FUNCTION-BASED ALGORITHMS FOR BIOLOGICAL SEQUENCES

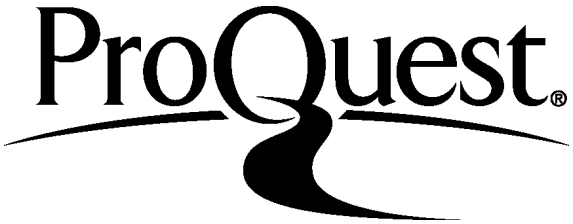
By

Pragyan (Sheela) P. Mohanty

M.S. Electrical Engineering, Southern Illinois University Carbondale.

A Dissertation
Submitted in Partial Fulfillment of the Requirements for the
Doctor of Philosophy Degree

Department of Electrical and Computer Engineering
in the Graduate School
Southern Illinois University Carbondale
December, 2015



DISSERTATION APPROVAL

FUNCTION-BASED ALGORITHMS FOR BIOLOGICAL SEQUENCES

By

Pragyan (Sheela) P. Mohanty

A Dissertation Submitted in Partial

Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

in the field of Electrical and Computer Engineering

Approved by:

Dr. Spyros Tragoudas, Chair

Dr. Haibo Wang

Dr. Lalit Gupta

Dr. Ada Chen

Dr. Kanchan Mondal

Graduate School
Southern Illinois University Carbondale
June 2015

AN ABSTRACT OF THE DISSERTATION OF

PRAGYAN P. MOHANTY, for the Doctor of Philosophy degree in ELECTRICAL AND COMPUTER ENGINEERING, presented on June 11, 2015, at Southern Illinois University Carbondale.

TITLE: FUNCTION-BASED ALGORITHMS FOR BIOLOGICAL SEQUENCES

MAJOR PROFESSOR: Dr. Spyros Tragoudas

Two problems at two different abstraction levels of computational biology are studied. At the molecular level, efficient pattern matching algorithms in DNA sequences are presented. For gene order data, an efficient data structure is presented capable of storing all gene re-orderings in a systematic manner. A common characteristic of presented methods is the use of binary decision diagrams that store and manipulate binary functions. Searching for a particular pattern in a very large DNA database, is a fundamental and essential component in computational biology. In the biological world, pattern matching is required for finding repeats in a particular DNA sequence, finding motif and aligning sequences etc. Due to immense amount and continuous increase of biological data, the searching process requires very fast algorithms. This also requires encoding schemes for efficient storage of these search processes to operate on. Due to continuous progress in genome sequencing, genome rearrangements and construction of evolutionary genome graphs, which represent the relationships between genomes, become challenging tasks. Previous approaches are largely based on distance measure so that relationship between more phylogenetic species can be established with some specifically required rearrangement operations and hence within certain computational time. However because of the large volume of the available data, storage space and construction time for this evolutionary graph is still a problem. In addition, it is important to keep track of all

possible rearrangement operations for a particular genome as biological processes are uncertain.

This study presents a binary function-based tool set for efficient DNA sequence storage. A novel scalable method is also developed for fast offline pattern searches in large DNA sequences. This study also presents a method which efficiently stores all the gene sequences associated with all possible genome rearrangements such as transpositions and construct the evolutionary genome structure much faster for multiple species. The developed methods benefit from the use of Boolean functions; their compact storage using canonical data structure and the existence of built-in operators for these data structures. The time complexities depend on the size of the data structures used for storing the functions that represent the DNA sequences and/or gene sequences. It is shown that the presented approaches exhibit sub linear time complexity to the sequence size. The number of nodes present in the DNA data structure, string search time on these data structures, depths of the genome graph structure, and the time of the rearrangement operations are reported.

Experiments on DNA sequences from the NCBI database are conducted for DNA sequence storage and search process. Experiments on large gene order data sets such as: human mitochondrial data and plant chloroplast data are conducted and depth of this structure was studied for evolutionary processes on gene sequences. The results show that the developed approaches are scalable.

DEDICATION

To the supreme power who I always believe. To my constantly encouraging parents, ever-supporting husband and two patient children.

ACKNOWLEDGMENTS

I would like to thank my dissertation adviser and committee chair Dr. Tragoudas for his invaluable guidance, insights, motivations and immense knowledge leading to this research and writing of this dissertation document. His guidance and patience helped me throughout this PhD research. My sincere thanks also goes to the four members of my dissertation committee Dr. Haibo Wang, Dr. Lalit Gupta, Dr. Ada Chen and Dr. Kanchan Mondal for their help and guidance on writing this document.

I would also like to express my sincere gratitude to Dr Sanjeev Kumar for the opportunity of working on an NSF funded project during a good part of my PhD studies. Many thanks to the Institutional Research staff at SIUC, especially Chris and Tim for their support while writing my dissertation. I would like to thank my fellow graduate students in Design and Automation lab for their support and productive discussions throughout my research work.

Last but definitely not least many thanks goes to my husband Dr. Manoj Mohanty for his constant support and guidance throughout this PhD work, to my parents and two sisters for their encouraging words and my children Anjalika and Aadarsh for their patience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The growing interest in genome research has caused an explosive growth in biological databases making it an increasingly challenging task to store and retrieve data efficiently. The length of the sequences and the expected depth of evolutionary relatedness introduces challenges in storing and manipulating biological data.

Key challenges include (a) how to manage the huge biological data in a compact manner and (b) how to provide a fast access of information through pattern search. It is well known that use of data structures (instead of a database or file) greatly accelerates computation speed, provided that the data structure is compact. Many existing search methods are based on the sequential database scanning which adversely affects the search efficiency. This motivated us to develop and implement novel algorithms for efficient data storage in terms of scalability of database size and very fast and efficient search methods in this dissertation research.

DNA (deoxyribonucleic acid) is large micro-molecule present in living organisms and is the main constituent of chromosomes that carries genetic information. It consists of two nucleotide strands. Each strand consists of a string of molecules called bases. The genetic information of DNA is encoded in the strands. The bases are typically referred as A, G, C and T, representing the initial letters of their name: Adenine, Guanine, Cytosine and Thymine. A strand of DNA, which can be up to 3 billion base pair long (in case of human), is expressed as a string over the finite set {A, C, G, T}. An example of a DNA strand is *AGCGGTGCG*. It is essential to be able to identify the locations of prespecified base sequences in a DNA database in order to justify certain genetic behaviors. Searches are complicated because some bases in the prespecified base sequence may be altered. Such searches are called approximate.

Changes of positions of the bases such as exchanging the positions of A with T in

the above example cause a local mutation. Evolution happens due to the exchanging of gene positions in a gene sequence present in a Genome. Such exchanges are called transpositions. A gene is a sequence of bases. The genes in the gene sequence or chromosome are represented by numbers $1, 2....n$. A unichromosomal genome is the genome that contains one gene sequence. For example, a unichromosomal genome with n=5 genes can be represented by 53421. Genome rearrangements are integral part of the evolutionary information between the species. Hence it is important and essential to develop an evolution based framework for embedding the questions regarding genome rearrangements. However because of the availability of vast data sets scientists tend to reconstruct the phylogenies based on gene orders, using a measure of evolutionary distance between two genomes which is a difficult computational problem by itself. Contrary to the traditional approaches this dissertation presents an evolution based graph structure to accommodate all possible rearrangements due to transpositions. The main algorithmic and computational strategies in the framework presented in this dissertation are based on well known and well established data structure used in Electronic Design and Automation area. The content of the document is arranged as follows:

Chapter 3 of this work deals with the problem of storing the DNA sequences compactly and decreasing the search time efficiently for exact search procedure. That is to provide algorithms, those report all sub-strings (their positions) that match the search-string S exactly. In this chapter novel binary function based encoding scheme for compact DNA sequence storage and search strings are presented. Very fast algorithms for exact search procedures are presented which operate on one of the encoded and stored DNA sequence. The encoding schemes and search procedures make use of parallel merging techniques which is very first. Experimental results section of this chapter proves these facts. The preliminary work of this chapter is published in 2011 ISCA conference of Bioinformatics and Computational Biology [44] and part of the advanced version of this work is published in Journal of Emerging Technologies in Computing [46]. A part of this

chapter is also published as a part of the paper in 2012 ISCA conference of Bioinformatics and Computational Biology [45].

Chapter 4 presents a scalable approximate search method on one of the compactly stored DNA sequences. That is to provide algorithms, those report all sub-strings (their positions) that match S with a mismatch score less than a pre-specified value, K. First an algorithm for approximate search procedure is developed for relatively shorter search strings and then a generalized approximate search algorithm for very long search strings are developed and presented in this chapter. Part of the Chapter 4 is published in Journal of Emerging Technologies in Computing [46].

Chapter 5 introduces a novel function based method to handle all possible rearrangements such as *transposition* operations in genomes efficiently and to store each rearranged gene sequences (chromosomes) in a very efficient data structure to construct the evolutionary graph structure. The approach presented in this chapter develops algorithms to transform the gene sequences to Boolean functions and stores them in the canonical data structure. The algorithm presented in this chapter for genome rearrangements does the rearrangement operations in parallel for the evolutionary processes such as *transposition* of multiple Genomes. There are also algorithms proposed for operators to find the history of a particular gene sequence. The experimental results section of this chapter shows that chromosomes from multiple Genomes can be rearranged and stored using *transposition* methods in polynomial time.

All the algorithms developed in this dissertation use binary functions and their representation on a very efficient data structure called the Binary Decision Diagrams (BDDs). The search algorithms uses built-in logical BDD operators. The time complexity of each operator is polynomial to the size of BDD. Chapter 5 elaborates on BDDs.

Concluding remarks are presented in Chapter 6.

CHAPTER 2

BINARY DECISION DIAGRAMS FOR BOOLEAN FUNCTIONS

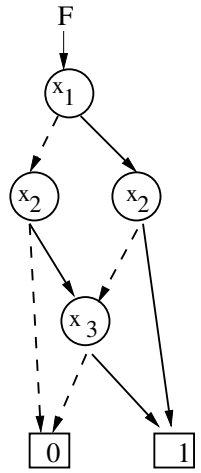
A Boolean function F consists of n binary variables (x_1,...,x_n), where each x_i has a binary value either 0 or 1. Given an assignment of binary values to each x_i, a boolean function evaluates either to 0 or 1. Given a function F(x_1,...,x_n), a product of all its variables in negative or positive polarity and which evaluates the function to 1 is called a *minterm*. When the product evaluates the function to 0, it is called an *offset minterm*. A product of variables, which does not necessarily include all variables, is called a *cube*. The variables that are not present in the cube are called the *Don't care* variables. These are the sets of input variables for which the output does not change in the function.

One of the many representations of Boolean function is in a tabular form and is called *truth table*. An explicit listing of all the minterms of the function is provided in its truth table. In the truth table, each element x_i of the domain has a row of the table listing the domain element x_i and the corresponding function value $f(x)$. The *truth table* is a canonical form for representing a function. BDD is a very compressed form of truth table. An example of a Boolean function $F = x_1'x_2x_3 + x_1x_2$ in the form of truth table, binary tree and binary decision diagram is given in Figure 2.1. One example of a *minterm* of the function is the product of variables $x_1'x_2x_3$ that evaluates the function to be 1. In the Function F, x_1x_2 is a *cube* where x_3 is a *don'tcare* variable.

Reduced Ordered Binary Decision Diagrams (ROBDDs) which are called BDDs in short are directed acyclic graph data structures and are widely used to represent Boolean functions compactly and uniquely. BDDs were first introduced by Akers [14] and further developed by Bryant [18]. The state-of-the-art data structure have gained wide spread application because of their compact canonical representation and efficient manipulation. A BDD has two terminal nodes labeled as 0 and 1 and many non-terminal internal nodes known as decision nodes. A BDD of a function F represents the function as rooted

x_1	x_2	x_3	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

(a) Truth Table



(b) Binary Decision Diagram

Figure 2.1. The Truth table and the BDD for $F = x_1'x_2x_3 + x_1x_2$

acyclic graph. Each non-terminal node N is represented by a variable v and its two edges directed towards two successor nodes: the "$Then$" (T) or $true$ edge and the "$Else$" (E) or $false$ edge. Each path from the root to sink-1 (or sink-0) gives a true (or false) output of the represented function. A *level* of a BDD of function $F(x_1..x_n)$ is a set containing all non-terminal nodes labeled with one particular variable x_i. A BDD is called ordered if the label of each internal node has a lower index than the label of its child-nodes. Being canonical, each BDD node represents a unique function. In Figure 2.1, $true$ edges are shown as solid lines, and the $false$ edges are shown as dotted lines. Logical operations on Boolean formulae, such as AND, OR, and XOR are performed in time polynomial to the size of the BDD graph. They are implemented efficiently because BDD is a canonical form for function representation. Since it is a canonical form, redundant nodes of the graph are eliminated and equivalent sub-trees are shared. Several BDD packages exist. We use the CUDD package [11].

The performance of function manipulation in any existing BDD package is boosted by the caching principles. A cache is where intermediate computation results are stored for

future reuse. In addition, all BDD packages use built-in operators which are fundamental function manipulation algorithms. These operators are used as subroutines for more complex algorithms on the stored functions. Each BDD package has its own set of BDD operators, whose time complexity is typically a lower order polynomial time function of the number of the nodes in the BDD graph. The CUDD built-in operators used by the developed algorithms are outlined as follows.

The built-in logical operations such as logical AND (*Cudd_bddAnd* [11]), logical OR (*Cudd_bddOr* [11]), logical co-factoring(*Cudd_bddCofactor* [11]) with respect to a variable etc. are used in the proposed algorithms. These are recursive procedures and hence very fast.

The logical AND of two functions f and g that produces a value of true if and only if both of its operands are true. Similarly, the logical OR of two functions f and g, that produces a value of false if and only if both of its operands are false. The logical cofactor of a function $f(x_1, x_2, ..., x_i, ..., x_n)$ with respect to variable x_i is $fx_i = f(x_1, x_2, ..., 1, ..., x_n)$ and cofactor of f with respect to variable x_i' is $fx_i' = f(x_1, x_2, ..., 0, ..., x_n)$. Detail explanation of the implementation of one of the BDD operators (*Cudd_bddAnd* [11]) is considered in next few paragraphs.

Let us examine in detail the special case that computes the conjunction (AND) of two BDDs. The pseudo code of AND operation is shown in Algorithm 1 [11]. All Boolean operations of two operands can be computed efficiently by the procedures that also compute the conjunction (AND) and other operations. This efficient approach is followed in many CUDD packages and the proposed approach uses the CUDD package by [11]. Algorithm 1 [11] employs two major data structures, one is the *unique table* and the other one is the *computed table*. The *unique table* is used to guarantee that a specific node is unique. If two nodes contain the same children and represent the same variable, they should be merged into one node. The *computed table* stores results of previous computation so that repeated computation are avoided. All these tasks require constant

time. If none of them prevails the *computed table* is consulted. The order of the operands is immaterial because conjunction is commutative. If the *computed table* lookup fails to return a result, the procedure solves the problem recursively (as shown in Algorithm 1). This algorithm first checks if terminal node has reached or else if the computed table already has an entry then it returns the result (see line 1-6). If these are not the case then it computes the *then* and *else* children of the resultant function by applying the AND procedure recursively (see line 7-10). In Algorithm 1, f_x and g_x are the *then* children of f and g respectively. Similarly f'_x and g'_x are the *else* children of f and g respectively. Once the resultant function is found the algorithm inserts that to *unique table* and returns the result (see line 12-16). The following example explains the Cudd_bddAnd operation of two functions f and g.

Consider functions f and g of the figure. The computation of $h = Cudd\_bddAnd(f, g)$ proceeds as in Example 1. The BDD of two functions f and g and the BDD of resulting function h are given in Figure 2.2.

Example 1: $AND(f, g) = (a, AND(p, t), AND(q', 0))$

$= (a(b, AND(1, 1), AND(0, u')), AND(q', 0)$

$= (a, (b, 1, 0), 0)$

$= (a, p, 0)$*unique table lookup*

In the above example the procedure first computes the *then* and *else* child by applying the AND operation recursively with respect to the first variable a. $AND(p, t)$ computes the AND of *then* children of f and g and $AND(q', 0)$ computes *else* children of f and g. Then the same procedure was followed recursively with respect to variable b for corresponding children nodes (see line 2 of Example 1). As it can be seen from the figure, since $AND(1, 1)$ is 1, and $AND(0, u')$ and $AND(q', 0)$ are 0, this leads to $(a, (b, 1, 0), 0)$(see line 3 of Example 1). Then before creating the node (b,1,0) the procedure looks up the *unique table*. In this case the node is already part of f. Therefore, of the three nodes of h(two internal plus one terminal node), only one needs to be created.

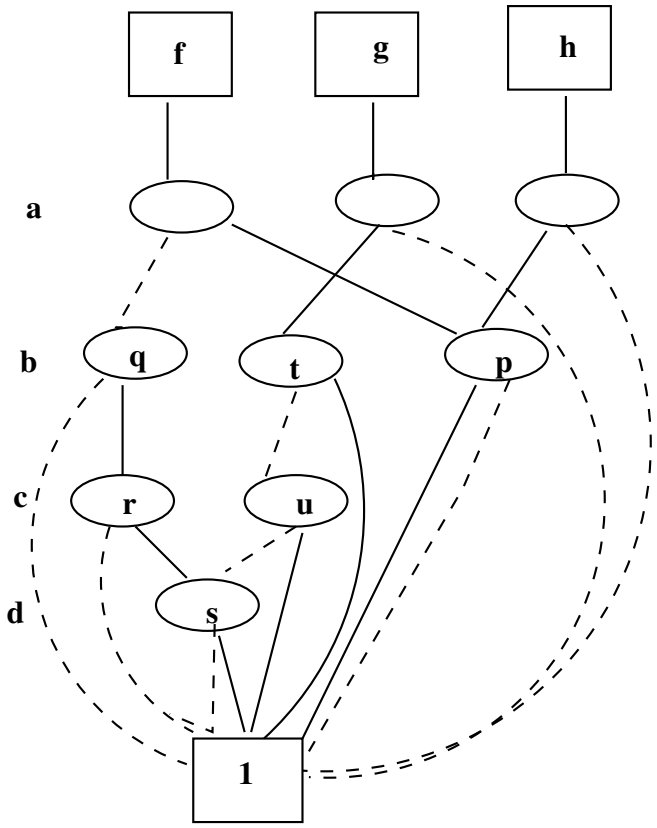


Figure 2.2. BDDs of a Cudd_bddAnd operation [11]

The time complexity of procedure AND (conjunction) between two functions f and g is established by observing that each pair of nodes u in f and v in g is examined at most four times (due to paths with different complementary parity reaching u and v). The recursion implementation of AND built-in operation requires $O(|f|.|g|)$ time, where $|f|$ is the number of nodes in f and $|g|$ is the number of nodes in g. However, because of the computed table, the recursion process is accelerated tremendously. This is the fundamental principle in implementing AND recursively. Each new result is stored in an entry of the table determined by computing a hash function. Each entry holds exactly one result and every conflict, results in the eviction of older result. The efficiency of such recursion implementations in BDD-based algorithms in electronic design automation is described in [36], [10] [25], among others. The time complexity of OR is also in

$O(|f|.|g|)$. This operation is also implemented recursively, and is very similar to AND. Another important built-in operation which is used in our algorithms is the computation of the co-factoring ($Cudd\_bddCofactor$ [11]) of a function with respect to one of its input variable or a function. Using this operator, cofactor of a BDD with respect to any node is efficiently computed using simple recursion. The time complexity of this operation depends on the number of nodes at each level and number of levels to be traversed. If the variable with respect to which the co-factoring is done is at the top then it takes less time than if it is towards the terminal node. Hence time complexity of each $Cudd\_bddCofactor$ operation is O($logx\ n_i\ n_j$), where x is the number of level traversed, n_i and n_j are the number of nodes at each level for each function.

Yet another built-in function called $Cudd\_bddclosestCube$ [11] is used in Algorithm 7 in Section 4.3. It finds a cube of input function F at a minimum Hamming distance K from the minterms of another function G. It invokes three sub-functions called $cuddBddClosestcube$,$separateCube$ and $CuddAddBddDoPattern$ which are described in the following paragraph.

Sub-function $CuddBddClosestcube$ performs the recursive step of the function $Cudd\_bddclosestCube$ to compute the distance of each cube of F to all the minterms of G, and represent this cube with distance as a single Algebraic Decision Diagram (ADD). The ADD which is a multi-terminal BDD represents a function from a factored Boolean domain to a real-valued range as a directed acyclic graph. It is a decision tree with re-convergent branches and real-valued terminal nodes. ADDs like BDDs enforce a strict variable ordering on the decisions from the root to the terminal node, enabling a minimal, canonical diagram to be produced for a given function. Thus, two identical functions will always have identical ADD representations under the same variable ordering. For detail information on ADD, Colorado University Decision Diagram Package [11] can be referred.

Sub-function $cuddBddClosestcube$ returns the cube at a hamming distance K or less, if successful; NULL otherwise. It calculates the Hamming distance as a side effect.

Sub-function *cuddBddClosestcube* uses a four-way recursion to examine all four combinations of cofactors of functions f and g according to the following formula [11] in Equation 1 and hence they are very fast.

$$H(f,g) = \min \{\mathrm{H}(f_t,g_t), \mathrm{H}(f_e,g_e), \mathrm{H}(f_t,g_e)+1, \mathrm{H}(f_e,g_t)+1 \} \qquad (1).$$

Let f_t and f_e denote the $Then$ and the $Else$ function of f, and g_t and g_e denote $Then$ and $Else$ functions of function g at one particular level. *cuddBddClosestcube* works on two BDDs by traversing and comparing between $Then$ and $Else$ functions of each node at each level as stated in the equation 1. It is computed recursively which returns Hamming distance by examining the four combinations of cofactors of two input function F and F_s. Once the distance is calculated the distance and cube are combined to form ADD (Algebraic Decision Diagram).

In the next step of the function *Cudd_bddclosestCube* another sub-function *separateCube* is called to separate the cube from the distance and returns the cube of minimum distance.The sub function *separateCube* finds out which branch of the ADD points to the distance and replaces the top node of that branch with one pointing to zero instead. Therefore we are left with the only cube that points to distance.

Finally in *Cudd_bddclosestCube*, the cube is converted from ADD to BDD by using another built-in sub-function *CuddAddBddDoPattern*. This is a procedure to create a BDD by replacing all ADD terminals which are greater than K with $'0'$ and rest with $'1'$ This is a recursive procedure that converts a ADD to BDD and returns a pointer to the resulting BDD if successful, NULL otherwise.

