A CASE STUDY WHERE BIOLOGY INSPIRED A SOLUTION TO A COMPUTER SCIENCE PROBLEM

John R. Koza
Computer Science Department
Stanford University
Stanford, CA 94305 USA
EMAIL: Koza@CS.Stanford.Edu
WWW: http://www-cs-faculty.stanford.edu/~koza/
David Andre
Visiting Scholar
Computer Science Department
Stanford University
EMAIL: andre@flamingo.stanford.edu
WWW: http://www-leland.stanford.edu/~phred/

ABSTRACT

This paper 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 a vexatious problem from the domain of automated machine learning, namely the problem of architecture discovery. Six new architecture-altering operations for genetic programming were motivated by the way that new biological structures, functions, and behaviors arise in nature using gene duplication. Genetic programming with the new architecture-altering operations was then applied to the transmembrane protein segment identification problem. 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.

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 prespecify the architecture of the ultimate solution to his problem before he can apply the technique to his problem.

In nature, sexual recombination (crossover) exchanges alleles (gene values) at particular locations (loci) along the chromosome (a molecule of DNA). John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the naturally-occurring evolutionary process using analogs of crossover and mutation can be applied to solving artificial problems using what is now called the *genetic algorithm*.

In both natural evolution and the artificial evolutionary process implemented by the genetic algorithm, crossover merely exchanges alleles (gene values) at particular locations along an already-existing fixed-size chromosome. These exchanges do not explain how new structures, new functions, and new behaviors arise in nature. Indeed, in nature, there is not only short-term optimization of alleles in their fixed locations within a fixed-size chromosome, but long-term emergence of new proteins (which, in turn, create new structures, functions, and behaviors). The emergence of new proteins *alters the architecture* of the chromosome. Indeed, genome lengths in nature have generally increased with the emergence of new and more complex organisms (Dyson and Sherratt 1985).

Returning to genetic algorithms, a change in the architecture of a chromosome corresponds to a dynamic alteration, during a run of the algorithm, of the user-created mapping (the encoding and decoding) between points from the search space of the problem and instances of the artificial chromosome.

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 Kinnear 1994 and Koza and Rice 1992. 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 that main program.

Genetic Programming II: Automatic Discovery of Reusable Programs (Koza 1994a, 1994b) 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 concurrently being evolved.

Figure 1 shows an overall computer program (presented as a parse tree) consisting of a result-producing main branch (`RPB`) on the right and one function-defining branch on the left. The function-defining branch defines a two-argument automatically defined function (called `ADF0`). The result-producing branch invokes the subroutine (`ADF0`) from two different places (at 481 and 487) using different pairs of arguments for the two calls. In particular, the terminals `D1` and `D2` are used for the first call at 481 and the terminal `D3` at 488 and the sub-expression `(NOR D4 D0)` at 489 are used for the second call. The function-defining branch has an argument list at 412 consisting of two dummy arguments (formal parameters), `ARG0` and `ARG1`. The dummy arguments appear in the body of the function-defining branch; however, when the subroutine `ADF0` is called by the result-producing branch, these dummy arguments are instantiated with (generally) different values.

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 (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 (function-defining and result-producing) of the overall program.

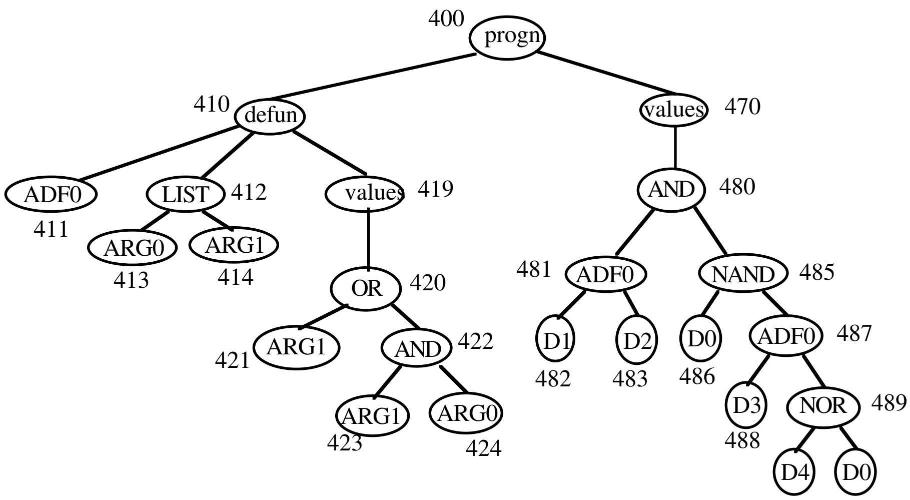

Figure 1 Program consisting of one two-argument function-defining branch (`ADF0`) and one result-producing branch.

