

Evolution of Iteration in Genetic Programming

John R. Koza

Computer Science Department
Stanford University
Stanford, California 94305-9020 USA
E-MAIL: Koza@Cs.Stanford.Edu
WWW: <http://www-cs-faculty.stanford.edu/~koza/>

David Andre

Visiting Scholar
Computer Science Department
Stanford University
Stanford, California 94305 USA
E-MAIL: Andre@Flamingo.Stanford.Edu

ABSTRACT

The solution to many problems requires, or is facilitated by, the use of iteration. Moreover, because iterative steps are repeatedly executed, they must have some degree of generality.

An automatic programming system should require that the user make as few problem-specific decisions as possible concerning the size, shape, and character of the ultimate solution to the problem.

Work first presented at the Fourth Annual Conference on Evolutionary Programming in 1995 (EP-95) demonstrated that six then-new architecture-altering operations made it possible to automate the decision about the architecture of an overall program dynamically during a run of genetic programming. The question arises as to whether it is also possible to automate the decision about whether to employ iteration, how much iteration to employ, and the particular sequence of iterative steps.

This paper introduces the new operation of restricted iteration creation that automatically creates a restricted iteration-performing branch out of a portion of an existing computer program during a run a genetic programming. Genetic programming with the new operation is then used (in conjunction with the other architecture-altering operations first presented at EP-95) to evolve a computer program to solve a non-trivial problem.

1. Introduction

An automatic programming system should require that the user make as few problem-specific decisions as possible prior to presenting his problem to the system. One of the most persistent and vexatious aspects of automated machine learning from the earliest times has been the requirement that the human user predetermine the size, shape, and character of the ultimate solution to the problem (Samuel 1959). The size, shape, and character 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.

1.1. Evolving Single-Part Programs

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 scientific and engineering problems using what is now called the *genetic algorithm*.

Genetic programming extends Holland's genetic algorithm to the task of automatic programming. Early work on genetic programming demonstrated that it is possible to evolve a sequence of work-performing steps in a single result-producing branch (that is, a one-part "main" program). 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 and terminals). See also Koza and Rice (1992).

In the most basic form of genetic programming (where only a single result-producing branch is evolved), genetic programming demonstrated the capability to discover a sequence (as to both its length and its content) of work-performing steps that is sufficient to produce a satisfactory solution to several problems, including many problems that have been used over the years as benchmarks in machine learning and artificial intelligence. Before applying genetic programming to a problem, the user must perform five major preparatory steps, namely identifying the terminals (inputs) of the to-be-evolved programs, identifying the primitive functions (operations) contained in the to-be-evolved programs, creating the fitness measure for evaluating how well a given program does at solving the problem at hand, choosing certain control parameters (notably population size and number of generations to be run), and determining the termination criterion and method of result designation (typically the best-so-far individual from the populations produced during the run).

1.2. Evolving Multi-Part Programs

Later work on genetic programming demonstrated that it is possible to co-evolve the work-performing steps of one or more function-defining branches (automatically defined functions) concurrently with the work-performing steps of a result-producing branch so as to gain leverage inherent in reusable and parameterizable subroutines. In ordinary computer programs, subroutines provide a hierarchical mechanism to exploit, *by reuse and parameterization*, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments.

Accordingly, *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 (ADF)* is a function (sometimes also called a subroutine, subprogram, DEFUN, procedure, module) whose work-performing body is dynamically evolved during a run of genetic programming and which may be called by a calling main program (or calling subprogram) whose work-performing body is concurrently being evolved. Genetic programming with automatically defined functions has demonstrated the capability of co-evolving the work-performing steps of the function-defining branches (automatically defined functions) concurrently with the work-performing steps of a result-producing branch.

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 user may supply the specification of the architecture as an additional sixth preparatory step; however, it is preferable, in many situations, to automate this decision so that the user is not required to prespecify the architecture.

1.3. Evolution of Architecture

Recent work on genetic programming has demonstrated that it is possible to evolve the architecture of an overall program dynamically during a run of genetic programming using six recently developed architecture-altering operations, namely branch duplication, argument duplication, branch deletion, argument deletion, branch creation, and argument creation. These six recently developed architecture-altering operations introduced at the Fourth Annual Conference on Evolutionary Programming (Koza 1995) provide an automated way to enable genetic programming to dynamically determine, during the run, whether or not to employ function-defining branches, how

many function-defining branches to employ, and the number of arguments possessed by each function-defining branch.

