table that records for each sequence length, the number of sequences containing the repeated sequence, where an occurrence allows for a small, fixed number of substitutions. Sinha and Tompa(2000) also used enumeration method but incorporated statistical validation and developed the algorithm YMF (Yeast Motif Finder). The statistical validation used is the z-scores, which is the number of standard deviations by which its observed number of instances in the actual input sequences exceeds its expected number of instances, supposing the input sequences had been random. This validation improved the accuracy of the YMF, and when the authors compared YMF with MEME and AlignACE, they reported an outstanding performance.

Another popular clustering technique in the pattern-driven approach is suffix tree. A Suffix tree is a lexicographically interconnected data structure built over all the suffixes of a string. It provides efficient access to all substrings of the string and can be constructed in linear time and space while exposing the internal structure of a sequence in a deeper way than any other data structure. Due to its versatility, no other data structure exists with diverse applications in string processing as the suffix tree. (Carlvalho et al, 2004, Apostolico et al, 2008, Pizzi et al, 2011).

A Suffix tree for a string *S* is a tree whose edges are labeled with strings, such that each suffix of *S* corresponds to exactly one path from the tree's root to a leaf. (Adebiyi and Kaufmann, 2002). By definition, a suffix tree ST of an *n*-character string S is a rooted directed tree with exactly *n* leaves, numbered from 1 to *n*. Each internal node, other than the root R, has at least two children and each edge is labeled with a nonempty substring of S. No two edges out of a node can have edge-labels beginning with the same character. The key feature of the suffix tree is that for any leaf *i*, the label of the path from the root to the leaf *i* spells out exactly the suffix of S that starts at position *i*.

It has been applied to fundamental string problems such as finding the longest repeated substring, finding all squares or repetitions in a string, computing substring statistics and string comparison. It is particularly useful for finding a small sequence of symbols in a large one, a common ancestor of two different strings and common substrings of two different strings. The suffix tree of a string *S* is a tree with O(n) nodes and edges and n leaves. Each leave in the suffix tree is associated with an index $i(1 \leq i \leq n)$. This is made possible by the concatenation of a special character $ usually called the sentinel which does not occur in the string *S*. Thus ensuring that no *Si* is a proper prefix of *Sj* and there is a one-to-one correspondence between S$ and the leaf of the suffix tree (Adebiyi, 2002).

The construction of a suffix tree in linear time is a problem already addressed by Weiner(1973), McCreight(1976) and Ukkonen (1992). Kurtz(1999), improved on this design by implementing the space efficient construction using the suffix links and hash table.

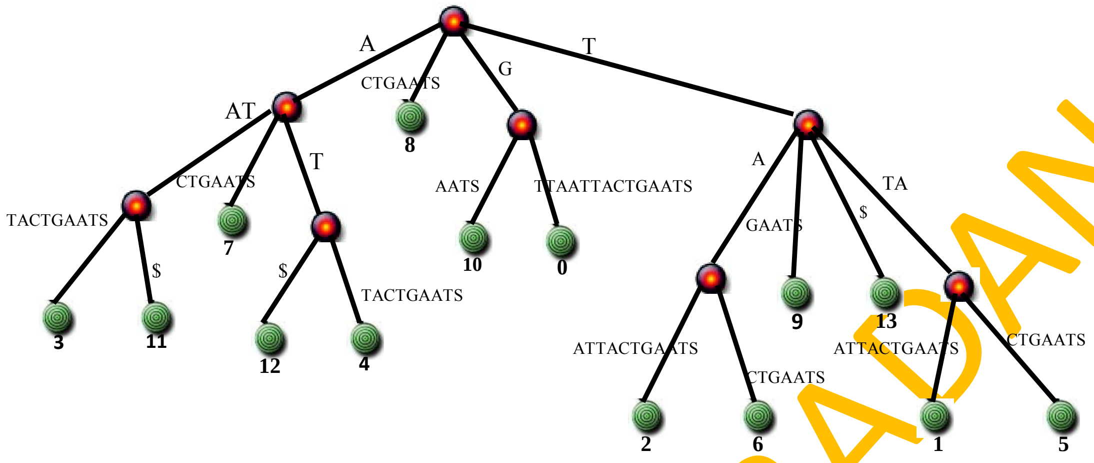


Figure 2.2: The suffix tree for DNA Sequence GTTAATTACTGAAT$

Some basic properties of the suffix tree are enumerated below:

1. The suffix tree is an acyclic tree, and a path of a node is a string from root to that node
2. A branch of the suffix tree may represent any nonempty substring of S.
3. Every suffix of the string S is represented by a leaf in the suffix tree
4. Each node of the suffix tree that is not a leaf, except the root, must have at least two offspring branches.
5. The length of an edge label of the suffix tree can be found in O(1) time
6. If a leaf of the suffix tree represents a string U, then every suffix of U is also represented by another leaf in the suffix tree.
7. If a node w is an ancestor of another node x, then the string that w represents is the prefix of the string that x represents.
8. If nodes $x_1, x_2, \ldots x_k$ represent the strings $X_1, X_2, .. X_k$ respectively, then the lowest common ancestor of $x_1, \ldots, x_k$ represents the longest common prefix of $V_1, .., V_k$
9. If two suffixes of S share a prefix, y, then they must share the path leading to the extended locus of y, the common prefix.

The suffix tree construction for a set of *N* input sequences, called a *generalized suffix tree*, can be easily achieved by consecutively building the suffix tree for each string of the set. The resulting suffix tree is built in time proportional to the sum of all the string lengths.

The first simple motif algorithm to use the suffix tree was developed by Sagot (1998**)**. The suffix tree was constructed and used to represent the sequences, returning all the traversal from the root node to the leaf node as unique patterns. The use of the suffix tree for preprocessing and organizing the input data resulted in an accelerated search for motifs. This implementation addressed to a large extent the speed bottleneck inherent in pattern-driven based methods. The algorithm was tested on DNA sequences of Yeast. This was followed by Apostolico *et al*.(2001) who developed the VERBUMCULUS algorithm and applied it to protein sequences. Eskin and Pevzner (2001) used a variant of the suffix tree called a mismatch tree to develop the MITRA algorithm which detected complex motifs with mutation successfully.

A very popular simple motif discovery tool that also used the suffix tree is the WEEDER algorithm by Pavesi et al.,(2001). WEEDER successfully identified motifs of unknown length in DNA and protein sequences.   A recent survey of Motif discovery tool by Mandas et al., (2007) revealed that WEEDER outperformed many other motif discovery tools such as MEME, AlignACE, ANN-Spec, Consensus, MITRA and MotifSampler**.** This success was attributed to its flexibility of parameter specification. Users are allowed to estimate apriori the probability of finding a given motif according to its length and the maximum number of mutations allowed for its occurrences. It also allows users to choose an optimal trade-off between time and accuracy: this ensured a judicious choice of prediction of motifs found in the data set, since only the strongest motifs would be reported as optimal.

The motif discovery tool developed in this research, STGEMS was compared with WEEDER in terms of speed and accuracy of identifying simple motifs. STGEMS outperformed it.

2.2.2.2 Structured Motif Discovery Tools using Pattern Driven-based Approach

Structured motifs are complex patterns with spaces separating them. The first structured motif extraction tool was developed by Marsan and Sagot (2000**).** They extended the simple motif extraction algorithm developed by Sagot(1998) to extract structured motifs. Their algorithm, SMILE proposed two solutions for extracting structured motifs on the suffix tree. In the first solution, the structured motif template consists of two components with a gap range between them. The algorithm starts by building a generalized suffix tree for the input sequences and then extracting the first component. In order to extract the other component, a jump is made in the sequences from the end of the first component to the second within the gap range. In the second solution, the suffix tree is modified temporarily so as to extract the second component from the modified suffix tree directly. SMILE proved inefficient in terms of its time and space complexity which were exponential in the number of gaps between the two components. The detail algorithm of Sagot(1998) for simple and structured motif mining on the suffix tree is provided in section 3.5.1.

Carvalho *et al*., (2004) attempted to reduce the time complexity during the extraction of the structured motifs by SMILE and developed a parallel algorithm, called PSMILE. PSMILE used the technique of partitioning the structured motif searching space, that is, the most demanding part of the algorithm was decomposed into a number of subprocesses that were loosely coupled and therefore could be executed simultaneously on different processors. This achieved a time speedup which is linear on the number of available processing units.

A year later, the same authors developed the RISO algorithm, an improvement on the SMILE algorithm. (Carvalho *et al.*, (2005). This improvement is in twofold: the first, instead of constructing the whole suffix tree for the input sequence, it built a suffix tree only up to a certain level, which was called the factor tree, this resulted in saving appreciable space. Secondly, a new data structure called box-link was introduced to store the information about how to jump within the DNA sequences from one simple motif component to the subsequent one in the structured motif. This accelerated the extraction process and avoided the exponential time and space consumption that prevailed in the case of SMILE. In RISO, after the generalized factor tree was built, the box-links were constructed by exhaustively enumerating all the possible structured motifs in the sequences and they were added to the leaves of the factor tree. Then the extraction process began, during which the factor tree was temporarily and partially modified in order to extract the subsequent simple motifs. RISO needed a lot of computation at this stage since the box-link construction, the structured motif occurrences were exhaustively enumerated and the threshold of the sequences was never used to prune the candidate structured motifs.

Pisanti *et al.*, (2006) provided an improvement on the RISO algorithm by developing the RISOTTO algorithm. RISOTTO incorporated boxlinks data structure with the suffix tree. While traversing the tree, RISOTTO adopted a depth-first visit of the motif tree and does not attempt to extend the node if the maximal length was determined or the quorum was no longer satisfied. The main improvement of RISOTTO on RISO was its ability to store information concerning maximal extensibility of factors. This was done in order to avoid extending motifs that are unlikely candidates. RISOTTO was shown to outperform RISO in terms of computational speed. However, it incurred an extra cost due to the space required to store the extensibility information. RISOTTO successfully identified structured motifs in yeast and bacteria respectively.

Another popular structured motif extraction tool is EXMOTIF by Yongqiqng Zang *et al*., (2006). EXMOTIF used a variant of the suffix tree, consisting of inverted index of symbol positions. This was used to enumerate all structured motifs by positional joins over the index. By considering the variable gap constraints at the same time as the joins were considered, an appreciable speed was achieved. The algorithm utilized a structured motif template defined based on some desired parameters such as length of motif and the gap range allowed. It used a hash table to store the computed motifs thus facilitating a speedy lookup and extracted all the repeated patterns after statistically validating them. EXMOTIF was reported to outperform RISO in both approximate and exact matching and superior to RISSOTO in showing the actual occurrences of the structured motifs instead of the relative frequency of the occurrence as obtained using RISOTTO.

Zare-Mirakaba et al, (2009) presented another structured motif discovery algorithm based on the suffix tree called MotifST (Motif finding using Suffix Trie). The algorithm uses a depth first search scheme to search for relevant motifs in a DNA sequence returning gapped motifs and motifs with mutations. The algorithm ran in linear time and when compared to other popular motif discovery tools like MEME and WEEDER, it had a better speed performance.

It is clear from this review that there is a need for structured motif discovery tool for the challenging sequence of P.falciparum and that influenced the STGEMS algorithm which when compared with these two popular structure motif discovery tools described above

((RISOTTO and EXMOTIF) had a better speed of execution and it also mined structured motifs from the challenging genomic sequence of the malaria parasite.

2.2.3 STATISTICAL BASED MOTIF DISCOVERY TOOLS

Statistical based method uses a two-phase iterative procedure where in the first step the likeliest occurrences of the motif are identified, and the second step adjusts the model for the motif which is usually represented by a position scoring weight matrix(PSWM) model based on the occurrences of the motifs determined in the previous step. In the first iteration the parameters of the initial model are usually set randomly. The limitation in this method is sensitivity to noise in the data and the fact that they are not guaranteed to converge to a global maximum since they employ some form of local search, such as Gibbs sampling, expectation maximization (EM) or greedy algorithms that may converge to a locally optimal solution.

There is no record of statistical based method for structured motif discovery in the literature; this is attributed to the complexity of the process involved in identifying the spaces that occur in structured motifs.

2.2.3.1 Simple Motif Discovery Tools based on statistical Approach.

The first simple motif tool based on statistical approach was developed by Hert et al.,(1990). The implementation was a greedy probabilistic model-based algorithm for discovering a matrix representation of sequences by finding the site with the highest information content. This algorithm has been fundamentally improved upon over the years and the latest implementation is called CONSENSUS by Hertz and Stormo(1999). The researchers provided a method to estimate the statistical significance of a given information content score based on large deviation statistics. They extracted DNA and protein patterns with statistically significant alignments of multiple sequences, stating that valuable insights could be obtained by aligning a set of related DNA, RNA or protein sequences and that such alignment could be useful in determining functional or evolutionary relationships.

An improvement on the greedy probability method is the Expectation Maximization which was introduced by Lawrence and Reilly (1990) and used Bailey and Elkan (1995) in developing MEME (Expectation Maximization Motif Elicitation). MEME is a popular simple motif discovery tool developed with the aim of discovering new motifs in a set of sequences where limited knowledge about the motif that could be present is available. MEME introduced three main novel ideas for discovering motifs: It used as starting point for the EM algorithm, the subsequences that actually occur in the sequences thus, increasing the probability of finding globally optimum motifs. It removed the assumption that each sequence contains exactly one occurrence of the shared motif and incorporated a method for erasing shared motifs after they are found, this ensures that several distinct motifs could be found in the same set of sequences. This is especially crucial in situations when different motifs appear in different sequences or when a single sequence contains multiple motifs. STGEMS was compared with MEME and it outperformed it.

An improvement on the MEME algorithm is the PHYME (Phylogenetic Motif Elicitation) by Sinha et al.,(2004). PHYME used EM technique and improved on it by the incorporation of an evolutionary model trained using Hidden Markov model. This combination increased the sensitivity of the algorithm appreciably. In addition, the phylogenetic tree output of the algorithm makes it easier to trace the genetic relationship of the sequences at a glance.

Gibb Sampling technique is another popular statistically method that has been used as the operating principles of many motif extraction algorithms. Roth *et al.,* (1998) implemented AlignACE (Aligns Nucleic Acid Conserved Elements) based on the Gibbs Sampling probability technique. Liu et al.,(2001) used a variant of Gibbs Sampling to design the BioProspector algorithm which is another variant of the Gibbs Sampling algorithm and an improvement on the algorithm of Roth *et al.,* (1998).

Gene Enrichmnent analysis is a potent statistical approach introduced by Eden et al (2007) and used to develop DRIM (Discovery of Rank Imbalanced motifs) algorithm. Gene enrichment analysis is the use of some statistical test (for example standard deviation, geometric mean, hypergeometric mean among others) to measure how enriched a particular set of gene is, in terms of its functional annotation compared to the other set of genes in the genome. DRIM incorporated the gene enrichment concept using the geometric mean and identified sequence motifs in lists of ranked DNA sequences. DRIM successfully identified simple motifs in the yeast genome, that is, novel transcription factor binding sites.

Young et al (2008) also used the gene enrichment technique and developed the GEMS algorithm **(**Gene Enrichment Motif Searching) incorporating the statistical test of hypergeometric mean instead of the geometric mean used by Eden et al. (2007). GEMS also introduced the position weight matrix optimization principle which improved the accuracy of the motifs discovered.
GEMS algorithm is not an ab-initio motif discovery tool, it requires an already existing cluster as candidate motif to perform gene enrichment analysis on. Therefore, it obtained the cluster for its analysis from the online Gene Ontology(GO) database.

www.geneontology.org. Gene Ontology is an online repository of sequence genes of different organisms maintained by the National Centre for Biotechnology Information (NCBI). www.ncbi.org.

GEMS algorithm is the only algorithm that had successfully identified simple motifs from the challenging sequence of malaria parasite, all the other algorithms had been able to identify motifs from the other model organisms[3] such as yeast and bacteria but not from malaria parasite. Nevertheless, GEMS's inability to identify structured motifs is a major concern. This is because important proteins in eukaryotic organisms like the malaria parasite exist as structured motifs. Moreover, GEMS could not obtain the initial gene cluster it needs as a starting cluster (candidate motif). It depends solely on the gene ontology database for this initial cluster. Consequently, GEMS is incapable of identify optimal simple motifs for organisms that do not have an entry in the gene ontology database.

Some motif discovery algorithm combine pattern-driven approach with statistical based approach while others combine machine learning and statistical methods. This is because the hybrid is capable of inheriting the desired features of the two approaches. (Modan et al, 2007, Pizzi et al, 2011, Makolo et al., 2011).

2.2.4 MOTIF DISCOVERY TOOLS BASED ON COMBINATORIAL APPROACH

A number of motif discovery algorithms combine two or more approaches to get a hybrid approach, which inherits desired features of the various approaches. This concept was reported by Modan et al., (2007) in their study on survey of motif discovery tools. (Kellis et al, 2004, Modan et al, 2007).

[3] A model organism is a non human specie that is extensively studied to understand particular biological phenomena, with the expectation that discoveries made in the organism model will provide insight into the workings of other organisms

A popular motif discovery tool in this category is MUSA (Motif finder with UnSupervised Approach) based on a combination of machine learning and statistical technique. (Mende et al, 2006). MUSA used a bi-clustering algorithm that operates on a matrix of co-occurrences of simple motifs and computed the statistical significance using position weight matrix. MUSA successfully identified complex biologically significant motifs with a performance that was independent of the composite structure of the motifs being sought. MUSA could be used as a standalone tool or as a tool to determine the parameters required to run other motif discovery tools already available because of its effective statistical significance assessment method. MUSA was validated both with synthetic and real data from yeast, and it was able to discover new biologically significant motifs that had eluded searches performed using other motif finders such as MEME and AlignAce. Mendes et al (2006).

BioProspector by Liu et al.,(2001) also combined Gibbs sampling statistical technique with a machine learning markvov model.

A common choice among researchers of motif discovery tools is a combination of pattern-driven and statistical-based methods since this approach guarantees that the sensitivity of the statistical based method be complemented with the speed efficiency of pattern-driven techniques. An example of this is the STEME (Suffix Tree and Expectation Maximization for Motif Elicitation) algorithm by Reid and Wernisch (2011**)**. It combined the suffix tree, a pattern-driven approach with Expectation Maximization, a statistical approach**.** The incorporation of the Suffix Tree improved the speed limitation of the expectation maximization based algorithms (such as MEME). STEME was demonstrated to have a better empirical run time than MEME for the same data set.

2.2.5 EVALUATION OF MOTIF INFERENCE TOOLS

It is evident from the review of motif inference tools provided in the session above that a large number of motif inference algorithms are available and it would be beneficial to users to have some guidance in choosing the best tools for their motif finding endeavor. However, performance comparison of different motif finding tools and identification of the best tools have proven to be a difficult task because tools are designed based on algorithms and motif models that are diverse and complex and the incomplete understanding of the biology of regulatory mechanism makes adequate evaluation a challenge.

A common practice among authors of various motif inference tools is to test their algorithm against a few available algorithms using both biological sequence data and synthetic data sets with planted motifs or some known experimentally extracted motifs. Thus, a comparative evaluation of the performance of the new algorithm compared with existing algorithms is reported. In the same vein, our novel computational inference technique was compared with five popular motif inference tools. In addition to comparing the predicted motifs with those extracted from biological experiments.

Since transcription associated proteins control gene expression, their identification is important in understanding the biology of the organism. Therefore, biologists and computer scientists have been very interested in identifying efficient computational tools for their prediction. This is the crux of this research. We therefore investigated different experiments and performance comparisons to guide our research. Below are some of such experiments and comparison.

Hu *et al*, 2005 conducted a comprehensive benchmark experiment for performance comparisons of five sequence-based motif finding algorithms using large datasets generated from the biological database (RegulonDB). Their study differs from the benchmark experiments of Tompa *et al*.(2005) in that Tompa *et al*.2005, allowed the algorithm developers to fine-tune the running parameters and reported the best results while Hu *et al*.2005 allowed minimal parameter tuning during performance evaluation. They also suggested that performance evaluation based only on the predictions with the highest score has the risk of penalizing some practically effective algorithms, since in many cases the predicted motifs with the highest score are not the motif with highest accuracy. The Five algorithms assessed by the authors were AlignACE, MEME, BioProspector, MDScan and MotifSampler. The authors defined a set of prediction performance indexes for the algorithms and conducted comparative evaluations of the algorithms in terms of their prediction accuracy, scalability and the reliability of their significance scores with the existing biological database. The prediction accuracy measures used by these authors were nucleotide level accuracy, binding site level accuracy and sequence and motif level accuracy. The study showed that the performance of the algorithms tested was low, with around 15 to 25% accuracy at the nucleotide level and 25 to 35% at the binding site level for sequences of 400 nucleotides long. However, the algorithms were capable of predicting at least one binding site correctly more than 90% of the time. Among the factors that affect the prediction accuracy, the sequence length was found to be the most critical; the performance of all algorithms degraded significantly as the sequence length increased.

We conclude this review of motif inference tools, by reporting that there is no evidence as to which category of motif inference algorithm performs better than the other. Pattern driven methods might perform well on some data set but poorly on another type of data set. The same applies to statistical based approach and machine learning approach. However, a better performance is achieved when a combination of two or more approaches are employed, since the algorithm will inherit the benefits of these approaches while reducing their limitations (Pizzi, 2011.). This recommendation informed the design framework of our novel motif inference algorithm STGEM, which incorporates the benefit of pattern driven approach using suffix tree data structures for improved speed, with the statistical approach of gene enrichment using hypergeometric function as well as with the machine learning component of implementing the similarity check mechanism of a multiobjective genetic algorithm to further confirm the effectiveness of our technique in mining structured motif from the challenging sequence of P. *falciparum.* The table 2.6 below shows a summary of the motif discovery tools reviewed in this research.

Table 2.6: A list of some popular Motif Discovery Tools

S/ N	Algorithm	Category	Operating Principle	Strengths	Weakness	Reference
1.	By Hert et al	SBA	Greedy Algorithm	Simple to implement	It is not time efficient	Hertz and Stormo(1990)
2.	MEME	SBA	Expectation maximization	Prior knowledge of the sequence is not required	It cannot run large data set at once	Bailey and Elkan (1995)
3.	AlignACE	SBA	Gibbs Sampling	Displays frequency of non site sequence at a glance	Not time efficient	*Roth et al, (1998)*
4.	CONSENSU S	SBA	Weight Matrix	Detects evolutionary relationship	Building the evolution tree takes time	Hertz and Stormo(1999)
5.	PhyME	SBA	EM	Shows evolutionary relationship at a glance	Extra time to construct the evolution tree	Sinha et al., (2004)
6.	Oligo-Analysis	PDA	Enumeration	Easy to implement	It cannot handle motifs with mutation	Van Helden *et al.* (1998).
7.	WEEDER	PDA	Suffix Tree	Allow flexible parameter specification	It can only return simple motif	Pavesi(2001)
8.	By Sagot	PDA	Suffix Tree	Improved speed	It can only return simple motifs	Sagot(1998)
9.	By Tompa (1999)	PDA	Enumeration	Good at discriminating randomly occurring motif	Cannot handle motifs with mutations	Tompa (1999)
10 .	Verbumculus	PDA	Suffix tree	Improved speed of execution	It can only return simple motifs	Apostolico *et al.* (2000)
11 .	SMILE	PDA	Suffix Tree	It can identify complex structured motifs	Space inefficient	Marsan and Sagot(2000)
12 .	YMF	PDA	Enumeration	Allow flexible parameter specification	It can only return simple motif	Sinha and Tompa(2000)
13 .	BioProspecto r	SBA & MLA	Gibbs Sampling and hidden markcov	Allows Multiple optimal motif detection	Very slow with large data set	Liu et al.,(2001)
14 .	DRIM	SBA	Hyper geometric Framework	Added feature of Ranking motifs	Too slow especially for large data set	Eden et al (2007)
15 .	GEMS	SBA	Gene Enrichment	Identified simple motifs in the malaria parasite	Cannot identify structure motifs	Young et al (2008)
16 .	MITRA	PDA	PrefixTree/Mis match tree and Graph	Allow preprocessing of sequences	Space inefficient	Eskinand Pevzner(2002)
17 .	PSMILE	PDA	Suffix Tree	Partitioning of search space that can run on parallel systems	Extra cost of space due to the partitioning	Carvalho et al (2004)
18 .	RISO	PDA	Box links and suffix tree	Additional speed gain due to boxlinks	Additional Space requirement for the box link	Carvalho et al (2005)
19 .	RISOTTO	PDA	Box links and suffix tree	Good for long complex motifs	Extra space need to store Extensibility information	Pisanti et al., (2006)
20	EXMOTIF	PDA	Inverted index of symbols and	actual occurrences of	Additional space	Zang and Zaki (2006)

			hash table	the structured motifs instead of the relative frequency	requirement for storing the symbols	
21.	FMGA	MLA	Genetic Algorithm	Can handle difficult search space	Time consuming	Liu et al. (2004)
22.	GAME	MLA	Genetic Algorithm	Return high fitness motif	Inconsistent in multiple runs	Wei and Jensen (2006)
23.	MUSA	MLA & SBA	Biclustering and PSSM	No need to specify parameter and can be used to determine the parameter needed for other algorithms	The speed is unacceptable especially for large data set	Mendes et al(2006)
24.	MOGAMOD	MLA	Multi Objective Genetic Algorithm	Handles multiple optimal motifs efficiently	It is time consuming	Mehmet Kaya (2007)
25.	By Mehme Kaya	MLA	Multi-objective GA	Can identify structured motifs	It is time consuming	Mehmet Kaya (2009)
26	MOTIFST	PDA	Suffix Tree	Fast	It cannot identify motif in the malaria parasite genome	Zare-Mirakaba et al. (2009)
27.	STEME	PDA &SBA	Suffix tree , Expectation maximization	Fast and very sensitive	Can only identify simple motifs	John E. Reid and Lorenz Wernisch (2011)

PDA stands for Pattern Driven Approach SBA stands for Statistical Based Approach, MLA stands for Machine Learning Approach.

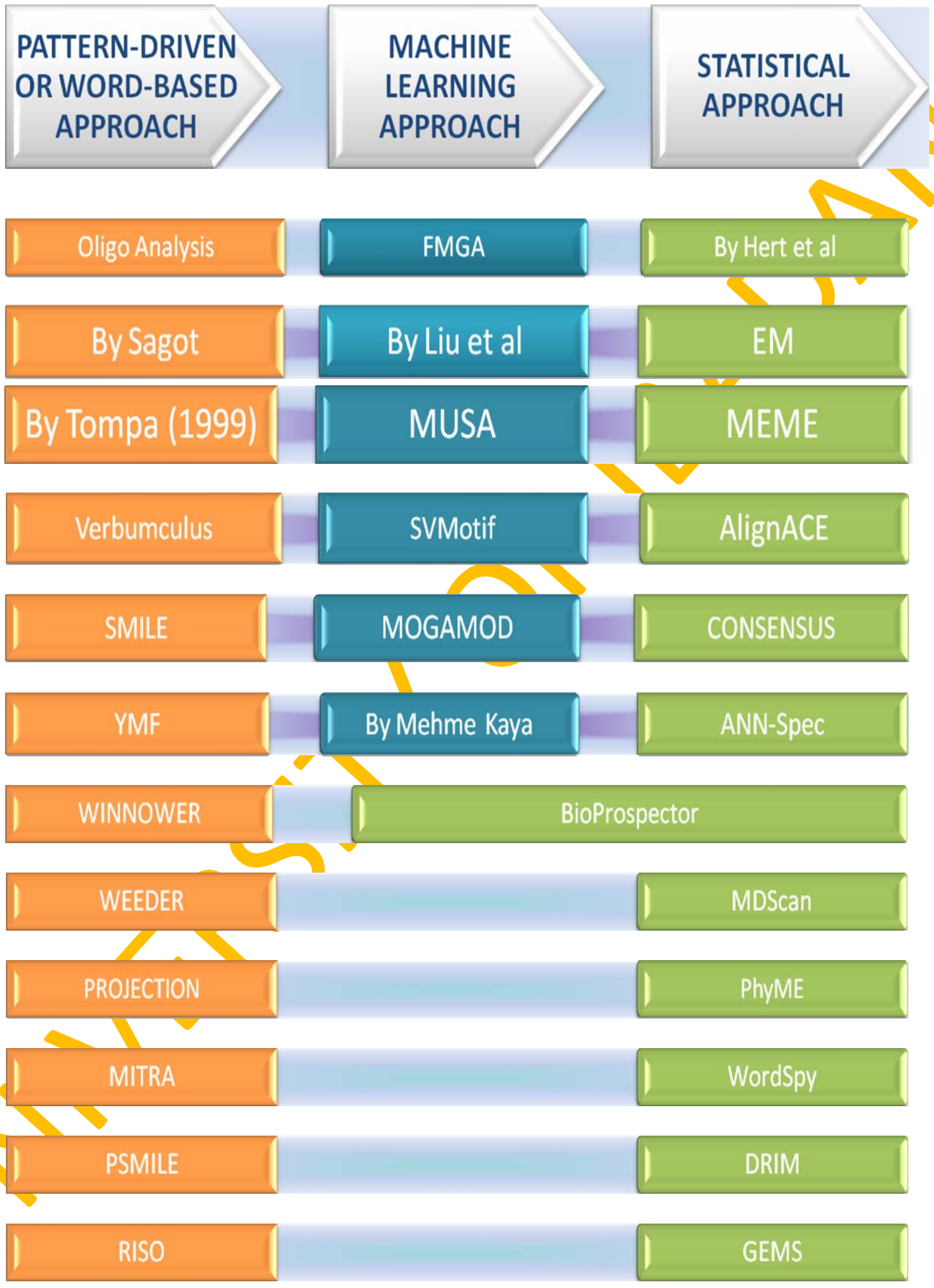


Figure 2.3: A Taxonomy of Popular Motif Discovery Tools

2.3 ESTABLISHMENT OF GAPS FROM REVIEW OF RELATED WORKS

Some of gaps identified in our review of existing motif discovery algorithms are as follows:

1. The need for a holistic motif discovery tool that takes input sequence from any organism and returns a viable list of optimal motifs. This is a limitation inherent in the popular GEMS algorithm by Young et al., 2008, which can only return optimal motifs based on candidate motifs with clusters in the gene ontology database.
2. The absence of motif discovery algorithm that can mine structured motifs from organisms with peculiarity in their genomic structure, for example the malaria parasite genome. Important genomic components such as the DNA binding site in eukaryotic organism occur as structured motif and needs to be mined.
3. The need for a motif discovery tool that in addition to handling genomic structure peculiarities incorporates the phylogenetic relationship. This will involve developing an evolutionary tree of the motifs from different organisms with the aim of providing evolutionary information.
4. The need for an integrated sequence analysis tool that combines gene finding, sequence alignment and motif discovery task.

This research, attempts to fill the gaps in 1 and 2 above which are the main limitations identified in the GEMS algorithm. The motivation for this research springs from these identified limitations and therefore proposed a computational inference technique for mining structures motif, an improvement on the GEMS algorithm.

2.4 SUMMARY AND CONCLUSION

In this chapter, we started by presenting the main concepts and theories of sequence analysis algorithms. We highlighted the three main classes which are Gene Prediction, Sequence Alignment and Motif Discovery Algorithms. We expounded the basic principles behind each class of algorithm, while focusing on motif discovery algorithms, our interest in this research. Our review of related works on motif discovery tools was based on the three main approaches adopted by most motif discovery tools namely: the pattern-driven, statistical and machine learning approach and we reviewed some simple and structured motif discovery tools based on these various approaches.

The variety of techniques adopted in the design paradigm of the various motif inference tools shows an increasing effort of researchers to develop efficient algorithms for genomic functions predictions. The efficiency of these algorithms is measured in terms of their time and space complexity. The important role played by the choice of data structure in the performance of the algorithm was also shown.