The following focuses on time complexity analysis of *Cudd_bddclosestCube*. Let, K be the input distance and N_n and M_m denote the number of nodes in the BDD representation of both functions. The upper bound of the distance is used as the bound of the depth of the recursion. Hence time complexity increases with increasing hamming distance. The overall time complexity of *Cudd_bddclosestCube* is $\mathrm{O}(KN_nM_m)$.

Another important built-in operation which is used in our algorithms is a swap function

called Cudd_bddSwapVariables [11]. This function swaps two sets of variables of the same size (x and y) in the BDD of a function f.

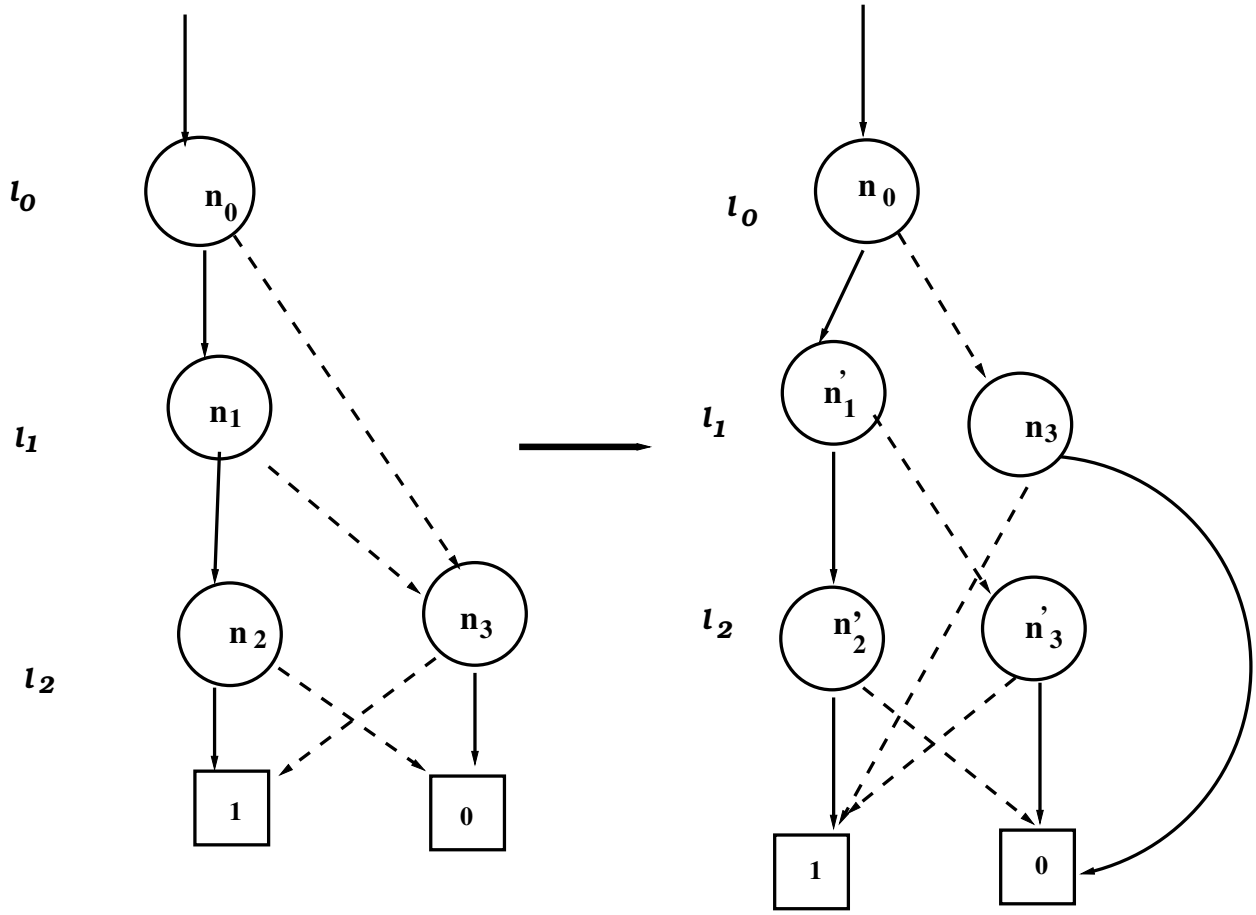


Figure 2.3. Example of Gene Swap In a BDD

Algorithm 1: Conjunction Algorithm AND(f,g) [11]

Input: Functions f and g.

Output: AND result of f and g.

```
1  AND (f,g);
2  if terminal case then
3  |  return result;
4  end
5  if computed table has entry{f g} then
6  |  return result;
7  else
8  |  let x be the first variable of {f g}
9  |  t = AND(f_x, g_x);
10 |  e = AND(f'_x, g'_x);
11 end
12 if t = e then
13 |  return r =t ;
14 end
15 r = FindOrAddUniqueTable(x,t,e);
16 insertComputedTable({f ,g},r)
17 return r;
```

Example 2: The BDD presented in the left side of Figure 2.3 has a permutation $l_0 l_1 l_2$. The right side of this figure depicts the permuted BDD of the same function. In order to swap level l_1 with l_2 we have to change the permutation to $l_0 l_2 l_1$. Hence one has to apply the swapping routine described in In Algorithm 11 to all nodes at level l_1. In this particular example there is one node in level l_1 and are two nodes in level l_2, but the affected node is n_1 . Using cofactoring it can be represented as $n_1 = (l_1, n_2, n_3)$. We can rewrite the equation for node n_1 as

$n_1 = l_1(l_2, 1, 0)(l_2, 0, 1)$.

From swapping procedure above we know that the resulting new node is

$n_1' = l_2(l_1, 1, 0)(l_1, 0, 1)$ which can be written as $n_1' = (l_2, n_2', n_3')$. In this case n_3 stays unchanged, because it directly points to terminal nodes and can be explained as follows:

$n_3 = (l_2, 0, 1)$

After Swapping:

$n_3 = (l_1, 0, 1)$

Swapping variables does not change the functionality of the original function.

The Swap operation is a local operation and only involves the number of nodes that are in the two levels that are to be swapped. Hence this is a linear operation and proportional to the number of nodes involved in the swapping. Space taken during swap is only due to temporary node creations during SWAP procedure. In a symmetric function the overall number of nodes does not change.

CHAPTER 3

DNA SEQUENCE STORAGE AND EXACT SEARCH PROCEDURES TO IDENTIFY A DNA STRING

3.1 INTRODUCTION

The continuous growth in biological data lengthens the search procedure. This makes it essential for researchers to work on improving both data storage and timing factor of search process. All the biological data are stored in huge databases where space is not a concern. Databases are normally too large to fit into the main memory of a computer. The data of a database resides in secondary memory, generally on one or more magnetic disks. However, to execute queries or modifications on data, the data must be transferred to main memory for processing and then transferred back to disk for persistent storage. These procedures take a lot of time. For faster data processing, data structures are ideal. Although data structures are more efficient for accessing data than external databases, memory problems could occur if they are not built efficiently and compactly.

The algorithms proposed here are centered on a very efficient data structure called Binary Decision Diagrams (BDDs). The presented algorithms in the paper are developed to generate state of the art tool sets for very fast search process and efficient storage of biological sequences similar to the algorithms developed for design, verification and test automation technologies of digital VLSI circuits. BDDs are used extensively in Electronic Design and Test Automation; see [27, 21, 36, 10, 25] among others.

A pattern search or a string search is for finding the occurrences of a particular sub-string (or multiple sub-strings) of a DNA or protein sequence within a large DNA/Protein sequence. In molecular biology, there are several databases holding DNA, RNA and amino acid strings that are continuously growing. Therefore, there is a need of very fast and scalable string matching algorithms to find the patterns quickly and efficiently. The key challenges are (a) how to manage the huge biological data in a compact manner (b)

how to provide a fast access of information through pattern search.

String search or pattern matching algorithms can be exact or approximate. In exact pattern matching the input is a DNA sequence of N bases (usually called *text* in literature) and a search-string (also called *pattern* or *query string* in literature) S of M bases. The goal is to identify all the sub-strings in the DNA sequence that are identical to the search-string S, and for each such sub-string to report its position within the DNA. i.e., the base number of the first base in the sub-strings.

Example 1: Let's consider the DNA sequence $D = \mathbf{AG\underline{CG}}\underline{CG}\mathbf{AGCG}$ and sub-string $S = AGCG$. In this case $N = 10$ and $M = 4$, respectively. There are two sub-strings in the DNA that match S. They are highlighted in bold font, and their positions are 1 and 7 respectively.

String search has many applications; the main applications of proposed algorithms are geared towards the retrieval of biological sequences for which storage and analysis are challenging algorithmic problems. The proposed methods can be applied in other biological sub problems such as: sequence alignments (global alignments [39], local alignments [63], alignment of similar strings); computing the Longest Common Sub-sequences, finding exact motifs [64] etc.

The objective of this chapter is to store the DNA sequences compactly and decrease the search time efficiently for exact search procedure. That is to provide algorithms, those report all sub-strings (their positions) that match S exactly.

This chapter presents two methods for preprocessing the database and storing DNA sequence in a compact data structure. Although this approach can be applicable to any type of sequence database and string search, but this case is aimed for applications on DNA. The proposed approach concentrates on the offline version of DNA string matching case where DNA sequence stays unchanged for searches with different search-string, and we have the whole sequence available before the searches. Such a sequence can first be preprocessed into a suitable form that makes the subsequent searches faster. Hence we

propose algorithms for preprocessing of the DNA sequence and the associated algorithms to search for exact occurrences of search-strings with an upper bound in one of the data structures developed for preprocessed sequence.

3.2 BACKGROUND AND RELATED WORK

There are several versions of the methods for both exact and approximate search procedure, where the pattern is preprocessed but the text is not [37, 4, 9]. Their complexity varies from $O((K + LogM)N/M)$ to $O(MN)$. When they are operating directly on the database, they are called online version. Such approaches are slow when they are operating on very large strings. Offline versions are more suitable for large string searches where typically a data structure is built prior to any string searches. Examples of offline methods can be found in the literature [47, 22, 23, 30, 7] among others.
In index based searching, some kind of index is built and used to locate the exact occurrences of the selected pattern pieces. There are basically four different data structures used in literature for indexing method [7]. $Suffix\ trees$ allow searching for any sub-string of the text. $Suffix\ arrays$ allow same operation but are slower.
$Q-grams$ and $Q-samples$ consider searches for text sub-strings no longer than Q. The proposed method also searches for sub-strings no longer than an upper bound.
The indexing method used for exact pattern search proposed in [30] constructs index table for the whole DNA sequence for exact pattern matching for multiple strings. However search time for very large DNA sequences are not reported in [30]. A more generalized version of exact matching is reported in [23] using suffix arrays. A sparse suffix array is used as underlying data structure for the storage of DNA sequences in this method. The method proposed in [23] finds maximal exact matches in large sequence data. A maximal exact match (MEM) is defined as a match between two sequences that can't be extended in either direction towards the end or beginning of the sequences without introducing a mismatch.

Although suffix arrays are more space-preserving than suffix trees, they still require large memory space for indexing massive data such as the whole genome of a human. The space consumption of an ordinary suffix array for a given DNA sequence of N characters is $O(NlogN)$. The occurrences of a sub-string of length M characters can be found in $O(MlogN)$ time by searching the pattern on the suffix array by binary search procedure. There are several attempts to reduce the space of the suffix trees or arrays [6] [3] [48]. They built a data structure for the index. Then to answer queries they need both text as well as the index. Hence extra space is still required. The space and time performance of proposed method is compared to an index based method developed by [43] which uses a suffix tree. This method only indexes beginning of blocks produced by Ziv-lempel compression. The general idea of Ziv-lempel compression is to replace sub-strings in the text by a pointer to a previous occurrence of them. This type of compression algorithm is based on a dictionary of blocks. As each new block computed it is added to the dictionary. Two data structures (LZtrie and RevTrie) are created using blocks of text. Then pattern search is done by finding the occurrence in a single block or in two blocks or more than two blocks.

The size bottleneck of suffix trees has made the research turn into looking for more space-economic variants of suffix trees. One popular alternative is the suffix array [Manber and Myers 1993]. It basically removes the constant factor of suffix trees to 1, as what remains from suffix trees is a lexicographically ordered array of starting positions of suffixes in the text that occupies *nlogn* bits. Many tasks on suffix trees can be simulated by log n factor slowdown using suffix arrays. With three additional tables, suffix arrays can be enhanced to support typical suffix tree operations without any slowdown [Abouelhoda et al. 2004]. However the most space efficient suffix tree occupies 25GB main memory, hence needs to reside in the disk.

Many existing methods reduce the search space using matching statistics. Such approaches return a subset of matching sub-strings as opposed to the proposed method

which precisely returns all the matching sub-strings. BLAST [39] and FASTA [40] are some of such fast search engines which do not ensure that all matching sub-strings will be found. BLAST is a widely used search tool to find best local alignments. This is a heuristic approach based on Smith-Waterman [35] algorithm. BLAST uses a strategy which is expected to find most matches, but sacrifices on the accuracy of the DNA position that matches the input string in order to gain speed. Sequences are filtered to remove low complexity regions. It finds W consecutive bases of the query string that match exactly to the database sequence (this is also called seed). Once an exact word-match is found between the query sequence and a database sequence, the comparison between the two sequences is extended towards left side and right side to obtain the final alignment. Increasing seed W results in a decrease of sensitivity while decreasing W results in too many random matches. Another statistical parameter that signifies the BLAST search is the *expectation value* E which is defined as the number of matches expected to occur randomly with a given score. The smaller the E value, the more likely the match to be significant. BLAST typically runs on NCBI database. However there is a version where one can set up sequence database on a computer and run blast queries. This is called Local BLAST. The default value of E for BLASTn (tool version of searching the nucleotide database) is 10. The proposed method is also compared against BLAST for time performance and actual number of occurrences of the search strings.

The methods proposed in [13] use weighted indexing procedures and use R-tree data structure for both exact and approximate pattern search. These methods are called the Basic Signature Index (BSI) and the Weighted Signature Index (WSI) methods respectively. These methods also use a filtering method to reduce the search space. In each of these methods a sliding window is placed at every possible location of a DNA sequence to extract its signature by considering the occurrence frequency of each DNA base. Then it stores a set of signatures using index based R-tree data structure, and

assigns a weight to each position of the window. It tries to scatter the signature over indexing space and tries to reduce the space. Then the query sequence is converted to multidimensional rectangle and looks for the overlapping signatures.

This R-tree based method presents a better performance in search procedures for exact matches as compared to suffix trees which has time complexity $O(M + k)$, where M is search-string length and k is number of occurrences.

There are two methods presented in this chapter to store DNA sequences. These methods target to generate and store DNA sequence implicitly which will enhance the future search technique in this sequence. We propose two DNA encoding schemes and generation techniques that are not only very fast and space efficient but also will enhance the future search performance. An algorithm for search string generation technique is also presented here which will operate on one of the developed data structure for DNA storage. Subsequently we have also presented two Algorithms for exact search procedure.

3.3 FUNCTION BASED APPROACH FOR PATTERN SEARCH

3.3.1 Position Specific Method for Generation of DNA Functions from DNA Sequences

In this proposed method, an input DNA sequence is converted to a Boolean function called the DNA function F. The number of variables in the minterms of these functions is calculated from the size (upper bound) of the search-string. Any sub-string of size upto L can be searched in this DNA sequence. Similarly, j number of sub-strings with an upper bound of L can also be searched in this DNA sequence. Hence each minterm of the DNA function corresponds to a DNA string of size L within the sequence where L is the maximum size for that string sequence in the DNA. These minterms also contain some additional variables called the position bits. During a search, the position bits are used to report the positions of matched strings in the DNA sequence.

The number of bits (or variables) in each minterm of the Boolean function is determined as follows: Each base is represented by two bits as shown in Table 1. Thus, there are twice as many such bits as the upper bound of the search-string size. For example, if the upper bound of a DNA search-string consists of 20 bases, then each minterm has 40 bits. Such bits are called variable bits. Then the additional position bits (besides the variable bits) are calculated from the length of the DNA sequence. The number of minterms present in the function also depends on the length of the DNA sequence and the upper bound of the search-string. Let N be the length of the DNA sequence in number of bases and the upper bound of search-string is L. Then the number of minterms in the DNA function is $M_n = N - L + 1$. The number of variable bits in each minterm is $\lceil log_2(2L) \rceil$ and the number of position bits are found out from the formula $\lceil log_2 (M_n) \rceil$. Then each of them is represented in binary form. For example a 4 - bit binary representation of decimal 10 is 1010. The proposed algorithm Function-Generate (Algorithm 2) generates, stores and manipulates function using BDDs which are canonical form for representing functions.

Table 3.1. DNA Encoding.

DNA Bases	Binary Value
A	00
G	01
C	10
T	11

The approach consists of two parts. The first part is a parser that transforms a conventional DNA string in to a function (see Algorithm 2). This algorithm is inherently linear to the size of the DNA. Once the DNA function is formed, searches for a DNA string are done very quickly, in time that depends on the number of the nodes in the

BDD. The number of nodes is much less than the length of the DNA. The string search process is also experimentally shown to exhibit sub-linear behavior. Therefore this approach is expected to perform very fast.

Algorithm Function-Generate which is given in Algorithm 2 transforms the DNA sequence into binary strings, using two bits per base as in Table 3.1. Procedure Generate $M_i = (m_0 m_1 ..... m_{2L-1} b_0 ... b_{k-1})$ creates first part (base part of each minterm) as well as the position part of M_i of the DNA function (see line 5 of Algorithm 2). It uses $2L$ variables, $m_0 m_1 ..... m_{2L-1}$, one binary variable per bit in the first part of the minterm M_i. Basically, each M_i is a single minterm function where literals m_j, for $0 \leq j \leq 2L - 1$ and b_f, $0 \leq f \leq k-1$, are set according to the respective bit value in the encoded DNA string. This procedure is implemented efficiently using built-in function *Cudd_bddAnd* from the BDD package. Once minterms M_i are generated, they are stored in function F using another built- in operation *Cudd_bddOr*. This is shown in procedure *CreateFunction* of Algorithm Function-Generate (See line 6 of Algorithm 2). Such an implementation requires time linear to the BDD size (the number of nodes in a BDD), which in practice, is much smaller than the size of DNA sequence. Hence the time complexity of Function-Generate is linear to the size of the DNA sequence. The following example illustrates the encoding process and DNA function generation.

Example 2: Let us consider a DNA sequence, $D = \{AGCGGTCGCGTGCG\}$. Each base is encoded using 2 bits (See Table 3.1). Hence the DNA sequence of length 14 is encoded to 28 bits as shown in Figure 3.1. Similarly, let us consider a search-string $S = \{AGCG\}$ and it is encoded to 00011001 using Table 3.1. The number of bases in the search-string, which is also the size of the search-string is 4. Therefore, there are $14 - (4 - 1) = 11$ minterms in the DNA function. Since the upper bound of the search-string is 4, there are 4 bases or 8 variables in each minterm. The first part of each minterm,(also called base part) consists of 8 bits. The second part of each minterm contains position bits and for this example it is $\lceil log_2 (11) \rceil = 4$. The first minterm is

encoded from the 1st sub-string ($AGCG$) which is in position 0. The minterm is 000110010000. The base part is 00011001 and the second part (position bits) is 0000. The next minterm is created from the next sub-string GCGG using same methodology, as shown in Figure 3.6.

