Two Ways of Discovering the Size and Shape of a Computer Program to Solve a Problem

John R. Koza
Computer Science Department
Stanford University
Stanford, California 94305
Koza@CS.Stanford.Edu   415-941-0336
http://www-cs-faculty.stanford.edu/~koza/

Abstract

The requirement that the user of a problem-solving paradigm prespecify the size and shape of the ultimate solution to a problem has been a bane of automated machine learning from the earliest times. This paper compares two techniques for automatically discovering the architecture of a multi-part computer program while concurrently solving the problem during a run of genetic programming. In the first technique, called *evolutionary selection*, the initial random population is architecturally diverse and there is a competitive *selection* among the various architectures during the run. In the second technique, called *evolution of architecture*, six new architecture-altering operations provide a way to *evolve* the architecture of a multi-part program in the sense of *actually changing* the architecture of the program dynamically during the run. The new architecture-altering operations are motivated by the naturally occurring operation of gene duplication, as described in Susumu Ohno's provocative book *Evolution by Means of Gene Duplication*, as well as the naturally occurring operation of gene deletion.

1. INTRODUCTION

The requirement that the user of a problem-solving paradigm prespecify the size and shape of the ultimate solution to a problem has been a bane of automated machine learning from the earliest times (Samuel 1959).

In nature, sexual recombination (crossover) exchanges alleles (gene values) at particular locations (loci) along the chromosome (a molecule of DNA). The DNA then controls the manufacture of various proteins that determine the structure, function, and behavior of the living organism. The resulting organism then spends its life attempting to grapple with its environment. Some organisms in a given population do better than others and are able to survive to the age of reproduction, produce offspring, and thereby pass on all or part of their genetic make-up to the next generation of the population. Over many generations, the population as a whole evolves so as to give increasing representation to traits (and, more importantly, co-adapted *combinations* of traits) that contribute to survival of the organism and the fruitful production of offspring. This process, which Charles Darwin called *natural selection*, tends to evolve near-optimal co-adapted sets of alleles in the chromosomes of the organism (given its environment).

John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975) described how an analog of the naturally-occurring evolutionary process can be applied to solving artificial problems using what is now called the *genetic algorithm*. Before applying the genetic algorithm to the problem, the user designs an artificial chromosome of a certain fixed size and then defines a mapping (encoding) between the points in the search space of the problem and instances of the artificial chromosome. For example, in applying the genetic algorithm to a multidimensional optimization problem (where the goal is to find the global optimum of an unknown multidimensional function), the artificial chromosome may be a linear character string (modeled directly after the linear string of information found in DNA). A specific location (a gene) along this artificial chromosome is associated with each of the variables of the problem. Character(s) appearing at a particular location along the chromosome denote the value of a particular variable (i.e., the gene value or allele). Each individual in the population has a fitness value (which, for a multidimensional optimization problem, is the value of the unknown function). The genetic algorithm then manipulates a population of such artificial chromosomes (usually starting from a randomly-created initial population of strings) using the operations of reproduction, crossover, and mutation. Individuals are probabilistically selected to participate in these genetic operations based on their fitness. The goal of the genetic algorithm in a multidimensional optimization problem is to find an artificial chromosome which, when decoded and mapped back into the search space of the problem, corresponds to a globally optimum (or near-optimum) point in the original search space of the problem.

The evolutionary process described above indicates how a globally optimum combination of alleles (gene values) within a fixed-size chromosome can be evolved.

In both the natural and artificial evolutionary processes, the crossover operation merely exchanges alleles (gene values) at particular locations along an *already-existing* fixed-size chromosome. The above description does not address the question of how totally new structures, new functions, new behaviors, and new species arise. Of course, 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 and thereby create new and more complex organisms). The emergence of new proteins corresponds to a change in 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, Brooks Low 1988).