In the last part of the chapter, we explored the application domain of the research which is the malaria parasite, providing an insight into its genomic composition and the challenge of the malaria disease . Key among these challenges is the parasite's high adaptability enabling it to survive in the host in spite of the adverse conditions.

In the past, large-scale genomic data were not as available as in present times. The increasing rate of current availability necessitated the study of how to develop efficient tools for mining sequenced data with the aim of elucidating important information needed to understand the complex biological makeup of organisms. The understanding of the regulatory mechanism of the malaria parasite, attained through knowledge of the transcription associated protein provides an insight into the development of drug targets required to combat the various strains of drug resistant malaria in existence.

This research, which was motivated by the lack of existing motif discovery algorithm for mining structured motifs in the challenging sequence of the malaria parasite, proposed a new methodology that combined the suffix tree clustering technique with gene enrichment analysis using hypergeometric scoring function. The detailed methodology is discussed in chapter three.

CHAPTER THREE

RESEARCH METHODOLOGY

In the previous chapter of this write up, we have reviewed extensively the operating principles or approaches used in motif discovery systems. These include the pattern-driven, statistical and machine learning. We emphasized that tools based on a combination of approaches usually yield better results. This notion informed the methodology adopted in the design of STGEMS which in an attempts to fill the gap identified in the review of existing motif discovery tools, proposed a holistic algorithm for mining simple and structured motifs particularly suited for organisms with peculiarity in their genomic structure.

The other feature of this research is our attempt to capture motifs in the glycolytic metabolic pathway of the P.*falciparum*, which is the core of the malaria parasite's high adaptability and consequently, its high resistance to the existing anti-malaria drugs. We hope to bring to notice the importance of identifying these genomic elements responsible for understanding the vital biological processes of the deadly malaria parasite.

3.1 DERIVATION OF STGEMS ALGORITHM

STGEMS algorithm has its roots in sequence analysis algorithm. It belongs to the class of motif discovery algorithms. Basically**,** Motif discovery tools adopt one or more of these three approaches namely pattern-driven, statistical and machine learning. STGEMS is a combinatorial approach based on pattern-driven and statistical approach. Figure 3.1 below shows the derivation of STGEMS.

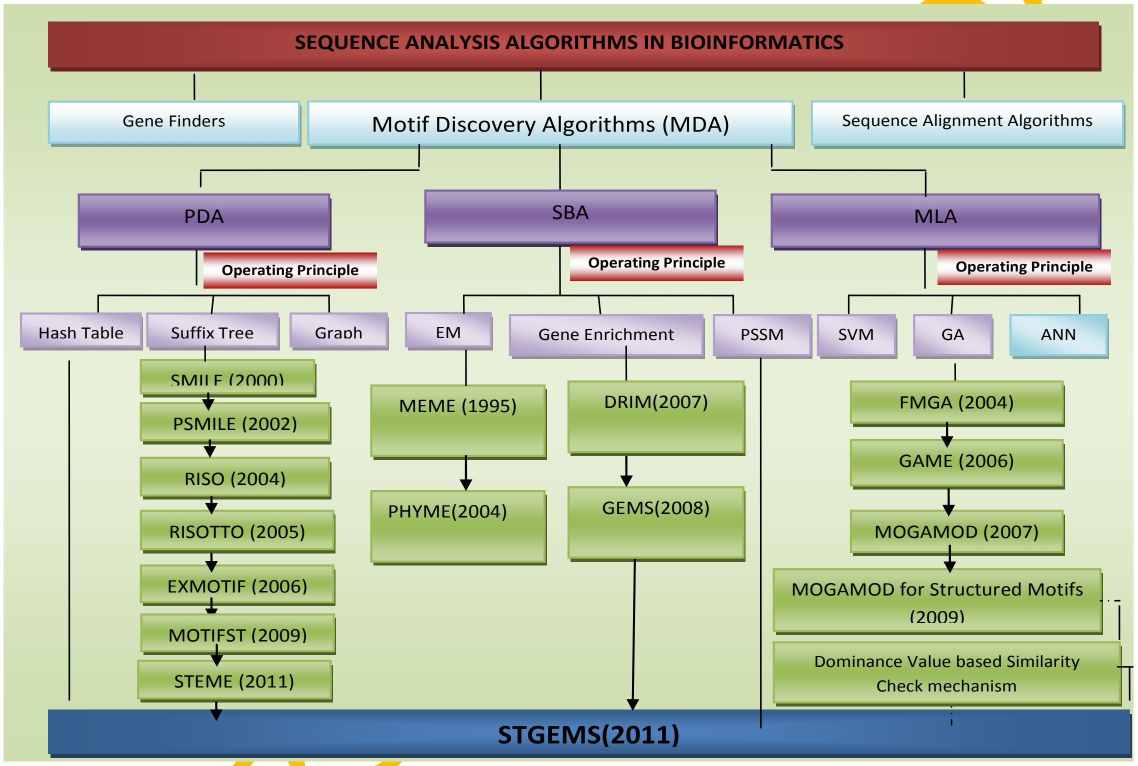


Figure 3.1 Derivation Tree of STGEMS Algorithm

STGEMS combined the clustering technique of the suffix tree with gene enrichment analysis, hence the origin of its name Suffix Tree Gene Enrichment Motif Searching (STGEMS). STGEMS is an improvement on the popular GEMS algorithm. It adopted the hypergeometric scoring function which is capable of identifying motifs from organisms with peculiar genomic structure such as the P.*falciparum* genome. STGEMS addressed the limitations inherent in GEMS by providing a holistic framework capable of predicting simple and structured motifs from any sequence data. It accepts any sequence data as input and outputs the optimal motifs present within the sequence. Unlike GEMS, STGEMS generates its own initial clusters using the suffix tree clustering technique. In addition, STGEMS is also capable of identifying simple and structured motifs from any organism especially, organisms with peculiarity in their genomic structure while GEMS only identified simple motifs. STGEMS also adopted the technique used by successful structured motif tools such as SMILE, EXMOTIF. This enabled it to successfully mine structured motifs.

As part of our research methodology, two popular similarity check mechanisms were implemented. The first is the similarity check mechanism based on position specific scoring matrix (PSSM) construction using hypergeometric scoring function (for example GEMS). The second constructed PSSM using dominance value of nucleotide (for example, MOGAMOD). The two methods were compared. The result of this comparison influenced the incorporation of the similarity check based on the hypergeometric scoring function into STGEMS' framework.

3.2 OPERATING PRINCIPLE OF STGEMS

The operating principle of the STGEMS algorithm is based on a combinatorial approach of pattern-driven using the suffix tree and the statistical approach of gene enrichment

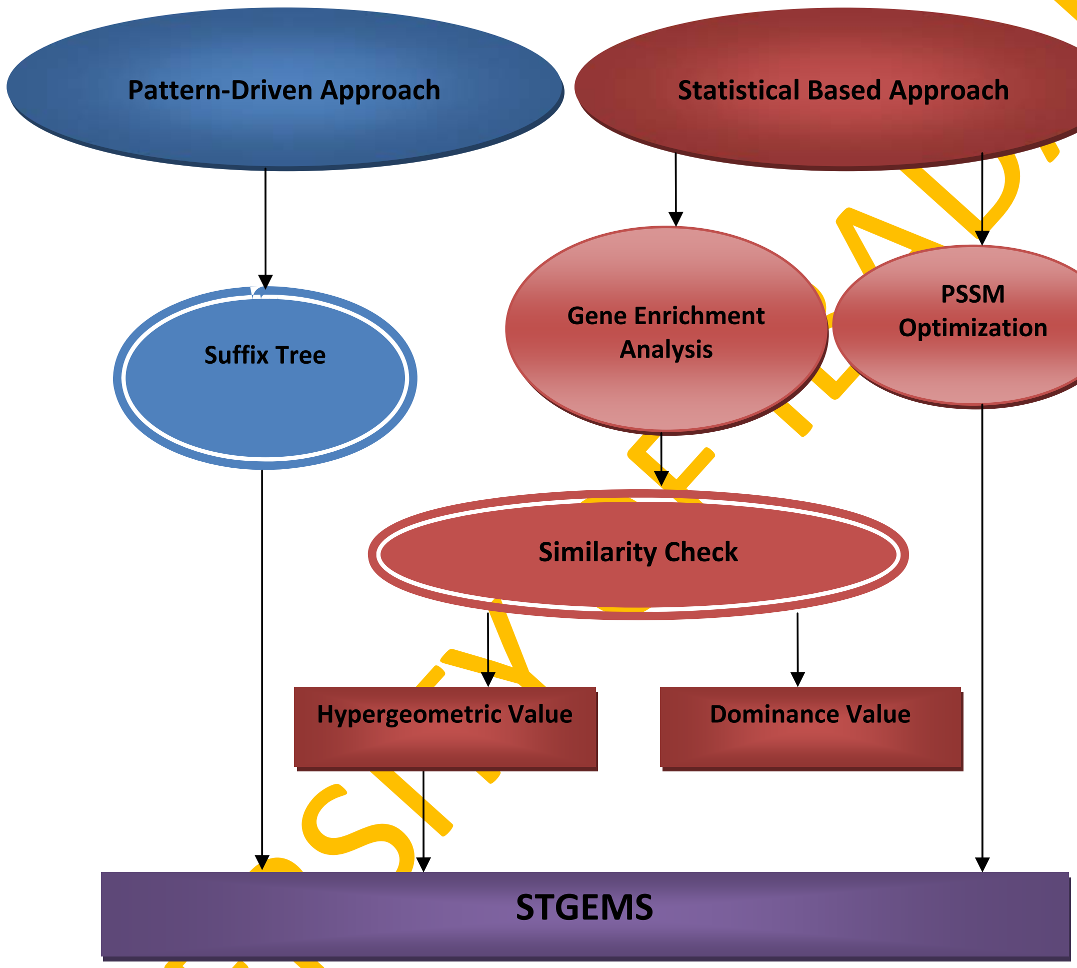


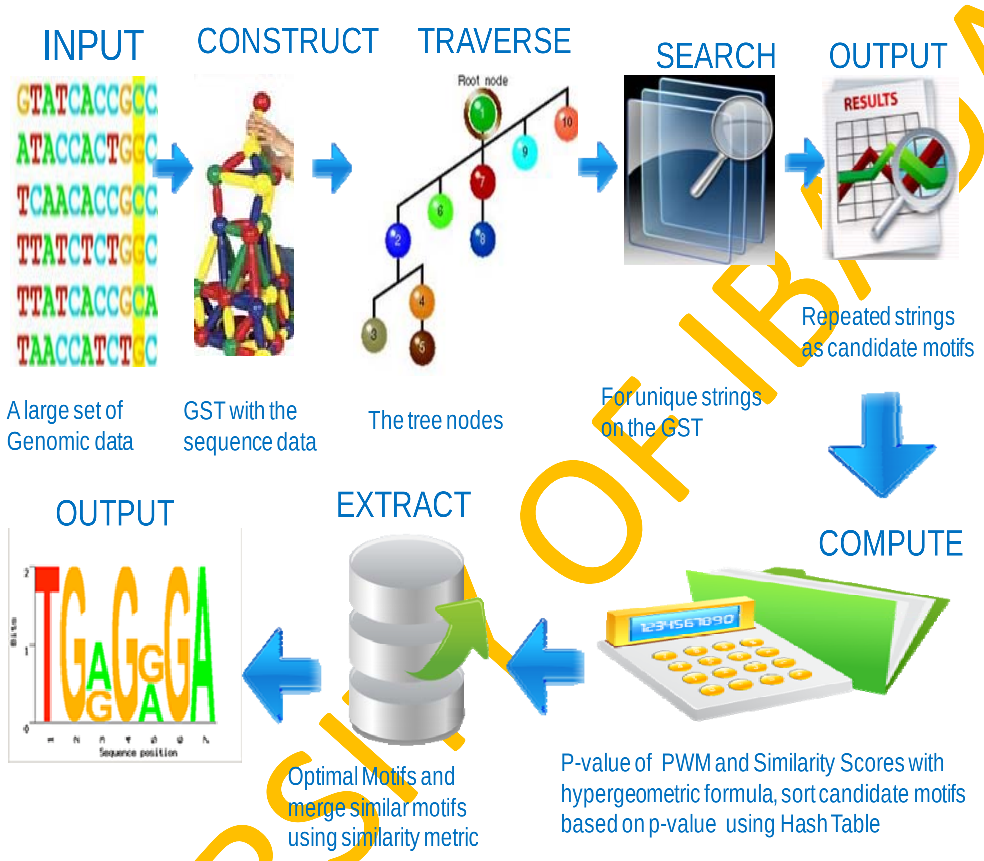
Figure 3.2 Schematic Representation of the Operating Principle of STGEMS

The suffix tree, has an inherent clustering mechanism that returns all repeated patterns (candidate motifs) at a remarkable speed. The justification for using the suffix tree data structure is because of its speed efficiency in searching for items that are subsets of an entire list. The suffix tree implementation adopted is the space efficient construction introduced by (Kurt et al, 1999) using suffix links.

The statistical significance of the extracted candidate motifs identified is computed by a gene enrichment analysis using the similarity check base on the hypergeometric scoring function which was complemented with position weight matrix optimization. This was used to rank the gene enrichment of the discovered motifs, thereby reporting only the optimal motifs. Similarity check is a measure of the degree of closeness of two strings. This is useful in computational biology where two slightly different patterns can represent the same motif due to the presence of a number of mismatches or mutation. For instance motifs AAAATGC, AACATGC, AAATTGC are similar motifs with one mismatch. Similarity score is used to filter off spurious motifs so that only optimal unique motifs are reported in the output. An implementation of similarity check mechanism based on dominance value of nucleotide could not identify relevant motifs from the challenging sequence of P*.falciparum*. This is due to its repetitive AT rich sequence which could result in the same motif being represented as multiple variants having the AT character repeated. The hypergeometric scoring function uses permutation and combination formula to eliminate the duplicates and reports only unique optimal motifs.

3.3 ARCHITECTURE OF STGEMS

The architecture of STGEMS is captured in the fig 3.3 and details of the logical flow of the process involve is described in fig 3.4.



INPUT
CONSTRUCT
TRAVERSE
SEARCH
OUTPUT
A large set of Genomic data
GST with the sequence data
The tree nodes
For unique strings on the GST
Repeated strings as candidate motifs
COMPUTE
P-value of PWM and Similarity Scores with hypergeometric formula, sort candidate motifs based on p-value using Hash Table
EXTRACT
Optimal Motifs and merge similar motifs using similarity metric
OUTPUT

Figure 3.3 Architecture of STGEMS

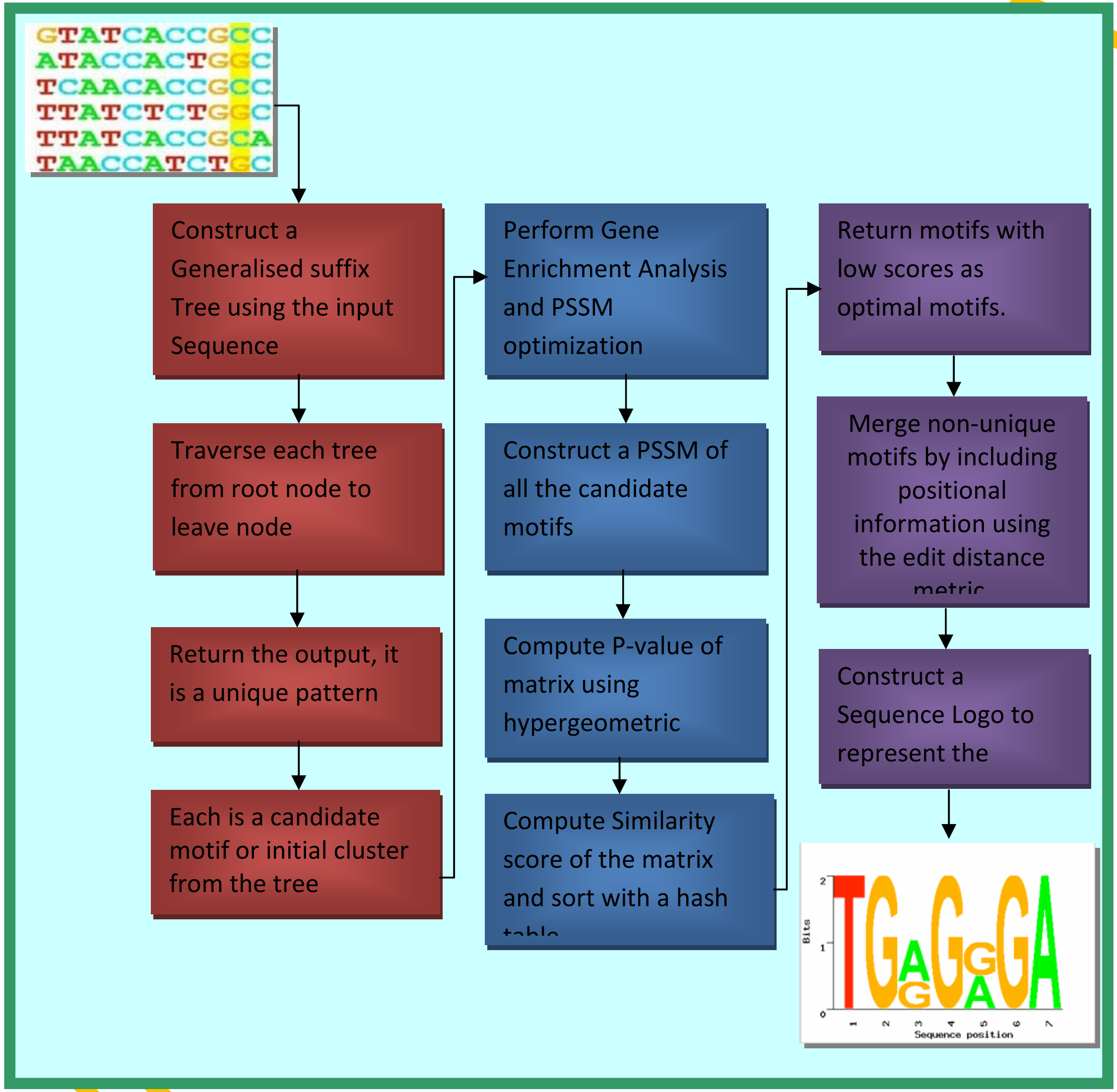

GTATCACCGCC
ATACCACTGGC
TCAACACCGCC
TTATCTCTGGC
TTATCACCGCA
TAACCATCTGC
Construct a Generalised suffix Tree using the input Sequence
Traverse each tree from root node to leave node
Return the output, it is a unique pattern
Each is a candidate motif or initial cluster from the tree
Perform Gene Enrichment Analysis and PSSM optimization
Construct a PSSM of all the candidate motifs
Compute P-value of matrix using hypergeometric
Compute Similarity score of the matrix and sort with a hash table
Return motifs with low scores as optimal motifs.
Merge non-unique motifs by including positional information using the edit distance metric
Construct a Sequence Logo to represent the
Bits
Sequence position

Figure 3.4 Logical Flow of STGEMS

STGEMS receives a list of DNA sequences as input, which contains unknown motifs that needs to be identified. This is used in the construction of a generalized suffix tree. The suffix tree is a data structure that is useful in representing a string or set of strings, they are well suited to algorithms that require efficient access to substrings by content rather than by position. (Adebiyi, 2002) The suffix tree construction reorganizes data into a form that facilitates searching and exposes sections of the strings that are repeated. In view of the fact that the core aim of this study is in searching for repeated patterns in a set of DNA sequences, the choice of the suffix tree data structure in the framework of STGEMS algorithm is adequate.

The tree is traversed to output unique patterns or candidate motifs. In the suffix tree construction, each traversal from the root node to a leaf node is a unique pattern. This is followed by the computation of position weight matrix (PWM) for the extracted unique patterns. The PWM or Position specific scoring matrix(PSSM) is a scoring matrix that shows the information content of the motifs, which depends on the frequency of occurrence of each of the characters in the identified pattern. Subsequently, the computation of the biological significance of the candidate motif is done by computing the similarity scores of the different motifs. The motifs with low similarity scores are reported as best optimal motifs.

The merging of similar motifs, that is, those with one or two variations in the character that make up the motifs is effected. These are merged using edit distance, before returning them as optimal motifs. The final output is the optimal motifs represented using a sequence logo.

A sequence logo is one of the standard output format for representing DNA or Proteins in Bioinformatics. Sequence logos are a graphical representation of an amino acid or nucleic acid multiple sequence alignment. It was first introduced by Schneider and Stephens (1990).

Each logo consists of stacks of symbols, one stack for each position in the sequence. The overall height of the stack indicates the sequence occurrence at that position, while the height of symbols within the stack indicates the relative frequency of each amino or nucleic acid at that position. To create sequence logos, related DNA, RNA or protein sequences, or DNA sequences that have common conserved binding sites, are aligned so that the most occurring parts create good alignments. A sequence logo can then be created from the conserved multiple sequence alignment. The sequence logo will show how well the nucleotides are conserved at each position: the fewer the number of nuleotides, the higher the letters will be, because the better the conservation is at that position. Different nucleotides at the same position are scaled according to their frequency. The height of the entire stack is the information measured in bits. Sequence logos can be used to represent conserved DNA binding sites.

Figure 3.5 shows a sequence logo diagram derived from counts of the nucleotide (the letters that made up the genes) in the translation initiation region of P.falciparum genes. Each letter is written in proportion to its frequency of occurrence. The letters are stacked together, If a nucleotide were used in all 25,000 genes it would be fully conserved and be drawn two bits tall. The most significantly biased nucleotides are -3 (A) and +4 (G).

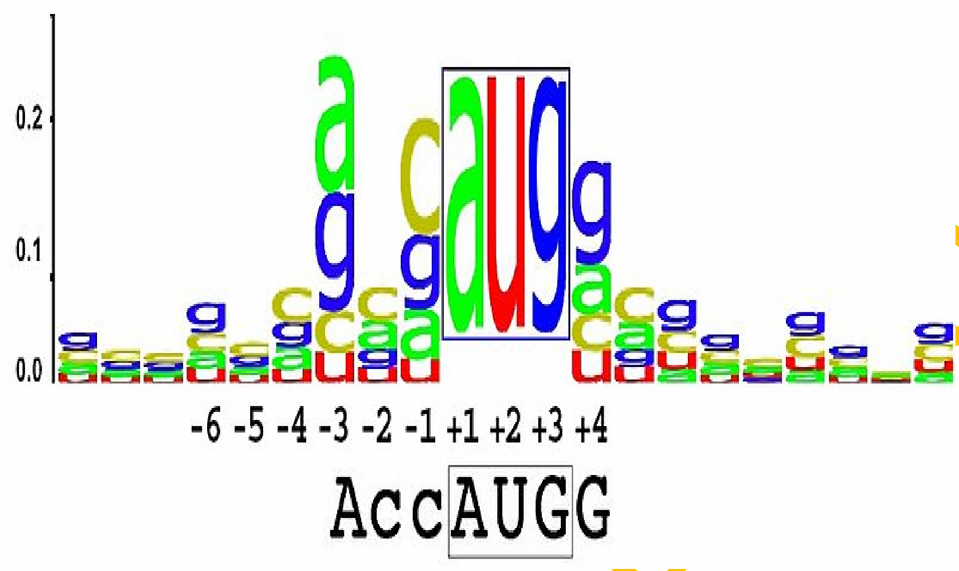


Figure 3.5: Sequence Logo

3.4 CONSTRUCTION OF SUFFIX TREES

The suffix tree implementation adopted in this research is the space efficient linked list construction by Kurtz(1999) . The algorithm for constructing the suffix tree ST is generally a successively insertion of suffixes of S$ from the longest to the shortest, into an initially empty tree ST_0 (which consist of the root only), in the following sequence of compact Σ^+ -trees: $ST_1,ST_2,...,ST_n,1,_{STn+1}$, where words($ST_i$)={ w Є Σ^* such that w is a prefix of S_j for some j Є [1, i] }.

Usually, a suffix tree is represented using *head* which is the longest prefix of the suffix tree ST_i while *tail* is the remainning suffix of ST_i such that , $head_i tail_i = ST_i$. Their locations and pointers on the tree can be denoted by *headloc, tailloc* and *tailptr.*

The heads and tails are represented by pointers into the tree and the input string S. An example of a successive construction of suffix trees for S = Tata is shown in fig 3.6 below and the procedure that encapsulate these processes are also given in fig 3.7

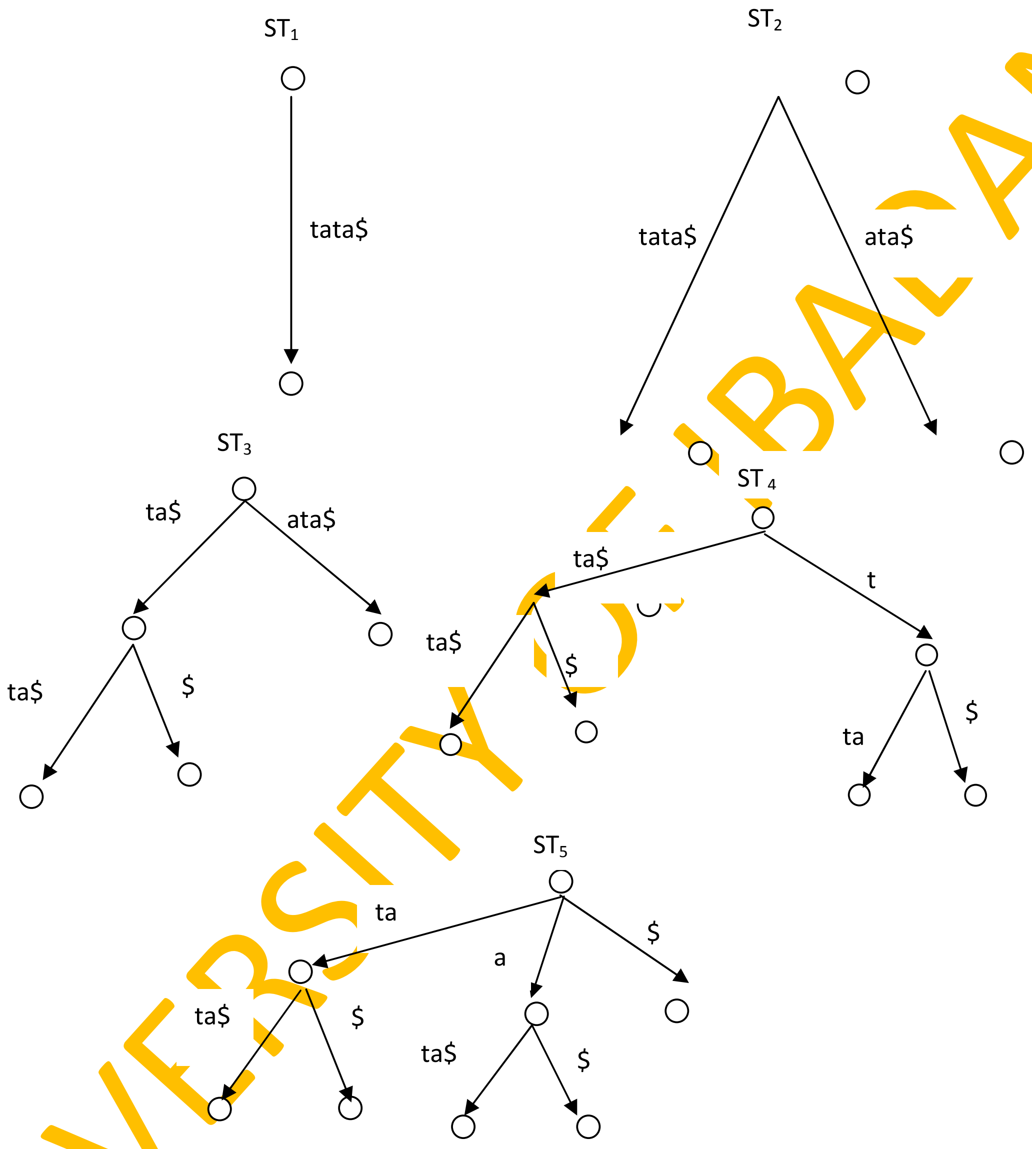


Figure 3.6: Iterative Construction of a Suffix Tree

3.4.1 COMPLEXITY ANALYSIS OF SUFFIX TREE CONSTRUCTION

The complexity analysis is achieved by examining the various components and processes involved in the construction of the suffix tree. The tree is constructed successively, by first inserting into the empty tree, then a decision is made based on whether the string exist on the tree or not, if it exist, it is not inserted, but if string is not already in existence on the tree, it is inserted, that is, a new leaf or branch node is appended in such a way that repetition is avoided. It follows that the best case scenario will be when the strings are not in existence and all that is needed is to insert leaf in depth first format while the worst case will be when there is need to search through the tree before inserting a leaf. Figure 3.6, shows that ST_1 is a compact Σ^+ -tree with only one edge from the root, that is, root+ S\$=$\overline{S_1}$. This can be constructed in constant time since it does not involve any iteration. The implementation algorithm for the suffix tree construction is shown in figure 3.7. Although the construction is involving, it is however, simple to implement. It is obvious that *headloc*$_i$ and *tailptr*$_i$ can be computed in constant amortized time. For the *if-case*, where *headloc*$_i$ = $\overline{u}$, then u = ε or $\overline{u}$ is a branching node in ST_{i-1} such that u = $head_i$ = $head_j$ for some j Є (2, i – l). This implies node $\overline{u}$ exists there already and what remains is to insert a new leaf edge labelled with *tail*$_i$. Therefore, it follows that operation *insertleaf* can be done in constant time. For the *if-case* where *head1oc*$_i$ = ($\bar{u}$, *av*) for some node $\overline{u}$ in ST_{i-1}, some character *a* Є Σ, and some string *v* Є Σ*, then $\overline{head}_i$ = $\overline{uav}$ does not exist in ST_{i-1}. Therefore, the *a*-edge $\overline{u}$+*avw* =$\overline{uavw}$ that exist in ST_{i-1} for some *w* Є Σ^+ has to be split into two, one representing the old *a*-edge $\overline{u}$+*avw* =$\overline{uavw}$ and the new edge $\overline{u}$+*av* = $\overline{uav}$, will be updated by insertleaf with $tail_i$. Operations required in splitedges involve only change of references and the creation of a new T_{branch}. It then simply follows, that operation splitedge can also be performed in constant time.

Therefore, the total running time of executing the *for loop* is O(n) since the iteration is a function of n. We can therefore infer that constructing a suffix tree can be carried out in linear time of the length of the input string.

It is important to note that, the node $\overline{head}_i$ is created with its head position *i.* This means that the branching nodes in ST are created in the order of their head position, and this is the order in which they are stored in table $_{Tbranch}$. Ultimately, a decision is made whether a branching node is small or large and then set the distance of a small node in operation split edge.

```
1. Procedure Construct(S)
2. Construct tree ST1
3. For $i = 2$ to n + 1 do//Construct $ST_i$ as follows
4. Compute $Headloc_i$
5. Compute $tailoc_i$
6. If ($headloc_I == \bar{u}$ ), where $\bar{u}$ is a node in $ST_{i-1}$
7. Insertleaf ($\bar{u}$, $tail$)$S_i$, $\longrightarrow$ where $\overline{S_i} = \bar{u}tail$
8. If ($head1oc_i$ == (ū, $av$)), $\exists$ There Exist an a-edge $\bar{u}+ avw = uavw$ in $ST_{i-1}$
9. Splitedge($\bar{u}$, $avw$) $\longrightarrow$ $\overline{uav}$ , $\overline{uavw}$
10. Insert leaf ($\overline{uav}$ , $tail_i$) $\longrightarrow$ $\overline{S_i}$
```

Figure 3.7 Kurtz(1999) algorithm for constructing a suffix tree ST.

Usually, a suffix tree is represented using *head* which is the longest prefix of the suffix tree ST_i while *tail* is the remainning suffix of ST_i such that , $head_i tail_i = ST_i$. Their locations and pointers on the tree can be denoted by *headloc, tailloc* and *tailptr.*

The heads and tails are represented by pointers into the tree and the input string S.