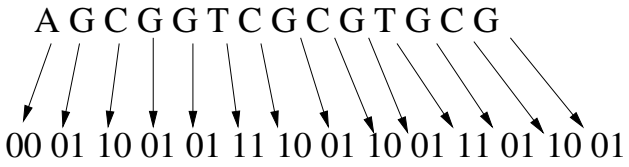


Figure 3.1. Encoding of a DNA Sequence

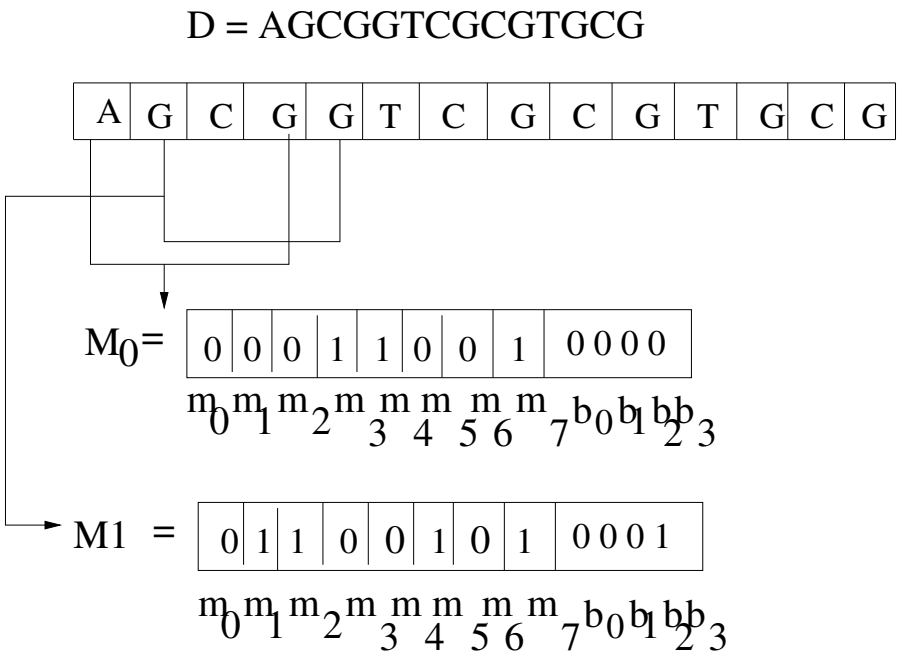


Figure 3.2. Minterms of a DNA Function

The BDD of a DNA function $D = AGCGGTCGCGTGCG$, generated by Algorithm 2 is given in Figure 3.3. The first eight nodes represent base part for each minterm of the DNA function and last set of nodes represent the position bits. This is a very compact graph where only one literal has two nodes. Function String-Generate in Algorithm 4 is

Algorithm 2: Function-Generate

Input: DNA sequence D and Upper bound (L) of search-string S

Output: BDD of DNA function(F)

1 Let k = $\lceil\ log_2(N - L + 1)\ \rceil$;

2 F = 0 (the empty function);

3 / * Encoding DNA sequence (D)

4 **for *each*** $i = 0$ *until* $N - L + 1$ **do**

5 **Generate** $M_i = Cudd\_bddAnd(m_0 m_1 ... m_{2L-1} b_0 ... b_{k-1})$ / * Built-in BDD operator

6 **Create Function** $F = Cudd\_bddOr(F, M_i)$; / * Built-in BDD operator

7 **end**

8 Report F;

used to generate another BDD graph for string function from search-string, $S : AGCG$ as shown in Figure 3.4. The nodes in the BDD graph of the string, $S = AGCG$ represent the bits encoded from the base part. There are no position bits in the string function. The reason for this transformation is that the problem of string search will reduce to a fast Boolean function manipulation problem as shown in Section 3.3.3

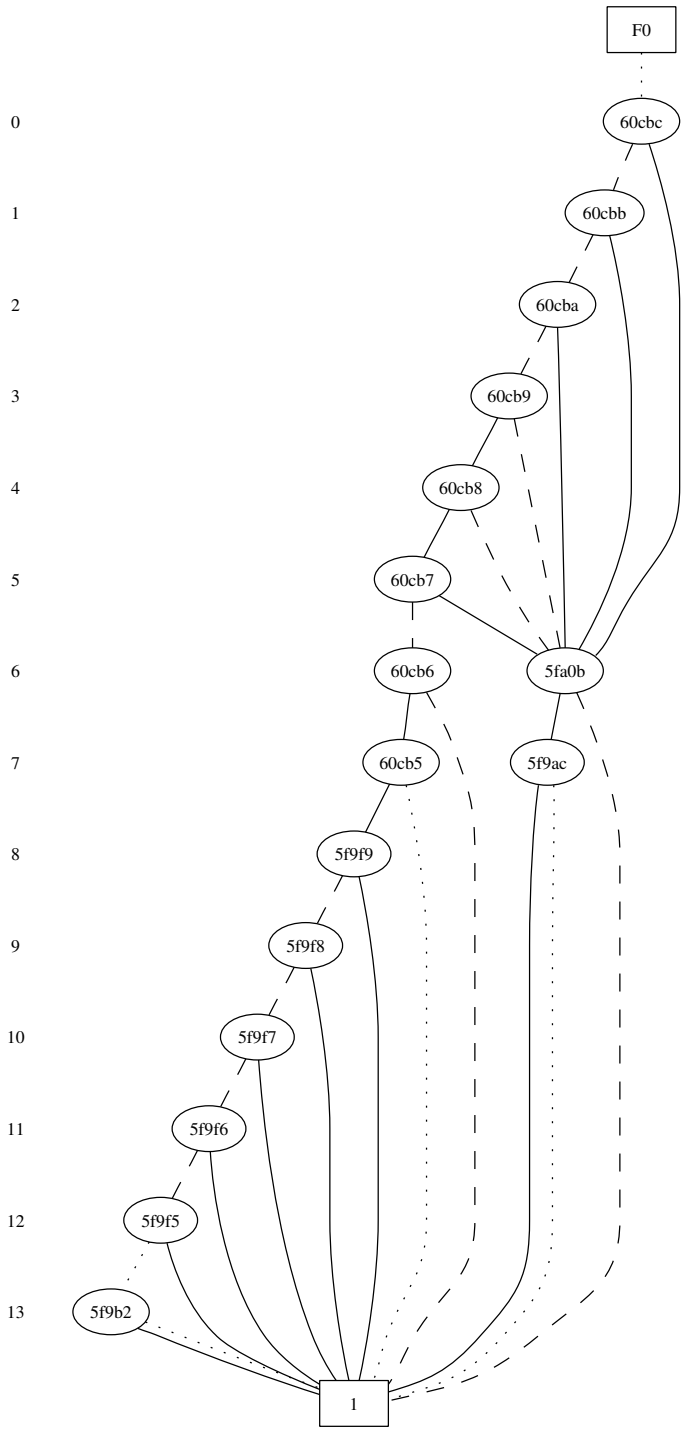


Figure 3.3. The BDD for a DNA Sequence D=AGCGCGAGCG

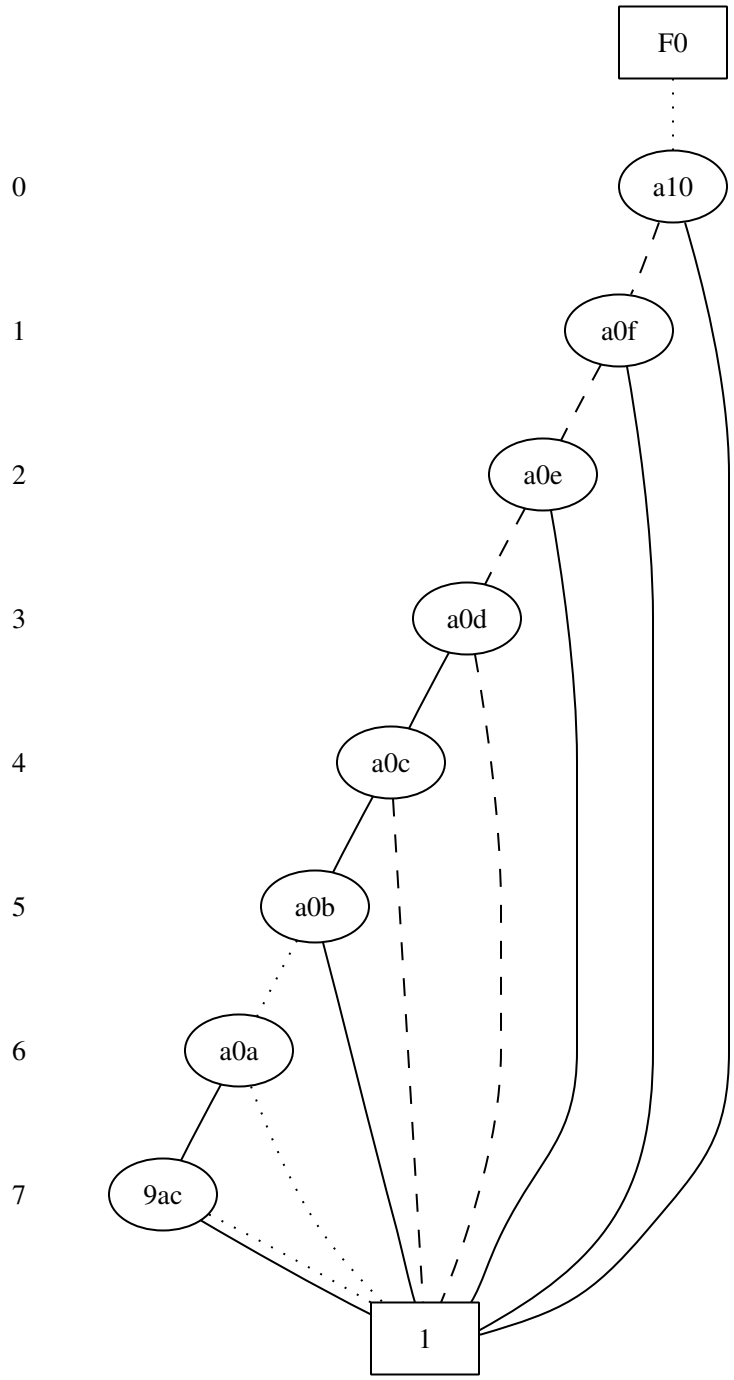


Figure 3.4. BDD for a Search-String S=AGCG

The following describes the time complexity of Algorithm 2 that generates the DNA function F. Let, N_n be the maximum number of nodes in the BDD that holds F. The

generation is done by successively ORing (using *Cudd_bddOr)* a string of $L + logN$ variables, where N is the size of the reference DNA sequence and L is the maximum size of sub-string. This is done in O$(N - L)$ times, because we have to traverse the DNA string. The ORing process is proportional to the number of nodes in the resultant BDD of the functions. The time complexity of each iteration is O $(L + logN)$. Hence the time complexity of this algorithm is $(N_n\ (L + logN))+\ N$.

Function generation is an inherently parallelization task, and thus scalable. One can use P processors to convert distinct portions of the DNA sequence in to separate functions. The number of processors depends on the time the user is willing to spend on the execution of Function-Generate. The i^{th} processor, $0 \leq i \leq P$, works on the i^{th} portion of the DNA string, and generates its BDD function using additional b bits adding at the beginning of the k b-bits for each minterm in its function. In particular, $logP$ more such bits will be used at the second part of each minterm in the function, and will form a $logP$ bit-long most significant portion in the b-part of each minterm. These extra bits should be assigned binary values uniformly so that the decimal equivalent of $logP$ bit, most significant part for each minterm in the function of processor i is exactly i. Once the functions are formed, each odd numbered processor sends its function to its adjacent even numbered processor which ORs the two functions.That way the number of functions has been halved. This process is repeated recursively by only focusing on the processor that have formed the new functions. The process terminates in $logP$ steps, each time performing an OR operation in parallel.

3.3.2 Encoding Process and Function Generation for DNA with Implicit Base Positions

The DNA sequence can be encoded into binary string by employing the encoding scheme as shown in Table3.1. Subsequently, a boolean function is generated from this binary string, which is called DNA function F. Every bit in the binary string is a minterm in

function F either in the on-set or in the off-set of F, as tabulated in the truth table of Figure 3.5.

The illustration of this process is given with an example.

Example 3:

To illustrate, consider a DNA sequence $D = GGGT$ of length L. As we scan through the DNA sequence D, the encoded binary bits(using the encoding table) in D are 01010111. The encoded binary bits from the DNA sequence is shown in Figure 3.6. In this case, each of both the onset and offset minterms of F represents an individual base bit. Algorithm 3 generalizes this process. The number of variables n in the minterms of the function F can be calculated from the equation L= 2^n.

Hence $n = logL/log2$

As we scan through this DNA sequence, we compute minterms m_0 and m_1 which corresponds to base G. Since base G is encoded as 01, m_0 is offset and m_1 is onset minterms in F. BDD operator $Cudd\_bddComputeCube$ is used to compute the minterms which takes constant time. This process is repeated for every subsequent base in DNA sequence. For each such base, minterms are computed and included in function F by using BDD operator $Cudd\_bddOR$. This way a BDD is constructed from these built in operators.

The Advantages of the proposed encoding algorithm is as follows: (a) The proposed method will only form one function per DNA sequence and will allow for multiple quick searches on the function for different string sizes.

(b) This method is much more compact, because it does not have any positional variables, instead the minterms of the function generated serve as the positions of the DNA bases.

(c)This particular encoding scheme can be used to generate any random function. The construction method of a random function does not necessarily require that we parse an input DNA sequence and this can save time. This in turn implies that in order to evaluate the capacity limitations of the data structure, one has to input very large DNA

Algorithm 3: Simple-DNA Function-Generation

Input: DNA sequence (D)

Output: DNA Function (F)

1: Calculate Number of variables (n)

2: Determine the number of minterms (M_n)

3: Initialize the function (F)

4: Initialize a variable $k = 0$)

5: / * set elementary BDDs corresponding to variables */

6: **for** i = 1 until M_n **do**

7: **if** D[i] = 'A' **then**

8: Go to next base

9: **else if** D[i] = 'G' **then**

10: next minterm = $Cudd\_bddComputeCube()$

11: /* Generate minterm from i+k+1 value and number of variables */

12: function $F = Cudd\_bddOR$ (F ,next minterm)

13: / * Built-in BDD operator to create function F

14: **else if** D[i] = 'C' **then**

15: minterm = $Cudd\_bddComputeCube$ () /* Generate minterm from i+k Value and number of variables*/

16: function $F = Cudd\_bddOR$ (F ,minterm) / * Built-in BDD operator

Algorithm 3: Simple-DNA Function-Generation continued

```
17:   else if D[i] = 'T' then
18:     minterm = Cudd_bddComputeCube () /* Generate minterm from i+k Value and
        number of variables*/
19:     next minterm = Cudd_bddComputeCube ()
20:     /* Generate minterm from i+k+1 Value and number of variables*/
21:     function F = Cudd_bddOR (F ,minterm) / * Built-in BDD operator
22:     function F = Cudd_bddOR (F ,next minterm) / * Built-in BDD operator
23:   else
24:     Not a valid base;
25:     k = k + 1;
26:   end if
27: end for
28: Report BDD of function F ;
```

sequences which is a very time consuming task. In contrast, the capacity limitations of the proposed data structure can be evaluated quickly using huge synthetic DNA sequences that can be generated very fast in an implicit manner to the length of the resulting sequence. It is only required to explicitly read a small DNA sequence and successive Oring it quickly to form large DNA sequences. It's also a parallelization task and can be done as mentioned in Section 3.3.1.

(d) Even if the positions of the DNA bases in this encoding scheme is not explicitly specified, a quick search can be done to access the positions using *Cudd_PrintMinterm* function from the CUDD package. This function operates in constant time.

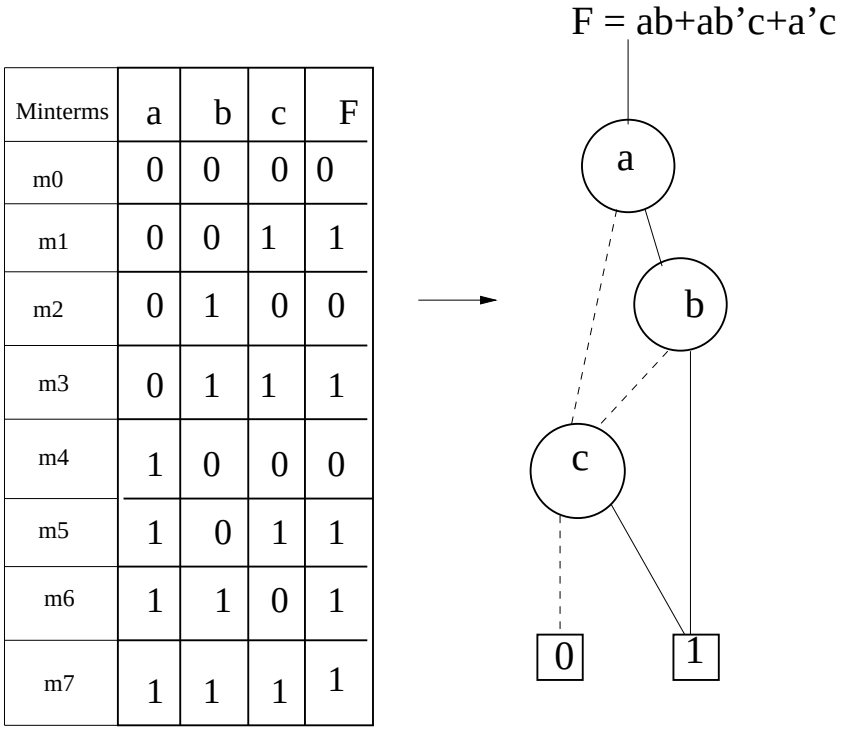


Figure 3.5. The truth table and the BDD for $F = ab + ab'c + a'c$

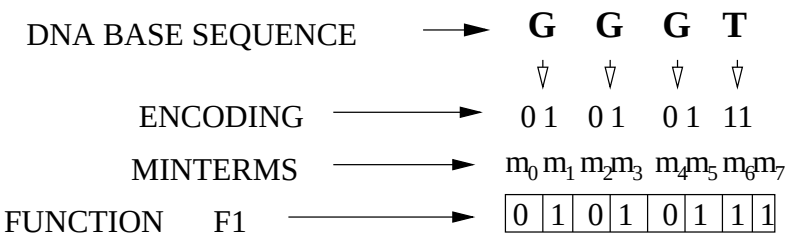


Figure 3.6. Encoding Process of DNA Bases

Scanning through the DNA sequence and encoding will always take the time O(n), where n is the number bases in a DNA sequence. Hence construction time is proportional to the number of bases in the DNA sequence. However the Algorithm is very space preserving. For example in order to store a sequence of base length 4 of a DNA sequence we need at most 8 minterms. Because of binary encoding scheme we only need 3 variables to create such function. As an example, we can generate a function for DNA sequence size 10^{12} with the help of about 40 variables. The experimental result section proves this fact.

3.3.3 Algorithms for Exact Pattern Search

A K-mismatch(F, F_S) algorithm denotes an algorithm that returns position in the DNA sequence, where the mismatch to the input string does not exceed number K. This section presents an exact string-search approach, called the 0-mismatch(F, F_S)-basic . The 0 indicates that the the mismatch is 0 or otherwise, the match is exact.
Algorithm 0-mismatch (F, F_S)-basic assumes that the string is represented as a binary function. This is accomplished in a pre-processing step by procedure Generate String-Function shown in Algorithm 4 (String-Generate). Algorithm String-Generate uses similar notion as Function-Generate to encode the search-string S. Let L denote an upper bound of the string S which is a part of the input description. If the string size is $s < L$ then the last $L - s$ bits will be considered as a *don't care* in the sequence and any encoding process applies for these bits. Note that the string function F_S consists of one

Algorithm 4: String-Generate

Input: Search-string (S), Upper bound of Search-string(L)

Output: BDD of search-string Function(F_S)

1 $F_S = 0$;

2 **for** ***each*** *i = 0 until 2L-1* **do**

3 **Generate** $M_i = Cudd\_bddAnd(m_0 m_1 ... m_{2L-1})$

4 **Generate String-Function** $F_S = M_i$

5 **end**

6 Report F_S;

minterm encoded from the entire search-string using Table 1, and has no position bits as described in Algorithm String-Generate.

In Algorithm 5 is called (0-mismatch(F ,F_S)-basic). The input consists of two binary functions: Function F that represents the DNA sequence is formed by Algorithm $Function - Generate$. Function F_S is formed by Algorithm $String - Generate$ and represents the search-string. Initially 0-mismatch forms a binary function T by the conjunction (AND operation) of functions F and F_S(see also step 1 of Algorithm 5). Using the built-in AND ($Cudd\_bddAnd$) operation, all the minterms, whose variable bits match the string are found. This is done implicitly. This is precisely the benefit of encoding the DNA sequence and strings, as functions.

Once these minterms are found, their position bits are extracted by using the built-in procedure $Cudd\_bddCofactor$ (see also steps 2-4 in Algorithm 5). This is also done implicitly (another advantage of using functions). Subsequently the number of appearances of the search-string in the DNA sequence is returned using built-in function $Cudd\_CountMinterm$. If the user is interested in finding that the string exists in the DNA sequence, $Cudd\_PickOneMinterm$ operation is invoked. Alternatively, if one wants

Algorithm 5: 0-mismatch(F ,F_S)-basic

Input: DNA Function(F) from Algorithm 2 and search-string Function(F_S)from Algorithm 4

Output: Number of occurrences of sub string S in DNA sequence D and position of S in D

```
1 Generate T(m_0 m_1 ... m_{2L-1} b_0 ... b_{k-1})=Cudd_bddAnd(F , F_S)
2 for each i = 0 until 2L - 1 do
3 |   P = Cudd_bddCofactor(T, m_i)  / * Built-in BDD operator
4 end
5 Find Positions( Cudd_PrintMinterm(P)) / * Built-in BDD operator
6 Find Number of occurrences(Cudd_CountMinterm(P))  / * Built-in BDD
  operator
```

to see all the positions, then *Cudd_PrintMinterm* operation is invoked to enumeratively present all of them. In general, all of the described procedures are built-in BDD operators where execution time is linear to the number of nodes in the BDD of the DNA function. The time complexity of the 0-mismatch algorithm experimentally shows to be sub linear to the DNA sequence size.

The following example demonstrates how the proposed algorithm works.

Example 4: Let us assume that the search-string is AGCG (as in the previous example) and the DNA sequence $D = AGCGGTCGCGTGCG$. The following describes the encoding of each minterm of the function using Algorithm 2.

Step1: The position bits are calculated to be 4. The first minterm has variable bits 00 01 10 01 which correspond to AGCG. The position bits are set to 0000. Likewise, the second minterm has variable bits 01 10 01 01 which correspond to GCGG. The position bits of

second minterm are set to 0001. Hence the DNA function F is encoded as F = 00 01 10 01 0000 + 01 10 01 01 0001 + ...

Step 2: Next the search-string is encoded using Algorithm 4 as 00 01 10 01 $--$$--$. The position bits for this search string are don't care bits.

Step 3: Next the string search procedure is done using Algorithm 5. The $Cudd\_bddAND$ operation in line 1 of Algorithm 5 returns the function T which has common the minterms of DNA function, F and string function, F_s. In this case $T = Cudd\_bddAND(F, F_s)$ = 00 01 10 01 0000. The position function, P is determined by the $Cudd\_bddCofactor$ in line 3 of Algorithm 5. In this example P = 0000. Hence the sub-string AGCG is at the position 0 in the DNA sequence D. The above is illustrated in Figure 3.7 with detailed encoding.

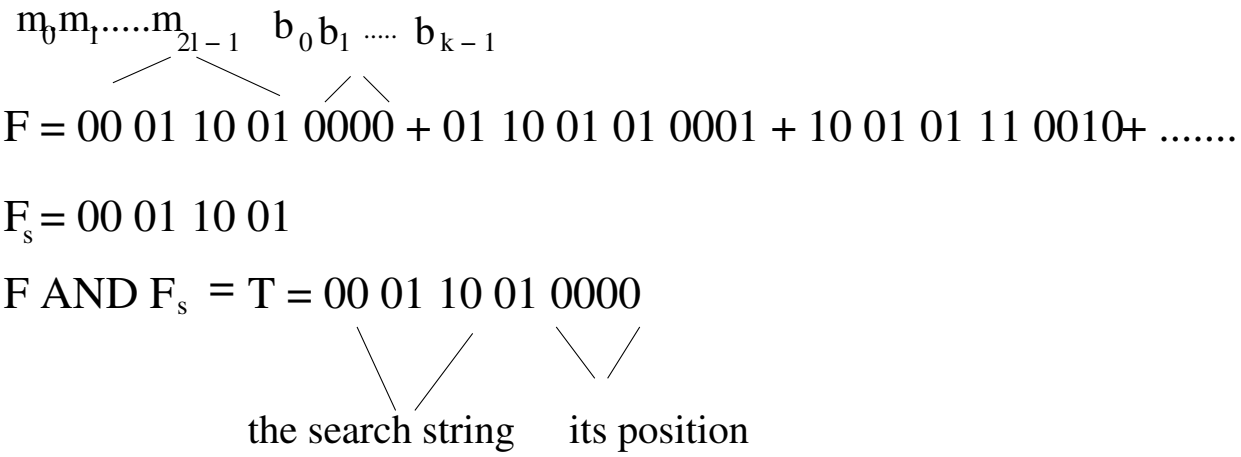


Figure 3.7. Search operation in a DNA Function

Algorithm 4 generates a string function from the search-string that has $2M$ context variables, where M is the size of the search-strings in number of bases. The string function does not have any position variable. This is done by $O(M)$ traversals on the search-string and at each traversal we create the bits for the minterm. This process takes $O(M)$ time. This process is very similar to generating DNA function except that the string function, F_S is a single minterm function.

