Chapter 14

Automated Synthesis by Means of Genetic Programming of Complex Structures Incorporating Reuse, Parameterized Reuse, Hierarchies, and Development

John R. Koza,[1] Matthew J. Streeter,[2] and Martin A. Keane[3]
[1]*Stanford University, Stanford, California*
[2]*Genetic Programming Inc., Mountain View, California*
[3]*Econometrics Inc., Chicago, Illinois*

Abstract: Genetic programming can be used as an automated invention machine to synthesize designs for complex structures. In particular, genetic programming has automatically synthesized complex structures that infringe, improve upon, or duplicate the functionality of 21 previously patented inventions (including analog electrical circuits, controllers, and mathematical algorithms). Genetic programming has also generated two patentable new inventions (involving controllers). Genetic programming has also generated numerous additional human-competitive results involving the design of quantum computing circuits as well as other substantial results involving antennae, networks of chemical reactions (metabolic pathways), and genetic networks. We believe that these results are the direct consequence of a group of techniques—many unique to genetic programming—that facilitate the automatic synthesis of complex structures. These techniques include automatic reuse, parameterized reuse, parameterized topologies, and developmental genetic programming. The paper describes these techniques and how they contribute to automated design.

Key words: Hierarchy, reuse, development, parameterized topologies, architecture-altering operations, automatically defined functions, automatically defined iterations, automatically defined loops, automatically defined recursions, automatically defined stores, circuits, controllers

1. INTRODUCTION

Genetic programming is an automatic method for solving problems. It is an extension of the genetic algorithm (Holland 1975). Genetic programming starts from a high-level statement of the requirements of a problem and attempts to produce a computer program that solves the problem. Specifically, genetic programming starts with a primordial ooze of thousands of randomly created computer programs and uses the Darwinian principle of natural selection; analogs of recombination (crossover), mutation, gene duplication, gene deletion; and certain mechanisms of developmental biology to progressively breed an improved population over a series of generations (Koza 1992, Koza 1994; Koza, Bennett, Andre, and Keane 1999; Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003; Banzhaf, Nordin, Keller, and Francone 1998; Langdon 1998; Langdon and Poli 2002).

Genetic programming can be used as an automated invention machine to synthesize designs for complex structures. In particular, genetic programming has automatically synthesized complex structures that infringe, improve upon, or duplicate the functionality in a novel way of 21 previously patented inventions including analog electrical circuits, controllers, and mathematical algorithms (table 14.1). In addition, genetic programming generated two patentable new inventions (controllers) for which patent applications are pending (Keane, Koza, and Streeter 2002). Genetic programming has also generated numerous additional human-competitive results, involving the design of antennae, networks of chemical reactions (metabolic pathways), and genetic networks (Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003) and the design of quantum computing circuits that are superior to those designed by human designers (Spector, Barnum, and Bernstein 1998, 1999; Spector, Barnum, Bernstein, and Swamy 1999; Barnum, Bernstein, and Spector 2000; Spector and Bernstein 2003).

We believe that these results are the consequence of a group of techniques that facilitate the automatic synthesis of complex structures by genetic programming. These techniques include developmental genetic programming, reuse, parameterized reuse, and parameterized topologies.

Section 2 describes developmental genetic programming. When genetic programming is used to synthesize the design of complex structures, it is often advantageous to use a developmental process that preserves syntactic validity and locality.

We believe that reuse is an essential precondition for scalability in automated design. Section 3 describes several aspects of reuse, including:

- reuse of useful substructures by means of the ordinary crossover (recombination) operation (section 3.1),
- reuse of useful substructures by means of subroutines (automatically defined functions) (section 3.2),
- parameterized reuse (section 3.3),
- creation of reuse by the architecture-altering operations (section 3.4),
- reuse of code by means of automatically defined iterations, automatically defined loops, and automatically defined recursions (section 3.5), and
- reuse of the results of executing code by means of automatically defined stores (memory) (section 3.6).

Section 4 describes the automatic creation of parameterized topologies. Section 5 describes the automatic creation of parameterized topologies containing conditional operators. Section 6 contains a discussion of future possibilities for genetic programming theory and practice.