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

At IJCAI-89, genetic programming was proposed as a domain-independent method for evolving computer programs that solve, or approximately solve, problems (Koza 1989). Genetic programming is an extension of the genetic algorithm in which the genetic population consists of computer programs (Koza 1992; Koza and Rice 1992; and Kinnear 1994). Genetic programming extended the trend toward increased complexity in the entities undergoing adaptation in the genetic algorithm that started with Holland's proposed broadcast language (1975), the classifier system (Holland and Reitman 1978), Steven F. Smith's variable-length entities (1980), Nichael Cramer's highly innovative and creative experiments in program induction (1985), Hicklin's reproduction and mutation of programs (1986), Cory Fujiki's application of all genetic operations to programs(1986), Fujiki and Dickinson's induction of if-then clauses for playing the iterated prisoner’s dilemma game (1987), Antonisse and Keller 's work in applying genetic methods to higher-level representations (1987), and Bickel and Bickel's application of genetic methods to if-then expert system rules (1987).

The programs evolved by genetic programming may be single-part programs (containing merely one result-producing branch) or multi-part programs (containing one or more main *result-producing branches* and one or more *function-defining branches*. An *automatically defined function* (*ADF*) is a function (i.e., subroutine, subprogram, `DEFUN`, procedure, module) that is 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 or other ADF) that is simultaneously being evolved (Koza 1994a, 1994b).

When single-part programs are involved, genetic programming automatically determines the size and shape of the solution (i.e., the size and shape of the program tree) as well as the sequence of work-performing primitive functions that can solve the problem. However, when multi-part programs and automatically defined functions are being used, the question arises as to how to determine the architecture of the programs that are being evolved. The *architecture* of a multi-part program consists of the number of function-defining branches (automatically defined functions) and the number of arguments (if any) possessed by each function-defining branch. When there is more than one function-defining branch in an overall program, the architecture also encompasses a specification of how the automatically defined functions may (or may not) hierarchically refer to one another.

In genetic programming (with or without automatically defined functions), sub-trees are exchanged by the crossover operation. Crossover is analogous to the exchanging of alleles (gene values) in a chromosome. The analog of a genome change corresponds to a dynamic alteration (during a run of genetic programming) of the architecture of an overall multi-part computer program.

Thus, the question arises as to 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 prespecify the architecture before the run starts). That is, is it possible to make the size and shape of the solution part of the *answer* provided to the user by an automated machine learning process, rather than part of the *question* formulated by the user.

Section 2 describes a first technique, called *evolutionary selection*, for determining the architecture of a multi-part program during a run of genetic programming. Section 3 describes naturally-occurring gene duplication. Section 4 describes a second technique, called *evolution of architecture*, that employs six new architecture-altering operations motivated by naturally-occurring gene duplication. Section 5 compares the two techniques as to computational effort, wallclock time, and average size of the evolved solutions (i.e., parsimony).

2. EVOLUTIONARY SELECTION OF THE ARCHITECTURE

One technique for creating the architecture of the overall program for solving a problem is to *evolutionarily select* the architecture dynamically during a run of genetic programming. This technique is described in chapters 21 – 25 of *Genetic Programming II* : *Automatic Discovery of Reusable Programs* (Koza 1994a). The technique of evolutionary selection starts with an architecturally diverse initial random population. As the evolutionary process proceeds, individuals with certain architectures may prove to be more fit than others at solving the problem. The more fit architectures will tend to prosper, while the less fit architectures will tend to wither away.

The architecturally diverse populations used with the technique of evolutionary selection require a modification of both the method of creating the initial random population and the two-offspring subtree-swapping crossover operation previously used in genetic programming. Specifically, the architecturally diverse population is created at generation 0 so as to contain randomly-created representatives of a broad range of different architectures. Structure-preserving crossover with *point typing* is a *one-offspring* crossover operation that permits robust recombination while guaranteeing that any pair of architecturally different parents will produce syntactically and semantically valid offspring.

2.1Example

The technique of evolutionary selection of the architecture can be illustrated by one run of the problem of symbolic regression of the Boolean even-5-parity function.

