Using a Genetic Algorithm for Detecting Repetitions in Biological Sequences

ADAM ADAMOPOULOS
Department of Medicine
Democritus University of Thrace
68100 Alexandroupolis
GREECE

KATERINA PERDIKURI
Department of Computer Engineering and Informatics
University of Patras
26500 Patras
GREECE

Abstract: - One of the most important goals in computational molecular biology is allocating repeated patterns in nucleic or protein sequences, and identifying structural or functional motifs that are common to a set of such sequences. In this paper we describe a new approach to detect the repetitions of fixed length in Biological Sequences using a Genetic Algorithm. The method involves evolving a population of patterns in an evolutionary manner and gradually improving the fitness of the population as measured by an objective function, which measures the approximate repetitions of the patterns in the given sequence. The general attraction of the approach is the ability to detect repeated schemas, thus inferring motifs of fixed length from biological sequences.

Key-Words: - Genetic Algorithms, Evolutionary Programming, Repetitions, Motifs, Biological Sequences, Pattern Matching

1 Introduction

Biological Sequences, such as DNA and Protein sequences, can be seen as long texts over specific alphabets, encoding the genetic code of living beings. Searching specific sub-sequences over those texts appears as a fundamental operation for problems such as assembling the DNA chain from the pieces obtained by experiments, looking for given DNA chains, or determining how different two genetic sequences are.

Other problems in Molecular Biology involve structure matching or searching for unknown repeated patterns, often called “motifs”. For various problems in molecular biology, in particular the study of gene expression and regulation, it is important to be able to infer repeated motifs or structured patterns and answer many biological questions, like for example what elements in sequence and structure are involved in the regulation and expression of genes through their recognition. The analysis of the distribution of repeated patterns permits biologists to determine whether there exists an underlying structure and correlation at a local or global genetic level.

In this work we address the problem of detecting repeated patterns in biological sequences using a Genetic Algorithm. Genetic Algorithms have been applied so far in the Multiple Molecular Sequence Alignment problem in order to identify similarities among sequences [1]. Our method involves evolving a population of patterns in an evolutionary manner and gradually improving the fitness of the population as measured by an objective function, which measures the approximate repetitions of the patterns in the given sequence. The general attraction of the approach is the ability to detect repeated schemas, thus inferring motifs of fixed length from biological sequences.

The structure of the paper is as follows. In Section 2 we give the basic theoretical background of our methodology and all the basic definitions used in the rest of the paper, in Section 3 we present the methodology and a brief time complexity analysis, while in Section 4 we give experimental results. Finally in Section 5 we conclude and discuss our future work and research interest in open problems of the area.

2 Background

In the relevant literature, a variety of algorithms for finding identical repetitions in DNA and protein sequences have been presented. In particular in [2], [3], [4] authors have presented efficient methods for finding identical repetitions in biosequences. In our method we use a Genetic Algorithm to efficiently compute the repetitions of fixed length in a biological sequence.

Genetic algorithms are stochastic approaches for efficient and robust searching. The GA methodology was invented by Holland [5] in order to study the natural adaptation phenomena and to incorporate the mechanisms of natural adaptation into computer systems, inspired by natural evolution methods [6], [7]. GAs, together with Evolution Strategies, Evolutionary Programming (EP), presented by Fogel et al. [8] and finally Genetic Programming (GP), proposed by Koza [9], form the field of Evolutionary Computation.

In Evolutionary Computation, a probabilistic algorithm maintains a population of individuals in every iteration *k*. Each one individual is a potential solution of the problem under consideration. With respect to the nature of the problem to be solved each individual is evaluated by an appropriate fitness function. Based on the fitness of the individuals a selection procedure, which mimics natural selection, is applied to form a new population (iteration *k + 1*). Some members of the new population undergo genetics-inspired transformations to form new solutions.

These transformations are either unary (mutation type), which create new individuals by small alterations in a single individual, or higher order (crossover type), which create new individuals by combining and exchanging, subparts of two (or in general, more) individuals. After the application of these transformation operators each individual of the new population is tested in order to evaluate the fitness of the solution it represents. After a reasonable number of generations the program converges to individuals that are optimal or near optimal (sub-optimal) solutions of the problem under consideration. Despite the simplicity of its implementation rules (selection, mutation and crossover), evolution is a powerful method of searching among an enormous number of possibilities for desired solutions. It is a massively parallel search method, which allows to the fittest (i.e. the optimal or near-optimal) solutions of each generation to survive, to be reproduced and finally to converge.

In the following paragraphs we give the basic definitions used in the rest of the paper.

