Recognizing Patterns in Protein Sequences
Using Iteration-Performing Calculations in Genetic Programming

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305-2140 USA
Koza@CS.Stanford.Edu
Phone: 415-941-0336, Fax.: 415-941-9430

Abstract: *This paper uses genetic programming with automatically defined functions (ADFs) for the dynamic creation of a pattern-recognizing computer program consisting of initially-unknown detectors, an initially-unknown iterative calculation incorporating the as-yet-undiscovered detectors, and an initially-unspecified final calculation incorporating the results of the as-yet-unspecified iteration. The program's goal is to recognize a given protein segment as being a transmembrane domain or non-transmembrane area of the protein. Genetic programming with automatic function definition is given a training set of differently-sized mouse protein segments and their correct classification. Correlation is used as the fitness measure. Automatic function definition enables genetic programming to dynamically create subroutines (detectors). A restricted form of iteration is introduced to enable genetic programming to perform calculations on the values returned by the detectors. When cross-validated, the best genetically-evolved recognizer for transmembrane domains achieves an out-of-sample correlation of 0.968 and an out-of-sample error rate of 1.6%. This error rate is better than that recently reported for five other methods.*

1. Statement of the Problem

The goal in this paper is to use genetic programming with automatically defined functions (ADFs) to create a computer program for recognizing a given subsequence of amino acids in a protein as being a transmembrane domain or a non-transmembrane area of the protein. Genetic programming will be given a set of differently-sized protein segments and the correct classification for each segment. The yet-to-be evolved recognizing program will consist of initially-unspecified detectors, an initially-unspecified iterative calculation incorporating the as-yet-undiscovered detectors, and an initially-unspecified final calculation incorporating the results of the as-yet-undiscovered iteration to produce the program's output.

Although genetic programming does not know the chemical characteristics or biological meaning of the sequence of amino acids appearing in the protein segment, we will show that the results have an interesting biological interpretation. Of course, the reader may ignore the biological interpretation and view this problem as a one-dimensional pattern recognition problem.

This problem is described in Weiss, Cohen, and Indurkhya (1993).

2. Biological Background

Proteins are polypeptide molecules composed of sequences of amino acids. There are 20 amino acids (also called residues) in the alphabet of proteins, and they are denoted by the letters A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, and Y. The biological structure and function of a protein is determined by its 3D structure and the locations of its atoms. This structure is in turn determined by the protein's sequence of amino acids.

A transmembrane protein is a protein that finds itself embedded in a membrane (e.g., a cell wall) in such a way that part of the protein is located on one side of the membrane, part is within the membrane, and part is on the opposite side of the membrane. The membrane involved may be a cellular membrane or other type of membrane. Transmembrane proteins often cross back and forth through the membrane several times and have short loops immersed in the different milieu on each side of the membrane. The length of each transmembrane domain and each loop or other non-transmembrane area are usually different. Transmembrane proteins perform functions such as sensing the presence of certain chemicals or certain stimuli on one side of the membrane and transporting chemicals or transmitting signals to the other side of the membrane. Understanding the functions of these biologically important proteins requires identification of their transmembrane domains. An average protein contains around 300 residues and a typical transmembrane domain contains around two or three dozen residues.

Biological membranes are of hydrophobic (water-hating) composition. Since the amino acid residues in a transmembrane domain of the most common type are exposed to the membrane, these residues have a pronounced, but not overwhelming, tendency to be hydrophobic. This tendency toward hydrophobicity is an overall characteristic of the entire segment (not of any particular one or two amino acids or any particular part of the segment).

Figure 1 shows the 161 amino acid residues of *mouse peripheral myelin protein 22*, one of approximately 30,000 proteins in the SWISS-PROT computerized database of proteins (Bairoch and Boeckmann 1991). The four transmembrane domains of this protein are boxed and located at residues 2–31, 65–91, 96–119, and 134–156.

```
MLLLLLGILF LHIAVLVLLF VSTIVSQWLV GNGHttdlwq ncttsalgav    50

qhcysssvse wLQSVQATMI LSVIFSVLAL FLFFCQLFTL TKGGRFYITG   100

FFQILAGLCV MSAAAIYTVR HSEWHVNTDY SYGFAYILAW VAFPLALLSG   150

IIYVILRKRE L                                             161
```

Figure 1 Primary sequence of mouse peripheral myelin protein 22.