This paper explores the issue of whether it is possible to determine the architecture of a multi-part program dynamically during a run of genetic programming (rather than require that the user predetermine that architecture before the run starts).

In genetic programming (with or without automatically defined functions), crossover is analogous to the exchanging of alleles (gene values) in a chromosome. In the case of genetic programming, sub-trees (containing a composition of the primitive functions and terminals of the problem) are exchanged by the crossover operation. The analog of a genome change corresponds to a dynamic alteration, during a run of genetic programming, of the architecture of a computer 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.

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 merely alters isolated alleles belonging to an existing chromosome with an existing architecture.

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." (Emphasis added).

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

In certain very rare and unpredictable instances, recombination does not occur in the usual way and, instead, a gene duplication occurs. A gene duplication is a rate and illegitimate recombination event that results in the duplication of a subsequence of a chromosome.

Ohno's 1970 book proposed the provocative thesis that the creation of new proteins (and hence new structures, functions, and behaviors in living things) begins with a gene duplication and that gene duplication is "the major force of evolution."

This paper 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 evolve the architecture of a multi-part program containing automatically defined functions *during a run* of genetic programming.

The potential usefulness of the architecture-altering operations has been previously demonstrated using simple problems such as Boolean parity problems (Koza 1995). In this paper, 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).

2. Gene Duplication and Deletion in Nature

Ohno's *Evolution by Gene Duplication* (1970) corrects the mistaken notion that natural selection is a mechanism for promoting change. Instead, Ohno emphasizes the essentially conservative role of natural selection in the evolutionary process:

> "...the true character of natural selection ... is not so much an advocator or mediator of heritable changes, but rather it is an extremely efficient policeman which conserves the vital base sequence of each gene contained in the genome. As long as one vital function is assigned to a single gene locus within the genome, natural selection effectively forbids the perpetuation of mutation affecting the *active* sites of a molecule." (Emphasis in original).

Ohno continues,

> "Only by the accumulation of *forbidden* mutations at the *active* sites can the gene locus change its basic character and become a new gene locus. An escape from the ruthless pressure of natural selection is provided by the mechanism of gene duplication. By duplication, a redundant copy of a locus is created. Natural selection often ignores such a redundant copy, and, while being ignored, it accumulates formerly forbidden mutations and is reborn as a new gene locus with a hitherto non-existent function." (Emphasis in original).

Ohno concludes,

> "Thus, gene duplication emerges as the major force of evolution."

Natural selection exerts considerable force in favor of maintaining a gene that encodes for the manufacture of a protein that is important for the successful performance and survival of the organism. But, after a gene duplication has occurred, there is no disadvantage associated with the loss of the *second* way of manufacturing the original protein. Consequently, natural selection usually exerts little or no pressure to maintain a second way of manufacturing the same protein. Over a period of time, the second gene may accumulate additional changes and diverge more and more from the original gene. Eventually the changed gene may lead to the manufacture of a distinctly new and different protein that actually does affect the structure and behavior of the living thing in some advantageous or disadvantageous way. When a changed gene leads to the manufacture of a viable and advantageous new protein, natural selection then preserves the new gene.

Ohno's provocative thesis is supported by the discovery of pairs of proteins with similar sequences of DNA and similar sequences of amino acids, but different functions. Examples include trypsin and chymotrypsin; the protein of microtubules and actin of the skeletal muscle; myoglobin and the monomeric hemoglobin of hagfish and lamprey; myoglobin used for storing oxygen in muscle cells and the subunits of hemoglobin in red blood cells of vertebrates; = and the light and heavy immunoglobin chains (Dyson and Sherratt 1985, Hood and Hunkapiller 1991). For the *Escherichia coli* bacteria, more than 30% of its proteins are the result of gene duplications (Riley 1993). These proteins include its DNA polymerases, dehydrogenases, ferredoxins, carbamoyl-phosphate synthetases, F-type ATPases, DNA topoisomerases. More complex organisms have a general tendency to have more different kinds of structures, to have more complex structures, to perform more different functions, to have more different kinds of behaviors, to have more expressed proteins, and to have longer genomes (Dyson and Sherratt 1985).

Gene deletion also occurs in nature. In gene deletion, there is a deletion of a portion of the string of nucleiotide bases that would otherwise be translated and manufactured into work-performing proteins in the living cell. After a gene deletion occurs, some particular protein that was formerly manufactured will no longer be manufactured. The absence of the protein may then affect the structure and behavior of the living thing in some advantageous or disadvantageous way. If the deletion is advantageous, natural selection will tend to perpetuate the change, but if the deletion is disadvantageous, natural selection will tend to lead to the extinction of the change.