The proposed Algorithm 5 inherently searches strings in parallel. This is an inherent property of function based search methods. For example, for any number of search strings s_i, the search algorithm can be used to report the matching sub-strings. Once the search string function F_s is constructed, the search time for multiple strings is practically identical to the search time for one string.

Example 5:

Step 1: Search string function for 3 search strings s_1, s_2, s_3 is expressed as: $F_s = ms_1+ms_2+ms_3$, where ms_1,ms_2,ms_3 are the minterms of search string function generated from strings s_1, s_2, s_3.

Step 2: Let the DNA function of a DNA sequence which contains all these strings and other strings is given by $F_d = ms_1+ms_2+ms_3 + ms_4+ms_5+ms_6$.

Step 3: The resulting function is: $F_r = (Cudd\_bddAND(F_d,F_s)) = ms_1+ms_2+ms_3$. $Cudd\_bddPrintminterm(F_r)$ will report the minterms, which are the matching strings.

Algorithm 5 does the implicit string search and reports the respective position of matching strings in DNA sequence D and number of occurrences. Let M_m be the number of nodes in the BDD that holds the respective string function F_S and N_n is the number of nodes of the BDD that holds respective DNA function F. Let n_a be the number of nodes in the resulting BDD. In practice, n_a may be larger than max $\{N_n, M_m\}$. An ANDing of DNA function, F and string function, F_S in line 1 requires O(M_m N_n), which in practice is O(N_n).

The for loop (lines 2-4) in Algorithm 5 consists of $O(k)$ co- factoring to eliminate all string variables, where k is the number of position variable. Those co-factoring guarantee that the resulting function consists of those minterms that matches string S. Every co factoring takes O($logx$ n_i n_j) time where, x is the maximum number of nodes at any level in the BDD and n_i , $n_j < n_a$. However k is typically $LogN$ and hence the complexity is in practice O(n_a $LogN$). Therefore the overall time complexity is O(n_a).

When the string size increases the number of variables in string function as well as

number of variables in the DNA function increases. In order to handle very large search-strings, Algorithm 0-mismatch(F,F_S)-basic is modified and is called 0-mismatch(F,F_S)-general(See also Algorithm 6). In this approach when the input search-string exceeds a certain length called threshold length S_{th}, it is split into multiple strings of roughly equal size and then multiple search functions are created one per sub-string from them. The value of S_{th} is determined and its value has a significant impact in the performance of the approach. Our experimental results in Section 5.5 show that value of S_{th} is around 100 to optimize the performance of the approach. The minterms of the DNA function are the same size as the minterms of each of these string functions. The overall search time is calculated by adding the search time for each of these string function. By proceeding this way, the memory requirement for storing the output function of each AND operation is significantly less and therefore the overall search process is significantly faster when comparing to 0-mismatch(F,F_S)-basic that performs a single AND operation.

Algorithm 6: 0-mismatch(F ,F_S)-general

Input: DNA Function(F) from Algorithm 2 and Search-string (S):SIZE = X

Output: Number of occurrences of S in D and BDD of search string Function(F_S) and Positions of Matching sub-strings In the DNA sequence D

1: Number of Search function to generate y= ⌈ X/s ⌉;

2: let s = optimum size;

3: let string-func-count = 0;

4: $S_t = X - y$;

5: $F_0 = Cudd\_bddAnd(m_0\ m_1 ... m_s)$;

6: T_0 =$Cudd\_bddAnd(F, F_0)$

7: / * Checking initial conditions

8: **if** T_0 = Null **then**

9: Print(String not present);

10: Exit;

11: **else**

12: **for each** i = 0 until 2L-1 **do**

13: P_0 =$Cudd\_bddCofactor(T_0), m_i)$ / * Built-in BDD operator

14: **end for**

15: **end if**

Algorithm 6: 0-mismatch(F ,F_S)-general-continued

16: **for each** j = 1 until y-1 **do**

17: String-Generate (F_{Sj}) ; / * from Algorithm 4

18: T_j = *Cudd_bddAnd*(F,$F_S j$);

19: **for each** i = 0 until 2L-1 **do**

20: P_j=*Cudd_bddCofactor*(T_j, m_i);

21: **end for**

22: **end for**

23: **if** Shift(P_{i-1},S_t) != P_{j-1} **then**

24: exit;

25: **else**

26: string-func-count ++;

27: j = j++;

28: **end if**

29: **if** string-func-count = y -1 **then**

30: **Cudd_PrintMinterm** (P_0) / * Find Positions using Built-in BDD operator

31: **Cudd_CountMinterm** (P_0) / * Find Number of occurrences using Built-in BDD operator

32: **end if**

If the threshold length is S_{th} for search-string is s, a DNA function of minterm size s is generated. Let us assume that the next search-string is of size $4s$. Four string functions Fs_0, Fs_1, Fs_2 and Fs_3 of size s are generated. In case four equal sized functions can not be formed, then *don't care* bits can be used to form the last function. Then the search for each string function is performed.

Example 5: Consider that the DNA sequence $D = AGCTAGCT$ and search-string $S = AGCT$. For simplicity, we will form two search functions from strings AG and CT. Hence search function Fs_0 will be formed from AG and search function Fs_1 will be formed from string CT. Position of search function formed from CT will be shifted by a number equal to the search-string size minus the number of the search function. Hence if the position of AG in the DNA sequence is 0 then position of CT will be $0 + (4 - 2) = 2$. The shift operation is called $Shift$(See also line 20 of Algorithm 6). Each shift operation will increase the minterms of a function by the number S_t. For example, Shift$(P, 2)$ will increase the minterms by 2. Where P is a function to be shifted. This shift function is an implicit procedure which operates on the nodes of the BDD of a graph instead of each minterm and hence is very fast. For example, if 000 is a minterm of a function p to be shifted, then applying shift(p,2) will shift the minterm 000 to 010. When the search is performed this condition is checked in order to find whether the whole string $AGCT$ is present in the DNA sequence. This enhanced Algorithm for exact pattern search is described in Algorithm 6. A detailed explanation on algorithm of shift function can be found in [45] Input to Algorithm 6 is the DNA function from Algorithm 2. It is assumed that the search-string size is longer than S_{th}. In Algorithm 6, initialization is done in lines 1 to 5. If the initial search-string is present, we proceed further or else we exit (See also lines 7-9). The steps in from line 10 to 20 are similar to string search Algorithm 5 and are repeated for each of the string function to be generated. Then it is checked whether the current position is an increment of the previous position (See lines 20 -26). If true, the current string function is generated from the part of the search-string

so the process continues. If not true, we proceed to the next function generation. Finally, if all the string functions are present, then the position of initial string function is printed. Now, let us consider the example, where DNA sequence $D = AGCTAGCT$ and search string $S = AGCT$. According to Algorithm 6, search function Fs_0 will be formed from AG and will be encoded as 0001 and search function Fs_1 will be formed from CT and will be encoded as 1011. After all the initial conditions are satisfied, function T_0 can be represented as $T_0 = 0001000 + 0001100$ and hence position function P_0 will be represented as $P_0 = 000 + 100$. Therefore, we proceed with next function Fs_1. For function Fs_1 T_1 is 0111010 +0111101. Then, we check for the condition shift(P_j, 2) is equal to (P_{j-1}). Since shift $(000 + 100) = 010 + 101$, we continue. Then print P_0 (Initial Position Function) will provide the position of the search-string which are 000 and 100. Let M_m and N_n be the number of nodes in the BDDs that hold the respective function F_m and N_n. Let n_a be the number of nodes in the resulting BDD. The time overhead of the proposed algorithm is the time to generate the extra search functions and the time to do each search. When the search-string size goes beyond a predetermined threshold S_{th}, we divide the search-strings to form multiple search functions. The number of AND operations is equal to the number of search functions, which is a small constant. ANDing (Cudd_bddAND) of functions F_n and F_m requires O(M_m N_n), which in practice is O(N_n). The space complexity of the search algorithms of our approach is proportional to the number of nodes in the resultant BDDs. The number of nodes of a DNA function are less than actual number of bases in the DNA sequence, i.e. O(n_a) $\ll O(N)$. This is definitely more compressed than the space of suffix tree approaches that need O(N^2) to report all possible position pairs that correspond to matches.

3.4 EXPERIMENTAL RESULTS

The experiments were performed on large DNA sequences in standard reference sequences (in FASTA format) downloaded from NCBI [42] databases and on some benchmarks

associated with the BLAST [39] tool ranging from (10 to 100 Mbps). Large reference sequences were chosen to evaluate the performance in large data sets. The DNA sequences downloaded from NCBI [42] databases have their identification number (GI Numbers) and are stored in a separate file. The sizes of different DNA sequences are reported in respective tables. The DNA sequences associated with BLAST tool are identified as GI:6164828 and GI: 308801197. The search process of several sub-strings of different lengths were done on these DNA sequences for exact search. The proposed methods (0-mismatch(F, F_S)-basic, 0 -mismatch(F, F_S)-general were evaluated for a wide range of search string (query) length and their search time for exact search procedure. Each of search strings are named as SB-sb, where s is the size of the search string in base. The modified DNA sequence of size s is denoted as $DB - sMB$ similar to modified search-string, where s is the size of the DNA sequence and MB denotes mega bases. The proposed algorithms have been implemented in the C language. Experiments were performed on the UBUNTU platform having 4GB RAM and with a speed of 3 GHz. This procedure was repeated for several sub-strings. The proposed exact string search method was compared with method [43] in terms of time and space performance. The code for the method proposed in [43] was downloaded from the site provided by the paper and implemented on the workstation that we used for implementing our algorithms. The proposed string search methods were also compared with BSI and WSI methods [13] in terms of time performance. We could not obtain the sources of the implementation of this index-based method from the authors. Hence we took the same size DNA sequences and search strings and compared against the results reported by the authors. The experiments of the proposed method were performed in the workstation having the same configuration as reported in the methods of [13].

Figure 3.8 provides the size of computer memory (in bytes) needed to store and corresponding BDD construction time (in millisecond) of a particular DNA sequence using Algorithm Function-Generate. Since the search process depends on the number of

nodes in the BDD of a DNA function, the number of nodes of the BDD of corresponding DNA sequence is also provided in this figure. This figure lists the of space requirement of a particular DNA sequence. The DNA sequences with their GI numbers are provided in the X-axis. The Y axis represents the comparison of memory size and BDD construction time to store a particular sequence with number bases present in that sequence. Note that the number of nodes and memory size of a BDD of a sequence is significantly less than the number of bases present in that sequence. This indicates that the storage space of a DNA function is much less than the DNA sequence itself. Although, the worst case performance is exponential, this condition never observed after performing so many experiments. In fact the actual run time is linear in the size of each operand.

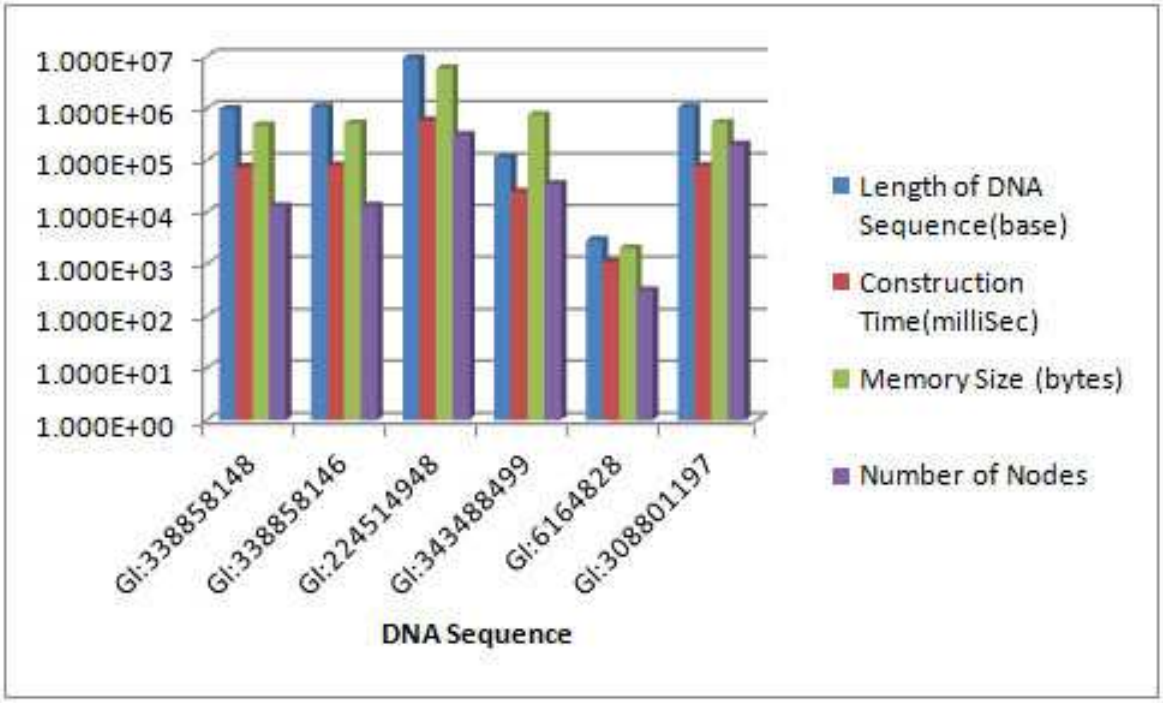


Figure 3.8. DNA sequence length in base vs BDD construction time in millisecond, number of nodes in BDD and memory size in bytes

Table 3.2 tabulates the results and demonstrates the capacity of our data structure which is capable of storing large DNA bases using Algorithm 3(Simple-DNA Function-Generation). Column 1 of Table 3.2 indicates the name of different DNA

Table 3.2. Performance of the data structure for genomes using Algorithm 3

Name of Genome	length of - Sequence (in bases)	Time to Construct (in sec)	nodes in BDD graph	number of variables
Human(GI:338858148)	987716	0.05	26135	20
Human(GI:338858146)	1064303	0.1	80532	25
Human(GI:224514948)	9412842	0.2	140032	25
Synthetic Data1	1048576	0.1	358	20
Synthetic Data2	1073741824	0.2	6309	30
Synthetic Data3	1.09x 10^{12}	2.1	129,389	40
Synthetic Data4	1.15x 10^{18}	4.2	214,505	60

sequences. Some of these data are real DNA sequences from NCBI database and some of the data marked as synthetic data are generated from the DNA functions of real DNA function by merging the sequence functions with *Cudd_bddOr* operations. Column 2 provides the length of the DNA sequences in bases, which is being stored in our compact data structure. It is observed that data structure created in the proposed method has the capability to store huge DNA sequences as shown in Column 2. The time taken in seconds to construct our data structure is shown in Column 3. Column 4 displays the number of nodes in the BDD graph. Column 5 of this result table presents the number of variables in the DNA function generated from a DNA sequence of size ($2^{variables}$). As shown in Table 3.2, the proposed method is capable of storing DNA sequence in the order of 1.15x10^{18}. Table 3.3 provides the time performance comparison of 0-mismatch (F, F_S)-basic and 0-mismatch (F, F_S)-general with existing approaches [43] [39] [13]based on various

searches on a 10 Mbps DNA sequence from [42]. The sequence size is chosen for comparison purpose for the method [43] and [13].

Column 1 of Table 3.3 provides the search-strings of various lengths and Column 2 provides the time to search in second by proposed method. Column 3 provides the search time for LZ-index method, column 4 and column 5 provide the search time for BSI and WSI methods respectively. Column 6 provides the search time for BLAST method. As it can be seen for lower search string length the proposed method outperforms the BLAST method and LZ-index method. The search time of LZ-index method is comparable to ours, but we could not go over search string size 60 bps for LZ-index. This could be the fact that the indices were constructed over 60 bps sequence size. However it can be seen from the results reported, that the search time for string length 20bps to 60 bps of LZ-index method increases with the size of search string and higher than proposed method. The search time for the proposed method is also faster than BLAST and BSI and WSI. The increase in search time with the search string length is linear.

A comparative study on time performance of 0-mismatch (F, F_S)-basic and 0-mismatch (F, F_S)-general procedures vs LZ-index and BLAST [39] is provided in Table 3.4. A 100Mbps DNA sequence was used to compare with BLAST [39] and disk based method of [43] in Table 3.4. Column 1 of Table 3.4 provides the search-strings of various lengths and column 2 provides the time to search in second by proposed method. It can be seen that the search time for these multiple patterns is less than the search time of these patterns, when the search is done sequentially. The search time for LZ-index method is provided in Column 4. Column 5 provides the search time for BLAST method.

Table 3.5 elaborates on the multiple pattern search time of the developed method. Search time of proposed method for 1 pattern of length corresponding to column 1 is provided in Column 2. Similarly the search time for 10, 100 and 1000 patterns is provided in Column 3, 4 and 5 respectively. The developed method inherently searches strings in parallel, hence as shown in this table, the search time for multiple strings is practically

identical to the search time for one string.

A comparison of the number of positions for exactly matched strings for both BLAST and proposed method (0-mismatch (F, F_S)-basic and 0-mismatch (F, F_S)-general) are provided in Figure 3.9. The X-axis represents the DNA sequences of various sizes from [42] and the input search-strings used for the search. The Y-axis represents the number of positions that matched the input string exactly. Often there is no exact match of the search-strings found in the DNA sequences. Hence the DNA sequences are modified by inserting search-strings at various positions in order to demonstrate that some positions that matched the string can be missed by BLAST. These results demonstrate that BLAST does miss some of the positions.

Table 3.3. CPU time performance of proposed method vs [43] [13] [39] in the 10Mbps DNA sequence from [42]

Search-string size (in base)	Proposed Method sec	LZ-index [43] sec	WSI [13] sec	BSI [13] sec	BLAST [39] sec
20	0.0000	0.007	N/A	N/A	0.010
30	0.0000	0.008	N/A	N/A	0.010
40	0.0000	0.009	N/A	N/A	0.010
60	0.0000	0.010	N/A	N/A	0.010
128	0.0000	N/A	5	5	2.000
256	0.0001	N/A	0.012	0.015	3 .000
512	0.0051(0.0015 [a])	N/A	0.008	0.012	3 .000
1028	0.01 (0.005 [a])	N/A	0.011	0.012	4.000
2056	0.050 [a]	N/A	N/A	N/A	6.000
4112	0.1200 [a]	N/A	N/A	N/A	7.000
8224	3.00 [a]	N/A	N/A	N/A	8.000
fragment-1(1347 bps)	0.005 [a]	N/A	N/A	N/A	4.000

[a]Results obtained using 0 -mismatch(F, F_S)-general

Table 3.4. CPU time performance of proposed method vs [43] [39] in the 100Mbps DNA sequence from [42]

Search-string size (in base)	Proposed Method sec	LZ-index [43] sec	BLAST [39] sec
20	0.0010	0.010	0.210
30	0.0011	0.018	0.210
40	0.0012	0.019	0.210
60	0.0013	0.010	0.210
128	0.015 (0.005 [a])	N/A	4.000
256	0.018 [a]	N/A	5 .000
512	0.58 [a]	N/A	7 .000
1028	1.01 [a]	N/A	8 .000
2056	2.050 [a]	N/A	9.000
4112	3.1200 [a]	N/A	10.000
8224	4.00 [a]	N/A	12.000
fragment-1(1347 bps)	1.11 [a]	N/A	7.000

[a]Results obtained using 0 -mismatch(F, F_S)-general

Table 3.5. CPU time performance for multiple strings of developed method in the 100Mbps DNA sequence from [42]

Search-string size (in base)	developed Method (1 Patterns) sec	developed Method (10 Patterns) sec	developed Method (100 Patterns) sec	developed Metho (1000 Patterns) sec
20	0.0010	0.0015	0.0016	0.0018
30	0.0011	0.0012	0.0018	0.0020
40	0.0012	0.0021	0.0019	0.0021
60	0.0013	0.0022	0.0024	0.0025
128	0.015	0.023	0.024	0.028
256	0.018	0.025	0.028	0.030
512	0.58	0.65	0.68	0.70
1028	1.01	1.21	1.25	1 .28
2056	2.050	2.150	1.165	2.168
4112	3.120	3.220	3.240	3.280
8224	4.00	4.20	4.25	4.450
fragment-1(1347 bps)	1.11	1.15	1.16	1.210

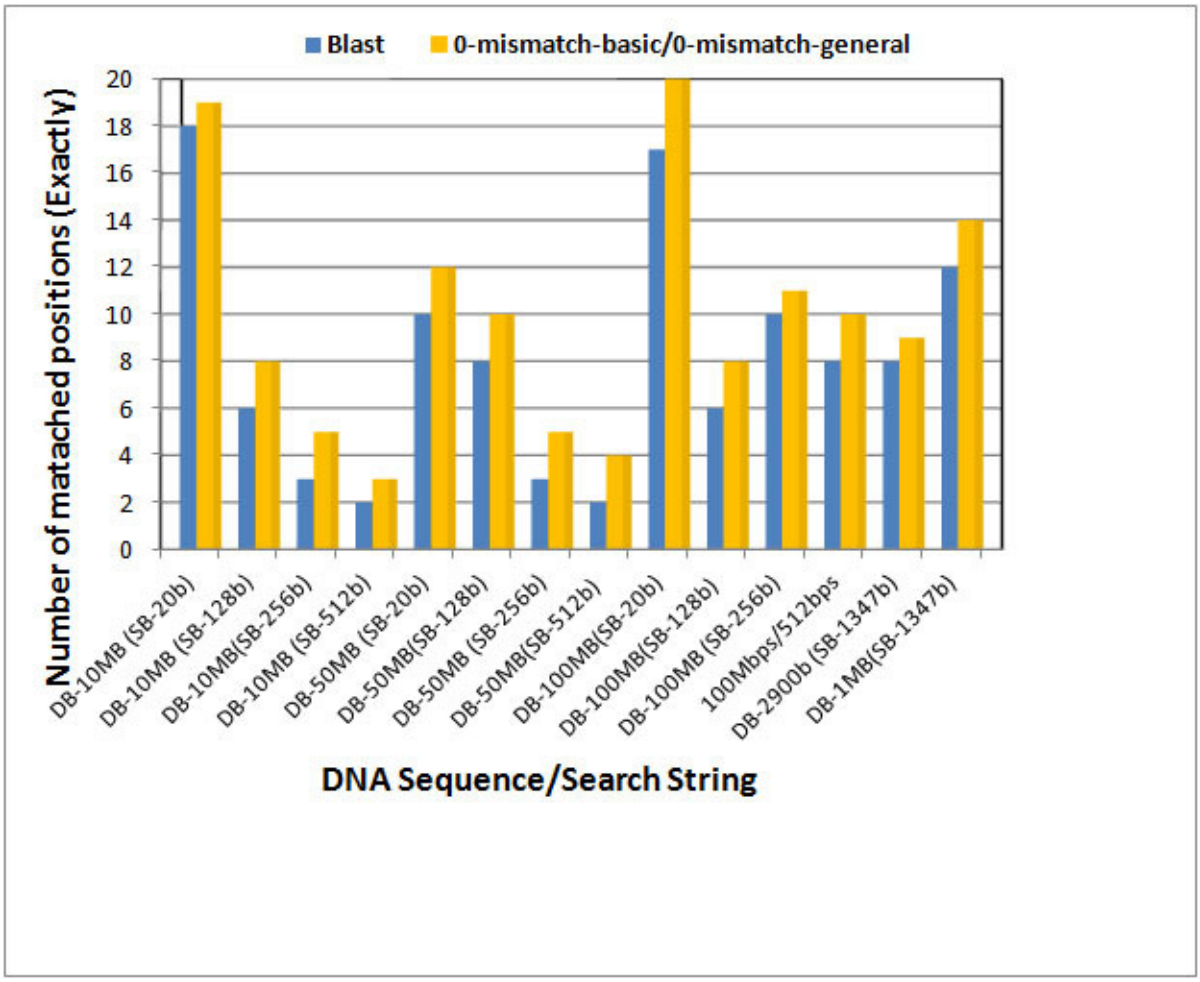


Figure 3.9. Number of search-strings reported in different DNA sequences using exact search procedure

3.5 CONCLUSION

The presented approach in this chapter uses the function-based method which compactly stores the biological data. The methods generate and store DNA sequences and target to enhance the future search techniques in these sequences. This chapter also provides novel Boolean function based exact search methods which operates on one of the data structures developed.