3.4.2 TRAVERSAL OF SUFFIX TREE WITH SUFFIX LINKS

The suffix link is an auxiliary structure that accounts for the most important acceleration element in the construction and traversal of a suffix tree. The suffix tree construction algorithm is based on the observation that constructing the suffix tree can be performed by iteratively expanding the leaves of a partially constructed suffix tree. Through the use of suffix links, which provide a mechanism for quickly traversing across sub-trees, the suffix tree can be expanded by simply adding the j+1 character to the leaves of the suffix tree built on the previous j characters. The algorithm thus relies on suffix links to traverse through all of the sub-trees in the main tree, expanding the outer edges for each input character.

Adding a new prefix to the tree is done by walking through the tree and visiting each of the suffixes of the current tree. The starting point is the longest suffix and then proceeds down to the shortest suffix, which is the empty string. Starting at the end of string S[*j* - 1 ..i] in the current tree, walk up at most one node to either the root or to a node *v* that has a suffix link from it; let □be the edge-label of that edge; assuming *v* is not the root, then traverse the suffix link from *v* to s(*v*); and walk down the tree from s(*v*), following a path labeled □to the end of S[1..*j*]; finally, extend the suffix to S[1..j+1] . Figure 3.8 shows a suffix tree with the connecting suffix links.

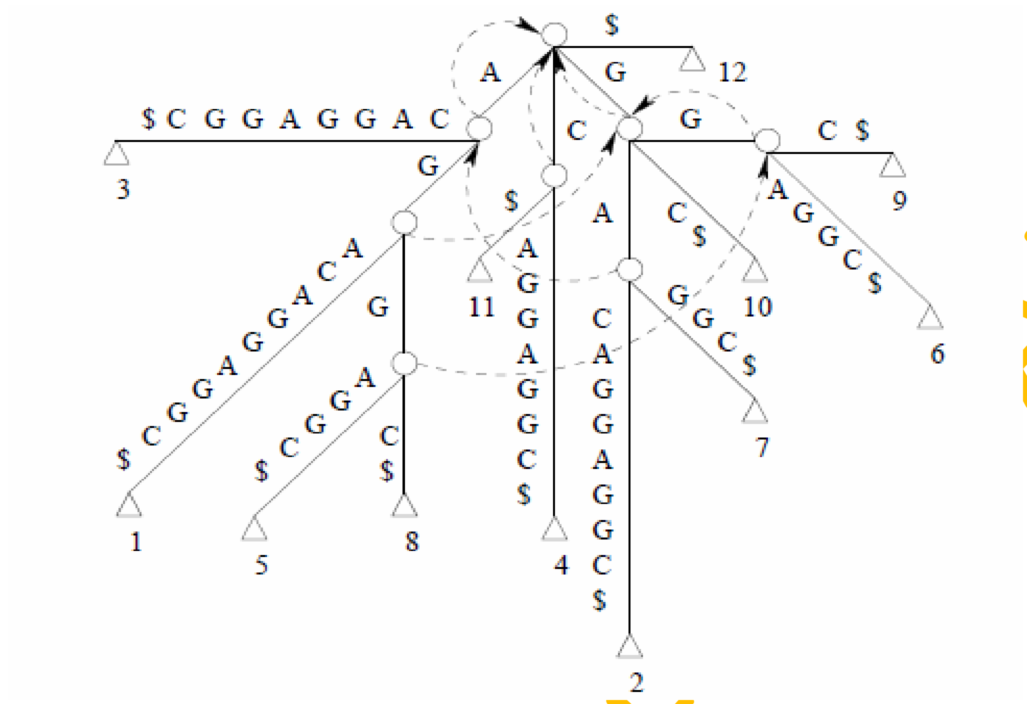


Figure 3.8 Suffix Tree with Suffix Links for String AGACAGGAGGC$.

Let string S[l..*i*] be *x* ε, where *x* is a single character and εis a (possibly empty) substring, and let (*v,* 1) be the tree-edge that enters leaf 1. The algorithm must find the end of string S[*2..i*] εin the current tree derived from *v*. The key is that node *v* is either the root or it is an interior node . If it is the root, then to find the end of ε the algorithm needs to walk down the tree following the path labeled ε.but if *v* is an internal node, then *v* has a suffix link out of it to node *s*(*v*). Furthermore, since s(*v*) has a path-label that is a prefix of string ε the end of string a must end the subtree of s(*v*). Consequently, in searching for the end of ε in the current tree, the algorithm need not walk down the entire path from the root, but can instead begin the walk from node s(*v*). This is the main advantage of including suffix links in the algorithm.

3.5 ALGORITHM FOR MOTIF EXTRACTION ON SUFFIX TREE

Simple motifs as well as structured motifs can be extracted using the suffix tree. Pisanti *et al*. (2006) presented algorithms with novel techniques for the extraction of simple and structured motifs using Suffix tree under the Hamming distance. However the speed of the algorithm was not remarkable. It is important to note that this has been done under the Edit distance by Adebiyi and Kaufmann (2002). The algorithms for simple and structured motif extraction are described below.

3.5.1 SIMPLE MOTIF EXTRACTION ALGORITHM ON SUFFIX TREE

The single motif extraction problem takes N sequences as input, q<N quorum, with a maximal number of error rate allowed and a minimal and maximal length for the motifs, k_{min} and k_{max}, respectively. The problem consists in identifying all motifs that occur in at least q input sequences. Such motifs are said to be valid. Sagot (1998) introduced an

efficient exact algorithm based on a suffix tree for extracting single motifs with mismatches. Usually, motifs are considered in lexicographical order starting from the empty word, and they are extended to the right as long as the quorum is satisfied until either a valid motif of maximal length is found, or the quorum is no longer satisfied. In both cases, a new motif is attempted. At each step, all nodes spelling e-occurrences of the current motif are taken into account.

A formal representation of a sketch of the algorithm of Sagot (1998) is shown in algorithm A below, where motif *m* is the one whose extension is being tried. At the beginning, ExtractSimpleMotif is evoked on the empty word. The algorithm recursively calls itself for longer motifs built by adding letters (step 4) and considers new ones (step1) when the extension fails (step2). A valid motif is spelled out whenever a motif whose length lies within the required minimal and maximal length is being considered (step 3). The order of generation of the motif is a depth-first visit of a complete tree *M* of all words of length K_{max} on the alphabet $\sum$. *M* is referred to as the motif tree and the algorithm does not need to allocate memory to it, only the suffix tree needs memory allocation. If we assume that the required length of the motif is *k* (that is $k_{min} = k_{max} = k$), and that at most e mismatches are allowed, the algorithm has worst case time complexity in $O(Nn_k v(e, k)$ where n_k is the number of tree nodes at depth *k,* and $v(e, k)$ is the number of words of length *k* that differ in at most *e* letters from a word *m* of length *k*. This value does not depend on *m*, and it holds that $v(e, k) \leq K^e |\sum|^e$. This upper bound is in practice not tight. However, a better bound cannot be obtained and therefore the time complexity is linear in the input size, but possibly exponential in the number *e* of mismatches. Since reasonable values for *e* are proportional to the value of *k*, this really places a practical bound on the length required for the motifs.

Algorithm A. Simple motif extraction

ExtractSimpleMotif(motif m)

```
1. for all $\alpha \in \sum$ do
2. if m $\alpha$ is valid then
3. if $|m\ \alpha| \geq k_{min}$ then spell out the valid motif (m $\alpha$ )
4. if $|m\ \alpha| < k_{max}$ then ExtractSimpleMotif (m $\alpha$ )
```

3.5.2 STRUCTURED MOTIF EXTRACTION ON SUFFIX TREE

A structured motif can be defined as an order of collection of simple motifs with gap constraint between each pair of adjacent simple motifs. For example the structured motif AT[115,136]GTCTATCG[121, 151]GTCGATGAC, has AT, GTCTATCG and GTCGATGAC as simple motifs and [115,136] and [121,151] as variable gap constraints, that is, ([minimum gap, maximum gap]) allowed between the adjacent simple motifs. More formally,

a structured motif is a pair (m, d) where $m = (m_i)_{1\leq i\leq p}$ is a p-tuple of single motifs and d= $(d_{min_i}, d_{max_i})_{1\leq i<p}$ is a (p-1)-tuple of pairs, denoting p-1 intervals of distance between the p single motifs. Each element m_i of a structured motif is called a box and its minimal and maximal length denoted by k_{min_i} and k_{max_i}, respectively.

The structured motif extraction problem takes as parameters N input sequences, a quorum $q \leq N$, p maximal error rate $(e_i)_{i\leq 1\leq p}$ (one for each of the p boxes), p minimal and maximal lengths $(k_{min_i})_{i\leq 1\leq}$p and (k_{max_i}) $)_{i\leq 1<}$p (one for each of the p boxes), and p-1 intervals of distance $(d_{min_i}, d_{max_i})_{i \leq 1 \leq p-1}$

With these parameters given, the problem involves searching for the contents of the boxes, which is the motifs, that have the structure defined by the parameters above and that satisfy the quorum. The algorithm for single motif extraction introduced in Sagot (1998) is the ancestor of a couple of others that infer structured motifs.

The optimization that was introduced can be applied to any of them.

In summary, the algorithm first builds the factor tree T of the input sequences, then it searches for all valid motifs of length at least k_{min} and up to k_{max} and after updating the data structure checks whether there is a second valid motif with the required interval between them.

From a formal perspective, the algorithm is described by Algorithm B assuming for simplicity that p = 2, where the motif *m* is the one whose extension is being attempted, and the value *i* indicates whether we are dealing with the first or the second box. Finally, λ denotes the empty word.

Algorithm B. Structured motif extraction

```
ExtractStructuredMotif(motif m)
1. for all $\alpha \in \sum$ do
2. if $m\ \alpha$ is valid then
3. if $|m\ \alpha| \geq k_{min}$ then spell out the valid motif (m $\alpha$ )
4. if $i = 2$ then spell out the valid motif
5. else update  $T$  ExtractStructuredMotif($\lambda$, 2)
6. if $|m\ \alpha| <$  then ExtractStructuredMotif (m $\alpha$ ,$i$)
```

3.6 STGEMS PROCEDURE FOR EXTRACTION OF SIMPLE AND STRUCTURED MOTIFS

The procedure below shows the extraction of simple and structured motifs. Following the extraction of Simple motif (SIM) with significant speed-up on the generalized suffix tree, it was then extended for the extraction of Structured motif (STM).

Procedure STGEMS(N)

```
1.    For all $\alpha \in \sum$ do
2.     For $i$ = 1 to N
3.    GST ← Construct tree(N)
4.    SIM ← ExtractSimpleMotif(N)
5.     While q > 2
6.     STM ← ExtractStructuredMotif(N)
7.    ValSIMs ←P-value(GEA)
8.    ValSTMS ←P-Value(GEA)
```

Where

GST: Generalised suffix tree;

SIM: Simple motif;

STM:Structured Motif;

q: quorum specif ied for valid structured motifs

ValSIMs: The valid simple motifs;

ValSTMS: valid candidate structured motifs

Line 2 to 4 mined all unique length of strings . In GST(Generalised Suffix Tree), these are words ending at the leave node. Line 7 and 8 among other things, implemented the GEA(Gene Enrichment Analysis) on GST output by the computation of PWMs from candidate the motifs and computing the similarity of any given motif in a promoter region to the PWM. Next, a similarity threshold was selected to determine how similar any motif in a promoter region must be to the PWM to be considered an actual instance of the motif or regulatory element. And lastly, this threshold was used with PWMs to extract optimal motifs. It is important to note that the computation of PWM from a seed demands the identification of all sequences differing from the seed word by one mismatch.

Since STMs are combination of boxes (SIMs), line 6 implemented the connection of these boxes on the suffix to determine STMs based on the quorum specified.

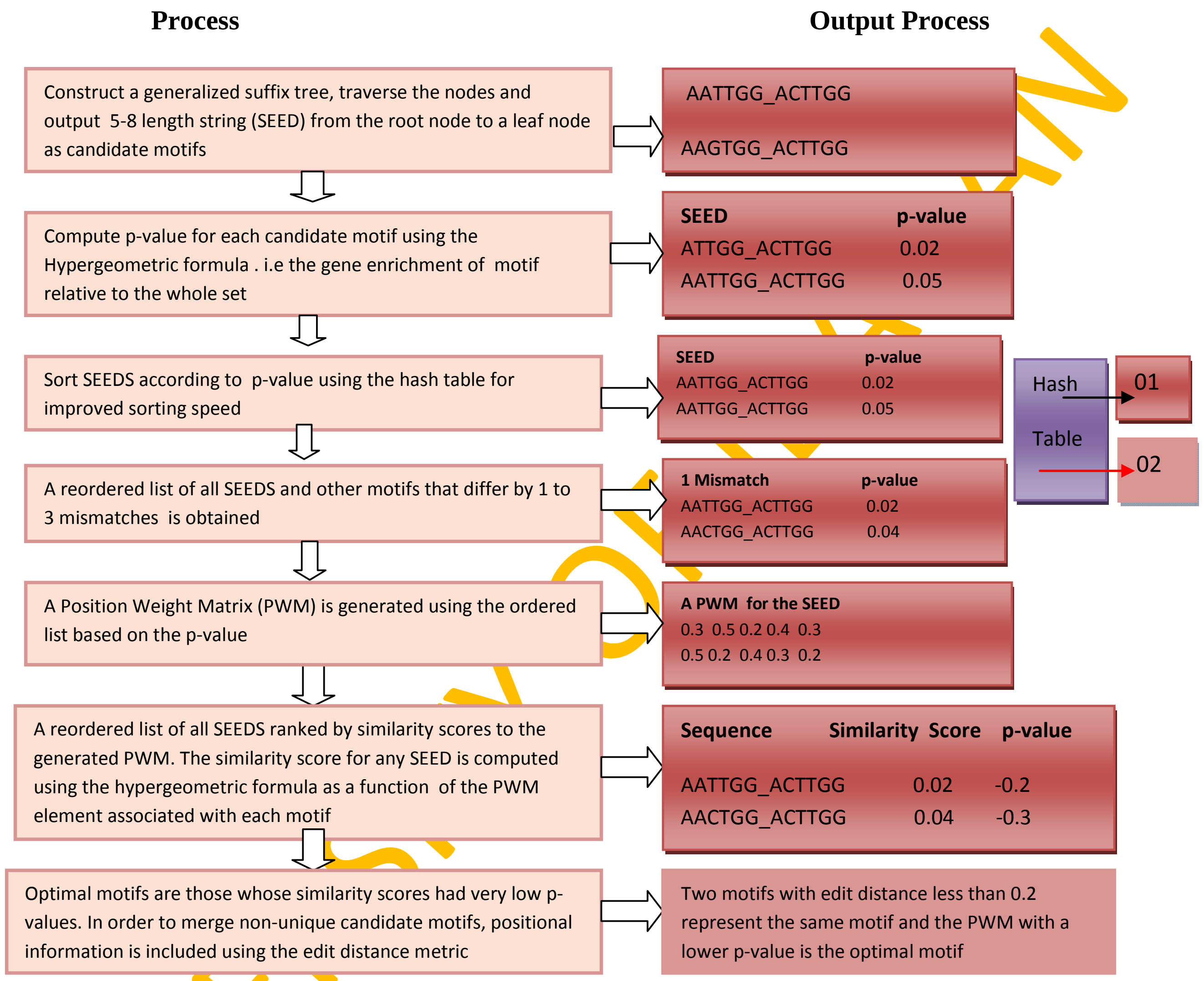
3.7 GENE ENRICHMENT WITH HYPERGEOMETRIC SCORING FUNCTION

STGEMS implementation details involves the extraction of all unique words of 5-8 length occurring in the sequence space, this was done by outputting all unique SEED from the root node of suffix tree to the leaf node, then a *p*-value enrichment score is computed using a hypergeometric formula below.

$$P(X,x,Y,y) = \sum_{i=y}^{\min(x,Y)} \frac{\binom{x}{i}\binom{X-x}{Y-i}}{\binom{X}{Y}}$$

Where X is the total set of genes, that is, positive and negative set, *x* a subset of the gene of interest, Y is the total promoter sequence that matches the genes, *y* is the subset of the promoters which fall within the cluster of interest. The hypergeometric formula is a standard statistical test used for gene enrichment analysis. It is a test that specifies whether a particular gene set is enriched for any functional annotations out of the full set of genes in the genome. The hypergeometric p-value equals the probability of finding *y* matches if one randomly select *Y* genes out of the total *X* gene collection. The smaller the p-value score for a candidate motif, the higher the likelihood of it being an optimal motif. The computation result produced a long list of words with associated *p*-values representing the probability of word enrichment in the entire sequence. The next stage consist in listing the words in ascending order with the most enriched candidates (lowest *p*-values) serving to seed the construction of PWMs one at a time. The hash table data structure was used in implementing the sorting of the words with the aim of achieving an improved speed.   All sequences differing from the seed word by one mismatch were then identified and re-listed by ascending p-value, before generating a PWM by individually weighing each word by its p-value score into the PWM.

The resulting PWM represents the probability of any given nucleotide occurring at a corresponding location in the candidate motif. The similarity of any sequence can be compared to the PWM through the calculation of a similarity score, which is the geometric mean of the corresponding matrix elements associated with the sequence. The similarity threshold selected determines the level of similarity that any given candidate motif must be to the PWM for it to be considered a true motif. The algorithm also adopts an optimal similarity threshold approach instead of using trial and error to guess the threshold for each candidate motif. This was achieved by first sorting all words by similarity to the PWM, then the *p*-values were re-calculated as more dissimilar words to the PWM were considered as motif instances using the hypergeometric scoring function and eventually identifying the similarity threshold that led to the lowest possible *p*-value. This entire process was repeated from the original seed word using two and three mismatches up to 40% of the word size to optimize mismatch levels in addition to similarity thresholds. The similarity and mismatch parameters that resulted in the lowest *p*-value were considered the best representation of a candidate motif. In addition, positional information using the edit distance metric was applied to merge non-unique candidate motifs, thus preventing repeated sequences being represented as new motifs.



Process

Output Process

Construct a generalized suffix tree, traverse the nodes and output 5-8 length string (SEED) from the root node to a leaf node as candidate motifs

AATTGG_ACTTGG

AAGTGG_ACTTGG

Compute p-value for each candidate motif using the Hypergeometric formula . i.e the gene enrichment of motif relative to the whole set

SEED	p-value
ATTGG_ACTTGG	0.02
AATTGG_ACTTGG	0.05

Sort SEEDS according to p-value using the hash table for improved sorting speed

SEED	p-value
AATTGG_ACTTGG	0.02
AATTGG_ACTTGG	0.05

A reordered list of all SEEDS and other motifs that differ by 1 to 3 mismatches is obtained

1 Mismatch	p-value
AATTGG_ACTTGG	0.02
AACTGG_ACTTGG	0.04

A Position Weight Matrix (PWM) is generated using the ordered list based on the p-value

A PWM for the SEED

0.3	0.5	0.2	0.4	0.3
0.5	0.2	0.4	0.3	0.2

A reordered list of all SEEDS ranked by similarity scores to the generated PWM. The similarity score for any SEED is computed using the hypergeometric formula as a function of the PWM element associated with each motif

Sequence	Similarity Score	p-value
AATTGG_ACTTGG	0.02	-0.2
AACTGG_ACTTGG	0.04	-0.3

Optimal motifs are those whose similarity scores had very low p-values. In order to merge non-unique candidate motifs, positional information is included using the edit distance metric

Two motifs with edit distance less than 0.2 represent the same motif and the PWM with a lower p-value is the optimal motif

Figure 3.9 Gene Enrichment Process

3.8 ANALYSIS OF THE STGEMS ALGORITHM

The STGEMS algorithm was analysed based on the time complexity, predictive accuracy and its sensitivity. The three sections below describes the techniques used in this analysis

3.8.1 TIME COMPLEXITY ANALYSIS

The analysis of STGEMS is done in terms of the time complexity involved in the running of the algorithm. In figure 3.8 above, it is clear that Line 2 can be run in O(N). It is important to point out that when the extraction of the seed is not implemented using the suffix tree it results into a $O(N^2)$ run time. Assuming the number of unique seeds is O(N), then from Sagot (1998), the computation of PWMs from seeds can be done in $O(NK|\sum|)$.

Where N is the length of the string, k is the size of the alphabets and $\sum$ is finite ordered set of alphabets.

In the worst case, the asymptotic run time of the functions embedded in Line 3 will cost $O(N^2)$, while on the average case (which is the situation in practice), this can be done in O(N). Line 4 also can be done in the worst case, $O(N^2)$, and in the average case O(N).

Therefore, the average case run time of STGEMS is $O(NK|\sum|)$, while on the worst case, it is $O(N^2)$.

3.8.2 PREDICTIVE ACCURACY

The predictive accuracy of motif inference algorithms are usually determined by computing the correlation coefficient as defined originally by Mathews (1975) and later adapted to the problem of gene finding evaluation by Burset and Guigo (1996) as

Mathew Correlation Coefficient (MCC) which can be calculated directly from the confusion matrix using the formula below:

$$\mathrm{MCC} = \frac{TP \times TN - FP \times FN}{\sqrt{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}}$$

Where TP is true positive, TN is the true negative, FN false negative and FP is false positive. The correlation coefficient was computed over four different data sizes. The Matthews correlation coefficient is used as a measure of the quality of binary classifications. It takes into account true and false positives and negatives and is generally regarded as a balanced measure which can be used especially when the classes are of very different sizes. The MCC is in essence a correlation coefficient between the observed and predicted binary classifications; it returns a value between −1 and +1. A coefficient of +1 represents a perfect prediction, 0 an average random prediction and −1 an inverse prediction.

A confusion matrix is a table that contains information about actual and predicted classifications done by a classification system. Performances of such systems are usually evaluated using the data in the matrix. It shows the relationships between true and predicted classes and it is used in evaluating the performance of a predictive algorithm in a supervised learning system. Our computational inference technique, STGEM is similar to a supervised classification in its output. The patterns or motifs predicted are compared with those obtained using biological experiments to confirm the authenticity of the algorithm in mining biologically relevant motifs. The confusion matrix returns 1 for true positive in cases where the predicted motif by STGEMS is the same as that extracted by the biological experiment and 0 if it is not the same. It returns 1 for true negative if the motif considered was not found either by STGEMS and the biological experiments and 0 otherwise.

The sensitivity of STGEMS was demonstrated by implementing a system using the method used in MOGAMOD as a benchmark to show the accuracy of STGEMS.

3.9 IMPLEMENTATION OF SIMILARITY CHECK MECHANISM OF MOGAMOD

The similarity check used in MOGAMOD measures similarity among all motif instances defining an individual solution using the dominance value of the nucleotides. In our implementation of the similarity mechanism, we started by first generating a position weight matrix from the motif patterns found by our generalized suffix tree in every sequence. Then, the dominance value (dv) of the dominant nucleotide in each column is found using this formula: $dv(i) = max\{f(b,i)\}$, 1,…., *l* where $f(b, i)$ is the score of the nucleotide b on column *i* in the position weight matrix, $dv(i)$ is the dominance value of the dominant nucleotide on column *i*, and *l* is motif length. The similarity objective function of motif M is the average of the dominance values of all columns in the position weight matrix.

$$Similarity(M) = \sum_{i=1}^{l} \frac{dv(i)}{l}$$

The likelihood of the candidate motif been discovered as a real motif depends on the value of the similarity score. In other words, the closer the value of the similarity M is to one, the greater the probability that the candidate motif M will be discovered as a true motif.

This similarity check approach used here differs from that used in GEMS algorithm and it proved ineffective in extracting optimal motif for the base-biased sequence of P.*falciparum*. The process in the implementation of the similarity check mechanism is encapsulated in figure 3.10 below

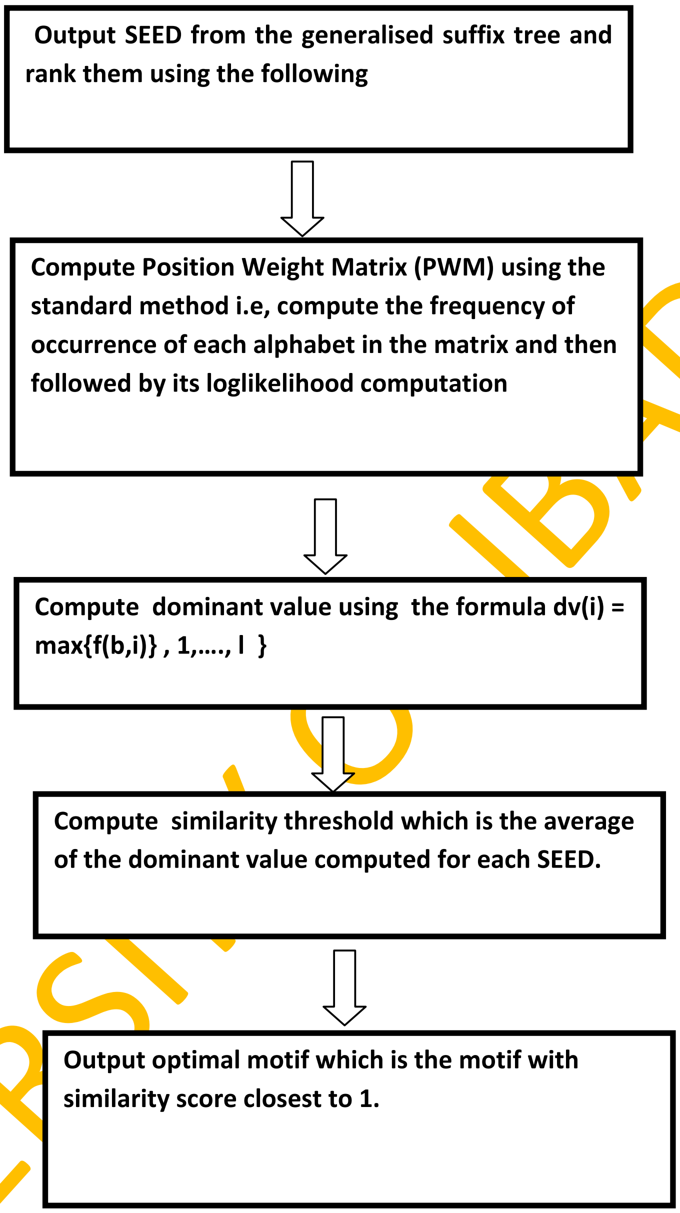

Output SEED from the generalised suffix tree and rank them using the following
Compute Position Weight Matrix (PWM) using the standard method i.e, compute the frequency of occurrence of each alphabet in the matrix and then followed by its loglikelihood computation
Compute dominant value using the formula dv(i) = max{f(b,i)} , 1,…., l }
Compute similarity threshold which is the average of the dominant value computed for each SEED.
Output optimal motif which is the motif with similarity score closest to 1.

Figure 3.10 MOGAMOD Similarity Check Implementation

3.10 STGEMS ANALYSIS USING MALARIA PARASITE GENOMIC DATA

Two different types of data set from the malaria parasite genome were used in running STGEMS in an attempt to extract useful information about the biology of P.falciparum. The first aims at identifying transcription associated proteins while the second identifies motifs in the glycolytic metabolic pathway. A brief description of the structure of genomic data and the importance of the biological elements mined in this research is provided in the sections below. We also provides an overview of the glycolytic metabolic pathway, an essential pathway in malaria parasite since it is fundamental to the survival of the parasite in the two hosts it depends on.

We recall that, *in vivo(wet lab* experiment*)* methods for DNA binding site predictions are very expensive and labour intensive. In addition the methods could not identify all the binding sites of a transcription factor (Barash *et al.,* 2003, Pizzi et al., 2011), making computational methods a good alternative.

Genomic data consist of a sequence of the alphabet {A, C, G, T} for *DeoxyriboNucleic Acid (*DNA) and {A, C, G, U} for RiboNucleic Acid (RNA). Each alphabet stands for a biological molecule. These sequences are called genes and they are the unit of information storage and transfer in living organisms.

It is important to note that the malaria parasite genome has a peculiar genomic sequence. Its genome has an AT content of about 90%. This AT composition is very high in comparison to other organisms. This peculiarity necessitates the development of a suitable motifs inference algorithm that puts the malaria parasite genome in good perspective. This is necessary to prevent false motifs which are copies of the same motif repeated with high ‘AT’ pattern being returned as valid motifs. STGEMS in incorporating the hypergeometric scoring function was able to mine valid transcription factors and DNA binding site from malaria parasite.

Transcription factors are proteins that bind to particular sequences upstream of genes called DNA binding sites. They either activate the transcription by assisting RNA polymerase binding or they inhibit it by blocking RNA polymerase binding. Through this process they control and regulate gene expression.

DNA binding sites are very short sequences (structured motifs of 6-20 base-pairs). Such sequences can appear anywhere in the genome without having the regulatory functionality. Only a small fraction of these sequences are actually *bona fide* targets of the transcription factor. Schneider *et al.*, (2007). Hence, methods to determine binding site suffer from a high false positive rate. The actual true positive prediction can only be verified by conducting some experiments such as, DNA footprinting, Chromatin Immuno-Precipitation (chIP), DNA-protein crosslinking (DPC), or X-ray crystallography (Miller et al., 2002) where each binding site is verified individually (Del *et al.*, 2007).

STGEMS was also applied to the glycolytic metabolic pathway, with the aim of identifying relevant motifs responsible for gene interactions at that level. This pathway is of special interest in the malaria parasite since it provides the energy needed for the survival of the parasite in the two hosts it inhabits. (Planes and Beasley, 2009). Thus, a great deal of research has been directed at characterizing the genes and proteins of the glycolytic pathway in an attempt to gain a better understanding of the pathway and to develop effective inhibitors to destroy the parasite. (Health et al.,2010, Huthmacher et al.,2010).

3.10.1 TRANSCRIPTION ASSOCIATED PROTEIN EXTRACTION ANALYSIS

The identification of transcription factors (simple motifs) and binding sites (structured motif) was effected by running three sets of experiments using different gene sets of *P.*

falciparum. The first experiment used the set of genes in the work of Flueck *et al.* (2010) which experimentally extracted regulatory elements for P.*falciparum,* that is, 1000 base pairs upstream of gene start codons. The second experiment used the set of genes used by Yuda *et al*. (2009) which identified transcription factors in the mosquito-invasive stage of malaria parasite. The two experiments aimed at inferring the ability and sensitivity of STGEMS in predicting correctly, the motifs already extracted by some known biological experiments. The third experiment targeted predicting new binding sites by using the 3D7 gene clones of *P. falciparum*, this contains about 3,000 genes from the intraerythrocytic developmental cycle.

3.10.2 GLYCOLYSIS METABOLIC PATHWAY ANALYSIS

The extraction of motifs in the glycolysis pathway was accomplished by running STGEMS on the twenty six genes known to be involved in the glycolysis pathway of P.falciparum harvested from .www.plasmodb.org.The genes involved in glycolysis metabolic pathways were extracted, by including all information of the annotation for each gene in *P. falciparum.* We applied STGEMS to scan their upstream regions for conserved sequence motifs involved in the regulation of gene. STGEM's output ranks the extracted motifs by comparing the similarity score with the p-value of the position weight matrix for each of the motifs, thus reporting only the optimal motifs. The glycolysis pathway genes used are PF10_0122, PF10_0155, PF11_0157, PF11_0208, PF11_0294, PF11_0338, PFL0780w, MAL13P1.40, PF13_0141, PF13_0144, PF13_0269, MAL13P1.324, PF14_0341, PF14_0378, PF14_0425, PF14_0598, PFB0210c, PFB0465c, PFC0275w, PFC0831w, PFD0660w, PFF1155w, PFF1300w, PFI0755c, PFI1105w and PFI1295c. STGEM program was executed with the DNA sequences of these genes as input with the aim of discovery relevant motifs. In order to validate the authenticity of the motifs identified by STGEMS, the popular GEMS algorithm was also applied to the same set of genes as a benchmark. This method of validation was motivated by the fact that the work of Young et al.,(2008) which developed GEMS algorithm, reported that the motifs discovered by GEMS had been biologically validated with follow up biological experiments, therefore, the biological relevance of the motif identified by STGEMS can be inferred based on its correlation with those identified by GEMS. The correlation coefficient was computed as the basis of comparison.

