Version 5 – Chapter 8 – November 14, 1995 for *Advances in Genetic Programming II* (1996) edited by Peter J. Angeline and Kenneth E. Kinnear, Jr.

Classifying Protein Segments as Transmembrane Domains Using Architecture-Altering Operations in Genetic Programming

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305-2140 USA
EMAIL: Koza@CS.Stanford.Edu
PHONE: 415-941-0336
FAX: 415-941-9430
WWW: http://www-cs-faculty.stanford.edu/~koza/

David Andre
Visiting Scholar
Computer Science Department
Stanford University
860 Live Oak Ave, #4
Menlo Park, CA 94025 USA
EMAIL: andre@flamingo.stanford.edu
PHONE: 415-326-5113
WWW: http://www-leland.stanford.edu/~phred/

John R. Koza and David Andre

The biological theory of gene duplication, concerning how new structures and new behaviors are created in living things, is brought to bear on the problem of automated architecture discovery in genetic programming. Using architecture-altering operations patterned after naturally-occurring gene duplication, genetic programming is used to evolve a computer program to classify a given protein segment as being a transmembrane domain or non-transmembrane area of the protein. The out-of-sample error rate for the best genetically-evolved program achieved was slightly better than that of previously-reported human-written algorithms for this problem. This is an instance of an automated machine learning algorithm rivaling a human-written algorithm for a problem.

8.1 Introduction and Overview

The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem.

Ideally, an automatic programming system should require that the user pre-specify as little as possible about the problem environment. In particular, it is desirable that the user not be required to specify the architecture of the ultimate solution to his problem before he can begin to apply the technique to his problem. The requirement that the human user prespecify the size and shape of the ultimate solution to his problem has been a vexatious aspect of automated machine learning from the earliest times [Samuel 1959]. The size and shape of the solution should be part of the *answer* provided by an automated machine learning technique, rather than part of the *question* supplied by the human user.

John Holland's pioneering *Adaptation in Natural and Artificial Systems* [1975] described how an analog of the naturally-occurring evolutionary process can be applied to solving artificial problems using what is now called the *genetic algorithm*. The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992] describes an extension of Holland's genetic algorithm in which the genetic population consists of computer programs (that is, compositions of primitive functions, terminals, and possibly automatically defined functions). See also Koza and Rice [1992]. Genetic programming has been demonstrated to be capable of evolving computer programs that solve, or approximately solve, a variety of problems from various fields, including many problems that have been used over the years as benchmark problems in machine learning and artificial intelligence. A run of genetic programming in its most basic form automatically creates the size and shape of a single-part main program as well as the sequence of work-performing steps in the main program.

No approach to automated programming is likely to be successful on non-trivial problems unless it provides some hierarchical mechanism to exploit – by reuse and parameterization – the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines provide this mechanism in ordinary computer programs. *Genetic Programming II: Automatic Discovery of Reusable Programs* [Koza 1994a, 1994b] deals with multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-callable subprograms.

An *automatically defined function* is a function (i.e., subroutine, procedure, `DEFUN`, module) that is dynamically evolved during a run of genetic programming in association with a particular individual program in the population and which may be invoked by a calling program (e.g., a main program) that is concurrently being evolved.

Figure 8.1 shows a hypothetical overall program consisting of a result-producing main branch (`RPB`) in the right half of the figure and one function-defining branch in the left part of the figure. The function-defining branch defines a two-argument automatically defined function (called `ADF0`). The main program (`RPB`) invokes the subroutine (`ADF0`) from two different places (the points labeled 481 and 487 in the figure) using different pairs of arguments for the two calls. The two arguments for the first call of `ADF0` are the terminal `D1` at 482 and the terminal `D2` at 483. The arguments for the second call of `ADF0` are the terminal `D3` at 488 and the sub-expression `(NOR D4 D0)` rooted at 489. The function-defining branch (`ADF0`) has an argument list at 412 consisting of two dummy arguments (formal parameters), `ARG0` and `ARG1`, and a five-point work-performing body starting at `OR` 420. The dummy arguments, `ARG0` and `ARG1`, appear in the body of the function-defining branch; however, when the subroutine `ADF0` is called by the main result-producing branch, these dummy arguments are instantiated with particular values (e.g., `D3` and `(NOR D4 D0)` for the call to `ADF0` at 487).

Various other formulations for subroutine creation in genetic programming have been created and analyzed by Angeline [1994], Angeline and Pollack [1994], Kinnear [1994], Rosca and Ballard [1994a, 1994b], and Spector [1996, in chapter 7 of this volume].

When programs with automatically defined functions are being evolved in a run of genetic programming, it becomes necessary to determine the architecture of the overall to-be-evolved program. The specification of the architecture consists of (1) the number of function-defining branches in the overall program, (2) the number of arguments (if any) possessed by each function-defining branch, and (3) if there is more than one function-defining branch, the nature of the hierarchical references (if any) allowed between each function-defining branch and the other branches of the overall program.