The experimental results measure and analyze the scalability and efficiency of various

search techniques presented in this chapter. The methods are compared with suffix tree and R-tree based search strategies. The worst time complexity for searching a suffix tree is the length of the pattern. Once the suffix tree of the sequence is constructed, it can be used for all searches. However, the suffix tree occupies a large amount of memory if the sequence is long. The experiments of the proposed approach with large DNA sequences reveal the fact that this approach is scalable and outperforms the suffix tree and R-tree based methods.

CHAPTER 4

APPROXIMATE PATTERN SEARCH

4.1 INTRODUCTION

In real world biological applications, most relevant sequences are "similar" than being exactly the same. DNA sequences frequently deformed because of evolutionary mutations as well as experimental errors. The correct sequence may have small difference than the sequence we are searching for. Thus K-mismatch queries are to be developed to complete the process of developing any search procedure.

This chapter focuses on the problem of string matching that allows errors, i.e. mismatches. The problem is also called the approximate string matching. This is a more realistic problem formulation and allows us to report all sub-strings that match the string S up to K mismatches, where K is at most M, and practically much less than M. The number of allowed mismatches is typically given as a part of the input and is denoted by input variable, K. This chapter presents a function-based approach to the approximate string search problem known as K-mismatch. The statement of problem is: given a short substring S of length M, and a long text T of length N, and a maximal number of errors K, to find all positions j of the patterns in the text that matches P with at most K errors.

Example 1: Let's consider the DNA sequence $D =$ **AG**$\underline{\textbf{CG}}\underline{CG}$**AGCG** and sub-string $S = AGCG$. In this case $N = 10$ and $M = 4$, respectively. There are two sub-strings in the DNA that match S. They are highlighted in bold font, and their positions are 1 and 7 respectively.

In *Example 1*, there are three sub-strings for $K = 1$. The third sub-string allows for one mismatch that starts at position 3 and is underlined. This definition of approximate matching is also referred to as the Hamming distance-based approximate matching.

There is another metric called the edit distance that allows minimum number of edits to transform one string into the other, with the allowable edit operations being insertion,

deletion, or substitution of a single character. The number of the mismatch within the sub-string may be defined as the mismatch score of the sub-string. In Example 1 the sub-string $CGCG$ which appears at base position 3 needs 1 edit to transform to the sub-string S.

The algorithms developed and presented in this chapter return all the matching strings and find the position of the strings in the DNA sequence which have size exactly $|S|$ bases and differ to the input string S by up to K bases.

4.2 BACKGROUND AND RELATED WORK

There are several versions of the methods exist for approximate search procedure, where the pattern is pre-processed but the text is not pre-processed [37], [4], [9]. Their complexity varies from $O((K + LogM)N/M)$ to $O(MN)$. When the search operators are operating directly on the database they are called online version. Such approaches are slow when they are operating on very large strings. Offline versions are more suitable for large string searches where typically a data structure is built prior to any string searches. Examples of offline methods can be found in [47, 22, 23, 30, 7] among others.

Many of the indexing methods used for approximate search process [29] are based on suffix trees. A suffix tree is a trie data structure built over all the suffixes of S. At the leaf nodes the pointers to the suffixes are stored. Each leaf represents a suffix and each internal node represents a unique substring of S. Every substring of S can be found by traversing a path from the root. Each node representing the substring ax has a suffix link that leads to the node representing the substring x. These suffix-tree-based methods construct a suffix tree on data and query string sequences. From the constructed suffix trees the search method is to traverse down and is to find a common ancestor node of both search string and data sequences. Search method is to traverse the sub-tree of the node found until k mismatches are encountered. The time complexity for approximate matching for most efficient suffix trees is $O(M + N + k)$ and uses $O(M + N)$ space. The

time complexity for approximate matching for most efficient suffix trees data structures, is $O(M+N+k)$ and uses $O(M+N)$ space [29]. Although suffix arrays data structures are more space-preserving than suffix trees, they still require large memory space for indexing massive data such as the whole genome of a human. The space consumption of an ordinary suffix array for a given DNA sequence of N characters is $O(NlogN)$. The occurrences of a substring of length M characters can be found in $O(MlogN)$ time by searching the pattern on the suffix array by binary search procedure. There have been several attempts to reduce the space of the suffix trees or arrays [6, 3, 48]. They built a data structure for the index. Then to answer queries, they need both text as well as the index, hence extra space is still required. The method developed by [43] uses a suffix tree. This method only indexes the beginning of blocks produced by Ziv-lempel compression. The general idea of Ziv-lempel compression is to replace sub-strings in the text with a pointer to a previous occurrence of them. This type of compression algorithm is based on a dictionary of blocks. As each new block computed it is added to the dictionary, two data structures (LZtrie and RevTrie) are created using blocks of text. Then, pattern search is done by finding the occurrence in a single block or in two blocks or more than two blocks. As described in chapter 2, Section 3.2 of this dissertation, many existing methods reduce the search space using matching statistics for approximate search procedure. The detail explanation of such search engines BLAST [39] and FASTA [40] are described in chapter 2, section 3.2. Another type of indexing method called weighted indexing method is also described in this section for exact search procedure. The weighted indexing method proposed by [13] use R-tree based data structure for approximate search procedure also. The detail explanation of such procedure also explained in chapter 2, section 3.2. The space and time performance of proposed method in this chapter is compared to above methods [43, 39, 13]

All of the proposed search algorithms in this dissertation chapter use very efficient built-in BDD library functions. BDDs enable us to manipulate a very large scale data set

in a small space and within a short time. The search time is proportional to the BDD size (number of nodes in the graph), rather than the number of bases in the DNA sequence. The set of positions in the DNA sequence that match the input string, will be reported as a Boolean function. The number of appearances are reported implicitly by a simple manipulation of the output function. The approach can be modified to report number of appearances in pre-specified ranges of a DNA sequence.

4.3 FUNCTION GENERATION AND STRING SEARCH FOR APPROXIMATE PATTERN SEARCH

Algorithm 7 describes the procedure of generating an approximate string function and subsequent string search. The input to this algorithm is the DNA function, F constructed from the DNA sequence from Algorithm 2 in Section 3.3.1 and the string function F_S constructed from the search string using Algorithm 4 in Section 3.3.3. Let F_{as} be the approximate string function which consists of all the *cube* (described in Chapter 2) that are at a distance K from the string function F_S. Algorithm 7 efficiently uses the built-in cudd function $Cudd\_bddclosestCube$ (described in Chapter 2) to find the approximate match of the search function F_S. This algorithm maintains function $Cube\_Func$ (see also line 4) which consists of all the minterms that are at distance K from F_S. These minterms represent the first set of matching strings. This function is generated by using built-in procedure, $Cudd\_bddclosestCube$(See also line 4). Then in line 2, $Cube\_Func$ is excluded from the DNA function using built-in BDD operators and saved as approximate string function (F_{as}). Steps 3-6 are repeated as shown by the while loop in line 5, in order to collect all the cubes with Hamming distance K from F_S. These cubes represent strings with K mismatches. Then the function is co factored to extract the position, the same way it was done in exact matching using procedure $Cudd\_bddCofactor$. This is shown in step 10 through 12. Finally, all the minterms of position function P are printed using $Cudd\_bddPrintminterm$ which represents the matching strings positions (See also line

12-13).

Algorithm 7: K-mismatch(F, F_S,K) with $K > 0$

Input: DNA Function(F) from Algorithm 9 and number of mismatching base K

Input: Search-string Function F_S from Algorithm 4

Output: Number of approximately matched sub-strings and matching positions

1 **Initialize** Approximate String Function $F_{as} = 0$; **Initialize** Cube_Func;

2 **while** *Cube_Func is not null* **do**

3 $F_{tempdna} = F$;

4 *Cube_Func* = *Cudd_bddclosestCube*(F, Fs, K);

5 F = *Cudd_bddAnd*($F_{tempdna}$, *Cube_Func'*);

6 F_{as} =*Cudd_bddOr*(F_{as}, Cube_Func);

7 **end**

8 **Generate**T($m_0 m_1 ... m_{2L-1}$ $b_0 ... b_{k-1}$)=*Cudd_bddAnd*(F, F_{as});

9 **for** ***each*** *i = 0 until 2L-1* **do**

10 P = *Cudd_bddCofactor(T, m_i)* ; / * Built-in BDD operator

11 **end**

12 Find Positions: *Cudd_PrintMinterm(P)* / * Built-in BDD operator

13 Find Number of occurrences:*Cudd_CountMinterm(P)* / * Built-in BDD operator

In the BDD of the DNA function at each level (described in Chapter 2), the distance K is set at the largest distance value. In order to find a large *cube* at minimum distance, we try to find a *cube* recursively using negative co-factor that has the same or smaller distance than a *cube* that has been found recursively in the earlier steps when using positive co-factors. We examine all four pairs of co-factors (positive and negative)of

functions f and g when both f and g depend on the top variable. In our case f is DNA function F and g is string function F_s. However, when g does not depend on top variable i.e., $g_t = g_e = g$, then Hamming distance $H(f, g)$ = min{H(f_t,g_t), H(f_e,g_e), H(f_t,g_e)+1, H(f_e,g_t)+1)} = min{H(f_t,g_t), H(f_e,g_e)}. Therefore in this case, we have two pairs of co-factors to compare.

The following example illustrates Algorithm 7. Consider the DNA function $F = a'b'c + a'bc' + ab'c' + ab'c + abc'$ and String function $F_s = ab'c$ and $K = 2$.

Figures 4.1 and 4.2 demonstrate the recursion procedure of *Cudd_bddclosestCube*. For simplicity in demonstration it is shown how to derive the first cube of function that is at distance K and one of the four way comparison of four co-factors. In Figure 4.1, BDD of a DNA function F and string function F_S is presented. Figure 4.2 represents the steps to find the Cube function that is at a distance K from the search function F_S.

The minimum Hamming distance between the two functions is minimum of the comparison of four co-factors from the next level. Hence when positive co-factor of the DNA function is compared with the positive co-factor of string function, the distance d is $0 + d_1$, where d_1 is the minimum distance from the next level as shown in Figure 4.2. In the next level which is shown in step 1, the distance $d1$ is calculated to be $1 + d_2$ where d_2 is the minimum distance from the next level. In the next level that is shown in step 2, the minimum distance is 1 (after comparing all the co-factors). Hence, now $d = 2$ and corresponding cube is abc'. As $K = 2$ the cube abc' corresponding to distance $d = 2$ is separated and returned. It can be seen that abc' is at a distance 2 from the search function $ab'c$.

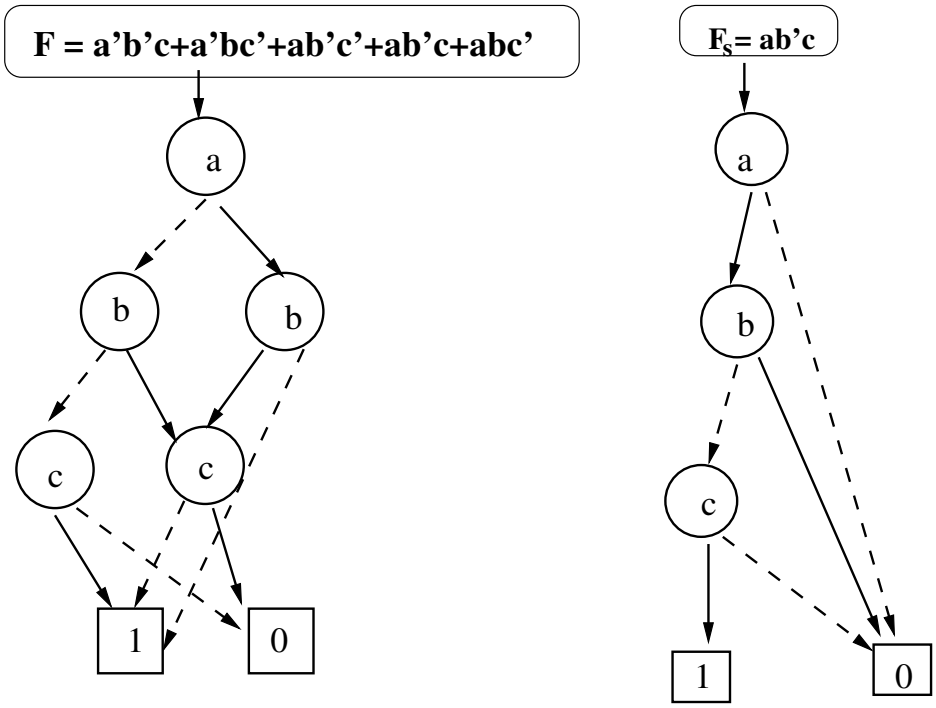


Figure 4.1. BDD of a DNA function F and String function F_S

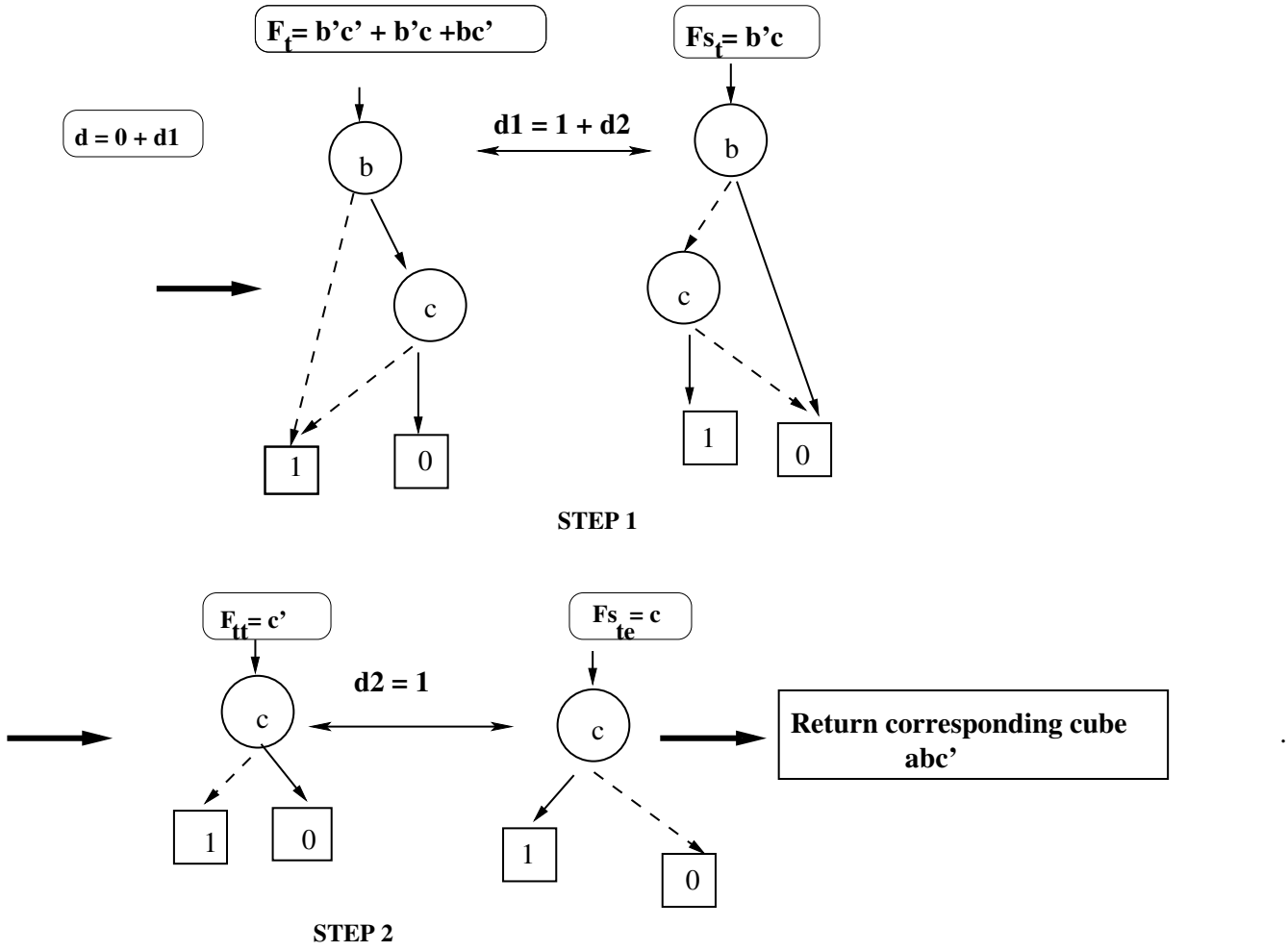


Figure 4.2. Comparison between Co-factors of Function F and F_S

The time complexity of Algorithm 7 mainly depends on the procedure, *Cudd_bddclosestCube*. *Cudd_bddclosestCube* function works similar to *Cudd_bddAnd* by traversing and comparing nodes of DNA function and search function. The depth of recursion depends on the distance d (mismatch factor K in the algorithm). Hence the over all time complexity of this procedure is $O(KN_nM_m)$, where N_n is the number of nodes in the BDD of the DNA function and M_m is the number of nodes in the BDD of search function. The time complexity is shown to be linear to the number of nodes present in the BDD graph of the DNA function.

In order to reduce the time due to iteration in the while loop of Algorithm 7 (see line 3 to 8), we tried to generate another function by modifying the build-in procedure *Cudd_bddclosestCube*. The new procedure *All_Bddclosestcube* returns a function which contains the entire cube that is at a Hamming distance K from the minterms of search string function F_s. Instead of an external loop in Algorithm 7 (See also line 3-8) an internal loop is placed in the built-in function *Cudd_bddclosestCube* and the function was named as *All_Bddclosestcube*. By using this function, all the possible approximate matching strings are found at one time. In this way the matching cube is not excluded from the DNA function each time, hence avoids the iteration. However we observed that the time difference is minimal hence the results due to modification were excluded from the experimental section.

4.4 A GENERAL ALGORITHM FOR APPROXIMATE STRING SEARCH

In order to handle very large search-strings, Algorithm K-mismatch(F,F_S, K) is modified and is called K-mismatch (F,F_S)-general (see also Algorithm 8). In this approach when the input search-string exceeds a certain length called threshold length S_{th}, it is split into multiple strings of roughly equal size similar to exact search procedure to handle very large string searches. Then multiple search functions are created one per sub-string from them. The value of S_{th} is determined and its value has a significant impact in the

performance of the approach. Our experimental results in Section 5.5 show that value of S_{th} is around 50 to optimize the performance of the approach. The minterms of the DNA function are the same size as the minterms of each of these string functions. The overall search time is calculated by adding the search time for each of these string function. By proceeding this way, the memory requirement for storing the output function of each AND operation is significantly less and therefore the overall search process is significantly faster than K-mismatch (F,F_S).

Algorithm 8: K-mismatch (F ,F_S)-general

Input: DNA Function(F) from Algorithm 9 and number of mismatching base K

Input: Search string Function F_S from Algorithm 4

Output: number of occurrences of substring S in DNA sequence$D$ and positions of matching sub-strings In the DNA sequence D

1 partition search strings to j sub-strings

2 **set** $total\_matches = 0$

3 **for each j**

4 **do** K-mismatch(F, F_S,K)

5 **return** K_j

6 /*K_j is the number of mismatches for jth substring */

7 $total\_mismatches = total\_mismatches + K_j$

8 **if** $total\_mismatches \leq$ K

9 Pj =cuddBdd_Or(pi);// Generate position function Pj

10 **Report Positions (** P_j **,** P_{j+1}**)**

11 **else**

12 $j = j + 1$;

13 Continue;

14 **Report number of positions matched for the given K**

Explanation: The idea of this algorithm is to split the pattern in to j pieces, search each piece in the DNA sequence BDD allowing K errors; provided that each search string piece is greater or equal to K. Then extend the approximate matches to complete the search and find the occurrences.

Step 1: The very first step is two split the search string into two disjoint sub-strings of size S_t (the optimum search string length). Then form the function from the DNA sequence of minterm of size of the same length as the sub-strings of the search string. For each of the sub-strings form the substring function and hence BDDs.

Step 2: The next step is do an approximate search for each of the substring functions using BDD *closest_cube* function. Then find the matched positions for the given K value (by checking total mismatches (see also steps 7 to 9 in Algorithm 8) and form a position function. Do the same for the next substring. If the position functions reported by the search of next substring is a shifted version of the position functions of previous substring then report the position else continue until done. Function Report Positions(see step 10) in the algorithm checks this shifting procedure and reports the positions.