3. 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. During each generation of the evolutionary process, a certain percentage of the individuals in the population participate in the architecture-altering operations. Meanwhile, Darwinian selection causes differential selection in favor of more fit individuals. And, during each generation, individuals in the population are modified by the operations of crossover and mutation. The result is that the evolutionary process will select against individuals in the population with architectures that are less suitable for solving the problem. Individuals with unsuitable architectures will tend to become extinct over a period of generations. Similarly, individuals with architectures that facilitate solving the problem will tend to prosper.

3.1. Branch Duplication

The operation of *branch duplication* duplicates one of the branches of a program. First, a program is selected from the population to participate in this operation. This step is performed probabilistically on the basis of fitness for this operation (and all the other architecture-altering operations described herein). Second, one of the function-defining branches of the selected program is picked as the branch-to-be-duplicated. Third, a uniquely-named new function-defining branch is added to the selected program, thus increasing 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. Fourth, 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), a random choice is made either to leave that invocation unchanged or to replace that invocation with an invocation of the new branch. The operation of branch duplication (and all the other architecture-altering operations described herein) always produces a syntactically valid program.

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 described herein) always produce a syntactically valid program (given closure).

Figure 2 shows the program resulting after applying the operation of branch duplication to figure 1. Specifically, the function-defining branch 410 of figure 1 defining `ADF0` (also shown as 510 of figure 2) is duplicated and a new function-defining branch (defining `ADF1`) appears at 540 in figure 2. There are two occurrences of 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 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 1, the choice is randomly made to replace `ADF0` 481 with `ADF1` 581 in figure 2. The arguments for the invocation of `ADF1` 581 are `D1` 582 and `D2` 583 in figure 2 (i.e., they are identical to the arguments `D1` 482 and `D2` 483 for the invocation of `ADF0` at 481 in figure 1). For the second invocation of `ADF0` at 487 of figure 1, `ADF0` is left unchanged.

The new branch is identical to the previously existing branch (except for the name `ADF1` at 541 in figure 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.

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 to 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 in Koza 1995.

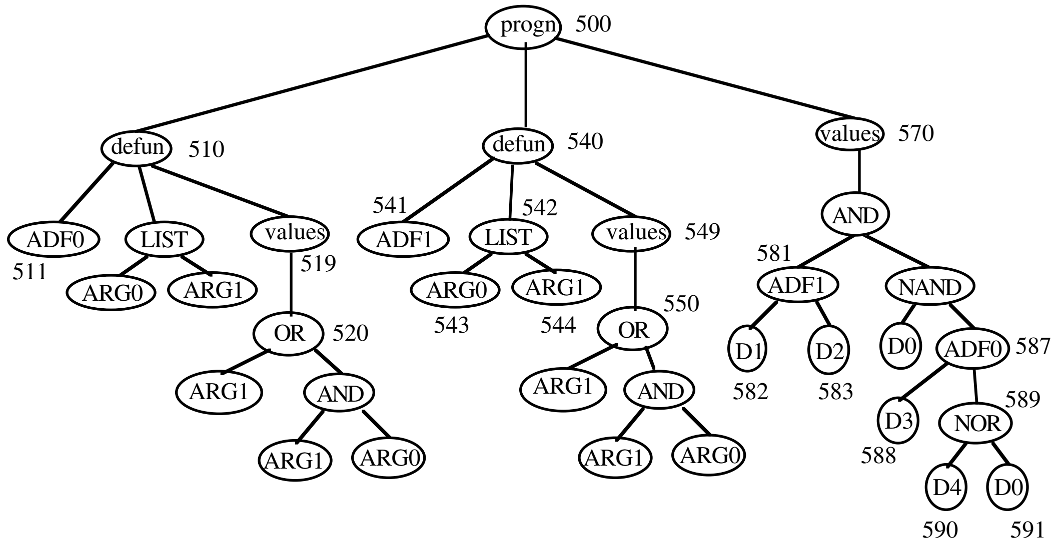

Figure 2 Program consisting of two two-argument function-defining branches (ADF0 and ADF1) and one result-producing branch.

3.2. Argument Duplication

In the operation of *argument duplication*, a uniquely-named new argument is added 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. Then, for each occurrence of the argument-to-be-duplicated anywhere in the body of picked function-defining branch of the selected program, a random choice is made either to leave that occurrence unchanged or to replace that occurrence with the new argument.

For each occurrence of an invocation of the picked function-defining branch anywhere in the selected program, the argument subtree corresponding to the argument-to-be-duplicated is identified and duplicated in that argument subtree in that invocation, thereby increasing the number of arguments in the invocation. The effect of this operation is to leave unchanged the value returned by the overall program.

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

3.3. 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 branch duplication in that any point of any branch may be picked and in that the picked point may even be in the result-producing branch.

3.4. Argument Creation

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

3.5. Branch Deletion

The operation of *branch deletion* deletes an automatically defined function. 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. This approach is, of course, not semantics-preserving.