3.11 DISCUSSION ON THE IMPLEMENTATION OF STGEMS

The implementation of STGEMS algorithm consist of two main modules as shown in Appendix B. Module *st.c* implemented the space efficient construction of the suffix tree using suffix links. The suffix tree was constructed successively starting from the root node and moving down the depth of the tree inserting new leaves that are not already in existence. If a part of the sequence string already exists on the tree, it is not inserted again. The tree construction continues until all the strings in the input is evaluated and added to the tree. Suffix links points to the next location to inset a new leaf and makes traversal faster.

The improved efficiency in using the suffix tree in STGEMS is achieved as follows: By using a suffix tree to enumerate all the initial clusters (SEEDS), if any two SEEDS are identical, they are not repeated; it suffices to choose one of the SEEDS. Also, while descending the suffix tree to enumerate the SEED, partial evaluations are made of the current motif based on the strings that are in the enumerated SEED so far. These partial evaluations are shared across all the SEEDS below the current node in the tree, thus there is no need to visit all the nodes. This is in contrast to other techniques that do not use the suffix tree in which case, they have to evaluate every base in each SEED thus leading to additional cost in time spent in traversing and returning the repeated pattern.

To extract the structured motif which is made up of two boxes of simple motifs and gaps, the motifs are considered in lexicographical order by a depth-first visit of the motif tree *M*. Every time we stop extending a motif, that is, when we backtrack in *M*, it is either because we found a valid motif of the maximal length, or because the quorum is no longer satisfied and we start to consider the next one in lexicographical order. The two connected boxes are returned as the structured motif with the gap specified in the parameter.

The SEED which is the output of the tree obtained by traversing from the root node to a leaf node is a unique pattern. This is fed as the input to *stgems.c* program which takes this seed and then computes the gene enrichment based on the hypergeometric scoring function. The position weight matrix is generated for each candidate motif which shows the frequency of occurrence of each alphabet in a specific position in the motif sequence. The p-value of each motif is computed using the hypergeometric formula. Then the motif is ranked by sorting it in ascending order relative to the computed p-value. The hash table was introduced to achieve an optimized sorting and ordering of the motifs. The hash function maps each possible occurrence of the motif to a unique slot index on the hash table. Collision in the hash table was handled using the overflow method which involves creating a second table for collisions and placing the motif at the first empty location.

The incorporation of these data structures in the development of STGEMS contributed immensely to its improved efficiency in terms of speed.

The implementation of the similarity mechanism used in MOGAMOD was relatively straightforward. The similarity check mechanism is based on the dominance value of the nucleotides which is computed using the standard method of generating position weight matrix. The position frequency matrix is first computed, that is the frequency of occurrence of each of the characters in a specific position. Then the dominance value of the nucleotide is computed as a measure of the similarity score. The implementation was not enhanced in any way since the purpose of this part of the research is to establish a comparative benchmark for the evaluation of the improved similarity mechanism used in STGEMS.

The final outputs of STGEMS are optimal motifs represented using sequence logos. The performance of any new motif discovery tools is evaluated by comparing the tool with existing tools using the same data set on and the same system specification. STGEMS was compared with several popular motif discovery tools. The results of these comparisons are discussed in chapter four.

CHAPTER FOUR

RESULTS AND DISCUSSION

We applied the methodology described in chapter three above to mine simple and structured motifs such as Transcription factors and DNA binding sites in *P. falciparum* and examined the speed of execution of the algorithm. Our findings are contained in the sections below. In addition, we report the result of running GEMS and STGEMS on the glycolytic metabolic pathways genes of *P. falciparum.*

4.1 EMPIRICAL RUNTIME COMPARISON OF STGEMS

In chapter three, we demonstrated the asymptotic runtime of STGEMS, In addition to that, we will show the comparison of the empirical runtime of STGEMS with the five motif discovery algorithms used in this research work. The set of genes used were downloaded from PlasmoDB. (An online database of P.f*alciparum* genes maintained by National Center for Biotechnology Information (NCBI) http://www.ncbi.nlm.nih.gov). The running time of STGEMS compared with the five popular motif discovery tools is shown in the table below and the graph thereafter.

Table 4.1 Running Time of STGEMS compared with other Motif discovery tools

Size of Data In BP	Running Time in Seconds MEME	WEEDER	GEMS	RISOTTO	EXMOTIF	**STGEMS**
20,000	36	15	26	12	10	05
40,000	43	23	34	22	20	16
60,000	55	35	56	31	30	27
80,000	68	45	69	47	41	39

Four different sizes of genes were used in the analysis i.e. 20,000 , 40,000, 60,000 and 80,000bp, this variation in gene sizes is chosen to enable a classification of the performance of the algorithms as a function of input size. The empirical runtime of the different algorithms was obtained by including a time stamp in the execution of the algorithm so that its output displayed the execution time. From table 4.1 above, it is obvious that the empirical run time of all the algorithms tested increased as the size of input increased as expected. The run time of the MEME algorithm which is a statistical based motif discovery tool was higher than all other algorithms. This is followed by GEMS which is also based on a statistical model. The pattern-driven methods WEEDER, RISOTTO and EXMOTIF are much faster than the statistical based tools; this speed performance gain is attributed to the fact that they all used the suffix tree data structure, which is known to enhance searching speed. EXMOTIF performed better than RISOTTO and WEEDER because it incorporated the use of suffix links in its implementation of the suffix tree. Our computational technique, STGEMS outperformed all the five algorithms compared with in improved runtime. This is because the framework of STGEMS was built on an efficient implementation of the suffix tree using linked list and hash table data structures unlike the other algorithms that did not incorporate these combined features.

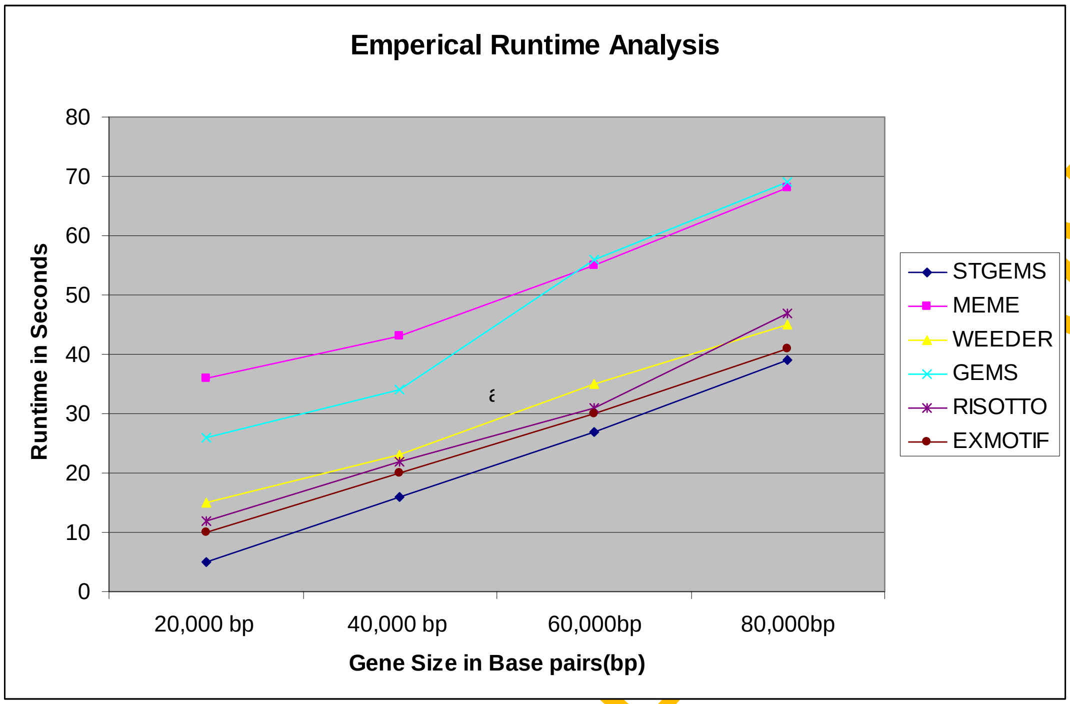


Figure 4.1: Empirical Run time for STGEMS and 5 motif discovery tools

In figure 4.1 above, which is the graph of the empirical runtime of STGEMS compared with those of MEME, WEEDER, GEMS, EXMOTIF and RISOTTO, It can be seen that MEME had the lowest performance among all the other tools compared since its running time over the set of the selected input was the highest, that is, the execution time of MEME in all instances was higher than the other tools. On the other hand, STGEMS had the lowest empirical run time which implied that it is a time efficient algorithm. The more efficient an algorithms is, the lower its empirical runtime.

4.2 PERFORMANCE OF STGEMS IN MINING BIOLOGICALLY VALIDATED MOTIFS

In addition to the empirical and asymptotic running time of STGEMS, it is also very effective in mining biologically motivated motifs in P.*falciparum*. To demonstrate this, two sets of sample genes in P.*falciparum,* which had been experimentally proven to co-regulate via structured motifs were used for testing. The implementation and testing of STGEMS was done in C programming language on Linux. The first experiment used the set of genes in the work of Flueck *et al.* (2010) which experimentally extracted binding sites for P.*falciparum*. i.e. 100 base pairs upstream of gene start codons as shown in table 4.2. Table 4.3 shows the result obtained running them on five popular motif discovery tools: RISOTTO, EXMOTIF, WEEDER, MEME and GEMS.

The second experiment used the set of genes used by Yuda *et al*. (2009) which identified transcription factors in the mosquito-invasive stage of malaria parasite shown in table 4.4. The resulting output using the five algorithms is depicted in table 4.5.

Flueck *et al.* (2010), showed experimentally, that the set of genes in table 4.2 co-regulate using the following motif: N(C/G/A)TGCA-4to5-(A/G/C)GTGC(A/G). 'N' indicates any of the four nucleotides A/C/G/T can occur at this position, while four to five gaps are between the two boxes.

Yuda *et al. (*2009) also show that experimentally, TAGCTA-100 to1500-TAGCCA and TAGCTA-100 to1500-TGGCTA are those structured motifs used in their co-regulation.

Table 4.2. Set of genes from Flueck *et al.*

Accession No	Description
PFF0645c	*Plasmodium falciparum* 3D7 , integral membrane protein, putative
PFI0265c	*Plasmodium falciparum* 3D7, high molecular weight rhoptry protein
PFE0075c	*Plasmodium falciparum* 3D7, rhoptry-associated protein 3
PFE0080c	*Plasmodium falciparum* 3D7, rhoptry-associated protein 2
PFC0120w	*Plasmodium falciparum* 3D7 , cytoadherence linked asexual protein 3.1
MAL7P1.208	*Plasmodium falciparum* 3D7, rhoptry-associated membrane antigen
PFI1730w	*Plasmodium falciparum* 3D7, cytoadherence linked asexual protein 9
PF14_0102	*Plasmodium falciparum* 3D7, rhoptry-associated protein 1
PFD0295c	*Plasmodium falciparum* 3D7 , apical sushi protein
MAL7P1.119	*Plasmodium falciparum* 3D7 , rhoptry-associated leucine zipper-like protein 1
PFI1445w	*Plasmodium falciparum* 3D7, high molecular weight rhoptry protein 2

Table 4.2 shows the accession number of the genes and their equivalent description. The accession numbers are unique and they uniquely identify each gene in all the gene databases, such as http://geneontology.org, http://plasmodb.org, http://genebank.org etc. The standard NCBI format for naming genes accession numbers is the short name of the organism as the first character and then a sequence generated number. For example in PFI1445w, the PF stands for *Plasmodium Falciparum.*

The gene accession numbers starting with MAL stands for Malaria, the old format for naming gene accession number uses the disease caused by the organism instead of the organism's short name.

Table 4.3. Output from running the algorithms on the DNA sequences in table 4.2

	IDENTIFIED BY					
Motifs	**RISOTTO**	**EXMOTIF**	**WEEDER**	**MEME**	**GEMS**	**STGEMS**
GGTGCG	YES	NO	NO	NO	NO	NO
CGTGCG	NO	NO	NO	NO	NO	NO
CTGCA	YES	NO	NO	NO	YES	YES
GTGCA	YES	YES	YES	YES	YES	YES
ATGCA	NO	YES	YES	YES	YES	YES
AGTGCG	NO	NO	NO	NO	YES	YES

Table 4.3 shows the output of running the six algorithms using a data set which has been biologically proven to contain consensus motifs or potential binding sites. The output of each algorithm was scanned for the occurrence of these consensus motifs shown in the first column of table 4.3. that is the consensus motifs. These consensuses have been shown by Flueck *et al.* (2010) to be valid binding sites in P.*falciparum* using biological experimental methods. The result of the analysis revealed that MEME,WEEDER and EXMOTIF are similar both in the number and type of motif discovered, For instance, out of the six motifs scanned for, only two were discovered by the three tools i.e. 'GTGCA', 'ATGCA' while 'GGTGCG', 'CGTGCG', 'CTGCA' and 'AGTGCG' were not found. RISOTTO exhibited a unique behaviour, the type and number of motifs discovered did not correspond to those discovered by the other tools. It found three motifs out of the six scanned for. GEMS and STGEMS were similar in the type and number of motifs found, they both discovered 'CTGCA', 'GTGCA', 'ATGCA' and 'AGTGCG'. This similarity between STGEMS and GEMS is because they both utilized the same mechanism in their methodology, that is, the hypergeometric motif enrichment search mechanism. The main advantage STGEMS has over GEMS is in the speed of execution and in its ability to mine both simple and structured motifs for the challenging genome of P.*falciparum.*

Table 4.4. Set of genes from the Mosquito invasive stage of malaria parasite.

Accession No	Description
PF08_0136b	*Plasmodium falciparum* 3D7 , von Willebrand factor A-domain related protein
PFC0905c	*Plasmodium falciparum* 3D7, oocyst capsule protein
PFL0550w	*Plasmodium falciparum* 3D7, HSP20-like chaperone
PFC0640w	*Plasmodium falciparum* 3D7,CSP and TRAP-related protein
PFD0425w	*Plasmodium falciparum* 3D7 , sporozoite invasion-associated protein 1, putative
PF08_0030	*Plasmodium falciparum* 3D7, conserved Plasmodium protein, unknown function
PFL2135c	*Plasmodium falciparum* 3D7, conserved Plasmodium protein, unknown function
MAL13P1.203	*Plasmodium falciparum* 3D7 , secreted ookinete protein, putative
PF10_0027	Plasmodium falciparum 3D7,conserved Plasmodium protein, unknown function
PFL2510w	*Plasmodium falciparum* 3D7, chitinase
PF13_0355	*Plasmodium falciparum* 3D7, secreted ookinete protein
PFD0435c	*Plasmodium falciparum* 3D7, conserved Plasmodium protein
PFE0360c	*Plasmodium falciparum* 3D7, conserved Plasmodium protein
PF14_0040	*Plasmodium falciparum* 3D7, secreted ookinete adhesive protein
PFF0975c	*Plasmodium falciparum* 3D7, conserved Plasmodium protein
PF10_0302	*Plasmodium falciparum* 3D7, 28 kDa ookinete surface protein
PF10_0303	*Plasmodium falciparum* 3D7, 25 kDa ookinete surface antigen precursor
PFC0420w	*Plasmodium falciparum* 3D7, calcium dependent protein kinase 3
PFI1145w	*Plasmodium falciparum* 3D7, perforin like protein 3

Table 4.5: Output from running the algorithms on the DNA sequences in table 4.4

	IDENTIFIED BY					
Motifs	**RISOTTO**	**EXMOTIF**	**WEEDER**	**MEME**	**GEMS**	**STGEMS**
TAGCTA	YES	NO	YES	NO	NO	NO
TGGCTA	NO	NO	NO	NO	NO	NO
TAGCCA	NO	NO	NO	NO	NO	NO

From table 4.3, it is can be observed that many of the experimentally extracted motifs were also mined by GEMS and by extension, our novel algorithm, STGEMS. This was not the same finding in table 4.5, as none of the two structured motifs was found by GEMS nor by STGEMS. Only RISOTTO and WEEDER discovered one motif each that is, 'TAGCTA' and 'TAGCTA' respectively. The other three tools, that is, MEME, GEMS and EXMOTIF could not also detect any of the motifs.

The observation recorded in table 4.5 gave the understanding that a number of fine tunings, which are not necessarily algorithmic, are needed to effectively mine the desired structured motifs in the set of table 4.4.

4.3 EVALUATION OF PREDICTIVE ACCURACY OF STGEMS

The predictive accuracy of STGEMS was evaluated by computing its correlation coefficient and comparing it with that of the other five algorithms used as benchmark in this research.

Table 4.6 below shows the prediction accuracy of STGEMS and five motif discovery tools. The dataset are averaged over four cross-validated test set produced from the set of biologically validated motifs. The result shows that STGEMS has a relatively high and stable correlation coefficient with varying sizes of input. For gene sizes of 20,000, 40,000, 60,000 and 80,000 respectively, the correlation coefficient remained as close to 1 as possible that is, 0.86, 0.87, 0.9 and 0.91 respectively. This shows the high accuracy of our computational technique. The correlation coefficient of GEMS algorithm is similar to that of STGEMS. This ability of STGEMS and GEMS to achieve a high predictive ability in predicting biologically relevant motif in the challenging sequence of P.*falciparum* is due to the hypergeometric mechanism employed. The predictive accuracy of MEME and WEEDER were similar, it can be categorized as an average performance while EXMOTIF had the worst performance, having a negative correlation coefficient for the 20,000bp gene size.

Table 4.6: Correlation Coefficient for STGEMS and the 5 Algorithms

Tools	Data sets			
	20,000BP	40,000BP	60,000BP	80,0000BP
STGEMS	0.86	0.87	0.9	0.91
MEME	0.3	0.4	0.45	0.5
WEEDER	0.4	0.5	0.45	0.53
GEMS	0.86	0.87	0.9	0.91
RISOTO	0.6	0.48	0.5	0.6
EXMOTIF	-0.3	0.24	0.37	0.41

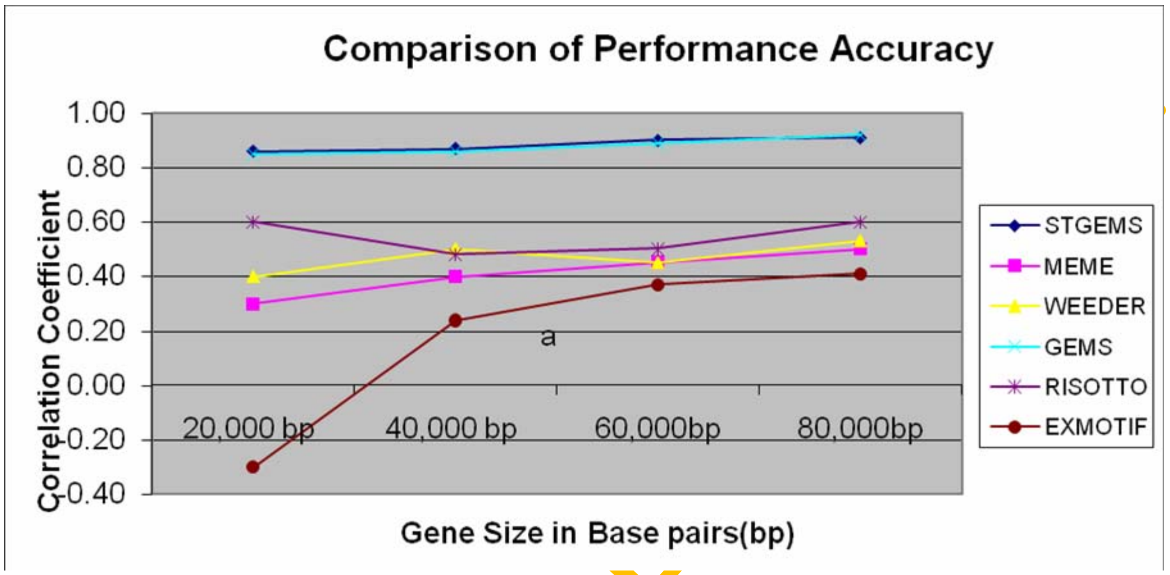
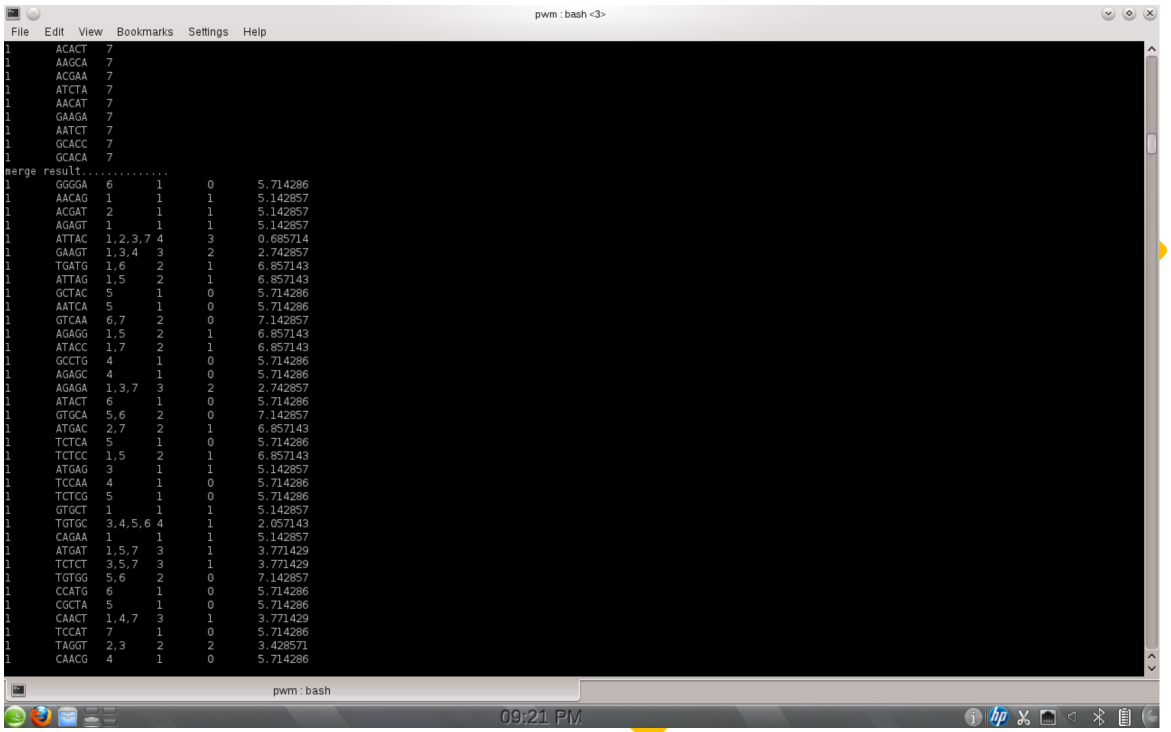


Figure 4.2: Performance Accuracy Comparison

4.4 STGEMS AND MINING NOVEL MOTIFS

STGEMS was run using the 3D7 genes from *P.falciparum* downloaded from PlasmoDB in an attempt to mine novel binding sites. An exhaustive listing of the genes and the extracted motif are found in Table 1 in appendix A. A snapshot of some of the results is shown in 4.3.



```
1       ACACT   7
1       AAGCA   7
1       ACGAA   7
1       ATCTA   7
1       AACAT   7
1       GAAGA   7
1       AATCT   7
1       GCACC   7
1       GCACA   7
merge result..............
1       GGGGA   6       1       0       5.714286
1       AACAG   1       1       1       5.142857
1       ACGAT   2       1       1       5.142857
1       AGAGT   1       1       1       5.142857
1       ATTAC   1,2,3,7 4       3       0.685714
1       GAAGT   1,3,4   3       2       2.742857
1       TGATG   1,6     2       1       6.857143
1       ATTAG   1,5     2       1       6.857143
1       GCTAC   5       1       0       5.714286
1       AATCA   5       1       0       5.714286
1       GTCAA   6,7     2       0       7.142857
1       AGAGG   1,5     2       1       6.857143
1       ATACC   1,7     2       1       6.857143
1       GCCTG   4       1       0       5.714286
1       AGAGC   4       1       0       5.714286
1       AGAGA   1,3,7   3       2       2.742857
1       ATACT   6       1       0       5.714286
1       GTGCA   5,6     2       0       7.142857
1       ATGAC   2,7     2       1       6.857143
1       TCTCA   5       1       0       5.714286
1       TCTCC   1,5     2       1       6.857143
1       ATGAG   3       1       1       5.142857
1       TCCAA   4       1       0       5.714286
1       TCTCG   5       1       0       5.714286
1       GTGCT   1       1       1       5.142857
1       TGTGC   3,4,5,6 4       1       2.057143
1       CAGAA   1       1       1       5.142857
1       ATGAT   1,5,7   3       1       3.771429
1       TCTCT   3,5,7   3       1       3.771429
1       TGTGG   5,6     2       0       7.142857
1       CCATG   6       1       0       5.714286
1       CGCTA   5       1       0       5.714286
1       CAACT   1,4,7   3       1       3.771429
1       TCCAT   7       1       0       5.714286
1       TAGGT   2,3     2       2       3.428571
1       CAACG   4       1       0       5.714286
```

Figure.4.3: Output of one of the modules of STGEMS

This is a snapshot of running one of the modules of STGEMS to extract motifs with their computed similarity score. The second column shows the candidate motif, the third column shows the number of the sequences in which the motif appears while the fourth column is the total number of occurrence of that motif. The fifth column shows the mismatches allowed while the last column is the similarity score for each candidate motif. These extracted motifs reported are the optimal motifs since the hypergeometric based similarity check mechanism employed, provided a ranking of all candidate motifs in such a way that only those with low similarity threshold relative to the position weight matrix generated are reported as optimal motifs. Based on that proposition, the following ten motifs were identified as optimal motifs by STGEMS: 'TCTAT' occurring in five sequences and having similarity value of 0.571429; 'TCTAA' occurred in three sequences with a value of 0.685714; 'TATCA' occurred in five sequences with a similarity value of 0.571429; 'TAACA' occurred in five sequences with a similarity value of 0.571429; 'GAGTA' occurred in three sequences and has a similarity value of 0.685714; 'TTATC' occurred in four sequences and have a value of 0.685714; 'TACAC' occurred in five sequences and has a similarity value of 0.571429; 'GATGA' occurred in four sequences with a similarity value of 0.685714; 'ATCAA' occurred in four sequences with a similarity value of 0.685714 and 'ACCTG' occurred in three sequences with a similarity value of 0.685714.

Table 4.7 The optimal novel motifs predicted by STGEMS from 3D7 P.falciparum genes.

Optimal motif	Similarity Score
TCTAT	0.571429
TCTAA	0.685714
TATCA	0.571429
TAACA	0.571429
GAGTA	0.685714
TTATC	0.685714
TACAC	0.571429
GATGA	0.685714
ATCAA	0.685714
ACCTG	0.685714

It can be observed from table 4.7 that row 1 to four are similar in the type of nucleotide and also the positional information, that is, they all have the' T', 'A' and 'C' nucleotide with 'A' and 'T' having a higher occurrence than 'C'(Reiterating the AT rich nature of P.*falciparum*) although they vary slightly. For instance,

Optimal motif	**Similarity**
TCTAT	0.571429
TCTAA	0.685714

Are very similar with a mismatch at the fifth position, thus can represent the same motif and

Optimal motif	**Similarity**
TATCA	0.571429
TAACA	0.571429

are also very similar with one mismatch at the third position and can represent the same motif. In chapter two, the importance of representing motifs graphically using sequence logo was explored. The primary purpose being that it shows at a glance, the relationship between motifs of interest in terms of their positional information. Figure 4.4 below shows the sequence logo representation of the four motifs above.

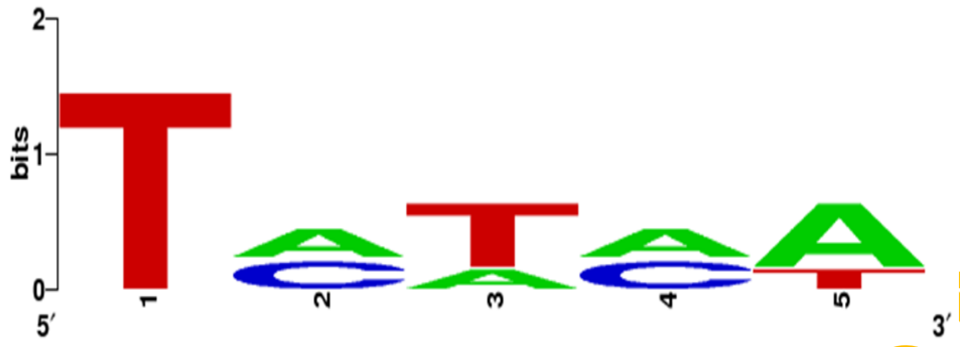


Figure 4.4: Sequence Logo of some of the optimal motifs predicted by STGEMS

The sequence logo representation of TCTAT, TCTAA, TATCA, TAACA is a multiple alignment of the sequences showing positional information. It is obvious from the diagram that ‘T’ and ‘A’ are dominant. In position 1, ‘T’ is present in all the four sequences and that explains why the logo of T is highest in that position. In position 2, ‘A’ and ‘C’ have equal distributions that is, two occurrences of ‘A’ and ‘C’ respectively. In position 3, ‘T’ occurred in three of the motifs while ‘A’ occurred in one. In position 4, ‘A’ and ‘C’ have equal occurrences while in position five, ‘A’ has three occurrences and ‘T’ has one occurrence.

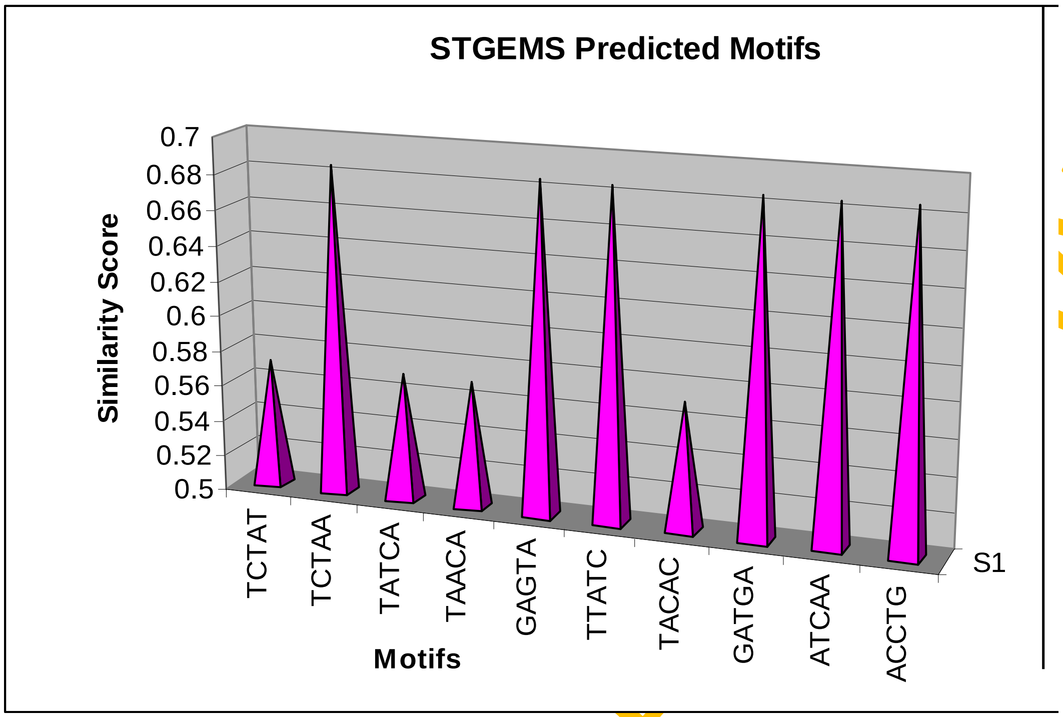


Figure 4.5: Similarity score of the optimal motifs predicted by STGEMS

From figure 4.5, it can be observed that the motifs with the lowest similarity score are TCTAT, TATCA ,TAACA and TACAC and they are the best motifs according to the gene enrichment method implored in the design framework of STGEMS. However, the biological validation of these predicted motifs is outside the scope of this project.
The structured motif was obtained by connecting two or more simple motifs and some spaces as indicated in the structured motif algorithm in chapter three. That is, e TCTAT_TAACA is an example of a structured motif.