Generation 0 consists of randomly-generated architecturally-diverse programs. In the illustrative run, the best-of-generation program from generation 0 scores 18 hits and contains a four-argument `ADF0` and a three-argument `ADF1`. The *argument map* of the set of automatically defined functions belonging to a program is the list (ordered set) containing the number of arguments possessed by each automatically defined function in the overall program. This program, therefore, has an argument map of {4, 3}.

The argument map of the best-of-generation individual typically changes from generation to generation. In generations 1, 2, 3, and 4 of this run, the argument map of the best-of-generation individual is {4, 4}, {3, 4, 4}, {3, 4, 4}, and {3, 5}, respectively. The problem is solved on generation 5 of this run with a 100%-correct program with an argument map of {3, 5}. In this solution, `ADF0` is three-argument Boolean rule 195 which performs the even-2-parity function on two of the three arguments. `ADF1` is a five-argument Boolean rule that performs the odd-2-parity function on two of the five arguments.

Figure 1 is a three-dimensional *branch histogram* showing, by generation, the number of programs in the population (4,000 for this particular run) with a specified number (from 0 to 5) of automatically defined functions. As can be seen from the front row of skyscrapers in this histogram, there is approximate equality in the number of programs in the population with zero, one, two, three, four, or five automatically defined functions in generation 0 for the technique of evolutionary selection. However, by generation 5, two is the most common number of automatically defined functions for programs in this run. Only 2% of the individuals in the population have no automatically defined functions by generation 5. In fact, in most other runs of this problem, the programs with no automatically defined functions become extinct in the population before a solution is found. The ADF-less programs tend to disappear because they accrue fitness, from generation to generation, more slowly than the programs with automatically defined functions.

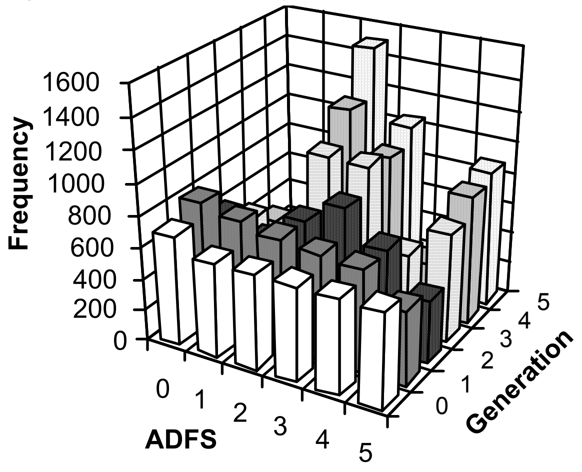

Figure 1 is a branch histogram for the technique of evolutionary selection of the architecture.

Note that, in the technique of evolutionary selection, although various different architectures are created at the initial random generation (generation 0), no new architectures are ever created (or altered) during the run.

3. GENE DUPLICATION IN NATURE

A gene duplication is an illegitimate recombination event that results in the duplication of a subsequence of DNA.

Susumu Ohno's seminal 1970 book *Evolution by Gene Duplication* proposed the provocative thesis that the creation of new proteins (and hence new structures and behaviors in living things) begins with a gene duplication and that gene duplication is

"the major force of evolution."

Gene duplications are rare and unpredictable events in the evolution of genomic sequences. The effect of a gene duplication is to create two identical ways of manufacturing the same protein. When a gene duplication occurs, there is no immediate change in the proteins that are manufactured by the living cell. In the terminology of computer science, gene duplication is a semantics-preserving operation. Then, over time, another genetic operation (e.g., mutation or crossover) may change one or the other of the two identical genes. Over short periods of time, the changes accumulating in a gene may have no practical effect. In fact, the changed part of the DNA will often not even produce a viable protein. However, as long as one of the two genes remains unchanged, the original protein manufactured from the unchanged gene will continue to be manufactured and the structure and behavior of the organism involved may continue as before. The changed gene is simply carried along in the DNA from generation to generation.

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. However, 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 viable 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 advantageous new protein, natural selection again starts to work to preserve that new gene.

Ohno's *Evolution by Gene Duplication* (1970) points out that ordinary 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).

