Gene Duplication to Enable Genetic Programming to Concurrently Evolve Both the Architecture and Work-Performing Steps of a Computer Program

John R. Koza
Stanford University
Computer Science Department
Stanford, California 94305 U. S. A.
Koza@Cs.Stanford.Edu

Abstract

Susumu Ohno's provocative book *Evolution by Gene Duplication* proposed that the creation of new proteins in nature (and hence new structures and new behaviors in living things) begins with a gene duplication and that gene duplication is "the major force of evolution." This paper describes six new architecture-altering operations for genetic programming that are patterned after the naturally-occurring chromosomal operations of gene duplication and gene deletion. When these new operations are included in a run of genetic programming, genetic programming can dynamically change, during the run, the architecture of a multi-part program consisting of a main program and a set of hierarchically-called subprograms. These on-the-fly architectural changes occur while genetic programming is concurrently evolving the work-performing steps of the main program and the hierarchically-called subprograms. The new operations can be interpreted as an automated way to change the representation of a problem while solving the problem. Equivalently, these operations can be viewed as an automated way to decompose a problem into an non-pre-specified number of subproblems of non-pre-specified dimensionality; solve the subproblems; and assemble the solutions of the subproblems into a solution of the overall problem. These operations can also be interpreted as providing an automated way to specialize and generalize.

1 Introduction

The goal of automatic programming is to create, in an automated way, a computer program that enables a computer to solve a problem. This goal (attributed to Arthur Samuel in the 1950s) can be stated as follows:

> How can computers learn to solve problems without being explicitly programmed?

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 extends the biologically motivated *genetic algorithm* described in John Holland's pioneering *Adaptation in Natural and Artificial Systems* (1975).

Genetic programming starts with a primordial ooze of randomly generated computer programs composed of the available programmatic ingredients and then applies the principles of animal husbandry to breed a new (and often improved) population of programs. The breeding is done in a domain-independent way using the Darwinian principle of survival of the fittest and an analog of the naturally-occurring genetic operation of crossover (sexual recombination). The crossover operation is designed to create syntactically valid offspring programs (given closure amongst the set of ingredients). Genetic programming combines the expressive high-level symbolic representations of computer programs with the near-optimal efficiency of learning of Holland's genetic algorithm.

Genetic Programming: On the Programming of Computers by Means of Natural Selection (Koza 1992) provides evidence that genetic programming can solve, or approximately solve, a variety of problems from a variety of fields, including many benchmark problems from machine learning, artificial intelligence, control, robotics, optimization, game playing, symbolic regression, system identification, and concept learning. A videotape (Koza and Rice 1992) shows examples. Recent additional work is described in Kinnear 1994. The sequence of the work-performing steps of the programs being evolved by genetic programming is not specified in advance by the user. Instead, both the number and order of the work-performing steps are evolved as a result of selective pressure and the recombinative role of crossover. However, this first book *Genetic Programming* has the limitation that the vast majority of its evolved programs are single-part (i.e., one result-producing main part, but no subroutines).

1.1 Background on Automatically Defined Functions

1 believe that no approach to automated programming is likely to be successful on non-trivial problems unless it provides some hierarchical mechanism to exploit, *by reuse* and *parameterization,* the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines do this in ordinary computer programs.

Accordingly, *Genetic Programming II: Automatic Discovery of Reusable Programs* (Koza 1994a, 1994b) describes how to evolve multi-part programs consisting of a main program and one or more reusable, parameterized, hierarchically-called subprograms.

An *automatically defined function (ADF)* is a function (i.e., subroutine, subprogram, DEFUN, procedure, module)

that is dynamically evolved during a run of genetic programming and which may be called by a calling program (or subprogram) that is concurrently being evolved. When automatically defined functions are being used, a program in the population consists of a hierarchy of one (or more) *reusable* function-defining branches (i.e., automatically defined functions) along with a main result-producing branch. Typically, the automatically defined functions possess one or more dummy arguments (formal parameters) and are reused with different instantiations of these dummy arguments. During a run, genetic programming evolves different subprograms in the function-defining branches of the overall program, different main programs in the result-producing branch, different instantiations of the dummy arguments of the automatically defined functions in the function-defining branches, and different hierarchical references between the branches.