The architecture-altering operations and automatically defined functions together provide an automated way to change the representation of a problem while solving the problem. Alternately, they can also be viewed as an automated way to decompose a problem into a non-prespecified number of subproblems of non-pre-specified dimensionality; to solve the subproblems; and to assemble the solutions of the subproblems into a solution of the overall problem. The architecture-altering operations can also be interpreted as providing an automated way to specialize and generalize during the problem-solving process.

1.4. Evolution of Iteration

Typical computer programs contain iterative operators that perform some specified work until some condition expressed by a termination predicate is satisfied. Iteration is an important element in computer programs. The solution to many problems requires, or is facilitated by, iteration. Moreover, because iterative steps are repeatedly executed, they must have some degree of generality.

The question arises as to whether it is possible to automate the decision about whether to employ iteration, how much iteration to employ, and the particular sequence of iterative steps in a computer program that is being evolved by genetic programming to solve a problem.

This paper introduces the new operation of restricted iteration creation that automatically creates a restricted iteration-performing branch out of a portion of an existing computer program during a run a genetic programming. Genetic programming with the new operation of restricted iteration creation is then used (in conjunction with the other architecture-altering operations first presented at EP-95) to evolve a computer program to solve a non-trivial problem.

1.5. Organization of This Paper

Section 2 provides background on genetic programming. Section 3 explains the new architecture-altering operation of restricted iteration creation. Section 4 demonstrates the application of the new operation of restricted iteration creation to the transmembrane segment identification problem. Section 5 compares the best genetically-evolved program achieved using the operation of restricted iteration creation with the results of previously reported human-written algorithms and with the results of two previous approaches using genetic programming. Section 6 is the conclusion.

2. Background on Genetic Programming

Execution of genetic programming consists of the following steps:

- (1) Generate an initial random population (generation 0) of computer programs.
- (2) Iteratively perform the following sub-steps until the termination criterion of the run has been satisfied:

- (a) Execute each program in the population and assign it (explicitly or implicitly) a fitness value according to how well it solves the problem.
- (b) Select program(s) from the population to participate in the genetic operations in (c) below.
- (c) Create new program(s) for the population by applying the following genetic operations.
 - (i) *Reproduction*: Copy an existing program to the new population.
 - (ii) *Crossover*: Create new offspring program(s) for the new population by recombining randomly chosen parts of two existing programs.
 - (iii) *Mutation*: Create one new offspring program for the new population by randomly mutating a randomly chosen part of one existing program.
 - (iv) *Branch duplication*: Create one new offspring program for the new population by duplicating one function-defining branch of one existing program and making additional appropriate changes to reflect this change.
 - (v) *Argument duplication*: Create one new offspring program for the new population by duplicating one argument of one function-defining branch of one existing program and making additional appropriate changes to reflect this change.
 - (vi) *Branch deletion*: Create one new offspring program for the new population by deleting one function-defining branch of one existing program and making additional appropriate changes to reflect this change.
 - (vii) *Argument deletion*: Create one new offspring program for the new population by deleting one argument of one function-defining branch of one existing program and making additional appropriate changes to reflect this change.
 - (viii) *Branch creation*: Create one new offspring program for the new population by adding one new function-defining branch containing a portion of an existing branch and creating a reference to that new branch.
 - (ix) *Argument creation*: Create one new offspring program for the population by adding one new argument to the argument list of an existing function-defining branch and appropriately modifying contents of the branch and references to the branch.
 - (x) *Restricted iteration creation*. Create one new offspring program for the new population by adding one new iteration-performing branch containing a portion of an existing branch and creating a reference to that new branch.
- (3) After satisfaction of the termination criterion (which usually includes a maximum number of generations to be run as well as a problem-specific success predicate), the single best computer program in the population produced during the run (the best-so-far individual) is designated as the result of the run. This

result may (or may not) be a solution (or approximate solution) to the problem.

Details of the basic genetic operations of reproduction, crossover, and mutation mentioned in (i) through (iii) of (2)(c) below are found in Koza (1992) or Koza (1994a). The six operations appearing as steps (2)(c)(iv) through (2)(c)(ix) are the recently developed architecture-altering operations (Koza 1995). The restricted iteration creation operation in (2)(c)(x) is the new architecture-altering operation that is described in detail in the next section of this paper.

3. The New Operation of Restricted Iteration Creation

The new operation of *restricted iteration creation* operates on one parental computer program and creates one new offspring program containing a new iteration-performing branch composed of a portion of an existing branch of the parental program.