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'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; 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; the protein of microtubules and actin of the skeletal muscle; and the light and heavy immunoglobin chains (Brooks Low 1988, Hood and Hunkapiller 1991, Dyson and Sherratt 1985).

The midge, *Chironomus tentans*, in figure 2, provides an additional example of gene duplication (Galli and Wislander 1993). In the sequence containing 3,959 nucleiotide bases of the DNA of *Chironomus tentans* (EMBL accession number X70063), the 732 nucleiotide bases located at positions 918–1,649 become expressed as a protein (called the "C. tentans Sp38–40.A" gene) containing 244 (i.e., one third of 732) amino acid residues. The 759 nucleiotide bases at positions 2,513–3,271 become expressed as a protein (called "C. tentans Sp38–40.B") containing 253 residues.

Figure 2 The midge, *Chironomus tentans*.

Both the "A" and the "B" proteins are secreted from the fly's salivary gland to form two similar, but different, kinds of water-insoluble fibers. The two kinds of fibers are, in turn, spun into one of two similar, but different, kinds of tubes. One tube is for larval protection and feeding while the other tube is for pupation. The two proteins are similar, but different. When the A" and the "B" proteins are aligned using the Smith-Waterman algorithm (Smith and Waterman 1981), there is 8l% identity between the two protein sequences. These two similar proteins arise as a consequence of a gene duplication. Immediately after the gene duplication occurred at some time in the distant past, there were two identical copies of the duplicated sequence of DNA. Over a period of millions of years since the initial gene duplication, additional mutations accumulated so that the two proteins are now only 81% identical (after alignment). More importantly, the two proteins now create different structures performing different functions in the fly.

More complex organisms have a general tendency to have more expressed proteins, more different kinds of structures, more complex structures, more different functions, and longer genomes (Dyson and Sherratt 1985). The new functions associated with gene duplication are consistent with the observed longer genomes of more complex organisms.

Gene deletion also occurs in nature. In gene deletion, there is a deletion of a portion of the linear 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 and there may be some change in the structure or behavior of the biological entity. 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, the change will tend to become extinct.

4. ARCHITECTURE-ALTERING OPERATIONS

Six new architecture-altering genetic operations provide a way of changing the architecture of the participating individuals changes during a run of genetic programming and thereby determining the architecture of a multi-part program dynamically during the run. Meanwhile, the Darwinian reproduction operation, the crossover operation, and the mutation operation continue to be performed.

4.1Branch 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.

Figure 3 shows an overall program consisting of one two-argument automatically defined function and one result-producing main branch (i.e., an argument map of {2}). Figure 4 shows the program resulting after applying the operation of branch duplication. Specifically, the function-defining branch 410 of figure 3 defining ADF0 (also shown as 510 of figure 4) is duplicated and a new function-defining branch (defining ADF1 at 540) appears in figure 4. There are two occurrences in figure 3 of invocations of the branch-to-be-duplicated, ADF0, in the result-producing branch of the selected program, namely ADF0 at 481 and 487. For each of these two occurrences, a random choice is made to either leave the occurrence of ADF0 unchanged or to replace it with the newly created ADF1. For the first invocation of ADF0 at 481 of figure 3, the choice is randomly made to replace ADF0 481 with ADF1 581 in figure 4. The arguments for the invocation of ADF1 581 are D1 582 and D2 583 in figure 4 (i.e., they are identical to the arguments D1 482 and D2 483 for the invocation of ADF0 at 481 as part of the original program in figure 3). For the second invocation of ADF0 at 487 of figure 3, the choice is randomly made to leave ADF0 unchanged. The new function-defining branch, ADF1, is identical to the previous function-defining branch, ADF0 (except for its name, ADF1, at 541 in figure 4). ADF1 is invoked with the very same arguments as ADF0 had been invoked. Consequently, the operation of branch duplication is a semantics-preserving operation.

Analogs of the naturally occurring operation of gene duplication have been previously used with genetic algorithms operating on character strings and with other evolutionary algorithms (Holland 1975; Goldberg, Korb, and Deb 1989; Lindgren 1991).