A successful recognizing program would identify a segment such as the 24-residue segment between positions 96–119, namely

FYITGFFQILAGLCVMSAAAIYTV (1)

as a transmembrane domain. Segment (1) is boxed and in upper case in figure 1.

A successful recognizer would also identify the 27-residue segment between positions 35–61, namely

TTDLWQNCTTSALGAVQHCYSSSVSEW (2)

as being a non-transmembrane area. Segment (2) is underlined and in lower case in figure 1.

The recognition problem will be solved without reference to any knowledge about the hydrophobicity of the 20 amino acids; however, we will use such knowledge to explain the problem and interpret the results. With the benefit of the Kyte-Doolittle scale for hydrophobicity (Kyte and Doolittle 1982), we can, however, see that two thirds of the 24 residues of segment (1) are in the hydrophobic category (containing I, V, L, F, C, M, or A), seven of the 24 residues (two Gs, two Ts, two Ys, and one S) are in the neutral category (containing G, T, S, W, Y, P), and only one residue (the Q at position 103) is in the hydrophilic category (containing H, Q, N, E, D, K, R). Even through there are some residues from all three categories in both segments, segment (1) is predominantly hydrophobic and is, in fact, a transmembrane domain.

In contrast, 13 of the 27 (about half) of the residues of segment (2) are neutral, eight (about a quarter) are hydrophobic, and six (about a quarter) are hydrophilic. This segment is not as strongly hydrophobic as segment (1) and is, in fact, a non-transmembrane area.

3. Background on Genetic Programming

Genetic programming (Koza 1992, Koza and Rice 1992) is an extension of the genetic algorithm (Holland 1975) in which the genetic population consists of computer programs (that is, compositions of primitive functions and terminals). Recent work in genetic programming is described in Kinnear 1994. Automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs in the population (Koza 1992, 1994; Koza and Rice 1992, 1994). Each program in the population contains one (or more) function-defining branches and one (or more) main result-producing branches.

4. Preparatory Steps

4.1 Architecture of the Overall Program

In applying genetic programming with automatically defined functions to a problem, we begin by deciding that the overall architecture of the yet-to-be-evolved recognizing program will have to be capable of categorizing the residues into useful categories, then iteratively performing some arithmetic calculations and conditional operations on the categories, and finally performing some arithmetic calculations and conditional operations to reach a conclusion. This suggests an overall architecture for the recognizing program of several (say three) automatically defined functions (detectors) for categorization, an iteration-performing branch for performing arithmetic operations and conditional operations for examining the residues of the protein segment using the as-yet-undiscovered detectors, and a result-producing branch for performing arithmetic operations and conditional operations for reaching a conclusion using the as-yet-undiscovered iteration.

4.2 Functions and Terminals of the Function-Defining Branches

Automatically defined functions seem well suited to the role of dynamically defining categories of the amino acids. If the automatically defined functions are to play the role of set formation, each defined function should be able to

interrogate the current residue as to which of the 20 amino acids it is. Since we anticipate that some numerical calculations will subsequently be performed on the result of the categorization of the residues, it seems reasonable to employ numerical-valued logic, as we often do in genetic programming (Koza 92), rather than Boolean-valued logic returning the non-numerical values of True and False). One way to implement this approach is to define 20 numerical-valued zero-argument logical functions for determining whether the residue currently being examined is a particular amino acid. For example, `(A?)` is the zero-argument residue-detecting function returning a numerical +1 if the current residue is alanine (A) but otherwise returning a numerical –1. A similar residue-detecting function is defined for each of the 19 other amino acids. Since we envisage that the automatically defined functions will be used for set formation, it seems reasonable to include the logical disjunctive function in the function set of the automatically defined functions. Specifically, `ORN` is the two-argument numerical-valued disjunctive function returning +1 if either or both of its arguments are positive, but returning –1 otherwise.