In its most general form, an iteration consists of an initialization step, a termination predicate, a work-performing body, and an update step. After executing the initialization step, the termination predicate is tested. If the iteration is not terminated, the work-performing steps in the body of the iteration and the update step are then executed and control is returned to the termination predicate.

As an example of a simple iteration, consider the following iteration involving an indexing variable. The starting calculation may set an indexing variable to an initial value; the terminate predicate may test whether the indexing variable equals or exceeds a final value; and the update calculation may increment the indexing variable. Typically the work-performing steps of the iteration depend on the current value of the iteration variable. Many iterative calculations work in conjunction with memory (state). The memory transmits information from one execution of the iterative calculation to the next.

As every programmer knows, iterative loops can be very time-consuming and, in the worst case, unending. In genetic programming, the computer programs in the population are initially created at random. These programs are subsequently subjected to modification by the crossover and mutation operations (and, possibly, various architecture-altering) operations. If no restrictions are imposed on iteration, iterative operators can run for long periods of time, can become nested (and therefore consume large amounts of computer time), or can acquire unsatisfiable termination predicates (and therefore consume endless amounts of computer time). It is therefore a practical necessity to impose some kind of *rationing* on iteration in evolved programs.

One approach to rationing involves the imposition of somewhat arbitrary external time-out limits on the number of iterative steps that may be performed.

In some problems, iterative calculations can usefully be performed over a particular known finite set. Examples include two-dimensional pattern recognition problems where an iteration might usefully be performed over the

entire two-dimensional array of pixels of known size, economic time series and signal processing problems where an iteration might usefully be performed over a certain known number of time steps, and protein or genomic sequence problems where an iteration might usefully be performed over the given linear sequence of amino acid residues or nucleotide bases. For such problems, each iteration can be restricted to one pass over the known finite set.

In *restricted iteration*, the start of the iteration is fixed; the update procedure is fixed; and the termination predicate is fixed. None of these three elements are susceptible to evolutionary modification. Consequently, excessively long iterations, nested iterations, and infinite loops are impossible. The amount of computer time therefore becomes capped (and thus tolerable). An overall computer program may consist of zero, one, or more iteration-performing branches and one result-producing branch (as well as function-defining branches).

When the overall program is executed, each iteration-performing branch is executed exactly once. That is, each iteration-performing branch performs one restricted iteration over the finite set. The value returned by the last execution of the body of the iteration is available as a terminal (called IPB0 for the first iteration-performing branch, IPB1 for the second, etc.). Then, the result-producing branch is executed. The result-producing branch may call (on zero, one, or multiple occasions) the function-defining branches, if any, of the overall program. The result-producing branch (and possibly even function-defining branches that may be created during the run) may refer (on zero, one, or multiple occasions) to the values (i.e., IPB0, IPB1, etc.) returned by the iteration-performing branches.

There are two avenues of communication between the iteration-performing branch(es) and the result-producing branch of the overall program. First, the function and terminal sets of the problem contain means to read and write memory variables. Thus, an iteration-performing branch may write a value to a particular memory variable and the result-producing branch may read that value. Second, the terminals, IPB0, IPB1, ..., may appear in the result-producing branch (and possibly even in function-defining branches that may be created during the run).

Given that restricted iteration is available, it may not be obvious in advance whether the solution of a particular problem requires any restricted iterations, exactly one such restricted iteration, or multiple restricted iterations. It would therefore be desirable to automate the decision as to whether to employ iteration at all, how many times to employ iteration, and the particular sequence of steps in each iteration. This automation can be realized using the new architecture-altering operation of restricted iteration creation.

The steps in the operation of restricted iteration creation are as follows:

(1) Select a program from the population to participate in this operation. The step of selecting this parental program for restricted iteration creation 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 this operation than a less fit program. A copy is first made of the selected program and the operation is then performed on the copy, so the original selected program remains unchanged in the population and is therefore available to be selected again for this (or any other) operation on the basis of its fitness.

(2) Randomly pick one of the branches of the selected program. If the selected program has only one branch, that branch is automatically picked. The picking of the branch may, as an option, be restricted to a result-producing branch or another category of branches.

(3) Randomly choose a point in the picked branch of the selected program. This chosen point will become the top-most point of the body of the to-be-created restricted iteration-performing branch.

((4) Add a uniquely-named new restricted iteration-performing branch to the selected program consisting of the subtree rooted at the chosen point, thus increasing, by one, the number of iteration-performing branches in the selected program.

(5) Replace the chosen point in the picked branch by the name of the new iteration-performing branch.

