Submitted on August 1, 1995 for AAAI Fall Symposium on Genetic Programming in Cambirdge on November 10–12, 1995.

Evolution of Both the Architecture and the Sequence of Work-Performing Steps of a Computer Program Using Genetic Programming with Architecture-Altering Operations

John R. Koza
Computer Science Department
Stanford University
Stanford, California 94305-2140 USA
E-MAIL: 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
Stanford, California 94305-2140 USA
E-MAIL: Andre@flamingo.stanford.edu
PHONE: 415-326-5113

ABSTRACT

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 prespecify the architecture of the ultimate solution to his problem.

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. This corresponds to the question of how new DNA that encodes for a new protein is created in more complex organisms.

This chapter describes how the biological theory of gene duplication described in Susumu Ohno's provocative book, *Evolution by Means of Gene Duplication*, was brought to bear on the problem of architecture discovery in genetic programming. The resulting biologically-motivated approach uses six new architecture-altering operations to enable genetic programming to automatically discover the architecture of the solution at the same time as genetic programming is evolving a solution to the problem.

Genetic programming with the architecture-altering operations 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 (without biochemical knowledge, such as the hydrophobicity values used in human-written algorithms for this task). The best genetically-evolved program achieved an out-of-sample error rate that was better than that reported for other previously reported human-written algorithms. This is an instance of an automated machine learning algorithm matching human performance on a non-trivial problem.

1. Introduction

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 the architecture of the ultimate solution to his problem before he can begin to apply the technique to his problem. One of the banes of automated machine learning from the earliest times has been the requirement that the human user prespecify the size and shape of the ultimate solution to his problem (Samuel 1959). I believe that 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). 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 videotape description of genetic programming can be found in Koza and Rice 1992.

A run of genetic programming in its most basic form automatically creates the size and shape of a single-part result-producing program as well as the sequence of work-performing steps in the program.

I believe that 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 do this in ordinary computer programs. Accordingly, *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a) describes how to evolve 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 simultaneously being evolved. A videotape description of automatically defined functions can be found in Koza 1994b.

When 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 (a) the number of function-defining branches (i.e., automatically defined functions) in the overall program, (b) the number of arguments (if any) possessed by each function-defining branch, and (c) if there is more than one function-defining branch, the nature of the hierarchical references (if any) allowed between the function-defining branches (and between the function-defining branches and the result-producing branch(es) of the overall program).

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. 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.

In nature, sexual recombination ordinarily recombines a part of the chromosome of one parent with a corresponding (homologous) part of the second parent's chromosome. Ordinary mutation occasionally alters isolated alleles belonging to a chromosome.

In his seminal 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."

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 (and then-controversial) 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 (Koza 1994d). The six new architecture-altering operations of branch duplication, branch creation, branch deletion, argument duplication, argument creation, and argument deletion enable genetic programming evolve the architecture of a multi-part program containing automatically defined functions (ADFs) *during a run* of genetic programming. The six architecture-altering operations enable genetic programming to implement what Ohno called the

> "the acquisition of new gene loci with previously non-existent functions."

2. New Architecture-Altering Operations

The six new 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.

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., the 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.

Details of all the new operations are in Koza 1994c.

The operation of branch duplication can be interpreted as a *case splitting*. Subsequent genetic operations may alter one or both of these two presently-identical function-defining branches and these subsequent changes to lead to a divergence in structure and behavior. This subsequent divergence may be interpreted as a *specialization* or *refinement*.

The operation of branch duplication as defined above (and all the other new operations described herein) always produce a syntactically valid program (given closure).

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.

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.

2.4. Argument Creation

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

2.5. Branch Deletion

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

When a function-defining branch is deleted, the question arises as to how to modify invocations of the branch-to-be-deleted by the other branches of the overall program. The alternative used herein (called *branch deletion with*

random regeneration) randomly generates new subtrees composed of the available functions and terminals in lieu of the invocation of the deleted branch and is not semantics-preserving. The operation of branch deletion can be interpreted as a *generalization* because it causes a common treatment to be given to two cases that previously received differentiated (specialized) treatment.