2. DEVELOPMENTAL GENETIC PROGRAMMING

When genetic programming is used to automatically create computer programs, the programs are ordinarily represented as program trees (i.e., rooted, point-labeled trees with ordered branches).

When genetic programming is used to synthesize the design of a cyclic graphical structure (such as an electrical circuit), a developmental process may be advantageously used to establish a mapping between the program trees (acyclic graphs) and the cyclic graphs. The developmental process may begin with a simple embryo consisting of a single modifiable wire that is not initially connected to the inputs or outputs of the to-be-synthesized circuit. The final circuit is developed by progressively applying the functions in a circuit-constructing program tree (evolved by genetic programming) to the embryo's initial modifiable wire (and to succeeding modifiable wires and modifiable components).

Table 14.1 Twenty-one previously patented inventions reinvented by genetic programming

Invention	Date	Inventor	Place	Patent
Darlington emitter-follower section	1953	Sidney Darlington	Bell Telephone Laboratories	2,663,806
Ladder filter	1917	George Campbell	American Telephone and Telegraph	1,227,113
Crossover filter	1925	Otto Julius Zobel	American Telephone and Telegraph	1,538,964
"*M*-derived half section" filter	1925	Otto Julius Zobel	American Telephone and Telegraph	1,538,964
Cauer (elliptic) topology for filters	1934–1936	Wilhelm Cauer	University of Gottingen	1,958,742, 1,989,545
Sorting network	1962	Daniel G. O'Connor and Raymond J. Nelson	General Precision, Inc.	3,029,413
Computational circuits	NA	Numerous	Numerous	Numerous
Electronic thermometer	NA	Numerous	Numerous	Numerous
Voltage reference circuit	NA	Numerous	Numerous	Numerous
60 dB and 96 dB amplifiers	NA	Numerous	Numerous	Numerous
Second-derivative controller	1942	Harry Jones	Brown Instrument Company	2,282,726
Philbrick circuit	1956	George Philbrick	George A. Philbrick Researches	2,730,679
`NAND` circuit	1971	David H. Chung and Bill H. Terrell	Texas Instruments Incorporated	3,560,760
PID (proportional, integrative, and derivative) controller	1939	Albert Callender and Allan Stevenson	Imperial Chemical Limited	2,175,985
Negative feedback	1937	Harold S. Black	American Telephone and Telegraph	2,102,670, 2,102,671
Low-voltage balun circuit	2001	Sang Gug Lee	Information and Communications University	6,265,908
Mixed analog-digital variable capacitor circuit	2000	Turgut Sefket Aytur	Lucent Technologies Inc.	6,013,958
High-current load circuit	2001	Timothy Daun-Lindberg and Michael Miller	International Business Machines Corporation	6,211,726
Voltage-current conversion circuit	2000	Akira Ikeuchi and Naoshi Tokuda	Mitsumi Electric Co., Ltd.	6,166,529
Cubic function generator	2000	Stefano Cipriani and Anthony A. Takeshian	Conexant Systems, Inc.	6,160,427
Tunable integrated active filter	2001	Robert Irvine and Bernd Kolb	Infineon Technologies AG	6,225,859

The functions in the circuit-constructing program trees may include

(1) topology-modifying functions that alter the topology of a developing circuit (e.g., series division, parallel division, via between nodes, via to ground, via to a power supply, via to the circuit's input signal, via to the circuit's output points),

(2) component-creating functions that insert components (i.e., resistors, capacitors, and transistors) into a developing circuit, and

(3) development-controlling functions that control the developmental process (e.g., cut, end).

To illustrate the developmental process, figure 14.1 shows a circuit-constructing program tree that produces the circuit shown in figure 14.2.