When automatically defined functions are being used in genetic programming, the initial random generation of the population is created so that every individual program has a constrained syntactic structure consisting of a particular architectural arrangement of branches. When crossover is to be performed, a *type* is assigned to each potential crossover point in the parental computer programs either on a branch-wide basis (called *branch typing)* or on the basis of the actual content of the subtree below the potential crossover point (called *point typing).* Crossover is then performed in a structure-preserving way (given closure) so as to ensure the syntactic validity of the offspring (Koza 1994a).

Genetic programming with automatically defined functions has been shown to be capable of solving numerous problems. More importantly, the evidence so far indicates that, for many problems, genetic programming requires less computational effort (i.e., fewer fitness evaluations to yield a solution with a satisfactorily high probability) with automatically defined functions than without them (provided the difficulty of the problem is above a certain relatively low break-even point). Also, genetic programming is usually yield solutions with smaller average overall size with automatically defined functions than without them (provided, again, that the problem is not too simple). That is, both learning efficiency and parsimony appear to be properties of genetic programming with automatically defined functions.

Moreover, there is also evidence that genetic programming with automatically defined functions is scalable. For several problems for which a progression of scaled-up versions was studied, the computational effort increases as a function of problem size at a *slower rate* with automatically defined functions than without them. In addition, the average size of solutions similarly increases as a function of problem size at a *slower rate* with automatically defined functions than without them. This observed scalability results from the profitable reuse of hierarchically-callable, parameterized subprograms within the overall program.

Five major preparatory steps required before genetic programming can be applied to a problem, namely determining (1) the set of terminals (i.e., the actual variables of the problem, zero-argument primitive functions, and constants, if any) for each branch, (2) the set of functions (e.g., primitive functions) for each branch, (3) the fitness measure (or other arrangement for implicitly measuring fitness), (4) the parameters to control the run, and (5) the termination criterion and result designation method.

1.2 The Problem of Architecture Discovery

When automatically defined functions are added to genetic programming, it is also necessary to determine the architecture of the yet-to-be-evolved programs. The specification of the architecture consists of (a) the number of function-defining branches 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.

Sometimes these architectural choices flow directly from the nature of the problem. Sometimes heuristic methods are helpful. However, in general, there is no way of knowing *a priori* the optimal (or sufficient) number of automatically defined functions that will prove to be useful for a given problem, or the optimal (or minimum) number of arguments for each automatically defined function, or the optimal (or sufficient) arrangement of hierarchical references.

If the goal is to develop a single, unified, domain-independent approach to automatic programming that requires that the user pre-specify as little direct information as possible about the problem, the question arises as to whether these architectural choices can be automated. Indeed, the requirement that the user predetermine the size and shape of the solution to a problem has been a bane of automated machine learning from the earliest times (Samuel 1959).

One way to automate these architectural choices for computer programs in general (called the technique of *evolutionary selection of architecture)* was described in chapters 21-25 of Koza 1994a. This technique starts with an architecturally diverse initial population (at generation 0) that has randomly-created representatives of a broad range of different architectures. As the evolutionary process proceeds, certain individuals with certain architectures will 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. Eventually a program with an appropriate architecture may emerge from this competitive and selective process. However, in this technique, no new architectures are ever dynamically created during the run. And, no architectures are ever dynamically altered during the run. There is only selection from amongst the architectures created at the beginning of the run.

This paper asks, and affirmatively answers, whether it is possible to enable genetic programming to dynamically alter the architecture of a multi-part program during a run while it is concurrently solving the given problem.

1.3 Recourse to Nature

A change in the architecture of a multi-part computer program during a run of genetic programming corresponds to a change in genome structure in the natural world. Therefore, it seems appropriate to consider the different ways that a genomic structure may change in nature.