The terminal set $\mathcal{T}_{fd}$ for each of the three function-defining branches, `ADF0`, `ADF1`, and `ADF2`, contains the 20 zero-argument numerical-valued residue-detecting functions. That is,

$\mathcal{T}_{fd}$ = {(A?), (C?), ... , (Y?)}.

The function set $\mathcal{F}_{fd}$ for the three function-defining branches `ADF0`, `ADF1`, and `ADF2` contains only the two-argument numerically-valued logical disjunctive function. That is,

$\mathcal{F}_{fd}$ = {ORN}.

4.3 Functions and Terminals of the Iteration-Performing Branch

Typical computer programs contain iterative operators that perform some specified work until some condition expressed by a termination predicate is satisfied. When we attempt to include iterative operators in genetically-evolved programs, we face the practical problem that both the work and the termination predicate are initially created at random and are subsequently subject to modification by the crossover operation. Consequently, iterative operators will, at best, be nested and consume enormous amounts of computer time or will, at worst, have unsatisfiable termination predicates and go into infinite loops. This problem can sometimes be partially, although arbitrarily, alleviated by imposing time-out limits (e.g., on each iterative loop individually and all iterative loops cumulatively).

In problems where we can envisage one iterative calculation being usefully performed over a particular known, finite set, there is an attractive alternative to permitting unrestricted evolution of iterative programs. For such problems, the iteration can be restricted to exactly one iteration over the finite set. The termination predicate of the iteration is fixed and is not subject to evolutionary modification. Thus, there is no nesting and there are no infinite loops.

In the case of problems involving the examination of the residues of a protein, iteration can very naturally be limited to the ordered set of amino acid residues of the protein segment involved. Thus, for this problem, there can be one iteration-performing branch, with the iteration restricted to the ordered set of amino acid residues in the protein segment. That is, each time iterative work is performed, the current residue of the protein is advanced to the next residue of the protein segment until the end of the entire protein segment is encountered. The result-producing branch produces the final output of the overall program.

Useful iterative calculations typically require both an iteration variable and memory (state). That is, the nature of the work typically varies depending on the current value of the iteration variable and memory is typically required to transmit information from one iteration to the next. In this problem the same work is executed as many times as there are residues in a protein segment, so the iteration variable is the residue at the current position in the segment. Depending on the problem, the iteration variable may be explicitly available or be implicitly available through functions that permit it to be interrogated. For this problem, the automatically defined functions provide a way to interrogate the residues of the protein sequence.

Memory can be introduced into any program by means of settable variables, `M0`, `M1`, `M2`, and `M3`. Settable variables are initialized to some appropriate value (e.g., zero) at the beginning of the execution of the iteration-performing branch. These settable variables may change as a result of the iteration.

The terminal set $\mathcal{T}_{ipb0}$ for the iteration-performing branch is

$\mathcal{T}_{ipb0}$ = {LEN, M0, M1, M2, M3, ←},

where ← represents floating-point random constants between –10.000 and +10.000 with a granularity of 0.001 and where `LEN` is the length of the current protein segment.

Since we envisage that the iteration-performing branch will perform calculations and make decisions based on these calculations, it seems reasonable to include the four arithmetic operations and a conditional operator in the

function set. We have used the four arithmetic functions (+, -, *, and %) and the conditional comparative operator IFLTE (If Less Than or Equal) on many previous problems, so we include them in the function set for the iteration-performing branch. The protected division function % takes two arguments and returns one when division by 0 is attempted (including 0 divided by 0), and, otherwise, returns the normal quotient. The four-argument conditional branching function IFLTE evaluates and returns its third argument if its first argument is less than or equal to its second argument and otherwise evaluates and returns its fourth argument.

Since a numerical calculation is to be performed on the results of the categorization performed by the function-defining branches, ADF0, ADF1, and ADF2 are also included in the function set for the iteration-performing branch.