4.2Argument 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.

4.3Branch Creation

The operation of branch creation 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.

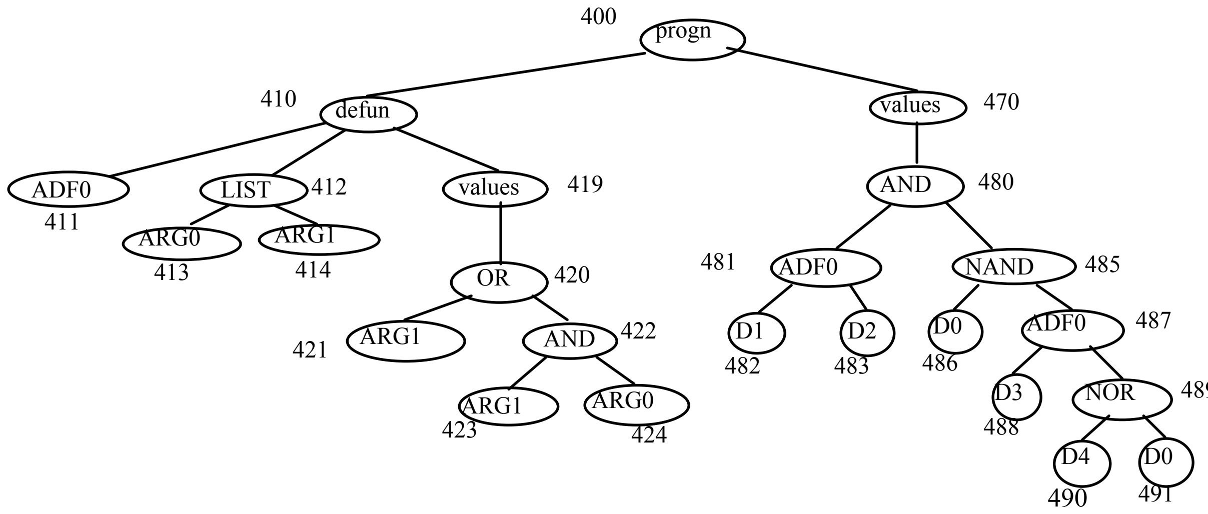

Figure 3 Program with an argument map of {2} consisting of one two-argument function-defining branch (ADF0) and one result-producing branch that invokes ADF0 twice (at 481 and 487).

This picked point becomes the top-most point of the body of the branch-to-be-created.

Details of this operation (and the other new operations below) are found in Koza 1994c. Branch creation is similar to, but different than, the compression (module acquisition) operation of Angeline and Pollack (1994).

4.4Argument Creation

The operation of argument creation creates a new dummy argument (formal parameter) within a function-defining branch of an overall program.

4.5Branch Deletion

The operations of argument duplication, branch duplication, branch creation, and argument creation create larger programs. The operations of argument deletion and branch deletion (described below) can create smaller programs and thereby counter-balance this growth.

The operation of branch deletion deletes one of the automatically defined functions. When a function-defining branch is deleted, the question arises as to how to modify invocations of the branch-to-be-deleted by the other branches of the overall program. One alternative (called *branch deletion with random regeneration*) is to randomly generate new subtrees composed of the available functions and terminals in lieu of the branch-to-be-deleted.

4.6Argument Deletion

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

4.7Creation of the Initial Random Population

When the architecture-altering operations are used, the initial population of programs may be created in any one of three ways. One possibility (called the "minimalist approach") is that each multi-part program in the population at generation 0 has a uniform architecture with exactly one automatically defined function possessing a minimal number of arguments appropriate to the problem. A second possibility (called the "big bang") is that each program in the population has a uniform architecture with no automatically defined functions (i.e., only a result-producing branch). This approach relies on the operation of branch creation to create multi-part programs during the run. A third possibility is that the population at generation 0 is architecturally diverse.

4.8Example

The architecture-altering operations will now be illustrated by a run of the even-5-parity problem. The run uses the “minimalist approach" in which each program in generation 0 consists of one result-producing branch and a two-argument function-defining branch.

