EAST WEST UNIVERSITY

Motif Discovery Using Genetic Algorithm

Submitted By:

Al Muttakin

ID: 2013-2-60-005

Supervised By:

Dr. Mohammad Rezwanul Huq

Assistant Professor

Department of Computer Science and Engineering

East West University

A thesis paper submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Computer Science and Engineering to the Department of Computer Science and Engineering.

April 2017

Abstract

Motif discovery in unaligned DNA sequences is a challenging problem in computer science and molecular biology. Finding a cluster of numerous similar subsequences in a set of biopolymer sequences is evidence that the subsequences occur not by chance but because they share some biological function. Motifs can be used to determine evolutionary and functional relationships. Over the past few years, many motif discovery tools have been designed and make available to public. In this paper, we represent an algorithm on motif discovery developed using Genetic Algorithm (GA). Our algorithm is originally based on a popular motif finding algorithm "Finding Motifs by Genetic Algorithm" (FMGA) developed by Falcon F.M Liu, with a handful of modifications to get better result. In our approach, we try to find potential Motifs from a group of promoter sequences of transcription start site (TSS). The Genetic operations such as mutation, crossover is performed using position weight matrix to reserve the completely conserved position. A rearrangement method is used to avoid the presence of a very stable local minimum. A preprocessing function is used to relate randomly generated initial motifs with the promoter sequences and a discursion function is used to minimize the computational time. We evaluated our result based on a fitness score and occurrence frequency of a candidate motif in a group of promoter sequence. Our approach give better result than the original FMGA algorithm which itself showed superior result with comparison to two other Motif finding algorithm namely Multiple Em for motif Elicitation (MEME) and Gibbs Sampler.

Declaration

I hereby declare that; this thesis paper was done under the CSE497 and has not been submitted elsewhere for requirement of any degree or any other region except for publication

Al Muttakin

ID: **2013-2-60-005**

Department of Computer Science and Engineering

East West University

Letter of Acceptance

This thesis paper is submitted by Al Muttakin, ID: 2013-2-60-005 to the Department of Computer Science and Engineering, East West University, Dhaka, Bangladesh is accepted as satisfactory for the partial fulfillment of the requirement for the degree of Bachelor of Science in Computer Science and Engineering on April 9th, 2017.

1. _____

Dr. Mohammad Rezwanul Huq

Assistant Professor (Supervisor)

Department of Computer Science and Engineering

East West University

Dhaka, Bangladesh

2. _____

Dr. Ahmed Wasif Reza

Associate Professor & Chairperson (Acting)

Department of Computer Science and Engineering

East West University

Dhaka, Bangladesh

Acknowledgement

First of all, I am grateful to Almighty Allah for my good health and wellbeing and patience that were necessary to complete this book.

I wish to express my sincere thanks to Dr. Ahmed Wasif Reza, Associate Professor & Chairperson (Acting), Department of Computer Science and Engineering, East West University, for providing me with all the necessary facilities for this thesis.

I am also grateful to Dr. Mohammad Rezwanul Huq, Assistant Professor, Department of Computer Science and Engineering, for supervising me on this thesis for past 8 months. I am extremely thankful and indebted to him for sharing expertise, valuable guidance and encouragement extended to me.

I take this opportunity to express gratitude to all of the Faculty members, Department of Computer Science and Engineering, for their help and support. Their expertise and guidance helped me throughout my study life and will help me to walk the path in future.

Finally, I would like to thank my parents for the unceasing encouragement, support and attention. Without their support, I would never be able to complete my study successfully.

Table of Contents

Abstract	i
Declaration	ii
Letter of Acceptance	iii
Acknowledgement	iv
Table of Contents	v
Contents	vi
List of Figures	viii
List of Tables	ix

Contents

Chapter 1 Introduction

1.1 Introduction ---------- 1

1.2 GA Overview ---------- 2

1.2.1 Selection ---------- 3

1.2.2 Mutation ---------- 3

1.2.3 Crossover ---------- 4

1.3 Objective ---------- 5

1.4 Related Terms and Issues ---------- 5

1.5 Outline ---------- 6

Chapter 2 Related Work

2.1 EM Methods ---------- 7

2.2 Gibbs Sampling Method ---------- 7

2.3 FMGA ---------- 8

2.3.1 Pseudocode ---------- 9

2.3.2 Flowchart ---------- 9

Chapter 3 Architecture of Proposed Algorithm

3.1 Method ---------- 10

3.2 Evaluation Criteria ---------- 10

3.2.1 Fitness Score Function ---------- 10

3.2.2 Total Fitness Score ---------- 12

3.2.3 Occurrence Frequency ---------- 13

3.3 Proposed Algorithm ---------- 14

3.3.1 Pseudocode ---------- 15

3.3.2 Flowchart ---------- 17

3.4 Operations ---------- 18

3.4.1 Preprocessing ---------- 18

3.4.2 Mutation ---------- 19

3.4.3 Crossover ---------- 20

3.4.4 Rearrangement ---------- 20

3.4.5 Discursion ---------- 21

Chapter 4 Result Analysis and Comparison

4.1 Result analysis and Comparison ---------- 22

Chapter 5 Conclusion & Future Work

5.1 Concluding Remark ---------- 26

5.2 Future Work ---------- 27

References

Appendix

List of Figures

Figure 1: General Architecture of GA. ---------- 04

Figure 2: Data structures used in MDGA. ---------- 06

Figure 3: Flowchart of original FMGA algorithm. ---------- 09

Figure 4: Illustration of fitness score calculation. ---------- 11

Figure 5: Illustration of TFS Calculation. ---------- 13

Figure 6: Illustration of Occurrence Frequency calculation. ---------- 14

Figure 7: Flowchart of MDGA. ---------- 17

Figure 8: Illustration of generating position weight matrix. ---------- 18

Figure 9: Illustration of preprocessing. ---------- 19

Figure 10: Illustration of Mutation. ---------- 19

Figure 11: Illustration of Crossover. ---------- 20

Figure 12: Illustration of Rearrangement. ---------- 21

List of Tables

Table 1: Predicted motifs of MDGA (Group 1). ---------- 23

Table 2: Predicted motifs of MDGA (Group 2) ---------- 23

Table 3: Predicted motifs of MDGA (Group 3). ---------- 23

Table 4: Comparison between Predicted motifs of MDGA and FMGA (Group 1). ---------- 24

Table 5: Comparison between Predicted motifs of MDGA and FMGA (Group 2) ---------- 24

Table 6: Comparison between Predicted motifs of MDGA and FMGA (Group 3) ---------- 25

Chapter 1

Introduction

1.1 Introduction

The gene is the fundamental unit of inherited information in deoxyribonucleic acid (DNA), and is defined as a section of base sequences that is used as a template for the copying process called transcription. The main idea in gene expression is that every gene contains the information to produce a protein. Gene expression begins with binding of multiple protein factors, known as transcription factors, to enhancer and promoter sequences. Transcription factors regulate the gene expression by activating or inhibiting the transcription machinery. Unraveling the mechanisms that regulate gene expression is a major challenge in biology. An important task in this challenge is to identify regulatory elements, especially the binding sites in deoxyribonucleic acid (DNA) for transcription factors. These binding sites are short DNA segments that are called motifs. Pattern discovery in DNA sequences is one of the most challenging issues in computer science and molecular biology nowadays.

So, in the simplest form, our problem is "Given a set of promoter sequences, whether we can detect overrepresented motifs that are good candidate for being transcription factor binding sites or carrying other biological meaning." A DNA motif is defined as a nucleic acid sequence pattern that has some biological significance such as being DNA binding sites for a regulatory protein, i.e., a transcription factor. Normally, the pattern is fairly short (5 to 20 base pairs (bp) long) and is known to recur in different genes or several times within a gene [1]. DNA motifs are often associated with structural motifs found in proteins. Motifs can occur on both strands of DNA. Transcription factors indeed bind directly on the double-stranded DNA. Sequences could have zero, one, or multiple copies of a Motif.

Now the question that will comes in our mind is, why are we considering motif finding a problem? This first step of gene expression, 'transcription', is finely regulated by a number of different factors, among which 'transcription factors' (TFs) play a key role binding DNA near the transcription start site of genes (in the 'promoter' region). The actual DNA region interacting with and bound by a single TF (called TFBS) usually ranges in size from 8–10 to 16–20 bp. TFs bind the DNA in a sequence-specific fashion, that is, they recognize sequences that are similar but not identical, differing in a few nucleotides from one another. So, given a set of sequences, if we can find an unknown pattern of m letters that occurs frequently in a group of sequences, a simple

enumeration of all m-letter patterns that appear in the sequences gives the solution. In the past, binding sites were typically determined through rigorous experiments in the laboratories. That way it was hard to find a potential candidate motif without any prior knowledge. But with the emerging of computational methods, if we can develop an efficient algorithm that can predict potential motif in DNA sequences without any prior knowledge rather by searching, we will be able to explore deeper into the hidden massage in DNA.

