An Object Oriented Environment for Artificial Evolution of Protein Sequences: The Example of Rational Design of Transmembrane Sequences

Mariusz Milik and Jeffrey Skolnick

Abstract

A system is presented for generating peptide sequences with desirable properties, using combination of neural network and artificial evolution. The process is illustrated by an example of a practical problem of generating artificial transbilayer peptides. The peptides generated in the process of artificial evolution have the physico-chemical properties of transmembrane peptides, and forms stable transmembrane structures in testing Monte Carlo simulations. The artificial evolution system is designed to emulate natural evolution; therefore it is of both practical and theoretical interest, both in terms of rational design of protein sequences and modeling of natural evolution of proteins.

1 INTRODUCTION

Most of versions of genetic algorithms (GA) are used as efficient methods for optimizing complex functions, but in the case of the subclass of GA called *artificial evolution* (AE), emphasis is placed on the biological realism of the model, with clear separation between genotype and the information encoded in the genotype. The system presented in this paper is an example of implementation AE into the area of rational design of protein sequences. In order to obtain the "natural" probability of amino acid mutations, the protein sequences are encoded in this model as a sequence of nucleic acids. The *isolation by distance* effect is simulated by placing elements of the population onto one-dimensional grid of circular shape, with one genotype per location. Each individual can compete and mate only with its neighbors; parents of each offspring are chosen by *random walk*. The procedure starts from the first parent location; the new organism survives if it wins in a tournament with its neighbors also chosen by random walk.

As a test, this algorithm was used for rational design of transmembrane peptide sequences. In this test, the output from a pattern-recognition neural network was used as a fitness function. The network was previously trained on a set of known transmembrane protein fragments. This hybrid neural network + GA system was able to generate new families of transmembrane peptides. The best-fitted sequences were tested by Monte Carlo simulation and proved to have the structural and dynamical properties typical of transbilayer peptides.

The hybrid system is designed to be universal and may be used to design other important peptide sequences, e.g., enzyme inhibitors or antibiotics. Because of its biological realism, the system may also be used for study of the natural evolution process and artificial life problems. It is written in C++ and is thought to be a seed of an Object Oriented library for modeling of biological systems.

2 DESCRIPTION OF THE OBJECTS

The Neural Network library contains actually two main classes: **Bp** (back-propagation) and **BpTest**.

The **Bp** class is defined as two multilayer back-propagation networks that share weight matrices. The first network is used in the training process, while the second one is used in an "on line" testing process. At the present level of development, only a three-layer (input/hidden/output) architecture is implemented, and weights are initialized randomly. Listing 1 (below) presents the essential members and methods of the **Bp** class.

```
class Bp {
  int inp;              // number of input nodes
  int hid;              // number of hidden nodes
  int out;              // number of output nodes
  int pat;              // number of training patterns
  int test_pat;         // number of testing patterns
  double eta;           // learning parameter
  double alpha;         // momentum parameter
  Matrix I;             // Training pattern matrix - size pat x inp
  Matrix It;            // Testing pattern matrix - size test_pat x inp
  Matrix W1;            // Weights input->hidden size: inp x hid
  Matrix H;             // training signals on hidden layer
  Matrix Ht;            // testing signals on hidden layer
  Matrix W2;            // output weights matrix - size: hid x out
  Matrix O;             // training output matrix - size: pat x out
  Matrix Ot;            // testing output matrix - size: test_pat x out
  Matrix T;             // training target matrix  (size: pat x out)
  Matrix Tt;            // testing target vector (size: pat x out)
  Vector b1;            // biases for hidden layer
  Vector b2;            // biases for output layer
public:
  Bp(){};                         // default constructor
  Bp(char* file_name);            // construction from file
  ~Bp();                          // destructor
  void Save(char* file_name);     // save network on file
  void InitRandomly(double limits); // random start for weights
  void Forward();                 // calculation of out matrix
  void BackGradient(void);// gradient back-propagation learning cycle
};
```

The BpTest class is a simplified version of the **Bp** class, without learning methods. Objects from this class are used in the GA library as fitness functions.