4. Application of Restricted Iteration Creation to the Transmembrane Segment Identification Problem

The transmembrane segment identification problem is one of the more difficult problems to which genetic programming has previously been applied. Genetic programming has been previously applied to three versions of this problem:

(1) the set-creating version using genetic programming with pre-specification by the user of the architecture consisting of three zero-argument automatically defined functions and one iteration-performing branch (ch. 18.5 through 18.9 of Koza 1994a),

(2) the arithmetic-performing version using genetic programming with pre-specification by the user of the architecture consisting of three zero-argument automatically defined functions and one iteration-performing branch (ch. 18.10 and 18.11 of Koza 1994a),

(3) the architecture-altering version using the six recently developed architecture-altering operations of branch duplication, argument duplication, branch deletion, argument deletion, branch creation, and argument creation (Koza and Andre 1996).

In this section, the new architecture-altering operation of restricted iteration creation is applied to the transmembrane segment identification problem.

4.1. The Transmembrane Segment Identification Problem

Proteins are polypeptide molecules composed of sequences of amino acids (Stryer 1995). 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).

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 (Yeagle 1993). 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. Since 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 goal in the transmembrane segment identification problem is to classify a given protein segment (i.e., a subsequence of amino acid residues from a protein sequence) as being a transmembrane domain or non-transmembrane area of the protein (without using biochemical knowledge concerning hydrophobicity used by human-written algorithms for this task).

A correct classification cannot be made by merely examining a particular position in the given protein segment, by merely testing for the presence or absence of any one particular amino acid residue in the segment, or by merely analyzing any small combination of positions within the segment. Success in this problem involves integrating information over the entire protein segment. Thus, this problem seems especially appropriate for illustrating the new operation of restricted iteration creation.

4.2. Preparatory Steps

We decided to start the run of this problem with an architecturally uniform initial random population with a "minimalist" structure in which each program in the population consists only of a single result-producing branch, RPB. That is, there are no automatically defined functions (ADFs) and no iteration-performing branches (IPBs) in the population at generation 0. Any automatically defined functions or iteration-performing branches that are needed to solve the problem will have to be created dynamically during the run of genetic programming.

The transmembrane segment identification problem deals with protein segments, each of which is of a known, finite length. Therefore, we decided that each iteration will be restricted in the sense that it will consist of one pass over the current protein segment. This decision to use restricted iteration effectively caps the amount of computer time that can be expended in evaluating any one program. Specifically, the iteration will start by pointing to the first position of the protein segment; the transition rule for the iteration will consist of advancing the pointer to the next position of the protein segment; and the iteration will terminate when it points to the last position of the protein segment. We also decided that the iteration-performing branch(es) would not possess an explicit iterative index,

but, instead, residue-detecting functions would be used to sense the presence or absence of a particular amino acid residue at the current position of the protein segment.

There are a total of 51 functions and terminals in the function set and the terminal set of this problem, including 12 initial functions, 28 initial terminals, four potential functions, and seven potential terminals.

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

$$\mathcal{F}_{\text{initial}} = \{+, -, *, \%, \text{IFGTZ}, \text{ORN}, \text{SETM0}, \text{SETM1}, \text{SETM2}, \text{SETM3}, \text{SETM4}, \text{SETM5}\}$$

taking 2, 2, 2, 2, 3, 2, 1, 1, 1, 1, 1, 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 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 six one-argument setting functions, SETM0, SETM1, ..., SETM5, can be used to set the six settable memory variables, M0, M1, ..., M5 to a particular value. These setting functions operating on specific settable variables (Koza 1992, 1994a) are 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).

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

$$\mathcal{T}_{\text{initial}} = \{\leftarrow, \text{M0}, \text{M1}, \text{M2}, \text{M3}, \text{M4}, \text{M5}, \text{LEN}, (\text{A?}), (\text{C?}), \dots, (\text{Y?})\}.$$

← represents floating-point random constants between -10.0 and +10.0 chosen using a uniform probability distribution. 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 and terminals (initial and potential). These 200 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, M1, M2, M3, M4, and M5 are a settable memory variables. Each is zero when execution of a given overall program begins.

LEN is 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 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 an iteration-performing branch), the residue-detecting function is evaluated for the current residue of the iteration.

There are no iteration-performing branches at generation 0. However, once the restricted iteration creation operation is performed, iteration-performing branches (and the terminals representing their return values, IPB0, IPB1, ...) begin to appear in the population.