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 by *argument deletion with random regeneration*.

2.7. Structure-Preserving Crossover

When the architecture-altering operations are used, the initial population is created in accordance with some specified 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.

3. 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.

This chapter considers the problem of deciding whether a given protein segment is a transmembrane domain or 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-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 (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 after starting with a population of programs with *no* automatically defined functions at all and whether it is possible for genetic programming, with the new architecture-altering operations, to dynamically determine an architecture that is capable of producing a satisfactory solution to the problem.

3.1. Transmembrane Domains in Proteins

Proteins are polypeptide molecules composed of sequences of amino acids. There are 20 amino acids (also 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).

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 often perform functions such as sensing the presence of certain particles or certain stimuli on one side of the membrane and transporting particles or transmitting signals 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 here is to create a computer program for correctly classifying whether a particular protein segment (i.e., a subsequence of amino acid residues extracted from the entire protein sequence) such as

```
FYITGFFQILAGLCVMSAAAIYTV
```

or

```
TTDLWQNCTTSALGAVQHCYSSSVSEW
```

is a transmembrane domain or a non-transmembrane area of the protein. Success in this problem involves integrating information over the entire protein segment. Both segments contain some residues that are hydrophobic, neutral, and hydrophilic (water-loving). A correct classification cannot be made by merely examining any one particular amino acid residue at particular position in the given protein segment nor even by merely examining any small combination of positions. Success in this task. As it happens, the first segment (24 residues) 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.

3.2. Preparatory Steps

When the "minimalist" approach is used in conjunction with architecture-altering operations in genetic programming, all programs in the initial random population have a minimal structure.

Figure 1 shows the "minimalist" structure used in generation 0 here 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).

Figure 1 Overall structure of programs for generation 0 with one iteration-performing branch, `IPB`, and one result-producing branch, `RPB`.

There will be a total of 38 functions and terminals in function set and the terminal set of this problem:

- 7 initial functions,
- 23 initial terminals,
- 4 potential functions, and
- 4 potential terminals.

3.2.1. Function Set

When the architecture-altering operations are used, both functions and terminals can freely 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 must abandon the approach to defining the function set and terminal set of the problem used in *Genetic Programming* (Koza 1992) and *Genetic Programming II* (Koza 1994a) and, instead, define

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

As it happens, for this problem, the initial function set, $\mathcal{F}_{\text{initial}}$, is common to both the result-producing branch, `RPB`, and the iteration-performing branch, `IPB`, 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

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

taking 2, 2, 2, 2, 3, 1, and 2 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 evaluates and returns its second argument if its first argument is greater than or equal zero, but otherwise evaluates and returns its third argument.

The one-argument setting function, SETM0, can be used to set the settable memory variable, M0, to a particular value.

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

Note that when the minimalist approach is used in conjunction with architecture-altering operations, 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 4 automatically defined functions, each possessing between 0 and 4 dummy arguments, was established. Thus, the set of potential additional terminals, $\mathcal{T}_{\text{potential}}$, for this problem consists of

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

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

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

each taking an as-yet-unknown number of arguments (between 0 and 4).

3.2.2. 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,

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

Here ← represents floating-point random constants between –10.000 and +10.000. Since we want to encode each point (internal or external) 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 (initial and potential) in function set and terminals (initial and potential) in the terminal set. These 200 or so initial random constants are adequate for this problem because they frequently recombine in later generations of the run using various arithmetic functions to produce new constants.

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

LEN is the length of the current protein segment.

(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 (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 the 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,

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

Of course, the residue-detecting functions may migrate into the body of a yet-to-be-created automatically defined function that may, in turn, become referenced by some yet-to-be-created call from the result-producing branch. We deal with this possibility by specifying that when a residue-detecting function is called directly by the result-producing branch, RPB (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 the 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 mirrors what happens when a programmer carelessly references an array using a leftover index from a consummated loop.

Because we used numerically valued logic (i.e., the ORN function) and numerically valued residue detecting functions in conjunction with other numerically valued functions, the set of functions and terminals is closed in 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.