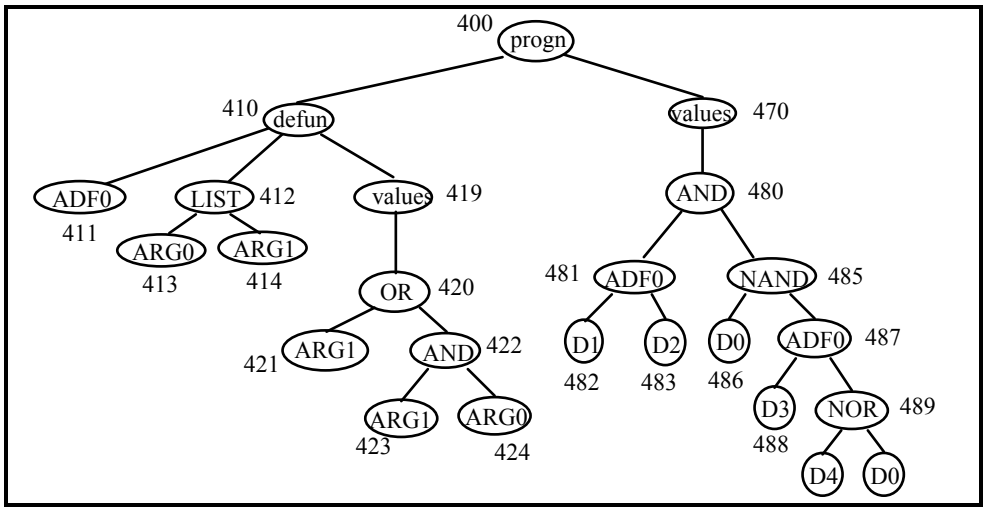

Figure 8.1
Program consisting of one two-argument function-defining branch (`ADF0`) and one result-producing branch.

The question of how to automatically create the architecture of the overall program in an evolutionary approach to automatic programming, such as genetic programming, has a parallel in the biological world: how new structures and behaviors are created in living things. Most structures and behaviors in living things are the consequence of the action and interactions of proteins. Proteins are manufactured in accordance with instructions contained in the chromosomal DNA of the organism. Simple bacteria sustain life with as few as 1,800 different proteins while humans have an estimated 100,000 proteins.

Thus, the question of how new structures and behaviors are created in living things corresponds to the question of how new DNA that encodes for a new protein is created. Ordinary recombination and mutation do not provide the answer. In nature, sexual recombination ordinarily recombines part of an existing chromosome of one parent with a corresponding (homologous) part of an existing chromosome from a second parent. Ordinary mutation occasionally alters isolated alleles belonging to a chromosome.

In his provocative 1970 book *Evolution by Gene Duplication*, Susumu Ohno points out that simple point mutation and crossover are insufficient to explain major evolutionary changes.

> "...while allelic changes at already existing gene loci suffice for racial differentiation within species as well as for adaptive radiation from an immediate ancestor, *they cannot account for large changes in evolution*,

> because large changes are made possible by the acquisition of new gene loci with previously non-existent functions." (Emphasis added).

So what is the origin of "new gene loci with previously non-existent functions"?

A gene duplication is a rare illegitimate recombination event that results in the duplication of a lengthy subsequence of a chromosome. Ohno's 1970 book proposed the provocative thesis that the creation of new proteins (and hence new structures and behaviors in living things) begins with a gene duplication and that gene duplication is "the major force of evolution."

This chapter shows how the naturally-occurring mechanism of gene duplication (and the complementary mechanism of gene deletion) motivated the addition of six new architecture-altering operations to genetic programming. The six architecture-altering operations of branch duplication, branch creation, branch deletion, argument duplication, argument creation, and argument deletion enable genetic programming to evolve the architecture of a multi-part program containing automatically defined functions *during a run* of genetic programming. The six architecture-altering operations enable genetic programming to implement what Ohno called "the acquisition of new gene loci with previously non-existent functions."

The potential usefulness of the architecture-altering operations (and the additional computational effort required by these operations) has been previously discussed in the context of simple problems, such as learning the Boolean parity functions [Koza 1994c, 1995a, 1995b, and 1995c]. In this chapter, genetic programming with the architecture-altering operations is applied to one of the more difficult problems in *Genetic Programming II*, namely the problem of evolving a computer program to classify a given protein segment as being a transmembrane domain or non-transmembrane area of the protein (without using biochemical knowledge, such as the hydrophobicity values, used in human-written algorithms for this task). The best genetically-evolved program achieves an out-of-sample error rate that is slightly better than that reported for other previously reported human-written algorithms. This is an instance of an automated machine learning algorithm rivaling human performance on a non-trivial problem.

8.2 Architecture-Altering Operations

The six architecture-altering genetic operations provide a way of evolving the architecture of a multi-part program during a run of genetic programming. These operations are performed, sparingly, during each generation of a run of genetic programming along with the usual operations of Darwinian reproduction, crossover, and mutation.

8.2.1 Branch Duplication

The operation of *branch duplication* duplicates one of the branches of a program in the following way:

1. Select a program from the population.

2. Pick one of the function-defining branches of the selected program as the branch-to-be-duplicated.

3. Add a uniquely-named new function-defining branch to the selected program, thus increasing, by one, the number of function-defining branches in the selected program. The new function-defining branch has the same argument list and the same body as the branch-to-be-duplicated.

4. For each occurrence of an invocation of the branch-to-be-duplicated anywhere in the selected program (e.g., a result-producing branch or any other branch that invokes the branch-to-be-duplicated), randomly choose either to leave that invocation unchanged or to replace that invocation with an invocation of the newly created function-defining branch.

The step of selecting a program for all the operations described herein is performed probabilistically on the basis of fitness, so that a program that is more fit has a greater probability of being selected to participate in the operation than a less fit program.

The operation of branch duplication as defined above (and all the other architecture-altering operations to be described herein) always produce a syntactically valid program (assuming that the function and terminal set originally satisfied the closure property).