In nature, sexual recombination ordinarily recombines a part of the chromosome of one parent with a homologous part of the second parent's chromosome. However, in certain rare and unpredictable occasions, recombination does not occur in this normal way. A gene duplication is an illegitimate recombination event that results in the duplication of a lengthy subsequence of a chromosome. Susumu Ohno's seminal book *Evolution by Gene Duplication* (1970) advanced the thesis that the creation of new proteins (and hence new structures and new behaviors in living things) begins with a gene duplication.

The six new architecture-altering operations for genetic programming described in this paper are motivated by the naturally occurring mechanisms of gene duplication and gene deletion in chromosome strings.

The six new architecture-altering operations can be viewed from five perspectives. First, the new architecture-altering operations provide a new way to solve the potentially vexatious problem of determining the architecture of the overall program in the context of genetic programming with automatically defined functions. Second, the new architecture-altering operations provide an automatic implementation of the ability to specialize and generalize in the context of automated problem-solving. Third, the new architecture-altering operations provide a way to automatically and dynamically change the representation of the problem while simultaneously solving the problem. Fourth, the new architecture-altering operations provide a way to automatically and dynamically decompose problems into subproblems and then automatically solve the overall problem by assembling the solutions of the subproblems into a solution of the overall problem. Fifth, the new architecture-altering operations provide a way to automatically and dynamically discover useful subspaces (usually of lower dimensionality than that of the overall problem) and then automatically assemble a solution of the overall problem from solutions applicable to the subspaces.

1.4 Outline of this Paper

Section 2 of this paper describes the naturally occurring processes of gene duplication and gene deletion. Section 3 describes the six new architecture-altering operations. Section 4 describes an actual run that solves that the problem of symbolic regression of the Boolean even-5-parity function while the architecture is being simultaneously evolved. Section 5 compares the computational effort required for five different ways of solving the problem. It concludes that the cost of automated architecture discovery is less than the cost of solving the problem without automatically defined functions (but more than that required with a fixed, user-supplied architecture that is known to be a good choice for this problem).

2 Gene Duplication in Nature

Gene duplications are rare and unpredictable events in the evolution of genomic sequences. In gene duplication, there is a duplication of a lengthy portion of the linear string of nucleiotide bases of the DNA in the living cell. After a sequence of bases that code for a particular protein is duplicated in the DNA, there are two identical ways of manufacturing the same protein. Thus, there is no immediate change in the proteins that are manufactured as a result of a gene duplication.

Over time, however, some other genetic operation, such as 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 be of no practical effect or value. As long as one of the two genes remains unchanged, the original protein manufactured from the unchanged gene continues 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.

Ohno's *Evolution by Gene Duplication* corrects the mistaken notion that natural selection is a mechanism for promoting change. Natural selection exerts a powerful force in favor of maintaining a gene that encodes for the manufacture of a protein that is important for the survival and successful performance 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 a particular protein. Over time, the second gene may accumulate additional changes and diverge more and more from the original gene. Eventually the changed gene may lead to the manufacture of a distinctly new and different protein that actually does affect the structure and behavior of the living thing in some advantageous or disadvantageous way. When a changed gene leads to the manufacture of a viable and advantageous new protein, natural selection again works to preserve that new gene.

Ohno also points out that ordinary point mutation and crossover are insufficient to explain major 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."

Ohno continues,

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

Ohno concludes,

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

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 distinctly different functions. Examples include trypsin and chymotrypsin; the protein of

microtubules and actin of the skeletal muscle; myoglobin and the monomeric hemoglobin of hagfish and lamprey; myoglobin used for storing oxygen in musde cells and the subunits of hemoglobin in red blood cells of vertebrates; and the light and heavy immunoglobin chains.

In gene deletion, there is a deletion of a subsequence of nucleiotide bases that would otherwise be translated into work-performing proteins in the cell.