4.5 APPLICATION OF STGEMS AND GEMS TO METABOLIC PATHWAY GENES

Having explained in chapter two, the importance of the metabolic pathway in understanding the biology of P.*falciparum* especially its role in drug target discovery, our computational inference technique, STGEMS was used on the glycolytic metabolic pathway genes. This glycolysis pathway shows the interactions involved in producing the necessary energy needed by the parasite to survive while in the two host it depends on, thus understanding the genes involved in this process will aid the design of effective drug target to terminate the organisms source of energy and invariably, lead to its eventual destruction. STGEMS was used to scan the regulatory genes in this pathway with a view to predicting conserved motifs which have biological significance.
In order to validate the relevance of the motifs identified by STGEMS, the popular GEMS algorithm was used in running the same set of genes as a benchmark. This method of validation was motivated by the fact that in the work of Young et al.,(2008) which developed GEMS algorithm, reported that the motifs discovered by GEMS had been biologically validated with follow up biological experiments, therefore, the biological relevance of the motif identified by STGEMS can be inferred based on its

correlation with those identified by GEMS. The motifs discovered by GEMS and STGEMS in the glycolytic pathways were the same. The computation of the correlation value of the comparison of the motifs discovered by the two algorithms relative to biologically validated motifs gave a correlation value of 0.98 which is very close to 1, the perfect prediction value. The sequence logo representation of some of the common motifs are shown in the sequence logo diagram in figure 4.6.

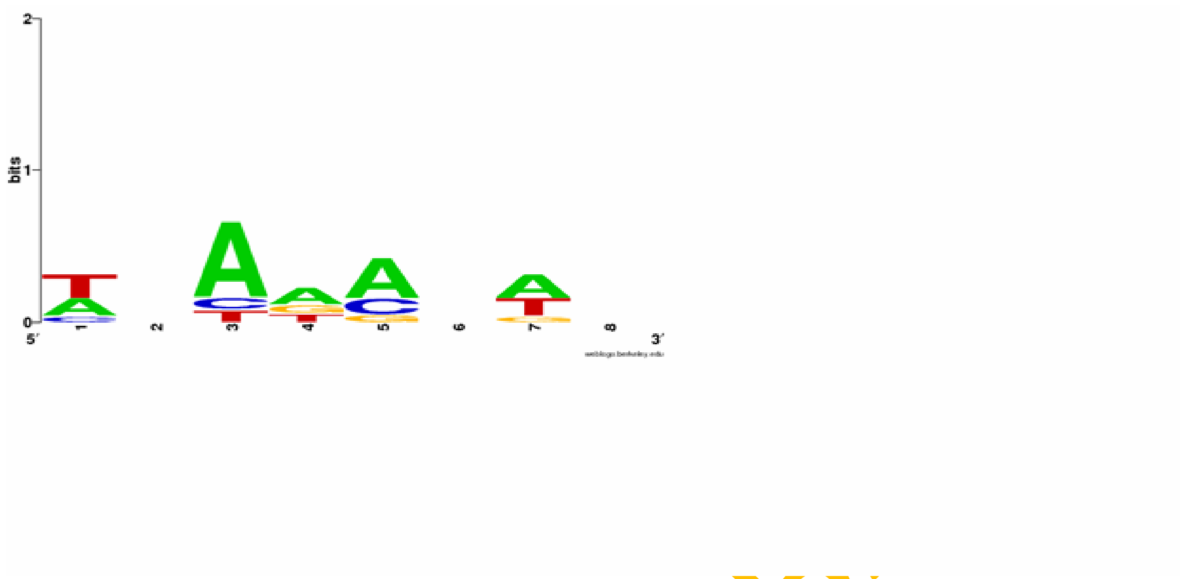


Figure 4.6: A sequence Logo of the common motifs extracted by GEMS and STGEMS.

4.6 COMPARISON OF THE SIMILARITY CHECK MECHANISM OF GEMS AND MOGAMOD

In chapter three, the implementation of the similarity check mechanism employed in the designs of GEMS and MOGAMOD algorithms were explored. We now present the result of the implementation by providing a comparative analysis of the motifs discovered using the two mechanisms. A graph of the value of the correlation coefficient of the motif discovered using the mechanism of GEMS and MOGAMOD compared with the experimentally identified motifs in Flueck et al.,(2010) is shown in figure 4.7 below.

Table 4.9: Motif identification with GEMS and MOGAMOD's similarity check mechanism

	IDENTIFIED BY	
Motifs	**MOGAMOD**	**GEMS**
GGTGCG	NO	NO
CGTGCG	NO	NO
CTGCA	NO	YES
GTGCA	NO	YES
ATGCA	NO	YES
AGTGCG	NO	YES

Computing correlation coefficient as a measure of prediction ability, we obtain the table 4.10

Table 4.10: Correlation Coefficient value of GEMS and MOGAMOD's similarity check

Motifs	**MOGAMOD**	**GEMS**
Sample 1	-0.01	0.6
Sample 2	-0.01	0.72
Sample 3	-0.1	0.78
Sample 4	-0.01	0.81
Sample 5	-0.1	0.84
Sample 6	-0.11	0.85

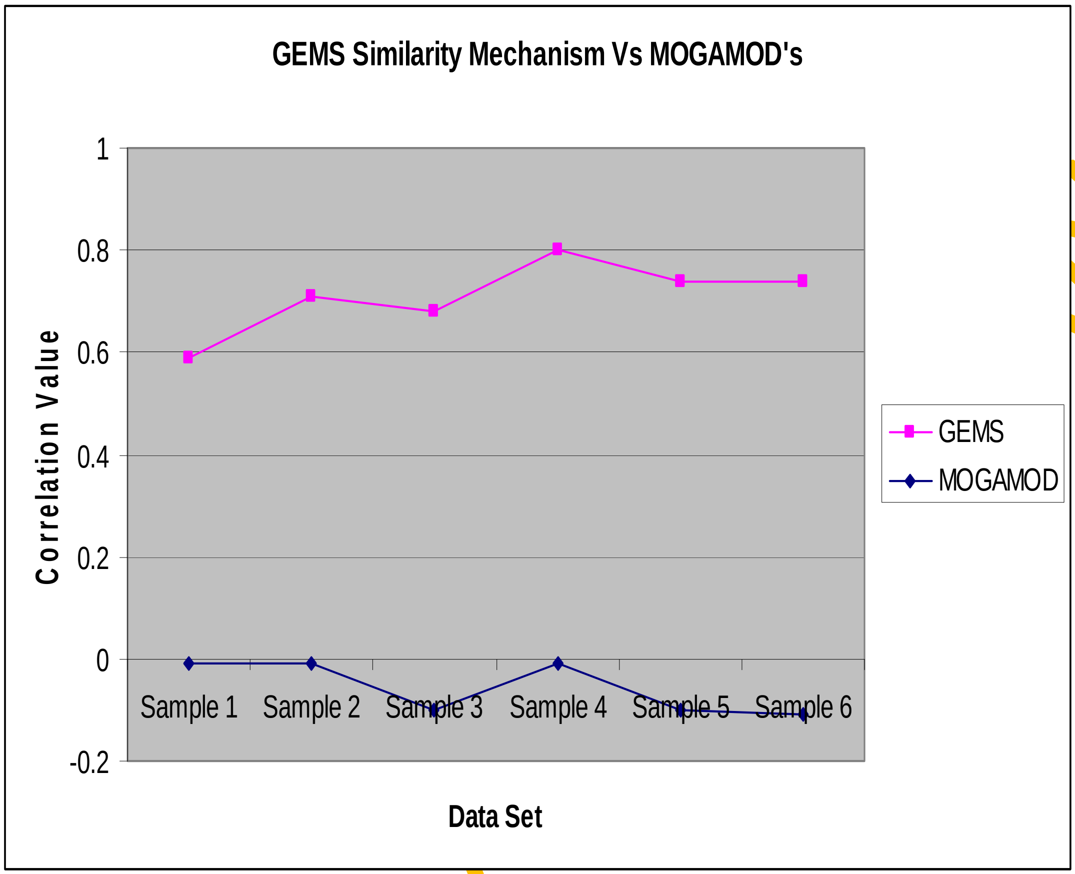


Figure 4.7: Similarity Check of GEMS and MOGAMOD compared

The correlation value of the MOGAMOD's mechanism was very low; all the values are negative which indicate a poor predictive accuracy or inverse prediction. On the other hand, the result obtained using GEMS's mechanism show a high correlation value of 0.6, 0.72, 0.78 and 0.87 respectively. We can therefore conclude that the implementation that incorporated the similarity mechanism of MOGAMOD performed very poorly in predicting relevant motifs as opposed to the implementation that incorporated GEMS similarity mechanism. This success as we explained in chapter one is attributed to the hypergeometric scoring function incorporated into the similarity mechanism of GEMS and made it possible to mine relevant biologically motivated motifs from the challenging genome of P.*falciparum*.

In spite of the reported remarkable performance of MOGAMOD, STGEMS outperformed MOGAMOD in terms of accuracy and runtime when tested with the same data set. Moreover, MOGAMOD could only identify motifs from other model organisms like yeast and bacteria but not from the malaria parasite, while STGEMS identified simple and structured motifs in these organisms.

CHAPTER FIVE

CONCLUSION AND RECOMMENDATIONS

5.1 CONCLUSION

Motif discovery is presented in this research as the process of identifying and extracting patterns believed to have biological importance and therefore necessary for understanding the complex biological mechanisms of living organisms. The availability of efficient motif discovery algorithms to mine these patterns is crucial to the acquisition of this vital knowledge. The consideration of the peculiarity of the data set is an important factor in developing efficient algorithms to extract these complex genomic patterns. In this regard, we proposed a combinatorial model using the suffix tree, a pattern-driven approach and gene enrichment analysis, a statistical approach which we tagged STGEMS. This combinatorial approach guaranteed the incorporation of the speed efficiency of pattern-driven method with the improved sensitivity of statistical based methods. STGEMS utilized the suffix tree and the hypergeometric scoring function as a similarity check mechanism in its gene enrichment analysis. This enabled it to identify simple and structured motifs from organisms with peculiar genomic structures such as the malaria parasite. The key role played by these transcription associated proteins which are simple motifs (transcription factors) and structured motifs (DNA binding site) identified by STGEMS in malaria research was highlighted.

We stressed that the importance of a good choice of data structures in the design paradigm of motif discovery tools cannot be overemphasized. Considering the exponential growth of genomic data availability, the accessibility of high performance algorithms for efficient analysis of these sequence data is paramount. This research presents the criteria for choosing data structures for motif discovery algorithms and the justification for the data structures used in STGEMS.

The linear run time algorithm developed proved effective in mining transcription factors and binding sites from the challenging genome of the malaria parasite, *P.falciparum.* This was demonstrated by comparing motifs identified by STGEMS with those extracted from biological experiments. The high correlation obtained by this comparison reveals the high sensitivity of STGEMS.

We proved the speed performance of STGEMS by comparing its empirical runtime with that of five popular motif discovery tools. The average runtime for the three categories of data set used (small, medium and large) revealed that STGEMS ensures timely efficient performance over the popular algorithms we compared with. That is three simple motifs tools :WEEDER, MEME, and GEMS and two structured motifs tools: RISSOTO and EXMOTIF.

The research demonstrated the accuracy of STGEMS in extracting biologically motivated structured motifs in the challenging sequence of *P. falciparum* which existing algorithms could not. DNA binding sites in *P. falciparum,* a eukaryotic organism exist as structured motifs and necessary for identifying the viable drug targets to eliminate the drug resistant strains of the parasite.

The research employed STGEMS in scanning the glycolytic metabolic pathway genes of P.falciparum for relevant motifs and it identified biologically motivated motifs in the pathway. Thus providing a better understanding of the functional genes responsible for the high adaptability of the malaria parasite, which makes it able to survive in two different hosts in spite of the adverse condition introduced by anti malaria drug action. In providing a deeper understanding of the various gene interactions involved in the adaptability, this forms the basis of discovering efficient drug therapies to destroy this deadly parasite

In the same vein, STGEMS and GEMS were tested with the glycolytic metabolic pathway genes and they discovered the same motifs. This result further confirms the high sensitivity of our computational inference technique, since it had a high correlation with the biologically validated motifs discovered by GEMS.

We developed a computational tool for mining both simple and structured motifs with an improved runtime demonstrated by computing its asymptotic runtime which is linear in the length of sequence. Our computational tool also has a high accuracy in mining biologically motivated motifs. This was revealed by its ability to identify similar motifs extracted using experimental methods. In addition, novel DNA binding sites, which are, viable drug targets were extracted on a large scale from the *P.falciparum* genome.

We also implemented and compared two popular similarity check mechanisms. The first was based on hypergeometric scoring function while the second was based on the dominance value of nucleotides occurring in the candidate motifs. The result of the comparison influenced our choice of the hypergeometric based similarity check mechanism. This consequently enabled STGEM to mine relevant optimal motifs from *P.falciparum* while the other implementation based on dominance value could not.

Our results showed that STGEMS is a valuable tool that can enable malaria researchers and other biologists to effectively produce anti-malarial drugs given that the tool identifies the relevant drug targets (binding sites and transcription factors). With this new tool, a great step can be taken in confronting the challenge to eradicate malaria in Nigeria and other countries to which the disease is endemic.

5.2 CONTRIBUTION TO KNOWLEDGE

Our contribution to knowledge is two-fold:

We implemented an efficient and effective algorithm for mining simple motifs on the suffix tree. This led to a significant speed up in the run time of the algorithm. We then extended this to mine structured motifs. The resulting algorithm runs in linear time. Following this, we empirically proved the high sensitivity of the resulting algorithm to mining motifs from sequences like we have in *P. falciparum* and compared the similarity check mechanism of GEMS against that used in another popular algorithm for extracting structured motifs, a multi-objective genetic algorithm, MOGAMOD.

The results obtained validated the high sensitivity of the similarity check mechanism employed in GEMS and also showed that a careful deployment of this mechanism improved the sensitivity level of the resulting algorithm, STGEMS. The end results gave us room to exhaustively mine structured motifs.

Secondly, we successfully identified motifs in the glycolytic metabolic pathway of *P. falciparum*

5.3 RECOMMENDATIONS/FUTURE PERSPECTIVE

This present work has given lead to some future studies. The need to formalize a number of fine tunings to exhaustively extract biologically motivated structured motifs was identified. Such fine tunings include the determination of biological motivated gaps between the boxes (simple motifs) of structured motifs and the size of the boxes. This biological validation will be done as a future work.

One of the gaps identified in our review of motif discovery algorithms is the need for a motif discovery tools which incorporate phylogenetic relationships in its framework and at the same time puts sequences with peculiar genomic structure in good perspective. This is crucial in revealing evolutionary insights among the different organisms. This will be examined in a future work.

The need to develop an integrated sequence analysis tool was observed. This tool will incorporate adequate methodology for gene prediction, motif discovery and sequence alignment in a holistic framework. We hope to take this up as a future work.

REFERENCES

Adebiyi, E. and Kaufmann, M.(2002). Extracting common motifs under the levenshtein measure: Theory and Experimentation. *WABI.*

Adebiyi, E. F.(2003) Pattern discovery in biology and strings sorting: theory and experimentation. PhD Thesis Tubingen, Shaker Publishing Company Inc.

Apostolico, A., Parida, L. & Rombo,S.E. (2008). Motif patterns in 2D, *Theoretical Computer* Science 390(1): 40–55

Apostolico A, Bock M E, Lonardi S, Xu X. (2001). Efficient detection of unusual words. J. Comput. Bio., 7(1/2): 71-94.

Barash, Y. et al (2003). Modeling dependencies in protein-DNA binding sites. *In Proceedings of RECOMB-03, pp. 28–37.*

Bailey, T. and Elkan, C. (2000). MEME: discovering and analyzing DNA and protein sequence motifs. *Nucleic Acids Research* Vol. 34, Web Server issue W369–W373.

Bischo, E. and Vaquero, C.(2010) In silico and biological survey of transcription-associated proteins implicated in the transcriptional machinery during the erthrocyctic development of Plasmodium falciparum, *BMC Genomics*, 11:34.

Bock, C., Paulsen, M,, Tierling, S., Mikeska, T. and Lengauer T. (2006) CpG island methylation in human lymphocytes is highly correlated with DNA sequence, repeats, and predicted DNA structure. *PLoS Genetics* 2: 243–252.

Bozdech, Z., and Ginsburg, H. (2005). Data mining of the transcriptome of Plasmodium falciparum: the pentose phosphate pathway and ancillary processes. *Malar. J.*, 4 (1), 17–29.

Bulashevska, S., Adebiyi, E., Brors, B., and Eils, R.(2007). New insights into the genetic regulations of Plasmodium falciparum obtained by Bayesian probabilistic modelling. *Gene Regulation and System Biology*, 1, 137-149.

Bussemaker, H., Li, H., Siggia, E. (2001) Regulatory element detection using correlation with expression. *Nat Genet* 27: 167–71.

Burset, M. and Guigo, R. (1996). Evaluation of gene structure prediction programs. *Genomics* 34(3), 353-67.

Carvalho, A., Freitas, A., Oliveira, A., Sagot, M. (2004) A parallel algorithm for the extraction of structured motifs. *19th ACM Symposium on Applied Computing*. pp. 147–153.

Carvalho, A., Freitas, A., Oliveira, A., Sagot, M. (2004). Efficient Extraction of Structured Motifs Using Box-links. *String Processing and Information Retrieval Conference*. pp. 267–278.

Cawley, S., Wirth, A., Speed, T.(2001). PHAT: A gene finding program for Plasmodium falciparum. *Mol Biochem Parasitol.* 2001;118:167–174.

Coello, C.A.C. and Pulido, G.T.(2001). A micro-genetic algorithm for multiobjective optimization. *In Proceedings of Evolutionary multi-criterion optimization, First international conference*, pp 120-128.

Corne, D.W., Knowles, J.D., Oates, M.J (2000). The Pareto envelope-based selection algorithm for multiobjective optimization. *In Proceedings of sixth international conference on parallel problem solving from Nature*, pp 18–20..

Crooks, G.E, Hon, G., Chandonia, J.M., Brenner, S.E. (2004). WebLogo: A sequence logo generator, *Genome Research*, 14:1188-1190.

Cui, L. and Miao, J. (2010) Chromatin-mediated epigenetic regulation in the malaria parasite Plasmodium falciparum. *Eukar Cell*, 10:1128.

Daniel I. Morariu, Radu G. CreŃulescu, Lucian N. VinŃan (2011). Using Suffix Tree Document Representation in Hierarchical Agglomerative Clustering. *Journal of World Academy of Science, Engineering and Technology Vol.59 pp 16-34.*

Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.(2002). A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans Evol Comput* 6(2):182–97.

Del, V., Ernst, P., Falkenhahn, M., Fladerer, C., Glatting, K., Suhai, S. and Hotz-Wagenblatt, A. (2007). ProtSweep, 2Dsweep and DomainSweep: protein analysis suite at DKFZ. *Nucleic Acids Res*. 35, W444–W450.

Deitsch, K. et al.(2007) Mechanisms of gene regulation in Plasmodium. *Am J. Trop. Med. Hyg*. 77(2), 201-8.

Dyer, M.D, Murali, T.M. and Sobral, B.W. (2007) Computational prediction of host-pathogen protein interactions *BMC Bioinformatics*; Vol. 23; 159-166.

Eden, E., Lipson, D., Yogev, S. and Yakhini, Z. (2007) Discovering motifs in ranked lists of DNA sequences. *PLoS Comput Biol* 3(3): 239-243.

Eskin, E. and Pevzner, P. (2002) Finding composite regulatory patterns in DNA sequences. *BMC Bioinformatics*, 18 Suppl 1:S354,63-70.

Fatumo, S., Plaisma, K., Mallm, J.P., Schramm, G., Adebiyi, E., Oswald, M., Eils, R. and Koenig, R.(2009). Estimating novel potential drug targets of Plasmodium falciparum by analysing the metabolic network of knock-out strains in silico. *ScienceDirect Infection, Genetics and Evolution,* 9(3), 351-358.

Fatumo, S., Kitiporn, P., Adebiyi, E., and Koenig, R.(2010). Comparing metabolic network models based on genomic and automatically inferred enzyme information from Plasmodium and its human host to define drug targets in-silico. *ScienceDirect Infection, Genetics and Evolution*, 10(4), 1-9.

Flueck, C., Bartfai, R., Niederwieser, I., Witmer, K., Alako, B., Moes, S., Bozdech, Z., Jenoe,P., Stunnenberg, H. and Voss, T. (2010) A major Role for the Plasmodium falciparum ApiAP2 Protein PfSIP2 in Chromosome End Biology. *PLoS Pathog* 6(2): e1000784.

Flum, Jörg; Grohe, Martin (2006). Parameterized Complexity Theory. Springer. p. 417. http://www.springer.com/east/home/generic/search/results?SGWID=5-40109-22-141358322-0. Retrieved 2010-03-05

Fonseca, C.M. and Fleming, P.J. (1993). Multiobjective Genetic Algorithms. *In IEEE colloquium on 'Genetic Algorithms for Control Systems Engineering'* (Digest No. 1993/130).

Gardner, M., Hall, N., Fung, E., White, O., Berriman, M., Hyman, R., Carlton, J.M., Pain, A. (2002). Genome sequence of the human malaria parasite Plasmodium falciparum. *Nature* 419 (6906): 498–511.

Ginsburg, H. (2006) Progress in in-silico functional genomics: the malaria Metabolic Pathways database. *Trend Parasitol*. 22, 238-240.

Hajela, P. lin, C. (1992) Genetic search strategies in multicriterion optimal design. *Struct Optimization*;4(2):99–107.

Hertz, F., Hartzell, G. and Stormo, G. (1990) Identification of consensus patterns in unaligned DNA sequences known to be functionally related. *Comput Appl Biosci* 6:81-92.

Hertz, F. and Stormo, G. (1999): Identifying DNA and protein patterns with statistically significant alignments of multiple sequences. *BMC Bioinformatics*, 15:563-577.

Health, A.P et al. (2010) Finding metabolic pathways using atom tracking. *BMC Bioinformatics*, 26 (12), 1548-1555.

Heinemeyer, T. et al., (1998). Databases on transcriptional regulation: TRANSFAC, TRRD, and COMPEL. *Nucl. Acids Res*., 26:364–370.

Holland J.H (1975). Adaptation in Natural and Artifcial Systems. MIT Press.

Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2007) Introduction to Automata Theory, Languages, and Computation, Addison Wesley, Boston/San Francisco/New York (page 368)

Horn, J., Nafpliotis, N., Goldberg, D. (1994) A niched Pareto genetic algorithm for multiobjective optimization. *In Proceedings of the first IEEE conference on evolutionary computation. IEEE world congress on computational intelligence*, 27–35.

Hu, J., Li, B., Kihara, D. (2005) Limitations and potentials of current motif discovery algorithms. *Nucleic Acids Res*, 33:4899-4913.

Huthmacher et al.(2010). Antimalarial drug target in Plasmpdium falciparum predicted by stage specific metabolic networks. *BMC Systems Biology*, 4:120, 1186-1704

Iyer, V.R., Horak, C.E., Scafe, C.S., Botstein, D., Snyder, M., Brown, P.O (2001) Genomic binding sites of the yeast cell-cycle transcription factors SBF and MBF. *Nature*, 409(6819):533-8.

Jiang, D., Pei, J., Zhang, A. (2003) DHC: A Density-based Hierachical Clustering Method for Time-series Gene Expression Data. *In Proceeding of BIBE2003: 3RD IEE International Symposium on Bioinformatics and Bioengineering*,10-12.

Kaya, M. (2009) MOGAMOD: Multi-Objective Genetic Algorithm for Motif Discovery, *Expert Systems with Applications*, 36 (2): 1039-1947.

Keich, U. and Pevzner, P. A. (2002). Finding motifs in the twilight zone *BMC Bioinformatics*, 18(10):1374-1381.

Kellis, M., Patterson, N., Endrizzi, M., Birren, B., and Lander, E.S. 2004. Sequencing and comparison of yeast species to identify genes and regulatory elements. *Nature* 423, 241–254.

Knowles, J.D. and Corne, D.W.(2000). Approximating the nondominated front using the Pareto archived evolution strategy. *Evol Comput*;8(2):149–72.

Konak, A., David, W., Coitb, A., Smith, E. (2006). Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety* 91 pp 992–1007.

Kurtz, S. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149-1171, 1999.

Lawrence, C.E. and Reilly, A.A(1990) : An expectation maximization algorithm for the identification characterization of common sites in unaligned biopolymer sequences. *Proteins*, 7:41-51.

Liu, X., Brutlag, D.L., Liu, J.S. (2001). BioProspector: discovering conserved DNA motifs in upstream regulatory regions of co-expressed genes. *Pac. Symp. Biocomput*. 2001;6:127–138.

Liu, F.M., Tsai, J.J., Chen, R.M., Chen, S.N. and Shih, S.H. (2004). FMGA: finding motifs by genetic algorithm. *Fourth IEEE Symposium on Bioinformatics and Bioengineering* , 459.

.Lu, H., Yen, G.G.(2003). Rank-density-based multiobjective genetic algorithm and benchmark test function study. *IEEE Trans Evol Comput* ;7(4):325–43.

MacIsaac, K., Wang, T., Gordon, B., Gifford, D., Stromo G, et al. (2006) An improved map of conserved regulatory sites for Saccharomyces cerevisiae.
BMC Bioinformatics 7.

Makolo, Angela, Ezekiel Adebiyi and Osofisan Adenike (2011). Mining Structured Motifs with Gene Enrichment Motif Searching on Suffix tree. Journal of Computer Science and its Applications 18(1) : 71-78.

Makolo, Angela, Ezekiel Adebiyi and Osofisan Adenike (2012). Comparative Analysis of Similarity Check Mechanism for Motif Extraction. IEEE African Journal of Computing & ICT. 5(1) : 23-40.

Marsan L, Sagot, M. (2000). Algorithm for extraction of structured motif using a suffix tree with an application to promoter and regulatory site consensus identification. *J Comput Biol*, 7:345-362.

Mendes, N.D., Casimiro, A.C., Santos, P.M., Correia, I.S, Oliveira, A.L., Freitas, A.T. (2006). MUSA: A Parameter Free Algorithm For the Identification of Biologically Significant Motifs. *Bioinformatics (Oxford Journals)* 22(24): 2996-3002.

Miller, L.H., Baruch, D.I., Marsh, K. and Doumbo, O.K. (2002). The pathogenic basis of malaria. *Nature* 415, 673–679.

Modan, K. D. and Ho-Kwok, D. (2007). A survey of DNA motif finding algorithms. *In Proceedings of the Fourth Annual MCBIOS Conference on Computational Frontiers in Biomedicine*. 4(6): 234-248.

Murata, T., Ishibuchi, H. (1995) MOGA: multi-objective genetic algorithms. *In Proceedings of the 1995 IEEE international conference on evolutionary computation*, pp 29-36.

Naresh K.N and Shrish V. (2011).Software Bug Classification using Suffix Tree Clustering (STC) Algorithm. *CiiT International Journal of Data Mining Knowledge Engineering, Vol. 1, N*o. 7, pages 28 – 37.

Ortet P, Bastien O (2010). "Where Does the Alignment Score Distribution Shape Come from?". *Evolutionary Bioinformatics* **6**: 159–187.

Oyelade, J., Ewejobi, I., Brors, B., Eils, R. and Adebiyi, E.(2010): Computational identification of signaling pathways in Plasmodium falciparum. *Elsevier Journal Infect Genet Evol*., 7(10), 991-996

Pavesi, G., Manke, T and Martin, V.(2001). An algorithm for finding sequence of unknown length . *Bioinformatics (Oxford Journals).* 17(2), pages S207-S214.

Pevzner P, Sze S. (2000): Combinatorial approaches to finding subtle signals in DNA sequences. *In Proceedings of the Eighth International Conference on Intelligent Systems on Molecular Biology*, San Diego, CA, 269-278.

Pisanti, N., Carvalho, A., Marsan, L. and Sagot, M.F.(2006). RISOTTO: Fast extraction of motifs with mismatches. *In seventh latin American theoretical information symposium.231-241.*

Pizzi, C., Rastas, P. & Ukkonen, E. (2011). Motif Discovery with Compact Approaches - Design and Applications, *IEEE/ACM Trans. Comput. Biology Bioinform.* 8(1): 69–79.

Planes, F.J and Beasley, J.E.(2009) Path finding approaches and metabolic pathways. *Discrete Applied Mathematics*, 157, 2244-2256.

Ponts, N. et al.(2010). Nucleosome occupancy at transcription start sites in the human malaria parasite: A hard-wired evolution of virulence? *Infect Genet Evol*, 10:1016.

Reid J.E. and Wernisch L. (2011). STEME: efficient EM to find motifs in large data sets. *Nucleic Acids* 10.1093, 1–10.

Roth, F.P., Hughes, J.D., Estep, P.W. and Church, G.M.(1998) Finding DNA regulatory motifs within unaligned noncoding sequences clustered by whole-genome mRNA quantitation. *Nature Biotechnology*, 16:939-945.

Sandelin, A., et al (2004). Jaspar: an open access database for eukaryotic transcription factor binding profiles. *Nucl. Acids Res*., 32:D91–D94

Sagot, *M.(1998).* Spelling approximate repeated or common Motifs*, PLoS Computational Biology*, 18:931-941

Schaffer, J.D.(1985). Multiple objective optimization with vector evaluated genetic algorithms. *In Proceedings of the international conference on genetic algorithm and their applications*, 6:93-98.

Schneider, T.D and Stephens RM. (1990). Sequence Logos: A New Way to Display Consensus Sequences. *Nucleic Acids Res*. 18:6097-6100

Schneider, A.G., Mercereau-Puijalon, O.(2005). A new Apicomplexa-specific protein kinase family: multiple members in Plasmodium falciparum, all with an export signature. *BMC Genomics* 6(1),30.

Sinha, S., Tompa, M.(2000). A statistical method for finding transcription factor binding site. In *Proceedings of the Eighth International Conference on Intelligent Systems on Molecular Biology*, San Diego, CA 2000, 344-354.

Sinha S, Blanchette M, Tompa M: PhyME: A probabilistic algorithm for finding motifs in sets of orthologous sequences**.** *BMC Bioinformatics* 2004 , **5:**170.

Srinivas, N. and Deb, K.(1995). Multiobjective optimization using nondominated sorting in genetic algorithms. *J Evol Comput* 1994;2(3):221–48.

Teklehaimanot, A., Singer, B., Spielman, A., Tozan, Y., & Schapira, A. (2005). Coming to grips with malaria in the new millennium. *Earthscan*. 23-27.

Tompa, M. (2001). Identifying functional elements by comparative DNA sequence analysis. *Genome Res*, 11:1143-1144.