The following example describes the search process using each step of the algorithm.

Example 1: Let us consider a DNA sequence D = AGCTAGCT and search string S= AGTT with a K = 1. For K = 1 there should be two matching positions, position 1 and 5.

Step 1: Let's say the optimum length is 2, hence the search string AGTT is split in to two sub strings AG and TT of length equal to 2 (see line 1 of Algorithm 1).

Step 2: Do the search of sub-string AG for K = 1 using *Bddclosest_cube*. For position 1 there is a match for K=0, there are no other matches until position 5. Then there is a match at position 5 for K =0. Similarly do a search for substring TT for K= 1. For position 3 there is a match for K= 1 and there is match at position 4 for K = 1. There is also match at position 7 for K =1.

Step 3: Next step is to find out if positions in the second group are the 2 shifted version of first set.For total K value less than equal to K find if the position function in 2nd set is a sifted version of first one. In this case position 4 is two shifted of position 1 and position 7 is two shifted of position 5. Hence the approximately matched stings are in the position 1 and 4 for K = 1.

4.5 EXPERIMENTAL RESULTS

The experimental conditions for approximate search procedure are same as the experimental conditions as in Chapter 3. The experiments were performed on large DNA sequences of standard reference sequences (in FASTA format) downloaded from NCBI [42] databases and on benchmarks associated with the BLAST [39] tool ranging from (10 to 100 Mbps). Large reference sequences were chosen to evaluate the performance in large data sets. The DNA sequences downloaded from NCBI [42] databases have their identification number (GI Numbers) and are stored in a separate file. The sizes of different DNA sequences are reported in respective tables. The search process of several sub-strings of different lengths were done on these DNA sequences for approximate search and allowing K mismatches. Search process was also performed using search string that was associated with local BLAST tool and is identified as fragments GI:6164828 and GI: 308801197.

The developed algorithms have been implemented in the C language. Experiments were performed on UBUNTU platform having 4GB RAM and with a speed of 3 GHz. This procedure was repeated for several sub-strings. The proposed string search methods were compared with method [43] in terms of time and space performance. The code for the method proposed in [43] was downloaded from the site provided by the paper and implemented on the workstation that we used for implementing our algorithms. The proposed string search methods were also compared with BSI and WSI methods [13] in terms of time performance. Since the code could not be obtained from the authors [13], the experiments of the proposed method were performed in the workstation having the same configuration as reported in the methods [13].

The performance is compared to local BLAST. For a fair comparison, as explained in Section 4.1, BLAST was loaded to the same workstation, where proposed approach was evaluated. BLAST search time is not always consistent even if the string size is same. Therefore the average time of several experiments was taken for a particular search string

size.

The developed methods K-mismatch (F, F_S, K),and K-mismatch(F, F_S,K)-general were evaluated for a wide range of search string (query) length and their search time for approximate search procedures. Experiments were conducted for 20 search strings each and average was calculated for each search string size (20, 128,256 etc.). These search-strings are randomly chosen and modified to fit the same size as the existing methods [43] [13] [39]. These search-strings were derived from the sequences of NCBI databases [42].

While comparing to BLAST, we listed the sequences provided with the tool. In the proposed approximate search procedure (K-mismatch (F, F_S,K) and (K-mismatch (F, F_S,K)-general), the experiments were performed to record the search time for various K values. The value of K is varied from 0 (for exact search) to 15. Then the experiments were conducted by recording various search time with the size of the DNA sequence.

A comparative study on time performance of k-mismatch (F, F_S K) with K $\geq$ 0 and K-mismatch (F, F_S)-general procedures vs LZ-index and BLAST [39] is provided in Table 4.1. A 100Mbps DNA sequence was used to compare with BLAST [39] and disk based method of [43] in Table 4.1. Column 1 of Table 4.1 provides the search-strings of various lengths and column 2 provides the time to search in second by proposed method. It can be seen that the search time for these multiple patterns is less than the search time of these patterns, when the search is done sequentially. The search time for LZ-index method is provided in Column 4. Column 5 provides the search time for BLAST method. It can be noted that the search time for the proposed method K-mismatch (F, F_S)with K $\geq$ 0 and K-mismatch (F, F_S)-general is significantly less than all the existing methods [43], [39] for any of the DNA sequence size. This is due to the fact that the search time is linear to the number of nodes in the BDD of the DNA sequence, which is much less than the number of bases in the sequence. The performance of proposed method remains faster than other two methods. For longer strings the increase in time is

almost linear. It may be noted that the method [13] can not handle search string size above 1280 bases. Hence the search time is not available for such search string sizes listed above in Table 4.1. The number of nodes in the BDD of the resulting function reduces significantly by using partition method. The Performance is enhanced because of the reduction in storage space.

The comparison of the number of positions for approximately matched strings for both BLAST and developed method ($K - mismatch(F, Fs, K)$) in Section 4.3) are provided in Figure 4.3. The X-axis lists the DNA sequences of various sizes from [42] and the input search-strings used for the search. The Y-axis represents the number of positions that matched the input string approximately. We found out that the percentage of mismatched bases by local BLAST was on an average 10 %. Then, we set K value to be 10 % of the number of bases of search string. These results show that BLAST misses some of the positions as in exact search procedure. It has been seen that for longer DNA sequence size the number of matched position missed by BLAST is more.

Table 4.1. CPU time performance of proposed method vs [43] [39] in the 100Mbps DNA sequence from [42]

Search-string size (in base)	Proposed Method sec	LZ-index [43] sec	BLAST [39] sec
20	0.010	0.110	0.210
30	0.015	0.118	0.220
40	0.022	0.219	0.350
60	0.565 (0.055 [a])	0.35	0.510
128	1.015 (0.945 [a])	N/A	4.000
256	2.018 [a]	N/A	5 .000
512	3.58 [a]	N/A	7 .000
1028	4.01 [a]	N/A	8 .000
2056	6.050 [a]	N/A	9.000
4112	7.1200 [a]	N/A	10.000
8224	8.00 [a]	N/A	12.000
fragment-1(1347 bps)	5.11 [a]	N/A	7.000

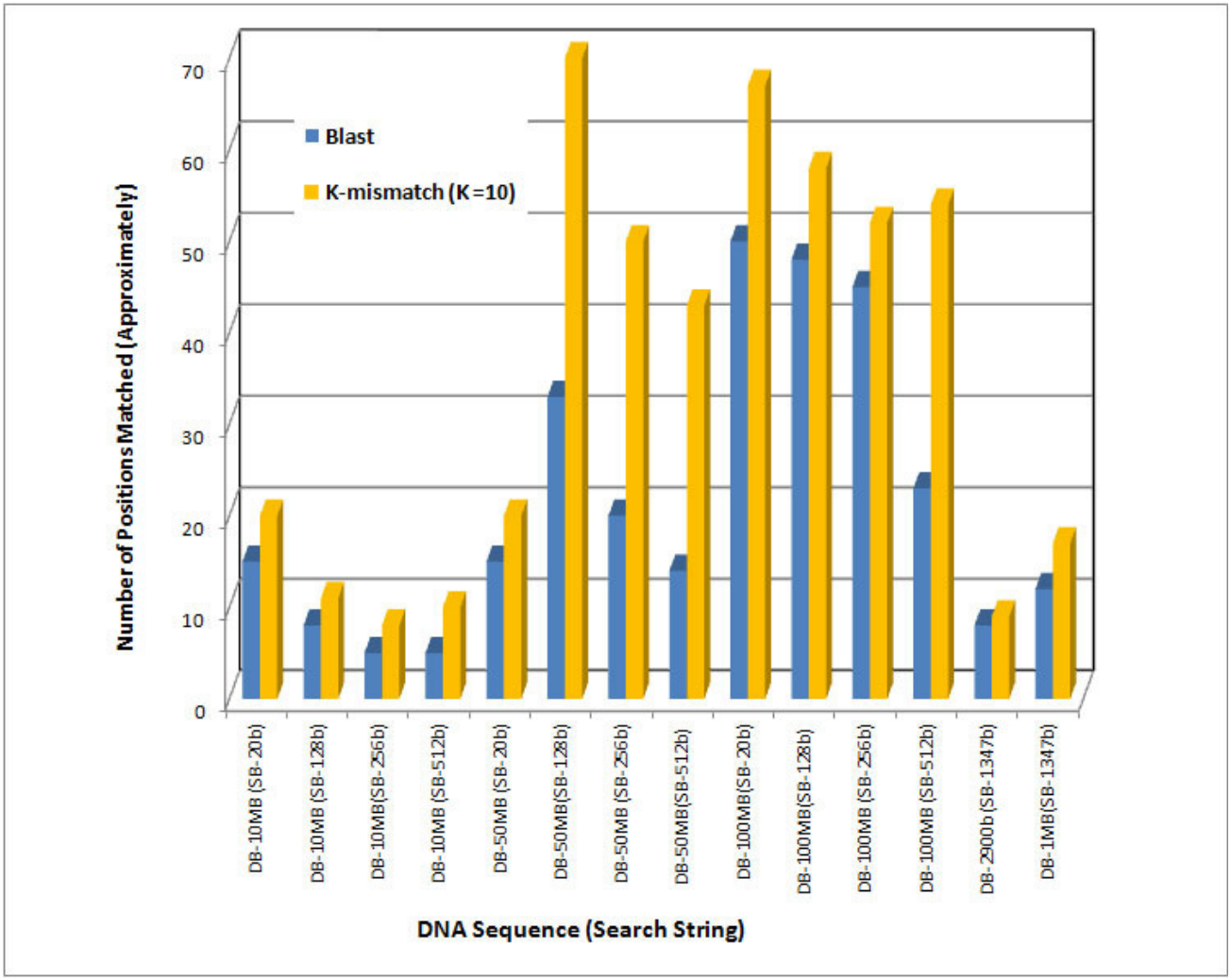


Figure 4.3. Comparison of number of matched positions in different DNA sequences using approximate search procedure

Finally, a variation of the developed approach ($K - mismatch(F, Fs, K)$) with different error rates has been studied. In Figure 4.4, the search times for different K values which varies from 1% to 15 % of search string size are provided and compared with [13]. The DNA sequence used is of length 10Mbps from [42] for a search string size 100. Again the sequence size was chosen to compare with method [13] as Sequence size longer than 10Mbps was not reported by [13]. The X-axis represents various K (from 1% to 15%). The Y-axis represents the time taken to do the approximate search in second.

It has been found out that with longer search string and higher error rate (K value), the search time increases. However it is significantly less than the existing BSI and WSI method [13]. The results demonstrate that with the increasing K value the search time increases but the developed method outperforms the other existing methods in performance.

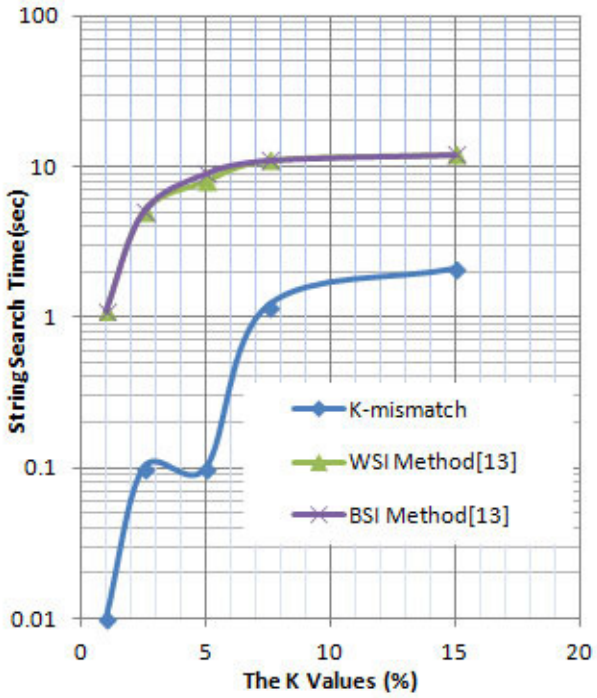


Figure 4.4. Search Time for Various K Value (on a 10 Mbps DNA sequence from [42])

4.6 CONCLUSION

Chapter 4 of this dissertation document presents novel algorithms for a more general that is approximate search process in a DNA sequence stored compactly as a function. The complexities of developed algorithms depend on the built-in procedures used in these algorithms. The developed algorithms use these efficient built-in procedures from the CUDD package which takes time proportional to the number of nodes in the BDDs of the functions. The number of nodes present in the BDDs of a DNA function is significantly

less than the actual DNA sequence size. We have compared our methods with existing data structure based methods such as BSI method and WSI methods [13] LZ-index [43] and BLAST [39] search for performance evaluation. Experiments with real biological data show that the developed method out performs any existing method. BLAST is capable of handling multiple search-strings at the same time. Our method can be modified to efficiently handle multiple strings searches by utilizing more sophisticated string search functions.

CHAPTER 5

GENOME REARRANGEMENTS

5.1 INTRODUCTION

Evolution happens due to both local and global mutations of DNA molecules. Local mutations (point mutations) take place due to substitution, insertion, or deletion of single nucleotides (A, G, C or T), while global mutations (genome rearrangements) take place due to the change of the DNA molecules on a large scale. A *chromosome* in a *genome* is a sequence of nucleotides and is represented by an ordering of certain oriented markers called *genes*. A genome which contains a single chromosome is called unichromosomal genome and one which contains multiple chromosomes is called multichromosomal genome.

Genome rearrangement is a new and important research area that studies the gene orders and the evolution of gene families. Beyond the traditional sequence-based analysis, it is possible to extract additional phylogenetic information from the gene order. Gene order data present significant mathematical challenges which are not encountered when dealing with DNA sequence data. Many evolutionary events may affect the gene order and gene content of a genome, and each of these events creates its own challenges [71]. These genome rearrangements reveal important information about the evolutionary history of genomes. With the development of fast sequencing techniques, hundreds of genomes are available to date and it becomes a challenging task to tackle the problem of reconstruction of the evolutionary history due to the genome rearrangement especially when all possible rearrangement operations are considered. A phylogenetic tree captures specification events (also called rearrangement operations) among multiple organisms. Constructing a phylogenetic tree requires inferring ancestral relationship among multiple organisms based on currently available data.

Studying genome rearrangements is an important tool that aids in the understanding of

evolutionary events. Comparative analysis of plant or insect genomes is expected to yield significant insights into evolution, development, and regulation [69]. However storing all the genomes resulting from all possible rearrangements is a difficult task.

For example the Campanulaceae Chloroplast data set has 13 genomes and 105 markers (genes). It is one of the most challenging genome rearrangement data sets studied by [70] and [71]. The variety of rearrangements in these plants exceeds any reported rearrangements in any group of land plants. Hence it makes this data set a challenging problem for any genome rearrangement study [57].

We have chosen Campanulaceae Chloroplast data set to analyze relatively low gene length chromosomes. The human-mouse-cat data consists of 193 markers, shared by all three species. We have used this data set to analyze relatively high gene sequence length.

Development of new algorithms will make genome rearrangement analysis more reliable and efficient, while potentially revealing new evolutionary patterns. In addition, the algorithms will enable a better understanding of the mechanisms and the rate of gene rearrangements in genomes, and the importance of the rearrangements in shaping the organization of genes within the genome.

With the availability of a large number of fully sequenced genomes, particularly from closely related species, there is a need for reconstructing detailed genome-wide evolutionary histories. This may help studies on diseases related to rearrangements of certain genes. For example, a list of diseases related to particular genes in human chromosomes can be found in [67]. Only a few genes out of thousands available in human chromosomes can be analyzed for gene related diseases. Some examples of data sets studied are given in [72]. As stated by Bader, ultimately this information can be used to identify microorganisms, develop better vaccines, and help researchers better understand the dynamics of microbial communities and biochemical pathways [66].

There are many types of genome rearrangements such as *reversal, transposition, trans-location, fusion, fission* etc. The next few paragraphs give general definitions of

several biological terms that are used in genome rearrangement research.

A *gene* which is a segment of a DNA sequence is represented as a positive integer or a sign integer. The genes are represented by numbers 1, 2, ..., n. The two orientations of gene i are represented by i and -i. Most comparative mapping techniques consider only the physical locations and relative order of genes in each chromosome, but ignore the gene orientations. This is due to the fact that all the sequencing methods do not provide gene orientations. It also turns out that the genome rearrangement problem (unichromosomal and multichromosomal) for unsigned permutations is NP-hard, but the same problems for signed data can be done in polynomial time. The method proposed in this chapter can handle gene orientations of the genes.

A *gene sequence* or *chromosome* is a permutation of genes over E, where E is a set of genes. Unless clearly stated, each *gene* appears exactly once in a *chromosome*. For example, a uni-chromosomal genome with $n = 5$ genes which has both the orientations can be represented as $2\ {-1}\ 3\ {-5}\ 4$.

A *genome* is a collection of *chromosomes*. A chromosome or gene sequence G is represented by a gene ordering. Let G be denoted as $g_1 g_2 ... g_n$, where $g_1, ..., g_n$ are the genes in position 1 to n. Rearrangement operations such as *transposition* work on strings of *genes* $G = g_1 ... g_n$ of length $|G| = n$.

Exchanging DNA segments (identified as a number) or genes from their respective positions in the genome is called *transposition*. Transposition $T_{i,j}$ transposes the gene from position i to j in G, That is transposition $T_{i,j}$ exchanges the gene position g_i and g_j. So $T_{i,j}\ (g_1 ... g_n) = g_1 ... g_{i-1} g_j g_{i+1} ... g_{j-1} g_i g_{j+1} .. g_n$. In the following example, a few transposition operations are provided and the final rearranged gene sequence is presented. For 4 transpositions to occur in a gene-sequence G = 5 1 2 3 4 a greedy algorithm might need 4 steps:

Initial gene sequence: 5 1 2 3 4

Sequence after $T_{1,2}$: 1 5 2 3 4

Sequence after $T_{2,3}$: 1 2 5 3 4

Sequence after $T_{3,4}$: 1 2 3 5 4

Sequence after $T_{4,5}$: 1 2 3 4 5

However the approach that is proposed in this section allows us to record gene sequences once K number T_{ij} transpositions have been applied, for any value of K and for any i, j values $1 \leq$ i, j$\geq$ n.

This is a powerful data structure that allows us to keep systematically all possible gene sequences that have evolved from the original sequence due to transposition operations.

5.2 BACKGROUND AND RELATED WORK

The traditional phylogenetic tree construction is based on analysis of individual genes. In the past, constructing a phylogenetic tree is done by comparing nucleotide sequences of a single gene or a few genes [60] and there are several representative approaches. The neighborhood joining method is a heuristic, and the method greedily merges a pair of Genomes based on the minimum evolution principle. Maximum parsimony (MP) methods find a topology with the minimum number of mutations (called the lowest parsimony score), and maximum likelihood (ML) methods attempt to find a tree with the highest likelihood value under a certain evolutionary model. The MP and ML methods are generally more accurate than the neighbor-joining method but also more computationally expensive. These methods, using nucleotide sequence data, find a reasonably accurate tree topology for close genomes.

In contrast to the individual gene comparison, there are genome rearrangement studies that are based on genome wide analysis of gene orders rather than individual gene comparison [62] [61]. Two important rearrangement operations *transposition* and *reversal* were studied in [49] and [54]. They considered reversal and transpositions as an approach for understanding the genome arrangements related to mammalian genomes. Many current evolutionary approaches to genome rearrangement are based on the study

of biological sequences by [53, 56, 58], among others. Historically, work involving evolution is done on sorting gene-sequences hence calculating reversal and transposition distance. The genome rearrangement with transposition were studied in [54].

The very first work on computing evolutionary distance (also called reversal distance) was started by Keceoglu and Sankoff [52]. The algorithm proposed by these authors is an exact algorithm which finds an optimal solution in O($m.n$)) time and O(n^2) space. Let n denote the number of genes in m genomes. They used branch and bound method to find minimum reversal distance. Since then, many related work been done to improve this exact algorithms and many approximate algorithms are also proposed. The method in [49] gave an approximation algorithm with 1.5 approximation ratio and worst case execution time O($n^{3/2}$). The first polynomial time algorithm was given by Hannenhalli and Pevzner and it runs in O(n^4) for permutation of n genes. Berman etal [73] exploited this algorithm and developed a new approximation algorithm which runs in O(n^2) time. A linear-time algorithm for computing reversal distances was proposed by [50]. The improvements due to better bounding and new search ordering (the layered method) are discussed in [51] later. All these algorithms give reversal distance as well as series of reversals. However the time complexity mentioned here are for a unichromosomal genome. It is a NP-hard problem to represent multichromosomal genome and give all the reversals. All of the traditional studies of genome rearrangements involve solving combinatorial problems to find the shortest series of reversals or transpositions to transform one genome to another. Some of these studies were done by [55] and [53].

The proposed approach here is to consider all possible transpositions to construct an evolution structure which will represent the evolution of different genomes due to all possible transpositions.

The methods developed here are capable of creating a graph structure for all possible gene sequences generated due to the evolutionary effect. The unique encoding scheme targets future efficient retrieval of the ancestral gene sequences. In evolutionary

approaches, one of the basic ways to construct a genome graph is by applying a sequence of transformations (transposition or reversal). A genome graph represents the evolution of genomes due to rearrangement operations. A detailed explanation is provided in Section 5.3.

A function-based method is also proposed to handle all possible rearrangements using *transposition* operations on the genome efficiently and store each resulting genome in a very efficient data structure called the evolution structure.

The proposed approach handles all possible *transpositions* of an initial genome. The evolved genomes are kept in groups based on their transposition distance from the initial genome. The method proposed here, transforms the gene sequences (chromosomes) to Boolean functions and stores them in a canonical data-structure. The algorithms proposed here are for the genome rearrangements based on *transposition* operations. Operations such as reversals can be interpreted using transpositions.

Experimental results show that evolved chromosomes can be rearranged and stored efficiently. Both time and space efficiency of the proposed method is analyzed.

The novelty of this approach is a function- based unique encoding scheme and strategy to construct an evolutionary structure, which is based on parallel transpositions. A function based algorithm operates on the evolutionary structure iteratively to perform all possible transpositions on the genomes at a given transposition distance. Then it forms the collection of genomes at the next transposition distance. Each transposition distance consists of several BDD functions.