2.1 Basic Definitions

Let Σ be a finite alphabet, which consists of a set of characters (or symbols). The cardinality of an alphabet, denoted by $|\Sigma|$, expresses the number of distinct characters in the alphabet. In the case of DNA sequences the alphabet consists of four nucleotides: $\Sigma_{DNA} = \{a,c,g,t\}$, while in the case of protein sequences, the alphabet consists of twenty amino acids.

DNA and protein sequences can be seen as long texts over specific alphabets representing the genetic code of living beings. A string or word is a sequence of zero or more characters drawn from an alphabet. The set of all non-empty words over the alphabet Σ is denoted by Σ^+. A word w of length n is represented by:

$$w[1..n] = w[1]w[2] \ldots,w[n],$$

where $w[i] \in \Sigma$, for $1 \le i \le n$, and $n = |w|$ is the length of w. The empty word is the empty sequence (of zero length) and is denoted by ε; A factor f of length p is said to occur at position i in the word w if $f = w[i,\ldots, i+p-1]$. In other words f is a substring of length p occurring at position i in word w. A word has a repetition when it has at least two consecutive equal factors.

In our approach every member or individual of the population is considered a factor f of length ℓ, where ℓ is an input variable to our algorithm, which represents the size of the repeated patterns under investigation.

2.2 The Suffix Tree

The Suffix Tree is a fundamental data structure supporting a wide variety of efficient string searching algorithms. In particular, the Suffix Tree is well known to allow efficient and simple solutions to many problems concerning the identification and location either of a set of patterns or repeated substrings (contiguous or not) in a given sequence. The reader can find an extended literature on such applications in [12].

In our algorithm we use the Suffix Tree of the input sequence to locate the occurrences of the repeated patterns. The pattern matching problem is solved by traversing the Suffix Tree in a top-down procedure and reporting the positions where a pattern *p* occurs in *S*. Starting from the root we are descending the tree following the edge dictated by

the characters of the pattern. If we have a mismatch at any point then there is no occurrence of the pattern in the input sequence. On the contrary, if all the characters of the pattern are consumed then we report all the leafs (positions of the sequence) spanning from the point of the ST where the matching ended.

Thus, for the reporting from the point that the matching has ended, we traverse (e.g. depth first search) the sub-tree and we report all the positions we discover on the leafs. If *k* leafs are spanned then the above procedure will take *O(k)* time because the tree is at least binary thus having *O(k)* internal nodes. Thus the pattern matching problem is linear to the size of the patterns.

3 Methodology

In this section we present the genetic algorithm developed and the proposed methodology. The population we consider in the GA consists of a population of p words of length ℓ. For each particular run, the populations size p (i.e. the number of individuals) of each generation, as well as the length ℓ of each one word of the population is kept constant. After establishing a population of words the population is randomly initialized. When the initialization procedure is completed all words of the population are random strings drawn from the Σ_{DNA} alphabet. The fitness *f* of each particular word is evaluated considering as fitness (or evaluation) function the number of approximate occurrences (repetitions) of the word in the input sequence. Therefore:

f(word) = matching characters/ length of word

The overall structure of the method is shown in Figure 1.

FIND REPETITIONS (X, ℓ, p, n, p_m, p_c, elitism)
Initialize population of words
WHILE n ≥ 1, **DO**
Evaluate-Fitness: compute the repetitions of each word of the population;
Produce Next Generation: compute the next generation;
If elitism = 0, perform elitism;
If $p_m \leq$ const, perform mutation;
If $p_c \leq$ const, perform uniform crossover;
Report individuals in descending order

Figure 1. The algorithm computes all repetitions of length ℓ for the input sequence X. The number of iterations n as well as the size of the population p is specified by the user. The probabilities of mutation and crossover, p_m and p_C respectively, as well as the elitism parameter are user-defined.

To go from one generation to the next, children are derived from parents that are chosen by some kind of natural selection. To create a child, an operator is selected that can be a crossover (mixing the contents of the two parents) or a mutation (modifying a single parent). Each operator has a probability of being chosen.

The algorithm is divided in two stages. The first one is the evolutionary phase where the new population of individuals/words is generated and the searching phase where each individual is evaluated by counting its number and exact positions of occurrences using the Suffix Tree data structure.

3.1 Evolutionary phase

When individuals' fitness evaluation is terminated, the selection procedure is applied in order to select the words for reproduction and the generation of the new population. Elitism is also applied during the selection phase. Elitism ensures that a user-defined number of the best individuals of each generation will be copied into the succeeding generation.