Similarly, there are no automatically defined functions (ADF0, ADF1, ...) or dummy variables (ARG0, ARG1, ...) in generation 0. However, once the architecture-altering operations are performed, the functions ADF0, ADF1, ... and the dummy variables ARG0, ARG1, ... begin to appear in the population. For practical reasons, a maximum of 3 iteration-performing branches was established. Similarly, a maximum of four automatically defined functions, each possessing between zero and four dummy variables, was established. Thus, the set of potential additional terminals, $\mathcal{T}_{\text{potential}}$, for this problem consists of

$$\mathcal{T}_{\text{potential}} = \{IPB0, IPB1, IPB2, ARG0, ARG1, ARG2, ARG3\}.$$

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).

Note that shortly after iteration-performing branch(es) and automatically defined function(s) are created, the

residue-detecting functions will begin to migrate into these newly created branches. Moreover, the automatically defined functions will be referenced by yet-to-be-created calls from the result-producing branch or iteration-performing branch(es). When a residue-detecting function appears within a created automatically defined function that is called from within a created iteration-performing branch, it is, of course, evaluated for each position of the protein segment as the iteration proceeds. When a residue-detecting function appears in a program with no iteration-performing branches (i.e., in the result-producing branch or a created automatically defined function), the residue to which it is pointing is formally undefined. As a matter of convention here, it is evaluated for the first position of the protein segment (i.e., as if the iterative index is pointing to the first residue of the protein sequence). When a residue-detecting function appears within the result-producing branch or a created automatically defined function of a program with one or more iteration-performing branches, the residue to which it is pointing is also undefined. As a matter of convention here, it is evaluated for the leftover value of the iterative index (i.e., as if the iterative index is pointing to the last residue of the protein sequence).

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.

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.

4.2.2. *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 $(1 - C)/2$. 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 ch. 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*).

4.2.3. Parameters

Population size, M , was 64,000.

The operation of restricted iteration creation is, like the other architecture-altering operations, used sparingly on each generation. The percentage of operations on each generation after generation 6 was 85% crossovers; 10% reproductions; 0% mutations; 1% restricted iteration creations; 1% branch duplications; 1% argument duplications; 0.5% branch deletions; 0.5% argument deletions; 1% branch creations; and 0% argument creations. Since we did not want to waste large amounts of computer time in early generations where only a few programs have any automatically defined 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 6 to 70% crossovers; 10% reproductions; 0% mutations; 6% restricted iteration creations; 2% branch duplications; 2% argument duplications; 2% branch deletions; 2% argument deletions; 6% branch creations; 0% argument creation.

A maximum size of 200 points was established for the result-producing branch, each of the yet-to-be-created iteration-performing branches, and each of the yet-to-be-created function-defining branches. The other parameters for the runs were the default values specified in Koza (1994a).

4.2.4. Termination Criterion and Results Designation

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

4.2.5. Parallel Implementation

The problem (coded in ANSI C) was run on a medium-grained parallel Parystec computer system consisting of 64 80 MHz 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 1989) was used. That is, subpopulations (called *demes* after Wright 1943) were situated at the processing nodes of the system. Population size was $Q = 1,000$ at each of the $D = 64$ demes for a total population size of 64,000. 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.

4.3. Results

It is difficult to understand the operation of most of the programs that are evolved using genetic programming. One practical way to obtain understandable evolved programs is to harvest more than the usual single high-fitness program from a run. When this approach is used, the run is not terminated as soon as the first high-fitness program is created, but is, instead, continued until a number of high-fitness programs have been created. For simplicity, we limited our harvesting to pace-setting best-of-generation programs reported from the 64 processing nodes of the parallel computer system. Specifically, we harvested five different evolved programs from generations 34, 37, 40, 42, and 43 of our first run of this version of the problem. The programs harvested from generations 40 and 42 had an out-of-sample error rate of 1.6% and the other three harvested programs had an out-of-sample error rate of 2%. All five of these programs were superior to the algorithms written by knowledgeable human investigators (which had error rates of between 2.5% and 2.8%). Other subsequent runs of this problem also produced equally successful results.

4.3.1. The Myopic Performance of the Best of Generation 0

The initial random population of a run of genetic programming is a blind random search of the search space of the problem. As such, it provides a baseline for comparing the results of subsequent generations. Figure 1 shows that the architecture of the best-of-generation program consists only of one result-producing branch, RPB.

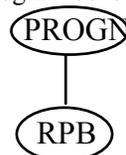


Figure 1 Architecture of best-of-generation program from generation 0.