Tompa, M., Li, N., Bailey, T., Church, G., De Moor, B., Eskin, E., Favorov, A., Frith, M., Fu, Y., Kent, W., Makeev, V., Mironov, A., Noble, W., Pavesi, G., Pesole, G., Regnier, M., Simonis, N., Sinha, S., Thijs, G., Van Helden, J., Vandenbogaert, M., Weng, Z., Workman, C., Ye, C. and Zhu, Z. (2005). Assessing computational tools for the discovery of transcription factor binding sites. *Nat Biotechnol* , 23:137-144.

Tomovic, A. and Oakeley, E. J. (2005), Position dependencies in transcription factor binding sites. *Bioinformatics* 23(8):933-941

Tompa, M.(1999) An Exact Method for Finding Short Motifs in Sequences with Application to the Ribosome Binding Site Problem *7th Intl. Conf. Intelligent Systems for Molecular Biology*, 23:33-41.

Tuteja, R. (2007). Malaria - an overview. *FEBS Journal*, 274, 4670-9.

Vanet, A., Marsan, L., Labigne, A., Sagot, M. (2000). Inferring regulatory elements from a whole genome. An analysis of Helicobacter pylori σ^{80} family of promoter signals. *J.Mol.Biol*, 297:335.

Van, H. , Andre, B., Collado-Vides, J. (1998). Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies. *J Mol Biol* , 281:827-842.

Weiner, P.(1973). Linear pattern matching algorithm. *Proc. 14th IEEE Symposium on Switching and Automata Theory* 1-11. 190, 193

Wei, Z. and Jensen, S.(2006). GAME: detecting cis-regulatory elements using a genetic algorithm. *Bioinformatics*, 22:1577-1584.

Westenberger, S. et al.(2009). Genome-wide nucleosome mapping of Plasmodium falciparum reveals histone-rich coding and histone-poor intergenic regions and chromatin remodelling of core and subtelomeric genes. *BMC Genomics*, 10:610.

Yen, G., and Lu, H. (2003). Dynamic multiobjective evolutionary algorithm: adaptive cell-based rank and density estimation. *IEEE Trans Evol Comput* 7(3):253–74.

Young, J., Johnson, J., Benner, C., Yan, F., Chen, K., Roch, K., Zhou, Y., Winzeler, E. (2008): In silico discovery of transcription regulatory elements in *Plasmodium falciparum*. *BMC Genomics* ,9:70

Yuda, M., Iwanaga, S., Shigenubu, S., Mair, G., Janse, C., Waters, A., Kato, T. and Kaneko, I.(2009) Identification of a transcription factor in the mosquito-invasive stage of malaria parasite. *Molecular Microbilogy*, 71, 1402-1414.

Zang, Y., and Zaki, M. (2006). EXMOTIF: Efficient structured motif extraction. Algorithms for Molecular Biology. *BMC Bioinformatics,* 1:21.

Zare-Mirakabad F, Davoodi, P, Ahrabian, H, Nowzari-Dalini, A Sadeghi, M, Goliaei.B (2009): Finding Motifs Based on Suffix Trie. Journal of Advanced Modelling Optimization Vol 23: page 20-31.

Zhou Y, Young JA, Santrosyan A, Chen K, Yan SF, Winzeler EA(2005): In silico gene function prediction using ontology-based pattern identification. Bioinformatics, 21(7):1237-1245.

Zitzler, E., Deb, K., Thiele, L.(2000). Comparison of multiobjective evolutionary algorithms: empirical results. *Evol Comput* ;8(2):173–95.

Zitzler, E., Thiele, L. (1999). Multiobjective evolutionary algorithms: a comparative case study and the strength Pareto approach. *IEEE Trans Evol Comput* . 3(4):257–71.

Zitzler, E., Laumanns, M. and Thiele, L. (2001). SPEA2: improving the strength Pareto evolutionary algorithm. *IEEE Trans Evol Comput* . 2(3):267–71.

APENDIX A

The predicted motifs

Result of STGEMS for novel motif discovered

Motif	Seq present	**No.of occu**	**mismatch**	**Similarity value**
GGGGA	1	1	0	5.714286
AACAG	1	1	1	5.142857
ACGAT	2	1	1	5.142857
AGAGT	1	1	1	5.142857
ATTAC	1,2,3,7	4	3	0.685714
GAAGT	1,3,4	3	2	2.742857
TGATG	1,6	2	1	6.857143
ATTAG	1,5	2	1	6.857143
GCTAC	5	1	0	5.714286
AATCA	5	1	0	5.714286
GTCAA	6,7	2	0	7.142857
AGAGG	1,5	2	1	6.857143
ATACC	1,7	2	1	6.857143
GCCTG	4	1	0	5.714286
AGAGC	4	1	0	5.714286
AGAGA	1,3,7	3	2	2.742857
ATACT	6	1	0	5.714286
GTGCA	5,6	2	0	7.142857
ATGAC	2,7	2	1	6.857143
TCTCA	5	1	0	5.714286
TCTCC	1,5	2	1	6.857143
ATGAG	3	1	1	5.142857
TCCAA	4	1	0	5.714286
TCTCG	5	1	0	5.714286
GTGCT	1	1	1	5.142857

TGTGC	3,4,5,6	4	1	2.057143
CAGAA	1	1	1	5.142857
ATGAT	1,5,7	3	1	3.771429
TCTCT	3,5,7	3	1	3.771429
TGTGG	5,6	2	0	7.142857
CCATG	6	1	0	5.714286
CGCTA	5	1	0	5.714286
CAACT	1,4,7	3	1	3.771429
TCCAT	7	1	0	5.714286
TAGGT	2,3	2	2	3.428571
CAACG	4	1	0	5.714286
CATTA	5	1	0	5.714286
TAGGG	6	1	0	5.714286
CGCTT	3	1	1	5.142857
CAACA	1,2,4,5	4	2	1.714286
CGTAC	1	1	1	5.142857
CATTG	2,4,5	3	1	3.771429
TAGGA	2,4,5	3	1	3.771429
TAGGC	6	1	0	5.714286
CGTAA	7	1	0	5.714286
CGACA	1	1	1	5.142857
CACTT	7	1	0	5.714286
CGACC	2	1	1	5.142857
TCGCT	3	1	1	5.142857
TGGGG	6	1	0	5.714286
TTTCC	2,3,7	3	2	2.742857
CACTG	3,4	2	1	6.857143
TTCAA	4	1	0	5.714286
TTCAC	2,4	2	1	6.857143
TGGGT	1	1	1	5.142857
CCTGA	4,6	2	0	7.142857
TTCAG	4	1	0	5.714286
CACTA	7	1	0	5.714286
CTATA	1,4	2	1	6.857143
CTATG	2,6	2	1	6.857143
CCTGT	1,2,3	3	3	0.685714
ACTTA	1,2	2	2	3.428571
ACTTC	5,7	2	0	7.142857
CTATT	1,5,6	3	1	3.771429
TTGCC	2	1	1	5.142857
ACTTT	1,3,4,6	4	2	1.714286
GTTCG	3	1	1	5.142857
GTTCC	1,4	2	1	6.857143
GTTCA	5,6	2	0	7.142857
TTGCT	2	1	1	5.142857
ACGTA	1,7	2	1	6.857143
CTTGA	6	1	0	5.714286
ACAAG	1	1	1	5.142857
TAATG	5	1	0	5.714286
TAATC	5,6	2	0	7.142857

GGTGT	3	1	1	5.142857
ACGTT	2,4,5	3	1	3.771429
CTTGT	1,6,7	3	1	3.771429
GACAA	1,2,4,5,7	5	2	1.142857
GACAC	6	1	0	5.714286
AACCT	2	1	1	5.142857
GGTGC	1	1	1	5.142857
ACCGA	4,5	2	0	7.142857
GGTGA	5	1	0	5.714286
AAAGG	5,6	2	0	7.142857
AGGCA	6	1	0	5.714286
CTGGA	1	1	1	5.142857
GATCT	7	1	0	5.714286
AACCG	5	1	0	5.714286
GACAT	1,7	2	1	6.857143
GTCTT	2	1	1	5.142857
GCTCA	1,5	2	1	6.857143
ACCGT	5	1	0	5.714286
GAGCA	4	1	0	5.714286
GTACC	7	1	0	5.714286
ATAAC	1,4,7	3	1	3.771429
GTACG	1,2	2	2	3.428571
ATGTC	6,7	2	0	7.142857
ATAAG	2,3,4,5,6	5	2	1.142857
ATGTG	1,2,5,7	4	2	1.714286
TCCTA	1,3,6,7	4	2	1.714286
TCCTG	6	1	0	5.714286
GTACT	1,3,4,7	4	2	1.714286
TCTAA	1,2,3	3	3	0.685714
TCCTT	2,4	2	1	6.857143
TCACA	5	1	0	5.714286
TCACC	4	1	0	5.714286
TCACG	2	1	1	5.142857
CGTTA	2,6	2	1	6.857143
TCTAT	1,2,3,4,7	5	3	0.571429
CGTTC	2	1	1	5.142857
TGCCC	2	1	1	5.142857
CGTTG	3,4	2	1	6.857143
TGAGA	3,7	2	1	6.857143
AGATT	1,6,7	3	1	3.771429
ATCGT	3	1	1	5.142857
TCACT	5,7	2	0	7.142857
TGAGG	7	1	0	5.714286
TATCA	1,2,3,4,5	5	3	0.571429
CGTTT	3	1	1	5.142857
TATCC	3,6	2	1	6.857143
TGCCT	4	1	0	5.714286
TCGAA	5,7	2	0	7.142857
TTCTC	1,7	2	1	6.857143
AGATA	1,6	2	1	6.857143

CACAC	1,7	2	1	6.857143
TGAGT	3	1	1	5.142857
CACAA	1	1	1	5.142857
CGAAA	1,7	2	1	6.857143
CGGTA	4	1	0	5.714286
TATCT	1,2,6,7	4	2	1.714286
CGAAC	7	1	0	5.714286
TACGT	1,2,7	3	2	2.742857
TCGAT	7	1	0	5.714286
CGAAG	1	1	1	5.142857
TTTAC	1,7	2	1	6.857143
TTTAG	7	1	0	5.714286
TACGG	3	1	1	5.142857
TTACA	1,3,7	3	2	2.742857
CGAAT	4,5	2	0	7.142857
TTACC	1,3,5	3	2	2.742857
CCCCA	2	1	1	5.142857
TTACG	3,7	2	1	6.857143
TTACT	1,3,4,6,7	5	2	1.142857
TAGTT	1,2,4,5,7	5	2	1.142857
ACATC	1,4,6,7	4	1	2.057143
TTGAG	3	1	1	5.142857
CATGA	3,4	2	1	6.857143
ACATG	3,4,7	3	1	3.771429
TAAAG	5	1	0	5.714286
CATGC	5,7	2	0	7.142857
TAGTA	3,4,6	3	1	3.771429
CATGG	5,6	2	0	7.142857
TTGAT	3,4,6	3	1	3.771429
CATGT	1,5	2	1	6.857143
AAGGT	3,6	2	1	6.857143
AGTAC	4,5	2	0	7.142857
GGCAC	6	1	0	5.714286
AGTAG	1	1	1	5.142857
CTCCC	1,5	2	1	6.857143
CTAGA	4	1	0	5.714286
AAGGA	3,5	2	1	6.857143
AGTAT	1,6,7	3	1	3.771429
GATAA	2,7	2	1	6.857143
GACTT	3,6	2	1	6.857143
GATAC	7	1	0	5.714286
ACTGA	3	1	1	5.142857
CTCCT	5	1	0	5.714286
GAACA	1,6	2	1	6.857143
GAACC	1	1	1	5.142857
ATATC	1	1	1	5.142857
GCAGA	1	1	1	5.142857
GCCCC	2	1	1	5.142857
AGGAA	3,4,7	3	1	3.771429
ACTGT	4	1	0	5.714286

AATGG	1,3,5	3	2	2.742857
GAACT	7	1	0	5.714286
AGCGT	6	1	0	5.714286
GAGAA	1,3	2	2	3.428571
AGGAT	2	1	1	5.142857
GTGTA	4	1	0	5.714286
GTAAC	5	1	0	5.714286
TCTTA	2,6,7	3	1	3.771429
GTAAG	7	1	0	5.714286
TCTTC	5,6	2	0	7.142857
TCTTG	6	1	0	5.714286
ACGGG	3	1	1	5.142857
GTAAT	1,2,4	3	2	2.742857
GTGTT	3,4,6	3	1	3.771429
GGTTT	2,3,5,7	4	2	1.714286
ACGGT	4	1	0	5.714286
AGAAT	1,6	2	1	6.857143
ATTGA	3,7	2	1	6.857143
ATTGG	3,7	2	1	6.857143
TCAAA	1,2	2	2	3.428571
TCGTA	5	1	0	5.714286
TCAAC	1,4	2	1	6.857143
GGTTA	1,6	2	1	6.857143
TCGTC	6	1	0	5.714286
TCGTG	2	1	1	5.142857
AGAAC	6	1	0	5.714286
TGCAC	1,3,4,5	4	2	1.714286
CGATA	2,5	2	1	6.857143
CACCT	4	1	0	5.714286
TCAAT	3	1	1	5.142857
TCGTT	2,3	2	2	3.428571
TGTCT	2	1	1	5.142857
ATGGA	1	1	1	5.142857
CAAGA	1	1	1	5.142857
ATGGG	1	1	1	5.142857
ATGGC	3	1	1	5.142857
CAAGC	6	1	0	5.714286
CACCA	1,5,7	3	1	3.771429
CGATT	7	1	0	5.714286
TGGCG	3	1	1	5.142857
ATGGT	3,5,6	3	1	3.771429
TCCGT	3	1	1	5.142857
TAGAT	1,4,6	3	1	3.771429
CCTCA	1,3	2	2	3.428571
TTAAC	1,3,6	3	2	2.742857
TTGTC	2,7	2	1	6.857143
TTAAG	2,4	2	1	6.857143
TTGTG	1,3,4	3	2	2.742857
TAGAG	1,3	2	2	3.428571
TAGAC	1,4	2	1	6.857143

TAACT	1,6	2	1	6.857143
CCCAT	1,2	2	2	3.428571
GTTGT	1,6	2	1	6.857143
CGTGT	2	1	1	5.142857
AGTTA	2,3	2	2	3.428571
AGTTC	1	1	1	5.142857
TAACG	5	1	0	5.714286
GTTGG	7	1	0	5.714286
TAACC	2,5,6	3	1	3.771429
TTCGC	3	1	1	5.142857
TAACA	1,2,3,4,7	5	3	0.571429
GATTA	1,3,6,7	4	2	1.714286
TACTT	1,2,6,7	4	2	1.714286
GATTC	4,7	2	0	7.142857
AGTTT	5,7	2	0	7.142857
TTCGT	5,6	2	0	7.142857
GATTG	4,6	2	0	7.142857
CTTCA	6	1	0	5.714286
CTTCC	6	1	0	5.714286
CTCAA	1	1	1	5.142857
GGACA	5	1	0	5.714286
CGGGT	3	1	1	5.142857
TACTC	1,4,6	3	1	3.771429
GATTT	1,3,7	3	2	2.742857
AGGTA	2	1	1	5.142857
TACTA	4,7	2	0	7.142857
AGGTG	6	1	0	5.714286
CTTCT	7	1	0	5.714286
AACGT	1,4	2	1	6.857143
GAGTA	1,2,3	3	3	0.685714
GAAAC	1,2,5	3	2	2.742857
AGGTT	1	1	1	5.142857
GTATC	4	1	0	5.714286
GAAAG	1,6,7	3	1	3.771429
GTATG	4,7	2	0	7.142857
GGGAA	6	1	0	5.714286
GGGAC	6	1	0	5.714286
ACAGA	7	1	0	5.714286
AACGG	4	1	0	5.714286
AACGC	5	1	0	5.714286
AACGA	2	1	1	5.142857
GGTAT	4	1	0	5.714286
ACAGT	1,4	2	1	6.857143
AAGTT	3	1	1	5.142857
GACGA	1	1	1	5.142857
GGTAC	3	1	1	5.142857
GGTAA	2,3,7	3	2	2.742857
AAGTG	1,3,5,6,7	5	2	1.142857
TCATG	1	1	1	5.142857
TGCTA	2	1	1	5.142857

TGCTC	1	1	1	5.142857
GCATT	1	1	1	5.142857
AGACT	3	1	1	5.142857
TCATT	5,6,7	3	0	3.885714
ATCCA	7	1	0	5.714286
TGTAA	1,4	2	1	6.857143
AGACA	1,4,7	3	1	3.771429
TGCTT	1,2,6	3	2	2.742857
AGACG	1	1	1	5.142857
TGACA	2,7	2	1	6.857143
ATCCT	1,3,6	3	2	2.742857
AATTG	1,5,6	3	1	3.771429
ATAGT	1	1	1	5.142857
GTGGA	5	1	0	5.714286
GTGGG	6	1	0	5.714286
TCTGG	1	1	1	5.142857
TTATC	1,2,3,7	4	3	0.685714
AGCTC	5	1	0	5.714286
TGGAA	1,3,5	3	2	2.742857
CCCTC	1,3	2	2	3.428571
TGGAC	5	1	0	5.714286
TTATG	6,7	2	0	7.142857
GTGGT	4	1	0	5.714286
TAGCT	5	1	0	5.714286
CCTAA	1,5,6	3	1	3.771429
TGGAT	1,3,7	3	2	2.742857
CCCTT	5	1	0	5.714286
TAGCA	7	1	0	5.714286
CCTAT	3,7	2	1	6.857143
CAATT	5,6,7	3	0	3.885714
CGAGA	3	1	1	5.142857
CATCA	4	1	0	5.714286
CCGAA	4	1	0	5.714286
CATCC	1	1	1	5.142857
TTTGG	2,4,6	3	1	3.771429
TACAG	4,7	2	0	7.142857
CTCTA	5,7	2	0	7.142857
CGAGT	2	1	1	5.142857
CTCTC	3	1	1	5.142857
TACAC	1,2,3,5,7	5	3	0.571429
CATCG	7	1	0	5.714286
CATCT	6	1	0	5.714286
CCGAT	5	1	0	5.714286
CTTAA	3	1	1	5.142857
CTCTT	3,4,5,6	4	1	2.057143
CTTAC	3,5	2	1	6.857143
GGAAA	1,3,4,6,7	5	2	1.142857
CTTAG	3,6	2	1	6.857143
GAATG	1,3,4	3	2	2.742857
TTGGA	3	1	1	5.142857

CTACA	7	1	0	5.714286
GGAAG	5	1	0	5.714286
GGGTG	1,5	2	1	6.857143
CTACG	5	1	0	5.714286
CTTAT	1	1	1	5.142857
GAATT	1,3,4,5	4	2	1.714286
ACTCA	1	1	1	5.142857
GGAAT	3	1	1	5.142857
GGGTT	3	1	1	5.142857
TTGGT	2,4,6	3	1	3.771429
ACTCC	5	1	0	5.714286
GTTTT	1,6	2	1	6.857143
AAGAT	1,2,4,6	4	2	1.714286
AGTGA	1	1	1	5.142857
AGTGC	3,4,6	3	1	3.771429
GGCGA	3	1	1	5.142857
ACTCT	3,4,5,6	4	1	2.057143
GTTTG	2	1	1	5.142857
AGTGG	4	1	0	5.714286
ACCAT	1	1	1	5.142857
GTTTA	2,4,5,7	4	1	2.057143
AAGAC	1,3,7	3	2	2.742857
GATGA	1,2,3,6	4	3	0.685714
AGTGT	5,7	2	0	7.142857
ACGCA	2,7	2	1	6.857143
AAACT	3,7	2	1	6.857143
GATGG	3	1	1	5.142857
AAACG	1,2,4	3	2	2.742857
GCAAA	3,4,5,7	4	1	2.057143
GATGT	3	1	1	5.142857
TGTTG	1,3,4,6,7	5	2	1.142857
AAACC	2	1	1	5.142857
ACGCT	5	1	0	5.714286
AGGGA	6	1	0	5.714286
ATTCC	3,4	2	1	6.857143
AGCAT	6	1	0	5.714286
AATAC	1,6	2	1	6.857143
ATCAA	1,2,3,5	4	3	0.685714
AACTT	1	1	1	5.142857
TGTTT	1,5,6,7	4	1	2.057143
ATTCG	5,6	2	0	7.142857
GAGGG	5	1	0	5.714286
AGCAG	1	1	1	5.142857
GCTTG	1	1	1	5.142857
AGCAC	4	1	0	5.714286
AACTC	3	1	1	5.142857
AACTA	1,5	2	1	6.857143
CACGT	1	1	1	5.142857
GAGGT	1	1	1	5.142857
TGGTG	3,6	2	1	6.857143

GCTTC	6	1	0	5.714286
GTAGT	7	1	0	5.714286
TGGTA	7	1	0	5.714286
ATGCC	4	1	0	5.714286
TGGTT	2,5,6,7	4	1	2.057143
CACGC	2,7	2	1	6.857143
TCCCA	1	1	1	5.142857
CCTTG	7	1	0	5.714286
ATGCT	1,2	2	2	3.428571
CCTTT	2,3	2	2	3.428571
TCCCT	1,3,5	3	2	2.742857
TCAGT	4	1	0	5.714286
CAAAG	5,6	2	0	7.142857
CAAAC	1,2	2	2	3.428571
CAGTA	1	1	1	5.142857
TATGG	1,3,7	3	2	2.742857
CCAAC	2	1	1	5.142857
CGTCA	6	1	0	5.714286
CCAAG	7	1	0	5.714286
CGCAA	7	1	0	5.714286
TAAGT	4,5	2	0	7.142857
TACCT	1,5	2	1	6.857143
CCAAT	4,5	2	0	7.142857
CCGTT	3,5	2	1	6.857143
CTTTA	1	1	1	5.142857
CTTTC	1,5	2	1	6.857143
TTCCC	1,3	2	2	3.428571
CATAG	3	1	1	5.142857
TAAGA	3,4,5,6	4	1	2.057143
TTAGA	1,3	2	2	3.428571
TTCCG	3	1	1	5.142857
CTTTG	4,5	2	0	7.142857
TTAGC	6,7	2	0	7.142857
TACCA	1	1	1	5.142857
GGATG	2,3	2	2	3.428571
TTAGG	2,4,5,6	4	1	2.057143
TAAGC	7	1	0	5.714286
TTCCT	2,4,5,7	4	1	2.057143
GGATT	1,7	2	1	6.857143
TTAGT	4,6	2	0	7.142857
ACCTA	5	1	0	5.714286
GTTAT	1,3,4,7	4	2	1.714286
ACCTG	1,2,3	3	3	0.685714
CTGTA	1,2,5	3	2	2.742857
CTAAC	5	1	0	5.714286
CTAAG	5,7	2	0	7.142857
CTGTG	4	1	0	5.714286
GTTAC	1,4,7	3	1	3.771429
ACTAA	2	1	1	5.142857
GTTAA	1,3,6,7	4	2	1.714286

ACCTT	4	1	0	5.714286
ACTAC	7	1	0	5.714286
CTAAT	2,6	2	1	6.857143
ACACC	5	1	0	5.714286
ACACG	1,7	2	1	6.857143
ACTAT	1,2,5	3	2	2.742857
GCGAG	3	1	1	5.142857
CTCGA	5	1	0	5.714286
ACACT	3,5,6,7	4	1	2.057143
AAGCA	1,4,6,7	4	1	2.057143
GCACT	4,6	2	0	7.142857
ACGAC	1,2	2	2	3.428571
GACCA	2	1	1	5.142857
ATCTA	2,7	2	1	6.857143
GAAGC	1,4	2	1	6.857143
ATCTG	1	1	1	5.142857
GCACC	1,4,5,7	4	1	2.057143
AATCT	4,6,7	3	0	3.885714
GCACA	3,7	2	1	6.857143

APENDIX B

Some of the predicted structured motifs by STGEMS

Structured Motif
GGGGA_ACGAT
AACAG_ATTAG
ACGAT_GCTAC
AGAGT_ATTAG
ATTAC_GCTAC

GAAGT_AATCT
TGATG_ATTAG
ATTAG_GCCTG
GCTAC_TCTCC
AATCA_ATACT
GTCAA_GAAGT
AGAGG_GCCTG
ATACC_GCCTG
GCCTG_ATACT
AGAGC_GAAGT
AGAGA_TCTCC
ATACT_GAAGT
GTGCA_GCGAG
ATGAC_GAAGT
TCTCA_ACACT
TCTCC_GCGAG
ATGAG_GCCTG
TCCAA_ATACT
TCTCG_ACACT
GTGCT_TCTCC
TGTGC_GCCTG

APPENDIX C

APENDIX C : SOURCE PROGRAM

APENDIX C : SOURCE PROGRAM

Program Extract Gene-Id

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>

#define LINE 80

void main(){
     FILE *fp1;
     char buffer[LINE], hold[LINE];
     int len, i, j;
     //strcpy(input, "TAATATTATTCTTTATTCGGTG");
     //strcpy(input2, "TAATAAAGCTCTTCGCTCGGTC");

     if((fp1 = fopen("Pf3D7genes-fasta.txt", "r")) == NULL)
          fprintf(stderr, "Error opening file\n");

     while(fgets(buffer, LINE-1, fp1) != NULL){
          len=strlen(buffer);
          if(buffer[0]=='>'){
               i=5; j=0;
               while(buffer[i]!=' '){
                    hold[j++]=buffer[i];
                    i++;
               }
               hold[j]='\0';
               printf("%s\n", hold);
          }
     }
     fclose(fp1);
}
```

```
/*
Author: Makolo, June 2011
This program implements suffix  tree part of STGEMS. It
modified the original program by  Stefan Kurtz (C) 1998
     * This ST construction use Linked list to acomplish
space afficiency as proposed by Kurtz
     * It construct the ST for any input submitted
*/

//\FILEINFO{mcc.c}

//\Ignore{
#include "bool.h"
#include "intbits.h"
#include "choice.h"
#include "mccdbg.pr"
#include "space.pr"
#include "showstr.pr"
#include "showibit.pr"
#define DEBUGDEFAULT(X)
DEBUG1(4,">%s\n",X);\DEBUGCODE(5,SHOWSTATE(state))
#define MEGABYTES(V)  ((double) (V)/((1 << 20) - 1))
#define VALIDINIT     0
#ifdef DEBUG
static void showvalues(void)
{
  SHOWVAL(SMALLINTS);
  SHOWVAL(LARGEINTS);
  SHOWVAL(MAXDISTANCE);
#if defined(MCCLG) || defined(MCCST)
  SHOWVAL(SMALLDEPTH);
#endif
  SHOWVAL(MAXTEXTLEN);
}
#endif
//}
/*
 This file contains code for the improved linked list
implementation,
 as described in \cite{KUR:1998,KUR:BAL:1999}. It can be
compiled with two options:
 \begin{itemize}\item for short strings of length \(\leq
2^{21}-1=2\) megabytes,we recommend the option
```

```
\texttt{MCCST}. This results in a representation which
requires
 \(2\) integers for each small node, and three integers for
each large node.
 See also the header file \texttt{mccdefST.h}. \item
 for long strings of length \(\leq 2^{27}-1=12\) megabytes,
we
 recommend the option \texttt{MCCLG}. This results in a
representation which requires
 \(2\) integers for each small node, and four integers for
each large node.
 See also the header file \texttt{mccdefLG.h}.
 \end{itemize}
*/
//\subsection{Space Management}
/*
 For a string of length \(n\) we initially allocate space
for
 \(\texttt{STARTFACTOR}\cdot\texttt{SMALLINTS}\cdot n\)
integers to store
 the branching nodes. This usually suffices for most cases.
In case we need
 more integers, we allocate space for
\(\texttt{ADDFACTOR}\cdot n\)
 (at least 16) extra branching nodes.
*/
#define STARTFACTOR 0.5
#define ADDFACTOR   0.05
#define MINEXTRA    16
/*
 These variables allow to check how many integers are
allocated for
 the branching nodes, without referring to \texttt{struct
MccState}.
 \texttt{firstallocated} refers to the last address, such
that at
 least \texttt{LARGEINTS} integers are available. So a
large node can
 be stored in the available amount of space.
*/

static Uint currentbranchtabsize = 0,
            *firstnotallocated;

/*
 \texttt{ALLOCFUNC} calls the functions which initially
allocate the space
 for the branching nodes and for the leafs of the suffix
tree. This function
 is exported.
*/
```

```
void ALLOCFUNC(Uint maxblocksize)
{
  if((currentbranchtabsize = (STARTFACTOR *
MULTBYSMALLINTS(maxblocksize+1))) < MINEXTRA)
  {
    currentbranchtabsize = MULTBYSMALLINTS(MINEXTRA);
  }
  DEBUG1(2,"#malloc(%u)\n",currentbranchtabsize);
  allocspace(LEAFBLOCK,sizeof(Uint),maxblocksize+2);

allocspace(ROOTCHILDRENBLOCK,sizeof(Uint),LARGESTCHARINDEX
+ 1);

allocspace(BRANCHBLOCK,sizeof(Uint),currentbranchtabsize);
}
/*
 Before a new node is stored, we check if there is enough
space available.
 If not, the space is enlarged by a small amount. Since
some global pointers
 directly refer into the table, these have to be adjusted
after reallocation.
*/
static void spaceforbranchtab(struct MccState *state)
{
  Uint extra, tmpheadnode, tmpchainstart;
  if(state->nextfreebranch >= firstnotallocated)
  {
    if((extra = (ADDFACTOR * (MULTBYSMALLINTS(state-
>textlen+1)))) < MINEXTRA)
    {
      extra = MULTBYSMALLINTS(MINEXTRA);
    }
    DEBUG1(2,"#all suffixes up to suffix %u have been
processed\n",state->nextfreeleafnum);
    DEBUG1(2,"#current space peak
%f\n",MEGABYTES(getspacepeak()));
    DEBUG1(2,"#to get %u extra space do ",extra);
    currentbranchtabsize += extra;
    DEBUG1(2,"realloc(%u)\n",currentbranchtabsize);

allocspace(BRANCHBLOCK,sizeof(Uint),currentbranchtabsize);
    tmpheadnode = NODEADDRESS(state->headnode);
    tmpchainstart = NODEADDRESS(state->chainstart);
    state->branchtab = (Uint *) reusespace(BRANCHBLOCK);
    state->undefchainstart = state->branchtab + LARGEINTS *
state->textlen;
    state->nextfreebranch = state->branchtab + state-
>nextfreebranchnum;
    state->headnode = state->branchtab + tmpheadnode;
    state->chainstart = state->branchtab + tmpchainstart;
    firstnotallocated = state->branchtab +
currentbranchtabsize - LARGEINTS;
  }
}

//\subsection{Initializing and Retrieving Headpositions,
Depth, and Suffixlinks}

/*
 We have three functions to initialize and retrieve head
positions, depth, and
 suffix links. The implementation depends on the bit
layout.
 \begin{enumerate}
 \item
 The function \emph{setdepthnum} stores the \emph{depth}
and the
 \emph{head position} of a new large node.
 \item
 The function \emph{setsuffixlink} stores the
\emph{suffixlink}
 of a new large node.
 \item
 The function \emph{getlargelink} retrieves the
\emph{suffixlink}
 of a large node, which is referenced by \emph{headpos}.
 \end{enumerate}
*/

#ifdef MCCST

static void setdepthheadpos(struct MccState *state,Uint
depth,Uint headpos)
{
  DEBUG2(4,"setdepth(%u)=%u\n",state-
>nextfreebranchnum,depth);
  DEBUGCODE(1,state->maxset = state->nextfreebranch + 2;
  if(ISSMALLDEPTH(depth))
  {
    *(state->nextfreebranch+1) |= SMALLDEPTHMARK;
  } else
  {
    *(state->nextfreebranch)
     = (*(state->nextfreebranch) & (MAXINDEX | LARGEBIT)) |
       ((depth << 11) & (7 << 29));
    *(state->nextfreebranch+1)
     = (*(state->nextfreebranch+1) & (MAXINDEX | NILBIT)) |
       ((depth << 14) & (127 << 25));
  }
  *(state->nextfreebranch+2) = (depth << 21) | headpos;
```