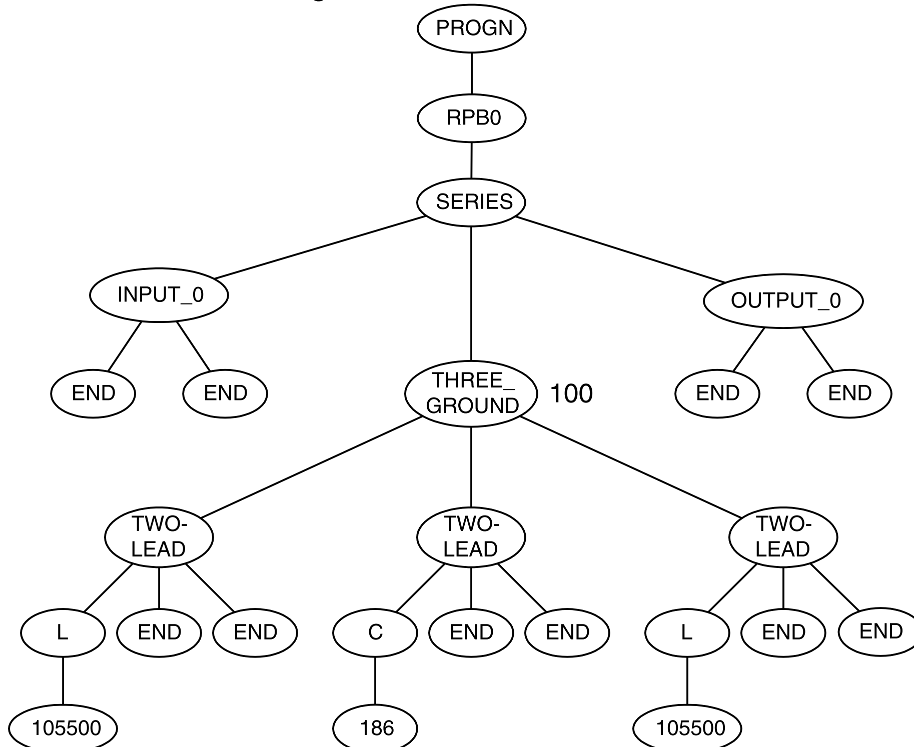

Figure 14.1 Circuit-constructing program tree that develops into one T-section

The fully developed circuit of figure 14.2 consists of one T-section containing two 105,500-micro-Henry inductors (one on each arm of the "T") and a 186-nanofarad capacitor on the vertical segment of the "T." In figure 14.2, the incoming signal VSOURCE and the source resistor RSOURCE at the far left as well as the load resistor RLOAD at the far right are part of a fixed test fixture that is not subject to evolutionary change.

In the circuit-constructing program tree of figure 14.1, the second argument of the `THREE_GROUND` topology-modifying function (described in detail in Koza, Bennett, Andre, and Keane 1999) is a `TWO_LEAD` component-creating function that creates a capacitor C1 connected between node 0 in figure 14.2 and node 6 in figure 14.2 (an intermediate node created by the operation of the `THREE_GROUND` function).

The first argument of the three-argument `SERIES` topology-modifying function in figure 14.1 is a two-argument `INPUT_0` topology-modifying function that makes a connection between the incoming signal and L1. The third argument of the `SERIES` function in figure 14.1 is a two-argument `OUPTUT_0` topology-modifying function that makes a connection between the circuit's output point and L2. The development-controlling `END` functions terminate the developmental process.

The first argument of the three-argument `THREE_GROUND` topology-modifying function in figure 14.1 is a `TWO_LEAD` function that creates the 105,500-micro-Henry inductor L1 connected between nodes 2 (one end of the embryo's now-replaced original modifiable wire) and 6 of figure 14.2. The values of components, such as capacitors and inductors, are established by value-setting subtrees that typically contain a multiplicity of arithmetic-performing functions and numerical values. However, for simplicity, figure 14.1 shows only the component's final numerical value. The second argument of the `THREE_GROUND` function in figure 14.1 is a `TWO_LEAD` function that creates the 186-nanofarad capacitor C1 connected between nodes 6 and 0 of figure 14.2. The third argument of the `THREE_GROUND` function in figure 14.1 is a `TWO_LEAD` function that creates the 105,500-micro-Henry inductor L2 connected between nodes 3 (the other end of the embryo's now-replaced modifiable wire) and 6 of figure 14.2.