We need a way to change the settable variables M0, M1, M2, and M3. The one-argument setting function SETM0 can be used to set M0 to a particular value. Similarly, the setting functions SETM1, SETM2, and SETM3 can be used to set the respective values of the settable variables M1, M2, and M3. Thus, memory can be written (i.e., the state can be set) with the setting functions, SETM0, SETM1, SETM2, and SETM3, and memory can be read (i.e., the state can be interrogated) merely by referring to the terminals, M0, M1, M2, and M3.

Thus, the function set $\mathcal{F}_{\text{ipb0}}$ for the iteration-performing branch IPB0 is

$\mathcal{F}_{\text{ipb0}}$ = {ADF0, ADF1, ADF2, SETM0, SETM1, SETM2, SETM3, IFLTE, +, -, *, %}.

taking 0, 0, 0, 1, 1, 1, 1, 4, 2, 2, 2, and 2 arguments, respectively.

4.4 Functions and Terminals of the Result-Producing Branch

The result-producing branch can then perform a non-iterative floating-point calculation and produce the final result of the overall program. The settable variables M0, M1, M2, and M3 provide a way to pass the results of the iteration-performing branch on to a result-producing branch.

The terminal set $\mathcal{T}_{\text{rpb0}}$ for the result-producing branch RPB0 is

$\mathcal{T}_{\text{rpb0}}$ = {LEN, M0, M1, M2, M3, ←}.

The function set $\mathcal{F}_{\text{rpb0}}$ for the result-producing branch is

$\mathcal{F}_{\text{rpb0}}$ = {IFLTE, +, -, *, %}

taking 4, 2, 2, 2, and 2 arguments, respectively.

A wrapper is used to convert the floating-point value produced by the result-producing branch into a binary outcome. If the genetically-evolved program returns a positive value, the segment will be classified as a transmembrane domain, but otherwise it will be classified as a non-transmembrane area.

4.5 Fitness Measure

Fitness will measure how well a particular genetically-evolved recognizing program predicts whether the segment is, or is not, transmembrane domain. Fitness is measured over a number of trials, which we call fitness cases. The fitness cases for this problem consist of protein segments.

When a genetically-evolved recognizing program in the population is tested against a particular fitness case, the outcome can be a true-positive, true-negative, false-positive, or false-negative. Fitness can be measured by the correlation coefficient C. When the predictions and observations each take on only two possible values, correlation is a general, and easily computed, measure for evaluating the performance of a recognizing program. Consider a vector in a space of dimensionality N_{fc} of the correct answers (with the integer 1 representing a transmembrane domain and the integer 0 representing a non-transmembrane area) and the vector of length N_{fc} of the predictions (1 or 0) produced by a particular genetically evolved program. The correlation C is the cosine of the angle in this space of dimensionality N_{fc} between the vector of correct answers and the vector of predictions. The correlation coefficient indicates how much better a particular predictor is than a random predictor. A correlation C of –1.0 indicates vectors pointing in opposite directions in N_{fc}-space (i.e., greatest negative correlation); a correlation of +1.0 indicates coincident vectors (i.e., greatest positive correlation); a correlation C of 0.0 indicates orthogonal vectors (i.e., no correlation).

The correlation C lends itself immediately to being the measure of raw fitness measure for a genetically evolved computer program. Since raw fitness ranges between –1.0 and +1.0 (higher values being better), standardized fitness ("zero is best") can then be defined as

$$\frac{1-C}{2}.$$

Standardized fitness ranges between 0.0 and +1.0, lower values being better and a value of 0 being the best. Thus, a standardized fitness of 0 indicates perfect agreement between the predicting program and the observed reality; a

standardized fitness of +1.0 indicates perfect disagreement; a standardized fitness of 0.50 indicates that the predictor is no better than random.