On each generation there were 74% crossovers; 10% reproductions; 0% mutations; 5% branch duplications; 5% argument duplications; 0.5% branch deletion with random regeneration; 0.5% argument deletion with random regeneration; 5% branch creations; and 0% argument creations. Minor parameters were as in Koza 1994a. On generation 13, a 100%-correct solution emerged in the form of a computer program with an argument map of {3, 2}. The result-producing branch of this solution invokes both ADF0 and ADF1. Three-argument ADF0 (which had only two arguments when it started at generation 0) performs Boolean rule 106, a non-parity rule. Two-argument ADF1 (which did not exist at all in generation 0) is equivalent to the odd-2-parity function.

Figure 5 is a branch histogram for this run using the architecture-altering operations. It shows that all programs at generation 0 had an argument map of {2} and that the distribution of number of function-defining branches spread out on subsequent generations.

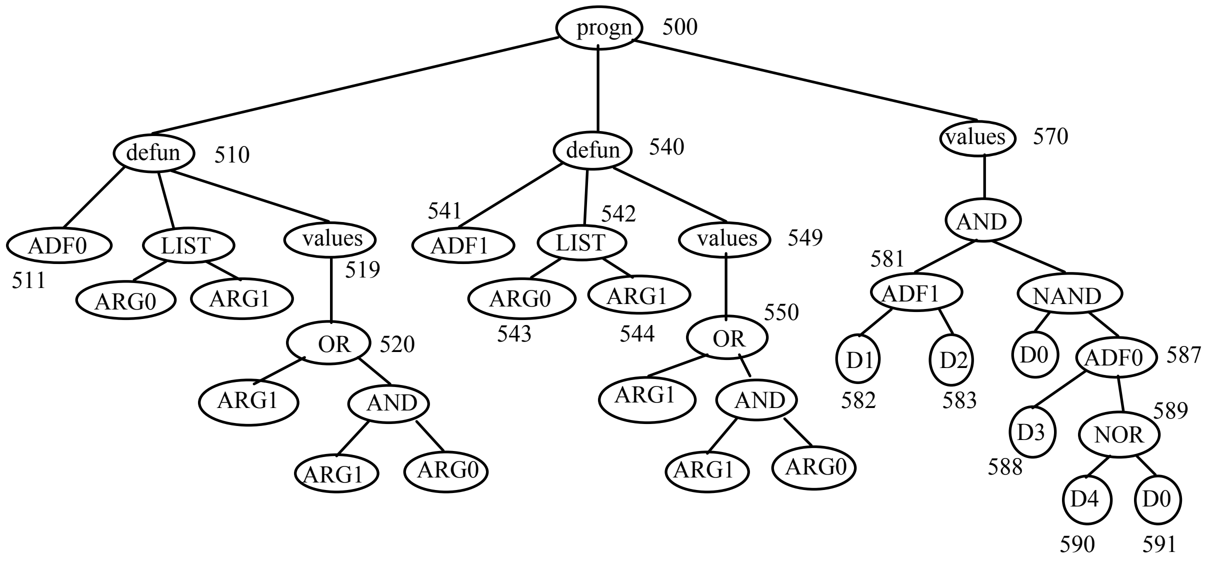

Figure 4 Program with an argument map of {2, 2} consisting of two two-argument function-defining branches (`ADF0` and `ADF1`) and one result-producing branch that invokes `ADF0` and `ADF1` one time each (at 581 and 587).