The best-of-generation program for generation 0 has an in-sample correlation of 0.3108. This 18-point program consists only of the result-producing branch shown below:

```

(setm2 (* (setm5 (setm0 (orn LEN M0)))
  (* (* (setm4 LEN) (setm4 (M?))) (%
    (setm1 (W?)) (setm4 (V?))))))
  
```

When simplified, this program returns +1 if the first residue of the protein segment is M (methionine), V (valine), or W (tryptophan), but returns -1 otherwise. M and V are hydrophobic on the Kyte-Doolittle hydrophobicity scale (Kyte and Doolittle 1982) and W is neutral. Note that this myopic program makes a decision for the entire protein segment (whose average length is 22) based on this manifestly inadequate test applied to a manifestly inadequate portion (only one residue) of the protein segment.

4.3.2. *A Myopic Iteration-Performing Branch*

One of the pace-setting programs from generation 1 has a 26-point result-producing branch, a 14-point iteration-performing branch, and an in-sample correlation of 0.4702. (A pace-setting program in an asynchronous parallel computer system is a best-of-generation program from one of the processing nodes that reports a new best level of fitness). However, even though the program has an iteration-performing branch, the classification of the entire protein segment is myopically done on the basis of just the last residue from the protein segment.

Figure 2 shows the architecture of this program consisting of one iteration-performing branch, IPB0, and one result-producing branch.

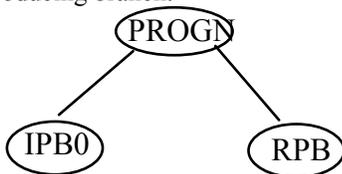


Figure 2 Architecture of program from generation 1.

4.3.3. *An Iteration-Performing Branch that Globally Integrates Information*

A later pace-setting program from generation 1 uses its newly created iteration-performing branch to achieve an in-sample correlation of 0.5760 by globally integrating information about the protein segment. However, its method is most unusual. Its three-point result-producing branch is

```
(orn (IPB0) (L?))
```

Its six-point iteration-performing branch, IPB0, is

```
(% (setm3 (orn (K?) M3)) (E?))
```

The amino acid residue K (lysine) is electrically charged and hydrophilic and therefore rarely occurs in a transmembrane domain. The settable variable M3 is iteratively set to the ORN of the previous value of M3 and the value (-1 or +1) returned by the residue-detecting function (K?). The effect is to scan the protein segment (averaging 22 residues) for the absence of K's since the value returned by the SETM3 on the final iteration is -1 if there are no K's in the segment. The test for E (glutamic acid) does not affect M3 and thus only becomes relevant for the last residue of the segment. If there is no E at the end of the segment (and there usually would not be in a transmembrane domain), the value returned by the iteration-performing branch, IPB0, is +1 if there are no K's in the segment. The result-producing branch then classifies the entire segment as a transmembrane domain if either the last residue is the hydrophobic residue L (leucine) or if there is an absence of hydrophilic K's in the segment.

All succeeding pace-setting programs have at least one iteration-performing branch that globally integrates information about the entire protein segment in some way.

4.3.4. *An Iteration-Performing Branch that Computes a Running Sum*

The first pace-setting program from generation 2 globally integrates information about the protein segment and achieves an in-sample correlation of 0.7224. Its one-point result-producing branch simply returns the value of IPB0 and its eight-point iteration-performing branch, IPB0, is

```
(setm3 (+ (* (H?) (E?)) (+ (V?) M3)))
```

This iteration-performing branch, IPB0, computes a running sum, M3. Each hydrophobic V residue (+4.2 on the Kyte-Doolittle scale) contributes +1; each residue that is neither E (-3.5 on the scale) nor H (-3.2 on the scale) contributes +1; an E or a H contributes -1.

4.3.5. *Emergence of Automatically Defined Functions*

The pace-setting program from generation 6 consists of a one-argument automatically defined function, an iteration-performing branch, and a result-producing branch. Figure 3 shows the architecture of this program from generation 6.

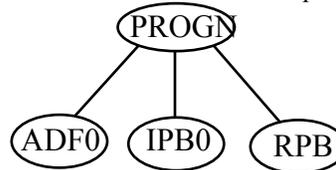


Figure 3 Architecture of program from generation 6.

Of course, automatically defined functions were created as early as generation 1; however, this program was the first pace-setting program with an automatically defined function.

4.3.6. *Emergence of Multiple Iteration-Performing Branches*

The first pace-setting program from generation 8 has multiple iteration-performing branches. One of these iteration-performing branches globally integrates information over the entire protein segment.