The base of the Genetic Algorithm Library is the **Ent** class; objects from this class represent individuals of the model population. These objects contains a byte string with the individual genetic information, fitness of the Ent and its age. The Ents are the elements of the **Population** class, where environment and mating rules are defined. Listing 2 (below) presents the most important members and methods of the **Ent** class definition:

```
class Ent {
  friend Ent* Cross(Ent* first, Ent* second);   // crossover
  friend Ent* DCross(Ent* first, Ent* second);  // double crossover

private:
  Byte* genome;      // a string with genetic information
  int ssn;           // a "personal number" of the Ent
  int size;          // size of the genome (bytes)
  int Nres;          // number of residues in a coded peptide
  double fitness;    // actual fitness of the Ent
  int age;           // age of the Ent (in epochs)

public:              // some more important methods
                            // CONSTRUCTORS:
  Ent(void);                // default (empty) constructor
  Ent(int s);               // random initiation of an Ent with a
                            //      genome length equal "s"
  Ent(int s, Byte* gen);    // initiation of an Ent with a genome
                            //      copied from the string "gen"
  Ent(FILE* fil);           // from text file stored on the disk
  Ent(char cp, FILE* fil);  // from binary file stored on the disk
                            // DESTRUCTOR
  ~Ent(void);
                            // NON MODIFYING FUNCTIONS:
  void Fitness(void);       // this function is defined separately
                            // for every model
  char* OutNA(char* na_sq) const; // write the genome as a NA sequence
  char* OutAA(char* aa_sq) const; //write the genome as an AA sequence
  void Save(FILE* fil);           // save the Ent as a text file
  void Save(char cp, FILE* fil);  // save the Ent as a binary file
                                  // MODIFYING FUNCTIONS:
  void Mutate(void); // the point mutation of the Ent
  void Older(void); //what happen when the Ent advances in years ;-)
};
```

A set of generated Ents with some additional methods, creates the class Population. Fragments of the definition of this class are presented in the Listing 3 (below):

```
class Population {
private:
 int name;          // name of the population (important in the case
                    //     of multi-population environments)
  int size;         // size of the population
  Ent** habitat;    // storage for Ents
  int ChooseFromNeighbors(int nr, int step, int nsteps);
                    // definition of the tournament method
public:
                                   // CONSTRUCTORS
  Population();      // default constructor
  Population(int pop_size, int gen_size);
                    // construction of a population of random Ents
  Population(char* file_name);        // constr. from a text file
  Population(char cp, char* fil_name); // constr from a binary file
                                 // DESTRUCTORS
  ~Population(void);
                               // NON MODIFYING FUNCTIONS
  void Save(char* file_name); // save the population as a text file
  void Save(char cp, char* file_name); // save the population as
                                       // a binary file
 void PrintAA(char* message);  // print the genes from the population
                               //    as peptide sequences
  void Mating_Scheme_A(int step, int nsteps);
                                // one of definitions of the mating
                                //   scheme (random walk)
};
```

3 GENETIC ALGORITHM PROCEDURE

The method of coding of protein sequences mimics the natural nucleic acid (NA) coding scheme. Every amino acid (AA) is coded by the corresponding set of NA triplets from the genetic code, and every NA is represented by two bits. The resulting scheme is presented in Table 1.

Every six bits in the model genome represent a signal in the natural genetic code. Because the model point mutation works on the lowest level (bit flips), the presented representation gives a nature-like probability of the amino acid mutation. Some transitions are more probable, some less, depending on the genetic code.

Every string is evaluated after translation into the equivalent protein sequence. The *evaluation function* (fitness function) is defined outside of the main body of the library and should be defined separately for every application of the method. In the presented work, the trained neural network was used as a fitness function. The system, however is open and every function with a protein sequence as an input and real number as an output may be used here as an evaluation (fitness) function.