Figure 8.2 shows the program resulting after applying the operation of branch duplication to the program of figure 8.1. Specifically, the function-defining branch 410 of figure 8.1 defining `ADF0` (also shown as 510 of figure 8.2) is duplicated and a new function-defining branch (defining `ADF1`) appears at 540 in figure 8.2. There are two invocations of the branch-to-be-duplicated, `ADF0`, in the result-producing branch of the selected program, namely `ADF0` at 481 and 487 of figure 8.1. For each occurrence, a random choice is made to either leave the occurrence of `ADF0` unchanged or to replace it with a reference to the newly created `ADF1`. For the first invocation of `ADF0` at 481 of figure 8.1, the choice is randomly made to replace `ADF0` 481 with `ADF1` 581 in figure 8.2. The arguments for the invocation of `ADF1` 581 are `D1` 582 and `D2` 583 in figure 8.2 (i.e., they are identical to the arguments `D1` 482 and `D2` 483 for the invocation of `ADF0` at 481 in figure 8.1). For the second invocation of `ADF0` at 487 of figure 8.1, `ADF0` is left unchanged.

The new branch is identical to the previously existing branch (except for the name `ADF1` at 541 in figure 8.2). Moreover, `ADF1` at 581 is invoked with the same arguments as `ADF0` at 481. Therefore, this operation does not affect the value returned by the overall program.

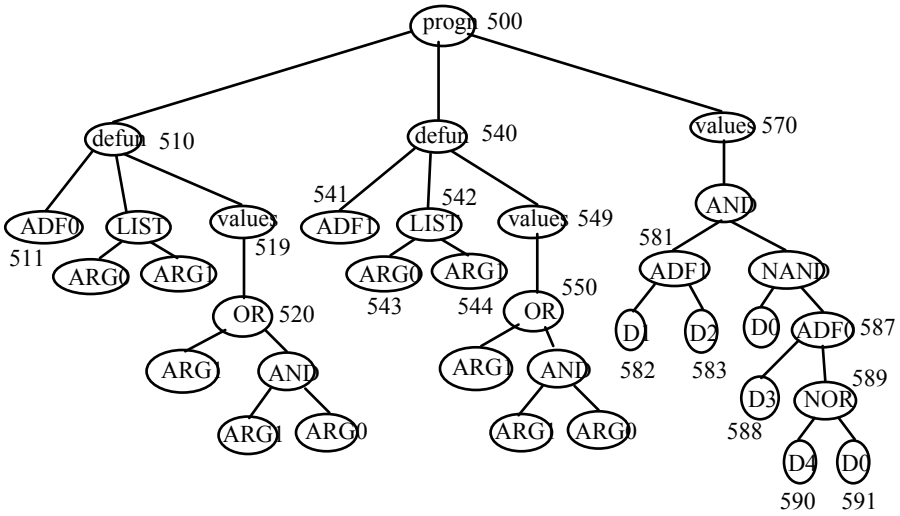

Figure 8.2
Program consisting of two two-argument function-defining branches (`ADF0` and `ADF1`) and one result-producing branch.

The operation of branch duplication can be interpreted as a *case splitting*. After the branch duplication, the result-producing branch invokes `ADF0` at 587 but `ADF1` at 581. `ADF0` and `ADF1` can be viewed as separate procedures for handling the two subproblems (cases). Subsequent genetic operations may alter one or both of these two presently-identical function-defining branches and these subsequent changes lead to a divergence in structure and behavior. This subsequent divergence may be interpreted as a *specialization* or *refinement*. That is, once `ADF0` and `ADF1` diverge, `ADF0` can be viewed as a specialization for handling for subproblem associated with its invocation at 587 and `ADF1` at 581 can be viewed as a specialization for handling its subproblem.

Details of this operation (and the other architecture-altering operations below) are found elsewhere [Koza 1994c, 1995a, 1995b, and 1995c].

8.2.2 Argument Duplication

The operation of *argument duplication* duplicates one of the dummy arguments in one of the automatically defined functions of a program in the following way:

1. Select a program from the population.

2. Pick one of its function-defining branches.

3. Choose one of the arguments of the picked function-defining branch as the argument-to-be-duplicated.

4. Add a uniquely-named new argument to the argument list of the picked function-defining branch of the selected program, thus increasing, by one, the number of arguments in its argument list.

5. For each occurrence of the argument-to-be-duplicated in the body of picked function-defining branch of the selected program, randomly choose either to leave that occurrence unchanged or to replace it with the new argument.

6. For each occurrence of an invocation of the picked function-defining branch anywhere in the selected program, identify the argument subtree corresponding to the argument-to-be-duplicated and duplicate that argument subtree in that invocation, thereby increasing, by one, the number of arguments in the invocation.

Because the function-defining branch containing the duplicated argument is invoked with an identical copy of the previously existing argument, this operation leaves unchanged the value returned by the overall program.

The operation of argument duplication can also be interpreted as a case-splitting.

8.2.3 Branch Creation

The *branch creation* operation creates a new automatically defined function within an overall program by picking a point in the body of one of the function-defining branches or result-producing branches of the selected program. This picked point becomes the top-most point of the body of the branch-to-be-created.

This operation generalizes the idea of branch duplication in that any point (not just the root) of any branch may be picked and in that the picked point may even be in the result-producing branch.

This operation of branch creation is similar to, but different from, the compression (module acquisition) operation described in Angeline [1994] and Angeline and Pollack [1994]. First, Angeline and Pollack place each new function (module) into a Genetic Library so that the new function is not specifically associated with the selected program that gave rise to it (i.e., is public, rather than private). In contrast, in the branch creation operation, the new function may be invoked only by the selected program in which it was originally created and of which it is a part. A second difference is that the body of the new branch created by the branch creation operation continues to be subject to the effects of other operations (such as crossover and mutation) in successive generations and is susceptible to continued change (i.e., the new branch is not encapsulated and protected). A third difference is that the branch creation operation may be applied to any branch (and, in particular, to function-defining branches themselves).