```
}

static void setsuffixlink(struct MccState *state,Uint
slink)
{
  *(state->setlink) = (*(state->setlink) & (255 << 24)) |
(slink | NILBIT);
  if(ISSMALLDEPTH(state->currentdepth))
  {
    *(state->nextfreebranch+1) |= (slink << 25);
    if(state->nextfreebranchnum & (~((1 << 7) - 1)))
    {
      *(state->nextfreebranch) |= ((slink << 17) & (255 <<
24));
      if(state->nextfreebranchnum & (~((1 << 15) - 1)))
      {
        state->leafbrother[state->nextfreeleafnum-1] |=
          ((slink << 9) & (255 << 24));
      }
    }
  }
}

static Uint getlargelink(struct MccState *state)
{
  SYMBOL secondchar;
  Uint succ, slink, headnodenum;
  DEBUGCODE(1,state->largelinks++);
  if(state->headnodedepth == 1)
  {
    return ROOT;             // link refers to the root
  }
  if(state->headnodedepth == 2)   // determine second
character of edge
  {
    if(state->headend == NULL)
    {
      secondchar = *(state->tailptr-1);
    } else
    {
      secondchar = *(state->tailptr - (state->headend -
state->headstart + 2));
    }
    return GETBRANCHINDEX(state->rootchildren[(Uint)
secondchar]);  // this leads to the suffix link node
  }
  if(ISSMALLDEPTH(state->headnodedepth))   // retrieve link
in constant time
  {
    slink = *(state->headnode+1) >> 25;
    headnodenum = NODEADDRESS(state->headnode);
    if(headnodenum & (~((1 << 7) - 1)))
    {
      slink |= ((*(state->headnode) & (255 << 24)) >> 17);
      if(headnodenum & (~((1 << 15) - 1)))
      {
        slink |= ((state->leafbrother[GETHEADPOS(state-
>headnode)]
                           & (255 << 24)) >> 9);
      }
    }
    return slink;
  }
  succ = state->onsuccpath;   // start at node on successor
path
  DEBUGCODE(1,state->largelinklinkwork++);
  while(!NILPTR(succ))        // linear retrieval of suffix
links
  {
    DEBUGCODE(1,state->largelinkwork++);
    if(ISLEAF(succ))
    {
      succ = LEAFBROTHERVAL(state-
>leafbrother[GETLEAFINDEX(succ)]);
    } else
    {
      succ = GETBROTHER(state->branchtab +
GETBRANCHINDEX(succ));
    }
    DEBUGCODE(1,state->largelinkwork++);
  }
  return succ & MAXINDEX;   // get only the index
}
#endif

#ifdef MCCLG

static void setdepthheadpos(struct MccState *state,Uint
depth,Uint headpos)
{
  if(ISSMALLDEPTH(depth))
  {
    *(state->nextfreebranch+2) = depth | SMALLDEPTHMARK;
  } else
  {
    *(state->nextfreebranch+2) = depth;
  }
  *(state->nextfreebranch+3) = headpos;
}

static void setsuffixlink(struct MccState *state,Uint
slink)
```

```
{
  Uint slinkhalf = slink >> 1;
  *(state->setlink) = (*(state->setlink) & EXTRAPATT) |
(slink | NILBIT);
  if(ISSMALLDEPTH(state->currentdepth))
  {
    *(state->nextfreebranch+2)
      |= ((slinkhalf << SMALLDEPTHBITS) & LOWERLINKPATT);
    if(state->nextfreebranchnum & (~LOWERLINKSIZE))
    {
      *(state->nextfreebranch+3) |= ((slinkhalf <<
SHIFTMIDDLE) & MIDDLELINKPATT);
      if(state->nextfreebranchnum & HIGHERSIZE)
      {
        state->leafbrother[state->nextfreeleafnum-1] |=
              ((slinkhalf << SHIFTHIGHER) & EXTRAPATT);
      }
    }
  }
}

static Uint getlargelink(struct MccState *state)
{
  SYMBOL secondchar;
  Uint succ, slinkhalf, headnodenum;
  DEBUGCODE(1,state->largelinks++);
  if(state->headnodedepth == 1)
  {
    return ROOT;       // link refers to root
  }
  if(state->headnodedepth == 2)  // determine second char
of egde
  {
    if(state->headend == NULL)
    {
      secondchar = *(state->tailptr-1);
    } else
    {
      secondchar = *(state->tailptr - (state->headend -
state->headstart + 2));
    }
    return state->rootchildren[(Uint) secondchar];
  }
  if(ISSMALLDEPTH(state->headnodedepth))  // retrieve link
in constant time
  {
    slinkhalf = (*(state->headnode+2) & LOWERLINKPATT) >>
SMALLDEPTHBITS;
    headnodenum = NODEADDRESS(state->headnode);
    if(headnodenum & (~LOWERLINKSIZE))
    {
      slinkhalf |= ((*(state->headnode+3) & MIDDLELINKPATT)
>> SHIFTMIDDLE);
      if(headnodenum & HIGHERSIZE)
      {
        slinkhalf
          |= ((state->leafbrother[GETHEADPOS(state-
>headnode)] & EXTRAPATT)
             >> SHIFTHIGHER);
      }
    }
    return slinkhalf << 1;
  }
  succ = state->onsuccpath;
  DEBUGCODE(1,state->largelinklinkwork++);
  while(!NILPTR(succ))   // linear retrieval of suffix link
  {
    DEBUGCODE(1,state->largelinkwork++);
    if(ISLEAF(succ))
    {
      succ = LEAFBROTHERVAL(state-
>leafbrother[GETLEAFINDEX(succ)]);
    } else
    {
      succ = GETBROTHER(state->branchtab +
GETBRANCHINDEX(succ));
    }
    DEBUGCODE(1,state->largelinkwork++);
  }
  return succ & MAXINDEX;   // get only the index
}
#endif

//\subsection{Insertion of Nodes}

/*
  The function \emph{insertleaf} inserts a leaf and a
corresponding leaf
  edge outgoing from the current \emph{headnode}.
  \emph{insertprev} refers to the node to the left of the
leaf to be inserted.
  If the leaf is the first child, then \emph{insertprev} is
\texttt{UNDEFINED}.
*/

static void insertleaf (struct MccState *state)
{
  Uint *ptr, newleaf = MAKELEAF(state->nextfreeleafnum);
  DEBUGDEFAULT("insertleaf");
  DEBUGCODE(1,state->insertleafcalls++);
  if(state->headnodedepth == 0)                // head is
the root
```

```
  {
    if(state->tailptr != state->sentinel)      // no \$-
edge initially
    {
      state->rootchildren[(Uint) *(state->tailptr)] =
newleaf;
      *(state->nextfreeleafptr) = VALIDINIT;
      DEBUG2(4,"%c-edge from root points to leaf
%u\n",*(state->tailptr),state->nextfreeleafnum);
    }
  } else
  {
    if (state->insertprev == UNDEFINED)  // newleaf = first
child
    {
      *(state->nextfreeleafptr) = GETCHILD(state-
>headnode);
      SETCHILD(state->headnode,newleaf);
    } else
    {
      if(ISLEAF(state->insertprev))    // previous node is
leaf
      {
        ptr = state->leafbrother + GETLEAFINDEX(state-
>insertprev);
        *(state->nextfreeleafptr) = LEAFBROTHERVAL(*ptr);
        SETLEAFBROTHER(ptr,newleaf);
      } else   // previous node is branching node
      {
        ptr = state->branchtab + GETBRANCHINDEX(state-
>insertprev);
        *(state->nextfreeleafptr) = GETBROTHER(ptr);
        SETBROTHER(ptr,newleaf);
      }
    }
  }
  RECALLSUCC(newleaf);     // recall node on successor path
of \emph{headnode}
  state->nextfreeleafnum++;
  state->nextfreeleafptr++;
}
/*
  The function \emph{insertbranch} inserts a branching node
and splits
  the appropriate edges, according to the canonical
location of the current
  head. \emph{insertprev} refers to the node to the left of
the branching
  node to be inserted. If the branching node is the first
child, then
```

```
  \emph{insertprev} is \texttt{UNDEFINED}. The edge to be
split ends
  in the node referred to by \emph{insertnode}.
*/

static void insertbranchnode(struct MccState *state)
{
  Uint *ptr, *insertnodeptr, *insertleafptr,
insertnodeptrbrother;
  DEBUGDEFAULT("insertbranchnode");
  spaceforbranchtab(state);
  if(state->headnodedepth == 0)      // head is the root
  {
    state->rootchildren[(Uint) *(state->headstart)] =
MAKEBRANCHADDR(state->nextfreebranchnum);
    *(state->nextfreebranch+1) = VALIDINIT;
    DEBUG2(4,"%c-edge from root points to branch node with
address %u\n",*(state->headstart),state-
>nextfreebranchnum);
  } else
  {
    if(state->insertprev == UNDEFINED)  // new branch =
first child
    {
      SETCHILD(state->headnode,MAKEBRANCHADDR(state-
>nextfreebranchnum));
    } else
    {
      if(ISLEAF(state->insertprev))  // new branch = right
brother of leaf
      {
        ptr = state->leafbrother + GETLEAFINDEX(state-
>insertprev);
        SETLEAFBROTHER(ptr,MAKEBRANCHADDR(state-
>nextfreebranchnum));
      } else                      // new branch = brother of
branching node
      {
        SETBROTHER(state->branchtab + GETBRANCHINDEX(state-
>insertprev),
                   MAKEBRANCHADDR(state-
>nextfreebranchnum));
      }
    }
  }
  if(ISLEAF(state->insertnode))   // split edge is leaf
edge
  {
    DEBUGCODE(1,state->splitleafedge++);
    insertleafptr = state->leafbrother +
GETLEAFINDEX(state->insertnode);
```

```
    if (state->tailptr == state->sentinel ||
        *(state->headend+1) < *(state->tailptr))
    {
      SETNEWCHILDBROTHER(MAKELARGE(state->insertnode),  //
first child=oldleaf
                         LEAFBROTHERVAL(*insertleafptr));
// inherit brother
      RECALLNEWLEAFADDRESS(state->nextfreeleafptr);
      SETLEAFBROTHER(insertleafptr,MAKELEAF(state-
>nextfreeleafnum)); // new leaf = right brother of old leaf
    } else
    {
      SETNEWCHILDBROTHER(MAKELARGELEAF(state-
>nextfreeleafnum),  // first child=new leaf
                         LEAFBROTHERVAL(*insertleafptr));
// inherit brother
      *(state->nextfreeleafptr) = state->insertnode;  //
old leaf = right brother of of new leaf
      RECALLLEAFADDRESS(insertleafptr);
    }
  } else  // split edge leads to branching node
  {
    DEBUGCODE(1,state->splitinternaledge++);
    insertnodeptr = state->branchtab +
GETBRANCHINDEX(state->insertnode);
    insertnodeptrbrother = GETBROTHER(insertnodeptr);
    if (state->tailptr == state->sentinel ||
        *(state->headend+1) < *(state->tailptr))
    {
      SETNEWCHILDBROTHER(MAKELARGE(state->insertnode), //
first child new branch
                         insertnodeptrbrother);        //
inherit right brother
      RECALLNEWLEAFADDRESS(state->nextfreeleafptr);
      SETBROTHER(insertnodeptr,MAKELEAF(state-
>nextfreeleafnum)); // new leaf = brother of old branch
    } else
    {
      SETNEWCHILDBROTHER(MAKELARGELEAF(state-
>nextfreeleafnum), // first child is new leaf
                         insertnodeptrbrother);        //
inherit brother
      *(state->nextfreeleafptr) = state->insertnode;   //
new branch is brother of new leaf
      RECALLBRANCHADDRESS(insertnodeptr);
    }
  }
  SETNILBIT;
  RECALLSUCC(MAKEBRANCHADDR(state->nextfreebranchnum)); //
node on succ. path
```

```
  state->currentdepth = state->headnodedepth + (Uint)
(state->headend-state->headstart+1);
  SETDEPTHHEADPOS(state->currentdepth,state-
>nextfreeleafnum);
  SETDEPTHSTAT(state->currentdepth);
  state->nextfreeleafnum++;
  state->nextfreeleafptr++;
  //DEBUGCODE(1,
  state->nodecount++;
}
//\subsection{Finding the Head-Locations}
/*
  The function \emph{rescan} finds the location of the
current head.
  In order to scan down the tree, it suffices to look at
the first
  character of each edge.
*/
static void rescan (struct MccState *state)
{
  Uint *nodeptr, *largeptr = NULL, distance = 0, node,
prevnode,
       nodedepth, edgelen, wlen, leafindex, headpos;
  SYMBOL headchar, edgechar;
  DEBUGDEFAULT("rescan");
  if(state->headnodedepth == 0)   // head is the root
  {
    headchar = *(state->headstart);  // headstart is
assumed to be not empty
    node = state->rootchildren[(Uint) headchar];
    //printf("follow %c-edge from root to
",headchar);SHOWINDEX(node);
    //printf("\n");
    if(ISLEAF(node))   // stop if successor is leaf
    {
      state->insertnode = node;
      return;
    }
    nodeptr = state->branchtab + GETBRANCHINDEX(node);
    GETONLYDEPTH(nodedepth,nodeptr);
    wlen = (Uint) (state->headend - state->headstart + 1);
    if(nodedepth > wlen)    // cannot reach the successor
node
    {
      state->insertnode = node;
      return;
    }
    state->headnode = nodeptr;        // go to successor
node
    state->headnodedepth = nodedepth;
    if(nodedepth == wlen)             // location has been
scanned
    {
      state->headend = NULL;
      return;
    }
    (state->headstart) += nodedepth;
  }
  while(True)   // \emph{headnode} is not the root
  {
    headchar = *(state->headstart);  // \emph{headstart} is
assumed to be nonempty
    prevnode = UNDEFINED;
    node = GETCHILD(state->headnode);
    while(True)             // traverse the list of
successors
    {
      if(ISLEAF(node))   // successor is leaf
      {
        leafindex = GETLEAFINDEX(node);
        edgechar = state->text[state->headnodedepth +
leafindex];
        if(edgechar == headchar)    // correct edge found
        {
          state->insertnode = node;
          state->insertprev = prevnode;
          return;
        }
        prevnode = node;
        node = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
      } else   // successor is branch node
      {
        nodeptr = state->branchtab + GETBRANCHINDEX(node);
        GETONLYHEADPOS(headpos,nodeptr);
        edgechar = state->text[state->headnodedepth +
headpos];
        if(edgechar == headchar) // correct edge found
        {
          break;
        }
        prevnode = node;
        node = GETBROTHER(nodeptr);
      }
    }
    GETDEPTHAFTERHEADPOS(nodedepth,nodeptr);     // get
info about succ node
    edgelen = nodedepth - state->headnodedepth;
    wlen = (Uint) (state->headend - state->headstart + 1);
    if(edgelen > wlen)     // cannot reach the succ node
    {
      state->insertnode = node;
      state->insertprev = prevnode;
      return;
    }
    state->headnode = nodeptr;    // go to the successor
node
    state->headnodedepth = nodedepth;
    if(edgelen == wlen)                    // location is
found
    {
      state->headend = NULL;
      return;
    }
    (state->headstart) += edgelen;
  }
}

/*
  The function \emph{taillcp} computes the length of the
longest common prefix
  of two strings. The first string is between pointers
\emph{start1} and
  \emph{end1}. The second string is the current tail, which
is between  the
  pointers \emph{tailptr} and \emph{sentinel}.
*/

static Uint taillcp(struct MccState *state,SYMBOL *start1,
SYMBOL *end1)
{
  SYMBOL *ptr1 = start1, *ptr2 = state->tailptr + 1;
  DEBUG0(4,">taillcp\n");
  DEBUG0(5,"[");
  DEBUGCODE(5,showstr(state->sentinel,ptr1,end1));
  DEBUG0(5,",");
  DEBUGCODE(5,showstr(state->sentinel,ptr2,state->sentinel-
1));
  while(ptr1 <= end1 && ptr2 < state->sentinel && *ptr1 ==
*ptr2)
  {
    ptr1++;
    ptr2++;
  }
  DEBUG1(5,"]=%u\n",(Uint) (ptr1-start1));
  return (Uint) (ptr1-start1);
}
/*
 The function \emph{scanprefix} scans a prefix of the
current tail
 down from a given node.
*/
```

```
static void scanprefix(struct MccState *state)
{
  Uint *nodeptr = NULL, *largeptr = NULL, leafindex,
nodedepth, edgelen, node,
       distance = 0, prevnode, prefixlen, headpos;
  SYMBOL *leftborder = (SYMBOL *) NULL, tailchar, edgechar
= 0;
  DEBUGDEFAULT("scanprefix");
  if(state->headnodedepth == 0)   // headnode is root
  {
    if(state->tailptr == state->sentinel)   // there is no
\$-edge
    {
      state->headend = NULL;
      return;
    }
    tailchar = *(state->tailptr);
    if((node = state->rootchildren[(Uint) tailchar]) ==
UNDEFINED)
    {
      state->headend = NULL;
      return;
    }
    if(ISLEAF(node)) // successor edge is leaf, compare
tail and leaf edge label
    {
      leftborder = state->text + GETLEAFINDEX(node);
      prefixlen = 1 + taillcp(state,leftborder+1,state-
>sentinel-1);
      (state->tailptr) += prefixlen;
      state->headstart = leftborder;
      state->headend = leftborder + (prefixlen-1);
      state->insertnode = node;
      return;
    }
    nodeptr = state->branchtab + GETBRANCHINDEX(node);
    GETBOTH(nodedepth,headpos,nodeptr);  // get info for
branch node
    leftborder = state->text + headpos;
    prefixlen = 1 + taillcp(state,leftborder+1,leftborder +
nodedepth - 1);
    (state->tailptr)+= prefixlen;
    if(nodedepth > prefixlen)   // cannot reach the
successor, fall out of tree
    {
      state->headstart = leftborder;
      state->headend = leftborder + (prefixlen-1);
      state->insertnode = node;
      return;
    }
    state->headnode = nodeptr;
    state->headnodedepth = nodedepth;
  }
  while(True)  // \emph{headnode} is not the root
  {
    prevnode = UNDEFINED;
    node = GETCHILD(state->headnode);
    if(state->tailptr == state->sentinel)  //  process \$-
edge
    {
      do // there is no \$-edge, so find last successor, of
which it becomes right brother
      {
        prevnode = node;
        if(ISLEAF(node))
        {
          node = LEAFBROTHERVAL(state-
>leafbrother[GETLEAFINDEX(node)]);
        } else
        {
          node = GETBROTHER(state->branchtab +
GETBRANCHINDEX(node));
        }
      } while(!NILPTR(node));
      state->insertnode = NILBIT;
      state->insertprev = prevnode;
      state->headend = NULL;
      return;
    }
    tailchar = *(state->tailptr);

    do // find successor edge with firstchar = tailchar
    {
      if(ISLEAF(node))   // successor is leaf
      {
        leafindex = GETLEAFINDEX(node);
        leftborder = state->text + (state->headnodedepth +
leafindex);
        if((edgechar = *leftborder) >= tailchar)   // edge
will not come later
        {
          break;
        }
        prevnode = node;
        node = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
      } else  // successor is branch node
      {
        nodeptr = state->branchtab + GETBRANCHINDEX(node);
        GETONLYHEADPOS(headpos,nodeptr);
        leftborder = state->text + (state->headnodedepth +
headpos);
        if((edgechar = *leftborder) >= tailchar)  // edge
will not come later
        {
          break;
        }
        prevnode = node;
        node = GETBROTHER(nodeptr);
      }
    } while(!NILPTR(node));
    if(NILPTR(node) || edgechar > tailchar)  // edge not
found
    {
      state->insertprev = prevnode;   // new edge will
become brother of this
      state->headend = NULL;
      return;
    }
    if(ISLEAF(node))  // correct edge is leaf edge, compare
its label with tail
    {
      prefixlen = 1 + taillcp(state,leftborder+1,state-
>sentinel-1);
      (state->tailptr) += prefixlen;
      state->headstart = leftborder;
      state->headend = leftborder + (prefixlen-1);
      state->insertnode = node;
      state->insertprev = prevnode;
      return;
    }
    GETDEPTHAFTERHEADPOS(nodedepth,nodeptr); // we already
know headpos
    edgelen = nodedepth - state->headnodedepth;
    prefixlen = 1 + taillcp(state,leftborder+1,leftborder +
edgelen - 1);
    (state->tailptr) += prefixlen;
    if(edgelen > prefixlen)  // cannot reach next node
    {
      state->headstart = leftborder;
      state->headend = leftborder + (prefixlen-1);
      state->insertnode = node;
      state->insertprev = prevnode;
      return;
    }
    state->headnode = nodeptr;
    state->headnodedepth = nodedepth;
  }
}

//\subsection{Completion and Initialization}
```

```
/*
  The function \emph{completelarge} is called whenever a
large node
  is inserted. It basically sets the appropriate distance
values of the small
  nodes of the current chain.
*/

static void completelarge(struct MccState *state)
{
  Uint distance, *backwards;
  if(state->smallnotcompleted > 0)
  {
    backwards = state->nextfreebranch;
    for(distance = 1; distance <= state->smallnotcompleted;
distance++)
    {
      backwards -= SMALLINTS;
      SETDISTANCE(backwards,distance);
    }
    state->smallnotcompleted = 0;
    state->chainstart = state->undefchainstart;
  }
  state->nextfreebranch += LARGEINTS;
  state->nextfreebranchnum += LARGEINTS;
  DEBUGCODE(1,state->largenode++);
}

/*
  The function \emph{linkrootchildren} constructs the
successor chain
  for the children of the root. This is done at the end of
the algorithm
  in one sweep over table \emph{rootchildren}.
*/

static void linkrootchildren(struct MccState *state)
{
  Uint *rcptr, *prevnodeptr, prev = UNDEFINED;
  for(rcptr = state->rootchildren; rcptr <= state-
>rootchildren + LARGESTCHARINDEX; rcptr++)
  {
    if(*rcptr != UNDEFINED)
    {
      if(prev == UNDEFINED)
      {
        SETCHILD(state->branchtab,MAKELARGE(*rcptr));
      } else
      {
        if(ISLEAF(prev))
        {
          state->leafbrother[GETLEAFINDEX(prev)] = *rcptr;
        } else
        {
          prevnodeptr = state->branchtab +
GETBRANCHINDEX(prev);
          SETBROTHER(prevnodeptr,*rcptr);
        }
      }
      prev = *rcptr;
    }
  }
  if(ISLEAF(prev))
  {
    state->leafbrother[GETLEAFINDEX(prev)] =
MAKELEAF(state->textlen);
  } else
  {
    prevnodeptr = state->branchtab + GETBRANCHINDEX(prev);
    SETBROTHER(prevnodeptr,MAKELEAF(state->textlen));
  }
  state->leafbrother[state->textlen] = NILBIT;
}

/*
  \newpage
  \emph{initMccState} allocates and initializes the data
structures for
  McCreight's Algorithm.
*/

static void initMccState(struct MccState *state,SYMBOL
*text,Uint textlen)
{
  Uint i, *ptr;

  state->text = state->tailptr = text;
  state->textlen = textlen;
  state->sentinel = text + textlen;
  state->branchtab = (Uint *) reusespace(BRANCHBLOCK);
  firstnotallocated = state->branchtab +
currentbranchtabsize - LARGEINTS;
  state->leafbrother = (Uint *) reusespace(LEAFBLOCK);
  state->headnode = state->nextfreebranch = state-
>branchtab;
  state->headend = NULL;
  state->headnodedepth = 0;
  state->rootchildren = (Uint *)
reusespace(ROOTCHILDRENBLOCK);
  for(ptr=state->rootchildren; ptr<=state-
>rootchildren+LARGESTCHARINDEX; ptr++)
```

```
  {
    *ptr = UNDEFINED;
  }
  for(i=0; i<LARGEINTS; i++)
  {
    state->branchtab[i] = 0;
  }
  state->nextfreebranch = state->branchtab;
  state->nextfreebranchnum = 0;
  SETDEPTHHEADPOS(0,0);
  SETNEWCHILDBROTHER(MAKELARGELEAF(0),0);
  SETBRANCHNODEOFFSET;
  state->rootchildren[(Uint) *text] = MAKELEAF(0);
  state->leafbrother[0] = VALIDINIT;
  DEBUG2(4,"%c-edge from root points to leaf
%u\n",*text,0);
  state->nextfreeleafnum = 1;
  state->nextfreeleafptr = state->leafbrother + 1;
  state->nextfreebranch = state->branchtab + LARGEINTS;
  state->nextfreebranchnum = LARGEINTS;
  state->insertnode = state->insertprev = UNDEFINED;
  state->smallnotcompleted = 0;
  state->chainstart = state->undefchainstart = state-
>branchtab + LARGEINTS * state->textlen;
  state->nodecount = 1;
  //state->maxset = state->branchtab + LARGEINTS - 1;
//\Ignore{

#ifndef SPACEOPT
  state->multitab[0] = 0;
  for(ptr = state->multitab+1; ptr <= state->multitab +
MAXDISTANCE; ptr++)
  {
    *ptr = SMALLINTS + *(rcptr-1);
  }
#endif
#ifdef DEBUG
  state->nodecount = 1;
  state->splitleafedge =
  state->splitinternaledge =
  state->largenode =
  state->smallnode =
  state->artificial =
  state->multiplications = 0;
  state->insertleafcalls = 1;
  state->maxset = state->branchtab + LARGEINTS - 1;
#ifdef MCCLG
  state->largelinks = state->largelinkwork = state-
>largelinklinkwork = 0;
  for(ptr = state->depthstat; ptr <=state-
>depthstat+MAXDEPTH; ptr++)
```

```
  {
    *ptr = 0;
  }
#endif
#endif

//}

}
//\subsection{Computing the Suffix Tree}

/*
  \emph{MCCFUNC} implements McCreight Algorithm compute the
suffix tree
  for a \texttt{text} of length \texttt{textlen}. For
explanations, see
  \cite{KUR:1998}. The number \((i)\) refers to the cases
of Section 6 in
  \cite{KUR:1998}.
*/

//@void MCCFUNC(struct MccState *state,SYMBOL *text,Uint
textlen)

//\Ignore{

void MCCFUNC(struct MccState *state,SYMBOL *text,Uint
textlen,void(*processhead)(struct MccState *,void *),void
*globalstruct)

//}

{

//\Ignore{

  if(textlen > MAXTEXTLEN)
  {
    fprintf(stderr,"Sorry, textlen = %u is larger than
maximal textlen = %u\n",
                   textlen,MAXTEXTLEN);
    exit(EXIT_FAILURE);
  }

//}

  DEBUGCODE(3,showvalues());
  initMccState(state,text,textlen);
  while(state->tailptr < state->sentinel || state-
>headnodedepth != 0 || state->headend != NULL)
  {
    if(state->headnodedepth == 0 && state->headend == NULL)
// case (1): headloc is root
    {
      (state->tailptr)++;
      scanprefix(state);
    } else
    {
      if(state->headend == NULL)  // case (2.1): headloc is
a node
      {
        FOLLOWSUFFIXLINK;
        scanprefix(state);
      } else               // case (2.2)
      {
       if(state->headnodedepth == 0) // case (2.2.1): at
the root do not use links
        {
          if(state->headstart == state->headend)  // rescan
not necessary
          {
            state->headend = NULL;
          } else
          {
            (state->headstart)++;
            rescan(state);
          }
        } else
        {
          FOLLOWSUFFIXLINK;    // case (2.2.2)
          rescan(state);
        }
        if(state->headend == NULL)  // case (2.2.3):
headloc is a node
        {
          SETSUFFIXLINK(state-
>nextfreebranch,NODEADDRESS(state->headnode));
          completelarge(state);
          scanprefix(state);
        } else
        {
#ifdef SPACEOPT
          if(state->smallnotcompleted == MAXDISTANCE)  //
insert artifical large node
          {
            DEBUGCODE(1,state->artificial++);
            DEBUG1(3,"#insert artifical large node
%u\n",state->nextfreebranchnum);
            SETSUFFIXLINK(state->nextfreebranch,state-
>nextfreebranchnum + LARGEINTS);
            completelarge(state);
          } else
          {
#endif
            if(state->chainstart == state->undefchainstart)
            {
              state->chainstart = state->nextfreebranch;
// begin a new chain
            }
            (state->smallnotcompleted)++;
            (state->nextfreebranch) += SMALLINTS;       //
case (2.2.4)
            (state->nextfreebranchnum) += SMALLINTS;
            DEBUGCODE(1,state->smallnode++);
#ifdef SPACEOPT
          }
#endif
        }
      }
    }

//\Ignore{

#ifdef APPLYSOMEFUNCTION

    if(globalstruct != NULL)
    {
      processhead(state,globalstruct);
    }

#endif

//}
    if(state->headend == NULL)
    {
      insertleaf(state);  // case (a)
    } else
    {
      insertbranchnode(state);  // case (b)
    }
    DEBUGCODE(5,SHWTABLE(state,False));
  }
  state->chainstart = state->undefchainstart;
  linkrootchildren(state);
//\Ignore{

  DEBUG1(2,"#integers for branchnodes %u\n",state-
>nextfreebranchnum);
  DEBUG4(2,"#small %u large %u textlen %u all %u ",
            state->smallnode,state->largenode,
            state->textlen,
            state->smallnode+state->largenode);
  DEBUG1(2,"ratio %f\n",
         (double) (state->smallnode+state-
>largenode)/state->nextfreeleafnum);
  DEBUG1(2,"#splitleafedge = %u\n",state->splitleafedge);
  DEBUG1(2,"#splitinternaledge = %u\n",state-
>splitinternaledge);
  DEBUG1(2,"#insertleafcalls = %u\n",state-
>insertleafcalls);
  DEBUG1(2,"#artificial = %u\n",state->artificial);
  DEBUG1(2,"#multiplications = %u\n",state-
>multiplications);
  DEBUGCODE(4,SHOWTABLE(state,True));
  DEBUGCODE(3,SHOWTREE(state));
#ifdef DEBUG
#ifdef SPACEOPT
  {
    Uint longchain = 0, chainsum = 0;
#if defined(MCCLG) || defined(MCCST)
    DEBUG3(2,"#largelinks %u largelinklinkwork %u
largelinkwork %u ",
              state->largelinks,state-
>largelinklinkwork,state->largelinkwork);
    DEBUG2(2,"#ratio1 %.4f ratio2 %.4f\n",
              (double) state->largelinkwork/state-
>largelinks,
              (double) state->largelinkwork/state-
>textlen);
#endif
    DEBUG1(2,"#longchain: %.7f\n",(double)
longchain/chainsum);
  }
#endif
#endif
  DEBUG2(1,"#%6u %6u\n",state->smallnode,state->largenode);
  DEBUGCODE(2,showspace());
  DEBUGCODE(1,CHECKTREE(state));
#ifdef DEBUG
#ifdef MCCLG
  {
    Uint i;
    for(i=0; i<MAXDEPTH; i++)
    {
      if(state->depthstat[i] > 0)
      {
        DEBUG2(2,"#Depth %u %u\n",i,state->depthstat[i]);
      }
    }
    DEBUG2(2,"#Depth>=%u %u\n",MAXDEPTH,state-
>depthstat[MAXDEPTH]);
  }
#endif
#endif
```

```
//}
}
//\Ignore{
#include "callfunc-gem.c"