With a view to solving this problem of motif discovery, a large number of algorithms are already developed and applied to various motif models over past decades. Most of these algorithms are designed to deduce motifs by considering the regulatory region (promoter) of several coregulated genes from a single genome. It is assumed that co-expression of genes arises mainly from transcriptional coregulation. As coregulated genes are known to share some similarities in their regulatory mechanism, possibly at transcriptional level, their promoter regions might contain some common motifs that are binding sites for transcription factors. In this paper, we represent a new algorithm "Motif Discovery using Genetic Algorithm" (MDGA) that will consider a group of promoter sequence of transcription start site of variable length from 800 bp to over 11000 bp and create a number of candidate motif randomly from those sequences. Then after passing the candidate motifs of first generation through a preprocessing, we will allow those candidate motifs to evolve through genetic operations like mutation, crossover for a number of generation and look for potential candidate motifs through all the generation of motifs that produced. To make it more related to biological meaning, we also introduce a deletion process of comparatively weak motifs that has no chance of further evolution. We also used the concept of two types of base pair (Purine (A, G) and Pyridine (T, C)) [2] in the scoring factor as these two types of tends alter within themselves during biological processes fairly frequently.

1.2 GA Overview

In computer science and operations research, a genetic algorithm (GA) [3] is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA) [4]. Genetic algorithms are commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection. In a genetic algorithm, a population of candidate solutions (called individuals, creatures, or phenotypes) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its chromosomes or genotype) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible. The population size depends on the nature of the problem, but typically contains several hundreds or thousands of possible solutions. Often, the initial population is generated randomly, allowing the entire range of possible solutions (the search space).

Occasionally, the solutions may be "seeded" in areas where optimal solutions are likely to be found. The general architecture of GA is illustrated in Figure 1. The common operators of GA are as follows:

1.2.1 Selection

During each successive generation, a portion of the existing population is selected to breed a new generation. Individual solutions are selected through a fitness-based process, where fitter solutions (as measured by a fitness function) are typically more likely to be selected. Certain selection methods rate the fitness of each solution and preferentially select the best solutions. Other methods rate only a random sample of the population, as the former process may be very time-consuming. The fitness function is defined over the genetic representation and measures the quality of the represented solution. The fitness function is always problem dependent.

1.2.2 Mutation

Mutation is a genetic operator used to maintain genetic diversity from one generation of a population of genetic algorithm chromosomes to the next. It is analogous to biological mutation. Mutation alters one or more gene values in a chromosome from its initial state. In mutation, the solution may change entirely from the previous solution. Hence GA can come to a better solution by using mutation. Mutation occurs during evolution according to a user-definable mutation probability. This probability should be set low. If it is set too high, the search will turn into a primitive random search.

For different genome types, different mutation types are suitable. Some types of mutation are as follows:

Bit string mutation: The mutation of bit strings ensues through bit flips at random positions.
Boundary mutation: This mutation operator replaces the genome with either lower or upper bound randomly. This can be used for integer and float genes.
Non-Uniform mutation: The probability that amount of mutation will go to 0 with the next generation is increased by using non-uniform mutation operator. It keeps the population from stagnating in the early stages of the evolution. It tunes solution in later stages of evolution. This mutation operator can only be used for integer and float genes.
Uniform mutation: This operator replaces the value of the chosen gene with a uniform random value selected between the user-specified upper and lower bounds for that gene. This mutation operator can only be used for integer and float genes.

Gaussian mutation: This operator adds a unit Gaussian distributed random value to the chosen gene. If it falls outside of the user-specified lower or upper bounds for that gene, the new gene value is clipped. This mutation operator can only be used for integer and float genes.

1.2.3 Crossover

In genetic algorithms, crossover is a genetic operator used to vary the programming of a chromosome or chromosomes from one generation to the next. It is analogous to reproduction and biological crossover, upon which genetic algorithms are based. Cross over is a process of taking more than one parent solutions and producing a child solution from them. There are methods for selection of the chromosomes. Two types of common crossover are as follows:

Single-point crossover: A single crossover point on both parents' organism strings is selected. All data beyond that point in either organism string is swapped between the two parent organisms. The resulting organisms are the children.

Two-point crossover: Two-point crossover calls for two points to be selected on the parent organism strings. Everything between the two points is swapped between the parent organisms, rendering two child organisms.

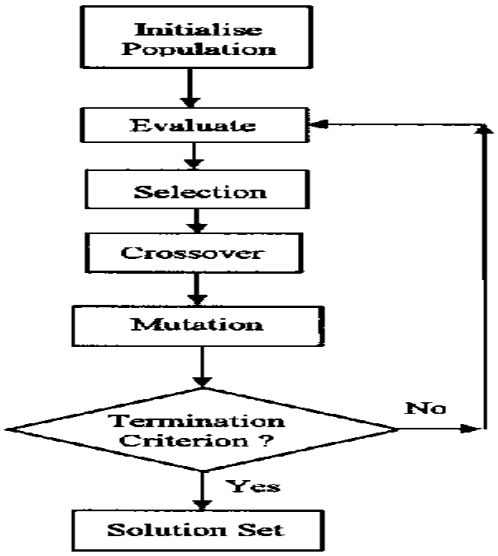


Figure 1: General Architecture of GA

1.3 Objective

The main objective of this thesis is to develop an algorithm for motif finding which can predict potential motifs with higher occurrence frequency from a set of promoter sequence that can regulate gene expression and be a good candidate for transcription factor binding site.

1.4 Related terms and Issues

Motif: A DNA motif is defined as a nucleic acid sequence pattern that has some biological significance such as being DNA binding sites for a regulatory protein, i.e., a transcription factor. Normally, the pattern is fairly short (5 to 20 base pairs (bp) long) and is known to recur in different genes or several times within a gene.

Promoter Sequence: A consensus sequence is defined as the sequence that reflects the most frequent base or amino acid at some position in a set of aligned DNA, RNA or protein sequences such as binding sites. Consensus sequences often represent conserved functional domains.

Data, MDGA deals with: A group of promoter sequence of transcription start sites of a particular genome.

Data Structure Used in MDGA: It uses 3 customized Data Structure

- ➔ WEIGHT_MATRIX: Store the position weight matrix of a candidate motif
- ➔ PATTERN: Store a particular motif, cross ponding fitness score, occurrence frequency, matching percentage.
- ➔ GENERATION_INFO: Store all information of all the patterns of a particular generation.

The data structures are defined in figure 2.

Framework and Programming language

- ⇨ C++
- ⇨ Windows Operating system

```
struct WEIGHT_MATRIX
{
	vector<double> wm[4];
};

struct PATTERN
{
	string pat;
	double tfs;
	int complete_match;
	double percentage;
};
```

```
struct GENERATION_INFO
{
	int number_of_pattern, max_match;
	double max_tfs, min_tfs;
	vector<PATTERN> patterns;
	map<string, int> unique_pattern;
	map<string, vector<PATTERN> > best_matches;
	map<string, WEIGHT_MATRIX> weight_matrix_all;
	map<string, pair<string, string> > parents;
	map<string, pair<string, string> > children;
};
```

Figure 2: Data structures used in MDGA

1.5 Outline

The rest of the report is organized as follows. In Chapter 2 we described the related work on motif finding in promoter sequences followed by the architecture of out proposed algorithm in Chapter 3. In Chapter 4 we analyze the result of our proposed algorithm and compare the result with the result of FMGA. Finally, in Chapter 5, we make the concluding remarks and stated the future work of our algorithm.

Chapter 2

Related Work

In this section, we will give a short overview of a few methods developed in past 2 decades that successfully predicted candidate motifs.

2.1 EM Methods

EM for motif finding was introduced by Lawrence and Reilly [5] and it was an extension of the greedy algorithm for motif finding by Hertz et [6]. It was primarily developed for protein motifs; however, it can also be applied for DNA motif finding. No alignment of the sites is required and the basic model assumption is that each sequence must contain at least one common site. The uncertainty in the location of the sites is handled by employing the missing information principle to develop an EM algorithm. This approach allows for the simultaneous identification of the sites and characterization of the binding motifs. The MEME algorithm by Bailey and Elkan [7] extended the EM algorithm for identifying motifs in unaligned biopolymer sequences. The aim of MEME is to discover new motifs in a set of biopolymer sequences where little is known in advance about any motifs that may be present. MEME incorporated three novel ideas for discovering motifs. First, subsequences that actually occur in the biopolymer sequences are used as starting points for the EM algorithm to increase the probability of finding globally optimum motifs. Second, the assumption that each sequence contains exactly one occurrence of the shared motif is removed. Third, a method for probabilistically erasing shared motifs after they are found is incorporated so that several distinct motifs can be found in the same set of sequences, both when different motifs appear in different sequences and when a single sequence may contain multiple motifs.