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, page 116) suggested that intrachromosomal gene duplication might provide a means of adaptively modifying the effective mutation rate by making two or more copies of a substring of adjacent alleles within an overall string. Cavicchio (1970) used intrachromosomal gene duplication in early work on pattern recognition using the genetic algorithm. Gene duplication is implicitly used in the messy genetic algorithm (Goldberg, Korb, and Deb 1989). Lindgren (1991) analyzed the prisoner's dilemma game using an evolutionary algorithm that employed an operation analogous to gene duplication applied to strings. Gruau (1994) used genetic programming to develop a clever and innovative technique to evolve the architecture of a neural network at the same time as the weights are being evolved.

3 New Architecture-Altering Operations

The six new architecture-altering genetic operations provide a way of evolving the architecture of a multi-part program during a run of genetic programming. Meanwhile, Darwinian selection continues to favor the more fit individuals in the population to participate in the operations of crossover and mutation.

3.1 Branch Duplication

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

(1) Select a program from the population.

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

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

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

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

Figure 1 shows an overall program consisting of one two-argument automatically defined function and one result-producing main branch. Figure 2 shows the program resulting after applying the operation of branch duplication to Figure 1. Specifically, the function-defining branch 410 of Figure 1 defining ADFO (also shown as 510 of Figure 2) is duplicated and a new function-defining branch (defining ADF1) appears at 540 in Figure 2. There arc two occurrences of invocations of the branch-to-be-duplicated, ADFO, in the result-producing branch of the selected program, namely ADFO at 481 and 487 of Figure 1. For each occurrence, a random choice is made to either leave the occurrence of ADFO unchanged or to replace it with a reference to the newly created ADF1. For the first invocation of ADFO at 481 of Figure 1, the choice is randomly made to replace ADFO 481 with ADF1 581 in Figure 2. The arguments for the invocation of ADFI 581 are DI 582 and D2 583 in Figure 2 (i.e., they are identical to the arguments DI 482 and D2 483 for the invocation of ADFO at 481 in Figure 1). For the second invocation of ADFO at 487 of Figure 1, ADFO is left unchanged.

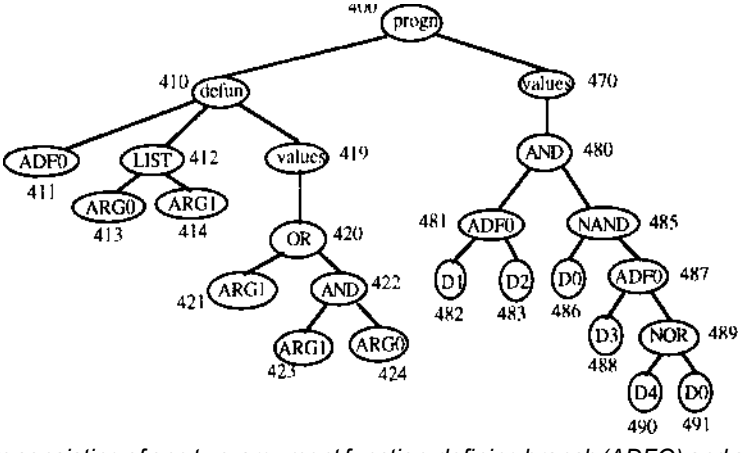

Figure J Program consisting of one two-argument function-defining branch (ADFO) and one result-producing branch.

The new branch is identical to the previously existing branch (except for the name ADF1 at 541 in Figure 2). Moreover, ADF1 at 581 is invoked with the same arguments as ADF0 at 481. Therefore, this operation does not affect the value returned by the overall program.

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

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

3.2 Argument Duplication

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

(1) Select a program from the population.

(2) Pick one of its function-defining branches.

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

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

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

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

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

The operation of argument duplication can also be interpreted as a case-splitting. The particular instantiations of the second and third arguments in the invocations of ADF0 provide potentially different ways of handling the two separate subproblems (cases).

3.3 Branch Creation

The *branch creation* operation creates a new automatically defined function within an overall program by picking a point in the body of one of the function-defining branches or result-producing branches of the selected program. This picked point becomes the top-most point of the body of the branch-to-be-created. The operation of branch creation is similar to, but different than, the compression (module acquisition) operation of Angeline and Pollack (1994).

3.4 Argument Creation

The *argument creation* operation creates a new dummy argument within a function-defining branch of a program. Details of all the new operations are in Koza 1994c.