8.2.4 Argument Creation

The *argument creation* operation creates a new dummy argument within a function-defining branch of a program.

8.2.5 Branch Deletion

The operation of *branch deletion* deletes one of the automatically defined functions.

When a function-defining branch is to be deleted, the question arises as to how the surviving branches of the overall problem are to handle existing references to the branch-to-be-deleted. The alternative used herein (called *branch deletion with random regeneration*) randomly generates new subtrees composed of the available functions and terminals (in the same manner as the mutation operation) in lieu of the invocation of the deleted branch. This approach is, of course, not semantics-preserving.

8.2.6 Argument Deletion

The operation of *argument deletion* deletes one of the arguments to one of the automatically defined functions of a program. When an argument is deleted, references to the argument-to-be-deleted may be handled by *argument deletion with random regeneration*.

8.2.7 Structure-Preserving Crossover

When the architecture-altering operations are used, the initial population is created in accordance with a particular constrained syntactic structure. As soon as the architecture-altering operations start being used, the population quickly becomes architecturally diverse. Structure-preserving crossover with point typing [Koza 1994a] permits robust recombination while simultaneously guaranteeing that any pair of architecturally different parents will produce syntactically valid offspring.

8.3 Transmembrane Domains in Proteins

Proteins are polypeptide molecules composed of sequences of amino acids. There are 20 amino acids residues (often just called residues) in the alphabet of proteins (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) [Stryer 1995]. For example, A stands for the residue alanine.

Membranes play many important roles in living things. A *transmembrane protein* [Yeagle 1993] is embedded in a membrane (such as a cell membrane) 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. 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. Transmembrane proteins perform functions such as transporting particles across the membrane and such as sensing the presence of certain particles or stimuli on one side of the membrane and communicating that information to the other side of the membrane.

Understanding the behavior of transmembrane proteins requires identification of the portion(s) of the protein sequence that are actually embedded within the membrane, such portion(s) being called the *transmembrane domain(s)* of the protein. The lengths of the transmembrane domains of a protein are usually different from one another; the lengths of the non-transmembrane areas are also usually different from one another. Biological membranes are of oily *hydrophobic* (water-hating) composition. The amino acid residues of the transmembrane domain of a protein that are exposed to the membrane therefore have a tendency (but not an overwhelming tendency) to be hydrophobic.

The problem is to create a computer program to correctly classify a particular protein segment (i.e., a subsequence of amino acid residues extracted from the entire protein sequence) such as

FYITGFFQILAGLCVMSAAAIYTV

or

TTDLWQNCTTSALGAVQHCYSSSVSEW

as a transmembrane domain or a non-transmembrane area of the protein.

The first segment consisting of 24 residues above comes from positions 96 and 119 of the mouse peripheral myelin protein 22 and is the third (of four) transmembrane domains in that protein. The second segment (27 residues) comes from positions 35 and 61 of the same protein and is a non-transmembrane area of the protein.

Success in this problem involves integrating information over the entire protein segment. Both transmembrane domains and non-transmembrane areas of proteins contain some residues that are hydrophobic, neutral, and hydrophilic (water-loving). A correct classification cannot be made by merely examining particular position in the given protein segment, by merely testing for the presence or absence of any one particular amino acid residue, or by merely analyzing any small combination of positions within the segment.

8.4 Classifying Protein Segments as Transmembrane Domains

Automated methods of machine learning may prove to be useful in discovering biologically meaningful information hidden in the rapidly growing databases of DNA sequences and protein sequences.

We now consider the problem of deciding whether a given protein segment is a transmembrane domain or a non-transmembrane area of the protein.

Algorithms written by biologists for the problem of classifying transmembrane domains in protein sequences are based on biochemical knowledge about hydrophobicity

and other properties of membrane-spanning proteins [Kyte and Doolittle 1982; von Heijne 1992; Engelman, Steitz, and Goldman 1986]. Weiss, Cohen, and Indurkhya [1993] proposed an algorithm for this version of the transmembrane problem using a combination of human ingenuity and machine learning.

This problem provides an opportunity to illustrate several different techniques of genetic programming, including the evolution of the architecture of a multi-part computer program using the new architecture-altering operations, the automatic discovery of reusable feature detectors, the use of iteration, and the use of state (memory) in genetically evolved computer programs.

Genetic programming has previously been demonstrated its ability to evolve a classifying program to perform this same task without using any biochemical knowledge in two different ways [chapter 18 of Koza 1994a]. In both instances, the architecture of the evolved program consisted of *three zero-argument* function-defining branches consisting of an unspecified sequence of work-performing steps, one iteration-performing branch consisting of an unspecified sequence of work-performing steps that could access the as-yet-undiscovered function-defining branches, and one result-producing branch consisting of an unspecified sequence of work-performing steps that could access the results of the as-yet-undiscovered iteration-performing branch.

The question arises as to whether it is possible to solve the same problem in a run starting with a population of programs with *no* automatically defined functions at all and in which genetic programming employs the architecture-altering operations to dynamically determine an architecture that is capable of producing a satisfactory solution to the problem (i.e., perhaps three zero-argument automatically defined functions, but perhaps some other architecture).

8.4.1 Preparatory Steps

The premise behind the architecture-altering operations is that the competitive evolutionary process should be allowed to select the best architecture for the problem. We use the "minimalist" approach in which each program in the initial random population (generation 0) has no automatically defined functions.

Figure 8.3 shows the common "minimalist" structure for generation 0 in which each program in the population consists of an iteration-performing branch, `IPB`, and a result-producing branch, `RPB`, but no automatically defined functions (`ADF`s).