2.2 Gibbs Sampling Method

Gibbs sampler for motif finding developed by Lawrence [8]. They did not apply this algorithm to DNA sequence but applied to protein sequence in the original article. Since one of the original assumptions of this algorithm was that there exists at least one instance of a motif in every sequence, the method is sometimes called the "site sampler". Gibbs sampler is a Markov Chain Monte Carlo (MCMC) approach: "Markov-Chain", since the results from every step depends only on the results of the preceding one like in EM. The statistical background of MCMC methods is explained in the book by Liu [9] and that of Gibbs sampling in the article by Liu [10]. In this algorithm, it is assumed that we are given a set of N sequences S^1, S^2, ……., S^N and we seek within

each sequence mutually similar segments of specified width W. The algorithm maintains two evolving data structures. The first is the pattern description, accompanied by an analogous probabilistic description of the "background frequencies" p^1, p^2, ……, p^{20}.

2.3 FMGA

Liu [11] developed the algorithm FMGA based on genetic algorithms (GAs) for finding potential motifs in the regions located from the -2000 bp upstream to +1000 bp downstream of the transcription start site. The mutation in GA is performed by using position weight matrices. to reserve the completely conserved positions. The crossover is implemented with specially designed gap penalties to produce the optimal child pattern. This algorithm also uses a rearrangement method based on position weight matrices to avoid the presence of a very stable local minimum, which may make it quite difficult for the other operators to generate the optimal pattern. The authors reported that FMGA performs better in comparison to MEME and Gibbs sampler algorithms. As we mentioned earlier, out algorithm is developed based on FMGA with some major modification.

Among these methods, Gibbs sampler has the advantage of spending lower computation time. MEME is superior to the other methods by its prediction accuracy, but has the drawback of taking enormous computation time. In this paper, we propose a new approach based on genetic algorithm to predict motifs. The predicted results obtained by using our approach are more accurate than that of Gibbs sampler and spend less computation time than MEME. But FMGA operate on a limited range of base pair (-2000 upstream to +1000 downstream) which make its scope limited to find optimal motif with better fitness score and occurrence frequency. It has also a limitation of operating on a smaller group of datasets. But our algorithm operates on a larger number of data set and happens to predict better motif with higher score and occurrence frequency than the original FMGA algorithm. Here we will provide the flowchart and the pseudocode of original FMGA algorithm and all the operations of it will be discussed with the operations of MDGA on letter chapter.

2.3.1 Pseudocode: FMGA

```
1. Initialization
2. Setting total number of iterations: M
3. Creating candidate motifs randomly: P1 ~ Pn
4. Import promoter sequences S1 ~ SL
5. While iteration number <= M
6.      While predicted_motif_unchanged > K
7.              Evaluating TFS for each candidate motif
8.              Keeping the candidate motifs with the highest TFS as
                the new generations.
                //The remaining candidate motifs
                //are created by weighted wheel selection
9.              Mutation using weight matrix to generate two parent
                patterns
10.             Crossover with ambiguity codes penalties to select the
                best child pattern for next generation
11. Rearrangement of candidate motifs
12. Increasing iteration number by 1
13. Output predicted motifs and corresponding TFS
```

2.3.2 Flowchart: FMGA

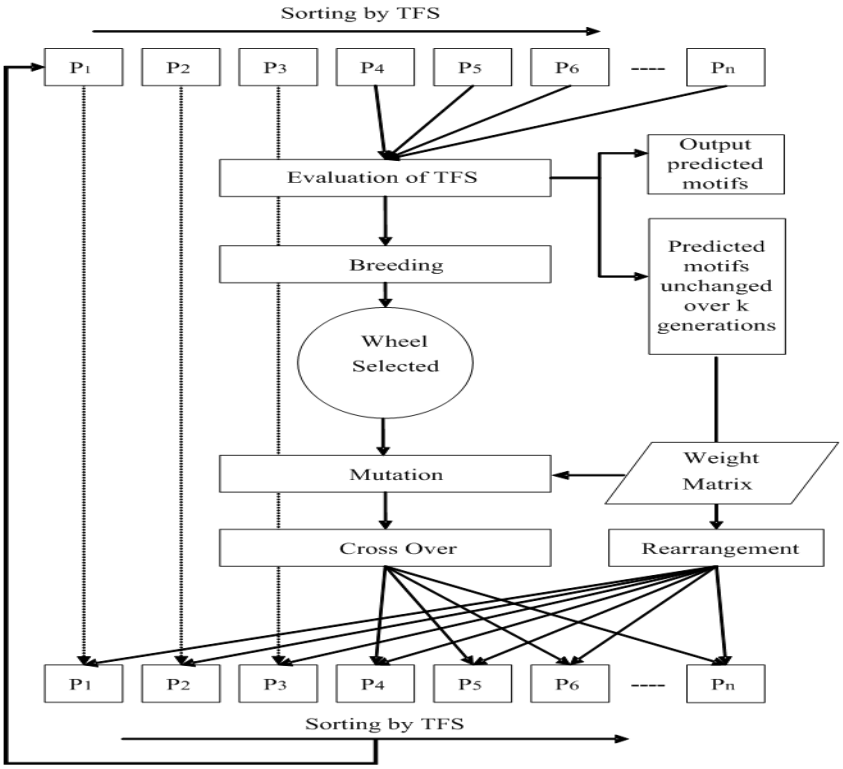


Figure 3: Flowchart of original FMGA algorithm

Chapter 3

Architecture of Proposed Algorithm

3.1 Method

Our method to predict motifs is to use a total fitness score function and occurrence frequency to find the optimal motif using genetic algorithm. Apart from the original FMGA algorithm, we introduced a new parameter in calculating the total fitness score on the basis of two types of base pair purine and pyridine. Like the original FMGA, we also use the general genetic algorithm framework and operators to serve as our basic architecture as sequence alignment by genetics algorithm (SAGA) [12] did. In the original FMGA, they only considered IUPAC ambiguity base pair [13] M, R, W, S, K, Y, N to calculate ambiguity code penalty. In the new algorithm, we also include other ambiguity base pairs that includes V, H, D, B to calculate ambiguity codes penalties to charge the scores of consensus sequences so that optimal motifs can be predicted more efficiently. We also introduce a change in the crossover operator where we keep both the child to be evaluated in the next generation. As a result, to keep the increasing number of candidate motif in check, we introduced a discursion function which eliminate comparatively weak motifs from each generation. And there by we removed the risk of a child that could be a better candidate motif, to be eliminated. This is one the major region of getting superior result than the original FMGA.

3.2 Evaluation Criteria

We evaluated out proposed algorithm on the basis of two parameters. Total fitness score and occurrence frequency. They are defined as follows:

3.2.1 Fitness Score Function

First let us consider the fitness score for a candidate motif in a single sequence. Given a motif pattern, there may have several regions in the sequence that match the motif pattern and each has a fitness score according to the fitness score function defined as follows:

$$FS(S_{mj}, P_n) = \sum_{i=0}^{k} match(Smji, Pni)/k$$

Where,

$$Match(S_{mij}, P_{ni}) = \begin{cases} 1 & if\ \ Smij = Pni \\ 0 & if\ \ Smij\ != Pni \end{cases}$$

Here, m is the index of sequences, i is the position within the motif, n is the index of motif patterns, k is the length of motif pattern, j is number of matched regions in the sequence. For example, the fitness score calculation of a motif P^1 in a promoter sequence S^1 is illustrated in the following figure:

P1: ACCGTA
Promoter S1: ATCCGGCTAACCGTACTATATTA
Fitness Score: 3/6 = .5 , 6/6=1

Figure 4: Illustration of fitness score calculation

In this case, we will take the fitness score of highest value. So finally, the fitness score function for a single sequence can be given by

$$FS(S_{mj}, P_n) = \max\left\{ \sum_{i=0}^{k} match(Smji, Pni)\ /k \right\}$$

In the original FMGA algorithm alongside the base pairs A, T, G, C they used ambiguity base pair M, R, W, S, Y, K into the calculation of fitness score. They redefined the fitness score function as follows:

$$match(S_{mji}, P_{ni}) = \begin{cases} 1 & if \quad S_{mji} = P_{ni} \quad for \quad P_{ni} \in \{A,T,C,G\} \\ 0.5 & if \quad S_{mji} = P_{ni} \quad for \quad P_{ni} \in \{M,R,W,S,Y,K\} \\ 0 & if \quad S_{mji} \neq P_{ni} \end{cases}$$

In our new algorithm, we take into consideration the two kind of base pair purine and pyridine as they tend to convert into each other frequently. So, these two types of base are represented by ambiguity base pair R (A, G) and Y (T, C). We also included the reaming ambiguity base pairs V, H, B, D that were grouped with N in the original FMGA with a small weightage. Giving R, Y more importance, we redefined out fitness score function as follows:

$$match(S_{mij}, P_{ni}) = \begin{cases} 1 & \text{if } S_{mij} = P_{ni} \text{ for } P_{ni} \in (A, T, G, C) \\ 0.5 & \text{if } S_{mij} = P_{ni} \text{ for } P_{ni} \in (R, Y) \\ 0.2 & \text{if } S_{mij} = P_{ni} \text{ for } P_{ni} \in (M, W, S, K) \\ 0.1 & \text{if } S_{mij} = P_{ni} \text{ for } P_{ni} \in (B, V, H, D) \\ 0 & \text{if } S_{mij} \text{ != } P_{ni} \end{cases}$$