The *error rate* is the number of fitness cases for which the recognizing program is incorrect divided by the total number of fitness cases. The error rate is a less general measure of performance for a recognizing program and copes less well with disparities in the numbers of positive and negative features than correlation; however, Weiss, Cohen, and Indurkhya (1993) use the error rate as their yardstick for comparing three methods in the biological literature with a new algorithm of their own created using the SWAP-1 induction technique. Therefore, we present our final results in terms of both correlation and error rate and we use error rate for the purpose of comparing results.

4.6 Fitness Cases

Release 25 of the SWISS-PROT protein data base contains 248 mouse transmembrane proteins averaging 499.8 residues in length. Each protein contains between one and 12 transmembrane domains, the average being 2.4. The transmembrane domains range in length from 15 and 101 residues and average 23.0 in length.

123 of the 248 proteins were arbitrarily selected to create the in-sample set of fitness cases to measure fitness during the evolutionary process. One of the transmembrane domains of each of these 123 proteins was selected at random as a positive fitness case for this in-sample set. One segment of the same length as a random one of the transmembrane segments that is not contained in any of the protein's transmembrane domains was selected from each protein as a negative fitness case. Thus, there are 123 positive and 123 negative fitness cases in the in-sample set of fitness cases.

The evolutionary process is driven by fitness as measured by the set of in-sample fitness cases. However, the true measure of performance for a recognizing program is how well it generalizes to different cases from the same problem environment. Thus, 250 out-of-sample fitness cases (125 positive and 125 negative) were created from the remaining 125 proteins in a manner similar to the above. These out-of-sample fitness cases were then used to validate the performance of the genetically-evolved recognizing programs.

Population size, *M*, was 4,000. The maximum number of generations to be run, *G*, was set to 21. The other parameters for controlling the runs of genetic programming were the default values specified in Koza (1994) and which have been used for a number of different problems.

5. Results

We now describe the best run out of 11 runs of this problem, that produced the best value of out-of-sample correlation of any run, namely 0.968. This high correlation was achieved on generation 20 by a program with an in-sample correlation of 0.976 resulting from getting 121 true positives, 122 true negatives, 1 false positive, and 2 false negatives over the 246 in-sample fitness cases. Its out-of-sample correlation of 0.968 is the result of getting 123 true positives, 123 true negatives, 2 false positives, and 2 false negatives over the 250 out-of-sample fitness cases. Its out-of-sample error rate is only 1.6%. This program consists of 105 points (i.e., functions and terminals in the bodies of the various branches) and is shown below:

```
(progn (defun ADF0 ()
         (values (ORN (ORN (ORN (I?) (H?)) (ORN (P?) (G?))) (ORN (ORN (ORN (Y?) (N?))
           (ORN (T?) (Q?))) (ORN (A?) (H?))))))
       (defun ADF1 ()
         (values (ORN (ORN (ORN (A?) (I?)) (ORN (L?) (W?))) (ORN (ORN (T?) (L?)) (ORN
           (T?) (W?))))))
       (defun ADF2 ()
         (values (ORN (ORN (ORN (ORN (ORN (D?) (E?)) (ORN (ORN (ORN (D?) (E?)) (ORN
           (ORN (T?) (W?)) (ORN (Q?) (D?)))) (ORN (K?) (P?)))) (ORN (K?) (P?))) (ORN
           (T?) (W?))) (ORN (ORN (E?) (A?)) (ORN (N?) (R?))))))
       (progn (loop-over-residues (SETM0 (+ (- (ADF1) (ADF2)) (SETM3 M0))))
              (values (% (% M3 M0) (% (% (% (- L -0.53) (* M0 M0)) (+ (% (% M3 M0) (%
         (+ M0
                         M3) (% M1 M2))) M2)) (% M3 M0))))))
```

Ignoring the three residues common to the definition of both `ADF1` and `ADF2`, `ADF1` returns 1 if the current residue is I or L and `ADF2` returns 1 if the current residue is D, E, K, R, Q, N, or P. I and L are two of the seven hydrophobic residues on the Kyte-Doolittle scale. D, E, K, R, Q, and N are six of the seven hydrophilic residues and P is one of the neutral residues.