A wrapper 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 value, the segment will be classified as a transmembrane domain, but otherwise it will be classified as a non-transmembrane area of the protein.

3.2.3. Fitness

Fitness measures how well a particular genetically-evolved classifying program predicts whether the segment is, or is not, transmembrane domain. The fitness cases for this problem consist of protein segments. The classification made by the 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; standardized ("zero is best") fitness is $\frac{1-C}{2}$.. 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 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 manner similar to the above 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*).

3.2.4. Parameters

The architecture-altering operations are intended to be used sparingly on each generation. 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 the result-producing branch, the iteration-performing branch, and each of the yet-to-be-created automatically defined functions.

The other parameters for controlling the runs of genetic programming were the default values specified in Koza (1994a).

The problem (coded in ANSI C) was run on a medium-grained 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 one 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 128,000. The initial random subpopulations were created locally at each processing node. Generations were run asynchronously on each node. After a generation of genetic operations was performed locally on each node, four boatloads, each consisting of B = 5% (the migration rate) of the subpopulation (selected on the basis of fitness) were 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.

Termination Criterion and Results Designation

Since perfect classifying performance was unlikely to occur, the run was monitored and manually terminated.

3.3. Results

Since a program that is superior to that written by knowledgeable human investigators was created during the first run (and we had no reason to create a performance curve for this problem), this run (taking about 23 hours of computer time) is the only run that we have made of this version of the problem.

The best-of-generation program for generation 0 has an in-sample correlation of 0.3604.

The best-of-generation program for generation 4 (with an in-sample correlation of 0.7150) has two automatically defined functions, each possessing two arguments. Since automatically defined functions did not exist in generation 0, these automatically defined functions exist in this program because of the architecture-altering operations. This program illustrates the emergence of hierarchy (involving ADF0 refers to ADF1) that occurs with the architecture-altering operations.

The second-best program of generation 6 (with in-sample correlation of 0.7212) is interesting because it 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 complicated hierarchy among ADF0, ADF1, ADF2, and ADF3.

The second-best program of generation 7 (with in-sample correlation of 0.7315) does not have any automatically defined functions at all. Not surprisingly, it is a rather large program. This program is the last program without automatically defined functions that shows up as a pace-setting program in this run (i.e., a best-of-generation program reported from one of the 64 processors that is better than any previously reported program). Thereafter, programs without automatically defined functions consistently run behind the pack in the competitive race to accrue fitness. This observation is a small additional indication in favor of the general proposition (which we believe, after the test of time, will prove to be usually true) that genetic programming produces solutions to problems with less computational effort with automatically defined functions than without them.

As the run progresses, many different architectures appear among the pace-setting individuals, including {2 2}, {0 3 2 1}, {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, an in-sample error rate of 3%, and an out-of-sample error rate of 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. Based on this, we designated the best-of-generation program from generation 28 as the best-of-run program.

Figure 2 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 2 Architecture of best-of-run program from generation 28.

After genetic programming evolves a satisfactory program for a problem, it is often difficult to ascertain the behavior of the resulting program. A lucky combination of circumstances permitted the analysis of the otherwise incomphrehensible 208-point program evolved on this run. First, even though a reference to ADF0 migrated into the 169-point result-producing branch, ADF0 ended up in an intron and played no role in the result produced by the result-producing branch. Second, none of the 20 residue-detecting functions were in the result-producing branch. Thus, the value returned by the result-producing branch is dependent 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. The range of values of LEN was known to be between 15 and 45 (from the fitness cases). Then, fortuitously, we were able to compute a maximum range of values that could be attained by the settable variable M0 for this particular program. We then embedded the evolved program (along with the wrapper) into a simple computer program that looped over the known range for LEN and the inferrable maximum range for M0.

Figure 3 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 clasifies 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 its known range and where M0 lies in its inferrable maximum range. Note that there is an exceedingly complex and confused partitioning of the plane near the origin; however, this part of the figure is irrelevant to us.