3.2.2 Total Fitness Score

Total fitness score of a motif is the summation of fitness score of best match from all the sequences. It represents the score of motif in a particular generation. The total fitness score is defined as follows:

$$TFS(S, P_n) = \sum_{m=1}^{L} FS(S_m, P_n)$$

For example, if we consider a candidate motif P_1 and 5 promoter sequences $S_1 \sim S_5$ then the calculation of total fitness score of the motif is illustrated in the following figure:

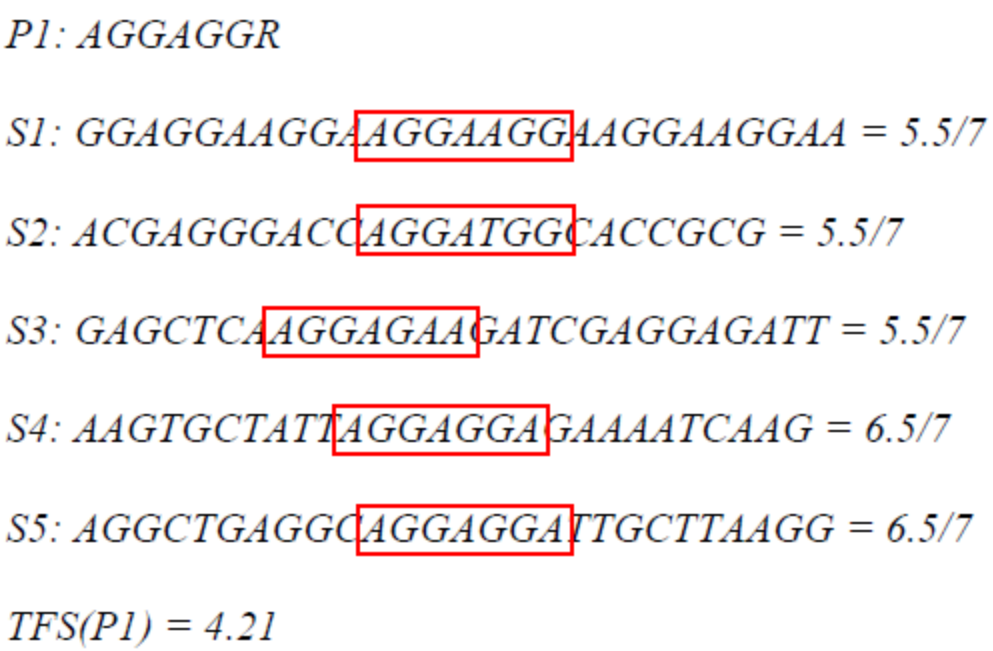


Figure 5: Illustration of TFS Calculation

3.2.3 Occurrence Frequency

In our new algorithm, we introduced a new parameter named occurrence frequency to evaluate our candidate motifs. Occurrence frequency (OF) of a given motif is defined as the number of sequences in a given dataset, in which, the candidate pattern matched completely with a subsequence of the promoter sequence.

For example, if we consider a candidate motif P_1 and 5 promoter sequences $S_1 \sim S_5$ then the calculation of occurrence frequency of the motif is illustrated in the following figure:

Figure 6: Illustration of OF calculation

3.3 Proposed Algorithm

We introduce our new algorithm MDGA in this section. Data is selected with an 800 bp length to over 11000 bp length from Transcription Start sites of different genome. In MDGA, the initial motif patterns with the same pattern length are created randomly. The users can set the pattern length. All the motif patterns will be different from the initial patterns after N-generation evolution. Preprocessing the first generation make the randomly generated motifs more suitable for evolution. Despite of starting with some randomly generated motifs, we preprocess the motifs and convert into worst possible motifs and start evolving. Mutation using weight matrix speed up the time to find potential motifs. During crossover unlike the original FMGA algorithm, we keep both the child to be evaluated on the next generation with other candidate motifs. This is to avoid the possibility of losing a globally potential motif. To check the increasing number of candidate motif, we use the discursion function with a variable discursion factor. The flowchart of MDGA is illustrated in Figure 7 and the pseudocode is given below.

3.3.1 Pseudocode: MDGA

```
MDGA(number_of_sequence)

1.  input -> sequences;
2.  input -> pattern_length;
3.  input -> number_of_pattern_in_generation1;
4.  for i <- 1 to number_of_pattern_in_generation1
5.           pattern[i] <- generate_random_pattern;
6.  present_gener_no <- 1
7.  generation_no <- 1;
8.  temp <- preprocess_generation1(present_gener);
9.  present_gener.patterns <- temp.fst;
10. present_gener.unique_pattern <- temp.snd;
11. count_same_gen <- 0;
12. count_max_tfs <- 0;
13. while generation_no <= 50
14.    if present_gener_no > 50
15.           break;
16.    for each pattern in present_gener
17.                  running_pat <- present_gener.patterns[i].pat;
18.                  present_gener.patterns[i].complete_match <- 0;
19.                  for each sequence in sequences
20.                         temp_pattern <- best_match_in_a_sequence
21.                         present_gener.best_matches[running_pat] <- temp_pattern
22.                         if temp_pattern.tfs == 1.0
23.                         present_gener.patterns[i].complete_match++;
24.                  present_gener.patterns[i].tfs <- calculate_total_fitness_score
25.                  present_gener.weight_matrix_all[running_pat] <-
       generate_weight_matrix
26.                  present_gener.patterns[i].percentage <-
       present_gener.patterns[i].complete_match/sequences.size
27.           if count_same_gen == 5 or count_max_tfs == 4
28.                  for each pattern in present_gener
29.                         if present_gener.patterns[i].tfs < present_gener.max_tfs
30.                                running_pat <- present_gener.patterns[i].pat;
31.                                temp.pat <- rearrange_pattern
32.                                temp.tfs <- -1;
33.                                next_gener.patterns[i] <- temp
34.                  iteration_no++;
35.                  count_same_gen <- 0;
36.                  count_max_tfs <- 0;
37.           else
38.                  present_gener.max_tfs <- calculate_max_tfs
39.                  present_gener.min_tfs <- calculate_min_tfs
40.                  present_gener.max_match <- calculate_max_match
41.                  for each pattern in present_gener
```

```
                        if present_gener.pattern[i].complete_match == 0
                              break;
                        best_patterns[i] <- present_gener.pattern[i]
                  present_gener <- discard_weak_patterns
                  present_gener.min_tfs <- calculate_min_tfs
                  for each pattern in present_gener
                        if present_gener.pattern[i].complete_match ==
      present_gener.max_match and present_gener.pattern[i].complete_match > 0
                                    next_gener.patterns <-
      present_gener.pattern[i]
                  for each pattern in present_gener
                        if present_gener.patterns[i].complete_match <
      present_gener.max_match or present_gener.patterns[i].complete_match == 0
                              present_gener.parents[running_pat] <- mutation
                              present_gener.children[running_pat] <- cross_over
                              temp.pat <-
      present_gener.children[running_pat].first;
                              temp.tfs <- -1;
                              next_gener.patterns[i] <- temp
                              temp.pat <-
      present_gener.children[running_pat].second;
                              temp.tfs <- -1;
                              next_gener.patterns[i] <- temp
                  if present_gener.unique_pattern == next_gener.unique_pattern
                        count_same_gen++
                  if present_gener.max_tfs == previous_gener.max_tfs
                        count_max_tfs++
                  present_gener_no++;
            previous_gener <- present_gener;
            present_gener <- next_gener;
      print_patterns
```