NA code	AA	binary code	NA code	AA	binary code
UUU	F	000000	AUU	I	100000
UUC	F	000001	AUC	I	100001
UUA	L	000010	AUA	I	100010
UUG	L	000011	AUG	M	100011
UCU	S	000100	ACU	T	100100
UCC	S	000101	ACC	T	100101
UCA	S	000110	ACA	T	100110
UCG	S	000111	ACG	T	100111
UAU	Y	001000	AAU	N	101000
UAC	Y	001001	AAC	N	101001
UAA	X	001010	AAA	K	101010
UAG	X	001011	AAG	K	101011
UGU	C	001100	AGU	S	101100
UGC	C	001101	AGC	S	101101
UGA	X	001110	AGA	R	101110
UGG	W	001111	AGG	R	101111
CUU	L	010000	GUU	V	110000
CUC	L	010001	GUC	V	110001
CUA	L	010010	GUA	V	110010
CUG	L	010011	GUG	V	110011
CCU	P	010100	GCU	A	110100
CCC	P	010101	GCC	A	110101
CCA	P	010110	GCA	A	110110
CCG	P	010111	GCG	A	110111
CAU	H	011000	GAU	D	111000
CAC	H	011001	GAC	D	111001
CAA	Q	011010	GAA	E	111010
CAG	Q	011011	GAG	E	111011
CGU	R	011100	GGU	G	111100
CGC	R	011101	GGC	G	111101
CGA	R	011110	GGA	G	111110
CGG	R	011111	GGG	G	111111

Table1. The amino acid representation used in the artificial evolution algorithm. Amino acids and nucleic acids are represented by their one-letter symbols. "X" denotes the "stop" triplets of the genetic code, which are not interpreted in the current version of the program.

One of the more annoying problems in genetic algorithm applications is *premature convergence*, in which population stagnates at a suboptimal solution. In the model presented, this problem is addressed by using the *isolation by distance* process and an *age* parameter. The isolation by distance is implemented in one-dimensional form. Every Ent in the model population inhabits one site in the circle-shaped habitat (an example may be the shore of a lake), and a probability of mating between pairs of Ents is a fast-declining function of their distance (calculated along the shore). Additionally, in order to find its own site (and survive), each newly created Ent

must win at least once in a set of duels with neighbors. A duel, in the presented implementation, consists of the comparison of the fitnesses of the Ents (with some random element). Depending on the implementation, the age parameter may be used here as an additional factor. The age parameter may decrease or increase fitness, changing the probability of survival for newly created individuals.
This method, used for large populations, creates a set of spatially isolated subpopulations of Ents with different genotypes. The interfaces between the subpopulations are the sources of variability in the population, as an effect of the recombination process.

The recombination and mutation processes operate on the lowest (bit) level of the model and are absolutely independent of the particular interpretation of the genetic code. In the model, the one- and two-point recombination types are implemented. In the both types of recombination, equal length substrings are exchanged (reciprocal recombination); this way recombination does not change the length of the chromosomes (sequences).

Mutation is modeled by a random process of bit-flipping on the chromosome, with an user-defined probability, independently on each position along the chromosome string. In this model, mutation occurs in parallel with recombination.

4 EXAMPLE OF APPLICATION OF THE HYBRID SYSTEM

The idea of creating protein sequences by artificial evolution process, may be best explained by using a simple, practical example. This method was used for evolving transmembrane peptides, starting from random sequences. The process consists of four main stages:

1. Preparation of the data-base of known transmembrane sequences,
2. Preparation of the fitness function by training the pattern recognition neural network on the set of known transmembrane sequences,
3. Artificial evolution of initial random sequences with the neural network as a fitness function, and
4. Testing obtained transmembrane peptides using Monte Carlo simulation.

Stage 1: Preparation of the data-base.