4.3.7. *Emergence of Cooperativity Among Iteration-Performing Branches*

The second pace-setting program from generation 11 has two iteration-performing branches that cooperatively integrate global information about the protein segment.

Its 12-point first iteration-performing branch, IPB0, is

```
(setm3 (+ (* (H?) (E?)) (+ (orn (setm2 M0) (set2 (W?))) M3)))
```

This first branch, IPB0, computes a running sum, M3. An increment of +1 is contributed by W (tryptophan); +1 is contributed by each residue that is neither E nor H; and -1 is contributed by either an E or a H (histidine). The settable variable, M3, is used for communication between the first and second iteration-performing branches.

The eight-point second iteration-performing branch, IPB1, (which, interestingly, is identical to IPB0 of the program from generation 2 cited above) makes an additional contribution to M3 based on H, E, and V (valine) as follows:

```
(setm3 (+ (* (H?) (E?)) (+ (V?) M3)))
```

The one-point result-producing branch of this program is simply (IPB1). That is, the terminal IPB1 is used for communication between the second iteration-performing branch and the final result-producing branch. The value of the result-producing branch is the running sum to which +1 is contributed by each V; +1 is contributed by each W; +2 is contributed by each residue that is neither E nor H; and -2 is contributed by either an E or a H. Note that a human programmer would never use two cooperative iteration-performing branches to compute this running sum M3. However, in this particular program, one iteration-performing branch enhances the performance of the other.

4.3.8. Emergence of Hierarchy among Automatically Defined Functions

A pace-setting program from generation 24 has a one-argument ADF1 and a zero-argument ADF3 such that ADF3 refers to ADF1 (and also to IPB1).

4.3.9. Emergence of Multiple Automatically Defined Functions and Multiple Iteration-Performing Branches

The pace-setting program from generation 26 has three one-argument automatically defined functions as well as two iteration-performing branches. Figure 4 shows this program

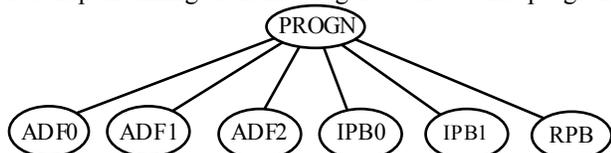


Figure 4 Architecture of program from generation 26.

4.3.10. A Best-of-Run Program from Generation 42

The best-of-generation program from processing node 2 of generation 42 scores 122 true positives, 122 true negatives, 1 false positive, and 1 false negative and has an in-sample correlation of 0.9938. It also scores 123 true positives, 123 true negatives, 2 false positives, and 2 false negatives. It has an out-of-sample error rate of 1.6%.

This program has two one-argument automatically defined functions (ADF0 and ADF1), two iteration-performing branches (IPB0 and IPB1) that cooperatively integrate global information about the protein segment, and one result-producing branch.

Figure 5 shows the architecture of this program.

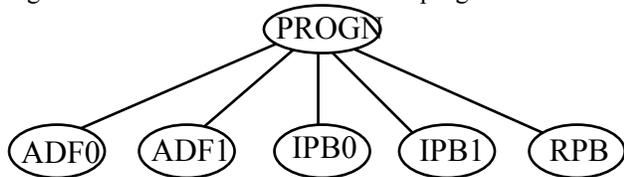


Figure 5 Architecture of program from generation 42.

The one-point result-producing branch returns the value returned by the second iteration-performing branch, IPB1.

The first automatically defined function, ADF0, has six points and is shown below:

```
(adf1 (+ (setm0 (E?))(setm4 (Q?))))
```

Since ADF1 merely returns its one argument, ADF0 returns 0 if the current residue is E or Q (glutamine) and otherwise returns -2. ADF0 also side-effects the settable variables M0 and M4.

The first iteration-performing branch, IPB0, has 112 points and is shown below:

```
(setm1 (- (- (setm1 (setm1 (- (setm1 M1)
(setm3 (setm3 (% (- (I?) (R?)) (adf0
(H?))))))))) (setm3 (setm3 (% (- (+
(V?) M3) (setm2 (+ (- (D?) (+ (V?)
(setm3 (+ (orn (Y?) (* (E?) (setm5
(orn (P?) (D?)))))))+(setm5 (orn M0
(L?)) M3)))))) (setm3 (R?)))))) (adf0
(% (setm1 (- (- (setm1 (setm1 (-
(setm1 M1) (setm3 (setm3 (% (- (I?)
(R?)) (adf0 (H?))))))))) (setm3 (setm3
(% (- (+ (V?) M3) (setm2 (+ (- (*
(setm5 (orn (P?) (R?))) (setm5 (orn
(P?) (D?)))) (L?)) (setm3 (orn (Q?) (%
M5 (V?))))))))) (setm5 (orn M0
(L?)))))) (setm3 (setm3 (% (- (F?)
(R?))(adf0 (H?)))))) (E?))))))
(setm3 (setm3 (% (- (F?) (R?))(adf0
(H?))))))
```