This chapter is organized as follows: In Section 5.3 a function-based encoding method for an evolutionary structure is explained. Definitions are also provided and the encoding process of a genome to a gene function is explained. The algorithm which generates the evolution structure using gene transpositions is given in Section 5.4. Section 5.5 explains the experimental procedure and provides the results of the experiments to support the proposed method. In this section it is shown that many species can be transposed and

accommodated in the genome graph. The results show that the proposed method is scalable. Finally Section 5.6 concludes the chapter.

5.3 EVOLUTIONARY GENOME STRUCTURE

In the developed method, the operations are performed based on transpositions of the genes in gene sequences; hence they are position specific. However in our encoding, gene positions are not explicitly stated. Instead, the positions are the order they appear in a chromosome. When transposition or reversal occurs,the position of these genes change. An evolutionary genome graph is constructed by applying transpositions in parallel. Another novelty of this method is that the genome graph is capable of encapsulating all possible rearrangement operations due to transpositions. Each stage contains the new co shared graph from the function that is changed due to transposition in the previous stage. Given an initial genome, the *evolutionary structure* is shown in Figure 5.1. It consists of different stages. The first stage is the genome function which represents the initial genome. It is shown by one vertex G_0 in Figure 5.1. Each vertex at the next stage represents sets of genome that have evolved by a transposition T_{ij} operation, on all the genomes at the previous stage. The evolutionary structure is constructed using this process by applying all possible transpositions at each stage. Each arrow from one stage to the next represents different T_{ij} operation on all the genomes at previous stage. The *depth* of the evolutionary structure is the number of stages.

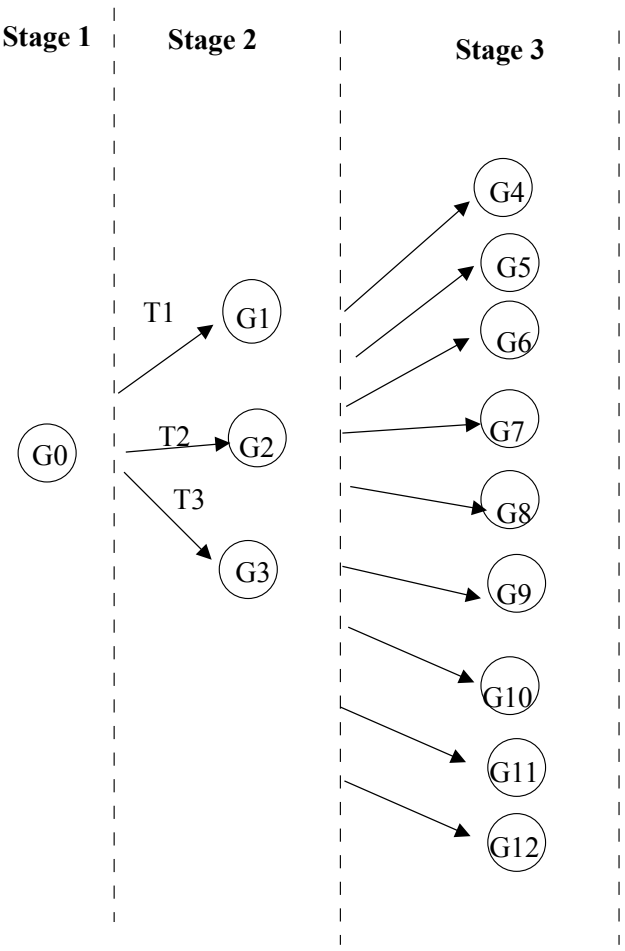


Figure 5.1. Evolution Graph of A Genome

In the proposed method, sets of genomes are represented by a Boolean function. Such genome function F_G are stored in a canonical data structure called BDD (see Chapter 2). The advantages of using a BDD is twofold. First, if properly represented, Boolean functions can be stored very efficiently in this data structure. Second,these functions can be manipulated efficiently in very little time by simply incorporating many built in base procedures in the manipulating algorithms.

In the encoding scheme proposed here, each gene sequence (which is originally represented by a sequence of numbers) corresponds to a minterm of a Boolean function. In each sequence, gene positions are according to the order they appear. For example, in

a gene sequence $G_1 = 2\ 1\ 4\ 3$, Genes 1, 2, 4 and 3 are in positions 2 , 1 ,3 and 4 respectively. For a unichromosomal genome with n=4 genes and with an orientation can be represented as $-3\ 4\ 2\ -1$. In our encoding scheme a positive literal stands for positively oriented gene and negative literal stands for negatively oriented genes.

Initially, we form a function consisting of a single minterm where all the genes appear in lexicographic order. That will be the reference function G_0 and the indexes will be the ids of the genes. Next, the function will be permuted using different Tij operation and different functions will be generated and explicitly kept as nodes at the next stage. Algorithms to either permute or to produce negatively oriented markers are minterm implicit procedures, i.e., they operate in parallel to all the minterms (in our case evolved genes) at the current stage. Detailed explanations on these procedures are given later in Section 5.4.

There are a number of BDD packages developed to store Boolean functions and provide built in logic operation procedures to manipulate these BDDs. We have used the CUDD [11] package for implementing the presented approach.

Algorithm 9 describes the procedure of generating a genome function. The number of onset minterms in the genome function at a particular stage is equal to the total number of gene sequence evolved in that stage due to Tij operations, for given i and j, to all the genes at the previous stage. This process avoids gene enumeration at the previous stage and transposes all these genes in parallel. That is, for transpositions of all the genes of positions i with all the genes of position j of all the sequences of a particular stage is performed at one time. Different stages are represented by stage numbers.

Let S is the number of stages constructed. Stage bits which are created using the log_2S equation, are added at the beginning of the minterms. These identify the stage that a particular genome belongs to and its count provides the depth of the evolved graph structure.

At each stage, we use tag bits to distinguished genes that have evolved from the previous

stage using some T_{ij} operations. Therefore each gene in some stage of the structure consists of three parts. Its gene sequence representation, the stage part and the tag part. All three parts are binary cubes and therefore the gene is a minterm of a single boolean function for the whole stage. Evolved genes(minterms) by some T_{ij} can be easily identified by their distinct tag value. For this reason, our genome structure represents them by distinct nodes.

5.4 ALGORITHMS FOR REARRANGEMENTS DUE TO TRANSPOSITIONS

Algorithm 9 (Genome-Function) presents the procedure of generating the initial genome function from a gene sequence. The input to Algorithm Genome-Function is a given gene sequence of a particular genome represented by numbers as mentioned in the previous section. The encoding scheme is as follows:

Algorithm 9: Initial Genome-Function

Input: Gene-sequence(G)

Output: initial Genome-Function F_G

1 **Create** n level fields each of length $log\ n$

2 M_i= **Generate Initial Minterm**

3 F_G = **Generate Initial Genome-Function**

4 **Report** F_G

In Algorithm Genome-Function, the first step is to calculate minterm bits to create the minterms of Genome-function. In line 1, variable bits are created to represent the stages of the genome function, to tag the transpositions and to encode the *genes* in a minterm of the genome function(see also line 2 in Algorithm 9). Tag variables that emulate function F_{ij} could be kept as external pointer. A minterm is generated from these variables to

generate the genome function. Lines 2 to 4 of the algorithm depict this procedure. As stated in line 3 of this algorithm, function "Generate Initial Minterms" takes the variables created and construct the single minterm of the Genome-function. A simple Cuddbdd_And operator is used on the variables in this function to generate the minterm. Finally, function "Generate Initial Genome-Function" takes the minterm created and generates the Genome-function. This operator is recursive and operates in linear time to each of its operands.

Algorithm 10: Operation Transpose (K)

Input: Initial Genome-Function F_G, Stage K

1 **Generate** initial Genome-Function F_G^K;

2 **For each** position pair P_{ij} (P_{ij}=1 to $\binom{n}{2}$) in Genome Graph F_G

3 $F_{G_{newtemp}}$ = SWAP (g_i,g_j, F_G^K); /* swap the order of all genes in position i and j

4 **Generate** tag cube for i,j and set the ids to all minterms;

5 $F_{Gnew} = F_G + F_{G_{newtemp}}$;

6 **End For**

7 **Tag** (SWAP);

8 **Update**(stage); /* updating Stage variables

9 **End For**

Algorithm 10 describes the procedure for transposition operation of two genes between positions i and j in all the gene sequences in a given stage. The first stage consists of only one genome; however the number of gene sequence grows exponentially in subsequent stages. The problem is to transpose gene g_i of position i with the gene g_j of position j for all the gene sequences at a given stage. The input to this algorithm is BDD of genome function F_G. At step 1, we generate the function that corresponds to all the genome sequences at a given stage k. These are precisely those minterms whose stage tag part

Algorithm 11: SWAP (g_i, g_j)

Input: g_i, g_j

1 int n; **For each** (l = 0; l *le* n; ++l)

2 bddX[n] = g_i

3 bddY[n] = g_j;

4 F_G = Cudd_bddSwapVariables(Bddmanager, F, bddX, bddY, n);

5 **End For** Return F_G;

matches k. Since we are considering all possible transpositions, position pair are not necessary as an input. They are only needed if few specific transpositions are to be done. Step 1 is generating the BDD of F_G^k, i.e, the function that consists of all gene sequences to be transposed. Step 2 is to modify all the genes in F_G^k using T_{ij} operations. These are done in $\binom{n}{2}$ iterations. Each iteration is a call to procedure SWAP. Each call to SWAP generates distinct functions. In the next stage which are co shared to form one BDD. The algorithm for sub-function **SWAP** is provided in Algorithm 11. This is a built- in BDD operation that simply exchanges the level of variables at stage i and j in function F_G^k. Let F^1 is a function in the initial stage. In order to form a function F_{ij}^2 we do a single swap between two positions i and j over function F^1 for the stage 2. This method benefits from stage 3 and onward. In particular for any function F_{ij}^k 1 $\leq$ i, j$\leq$ n and n$\geq$ k$\geq$ 3 we only need to do a single swap for all F_{ij}^{k-1} functions that we co shared in the BDD. Because of this single operation, the proposed algorithm requires O($n.n^2$),i.e. O(n^3) swap operations which is a polynomial time algorithm in terms of swap operation. A brute force approach would have required O($\binom{n}{2}^2$) operations which is an exponential quantity. Detail description of Algorithm 9 is given in Example 1.

Example 1: Let us consider an initial gene sequence or chromosome $G1 = 123$. At stage 1 (the very first stage) this will be encoded as a Genome-function F_{G1} of a single minterm

whose bits are encoded from this gene sequence $G1$. Figure 5.2 shows the initial BDD of the Genome-function at stage 1 and all the BDDs of the distinct functions generated after transpositions. Figure 5.3 represents the BDD of the co shared functions generated at stage 2.

Step 1 (Operation Transpose 1): The very first step of constructing the boolean function from the gene sequence is to calculate the total number of bits required to construct its minterms. Assume that we want to construct up to 3 stages. Then total number of stage bits required is: $log3 = 2$. Hence stage bits are: **00, 01, 10**. Similarly, Tag variables are also created to tag each swap. since there are 3 genes present in the sequence, for tagging purpose, we need $log3 = 2$ variables and number of gene bits required for each gene is $log3 = 2$. The initial Genome-function F_{G1} is encoded as 0000**000110**. In this minterm, the first two bits 00 represent the stage, that the sequence belongs to. The third and fourth bits are also set to 00 because no T_{ij} has been applied to the initial function. In this representation genes 1 to 3 appear in an order of 1 to 3, hence gene 1 is in position 1, gene 2 is in position 2 , gene 3 is in position 3. In general, all gene sequences kept in a BDD will correspond to minterms where the last six variables are 00,01 and 10 to represent gene variables at position(level) l_0, l_1 and l_2. This is shown in Step 2. Figure 5.2a shows the BDD of initial function.

Step 2(Operation Transpose 2): All possible transpositions will generate the gene sequences 213,132,312. In this case there are only $\binom{3}{2} = 3$ different transpositions, each resulting , to a different gene sequence. Figures 5.2a, 5.2b and 5.2c show how the BDD of each function looks like. Each function will have different variable order. These tad fields are set appropriately. Then they are co shared as shown in Figure 5.3 to form the compact BDD at the next stage.

Now the new function in stage 2 is $F_{G2} = 0100\mathbf{010010} + 0101\mathbf{111} + 0110\mathbf{111}$. Note that the stage variables which represent the stage of the gene sequence is also updated, represented by very first two bits (01). Observe that gene variables at different levels of

all the minterms in the function are denoted by bold faced letters in the above expression.

Step 3(Operation Transpose 3): In this case $\binom{3}{2}$ T_{ij} will be performed. Each one will operate on all the three minterms (gene sequences)of the function in Figure5.3. Each T_{ij} will generate minterms (gene sequence) in a way similar to the illustration in Step 2. That way the three T_{ij} will generate a function that contains a minterm.

Step 4(Operation Transpose 4): $\binom{3}{2}$ T_{ij} operations will result into all the minterms at the end of this stage. In this example, the number of gene sequences at the end of the step k is 3^k. However, only $\binom{3}{2} = 3$ SWAP operations are required per stage. The number of stages is O(4), i.e Algorithm 10 is called upto 4 times. Algorithm 10 has a loop at line 1. It executes $\binom{n}{2}$ times. Hence we have $\binom{n}{2}$ SWAP operations. This is O(n^3) SWAP operations. Each SWAP takes time linear to the number of BDD nodes of the Function. This is explained below.

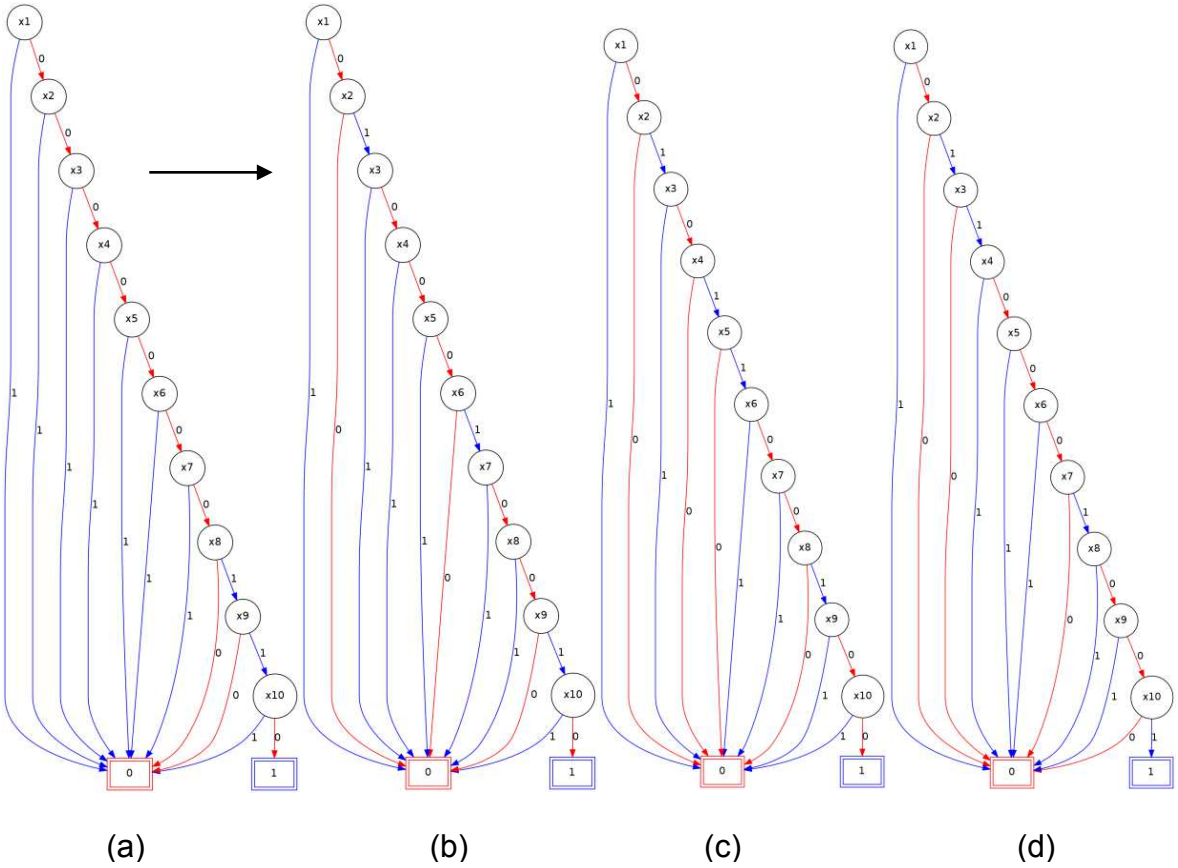


Figure 5.2. BDD of an Initial Genome and BDDs generated after transpositions at Stage 1

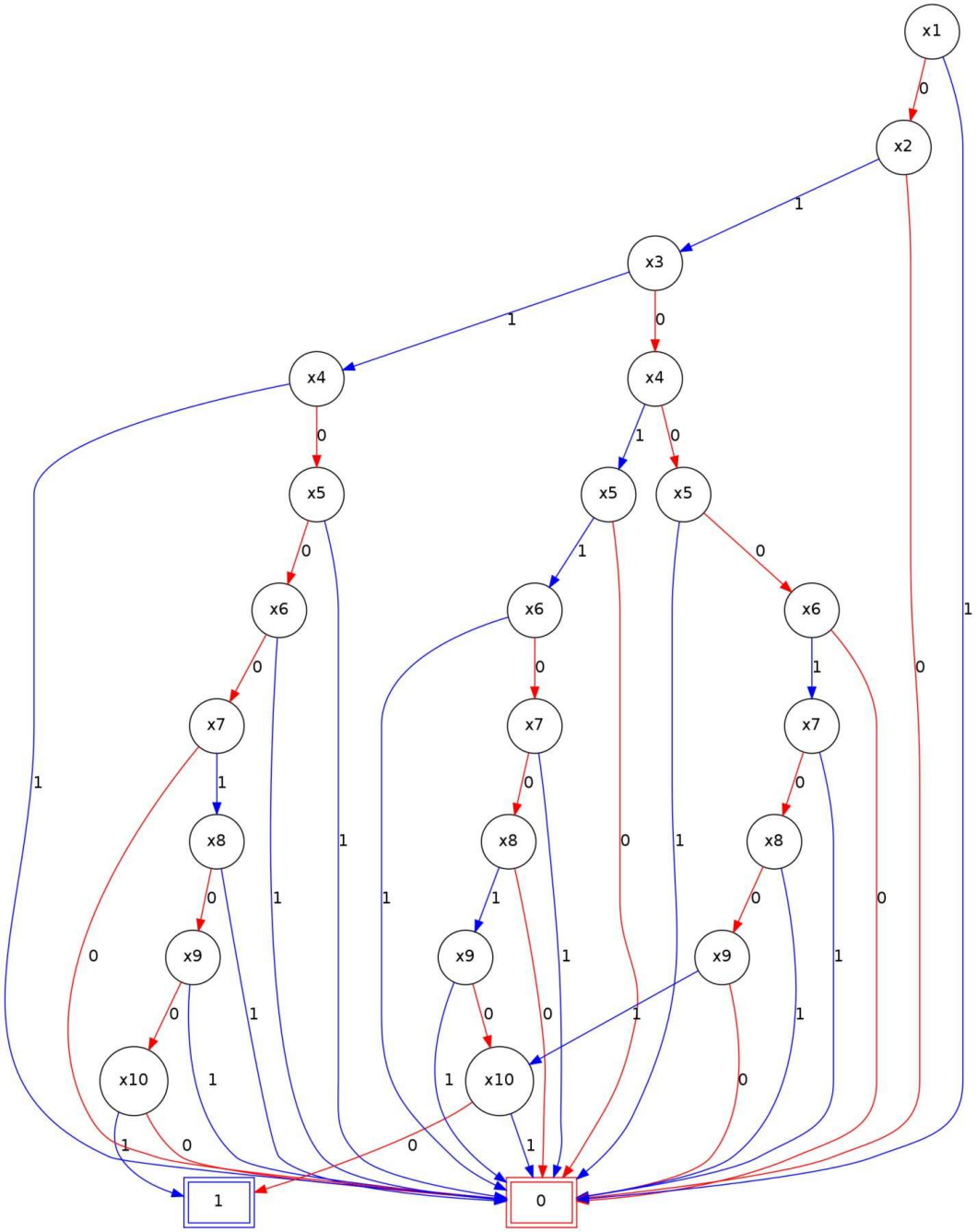


Figure 5.3. BDD of shared Genome function after transpositions at Stage 2

In Algorithm 11, gene g_i in the position i is swapped with gene g_j in the position j as shown in line 2, where g_i represents all the i_{th} literals of binary representation of genome and similarly the gene g_j with respect to position j. This swap operations are performed on the genome function using built-in BDD function Cudd_swapVariables. This operation swaps two sets of variables of the same size (x and y) in the BDD function. The two sets of variables are assumed to be disjoint. It returns a pointer to the resulting BDD if successful; NULL otherwise. In this case transposed gene function is created by swapping variable positions between i_{th} and j_{th} level as shown in line 4. Swap operation is a local operation that interchanges the variable order between i_{th} and j_{th} level [65]. It changes the order of variables in a BDD by simply exchanging the positions of these two sets of variables. This may result to a change in the size of the BDD.

5.5 EXPERIMENTAL RESULTS

To evaluate the performance of proposed algorithms in this chapter, we ran a series of experiments on data sets in [59] under a wide variety of settings. We generated genome graph structures from the chromosomes of these data sets. An initial gene order of n distinct genes was assigned at the root so it can evolve down to the different stages following the natural process of evolution by carrying out all possible predefined evolutionary events, that is all possible transpositions.

The algorithms are implemented and tested using different sizes of gene sequences for all possible transpositions. Experiments are performed by downloading the test data in [59] and also performed from random gene-sequences of various sizes. The data set used for the experiments are 13-genome Campanulaceae data sets and human data sets. The time and space performance of transposition operations are reported for various gene sequences taken from the data set in [59]. There are sets of gene-sequences of length nearly equal to 20,50 up to 200 genes. For convenience the name of the gene sequence is named as Gseq then followed by the size of the sequence. For example a gene sequence of size 20 is

represented as Gseq˜20.

Since each gene sequence contains distinct genes, our approach generates exactly the same BDD for all gene sequences of m genes. A genome graph structure was generated for the given gene sequences. The algorithms for transpositions were applied to the genome structure to do the rearrangement operations implicitly and hence a very space efficient evolution structure was developed. The number of genes per chromosome may be small but the goal is to derive all the rearrangement scenarios and it allows one to get some insights into a large-scale organization of the ancestors.