When a given program is executed, its iteration-performing branch is executed first. Then, its result-producing branch is executed. The result-producing branch communicates with the iteration-performing branch through memory (state) variables. The result-producing branch may call one or more of the automatically defined functions that may have been created by the architecture-altering operations.

Figure 8.3
Common structure of all programs for generation 0 consisting of one iteration-performing branch, IPB, and one result-producing branch, RPB.

A wrapper (output interface) is used to convert the floating-point value produced by the result-producing branch into a binary outcome. If the result-producing branch returns a positive numerical value, the segment will be classified as a transmembrane domain, but otherwise the segment will be classified as a non-transmembrane area of the protein.

There are a total of 38 functions and terminals in the function set and the terminal set of this problem, including 7 initial functions, 23 initial terminals, 4 potential functions, and 4 potential terminals.

8.4.1.1 Function Set

When the architecture-altering operations are used, both functions and terminals can migrate from one part of the overall program to another (both because of the action of the architecture-altering operations and because of the action of crossover using point typing). Consequently, we distinguish between

- the initial function set, $\mathcal{F}_{\text{initial}}$,
- the initial terminal set, $\mathcal{T}_{\text{initial}}$,
- the set of additional potential functions, $\mathcal{F}_{\text{potential}}$, and
- the set of additional potential terminals, $\mathcal{T}_{\text{potential}}$.

For this problem, the initial function set, $\mathcal{F}_{\text{initial}}$, is common to both the result-producing branch and the iteration-performing branch of each program; however, there is a specialized terminal set, $\mathcal{T}_{\text{rpb-initial}}$, for the result-producing branch, RPB, and a specialized terminal set, $\mathcal{T}_{\text{ipb-initial}}$, for the iteration-performing branch, IPB.

For purposes of creating the initial random population of individuals, the function set, $\mathcal{F}_{\text{initial}}$, for the result-producing branch, RPB, and the iteration-performing branch, IPB, of each individual program is

$$\mathcal{F}_{\text{initial}} = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\%}, \texttt{IFGTZ}, \texttt{ORN}, \texttt{SETM0}\}$$

taking 2, 2, 2, 2, 3, 2, and 1 arguments, respectively.

Here +, -, and * are the usual two-argument arithmetic functions and % is the usual protected two-argument division function.

The three-argument conditional branching operator IFGTZ ("If Greater Than Zero") evaluates and returns its second argument if its first argument is greater than or equal to zero, but otherwise evaluates and returns its third argument.

The one-argument setting function, SETM0, sets the settable memory variable, M0, to a particular value (and returns that value). This setting function operating on a specific settable variable [Koza 1992, 1994a] is the simplest kind of memory used in genetic programming. Teller's indexed memory [1994] and Andre's memory maps [1994] illustrate more complex ways of incorporating state and memory into genetic programming.

ORN is implemented as a two-argument numerically valued disjunctive function returning +1 if either or both of its arguments are positive, but returning –1 otherwise. ORN represents a short-circuiting (optimized) disjunction in the sense that if its first argument is positive, its second argument will not be evaluated (and any side-effecting function, such as SETM0, contained therein will remain unexecuted).

Note that the automatically defined functions (ADF0, ADF1, ...) and their dummy arguments (ARG0, ARG1, ...) do not appear in generation 0. However, once the architecture altering operations begin to be performed, ADF0, ADF1, ... and ARG0, ARG1, ... begin to appear in the population. For practical reasons, a maximum of four automatically defined functions, each possessing between zero and four dummy arguments, was established. Thus, the set of potential additional terminals, $\mathcal{T}_{\text{potential}}$, for this problem consists of

$$\mathcal{T}_{\text{potential}} = \{\texttt{ARG0}, \texttt{ARG1}, \texttt{ARG2}, \texttt{ARG3}\}.$$

The set of potential additional functions, $\mathcal{F}_{\text{potential}}$, consists of

$$\mathcal{F}_{\text{potential}} = \{\texttt{ADF0}, \texttt{ADF1}, \texttt{ADF2}, \texttt{ADF3}\},$$

each taking an as-yet-unknown number of arguments (between zero and four).

8.4.1.2 Terminal Set

For purposes of creating the initial random population of individuals, the terminal set, $\mathcal{T}_{\text{ipb-initial}}$, for the iteration-performing branch, IPB, is,

$$\mathcal{T}_{\text{ipb-initial}} = \{\leftarrow, \texttt{M0}, \texttt{LEN}, \texttt{(A?)}, \texttt{(C?)}, \ldots, \texttt{(Y?)}\}.$$

← represents floating-point random constants between –10.000 and +10.000 chosen using a uniform probability distribution [Koza 1992]. Since we want to encode each point (function or terminal) of each program tree in the population into one byte of memory in the computer, the number of different floating-point random constants is the difference between 256 and the total number of functions and terminals (initial and potential). These two hundred or so initial random constants are adequate for this problem because the various arithmetic functions frequently recombine them during the run to produce new constants.

M0 is a settable memory variable. It is zero when execution of a given overall program begins for a particular fitness case.

LEN represents the number of amino acid residues in the protein segment currently under consideration (i.e., the current fitness case).

(A?) represents 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 (from (C?) to (Y?)) is defined for each of the 19 other amino acids. Each time iterative work is performed by the body of the iteration-performing branch, 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. If a residue-detecting function is directly called from an iteration-performing branch, IPB (or indirectly called by virtue of being within a yet-to-be-created automatically defined function that is called by the iteration-performing branch), the residue-detecting function is evaluated for the current residue of the iteration.