The second iteration-performing branch, IPB1, has 45 points and is shown below:

```
(setm1 (- (setm1 M1) (setm3 (setm3 (% (-
(I?) (adf1 (* (setm0 (setm1 (orn (orn
(P?) (R?)) (- (setm1 M1) (setm3 (setm3
(iftgtz (setm4 (- (Y?) (R?))) (setm1
(Y?) IPB0)))))) (setm0 (* (setm0
(orn (K?) M0) (setm1 (orn (setm4
(setm1 (setm4 (P?)))) (Q?)))))))))
(adf0 (H?))))))
```

Both possible avenues of communication and cooperation are employed by this program. First, two of the six settable variables (M0 and M1) are set in IPB0 and referenced by IPB1 (as highlighted by bold-faced type in IPB1). Second, IPB1 contains a reference to the value returned by IPB0 (also highlighted by bold-faced type in IPB1).

5. Comparison of Eight Methods

Table 1 shows the out-of-sample error rate for eight different approaches to the transmembrane segment identification problem, including

(1) the three human-written algorithms of von Heijne (1992), Engelman, Steitz, and Goldman (1986), and Kyte and Doolittle (1982) for classifying transmembrane domains, as described in Weiss, Cohen, and Indurkha 1993,

(2) the result of Weiss, Cohen, and Indurkha (1993) using a machine learning technique along with a considerable amount of human ingenuity,

(3) the set-creating version using genetic programming with prespecification by the user of the architecture consisting of three zero-argument automatically defined

functions and one iteration-performing branch (ch. 18.5 through 18.9 of Koza 1994a),

4) the arithmetic-performing version using genetic programming with prespecification by the user of the architecture consisting of three zero-argument automatically defined functions and one iteration-performing branch (ch. 18.10 and 18.11 of Koza 1994a),

(5) the architecture-altering version using genetic programming employing the six recently developed operations of branch duplication, argument duplication, branch deletion, argument deletion, branch creation, and argument creation (Koza and Andre 1996), and

(6) the result using the new operation of restricted iteration creation (and the six recently developed architecture-altering operations), as described in this paper.

Table 1 Comparison of eight methods.

Method	Error
von Heijne 1992	2.8%
Engelman, Steitz, and Goldman 1986	2.7%
Kyte and Doolittle 1982	2.5%
Weiss, Cohen, and Indurkha 1993	2.5%
GP + Set-creating ADFs	1.6%
GP + Arithmetic-performing ADFs	1.6%
GP + ADFs + six architecture-altering operations	1.6%
GP + ADFs + six architecture-altering operations + restricted iteration creation operation	1.6%

As can be seen from the table, the error rate of all four versions using genetic programming are identical; all four are better than the error rates of the other four methods. All four 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.

6. Conclusion

The new architecture-altering operation of restricted iteration creation (along with the six previously developed architecture-altering operations) enable genetic programming to evolve a successful classifying program for the transmembrane segment identification problem starting from a population that initially contains no iterations and no automatically defined functions. That is, this new operation automates the decision about whether to employ iteration, how much iteration to employ, and the particular sequence of iterative steps. This reduces the number of decisions that the user must make prior to using genetic programming on a problem.

In the illustrative run, the first occurrence of an iteration-performing branch was degenerate and myopic. However, global integration of information by a single iteration-performing branch soon emerged. We then saw the emergence of multiple iteration-performing branches and,

finally, cooperation among the iteration-performing branches. Meanwhile, we also saw the emergence of a first automatically defined function, then multiple automatically defined functions, and finally hierarchical arrangements of automatically defined functions.

Bibliography

- Andre, David (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. 250–255.
- Andre, David and Koza, John R. 1996. 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 2*. 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. 1989. *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. (2nd ed. 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. 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). 1995. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. 695–717.
- Koza, John R. and Andre, David. 1996. Classifying protein segments as transmembrane domains using architecture-altering operations in genetic programming. Angeline, Peter and Kinnear, Kenneth E., Jr. (editors). 1996. *Advances in Genetic Programming 2*, Cambridge, MA: The MIT Press.
- 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.
- 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*. 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 Indurkha, 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.