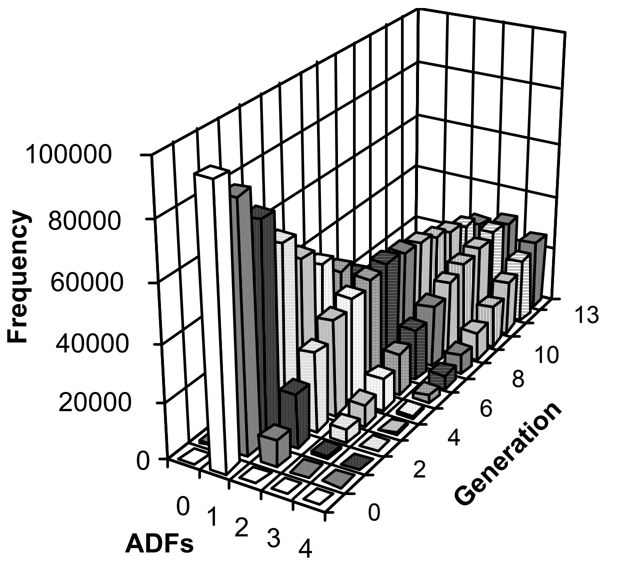

Figure 5 Branch histogram for technique of evolution of architecture using architecture-altering operations.

5. COMPARISON

We now use the even-5-parity problem to compare, over a series of runs, the performance of the architecture-altering operations for the following five approaches:

(A) **without automatically defined functions** (corresponding to the style of runs discussed throughout most of *Genetic Programming*),

(B) with automatically defined functions, **evolutionary selection** of the architecture (corresponding to the style of chapters 21–25 of *Genetic Programming II* on the evolutionary selection of the architecture), an architecturally diverse initial population, and structure-preserving crossover with point typing,

(C) with automatically defined functions, the **architecture-altering operations**, a potentially architecturally diverse population, and structure-preserving crossover with point typing,

(D) with automatically defined functions, a fixed user-supplied architecture (i.e., an argument map of {3, 2} that is known to be a good choice of architecture for this problem), and structure-preserving crossover with **point typing**,

(E) with automatically defined functions, the fixed known-good user-supplied architecture, and structure-preserving crossover with **branch typing** (corresponding to the style of most of *Genetic Programming II*).

Comparisons are made for computational effort, E (with 99% probability); wallclock time, $W(M,t,z)$ in seconds (with 99% probability); and the average structural complexity, $\bar{S}$ (all as described in detail in Koza 1994a).

The comparisons in table 1 all used a common population size, M, of 96,000 and a targeted maximum number of generations, G, of 76. The problem was run on a medium-grained parallel computer system. Different semi-isolated subpopulations (called *demes* in Wright 1943) are situated at the different processing nodes. The system consisted of a host PC 486 type computer running Windows and 64 Transtech TRAMs (containing one INMOS T805 transputer and 4 megabytes of RAM memory) arranged in a toroidal mesh. Generations were run asynchronously. There were D = 64 demes, a population size of Q = 1,500 per deme, and a migration rate (boatload size) of B = 8% (in each of four directions on each generation for each deme). Details of the parallel implementation are in Koza and Andre 1995.

As can be seen from table 1, both approaches for determining the architecture dynamically during the run (i.e., B and C) require less computational effort than solving the problem without automatically defined functions (approach A), but more computational effort than with the fixed, known-good, user-supplied architecture (approach E). That is, a price must be paid for dynamically determining the architecture. However, the price is intermediate between the extreme of not using ADFs at all and using ADFs with a fixed, known-good, user-supplied architecture. Approach B (evolutionary selection) is inferior, for this problem, to approach C (evolution of architecture) in that it requires greater computational effort, greater wallclock time, and produces bigger solutions (i.e., delivers less parsimony). In fact, the wallclock time for approach B exceeds that of the ADF-less approach (A) – perhaps because it involves so many inappropriate architectures.

Approach C produces the smallest-sized solutions, on average, for this problem (perhaps because it starts small and expands programs only as necessary). Approach E (using the most user-supplied information and the most economical form of tuping) requires the least computational effort.

Table 1 Comparison of the five approaches.