For purposes of creating the initial random population of individuals, the terminal set, $\mathcal{T}_{\text{rpb-initial}}$, for the result-producing branch, RPB, is,

$$\mathcal{T}_{\text{rpb-initial}} = \{\leftarrow, \texttt{M0}, \texttt{LEN}\}.$$

Note that a residue-detecting function may migrate into the body of a yet-to-be-created automatically defined function which may, in turn, become referenced by some yet-to-be-created call from a result-producing branch. We deal with this possibility by specifying that when a residue-detecting function is executed inside a result-producing branch (or called indirectly by virtue of being inside a yet-to-be-created automatically defined function that is referenced by a yet-to-be-created call from a result-producing branch), the residue-detecting function is evaluated for the leftover value of the iterative index (i.e., the last residue of the protein segment). This treatment protects the overall program from stopping or misbehaving due to an undefined variable; it mirrors what

happens when a programmer carelessly references an array using a leftover index from a consummated loop.

Because we use numerically valued logic (i.e., the ORN function), numerically valued residue-detecting functions, and other numerically valued functions, the set of functions and terminals is closed in the sense that any composition of functions and terminals can be successfully evaluated. This remains the case even after automatically defined functions (with varying numbers of arguments) begin to be created.

8.4.1.3 Fitness Measure

Fitness measures how well a particular genetically-evolved classifying program predicts whether a given segment is, or is not, transmembrane domain. The fitness cases for this problem consist of protein segments. The classifications made by a genetically-evolved program for each protein segment in the *in-sample* set of fitness cases (the *training set*) were compared to the correct classification for the segment. Raw fitness for this problem is based on the value of the correlation, *C*. Standardized ("zero is best") fitness for this problem is $\frac{1-C}{2}$.

The same proteins as used in chapter 18 of Koza 1994a were used here. 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 that was of the same length as the chosen transmembrane segment and that was 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. In addition, 250 out-of-sample fitness cases (125 positive and 125 negative) were created from the remaining 125 proteins in a similar manner in order to measure how well a genetically-evolved program generalizes to other, previously unseen fitness cases from the same problem environment (i.e., the *out-of-sample* data or *testing set*).

8.4.1.4 Parameters Controlling the Run

The architecture-altering operations are intended to be used sparingly on each generation. For our transmembrane problem, the percentage of operations on each generation after generation 4 was 86% crossovers; 10% reproductions; 0% mutations; 1% branch duplications; 1% argument duplications; 0.3% branch deletions; 0.3% argument deletions; 1% branch creations; and 0% argument creations. Since we do not want to waste large amounts of computer time in early generations where only a few programs have any automatically functions at all, we decided to get the run off to a fast start by

setting the percentage of branch creation operations for generations 1 through 4 to 30% (with 60% crossovers and 10% reproductions).

A maximum size of 200 points was established for each result-producing branch, each iteration-performing branch, and each yet-to-be-created automatically defined function.

The other parameters for controlling the runs of genetic programming were the same default values used in Koza [1994a].

8.4.1.5 Implementation on Parallel Computer

The problem (coded in ANSI C) was run on a parallel Parystec computer system consisting of 64 Power PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer (running Windows). The Power PC processors communicate by means of an INMOS transputer that is associated with each Power PC processor. The so-called *distributed genetic algorithm* or *island model* for parallelization [Goldberg l989] was used. That is, subpopulations (called *demes* after Sewell Wright 1943] were situated at the processing nodes of the system. Population size was Q = 2,000 at each of the D = 64 demes for a total population size of M = 128,000. The initial random subpopulations were created locally at each processing node. Generations were run asynchronously on each node. The time-consuming fitness evaluation of the subpopulation of the node and the genetic operations were performed locally on each node. Then, four boatloads, each consisting of B = 5% (the migration rate) of the subpopulation of the node, were selected on the basis of fitness and dispatched to each of the four toroidally adjacent nodes. Details of the parallel implementation of genetic programming can be found in Andre and Koza [1996, as chapter 18 of this volume].

8.4.2 Results

A program that is slightly superior to that written by knowledgeable human investigators was created by our first run. Several subsequent runs produced equally good results and all subsequent runs produced results that were nearly as good.

Focusing on our first run for purposes of discussion, the best-of-generation program for generation 0 has an in-sample correlation of 0.3604. The initial random population of a run of genetic programming is a blind random search of the search space defined by the chosen representation and provides a baseline for the progress of evolution during the run.

The best-of-generation program for generation 4 (with an in-sample correlation of 0.7150) has two automatically defined functions (and an argument map of {2 2}), each possessing two arguments. Since automatically defined functions did not exist in generation 0, the automatically defined functions found in this program are the

consequence of the architecture-altering operations. This particular program is interesting because it illustrates the emergence of a hierarchy (e.g., ADF0 refers to ADF1) that frequently occurs with the architecture-altering operations.

The second-best program of generation 6 (with in-sample correlation of 0.7212) has four automatically defined functions (possessing 0, 3, 2, and 1 arguments, respectively). This program (with an argument map of {0 3 2 1}) contains a complex hierarchy involving ADF0, ADF1, ADF2, and ADF3.

One of the pace-setting programs (i.e., a best-of-generation program reported from one of the 64 processors that is better than any previously reported program) from generation 7 does not have any automatically defined functions at all. As one would expect, this program (with in-sample correlation of 0.7315) is rather large. This program is noteworthy because it is the last program in this run without automatically defined functions that shows up as a pace-setting program. Thereafter, all programs without automatically defined functions consistently run behind the pack in the competitive race to accrue fitness. This observation (which also applies to every other run of this particular problem) is consistent with the proposition that automatically defined functions usually accelerate the solution of problems by genetic programming.

As this run progressed, many different architectures appeared among the pace-setting individuals, including programs with argument maps of {1},{2}, {0 0 2 2}, {3}, {0}, {2 1 2 2}, {1 3}, {3 2}, {0 1}, {0 0}, {0 0 0}, {0 0 0 0}, and {0 0 0 1}.