3.5 Branch Deletion

The operations of argument duplication, branch duplication, branch creation, and argument creation create larger programs. The operations of argument deletion and branch deletion can create smaller programs and thereby balance the persistent growth in biomass that would otherwise occur.

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

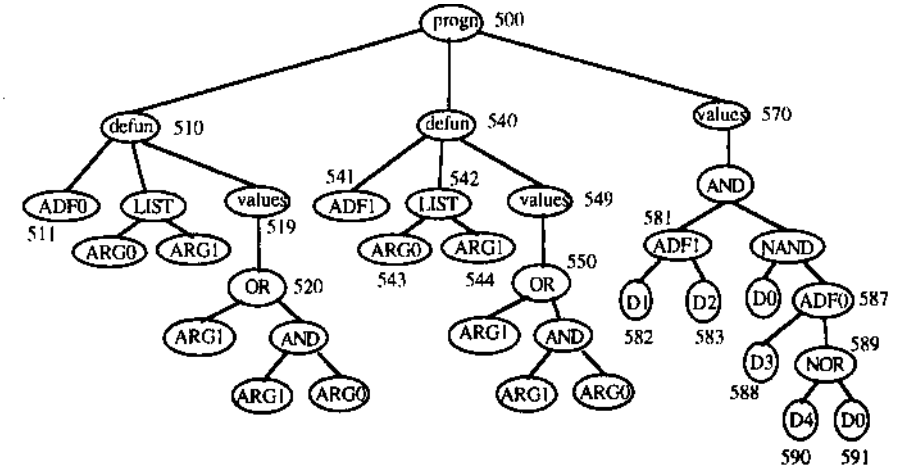

Figure 2 Program consisting of two two-argument function-defining branches and one result-producing branch.

When a function-defining branch is deleted, the question arises as to how to modify invocations of the branch-to-be-deleted by the other branches of the overall program. The alternative used herein (called branch deletion with random regeneration) randomly generates new subtrees composed of the available functions and terminals in lieu of the invocation of the deleted branch.

3.6 Argument Deletion

The operation of argument deletion deletes one of the arguments to one of the automatically defined functions of a program. When an argument is deleted, references to the argumenl-to-be-deleted may be corrected by argument deletion with random regeneration. The operations of argument deletion and branch deletion affect the value returned by the overall program. They may be viewed as a generalization in that some information that was once considered in executing the procedure is now ignored.

3.7 Creation of the Initial 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 branch creation to create multi-part programs in such runs. A third possibility is that the population at generation 0 is architecturally diverse (as described in Koza 1994a).

3.8 Structure-Preserving Crossover

When the architecture-altering operations arc used, the population quickly becomes architecturally diverse. Structure-preserving crossover with point typing (Koza 1994a) permits robust recombination while simultaneously guaranteeing that any pair of architecturally different parents will produce syntactically valid offspring.

4 Example of an Actual Run

The architecture-altering operations described herein will now be illustrated by showing an actual run of the problem of symbolic regression of the cven-5-parity function. Boolean parity functions are often used as benchmarks for experiments in machine learning because a change in any one input (environmental sensor) toggles the outcome. The problem is to discover a computer program that mimics the behavior of the Boolean even-k-parity problem for every one of the 2_k combinations of its k Boolean inputs. The primitive functions are AND, OR, NAND, and NOR.

A population size, M, of 96,000 was used. All runs solved well before the targeted maximum number of generations, G, of 76. The run uses the minimalist approach in which each program in generation 0 consists of one result-producing branch and one 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 deletions; 0.5% argument deletions; 5% branch creations; and 0% argument creations. Other minor parameters were chosen as in Koza 1994a.

The problem was run on a home-built medium-grained parallel computer system. In the so-called distributed genetic algorithm or island model for parallelization (Tanese 1989), different semi-isolated subpopulations (called demes after 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. There were D - 64 demes, a population size of Q = 1,500 per deme, and a migration rate of B = 8% (in each of four directions on each generation for each deme). Generations are run asynchronously. Details of the parallel implementation of genetic programming on a network of transputers can be found in Koza and Andre 1995.