The set of known, 21-residue long, transmembrane sequences was extracted from the protein sequence data-base. We used 2100 eucaryotic transmembrane sequences, each 21 residues long. Arginines were added on the both ends of the transbilayer sequences in order to simulate the hydrophilic effect of end groups. Twenty randomly chosen sequences obtained in this procedure were tested by the Monte Carlo - membrane peptide simula-

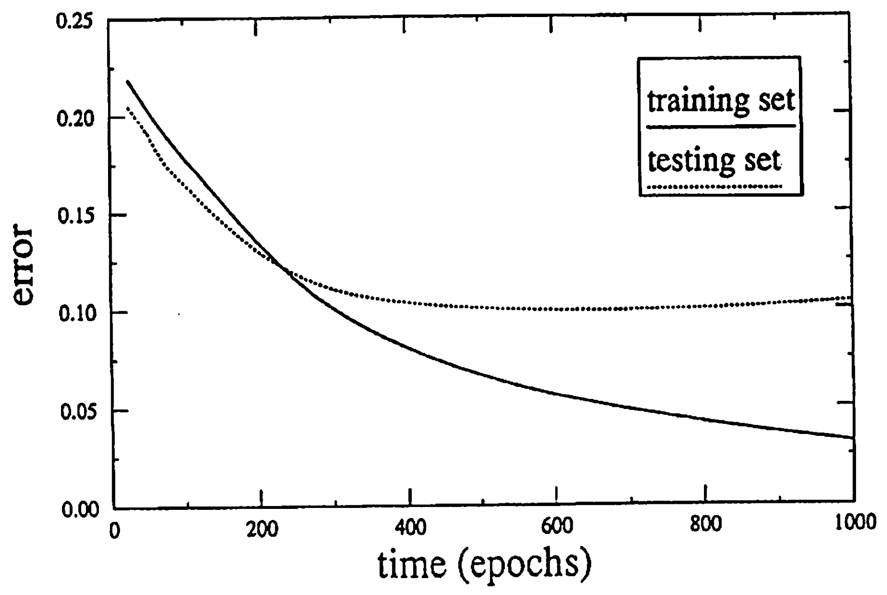

Figure 2. Learning curves for one of typical runs of the pattern recognition neural network. The error is defined according to standard definitions: mean square difference between output and target vectors. (for details see for example: Masters 1993). Difference in behaviour of these curves is probably due to memoriztion effect.

tion program (Milik and Skolnick 1993), and all of the peptides formed stable, transmembrane structures. These sequences represented the positive examples in the training procedure. An equal number of sequences of random amino acids were mixed with them as negative examples. The set of peptide sequences obtained in this way was then divided into five, equal size subsets. One of the subsets was held out as a control set. The remaining sets were used for the training and testing sets.

In order to be used as an input for the neural network system, the peptide sequences were translated into vectors of real numbers. In this translation scheme, every amino acid was represented by six of its basic physicochemical properties: hydrophobicity, bulkiness, refractivity, polarity, turn propensity and beta strand propensity (Argos 1987).

Stage 2: Training procedure.