The out-of-sample correlation generally increases in tandem with the in-sample correlation until generation 28 of this run. The best program of generation 28 scores an in-sample correlation of 0.9596, an out-of-sample correlation of 0.9681. The *error rate* is the number of fitness cases for which the classifying program is incorrect divided by the total number of fitness cases. The in-sample error rate is 3% for this individual; its out-of-sample error rate is 1.6%.

After generation 28, the in-sample correlation continues to rise while its out-of-sample counterpart begins to drop off. Similarly, after generation 28, the in-sample error rate continues to drop while its out-of-sample counterpart begins to rise. This kind of divergence is usually interpreted as the onset of overfitting. Accordingly, we designated the best-of-generation program from generation 28 as the best-of-run program for this run.

Figure 8.4 shows the architecture of the best-of-run program from generation 28 with 208 points (i.e., functions and terminals in the work-performing parts of its branches). This program has one zero-argument automatically defined function, ADF0; one iteration-performing branch, IPB; and one result-producing branch, RPB.

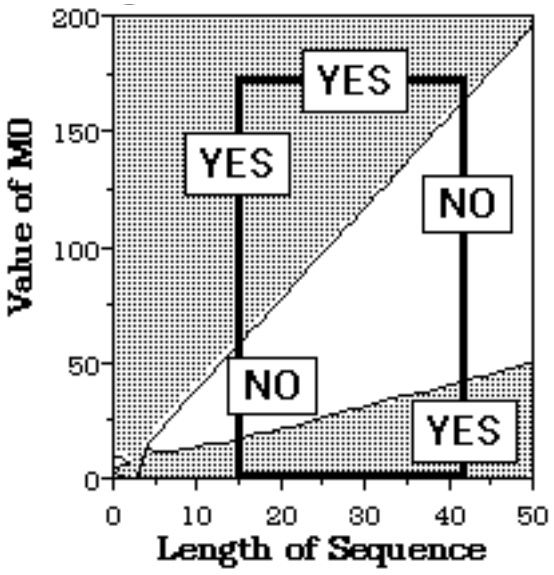

Figure 8.5
Two-part classification produced by the result-producing branch from the best-of-run program from generation 28 as a function of the length, LEN, of the protein segment and the value of M0.

After genetic programming evolves a satisfactory program for a problem, it is frequently very difficult to understand the behavior of the evolved program. A fortunate combination of circumstances permits the analysis of the otherwise incomprehensible 208-point program evolved on this particular run. First, even though a reference to ADF0 migrated into the 169-point result-producing branch, ADF0 fortuitously played no role whatsoever in the result produced by the result-producing branch. Second, none of the 20 residue-detecting functions appeared in the result-producing branch. Thus, the value returned by this particular result-producing branch depends only on the value of the settable variable M0 (communicated from the iteration-performing branch to the result-producing branch) and the length, LEN, of the current protein segment. That is, RPB was a function of only two variables, LEN and M0. The range of values of LEN represented by the fitness cases was, of course, known; LEN ranged between 15 and 45. We were then able to compute a maximum range of values that could be attained by the settable variable M0 for this particular program. We embedded the evolved program (along with its wrapper) into a computer program that looped over the known range for LEN and the inferable maximum range for M0.

Figure 8.5 shows a graph of the behavior of the seemingly incomprehensible 169-point result-producing branch from the best-of-run 208-point program from generation 28. The wrapperized output of this result-producing branch classifies a protein segment as a transmembrane domain for the shaded region (labeled "YES") and classifies the segment as a non-transmembrane area of the protein for the non-shaded region (labeled "NO"). The heavy black frame highlights the area where LEN lies in its known range and where M0 lies in its inferable maximum range.

A value of M0 as high as 168 can only be attained if a protein segment consisted of 42 consecutive electrically charged residues of glutamic acid (E). In fact, glutamic acid is only one of 20 amino acid residues and no actual protein has this bizarre composition. For actual protein segments, only values of M0 that are less than 50 are encountered.

Figure 8.6 is a cutaway portion of figure 8.5 that shows the behavior of the result-producing branch only for values of LEN between 15 and 42 and for values of M0 in the new, constrained range between 0 and 50 (highlighted with a new, smaller heavy black frame). The area inside the new, smaller frame is neatly partitioned into two parts.

We then perform a linear regression on the boundary that partitions figure 8.6 into two parts and find that

$$M0 = 3.1544 + 0.9357\ LEN.$$

These observations permit the operation of the evolved program for classifying a protein segment as a transmembrane domain or a non-transmembrane area of the protein to be stated as follows:

1. Create a sum, SUM, by adding in 4 for each E in the segment and 2 for each C, D, G, H, K, N, P, Q, R, S , T, W, or Y (i.e., the 13 residues other than E, A, M, V, I, F or L).
2. If the quantity

$$\left[\frac{SUM - 3.1544}{0.9357}\right] < LEN,$$

then classify the protein segment as a transmembrane domain; otherwise, classify it as a non-transmembrane area of the protein.

Note that glutamic acid (E) is an electrically charged and hydrophilic amino acid residue. Note also that A (alanine), M (methionine), V (valine), I (isoleucine), F (phenylalanine) and L(leucine) constitute six of the seven most hydrophobic residues on the often-used Kyte-Dolittle hydrophobicity scale [Kyte and Dolittle 1982].

8.4.3 Comparison of Seven Methods

Table 8.1 shows the out-of-sample error rate for the four algorithms for classifying transmembrane domains described in Weiss, Cohen, and Indurkhya [1993] as well as for three approaches using genetic programming, namely the set-creating version [sections 18.5 through 18.9 of Koza 1994a], the arithmetic-performing version [sections 18.10 and 18.11 of Koza 1994a], and the version reported herein.