Approach	Runs	Computational effort E	Wallclock (M,t,z)	Average Size $\bar{S}$
A - No ADFs	14	5,025,000	36,950	469.1
B - ADFs + Evolutionary Selection of Architecture	14	4,263,000	66,667	180.9
C - ADFs + Architecture-Altering Operations	25	1,789,500	13,594	88.8
D - ADFs + Point Typing + Fixed Architecture	25	1,705,500	14,088	130.0

E - ADFs + Branch Typing + Fixed Architecture	25	1,261,500	6,481	112.2

Approach D isolates the additional computational effort required by point typing as opposed to branch typing (approach E). Since the computational effort for approaches C and D are virtually tied, the cost of the architecture-altering operations for this problem is about the same as the cost of point typing. Note that all four approaches (B, C, D, or E) employing ADFs require less computational effort, require less wallclock time, and are more parsimonious than the ADF-less approach.

Acknowledgements

David Andre and Walter Alden Tackett programmed the above. The picture of the midge comes from "Destructive and useful insects" by C.L. Metcalf and W.P Flint.

References

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

Antonisse, H. J., and Keller, K. 1987. Genetic operators for high-level knowledge representations. *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum.

Bickel, A. S. and Bickel, R. W. l987. Tree structured rules in genetic algorithms. *Proceedings of the Second International Conference on Genetic Algorithms*. Lawrence Erlbaum.

Brooks Low, K. 1988. Genetic recombination: A brief overview. In Brooks Low, K. (editor) *The Recombination of Genetic Material*. San Diego: Academic Press. P. 1–21.

Cramer, Nichael Lynn. l985. A representation for the adaptive generation of simple sequential programs. *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Lawrence Erlbaum.

Dyson, P. and Sherratt, D. 1985. Molecular mechanisms of duplication, deletion, and transposition of DNA. In Cavalier-Smith, T. (editor). *The Evolution of Genome Size*. Chichester: John Wiley & Sons.

Fujiki, Cory and Dickinson, J. l987. Using the genetic algorithm to generate LISP source code to solve the prisoner's dilemma. *Proceedings of the Second International Conference on Genetic Algorithms.* Lawrence Erlbaum.

Fujiki, Cory. l986. *An Evaluation of Holland's Genetic Algorithm Applied to a Program Generator*. M. S. thesis, Computer Science Dept. , University of Idaho.

Galli, Joakim and Wislander, Lars. 1993. Two secretory protein genes in *Chironomus tentans* have arisen by gene duplication and exhibit different developmental expression patterns. *Journal of Molecular Biology*. 231: 324–334.

Goldberg, David E., Korb, Bradley, and Deb, Kalyanmoy. 1989. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*. 3(5): 493–530.

Hicklin, J. F. 1986. Application of the Genetic Algorithm to Automatic Program Generation. M. S. thesis, Computer Science Dept. , University of Idaho.

Holland, John H. 1975. *Adaptation in Natural and Artificial Systems*. Ann Arbor, MI: University of Michigan Press.

Holland, John H., and Reitman, J. S. l978. Cognitive systems based on adaptive algorithms. In Waterman, D. A., and Hayes-Roth, Frederick, *Pattern-Directed Inference Systems*. Academic Press.

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: MIT Press.

Koza, John R. 1989. Hierarchical genetic algorithms operating on populations of computer programs. *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. San Mateo, CA: Morgan Kaufmann. Volume I. Pages 768-774.

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: MIT Press.

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

Koza, John R. and Andre, David. 1995. *Parallel Genetic Programming on a Network of Transputers*. Stanford University Computer Science Department technical report STAN-CS-TR-95-1542. January 30, 1995.

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

Lindgren, Kristian. 1991. Evolutionary phenomena in simple dynamics. In Langton, Christopher, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen (editors). *Artificial Life II, SFI Studies in the Sciences of Complexity*. Volume X. Redwood City, CA: Addison-Wesley. Pages 295-312.

Ohno, Susumu. 1970. *Evolution by Gene Duplication*. New York: Springer-Verlag.

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

Smith, Steven F. 1980. *A Learning System Based on Genetic Adaptive Algorithms*. Ph.D. dissertation, University of Pittsburgh.

Smith, T. F. and Waterman, M. S. 1981.Identification of common molecular subsequences. *Journal of Molecular Biology*. 147: 195-197.

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