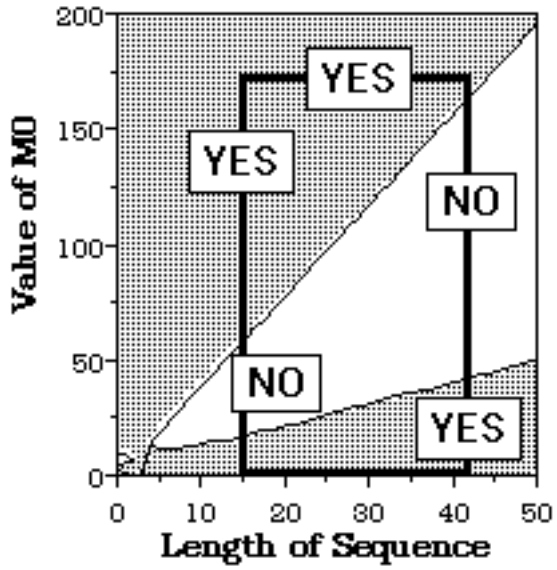

Figure 3 Behavior of the result-producing branch from the best-of-run program.

Although M0 can conceivably attain a value as high as 168, such a value can only be attained if the protein segment consists of 42 consecutive electrically charged residues of glutamic acid (E). However, glutamic acid is only one of 20 amino acid residues that appear in proteins. For actual protein segments, only values of M0 that are less than 50 are encountered.

Figure 4 is a cutaway portion of figure 3 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, contrained range between 0 and 50 (highlighted with a new, smaller heavy black frame). As can be seen, the area inside the new frame is neatly partitioned into two parts.

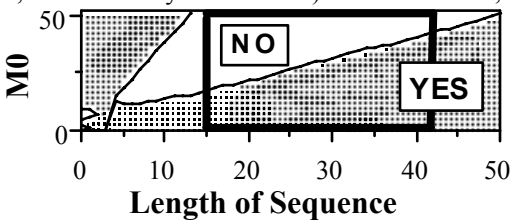

Figure 4 Cutaway portion of graph showing behavior of the result-producing branch.

We then perform a linear regression on the boundary that partitions figure 4 into two parts and find that

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

Thus, the evolved program for classifying a protein segment as a transmembrane domain or a non-transmembrane area of the protein can be restated as follows:

(1) Create a sum, SUM, by adding 4 for each E in the protein segment and 2 for each C, D, G, H, K, N, P, Q, R, S, T, W, or Y (i.e., the 13 residues that are neither E nor A, M, V, I, F or L).

(2) If $\left\lfloor \frac{SUM - 3.1544}{0.9357} \right\rfloor < LEN$,

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 and that A (alanine), M (methionine), V (valine), I (isoleucine), F (phenylalanine) and L(leucine) constitute six of the seven the most hydrophobic residues on the often-used Kyte-Dolittle hydrophobicity scale (Kyte and Dolittle 1982).

3.4. Comparison of Seven Methods

Table 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 (chapters 18.5 through 18.9 of Koza 1994a), the arithmetic-performing version (chapters 18.10 and 18.11 of Koza 1994a), and the version using the architecture-altering operations as reported herein.

Table 1 Comparison of seven 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%
GP + Set-creating ADFs	1.6%
GP + Arithmetic-performing ADFs	1.6%
GP + ADFs + Architecture-altering operations	1.6%

As can be seen from the table, 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 new 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.

Bibliography

Andre, David and Koza, John R. (1996). Parallel genetic programming on a network of transputers. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.

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, David E. l989a. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Holland, John 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. The second edition is currently available from The MIT Press 1992.

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

Koza, John R. 1994a. *Genetic Programming II: Automatic Discovery of Reusable Programs.* Cambridge, MA: The MIT Press.

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

Koza, John 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, John R., and Rice, James 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, Susumu. 1970. *Evolution by Gene Duplication*. New York: Springer-Verlag.

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

Stryer, Lubert. 1995. *Biochemistry*. W. H. Freeman. Fourth Edition.

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, Sewall. 1943. Isolation by distance. *Genetics* 28:114–138.

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