Table 8.1
Comparison of seven methods.

Method	Error rate
von Heijne 1992	2.8%
Engelman, Steitz, and Goldman 1986	2.7%

Kyte and Doolittle 1982	2.5%
Weiss, Cohen, and Indurkhya 1993	2.5%
GP + Set-creating ADFs	1.6%
GP + Arithmetic-performing ADFs	1.6%
GP + ADFs + Architecture-altering operations	1.6%

The error rate of all three versions using genetic programming are identical; all three are better than the error rates of the other four methods. Genetic programming with the architecture-altering operations was able to evolve a successful classifying program for transmembrane domains starting from a population that initially contained no automatically defined functions. All three versions using genetic programming (none of which employs any foreknowledge of the biochemical concept of hydrophobicity) are instances of an algorithm discovered by an automated learning paradigm whose performance is slightly superior to that of algorithms written by knowledgeable human investigators.

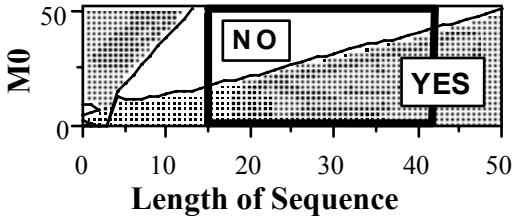

Figure 8.6

Cutaway portion of figure 8.5 showing the behavior of the result-producing branch for the area of interest.

8.5 Conclusion

The biological theory of gene duplication motivated six architecture-altering operations that enable genetic programming to evolve the architecture of a computer program during a run. Genetic programming with the architecture-altering operations was then applied to the problem of evolving a computer program to classify a given protein segment as being a transmembrane domain or non-transmembrane area of a protein (without biochemical knowledge, such as the hydrophobicity values used in human-written algorithms for this task). The out-of-sample error rate for the best genetically-evolved program achieved was slightly better than that of previously-reported human-written algorithms for this problem. This is an instance of an automated machine learning algorithm rivaling human performance on a problem.

The effect of the architecture-altering operations is to make the architecture of a satisfactory solution an emergent property of the genetic programming run (rather than a human choice made in advance of the run).

Acknowledgments

Simon Handley made numerous helpful comments concerning this paper.

8.6 Bibliography

Andre, D. and Koza, J. R. (1996). Parallel genetic programming: A scalable implementation using the transputer architecture. Chapter 18 in this volume. In Angeline, P. and Kinnear, K. E. (eds.) (1996) *Advances in Genetic Programming 2*, The MIT Press, Cambridge, MA.

Andre, D. (1994). Evolution of map making: Learning, planning, and memory using genetic programming. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press. Volume I. Page 250–255.

Angeline, P. J. (1994). Genetic programming and the emergence of intelligence. In Kinnear, K. E. Jr. (editor). *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

Angeline, P. J. and Pollack, J. B. (1994)**.** Coevolving high-level representations. In Langton, Christopher G. (editor). *Artificial Life III, SFI Studies in the Sciences of Complexity*. Volume XVII. Redwood City, CA: Addison-Wesley. Pages 55–71.

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.

Goldberg, D. E. (l989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

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

Kinnear, K. E., Jr. (1994). Alternatives in automatic function definition: A comparison of performance. In Kinnear, K. E. Jr. (editor). *Advances in Genetic Programming*. 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. (1994a). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, MA: The MIT Press.

Koza, J. R. (1994b). *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: The MIT Press.

Koza, J. R. (1994c). *Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming*. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Koza, J. R. (1995a). Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). *Evolutionary Programming IV*: *Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. Pages 695–717.

Koza, J. R. (1995b). Gene duplication to enable genetic programming to concurrently evolve both the architecture and work-performing steps of a computer program. *Proceedings of the 14th International Joint Conference on Artificial Intelligence*. San Francisco, CA: Morgan Kaufmann. Pages 734–740.

Koza, J. R. (1995c). Two ways of discovering the size and shape of a computer program to solve a problem. In Eshelman, Larry J. (editor). *Proceedings of the Sixth International Conference on Genetic Algorithms*. San Francisco, CA: Morgan Kaufmann Publishers. Pages 287–294.

Koza, J. R., and Rice, J. P. (1992). *Genetic Programming: The Movie*. 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.

Ohno, S. (1970). *Evolution by Gene Duplication*. New York: Springer-Verlag.

Rosca, J. P. and Ballard, D. H. (1994a). Hierarchical self-organization in genetic programming. In *Proceedings of the Eleventh International Conference on Machine Learning*. Morgan Kaufmann Publishers.

Rosca, J. P. and Ballard, D. H. (1994b). Learning by adapting representations in genetic programming. *Proceedings of the First IEEE Conference on Evolutionary Computation.* IEEE Press. Volume I. Pages 407–412.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development.* 3(3): 210–229.

Spector, L. (1996). Simultaneous evolution of programs and their control structures. Chapter 7 in this volume. In Angeline, P. and Kinnear, K. E. (eds.) *Advances in Genetic Programming 2*, The MIT Press, Cambridge, MA.

Stryer, L. (1995). *Biochemistry*. New York, NY: W. H. Freeman. Fourth Edition.

Teller, A. (1994). The evolution of mental models. In Kinnear, K. E. Jr. (editor). *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

von Heijne, G. (1992). 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.

Wright, S. (1943). Isolation by distance. *Genetics* 28:114–138.

Yeagle, P. L. (1993). *The Membranes of Cells*. Second edition. San Diego, CA: Academic Press.