3.3.2 Flowchart: MDGA

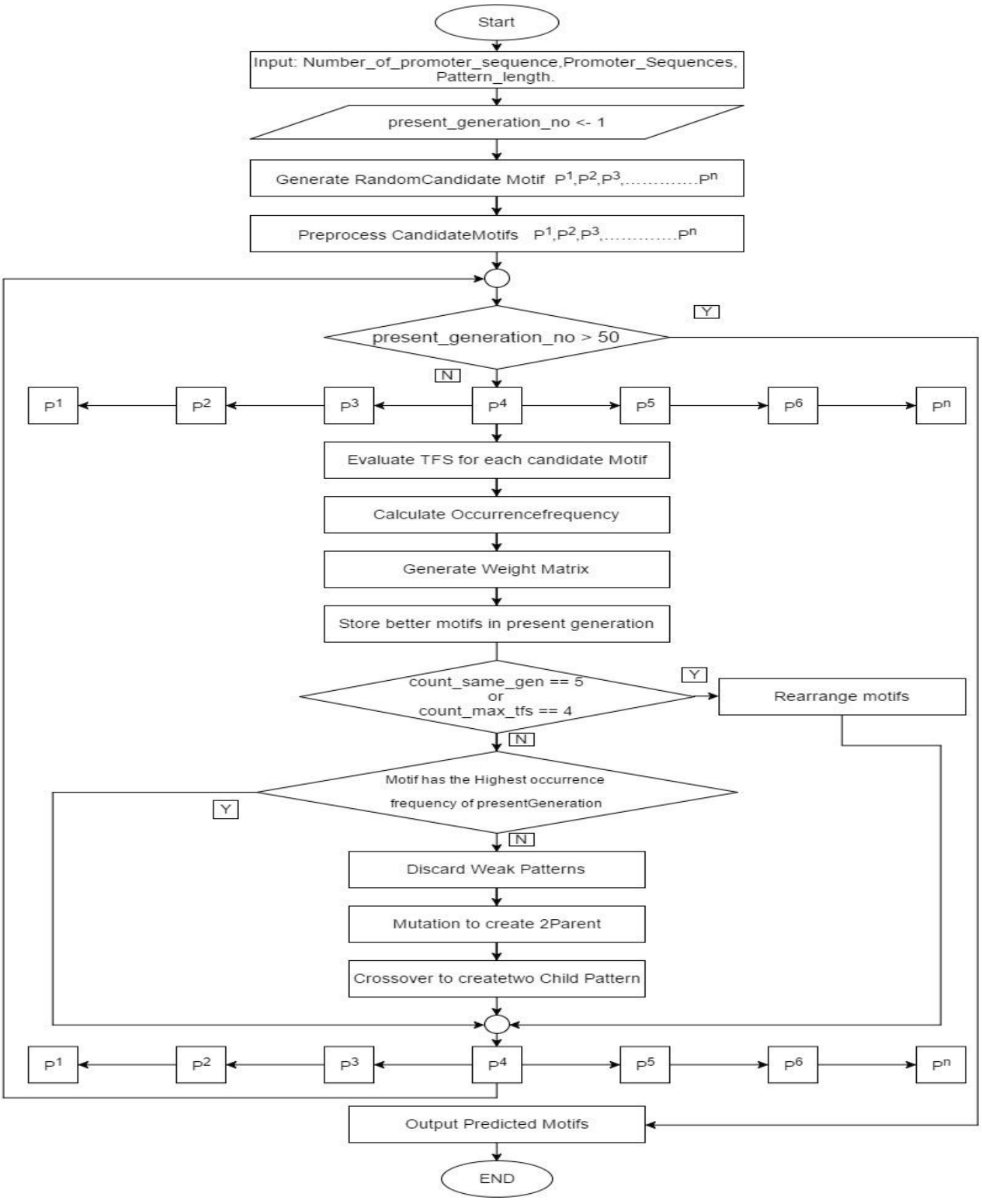


Figure 7: Flowchart of MDGA

3.4 Operations

The essential operations in MDGA are Preprocessing, Mutation, Crossover, Rearrangement and Discursion that can speed up the algorithm at the same time predict better motif. We introduce the operations as follows:

3.4.1 Preprocessing

After creating random motif at the beginning of the process we pass them through a preprocessing mechanism to convert them into ambiguity code base pair sequence. This is because while creating random motif, we have no idea about how the candidate motifs would be. After preprocessing, the candidate motifs will be fairly related to the promoter sequences and eventually they evolve into better candidate motif with high potential in later generations.

At the beginning of preprocessing, we generate position weight matrix of each pattern. The weight matrix is generated from the best matched pattern from each sequence cross ponding to a particular candidate motif. An illustration of creating a weight matrix is shown in figure 8.

Let us consider 5 best matched pattern of a candidate motif P^1 is given by, TGACGCA, TGACGCA, AGACGCA, TGACACA and AGACGCA. The score in weight matrix is calculated as the ratio of occurrences of corresponding base and the numbers of matched motifs. For example, in column 1 of the table, the numbers of occurrences of A are 2, the numbers of matched motif patterns are 5. So, the value is equal to 2/5 = 0.4.

TGACGCA
TGACGCA
AGACGCA
TGACACA
AGACGCA

	1	2	3	4	5	6	7
A	0.4	0	1	0	0.2	0	1
T	0.6	0	0	0	0	0	0
G	0	1	0	0	0.8	0	0
C	0	0	0	1	0	1	0

Figure 8: Illustration of generating position weight matrix

After generating the weight matrix, we insert ambiguity code on the basis of values in the weight matrix. The column with value 1 remain unchanged and the remaining base pair is changed into ambiguity code based on occurrence of the cross ponding base pairs.

For example, considering the following weight matrix of a candidate P_1, the illustration of preprocessing is shown in the following figure:

	0	1	2	3	4	5	6
A	.2	.3	.1	0	1	0	0
T	.1	0	0	1	0	.2	1
G	.5	.3	.9	0	0	.3	0
C	.2	.4	0	0	0	.5	0

NVRTABT

Figure 9: Illustration of preprocessing

3.4.2 Mutation

The aim of mutation is to create two parent motifs from a candidate motif to speed up the process of finding potential motif. It helps the crossover operation to create two child motifs for further evolution.

The mutation is done based on the weight matrix of a candidate pattern. First, we keep the base pair with value of 1 in a column unchanged and the rest of the base pairs are changed randomly. In our algorithm, for the first parent we use the base pair cross ponding to maximum value in a column. For the 2nd parent we use the base pair cross ponding to 2nd highest value in the column.

An illustration of mutation is as follows: Consider a weight matrix of a candidate patter P_1

	0	1	2	3	4	5	6
A	.4	0	1	0	.2	0	1
T	.6	0	0	0	0	0	0
G	0	1	0	0	.8	0	0
C	0	0	0	1	0	1	0

Parent 1: TGACGCA

Parent 2: AGACACA

Figure 10: Illustration of Mutation.

3.4.3 Crossover

The aim of crossover is to create two child motifs from two parent motifs produced in mutation. There are different kinds of crossover in biology. Here we will implement single point crossover.

First, we cut the two parent at the middle position. Then join left side of 1st parent with the right side of 2nd parent to create the 1st child motif. Then the right side of 1st parent is joined with the left side of 2nd parent to create the 2nd child motif. The illustration of crossover is shown in the following figure:

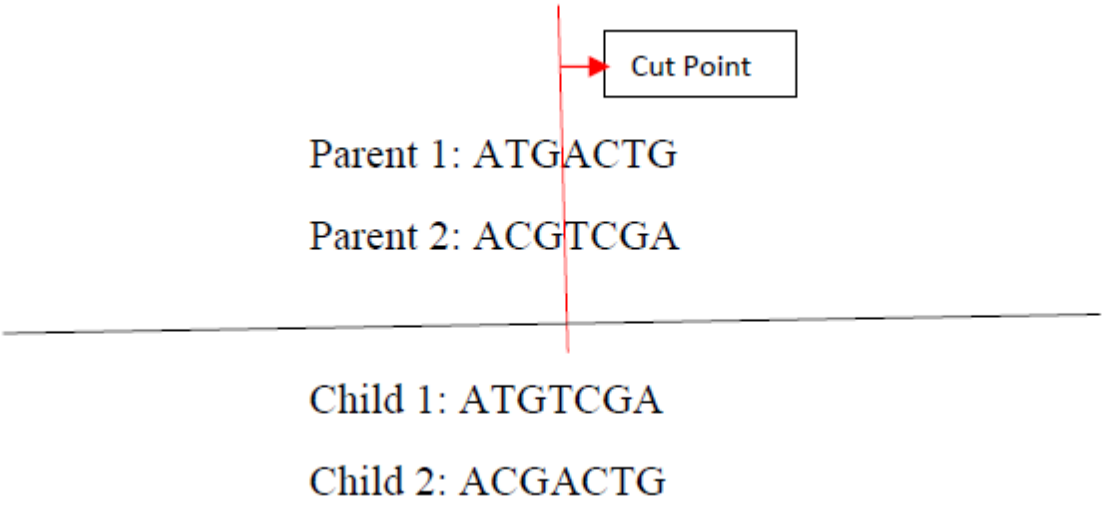


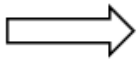
Figure 11: Illustration of Crossover.

3.4.4 Reengagement

In MDGA, if the predicted motifs are unchanged for more than K generations (e.g., K = 5) or if the max_occurrence_frequency remain unchanged for more than K generations, we rearrange the motifs by converting the ambiguity codes to A, T, C and G according to the weight matrix. The convention is to replace the ambiguity codes by the bases with the highest score in the corresponding column of the weight matrix.