The proposed algorithms have been implemented in the C++ language. Experiments were performed on a Unix platform with 6GB RAM and with a speed of 2.27 GHz. This procedure was repeated for several gene-sequences.

Time performance of the proposed algorithm for all possible transpositions of genes of different sizes (20-200) are reported in Table 5.1. In Column 1 of Table 5.1 number of gene sequences in a particular genome is provided. Time to do all possible transpositions in the first 5 stages is provided in Column 2 for various gene sequences. Column 3 lists the time taken to do the transpositions for first 10 stages in seconds. It can be seen from this result table, that a huge number of transpositions can be done in a very little time. For example, transposition operations of a gene sequence of length 100 can be done in an average of 65.57 and can go up to 10 stages, as shown in Table 5.1 and resulting gene sequences can be stored efficiently. Due to the efficiency of the algorithm many stages of the evolution tree can be constructed and stored in the form of this graph structure. For each stage, the number of transpositions remains the same. However, the number of minterms of rearranged gene function for each stage increases. Since the minterms of a gene function are stored as a BDD, they are compactly stored.

Table 5.1: Time performance of proposed algorithm for all possible transpositions

Genome	Average time to construct First 5 stages(Sec)	Average time to construct First 10 stages(Sec)
Gseq˜20	0.45	0.74
Gseq˜30	3.32	4.25
Gseq˜40	12.97	15.64
Gseq˜50	50.04	51.13
Gseq˜100	62.56	65.58
Gseq˜200	251.0	261.0

In Table 5.2, the space needed for all the transpositions to construct 10 stages for different gene sequence is provided in terms of number of nodes. The number of gene sequences which are represented as minterms of the genome function are also reported in this table to show the space performance. Column 1 of Table 5.2 lists gene sequences of different lengths. Same genes as in Table 5.1 is used. Information is repeated for 5 and 10 stages. The number of nodes for storage of all the gene sequences due to transpositions for these stages are provided in column 2 and 4 each up to 5 stages and up to 10 stages for each gene sequence respectively. Column 3 and 6 report number of minterms generated (corresponds to number of gene sequences) when all these stages have been constructed. From table 5.2, it is clear that the initial gene-sequence and all the gene-sequences due to all possible rearrangement operations can be stored efficiently.

Table 5.2: Space performance of proposed algorithm for all possible transpositions in 5 and 10 stages of evolution

genes	Stages	Number of Nodes	Number of minterms
Gseq~20	10	1460	5.49e+05
Gseq~30	10	5242	5.34e+08
Gseq~40	10	21654	5.55e+11
Gseq~50	10	48234	5.629e+14
Gseq~100	10	296688	6.529e+34
Gseq~200	10	666051	8.036e+59

In Table 5.3, maximum number of stages for different gene sequences is provided. In Column 1 the gene sequences are provided. Column 2 of Table 5.3 reports the maximum number of stages. The time needed to do all the transpositions are reported in Column 3. The space needed for all the transpositions in terms of number of nodes are reported in Column 4 and the number of minterms of the genome function which represents gene sequences are also reported in Column 5 of this table to show the space performance. Figure 5.4 presents the number of stages that can be achieved for various gene-sequence lengths. The X axis of the graph represents the gene sequences length and the Y axis represents the maximum number of stages that can be constructed for that particular gene sequence length. As the number of genes per gene sequence increases the number of stages that can be constructed, decreases. This indicates that for lower level of species which has fewer genes per sequence can cover a wide range of related species and grows upto large number of stages.

Table 5.3. Maximum number of stages constructed for various gene sequences

genes	Maximum number of stages	Time (sec)	Number of nodes	number of minterms
Gseq˜20	17	21.18	425861	5.451e+11
Gseq˜30	17	23.43	444512	6.153e+12
Gseq˜40	15	45.56	4523111	5.23e+24
Gseq˜50	12	52.66	487904	5.42e+29
Gseq˜60	11	58.05	594121	5.62e+31
Gseq˜100	10	65.58	596673	6.529e+34
Gseq˜200	10	261.0	665175	8.0351e+59

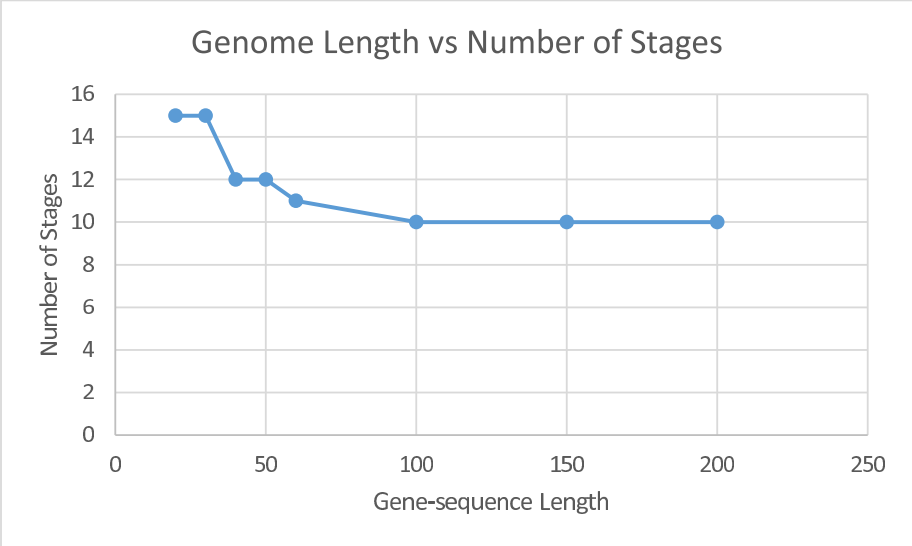


Figure 5.4. Number of stages reached for various chromosome length

5.6 CONCLUSION

A time and space efficient approach to construct a graph structure for evolution is proposed in this chapter. The presented structure accommodates all possible

rearrangements due to transposition. The rearranged genomes due to all possible transpositions are represented by a BDD. The advantage of such representation is to store all the genomes due to all possible transpositions compactly and also operators can be developed on this data structure to efficiently retrieve the history of the genomes. The transpositions operations are implemented using the built-in BDD operator in a procedure called SWAP. Experimental results demonstrate the efficiency in time and space complexity of the developed algorithms.

The compact nature of BDDs makes this data structure very space efficient. In the future, algorithms can be developed to build operators to operate on the constructed genome graph structure and can report the evolutionary history.

An operation operation orient can be developed to orient all the genes at position i. Such an operation is local and does not affect other variables in other levels. We can perform a BDD traversal until level i and then the polarity of that particular node can be changed. This again is done by co-factoring the "then" and "else" children of a particular node and then exchanging them for a new node. Hence the "then" edge of new node created now will point to the node that was pointed by "else" edge of the previous node and vice versa. This particular operation is a linear operation to the nodes of the BDD.

CHAPTER 6

CONCLUDING REMARKS FOR THE DISSERTATION

Novel scalable algorithms for biological sequence storage and retrieval are developed in this dissertation. They use binary functions to encode DNA sequences or genomes. They store and manipulate such encoded sequences using Binary Decision diagrams which are very compact. The data structures developed here for both DNA sequences and gene sequences do not store the sequences explicitly but it uses the unique encoding schemes of binary representation to generate the sequences and then store as BDDs. Hence they are very compact. The search methods do not operate directly on sequences but on a graph structure that contains the binary representation of the biological sequences. Chapter 1 presented an overall introduction of the research work of this dissertation. Chapter 2 provided a detailed background on BDDs, which are the base for the developed algorithms.

Chapter 3 presented a method for fast exact string search that relies on binary encoding and BDD representations to expedite the search process. Algorithms for encoding and storing of the DNA sequences were proposed in this chapter. The search procedures are developed to operate on one of the structures developed. Experimental results demonstrated that they are better than existing methods. They are very first and compact. The BDD-based building algorithms can be parallelized. This would further reduce the time complexity in constructing DNA functions from the DNA sequences. Implementation of a parallel version of the algorithm was also explained in Chapter 3.

Chapter 4 extended the DNA sequence algorithms to find approximate matches. This chapter presented a very efficient method for DNA pattern matching in DNA sequences. DNA sequence of different sizes have taken and are searched by taking different pattern sizes. The experimental evaluation demonstrated better performance when compared to some of the other popular algorithms, both in case of time and number of matches found.

Based on the experimental work carried out with DNA sequence data, this proposed approach was found to be scalable.

Chapter 5 presented an approach to construct an evolution structure for genomes. Molecular evolution studies are usually based on the analysis of individual genes rather than entire genomes. An alternative approach is to infer the evolutionary history of the entire genomes, rather than individual genes, based on the analysis of genome rearrangement operations. However because of the availability of huge data and the nature of rearrangement operations it is computationally intense to create a structure for the entire genome. Novel algorithms were proposed to encode and create a BDD based structure which accommodates all possible rearrangements due to transposition operations. The experimental results section provided the evidence that the approach operates in polynomial time to the BDD size and is capable of encapsulating huge number of gene sequences.

REFERENCES

[1] R. Adapa, E. Flanigan and S. Tragoudas, *Function-Based Test Generation for (Non-Robust) Path Delay Faults Using the Launch-off-Capture Scan Architecture*, ISQED, 717-722, 2007.

[2] S. J. Bedathur and J. R. Haritsa, *Search-Optimized Suffix-Tree Storage for Biological Applications*, 29-39, 2005.

[3] V. Makinen and G. Navarro, *Compressed Compact Suffix Arrays*, CPM, 420-433, 2004.

[4] G. Navarro and R. A. Baeza-Yates, *A New Indexing Method for Approximate String Matching*, CPM,163-185, 1999.

[5] G. Navarro, *NR-grep: a fast and flexible pattern-matching tool*, Softw., Pract. Exper., volume 31, number 13, 1265-1312, 2001.

[6] J. Tarhio and E. Ukkonen, *Approximate Boyer-Moore String Matching*, SIAM J. Comput., volume 22, number 2, 243-260, 1993.

[7] G. Navarro, R. A. Baeza-Yates, E. S. and J. Tarhio, *Indexing Methods for Approximate String Matching*, IEEE Data Engineering Bulletins, volume 24, number 4, 19-27, 2001.

[8] B. Soewito and N. Weng, *Methodology for Evaluating DNA Pattern Searching Algorithms on Multiprocessor*, International Conference on BioInformatics and BioEngineering, 570-577,2007.

[9] E. Sutinen and J. Tarhio, *Filtration with q-Samples in Approximate String Matching*, CPM, 50-63, 1996.

[10] R. Adapa and S. Tragoudas and M. K. Michael *Improved diagnosis using enhanced fault dominance*, Integration, volume 44, number 3, 217-228, 2011.

[11] F.SOMENZI, *Colorado University Decision Diagram Package* , http://vlsi.colorado.edu/ fabio/CUDD/, 2001.

[12] A. V. Aho, and M. J. Corasick, *Efficient string matching: an aid to bibliographic search*, Communications of the ACM, volume 18, 333 -340, 1975.

[13] W.C. Kim , S. Park , J. Won and S. Kim and J. Yoon, *An efficient DNA sequence searching method using position specific weighting scheme*, Jounal of Information Science, number 2, volume 32, pages 176-190, 2006 .

[14] S. B.Akers, *Binary Decision Diagrams*, IEEE Transanction in Computing, volume 27, 509–516, number 6, 1978.

[15] A. Amir , G. Benson and M. Farach, *Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files*, Journal of Computing Systems Sci., volume 52, 299-307, number 2, 1996.

[16] A Aziz , F. Balarin , Szu-T. Cheng , R. Hojati and T. Kam and S. C. Krishnan and R. K. Ranjan , *A BDD-Based Environment for Formal Verification*, DAC, pages 454-459, 1994.

[17] R. S. Boyer, and J. S. Moore, *A fast string searching algorithm*, Communications of the ACM, volume 20, 762-772, 1977.

[18] R. E.Bryant, *Symbolic Boolean manipulation with ordered binary-decision diagrams*, ACM Transactions on Comput. Surv., volume 24, pages 293–318, number 3, 1992.

[19] L. Chen, L. Shiyong and J. Ram , *Compressed Pattern Matching in DNA Sequences*, Proceedings of the 2004 IEEE Computational Systems Bioinformatics Conference, 62–68, 2004.

[20] L. Dudas, *Improved Pattern Matching to Find DNA Patterns*, International Conference on Automation, Quality and Testing, Robotics, volume 2, 345-349, 2006.

[21] A Garg , A. Di Cara , I. Xenarios , L. Mendoza , G. De Micheli, *Synchronous versus asynchronous modeling of gene regulatory networks*, Bioinformatics, volume = 24, 1917-1925, number 17, 2008.

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

[23] Z. Khan, J. S. Bloom , L. Kruglyak and M. Singh, *A practical algorithm for finding maximal exact matches in large sequence datasets using sparse suffix arrays*, Bioinformatics, volume 25, number 13 , 1609-1616, 2009.

[24] K. D. Morris, J. Pratt, *Fast pattern matching in strings*, IAM Journal on Computing, Vol 6(1), 323-350, 1977.

[25] M. K. Michael and S. Tragoudas, *Function-based compact test pattern generation for path delay faults*, IEEE Trans. VLSI Syst., number 8, volume 13, pages 996-1001, 2005.

[26] S. Minato, *Zero-suppressed BDDs for set manipulation in combinatorial problems*, Proceedings of the 30th international Design Automation Conference, pages 272-277, 1993.

[27] S Minato and Kimihito Ito, *Symmetric Item Set Mining Method Using Zero-suppressed BDDs and Application to Biological Data*, Transactions of the Japanese Society for Artificial Intelligence, number 2, volume 22, pages 156-164, 2007.

[28] R.S. Boyer. J. S Moore, *A Fast String Searching Algorithm,*, Communications of the Association for Computing Machinery, volume 20(10), pp. 762-772., 1977.

[29] Gonzalo Navarro, *A Guided Tour to Approximate String Matching*, ACM Computing Surveys, volume 33, 1999.

[30] Raju Bhukya and DVLN Somayajulu, *Multiple Pattern Matching Algorithm using Pair-count*, IJCSI International Journal of Computer Science Issues, volume Vol. 8, Issue 4, No 2, July 2011.

[31] S. B. Needleman, Christian D. Wunsch, *A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins*, Journal of Molecular Biology, volume 48, pages 444-453, 1970.

[32] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology, volume Vol. 215, No. 3., pages

pp. 403-410, 1990.

[33] David Sankoff, *Matching Sequences under Deletion/Insertion Constraints*, National Academy of Sciences of the USA, volume 69, pages 4-9, 1972.

[34] P. H. Sellers, *The Theory and Computation of Evolutionary Distances: Pattern Recognition*, Journal of Algorithms, volume 1, number 4, pages 359-373, 1980.

[35] S.Waterman, F.Temple and S. Michael , *"Identification of Common Molecular Subsequences"*, Journal of Molecular Biology, volume 147, pages 195197, 1981.

[36] N. Vemuri and P. Kalla and R. Tessier, *BDD-based logic synthesis for LUT-based FPGAs*, ACM Trans. Design Autom. Electr. Syst., volume 7, number 4, 501-525, 2002.

[37] S. Wu, and U. Manber, *Agrep-A Fast Approximate Pattern-Matching Tool*, Usenix Winter 1992 Technical Conference, pages 153 -162, 1992.

[38] S. Y. and G. De Micheli, *An application of zero-suppressed binary decision diagrams to clustering analysis of DNA microarray data*, Proceedings of the 26th Annual International Conference of the IEEE EMBS, 2004.

[39] BLAST, *(Basic Local Alignment Search Tool)*, url = http://blast.ncbi.nlm.nih.gov/Blast.cgi/.

[40] *FASTA*, url = http://www.ebi.ac.uk/Tools/fasta/.

[41] *G*enbank, url = "http://www.ncbi.nlm.nih.gov/genbank".

[42] NCBI *(National Center for Biotechnology Information)*, url = "http://www.ncbi.nlm.nih.gov/genbank".

[43] G. Navarro, *Indexing Text using the Ziv-Lempel Trie*, Journal of Discrete Algorithm, volume 18, pages 333 -340, 2005.

[44] Pragyan P. Mohanty and Spyros Tragoudas, *A Scalable Method for Identifying DNA Substrings Using Functions*, Proceedings of the ISCA , International Conference on Bioinformatics and Computational Biology, pages 178-183, 2011.

[45] Phaninder Alladi , Pragyan P. Mohanty and Spyros Tragoudas, *A Scalable Method*

for Arbitrary String Matches Using Functions, Proceedings of the ISCA , International Conference on Bioinformatics and Computational Biology, pages 125-130, 2012.

[46] Pragyan (Sheela) Mohanty and Spyros Tragoudas, *Scalable Offline Searches in DNA Sequences*, Journal of Emerging Technologies in Computing, volume 11, number 2, 2014.

[47] E. W. Myers, *A Sublinear Algorithm for Approximate Keyword Searching*, Algorithmica, volume 12, number 4/5, pages 345-374, 1994.

[48] K. Sadakane, *New text indexing functionalities of the compressed suffix arrays*, Journal of Algorithms, volume 48, number 2, pages 294-313, 2003.

[49] V. Bafna and P. A. Pevzner, *Genome Rearrangements and Sorting by Reversals*, FOCS, pages 148-157, 1993.

[50] D.Bader, A. Moret, B.M.E., and Yan, M., *A linear-time algorithm for computing inversion distances between signed permutations with an experimental study*, Journal of Computational Biology, pages 483-491, Volume 8, 2001.

[51] Moret B.M.E., Tang, J., Wang, L.S., and WarnowT., *Steps toward accurate reconstruction of phylogenies from gene-order data* Journal of Computing Systems Sci. (special issue on computational biology), 2002.

[52] John D. Kececioglu and David Sankoff, *Exact and Approximation Algorithms for the Inversion Distance Between Two Chromosomes*, CPM, pages 87-105, 1993.

[53] V. Bafna and Pavel A. Pevzner, *Sorting by Transpositions*, SIAM Journal of Discrete Math, number 2, volume 11, pages 224-240, 1998.

[54] S. Hannenhalli and P. A. Pevzner, *Towards a Computational Theory of Genome Rearrangements*, Computer Science Today, pages 184-202, 1995.

[55] V. Bafna and P. A. Pevzner, *Genome Rearrangements and Sorting by Reversals*, SIAM Journal in Computing, volume 25, number 2, pages 272-289, 1996.

[56] D. Sankoff and M. Blanchette, *Multiple Genome Rearrangement and Breakpoint*

Phylogeny, Journal of Computational Biology, volume 5, number 3, pages 555-570, 1998.

[57] G. Bourque and P.A. Pevzner, *Genome-Scale Evolution: Reconstructing Gene Orders in the Ancestral Species*, Genome Research, volume 12, pages 26-36, 2002.

[58] N. El-Mabrouk and D. Sankoff, *Hybridization and Genome Rearrangement*, CPM, pages 78-87, 1999.

[59] D. A. Bader, *University of New Mexico Albuquerque, NM 87131* , url = http://www.cs.unm.edu/ moret/GRAPPA/, 2004.

[60] D. Graur and W-H. LI, *Fundamentals of molecular evolution*, Fundamentals of molecular evolution, 2nd Edition , Sinauer Associates, Inc, Sunderland, Massachusetts,2000.

[61] J. Palmer, *Chloropast and mitochondrial genome evolution in land plants* , incell Organelles, pages 99 -133, 1992.

[62] J. Palmer and L Herbon, *Plant mitochondrial DNA evolves rapidly in structure* , Journal of molecular evolution, volume Vol. 27, pages 87 -97, 1988.

[63] S. F. Altschul and B. W. Erickson and H. Leung, *Local Alignment (with Affine Gap Weights)*, Encyclopedia of Algorithms, 2008.

[64] T. Marschall and S. Rahmann, *Efficient exact motif discovery*, Bioinformatics, volume 25, number 12, 2009.

[65] R. R. Rudell, *Dynamic variable ordering for ordered binary decision diagrams.* Proceedings of the 1993 IEEE/ACM International Conference on Computer-aided Design, vol. 12, pp. 4247, 1993.

[66] *understanding-genomic-evolution*, url =http://www.cse.gatech.edu/features/understanding-genomic-evolution.

[67] *Genetics Home Reference*, url = http://ghr.nlm.nih.gov/.

[68] A.G. Clark, *Drosophila 12 Genomes Consortium 2007 Evolution of genes and genomes on the Drosophila phylogeny*, Nature, 450: 203-218, 2007.

[69] T.C. Kaufman, D.W.Severson, G.E .Robinson , *The Anopheles genome and comparative insect genomics*, Science, 298:9798, 2002.

[70] M.Cosner , R.Jansen , B.Moret ,L. Raubeson , L.Wang ,T. Warnow , S. Wyman, *An empirical comparison of phylogenetic methods on chloroplast gene order data in campanulaceae*, Comparative Genomics, (Kluwer Academic Publ. Montreal, Canada), pp 99121, 2000.

[71] B.Moret ,L. Wang , T.Warnow ,S. Wyman , *New approaches for reconstructing phylogenies from gene order data* Proceedings of Int. Conference of Intell. Syst. Molecular Biology, pp 165173, 2001.

[72] *GRIMM*, url = http://grimm.ucsd.edu/MGR/examples.html/.

[73] P. Berman, S. Hannenhalli and M. Karpinski, *1.375-Approximation Algorithm for Sorting by Reversals*, booktitle:Algorithms - ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17-21, 2002, Proceedings, pp 200-210, 2002.

VITA

Graduate School
Southern Illinois University

Pragyan Paramita Mohanty

Email address: pmohanty@siu.edu

Southern Illinois University at Carbondale
Master of Science, Electrical Engineering, August 1997

Special Honors and Awards:

Member of Honors Society of Phi Kappa Phi

Dissertation Title: FUNCTION-BASED ALGORITHMS FOR BIOLOGICAL SEQUENCES

Major Professor: Dr. Spyros Tragoudas

Publications:

Pragyan (Sheela) Mohanty, Spyros Tragoudas: **Offline Searches in DNA Sequences**, Journal of Emerging technologies in Computing (JETC), Volume 11(issue 2), article 18 (2014)

Phaninder Alladi, Pragyan P. Mohanty, Spyros Tragoudas: **A Scalable Method for Arbitrary String Searches in DNA Sequence**, International conferences on Bioinformatics and Computational Biology (2012)

Pragyan P. Mohanty, Spyros Tragoudas: **A Scalable Method for Identifying DNA Substrings Using Functions**, International conferences on Bioinformatics and Computational Biology (2011)