//}

/*
Program callfunc-gem
This program implements Gene Enrichment using
Hypergeometric function
Author: Makolo June 2011.
*/
#ifndef BRJDIR
#include "my_header.h"
#include <time.h>
#include <stdlib.h>

int slidingwinlen=4;
//char substrxx[slidingwinlen];
//---
long N;
BOOL xx[FILESIZE]; // to further sort out exact maximal
repeats used as seeds

void bkread(struct MccState *state){
  Uint i;

  printf("\ntext len=%u and init char is %c\n", state-
>textlen, *state->text);
  N = state->textlen;
  for(i=0; i<state->textlen; i++){
    mineT[i]=*state->text++;
    xx[i]=false;
  }
  mineT[state->textlen]='\0';
  //printf("\n %s \n", mineT);
}
void displaytable(struct MccState *state)
{
  Uint *largeptr, *btptr, *succptr, *rcptr, i,
       succdepth, distance,
       nodeaddress, succ, depth, child, brother,
       headpos, suffixlink;
  Uint leafindex, edgelen;
  SYMBOL *leftpointer;

  printf(" Root:[");
  for(rcptr = state->rootchildren;
      rcptr <= state->rootchildren + LARGESTCHARINDEX;
      rcptr++)
  {
    if(*rcptr != UNDEFINED)
    {
      putchar('(');
      if(ISLEAF(*rcptr))
      {
        leftpointer = state->text + GETLEAFINDEX(*rcptr);
        showstr(state->sentinel,leftpointer,state-
>sentinel);
        printf(",Leaf %u)",GETLEAFINDEX(*rcptr));
      } else
      {
        succptr = state->branchtab +
GETBRANCHINDEX(*rcptr);
        GETBOTH(succdepth,headpos,succptr);
        leftpointer = state->text + headpos;
        showstr(state->sentinel,leftpointer,leftpointer +
succdepth - 1);
        printf(",%s %u)",ISLARGE(*succptr) ? "Large" :
"Small",
                                 GETBRANCHINDEX(*rcptr));
      }
      fflush(stdout);
    }
  }
  printf(",(~,Leaf %u)]\n",state->textlen);
  //remove state->nodecount
  btptr = state->branchtab + LARGEINTS; // skip the root
  for(i=1; i < state->nodecount; i++)
  {
    nodeaddress = NODEADDRESS(btptr);
    child = GETCHILD(btptr);
    brother = GETBROTHER(btptr);
    GETBOTH(depth,headpos,btptr);
    if(ISLARGE(*btptr))
    {
      printf(" L-Node %u\"",nodeaddress);
      suffixlink = GETSUFFIXLINKAC(btptr,depth);
      btptr += LARGEINTS;
    } else
    {
      printf(" S-Node %u\"",nodeaddress);
      suffixlink = nodeaddress + SMALLINTS;
      btptr += SMALLINTS;
    }
    showstr(state->sentinel,state->text + headpos,
                            state->text + headpos + depth -
1);

printf("\"(D=%u,SN=%u,SL=%u,C=",depth,headpos,suffixlink);
    SHOWINDEX(child);
```

```
        printf(",B=");
        SHOWINDEX(brother);
        printf(")[");
        fflush(stdout);
        succ = child;
        do
        {
          putchar('(');
          if(ISLEAF(succ))
          {
            leafindex = GETLEAFINDEX(succ);
            leftpointer = state->text + depth + leafindex;
            showstr(state->sentinel,leftpointer,state-
>sentinel);
            printf(",Leaf %u)",leafindex);
         indexcounter+=1;

            succ = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
          } else
          {
            succptr = state->branchtab + GETBRANCHINDEX(succ);
            GETBOTH(succdepth,headpos,succptr);
            leftpointer = state->text + depth + headpos;
            edgelen = succdepth - depth;
            showstr(state->sentinel,leftpointer,leftpointer +
edgelen - 1);
            printf(",%s %u)",ISLARGE(*succptr) ? "Large" :
"Small",
                                    GETBRANCHINDEX(succ));
            succ = GETBROTHER(succptr);
          }
        } while(!NILPTR(succ));
        printf("]\n");

        printf("indexcounter is %d\n", indexcounter);
        indexcounter=0;

        fflush(stdout);
      }
}
void listoccpos(Uint *btptr, struct MccState *state, Uint
patternlen,\
char mineT[DOUFILESIZE], BOOL allind)
{
  register Uint *succptr, succ, child, leafindex, i, k;
  char indexnum[SUPINDEXDIGITS];

  child = GETCHILD(btptr);
  succ =  child;
  do
```

```
    {
      if(ISLEAF(succ))
     {
        leafindex = GETLEAFINDEX(succ);
        if(allind){
          numhits+=1;
#ifdef DEBUG
          printf("%u,",leafindex);
#endif
        }else{
          k=0;
          for(i=leafindex+1; !isdigit(mineT[i]); i++);
          for(i=i; isdigit(mineT[i]);
indexnum[k++]=mineT[i++]);
          indexnum[k]='\0';
          indexnumi=atoi(indexnum);
          //printf(" %d,%s;",indexnumi,
superstr[indexnumi].string);
          break;
        }
        succ = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
      } else
        {
          succptr = state->branchtab + GETBRANCHINDEX(succ);
          listoccpos(succptr, state, patternlen, mineT,
allind);

          succ = GETBROTHER(succptr);
        }
    } while(!NILPTR(succ));
}
//getting a unique pattern equal to the pattern ends at a
leaf.
//all return zero except at ending at a leaf
BOOL finduniquepattern(char *pattern, struct MccState
*state,\
           char mineT[DOUFILESIZE], BOOL listocc, BOOL
allind)
{
  Uint *largeptr, *btptr, *rcptr, i, distance, nodeaddress,
child, brother,
    leafindex, edgelen, headpos, suffixlink;
  register Uint j, k, depth, succ, *succptr, succdepth,
leftpointer;
  BOOL seekpattern = false;
  short textptr = -1;
  Uint patternpos;
  register char auxchar;
  char indexnum[SUPINDEXDIGITS];
  register Uint patternlen, lencount=0;
  patternlen=strlen(pattern);
  auxchar=*pattern++;

  for(rcptr = state->rootchildren;
      rcptr <= state->rootchildren + LARGESTCHARINDEX;
      rcptr++)
  {
    if(*rcptr != UNDEFINED)
    {
      if(ISLEAF(*rcptr)) continue;
      else
      {
        succptr = state->branchtab +
GETBRANCHINDEX(*rcptr);
        GETBOTH(depth,headpos,succptr);
     //printf("headpos is %u\n", headpos);
        if(mineT[headpos]==auxchar){
       lencount+=1;

       //printf(",%s %u)\n",ISLARGE(*succptr) ? "Large" :
"Small",
       // GETBRANCHINDEX(*rcptr));
       // print show next char if *pattern is used
       //printf("mine[headpos] is %c, pattern is
%c\n",mineT[headpos], auxchar);
       for(j=headpos+1; j < headpos+depth &&
mineT[j]==(auxchar=*pattern++);\
          j++, lencount+=1){
         //printf("mine[j] is %c, pattern is
%c\n",mineT[j], auxchar);
       }// ends at a node (2)
       if(j==headpos+depth){
         seekpattern=true;
         //printf("headpos is %u\n", headpos);
         break;
       }// end in btw nodes (3)
       // depth > patternlen
       else if(patternlen==lencount){
         //printf("The index of the occurrence are\n");
         if(listocc) listoccpos(succptr, state, patternlen,
mineT, allind);
         //printf("\n");
         return 0;

       }else break;

     }
      }
    }
  }
  // finish checking the root children
  //printf("seekpattern is %d and depth is %u\n",
seekpattern, depth);
  // pattern is pointing presented to end of string so len
is 0
  //printf("len of pattern is %u\n", patternlen);

  btptr = succptr;
  //ends at a node just below the root
  if(seekpattern == true && depth==patternlen){
    //printf("The index of the occurrence are\n");
    if(listocc) listoccpos(succptr, state, patternlen,
mineT, allind);
    //printf("\n");

    return 0;
  }
  else if(seekpattern == true && depth < patternlen){
    for(; ;){
      child = GETCHILD(btptr);
      succ =  child;
      //printf("C=%u\n");
      //SHOWINDEX(child);
      //printf("\n");

      auxchar=*pattern++;
      do
     {
       if(ISLEAF(succ)){// ends at a leave (1)
         leafindex = GETLEAFINDEX(succ);
         leftpointer = leafindex + depth;
         //printf("deciding %c, %c, %d\n",
mineT[leftpointer], auxchar,\
               leftpointer);
         if(mineT[leftpointer]==auxchar){// comes to dead
end.
           lencount+=1;
           for(j=leftpointer+1; j<state->textlen &&
mineT[j]==*pattern++;\
              j++, lencount+=1);
           // ends at the end of leaf or in btw
           if(j==state->textlen || lencount==patternlen){
          //printf("The index of the occurrence are\n");
          //printf("%u,", leafindex);
          k=0;
          for(i=leafindex+1; !isdigit(mineT[i]); i++);
          for(i=i; isdigit(mineT[i]);
indexnum[k++]=mineT[i++]);
          indexnum[k]='\0';
          indexnumi=atoi(indexnum);
          //printf("%s,%s;",indexnum,
superstr[indexnumi].string);
          return 1;
           }
           return 0;
         }
         //printf("is Leaf %u\n",leafindex);
         succ = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
         if(NILPTR(succ)) return 0;

       }else
         {
           succptr = state->branchtab +
GETBRANCHINDEX(succ);
           GETBOTH(succdepth,headpos,succptr);
           edgelen = succdepth - depth;
           leftpointer = headpos + depth;
           //printf("at child level, headpos is %u\n",
headpos);
           //printf("deciding %c, %c, %d\n",
mineT[leftpointer], auxchar,\
               leftpointer);
       //getchar();

           if(mineT[leftpointer]==auxchar){
          lencount+=1;

          for(j=leftpointer+1; j<edgelen + leftpointer &&
mineT[j]==*pattern++;\
                j++, lencount+=1);
          //printf("here is %u, %u, %u\n",
j,edgelen+leftpointer,succdepth);
          // ends at a node (2)
          if(j==edgelen + leftpointer &&
succdepth==patternlen){
            //printf("The index of the occurrence are\n");
            if(listocc) listoccpos(succptr, state,
patternlen, mineT,allind);
            //printf("\n");
            //printf("The index of the occurrence are
%u\n", headpos);
            return 0;
          }// has not end
          else if(j==edgelen + leftpointer && succdepth <
patternlen){
            btptr = succptr, depth = succdepth;
            break;
          }
          // ends in btw node and node. (3)
          else if(lencount==patternlen){
            //printf("The index of the occurrence are\n");
            if(listocc) listoccpos(succptr, state,
patternlen, mineT,allind);
            //printf("\n");

            //printf("The index of the occurrence are
%u\n", headpos);
            return 0;
          }
          return 0;
           };// end if
           succ = GETBROTHER(succptr);
         }
     }while(!NILPTR(succ));
        if(NILPTR(succ)) return 0;

    };// infinite for loop end
  }
  return 0;
}

BOOL findpattern(char *pattern, struct MccState *state,\
           char mineT[DOUFILESIZE], BOOL listocc, BOOL
allind)
{
  Uint *largeptr, *btptr, *rcptr, i, distance, nodeaddress,
child, brother,
    leafindex, edgelen, headpos, suffixlink;
  register Uint j, k, depth, succ, *succptr, succdepth,
leftpointer;
  BOOL seekpattern = false;
  short textptr = -1;
  Uint patternpos;
  register char auxchar;
  char indexnum[SUPINDEXDIGITS];
  register Uint patternlen, lencount=0;

  patternlen=strlen(pattern);
  auxchar=*pattern++;

  for(rcptr = state->rootchildren;
      rcptr <= state->rootchildren + LARGESTCHARINDEX;
      rcptr++)
  {
    if(*rcptr != UNDEFINED)
    {
      if(ISLEAF(*rcptr)) continue;
      else
      {
        succptr = state->branchtab +
GETBRANCHINDEX(*rcptr);
```

```
          GETBOTH(depth,headpos,succptr);
       //printf("headpos is %u\n", headpos);
          if(mineT[headpos]==auxchar){
         lencount+=1;

         //printf(",%s %u)\n",ISLARGE(*succptr) ? "Large" :
"Small",
         // GETBRANCHINDEX(*rcptr));
         // print show next char if *pattern is used
         //printf("mine[headpos] is %c, pattern is
%c\n",mineT[headpos], auxchar);
         for(j=headpos+1; j < headpos+depth &&
mineT[j]==(auxchar=*pattern++);\
            j++, lencount+=1){
           //printf("mine[j] is %c, pattern is
%c\n",mineT[j], auxchar);
         }// ends at a node (2)
         if(j==headpos+depth){
           seekpattern=true;
           //printf("headpos is %u\n", headpos);
           break;
         }// end in btw nodes (3)
         // depth > patternlen
         else if(patternlen==lencount){
           //printf("The index of the occurrence are\n");
           if(listocc) listoccpos(succptr, state, patternlen,
mineT, allind);
           //printf("\n");
           return 1;

         }else break;

       }
        }
      }
    }
    // finish checking the root children
    //printf("seekpattern is %d and depth is %u\n",
seekpattern, depth);
    // pattern is pointing presented to end of string so len
is 0
    //printf("len of pattern is %u\n", patternlen);

    btptr = succptr;
    //ends at a node just below the root
    if(seekpattern == true && depth==patternlen){
      //printf("The index of the occurrence are\n");
      if(listocc) listoccpos(succptr, state, patternlen,
mineT, allind);
      //printf("\n");
```

```
        return 1;
      }
      else if(seekpattern == true && depth < patternlen){
        for(; ;){
          child = GETCHILD(btptr);
          succ =  child;
          //printf("C=%u\n");
          //SHOWINDEX(child);
          //printf("\n");

          auxchar=*pattern++;
          do
         {
           if(ISLEAF(succ)){// ends at a leave (1)
             leafindex = GETLEAFINDEX(succ);
             leftpointer = leafindex + depth;
             //printf("deciding %c, %c, %d\n",
mineT[leftpointer], auxchar,\
                   leftpointer);
             if(mineT[leftpointer]==auxchar){// comes to dead
end.
               lencount+=1;
               for(j=leftpointer+1; j<state->textlen &&
mineT[j]==*pattern++;\
                  j++, lencount+=1);
               // ends at the end of leaf or in btw
               if(j==state->textlen || lencount==patternlen){
              //printf("The index of the occurrence are\n");
              //printf("%u,", leafindex);
              k=0;
              for(i=leafindex+1; !isdigit(mineT[i]); i++);
              for(i=i; isdigit(mineT[i]);
indexnum[k++]=mineT[i++]);
              indexnum[k]='\0';
              indexnumi=atoi(indexnum);
              //printf("%s,%s;",indexnum,
superstr[indexnumi].string);
              return 1;
               }
               return 0;
             }
             //printf("is Leaf %u\n",leafindex);
             succ = LEAFBROTHERVAL(state-
>leafbrother[leafindex]);
             if(NILPTR(succ)) return 0;

           }else
             {
               succptr = state->branchtab +
GETBRANCHINDEX(succ);
               GETBOTH(succdepth,headpos,succptr);
               edgelen = succdepth - depth;
               leftpointer = headpos + depth;
               //printf("at child level, headpos is %u\n",
headpos);
               //printf("deciding %c, %c, %d\n",
mineT[leftpointer], auxchar,\
                   leftpointer);
           //getchar();

               if(mineT[leftpointer]==auxchar){
              lencount+=1;

              for(j=leftpointer+1; j<edgelen + leftpointer &&
mineT[j]==*pattern++;\
                j++, lencount+=1);
              //printf("here is %u, %u, %u\n",
j,edgelen+leftpointer,succdepth);
              // ends at a node (2)
              if(j==edgelen + leftpointer &&
succdepth==patternlen){
            //printf("The index of the occurrence are\n");
            if(listocc) listoccpos(succptr, state,
patternlen, mineT,allind);
            //printf("\n");
            //printf("The index of the occurrence are
%u\n", headpos);
            return 1;
              }// has not end
              else if(j==edgelen + leftpointer && succdepth <
patternlen){
            btptr = succptr, depth = succdepth;
            break;
              }
              // ends in btw node and node. (3)
              else if(lencount==patternlen){
            //printf("The index of the occurrence are\n");
            if(listocc) listoccpos(succptr, state,
patternlen, mineT,allind);
            //printf("\n");

//printf("The index of the occurrence are %u\n", headpos);
            return 1;
              }
              return 0;
               };// end if
               succ = GETBROTHER(succptr);
             }
         }while(!NILPTR(succ));
            if(NILPTR(succ)) return 0;

        };// infinite for loop end
```

```
  }
  return 0;
}


/*
void chopoutstr(int b, int e){
    int i, j=0;
    for(i=b; i<=e; i++)
     substrxx[j++]=mineT[i];
     substrxx[j]='\0';
}
*/
// added by Makolo&Adebiyi to return d SEED in form of
sliding window
void slidingwinfn(struct MccState *state){
int i, j, k=0;
char substr[slidingwinlen+1];

   for(i=0; i<=(state->textlen)-slidingwinlen; i++){
     k=0;
     for(j=i; j<=i+slidingwinlen-1; j++)
          printf("%c", substr[k++]=mineT[j]);
     substr[k]='\0';
     printf("...%s\n", substr);
     //chopoutstr(i, i+slidingwinlen-1);
     //printf("%s\n", substrxx);
   }
   //printf("%d\n", findpattern("TA", &state, mineT, false,
false));
}
#ifdef WITHCALLMCC

void CALLMCC(SYMBOL *text, Uint textlen)
{
 struct MccState state;
#ifdef DEBUG
  addspace(sizeof(struct MccState));
  addspace(textlen+1);
#endif
  ALLOCFUNC(textlen);
  MCCFUNC(&state,text,textlen,NULL,NULL);
  DEBUG0(2,"#");
#ifdef DEBUG
  showspace();
  DEBUG0(2,"\n");
  subtractspace(sizeof(struct MccState));
#endif
  currentbranchtabsize = 0;
  wrapspace();
}
```

```
#else
void mccsplit(char *filename,Uchar *text,Uint textlen,Uint
maxlen)
{
  Uint i, startpos, optlen;
  struct MccState state;
  SYMBOL *symboltext;
  short jj=0, k = 0;

  if(sizeof(SYMBOL) != sizeof(Uchar))
  {
    CALLOC(symboltext,SYMBOL,textlen);
    for(i=0; i<textlen; i++)
    {
      symboltext[i] = (SYMBOL) text[i];
    }
  } else
  {
    symboltext = (SYMBOL *) text;
  }

  if(maxlen == 0 || maxlen >= textlen)
  {
    optlen = textlen;
  } else
  {
    optlen = 1 + textlen/(textlen/maxlen+1);
  }
  for(startpos = 0; startpos < textlen; startpos += optlen)
  {
    if(startpos + optlen > textlen)
    {
      optlen = textlen - startpos;
    }
    ALLOCFUNC(optlen);
    DEBUG3(2,"#mcc of
%s[%u,%u]\n",filename,startpos,startpos+optlen-1);
    MCCFUNC(&state,symboltext+startpos,optlen,NULL,NULL);
  }
  if(sizeof(SYMBOL) != sizeof(Uchar))
  {
    free(symboltext);
  }

  displaytable(&state);
  bkread(&state);

  slidingwinfn(&state);

  wrapspace();
  // the suffix tree is built
```

```
}

#endif
#endif


// STGEMS Program
//
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stddef.h>

#define HASHSIZE 1111
#define slidingwinlen 5
#define winsize 6//slidingwinlen + 1
#define promoterlen 1000
#define LINE 80
#define NN 30
/*
This program implements STGEMS, by doing the following:
It extracts motif from submited input seq, returns the SEED
i.e d branch from
  root to leaf which is d unique pattern or SEED.
It use HASH table to store d position of the SEED, sorts it
and returns no of occurence
   of d SEED in each seq.
It implements the Hypergeometric distribution used in GEMS
i.e d combination function of N,n,M,m
It creates PWM of d SEEDs.
It computes P-Value and PWM of d SEEDs.
It computes similarity thresholh
It incorporates positional information by implementing Edit
distance
*/
static int occ[HASHSIZE], occ2[HASHSIZE];
char input[promoterlen], input2[promoterlen],
hashtab[HASHSIZE][winsize], patloc[HASHSIZE][NN];
char hashtab2[HASHSIZE][winsize], patloc2[HASHSIZE][NN];
char buffer[LINE];
double pvalue[HASHSIZE], pvalue2[HASHSIZE];

int hash_function(char *p){
     int hash_val=0;
     for(; *p; p++)
          hash_val = hash_val*6559+(*p);
     hash_val %= HASHSIZE;
     return abs(hash_val);
}
void chopoutstr(int b, int e){
     int i, j=0, index, found=0;
```

```
        char substr[winsize];

        for(i=b; i<=e; i++)
             substr[j++]=input[i];
        substr[j]='\0';
        index=hash_function(substr);
        printf("%s, %d\n", substr, index);

        if(!occ[index]){
             strcpy(hashtab[index], substr);//if hashtab loc
   is empty
             occ[index]=1; strcpy(patloc[index], "1");
             //printf("%s (%s), %d\n", substr, hashtab[index],
   occ[index]);
             //getchar();
        }else if(!strcmp(hashtab[index], substr))//if hashtab
   loc string is same with substr
             occ[index]+=1;
        else{//resolve collision
             //printf("%d\n", index);
             //getchar();
             for(i=index+1; i<HASHSIZE; i++){
                  if(occ[i]==0){
                       strcpy(hashtab[i], substr); found=1;
                       strcpy(patloc[i], "1");
                       occ[i]=1;
                       break;
                  }else if(!strcmp(hashtab[i], substr)){//if
   hashtab loc string is same with substr
                       occ[i]+=1; found=1;
                       break;
                  }
             }
             if(found==0){//start from the begin of the
   hashtab
                  for(i=0; i<index; i++){
                       if(occ[i]==0){
                            strcpy(hashtab[i], substr);
                            strcpy(patloc[i], "1");
                            occ[i]=1;
                            break;
                       }else if(!strcmp(hashtab[i],
   substr)){//if hashtab loc string is same with substr
                            occ[i]+=1;
                            break;
                       }
                  }
             }
        }
   }
   void chopoutstr2(int b, int e, int pronr){
```

```
	int i, j=0, index, found=0;
	static char substr[winsize];

	for(i=b; i<=e; i++)
		substr[j++]=input2[i];
	substr[j]='\0';
	index=hash_function(substr);
	printf("%s, %d\n", substr, index);

	if(!occ2[index]){
		strcpy(hashtab2[index], substr);//if hashtab loc
is empty
		occ2[index]=1; sprintf(patloc2[index], "%d",
pronr);
		//printf("%s (%s), %d\n", substr, hashtab[index],
occ[index]);
		//getchar();
	}else if(!strcmp(hashtab2[index], substr))//if hashtab
loc string is same with substr
		occ2[index]+=1;
	else{//resolve collision
		//printf("%d\n", index);
		//getchar();
		for(i=index+1; i<HASHSIZE; i++){
			if(occ2[i]==0){
				strcpy(hashtab2[i], substr); found=1;
				sprintf(patloc2[i], "%d", pronr);
				occ2[i]=1;
				break;
			}else if(!strcmp(hashtab2[i], substr)){//if
hashtab loc string is same with substr
				occ2[i]+=1; found=1;
				break;
			}
		}
		if(found==0){//start from the begin of the
hashtab
			for(i=0; i<index; i++){
				if(occ2[i]==0){
					strcpy(hashtab2[i], substr);
					sprintf(patloc2[i], "%d", pronr);
					occ2[i]=1;
					break;
				}else if(!strcmp(hashtab2[i],
substr)){//if hashtab loc string is same with substr
					occ2[i]+=1;
					break;
				}
			}
		}
	}
```

```
}

void mergetables(){
     int index, i, j, insdone=0;

     for(i=0; i<HASHSIZE; i++){
          if(occ2[i]==1){
               //printf("%s\n", hashtab2[i]);
               //getchar();
               insdone=0;
               index=hash_function(hashtab2[i]);
               if(occ[index]==0){ occ[index]=occ2[i];
strcpy(hashtab[index], hashtab2[i]);
                    strcpy(patloc[index], patloc2[i]);
               }else if(occ[index]==1 &&
0==strcmp(hashtab[index], hashtab2[i])){
                    strcat(patloc[index], ",");
strcat(patloc[index], patloc2[i]);
               }else if(occ[index]==1 &&
0!=strcmp(hashtab[index], hashtab2[i])){
                    //resolve collision and insert
                    for(j=index+1; j<HASHSIZE; j++){
                         if(occ[j]==0){ //cell empty
                              strcpy(hashtab[j],
hashtab2[i]); insdone=1;
                              occ[j]=1;
                              strcpy(patloc[j],
patloc2[i]);
                              break;
                         }else if(!strcmp(hashtab[j],
hashtab2[i])){//if hashtab loc string is same
                              strcat(patloc[j], ",");
strcat(patloc[j], patloc2[i]);
                              insdone=1;
                              break;
                         }
                    }
                    if(insdone==0){//start from the begin
of the hashtab
                         for(j=0; j<index; j++){
                              if(occ[j]==0){ //cell empty
                                   strcpy(hashtab[j],
hashtab2[i]);

                                   occ[j]=1;
                                   strcpy(patloc[j],
patloc2[i]);

                                   break;
                              }else if(!strcmp(hashtab[j],
hashtab2[i])){//if hashtab loc string is same
                                   strcat(patloc[j], ",");
strcat(patloc[j], patloc2[i]);
```

```
                                        break;
                                    }
                                }
                            }//
                        }
                    }
                }
            }

            int fact(int n){
                 if(n <= 1)
                      return 1;
                 return n * fact(n - 1);
            }

            int editdistance(char firststr[slidingwinlen], char
            secondstr[slidingwinlen]){
                 int i, dist=0;
                 for(i=0; i<slidingwinlen; i++)
                      if(firststr[i]!=secondstr[i]) dist+=1;
                 return dist;
            }

            void main(){
                 int i=1, j=1, k, M, m, len, segnr=1, pattnr=0, dist;
                 char hold[slidingwinlen]; //holding to thousand or
            temp hold
                 //for this test
                 int N=7, n=3; //n=3 indicates first three sequences
            constitute the positive set
                 double p_value=0, temp_pvalue=0;
                 int minnM;
                 double NM, A, B, minpvalue=0;

                 FILE *fp1, *fp2;
                 //strcpy(input, "TAATATTATTCTTTATTCGGTG");
                 //strcpy(input2, "TAATAAAGCTCTTCGCTCGGTC");

                 if((fp1 = fopen("gene-sample1.seq", "r")) == NULL)
                      fprintf(stderr, "Error opening file\n");

                 fgets(buffer, 160, fp1);// move from the ">" tag line
                 while(fgets(buffer, LINE-1, fp1) != NULL){
                      len=strlen(buffer);
                      if(buffer[0]!='>' && i==1){ buffer[len-1]='\0';
            strcpy(input, buffer); i++;}
                      else if(buffer[0]!='>' && i!=1){ buffer[len-
            1]='\0'; strcat(input, buffer);
                      }else break;
                 }
                 //for first input
```

```
     len=strlen(input);
     printf("%d\n%s\n", len, input);
     //getchar();

     for(i=0; i<(len-slidingwinlen+1); i++){
          chopoutstr(i, i+slidingwinlen-1);
     }

     for(i=0; i<=HASHSIZE; i++)
          if(occ[i]==1) printf("%d\t%s\t%s\n", occ[i],
hashtab[i], patloc[i]);// print all unique pattern


     fgets(buffer, 160, fp1);// move from the ">" tag line
     printf("%s\n", buffer);
     //getchar();
     while(fgets(buffer, LINE-1, fp1) != NULL){
          len=strlen(buffer);
          if(buffer[0]!='>' && j==1){ buffer[len-1]='\0';
strcpy(input2, buffer); j++;}
          else if(buffer[0]!='>' && j!=1){ buffer[len-
1]='\0'; strcat(input2, buffer);
          }else{
               segnr+=1;

               //for subsequence input
               len=strlen(input2);

               for(i=0; i<(len-slidingwinlen+1); i++){
                    chopoutstr2(i, i+slidingwinlen-1,
segnr);
               }

               for(i=0; i<=HASHSIZE; i++)
                    if(occ2[i]==1) printf("%d\t%s\t%s\n",
occ2[i], hashtab2[i], patloc2[i]);// print all unique
pattern

               //getchar();
               mergetables();
               printf("merge result..............\n");
               for(i=0; i<=HASHSIZE; i++)
                    if(occ[i]==1) printf("%d\t%s\t%s\n",
occ[i], hashtab[i], patloc[i]);// print all unique pattern

               //getchar();
               fgets(buffer, 160, fp1);// move from the ">"
tag line

               // initialize occ2, hashtab2, patloc2, etc
               j=1;
```

```
                    for(i=0; i<=HASHSIZE; i++){
                         occ2[i]=0;
                         strcpy(hashtab2[i], " ");
strcpy(patloc2[i], " ");
                    }
               }
          }
          fclose(fp1);
          //for the last sequence
          segnr+=1;

          len=strlen(input2);

          for(i=0; i<(len-slidingwinlen+1); i++){
               chopoutstr2(i, i+slidingwinlen-1, segnr);
          }

          for(i=0; i<=HASHSIZE; i++)
               if(occ2[i]==1) printf("%d\t%s\t%s\n", occ2[i],
hashtab2[i], patloc2[i]);// print all unique pattern

          //getchar();
          mergetables();
          printf("merge result..............\n");
          for(i=0; i<=HASHSIZE; i++)
               if(occ[i]==1){

                    p_value=0;
                    //compute M and m
                    len=strlen(patloc[i]);
                    j=0, k=0, M=0, m=0;
                    while(j<len){
                         if(patloc[i][j]==','){
                              hold[k]='\0'; k=0;
                              M+=1; if(atoi(hold) <= n) m+=1;
                         }else hold[k++]=patloc[i][j];
                         j++;
                    }
                    hold[k]='\0';
                    M+=1; if(atoi(hold) <= n) m+=1;

                    //compute p-value using the hypergeometric
distribution
                    if(n<M) minnM=n; else minnM=M;

          NM=((double)fact(N))/((double)(fact(M)*fact(N-M)));
                    for(j=m; j<=minnM; j++){

          A=((double)fact(n))/((double)(fact(j)*fact(n-j)));
                         B=((double)fact(N-n))/((double)(fact(M-
j)*fact(N-n-M-j)));
```

```
                        p_value+=(A*B)/NM;
                    }
                    pvalue[i]=p_value;
                    printf("%d\t%s\t%s\t%d\t%d\t%f\n", occ[i],
hashtab[i], patloc[i], M, m, pvalue[i]);// print all unique
pattern
                    occ2[pattnr]=occ[i];
strcpy(hashtab2[pattnr], hashtab[i]);
pvalue2[pattnr]=pvalue[i];
                    pattnr++;
                }
        //sort the content of hashtab in ascending order
pvalue
        minpvalue=1;
        for(i=0; i<pattnr-1; i++){
                minpvalue=pvalue2[i];
                for(j=i+1; j<pattnr; j++)
                    if(pvalue2[j]<minpvalue){
minpvalue=pvalue2[j]; k=j; }//k locate the current lowest

                temp_pvalue=pvalue2[i];
                strcpy(hold, hashtab2[i]);
                pvalue2[i]=pvalue2[k];
                strcpy(hashtab2[i], hashtab2[k]);
                pvalue2[k]=temp_pvalue;
                strcpy(hashtab2[k], hold);
        }
        for(i=0; i<pattnr; i++)
                printf("%d\t%s\t%f\n", occ2[i], hashtab2[i],
pvalue2[i]);// print all unique pattern

        printf("printing PWMs\n");
        //create PWM
        for(i=0; i<pattnr-1 && occ2[i]==1; i++){
                    printf("%s\n", hashtab2[i]);
                for(j=i+1; j<pattnr && occ2[j]==1; j++){
                    //printf("%d, %s\n", j, hashtab2[j]);
                    if(1==(dist=editdistance(hashtab2[i],
hashtab2[j]))){
                        printf("%s\n", hashtab2[j]);
                        occ2[j]=0;
                    }
                    //printf("%d, %s(%d)\n", j, hashtab2[j],
dist);
                    //getchar();
                }
                printf("printing next PWMs\n");
        }
}
```