The three-layer neural network was trained using the sets of peptide sequences depicted above. The goal of the training procedure was to obtain a neural network system that could distinguish between the natural (from data-base) and random sequences. The number of hidden-layer units and the learning procedure parameters (learning constant, momentum) were chosen in test runs. Once the architecture of the neural network system was

```
AAELXHASXESDTSHTSRRYKSH
AFNGARKXVLKFLYLVHALRLWX
AHHLRECALRLHLDYYHSITISR
AILWEPIVTENLXFIQSCLFAGV
AKRELVRLSKXSAFDVNRSPXAL
AMACPERRVCFRLNYLEESAVXT
ARITSRPCFTSHPAXLXLQVGFM
ARQDRLLIPPTFSXPVSGHSRDN
ARSLFTERSTSRQRTTQAXLPPG
AXQPCNRGSVCGIQGGPSTLSTL
AYCSEEGSXRGDWPILGGHSPAA
CDCFVKVDLKVDKRCASLLGXAV
CRAQNDTGXSLCLENHNLGDVSW
CSHSTSSSQHRHQSXYEWSSSSA
DCLVSKNIGGRYGRPSKGLSIPX
DRKHPLQRLLTVFCWYYXTDWKI
DTNEPADLPVXCGDLIMIIAIAG
DYLNXFGVEKKVGIFXAVAGGGT
EHDGVTTHRMICKARWPGHSMRA
EPLLIQRQXHTPPTYKVTRPKFL
ESCRTFXIKSHAPLYPPSIDEGR
EVPRPTDPPAGCASXIPARESQF
FDSTDGNFCYWMYLIFHSKHPMR
FKXKXVRFREACVFDYXVNHHSS
FMPAHYSCDXARSSRTSQSDLXQ
FNAASHLRRDRAPLELIEAYLIH
FQGTFPLCSRQWYGVILSRCVSP
FSRQREELLCVTRLLHLTAVAMV
GEEFQLITLQVSAITHRCTMVQP
GEFYPPEPLTFINLLPSYSIYVT
```

Figure 2. A small fragment of the initial random population of amino acid sequences. Sequences are obtained by interpretation of random sequences of "0" and "1" using the code from Table 1. The initial population was sorted alphabetically before printing. Amino acids are represented by their one-letter codes, "X" denotes "stop" signals and were not interpreted in the present simulation. Sequences obtain large penalties for the "X" code, therefore these mutation are "lethal".

set, the main training process was started. The process consisted of four independent runs, starting from different, random initial weights. The learning curves from one of the typical runs are shown in Figure 1. The figure shows the error of the neural network for training and testing sets during the learning procedure. The error on the training set decreases monotonically during the training procedure. The error calculated on the testing set decreases in the initial stage of the process, but later it stabilizes and even increases. This is probably an effect of memorization of the training set by neural network. The neural network with minimal error value for the control set was used as a fitness function in the artificial evolution procedure.

```
HACLSCYGPIGVNLRIVSCLVGR
HAIIIRYFGLVIILLTPRLSGAK
HAIQEHAIYVLAALRIGSILGMK
HAIQEHVIYVCAAQRIVSILGMK
HAIQEHVIYVLAALRIGCILGVK
HAIQEHVIYVLAALRIVSILGMK
HAIQEHVIYVLAALRIVSILGMR
HAIQEHVIYVLLDLGIVSILGMK
HAIVICFIGQVSGLGRYEVVLIR
HAIVQIDASHDGTTLIVSILGMK
HAIVQMDIYVLAPLRIVSILVMN
HAIWEHVIYVLAPLSCLSILGMK
HAIWEHVIYVLARLRIVSILGMK
HAIWEHVIYVLTRLRIVSTLETK
HAIWEHVIYVLVRLRIVSILGMK
HALASNVIYDLAALRIVSILGMK
HALASNVIYDLAALRIVSILGMK
HALASNVIYNLAALRIVSILGLK
HARNVRFYLLRARLRIVSILGMK
HATWEHVINVLLNLRIVSILEMK
HATWEHVIYVLAALRIVSILAMK
HATWEHVIYVLAALRIVSILEMK
HATWEHVIYVLAPLRIVSILGIR
HATWEHVIYVLARLRIVSILGMK
HATWQHVIYVLARLRIISILYWH
HATWQHVIYVLLDLRIVSILEMK
HATWQHVIYVLLMQRLVSILEMK
HAVHEHVMYVIAALRIVSSGICQ
HAVWEHVIYVLARLRIISILGMK
HGCPSFKIYVLGRLRIVSTLVGR
```

Figure 3. A small fragment of the final population of transmembrane amino acid sequences. These peptide sequences were obtained in the process of artificial evolution with the pattern recognition neural network as a fitness function.