In the iteration-performing branch, `M0` is the running sum of the differences of the values returned by `ADF1` and `ADF2`. `M0` will be positive only if the hydrophobic residues in the protein segment are so numerous that the occurrences of I and L outnumber the occurrences of the six hydrophilic residues and one neutral residue of `ADF2`.

M3 is the same as the accumulated value of M0 except that M3 lags M0 by one residue. Because the contribution to M3 in the iteration-performing branch of the last residue is either 0 or 1, M3 is either equal to M0 or M3 is one less than M0.

The result-producing branch is equivalent to

$$\frac{M_3^3}{M_0(M_0 + M_3)(Len + 0.53)}$$

The factor (- LEN -0.53) is always positive and therefore can be ignored in determining whether the result-producing branch is positive or non-positive. Because of the close relationship between M0 and M3, analysis shows that the result-producing branch identifies a protein segment as a transmembrane domain whenever the running sum of the differences, M0, is greater than zero, except for the special case when M0 = 1 and M3 = 0. This special case occurs only when the running values of M0 and M3 are tied at zero and when the very last residue of the protein segment is I or L (i.e., ADF1 returns 1).

Ignoring this special case, the operation of this overall best program can be summarized as follows: If the number of occurrences of I and L in a given protein segment exceeds the number of occurrences of D, E, K, R, Q, N and P, then classify the segment as a transmembrane domain; otherwise, classify it as non-transmembrane.

Table 1 shows the error rate for the four algorithms for recognizing transmembrane domains reviewed in Weiss, Cohen, and Indurkhya (1993) as well as the out-of-sample error rate of the best genetically-evolved program from the run described above. As can be seen, the error rate of the genetically-evolved program is the best of the five methods in the table.

Table 1 Comparison of five methods.

Method	Error rate
von Heijne 1992	2.8%
Engelman, Steitz, and Goldman 1986	2.7%
Kyte-Doolittle 1982	2.5%
Weiss, Cohen, and Indurkhya 1993	2.5%
Best genetically-evolved program	1.6%

6. Conclusions

Genetic programming with automatically defined functions was able to evolve a pattern-recognizing computer program for transmembrane domains in proteins consisting of initially-unspecified detectors, an initially-unspecified iterative calculation incorporating the as-yet-undiscovered detectors, and an initially-unspecified final calculation incorporating the results of the as-yet-undiscovered iteration.

7. Acknowledgements

James P. Rice of the Knowledge Systems Laboratory at Stanford University did the computer programming of the above on a Texas Instruments Explorer II$^+$ computer.

8. Bibliography

Bairoch, A. and Boeckmann, B. 1991. The SWISS PROT protein sequence data bank. *Nucleic Acids Research* 19: 2247–2249.

Engelman, D., Steitz, T., and Goldman, A. 1986. Identifying nonpolar transbilayer helices in amino acid sequences of membrane proteins. *Annual Review of Biophysics and Biophysiological Chemistry*. Volume 15.

Holland, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press 1975. Also Cambridge, MA: The MIT Press 1992.

Kinnear, K. E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge: Cambridge, MA: The MIT Press.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J. R. 1994. *Genetic Programming II*. Cambridge, MA: The MIT Press.

Koza, J. R., and Rice, J. P. 1992. *Genetic Programming: The Movie*. Cambridge, MA: The MIT Press.

Koza, J. R., and Rice, J. P. 1994. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: The MIT Press.

Kyte, J. and Doolittle, R. 1982. A simple method for displaying the hydropathic character of proteins. *Journal of Molecular Biology*. 157:105-132.

von Heijne, G. Membrane protein structure prediction: Hydrophobicity analysis and the positive-inside rule. *Journal of Molecular Biology*. 225:487–494.

Weiss, S. M., Cohen, D. M., and Indurkhya, N. 1993. Transmembrane segment prediction from protein sequence data. In Hunter, L., Searls, D., and Shavlik, J. (editors). *Proceedings of the First International Conference on Intelligent Systems for Molecular Biology*. Menlo Park, CA: AAAI Press.