When the total number of words for reproduction is selected, the crossover and mutation operators are applied. The crossover operator randomly chooses two words. If a random number is less than, or equals to a given crossover probability (or crossover rate) p_c, then a locus of diversion of the words is randomly selected, splitting each one of them in two subparts. Then, two offspring are created by cross changing these subparts. Else (i.e., in the case that the random number is greater than p_c) clones of the two individuals are copied unchanged to the new population. On the new offspring the mutation operation is applied.

The mutation operator acts on each single character of a word. A random number is selected and if it is less than, or equals to a given mutation probability (or mutation rate) p_m the character under consideration is replaced by another character of the Σ_{DNA} alphabet that is randomly selected with the condition to differ from the original character. With the completion of the crossover and the mutation procedures the new generation is formed and a new evolution circle may begin. As termination condition of the GA procedures described above is the evolution for a certain number of generations specified by the user. When the GA comes to an end, the resulted words are reported in a descending order with respect to their fitness. Since the fitness of each word represents the number of occurrences found in

the input sequence, these results are easily readable.

3.2 Searching phase

In the searching phase we use the Suffix Tree data structure in order to compute the number of occurrences and positions of every repeated word.

3.3 Time Complexity Analysis

In this sub section we present a brief time complexity analysis of our algorithm. As already described our algorithm operates in two phases: searching and evolution. During the searching phase, repeated patterns with high frequency are identified. In the evolution phase repeated motifs are progressively mutated or crossed until all the existing, repeated patterns have been identified. In the computational analysis we will use the following notation:

p: the size of the population in each generation
ℓ: the length of each individual/word
n: the number of iterations/ generations
|*X*|*:* the length of the input sequence
The total time complexity is:

$$n^* (p^*\ell + p^* logp) + |X|.$$

According to the above analysis, the time complexity of our algorithm is linear to the size |X| of the input biological sequence and thus it is more efficient than other approaches of the relative literature. The time complexity also depends on the size of the population p, the length ℓ of each pattern and the specified number of generations/ iterations n of the algorithm.

4 Experimental Results

In this section we present experiments on real data using the above-described GA. A C++ program has been written to implement the technique. It is available on request from the authors. In order to test our method we used as input data biological sequences (such as genome sequences) downloaded from the European Bioinformatics Institute- EBI Database (http://srs.ebi.ac.uk).

Our method efficiently discovers the fittest atom per generation, i.e. the atom with the maximum number of occurrences, while moreover the algorithm reports all the atoms with high frequency per generation.

In Figure 2 it is shown the evolution of the mean fitness value of the GA for the same example. In Figure 3 we present the evolution of the mean fitness value of the GA for patterns of length ℓ = 32. These lengths were randomly chosen only for purpose of demonstration. The algorithm finds the most frequent patterns for any length ℓ ≤ |X|/2 (if the pattern equals to |X|/2 it can appear at most twice in the input sequence).

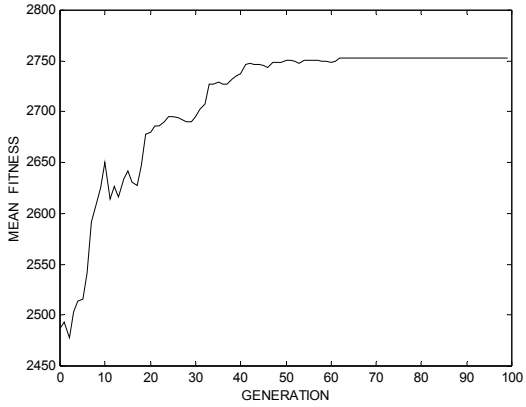

Figure 2. Evolution of the mean fitness value per generation for patterns of length ℓ =8.

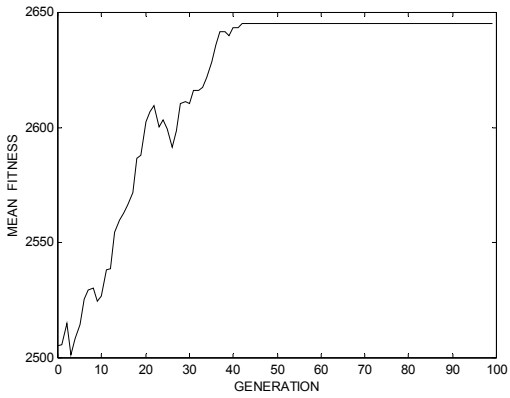

Figure 3. Evolution of the mean fitness value per generation for patterns of length ℓ =32.