Stage 3: Artificial Evolution.
The process of the artificial evolution started from a population of 1500 random peptide sequences. Figure 2 shows a fragment of this population. A set of 20 sequences was randomly chosen from the starting population and was tested by our Monte Carlo membrane peptide simulation procedure (Milik and Skolnick 1993). No sequences from this set could form a stable transmembrane structure in the simulation process. Figure 3 shows a fragment of population after 100 epochs of the process of artificial evolution, using the trained pattern-recognition neural network as a fitness function. The process generated a diversified population of peptide sequences with high fitness values. Most of the sequences are different from the se-

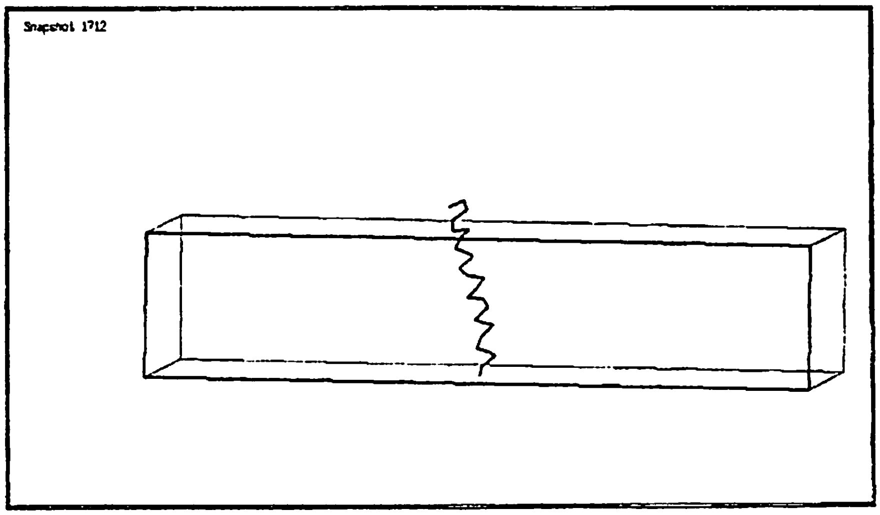

Figure 4. Example of a typical transmembrane structure formed by the peptide: RAIVICHIGRLCAVTEYTVFIVH (fitness: 0.998) obtained in the process of artificial evolution. The sequence of the peptide is not identical with any natural peptide from training or testing data-base. The figure shows the helical structure of the peptide in schematic Cα representation, spanning the model, hydrophobic membrane. The borders of the model membrane are represented by parallelograms. More about Monte Carlo method used in this paper can be find in work: Milik and Skolnick 1993.

quences used in the process of training of the neural network system. Twenty, randomly chosen sequences from the final population were tested in the Monte Carlo procedure, and all of them form transmembrane structures. Figure 4 shows an example of the typical transmembrane structure formed by a peptide generated in the Artificial Evolution process.

5. CONCLUSIONS

This paper presents a proposition of a method of implementation of the artificial evolution methods into the area of modeling of membrane-peptide and protein-peptide systems. This implementation, in cooperation with Monte Carlo simulation, may open up the possibility for development of new methods for the rational design of amino-acid sequences with desirable properties.

Additionally, the paper presents a set of object-oriented structures, which could be used in another artificial evolution and artificial life simulations. This set of programing tools is a basis for proposed object oriented environment for implementation of the artificial evolution methods into the area of protein and nucleic acid analysis.

Acknowledgments

The authors would like to thank Dr. David B. Fogel for his criticism and valuable comments.

References

Argos P. (1987). A Sensitive Procedure to Compare Amino Acid Sequences, *J.Mol.Biol.* 193: 385.

Masters T. (1993). *Practical Neural Network Recipes in C++*: Academic Press.

Milik M. and Skolnick J. (1993). Insertion of Peptide Chains into Lipid Membranes: An Off-Lattice Monte Carlo Dynamics Model. *Proteins*, 15:10.