On generation 13 of one run, a 100%-correct solution to the even-5-parity problem emerged in the form of a computer program with one three-argument automatically defined function and one two-argument automatically defined function. Three-argument ADF0 (which originally had only two arguments in 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. The result-producing branch of this program invokes both ADF0 and ADFI.

5 Performance of the New Operations

We now use the Boolean 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 runs in chapters 21-25 of Genetic Programming II), an architecturally diverse initial population, and structure-preserving crossover with point typing,

(C) with automatically defined functions, the architecture-altering operations described herein, an architecturally diverse population (after generation 0), and structure-preserving crossover with point typing,

(D) with automatically defined functions, a fixed user-supplied architecture that is known to be a good choice for this problem (i.e., one three-argument and one two-argument automatically defined function), and structure-preserving crossover with point typing, and

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

The comparisons are made for the following three performance characteristics: computational effort, E (with 99% probability); the wallclock time, W(M,t,z) in seconds(with 99% probability); and the average structural

complexity, S. These three measures are described in detail in Koza 1994a.

As Table 1 shows, all four approaches employing automatically defined functions (B, C, D, or E) require less computational effort than not using them (approach A). Approach E (which benefits from user-supplied architectural information) requires the least computational effort.

Approach C (using the architecture-altering operations) requires less computational effort than solving the problem without automatically defined functions (approach A), but more than with the fixed, user-supplied, known-good architecture (approach E).

Approach D isolates the additional computational effort required by point typing (relative to approach E). Greater computational effort is required by approach D than approach E. Since the computational effort for approach C is virtually tied with approach D, the cost of architecture-altering operations for this problem is not much greater than the cost of point typing.

Approach E consumes less wallclock time than approach C (using the architecture-altering operations), which, in turn, consumes less wallclock time than approach A (without automatically defined functions).

The average structural complexity, S, for all four approaches (B, C, D, or E) employing automatically defined functions is less than that for approach A (without automatically defined functions). Approach C (using the architecture-altering operations) has the lowest value of S (i.e., produces the most parsimonious solutions).

Acknowledgements

David Andre and Walter Alden Tackett wrote the computer program in C to implement the above.

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.

Cavicchio, Daniel J. 1970. Adaptive Search using Simulated Evolution. Ph.D. dissertation. Department of Computer and Communications Science, University of Michigan.

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

Gruau, Frederic. 1994. Genetic micro programming of neural networks. In Kinnear, Kenneth E. Jr. (editor). Advances in Genetic Programming. Cambridge, MA: The MIT Press. Pages 495-518.

Holland, John H. 1975. Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Ann Arbor, MI: University of Michigan Press. The second edition is currently available from The MIT Press 1992.

Kinnear, Kenneth E. Jr. (editor). 1994. Advances in Genetic Programming. Cambridge, MA: The MIT Press.

Koza, John R. 1989. Hierarchical genetic algorithms operating on populations of computer programs. In Proceedings of the 11th International Joint Conference on Artificial Intelligence. San Mateo, CA: Morgan Kaufmann. Volume 1. 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: The MIT Press.

Koza, John R. 1994c. Architecture-altering operations for evolving the architecture of a multi-part program in genetic programming. Stanford University Computer Science Department technical report STAN-CS-TR-94-1528. October 21, 1994.

Koza, John R. and 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.

Tanese, Reiko. 1989. Distributed Genetic Algorithm for Function Optimization. PhD. dissertation. Department of Electrical Engineering and Computer Science. University of Michigan.

Wright, Sewall. 1943. Isolation by distance. Genetics 28. Page 114-138.

Table 1 Comparison of the five approaches.

Approach	Runs	Computational effort E	Wallclock time $W(M,t,z)$	Average Size of solution $\widetilde{S}$
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, Known-Good Architecture	25	1,705,500	14,088	130.0
E - ADFs + Branch Typing + Fixed, Known-Good Architecture	25	1,261,500	6,481	112.2