5 Future Work

As previously analysed our method involves evolving a population of patterns in an evolutionary manner and gradually improves the fitness of the population as measured by an objective function. The general attraction of the approach is the ability to detect repeated schemas, thus inferring motifs of fixed length from biological sequences.

Our future work is two fold. The first one concerns the modification of the algorithm by assigning a credit to the operators of mutation and crossover. When creating a "*child-pattern"*, the choice of mutation or crossover (as expressed by the respective probabilities) is just as important as the choice of the *"parents-patterns"*. Therefore it makes sense to allow operators to compete for usage, just as parents do for survival, in order to make sure that the

most useful operator is likely to be used. Thus, each time a new individual is generated, if it yields some improvement over its parents, the operator that was directly responsible for its creation gets the largest part of the credit and so in the new generation we can dynamically change the probability of the mutation or crossover operator. This can reduce the time complexity needed to compute the mutation and crossover operation for the population in each generation.

The second research direction concerns the addition of one operator responsible for inserting gaps inside repeated patterns thus giving the possibility of inferring structured patterns from the input biological sequence.

Moreover an interesting problem arises from having *"don't care symbols"* in the input sequence. A "don't care" symbol (denoted as * or $), has the property of matching any symbol of a given alphabet or another *, as well. For example the pattern $p = AC*AT$ can match the string $s = A*G*T$. In this case the initial population will consist of patterns with *"don't care symbols"* or in other terms schemas. Using the same methodology we can compute the repetitions of such a sequence with "don't care" characters, as introduced in [14].

6 Conclusions

Detection of repeated patterns (as presented in the above paragraphs) has nowadays become a specific research field whose applications in molecular biology are of high impact. In fact, due to the huge amount of data entering genomic databases, there is an urgent need for tools that can help molecular biologist to interpret this data. In fact, a typical way to start analyzing new sequences is to group them into families that are assumed to be biologically related because they present similar function or structure, or because they are evolutionary related.

Compared to other techniques our algorithm is linear to the length of the input sequence and has the advantage of allowing the user to specify the exact length of the repetitions the biologist looks for. Taking into consideration the easy parallelisation of Genetic Algorithms we believe our method can be used in many practical applications.

Finally we believe that Genetic Algorithms can be successfully used as a practical way to solve many computationally difficult problems in the areas of Sequence Search and Alignment. They are intellectually satisfying in their simplicity and the way they attempt to mimic biological evolution.

References:

[1] C. Zhang, A.K. Wong, A genetic algorithm for multiple molecular sequence alignment. *Comput. Appl. Biosci*., Vol. 13, 1997, pp. 565–581.

[2] S. Kurtz, C. Schleiermacher, REPuter: fast computation of maximal repeats in complete genomes. *Bioinformatics*, Vol. 15, 1999, pp. 426–427.

[3] H. Martinez, An Efficient Method for Finding Repeats in Molecular Sequences. *Nucleic Acid Research*, Vol. 11, 1983, pp. 4626–4634.

[4] T. Tsunoda, M. Fukagawa, M, T. Takagi, Time and memory efficient algorithm for extracting palindromic and repetitive subsequences in nucleic acid sequences. *Pacific Symposium on Biocomputing*, Vol. 4, 1999, 202–213.

[5] J.H. Holland, *Adaptation in Natural and Artificial Systems*- Second Edition, MIT Press, 1992.

[6] D.E. Goldberg, *Genetic Algorithms in Search. Optimization and Machine Learning*. Addison-Wesley, 1989.

[7] M. Mitchell, *An Introduction to Genetic Algorithms,* MIT Press, 1996.

[8] L.J. Fogel, A.J. Owens, M.J. Walsh, *Artificial Intelligence through Simulated Evolution,* Wiley, 1996.

[9] J.R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection,* MIT Press, 1992.

[10] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs,* Springer-Verlag, 1994.

[11] C.H. Ooi, P. Tan, Genetic algorithms applied to multiclass prediction for the analysis of gene expression data, *Bioinformatics*, Vol. 19, 2003 pp. 37–44.

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

[13] J. Stoye, D. Gusfield, Simple and flexible detection of contiguous repeats using a suffix tree. *In proceedings of the 9th Annual Symposium on Combinatorial Pattern Matching (CPM)*, Vol. 1448 of Lecture Notes in Computer Science, 1998, pp. 140–152.

[14] C. Iliopoulos, M.Mohamed, L.Mouchard, K. Perdikuri, W. Smyth, A. Tsakalidis String Regularities with Don't Cares, *Nordic Journal of Computing,* Vol.10, 2003, pp. 40-51.