3.6. Argument Deletion

The operation of *argument deletion* deletes one of the arguments to one of the automatically defined functions of a program.

3.7. Structure-Preserving Crossover

When the architecture-altering operations are used, the initial population is created in accordance with a constrained syntactic structure that supports the structure shown in figure 1. 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.

4. Classifying Protein Segments as Transmembrane Domains

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.

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, 1994c). In these previous 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 evolve an architecture that is capable of producing a satisfactory solution.

4.1. Transmembrane Domains in Proteins

A *transmembrane protein* 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. 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. 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 to be hydrophobic. Success in this problem involves integrating information over the entire protein segment.

4.2. Preparatory Steps

The premise behind the architecture-altering operations is that the competitive evolutionary process should be allowed to select the best architecture. Thus, each program in the initial random population (generation 0) begins a result-producing branch and an iteration-performing branch, but no automatically defined functions.

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. 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 value, the segment will be classified as a transmembrane domain, but otherwise the segment will be classified as a non-transmembrane area.

Details about the function set, the terminal set, the parameters for controlling runs, and the implementation of the runs on a parallel computer system are found in Koza and Andre 1996a, 1996b.

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`}.

`IFGTZ` ("If Greater Than Zero") is a three-argument conditional branching operator that evaluates and returns its second argument if its first argument is greater than or equal zero, but otherwise evaluates and returns its third argument. `ORN` is a two-argument numerically valued disjunctive function. The one-argument setting function, `SETM0`, sets the settable memory variable, `M0`, to a particular value.

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, they begin to appear.

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,

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

$\Re$ represents floating-point random constants between –10.000 and +10.000. `M0` is a settable memory variable. It is zero when execution of a program begins for a particular fitness case. `LEN` represents the length of the current protein segment.

`(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 the body of the iteration-performing branch is executed, 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,

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

Because we use numerically valued logic (i.e., the ORN function), numerically valued residue-detecting functions (e.g., (A?)), and other numerically valued functions (such as the arithmetic functions), the set of functions and terminals is closed in the sense that any composition of functions and terminals can be successfully evaluated.

The fitness of a particular genetically-evolved classifying program is measured by the correlation between the wrapperized output of its result-producing branch and the correct classification, over an *in-sample* set of fitness cases (the *training set*) consisting of protein segments. 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.

4.3. Results

A program that is superior to that written by knowledgeable human investigators was created by our first run (and subsequent runs).

As the first run progressed, many different architectures appear among the best-of-generation individuals from various generations.

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 (and 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.

The evolved best-of-run program for classifying a protein segment as a transmembrane domain or a non-transmembrane area can be restated as follows:

(1) Create a sum, SUM, by adding in 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 other than E, A, M, V, I, F or L).

$$(2)\ \text{If} \left[\frac{\texttt{SUM} - 3.1544}{0.9357}\right] < \texttt{LEN},$$

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 and also 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 Kyte-Dolittle hydrophobicity scale (Kyte and Dolittle 1982).

4.4. 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 (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 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 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.

5. Conclusion

The biological theory of gene duplication was used to motivate 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 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 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.

Bibliography

Andre, D. and Koza, J. R. 1996a. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, Peter J. and

Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming II*. Cambridge, MA: The MIT Press.
Andre, D. and Koza, J. R. 1996b. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. In Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming II*. Cambridge, MA: The MIT Press.
Dyson, Paul and Sherratt, David. 1985. Molecular mechanisms of duplication, deletion, and transposition of DNA. In Cavalier-Smith, T. (editor). *The Evolution of Genome Size*. Chichester: John Wiley & Sons.
Engelman, D., Steitz, T., and Goldman, A. 1986. Identifying nonpolar transbilayer helices in amino acid sequences of membrane proteins. *Annual Review of Biophysics and Biophysiological Chemistry*. Volume 15.
Holland, 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.
Hood, Leory and Hunkapiller, Tim. 1991. Modular evolution and the immunoglobin gene superfamily. In Osawa, S. and Honjo, T. (editors). *Evolution of Life*. Tokyo: Springer-Verlag.
Kinnear, Kenneth E. Jr. (editor). 1994. *Advances in Genetic Programming*. Cambridge, MA: The MIT Press.
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. Evolution of a computer program for classifying protein segments as transmembrane domains using genetic programming. In Altman, Russ, Brutlag, Douglas, Karp, Peter, Lathrop, Richard, and Searls, David (editors). *Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology*. Menlo Park, CA: AAAI Press. Pages 244–252.
Koza, John R. 1995. 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, 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.
Riley, M. 1993. Functions of the gene products of *Escherichia coli*. *Reviews of Microbiology*. Volume 32. Pages 519–560.
von Heijne, G. 1992Membrane 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.