If ambiguity code doesn't exist in the candidate motif, then we alter the base pair in respective positions based on the weight matrix. Given the position weight matrix candidate motif P_1 = TGACGCA, an illustration rearrangement operation is as follows:

	0	1	2	3	4	5	6
A	.4	0	1	0	.2	0	1
T	.6	0	0	0	0	0	0
G	0	1	0	0	.8	0	0
C	0	0	0	1	0	1	0



Motif Before Rearrangement:

⇨ TGACGCA

Motif After Rearrangement:

⇨ AGACACA

Figure 12: Illustration of Rearrangement.

3.4.5 Discursion

As we keeping both the child produced by crossover, the number of candidate motif will keep multiplying in every generation. So, if we don't take any measure in reducing the number of candidate motif somehow, the calculation time of each generation will increase exponentially. To minimize the time complexity, after keeping as much motifs as possible, we introduce a discursion function associated by a discursion factor, in our algorithm. Initially the function will discard a little amount of candidate motif that has a low TFS value and less occurrence frequency from each generation. With the increase of candidate motif in each generation, the discursion factor will adjust itself to keep the number of motifs low. At the least the discursion function will never let the number of motifs to increase more than 4 times than the initial number of candidate motifs. To avoid discarding any potential candidate motif, we avoid discarding any candidate motif that has further chance of evolution through mutation and crossover.

Chapter 4

4.1 Result analysis and Comparison

We implemented our algorithm in C++ under windows operating system. The data source is obtained from TSSDB website [14]. The test sequences are of length 800 bp to over 11000 bp located around the origin of transcription start site. Three groups of genes are used to test our system. Three sets of genes are as follows:

Group1: E2F1, E2F2, E2F3, E2F4, E2F5, E2F6

Group2: TGFB1 (transforming growth factor, beta 1), TGFB1I1, TGFB2, TGFB3, TGFBI, TGFBR1, TGFBR2, TGFBR3, TGFBRAP1

Group3: (Tumor Suppressor Genes) BRCA1, BRCA2, CDKN1A, CDKN2A, CDKN2B, CDKN2C, CDKN2D, E-cadherin (CDH1), E2F1, FABP3, FHIT, M6PR, IGF2R, NF2, NME1, PTEN, TGFB1, TP53

There are in total 14 promoter sequence of different variant from Group 1. 17 sequences from Group 2 and 49 sequences from Group 3. We set the lengths of motif patterns to be 7 and 13 to test our algorithm. We set the initial number of randomly generated motifs for motif length 7 as 100, and motif length 13 as 75. For testing our algorithm, we let the candidate motif to be evolved up to 50 generation and then evaluated our result.

To compare our algorithm with the original FMGA, we implemented it under the same environment and language as ours and executed it on the same dataset up to 50 generations as ours. Comparing with the original FMGA, our algorithm showed superior result and successfully find more potential motif than the original FMGA algorithm. The detail result of our algorithm is illustrated in Table 1 - Table 3. The comparison is shown in Table 4 - Table 6.

In the tables, the columns ‘Complete Match’ represent the number of promoter sequence in which the cross ponding motif is found at least once. The column ‘Matching Percentage’ represent the ratio of ‘Complete Match’ and total number of sequence, multiplied by 100. In the result tables in our algorithm we showed top 3 predicted motifs with a higher ‘Matching Percentage.’

Motif Length	Predicted Motifs	TFS	Complete Match	Matching Percentage
7	AAGAAGT	14	14 / 14	100 %
	GCAGCAG	14	14 / 14	100 %
	GAAGGAA	13.9286	13 / 14	92.8571 %
13	ATGACATCACCAA	13.7308	8 / 14	57.1429 %
	AAAGAAATACCAG	12.6385	3 / 14	21.4286 %
	CATTCAGGCACCT	12.0308	3 / 14	21.4286 %

Table 1: Predicted motifs of MDGA (Group 1)

Motif Length	Predicted Motifs	TFS	Complete Match	Matching Percentage
7	AAAAAAA	16.9286	16 / 17	94.1176 %
	TTCTCCT	16.8571	16 / 17	94.1176 %
	GCCAGCC	16.8571	15 / 17	88.2353 %
13	GAGAATTTTTTTC	15.0308	3 / 17	17.6471 %
	GAGGGAGCCCCCT	14.7308	3 / 17	17.6471 %
	ATCATTTTTACCA	14.4769	3 / 17	17.6471 %

Table 2: Predicted motifs of MDGA (Group 2)

Motif Length	Predicted Motifs	TFS	Complete Match	Matching Percentage
7	AAAAAAA	47.9714	40 / 49	81.6327 %
	TAAAAAA	48.1429	39 / 49	79.5918 %
	GCTGCTG	48.1143	39 / 49	79.5918 %
13	TTTTTCTTTTTTT	44.0923	8 / 49	16.3265 %
	GGGAGGAGGCTGA	43.6231	8 / 49	16.3265 %
	TCCCTTTTTATAT	42.4615	8 / 49	16.3265 %

Table 3: Predicted motifs of MDGA (Group 3)

Motif Length	Algorithm	Predicted Motif	TFS	Complete Match	Matching Percentage	Remark
7	MDGA	AAGAAGT	14	14 / 14	100 %	MDGA Predicts better
		GCAGCAG	14	14 / 14	100 %	
	FMGA	GCAGCAG	14	14 / 14	100 %	
		TTCCTGT	13.9286	13 / 14	92.8571 %	
13	MDGA	ATGACATCACCAA	13.7308	8 / 14	57.1429 %	MDGA Predicts better
		AAAGAAATACCAG	12.6385	3 / 14	21.4286 %	
	FMGA	ATGTAACTCATGA	12.4231	3 / 14	21.4286 %	
		CTGCCTGAGCAAC	12.3846	3 / 14	21.4286 %	

Table 4: Comparison between Predicted motifs of MDGA and FMGA (Group 1)

Motif Length	Algorithm	Predicted Motif	TFS	Complete Match	Matching Percentage	Remark
7	MDGA	AAAAAAA	16.9286	16 / 17	94.1176 %	MDGA predicts better
		TTCTCCT	16.8571	16 / 17	94.1176 %	
	FMGA	TTCTCCT	16.8571	16 / 17	94.1176 %	
		AGCTGTT	16.7857	14 / 17	82.3529 %	
13	MDGA	GAGAATTTTTTTC	15.0308	3 / 17	17.6471 %	Similar result
		GAGGGAGCCCCCT	14.7308	3 / 17	17.6471 %	
	FMGA	TCTAAACAAAAAA	15.3462	3 / 17	17.6471 %	
		ATGAAATCATGTC	15.1923	3 / 17	17.6471 %	

Table 5: Comparison between Predicted motifs of MDGA and FMGA (Group 2)

Motif Length	Algorithm	Predicted Motif	TFS	Complete Match	Matching Percentage	Remark
7	MDGA	AAAAAAA	47.9714	40 / 49	81.6327 %	MDGA predicts better
		TAAAAAA	48.1429	39 / 49	79.5918 %	
	FMGA	AAAAAAA	48.2143	40 / 49	81.6327 %	
		AAAGAAA	48	36 / 49	73.4694 %	
13	MDGA	TTTTTCTTTTTTT	44.0923	8 / 49	16.3265 %	FMGA predicts better
		GGGAGGAGGCTGA	43.6231	8 / 49	16.3265 %	
	FMGA	TAAAAAAAAAAAA	46.3462	18 / 49	36.7347 %	
		TTTTTCTGTTTCT	44.5385	7 / 49	14.2857 %	

Table 6: Comparison between Predicted motifs of MDGA and FMGA (Group 3)

From the above comparison we see that, out of 6 comparisons from 3 groups, MDGA shows better result in 4 comparisons while shows similar result in 1 comparison. In group 2, for the motif length 7, MDGA predicted two motifs with 100% occurrence frequency that means those two motifs are found in all the sequences in group 1. On the other hand, in on the same group of sequences, FMGA predicted 1 motif with 100% occurrence frequency and another with 92% occurrence frequency. For a motif length of 13, MDGA predicted a motif with over 57% occurrence frequency while FMGA predicted a motif with over 21% occurrence frequency.

In group 2, for the motif length 7, MDGA predicted two motifs with over 94% occurrence frequency. That means out of 17 sequences, those motifs are found in 16 of them. For the same group of sequences, FMGA predicted one motif with 94% frequency and another with 82% occurrence frequency. For motif length 13, Bothe the algorithm predicted motif with equal occurrence frequency.

In group 3, for motif length 7, MDGA predicted one motif with around 82% occurrence frequency. That means out of 49 sequences that motif is found in 40 sequences. MDGA predict the 2^{nd} motif with around 80% occurrence frequency which is found in 39 sequences. Here FMGA predicted a motif with around 82% occurrence frequency which is on par with MDGA. But the 2^{nd} motif FMGA predicted have around 74% occurrence frequency with 36 complete match which is inferior to MDGA. For pattern length 13, FMGA predicted a motif with around 37% occurrence frequency and 18 complete match out of 49. While MDGA predicted a motif with over 16% occurrence frequency and 8 complete match out of 49. Among all the test cases, this is the only one where FMGA predicted better than MDGA. Here we can notice that, prediction becomes harder with the increase of motif length and number of promote sequences.

Chapter 5

5.1 Concluding Remarks

From the results presented above, the performance of MDGA is better than the original FMGA algorithm. We had shown that MDGA predicts better motif patterns than FMGA which itself is superior to two other popular algorithm MM and Gibbs Sampler. FMGA can predict more potential motifs than the other algorithms because the patterns are generated randomly during the operation processes of GA. This characteristic is used to overcome the possible problems of local minimum. But as the motifs are generated completely at random, so they have a very little chance of being related to the given sequences. As a result, the algorithm loses efficiency. In our algorithm, after generating random motifs, we preprocessed them on the basis of given sequences and make them related to given sequences. So, it can predict motif more accurately. In FMGA they also removed one child during the process of crossover. Now if both the children of a single motif are weaker, on the other hand if both the children of another motif is stronger, in FMGA, it will lose a strong child to a weak one. To come over this limitation, in our algorithm we keep both the children of all the patterns of a single generation and evaluate them as a whole in the next generation. Due to this reason, the number of candidate motifs increases in every generation which will increase the computation time of MDGA. To overcome that problem, we introduced a discursion function to keep the increasing number of candidate motifs in check. The discursion function is associated with a discursion factor. Tuning of discursion factor affect the final result to a great extent. If we keep the discursion factor high, that will increase the predicting accuracy of MDGA but will increase the computation time proportionally. As for the initial number of randomly generated candidate motifs, if we increase the number, that will give better result as well but will have the same drawback. Genetic algorithm solves the optimal problem based on the biological characteristics. It uses a simple way to cope with complex problems. In this paper, we had proposed a new approach to predict motifs based on the genetic algorithm. A lot of biological messages are hidden in promoter, and motif is one of the important messages. The motifs have the possibilities to be the binding sites of transcription factors. If the motifs can be predicted accurately, the biologists can then explore which transcription factors activate genes. In future MDGA can contribute a lot in this sector.

5.2 Future Work

- In the future, we will try to implement MDGA in a distributed parallel computing system to overcome the problem of a huge computation time. In that way, we will be able to extract even better result from MDGA.
- In the process of crossover, we used the primary single point crossover. In future, we will try to implement double point and uniform crossover to get even better result.
- At present MDGA can operate on only single strand DNA and RNA. In future, we will extend MDGA to operate on double strand DNA and RNA and try to identify domains and proteins.

References

[1] Rombauts S, Dehais P, Van Montagu M, Rouze P: PlantCARE, a plant cis acting regulatory element database. Nucleic Acids Res 1999, 27:295-296.

[2] http://www.diffen.com/difference/Purines_vs_Pyrimidines (Accessed on 2.1.2017)

[3] https://en.wikipedia.org/wiki/Genetic_algorithm (Accessed on 11.12.2016)

[4] https://en.wikipedia.org/wiki/Evolutionary_algorithm (Accessed on 11.12.2016)

[5] Lawrence CE, Reilly AA: An expectation maximization (EM) algorithm for the identification and characterization of common sites in unaligned biopolymer sequences. Proteins 1990, 7:41-51.

[6] Hertz GZ, Hartzell GW, Stormo GD: Identification of consensus patterns in unaligned DNA sequences known to be functionally related. Comput Appl Biosci 1990, 6:81-92.

[7] Bailey TL, Elkan C: Unsupervised learning of multiple motifs in biopolymers using expectation maximization. Machine Learning 1995, 21:51-80.

[8] Lawrence CE, Altschul SF, Boguski MS, Liu JS, Neuwald AF, Wootton JC: Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. Science 1993, 262:208-214.

[9] Liu JS: Monte Carlo Strategies in Scientific Computing Springer Series in Statistics; 2001.

[10] Liu JS, Neuwald AF, Lawrence CE: Bayesian models for multiple local sequence alignment and Gibbs sampling strategies. J Amer Statist Assoc 1995, 90:1156-1170.

[11] Liu FFM, Tsai JJP, Chen RM, Chen SN, Shih SH: FMGA: finding motifs by genetic algorithm. Fourth IEEE Symposium on Bioinformatics and Bioengineering 2004:459.

[12] Cedric Notredame and Desmond G. Higgins, “SAGA: Sequence alignment by genetic algorithm,” J. Nucleic Acids Research, 24, pp. 1515-1524, 1996.

[13] M. Scherf, A. Klingenhoff, T. Werner, “Highly Specific Localization of Promoter Regions in Large Genomic Sequences by PromoterInspector: A Novel Context Analysis Approach,” J. Mol. Biol. 297 (3), pp. 599-606, 2000.

[14] http://dbtss.hgc.jp/index.html (Accessed on 03.15.2017)

Appendix

Source Code of Helper Functions

```
double calculate_fitness_score(string str, string pattern)
{
	double result = 0;

	for (int i = 0; i < pattern.sz; i++)
	{
		if (str[i] == pattern[i])
		{
			result += 1;
		}
		else
		{
			result += if_not_match(str[i], pattern[i]);
		}
	}

	return result/pattern.sz;
}

double calculate_total_fitness_score(vector<PATTERN> best_matches)
{
	double result = 0;

	for (int i = 0; i < best_matches.sz; i++)
	{
		result += best_matches[i].tfs;
	}

	return result;
}

double if_not_match(char base1, char base2)
{
	if ((base1 == base_pair[0] && base2 == base_pair[2]) || (base1 == base_pair[2]
&& base2 == base_pair[0]))
	{
		return 0.5;
	}
	else if ((base1 == base_pair[1] && base2 == base_pair[3]) || (base1 ==
base_pair[3] && base2 == base_pair[1]))
	{
		return 0.5;
	}
	else if ((base1 == base_pair[0] && base2 == base_pair[3]) || (base1 ==
base_pair[3] && base2 == base_pair[0]))
	{
		return 0.2;
	}
```

```
	else if ((base1 == base_pair[0] && base2 == base_pair[1]) || (base1 ==
base_pair[1] && base2 == base_pair[0]))
	{
		return 0.2;
	}
	else if ((base1 == base_pair[2] && base2 == base_pair[3]) || (base1 ==
base_pair[3] && base2 == base_pair[2]))
	{
		return 0.2;
	}
	else if ((base1 == base_pair[1] && base2 == base_pair[2]) || (base1 ==
base_pair[2] && base2 == base_pair[1]))
	{
		return 0.2;
	}
	else if (base2 == 'M' && (base1 == base_pair[0] || base1 == base_pair[3]))
	{
		return 0.2;
	}
	else if (base2 == 'R' && (base1 == base_pair[0] || base1 == base_pair[2]))
	{
		return 0.5;
	}
	else if (base2 == 'W' && (base1 == base_pair[0] || base1 == base_pair[1]))
	{
		return 0.2;
	}
	else if (base2 == 'S' && (base1 == base_pair[2] || base1 == base_pair[3]))
	{
		return 0.2;
	}
	else if (base2 == 'Y' && (base1 == base_pair[1] || base1 == base_pair[3]))
	{
		return 0.5;
	}
	else if (base2 == 'K' && (base1 == base_pair[1] || base1 == base_pair[2]))
	{
		return 0.2;
	}
	else if (base2 == 'V' && (base1 == base_pair[0] || base1 == base_pair[2] ||
base1 == base_pair[3]))
	{
		return 0.1;
	}
	else if (base2 == 'H' && (base1 == base_pair[0] || base1 == base_pair[1] ||
base1 == base_pair[3]))
	{
		return 0.1;
	}
	else if (base2 == 'D' && (base1 == base_pair[0] || base1 == base_pair[1] ||
base1 == base_pair[2]))
	{
		return 0.1;
	}
```

```
	else if (base2 == 'B' && (base1 == base_pair[1] || base1 == base_pair[2] ||
base1 == base_pair[3]))
	{
		return 0.1;
	}
	else if (base2 == 'N' && (base1 == base_pair[0] || base1 == base_pair[1] ||
base1 == base_pair[2] || base1 == base_pair[3]))
	{
		return 0.0;
	}
	else
	{
		return 0.0;
	}
}
char select_base_pair_from_weight_matrix(WEIGHT_MATRIX weight_matrix, int index)
{
	if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] == 0)
	{
		return 'A';
	}
	else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] == 0)
	{
		return 'T';
	}
	else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] == 0)
	{
		return 'G';
	}
	else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] > 0)
	{
		return 'C';
	}
	else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] > 0)
	{
		return 'N';
	}
	else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] == 0)
	{
		return 'D';
	}
	else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] > 0)
	{
		return 'H';
	}
	else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] > 0)
	{
```

```
            return 'V';
      }
      else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] > 0)
      {
            return 'B';
      }
      else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] > 0)
      {
            return 'M';
      }
      else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] == 0)
      {
            return 'R';
      }
      else if (weight_matrix.wm[0][index] > 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] == 0)
      {
            return 'W';
      }
      else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] == 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] > 0)
      {
            return 'S';
      }
      else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] == 0 && weight_matrix.wm[3][index] > 0)
      {
            return 'Y';
      }
      else if (weight_matrix.wm[0][index] == 0 && weight_matrix.wm[1][index] > 0 &&
weight_matrix.wm[2][index] > 0 && weight_matrix.wm[3][index] == 0)
      {
            return 'K';
      }
}

char select_base_pair_with_max_value(WEIGHT_MATRIX weight_matrix, int index)
{
      double mx = max4(weight_matrix.wm[0][index], weight_matrix.wm[1][index],
weight_matrix.wm[2][index], weight_matrix.wm[3][index]);

      for (int j = 0; j < 4; j++)
      {
            if (weight_matrix.wm[j][index] != mx)
            {
                  weight_matrix.wm[j][index] = 0
            }
      }

      return select_base_pair_from_weight_matrix(weight_matrix, index);
}
```

```
char select_base_pair_without_max_value(WEIGHT_MATRIX weight_matrix, int index)
{
	double mx = max4(weight_matrix.wm[0][index], weight_matrix.wm[1][index],
weight_matrix.wm[2][index], weight_matrix.wm[3][index]);

	for (int j = 0; j < 4; j++)
	{
		if (weight_matrix.wm[j][index] == mx && mx != 1)
		{
			weight_matrix.wm[j][index] = 0;
			break;
		}
	}

	return select_base_pair_with_max_value(weight_matrix, index);
}

char select_base_pair_for_rearrangment(WEIGHT_MATRIX weight_matrix, int index, char
present_base_pair)
{
	double mx = max4(weight_matrix.wm[0][index], weight_matrix.wm[1][index],
weight_matrix.wm[2][index], weight_matrix.wm[3][index]);

	for (int j = 0; j < 4; j++)
	{
		if (mx != 1 && base_pair[j] == present_base_pair)
		{
			weight_matrix.wm[j][index] = 0;
			break;
		}
	}

	return select_base_pair_with_max_value(weight_matrix, index);
}

pair<string, string> cross_over(string parent1, string parent2, int incision_point)
{
	if (incision_point == 0)
	{
		incision_point = parent1.sz / 2;
	}

	string p1l, p1r, p2l, p2r, child1, child2;

	p1l.assign(parent1.bgn, parent1.bgn + incision_point);
	p1r.assign(parent1.bgn + incision_point, parent1.end);
	p2l.assign(parent2.bgn, parent2.bgn + incision_point);
	p2r.assign(parent2.bgn + incision_point, parent2.end);

	child1 = (p1l + p2r);
	child2 = (p2l + p1r);

	return make_pair(child1, child2);
}
```

```
pair<string, string> mutation(WEIGHT_MATRIX weight_matrix)
{
	string parent1, parent2;

	for (int i = 0; i < weight_matrix.wm[0].sz; i++)
	{
		parent1.psb(select_base_pair_with_max_value(weight_matrix, i));
		parent2.psb(select_base_pair_without_max_value(weight_matrix, i));
	}

	return make_pair(parent1, parent2);
}

string rearrange_pattern(WEIGHT_MATRIX weight_matrix,string running_pattern)
{
	string rearranged_pattern;

	for (int i = 0; i < pattern_length; i++)
	{
		rearranged_pattern.psb(select_base_pair_for_rearrangment(weight_matrix,
i,running_pattern[i]));
	}

	return rearranged_pattern;
}

string preprocess_pattern(WEIGHT_MATRIX weight_matrix)
{
	string preprocessed_pattern;

	for (int i = 0; i < weight_matrix.wm[0].sz; i++)
	{

	preprocessed_pattern.psb(select_base_pair_from_weight_matrix(weight_matrix,
i));
	}

	return preprocessed_pattern;
}

PATTERN best_match_in_a_sequence(string sequence, string pattern)
{
	double mx;
	PATTERN best_pattern;
	best_pattern.tfs = -1.0;

	for (int i = 0; i < sequence.sz - pattern.sz; i++)
	{
		string temp(sequence.bgn + i, sequence.bgn + i + pattern.sz);

		mx = calculate_fitness_score(temp, pattern);

		if (mx > best_pattern.tfs)
		{
			best_pattern.tfs = mx;
```

```
                best_pattern.pat = temp;
            }
        }

        return best_pattern;
}

PATTERN generate_random_pattern()
{
        PATTERN generated_pattern;

        for (int i = 0; i < pattern_length; i++)
        {
            int x = rand() % sequences.sz;
            int y = rand() % sequences[x].sz;

            generated_pattern.pat.psb(sequences[x][y]);
        }

        generated_pattern.tfs = -1;

        return generated_pattern;
}

pair<vector<PATTERN>,map<string,int>> preprocess_generation1(GENERATION_INFO
generation1)
{
        map<string, int> temp;
        PATTERN temppat;
        vector<PATTERN> preprocessed;

        for (int i = 0; i < generation1.patterns.sz; i++)
        {
            PATTERN temp_pattern;

            for (int j = 0; j < sequences.sz; j++)
            {
                temp_pattern = best_match_in_a_sequence(sequences[j],
generation1.patterns[i].pat);

        generation1.best_matches[generation1.patterns[i].pat].psb(temp_pattern);
            }

            generation1.weight_matrix_all[generation1.patterns[i].pat] =
generate_weight_matrix(generation1.best_matches[generation1.patterns[i].pat]);

            temppat.pat =
preprocess_pattern(generation1.weight_matrix_all[generation1.patterns[i].pat]);

            map<string, int>::iterator it;

            it = temp.find(temppat.pat);

            if (it == temp.end)
            {
```

```
                        temppat.tfs = -1;
                        preprocessed.psb(temppat);
                        temp.insert(make_pair(temppat.pat, 1));
                }
                else
                {
                        it->snd++;
                }
        }

        return make_pair(preprocessed,temp);
}

WEIGHT_MATRIX generate_weight_matrix(vector<PATTERN> all_matches)
{
        WEIGHT_MATRIX weight_matrix;
        int cnt_t, cnt_g, cnt_c, cnt_a;

        for (int i = 0; i < pattern_length; i++)
        {
                cnt_a = cnt_c = cnt_g = cnt_t = 0;

                for (int j = 0; j < all_matches.sz; j++)
                {
                        if (all_matches[j].pat[i] == base_pair[0])
                        {
                                cnt_a++;
                        }
                        else if (all_matches[j].pat[i] == base_pair[1])
                        {
                                cnt_t++;
                        }
                        else if (all_matches[j].pat[i] == base_pair[2])
                        {
                                cnt_g++;
                        }
                        else if (all_matches[j].pat[i] == base_pair[3])
                        {
                                cnt_c++;
                        }
                }

                weight_matrix.wm[0].psb((cnt_a*1.0) / (all_matches.sz*1.0));
                weight_matrix.wm[1].psb((cnt_t*1.0) / (all_matches.sz*1.0));
                weight_matrix.wm[2].psb((cnt_g*1.0) / (all_matches.sz*1.0));
                weight_matrix.wm[3].psb((cnt_c*1.0) / (all_matches.sz*1.0));

        }

        return weight_matrix;
}

GENERATION_INFO discard_weak_patterns(GENERATION_INFO present_gener, double
discarsion_factor)
{
```

```
	if (present_gener.number_of_pattern <=  (2*number_of_pattern_in_generation1))
	{
		discarsion_factor = (present_gener.max_tfs - present_gener.min_tfs) /
2.5;
	}
	else
	{
		discarsion_factor = (present_gener.max_tfs - present_gener.min_tfs) / 7;
	}

	int cnt = 0;

	for (int i = 0; i < present_gener.patterns.sz;)
	{
		PATTERN running_pat = present_gener.patterns[i];

		if ((running_pat.tfs < present_gener.max_tfs - discarsion_factor &&
check_ambiguity(running_pat.pat)) || cnt >= 2*number_of_pattern_in_generation1)
		{
			map<string, vector<PATTERN> >::iterator it1;
			map<string, int>::iterator it2;
			map<string, WEIGHT_MATRIX>::iterator it3;

			it1 = present_gener.best_matches.find(running_pat.pat);

			if (it1 != present_gener.best_matches.end)
			{
				present_gener.best_matches.erase(it1);

			}
			it2 = present_gener.unique_pattern.find(running_pat.pat);

			if (it2 != present_gener.unique_pattern.end)
			{
				present_gener.unique_pattern.erase(it2);

			}

			it3 = present_gener.weight_matrix_all.find(running_pat.pat);

			if (it3 != present_gener.weight_matrix_all.end)
			{
				present_gener.weight_matrix_all.erase(it3);
			}

			present_gener.patterns.erase(present_gener.patterns.bgn + i);
		}
		else
		{
			i++;
			cnt++;
		}
	}

	return present_gener;}
```