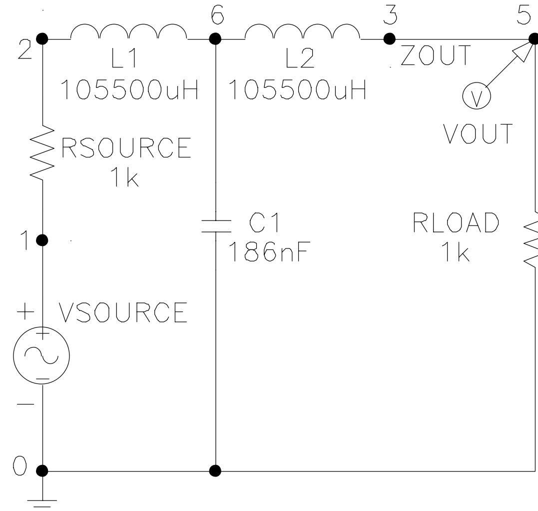

Figure 14.2 Circuit consisting of one T-section

3. REUSE

Anyone who has ever looked at a blueprint for a building, an electrical circuit, a protein molecule, a corporate organizational chart, a musical score, a city map, or a computer program will be struck by the massive reuse of certain basic substructures within the overall structure. Indeed, complex structures are almost always replete with modularities, symmetries, and regularities. Reuse can accelerate automated learning by avoiding "reinventing the wheel" on each occasion requiring a particular sequence of already-learned steps. We believe that reuse is the cornerstone of meaningful machine intelligence.

3.1 Reuse by Means of Crossover

The crossover (recombination) operation is one mechanism by which genetic programming can advantageously reuse useful substructures. For example, one way to construct a lowpass filter with a passband boundary of 1,000 Hz is by means of a cascade of identical T-sections. See Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003 for additional details of this illustrative problem of filter design.

A circuit (such as figure 14.2) consisting of one T-section is a lowpass filter, albeit an extremely poor one in that it does not sharply suppress frequencies higher than 1,000 Hz. The crossover operation may create a circuit-constructing program tree (such as the one shown in figure 14.4) by reusing genetic material from figure 14.1. The subtree rooted at the point labeled

200 in figure 14.4 corresponds to the subtree rooted at the `THREE_GROUND` function labeled 100 in figure 14.1. That is, the subtree that was responsible for the filtering ability of the single T-section of figure 14.2 is now embedded inside another individual that itself produces a T-section. The result of the execution of the circuit-constructing program tree in figure 14.4 is two identical T-sections (figure 14.3). A cascade consisting of these two identical T-sections (figure 14.3) is a slightly better lowpass filter than a single T-section. Note that the tree depicted in figure 14.4 could be the result of either a crossover of two individuals that each develop into a T-section or the result of a crossover of an individual with itself.

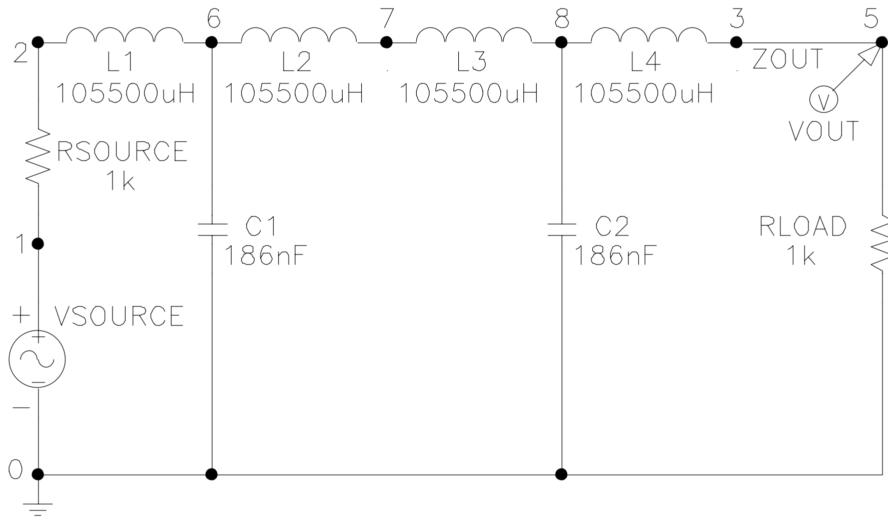

Figure 14.3 Circuit consisting of a cascade of two identical T-sections

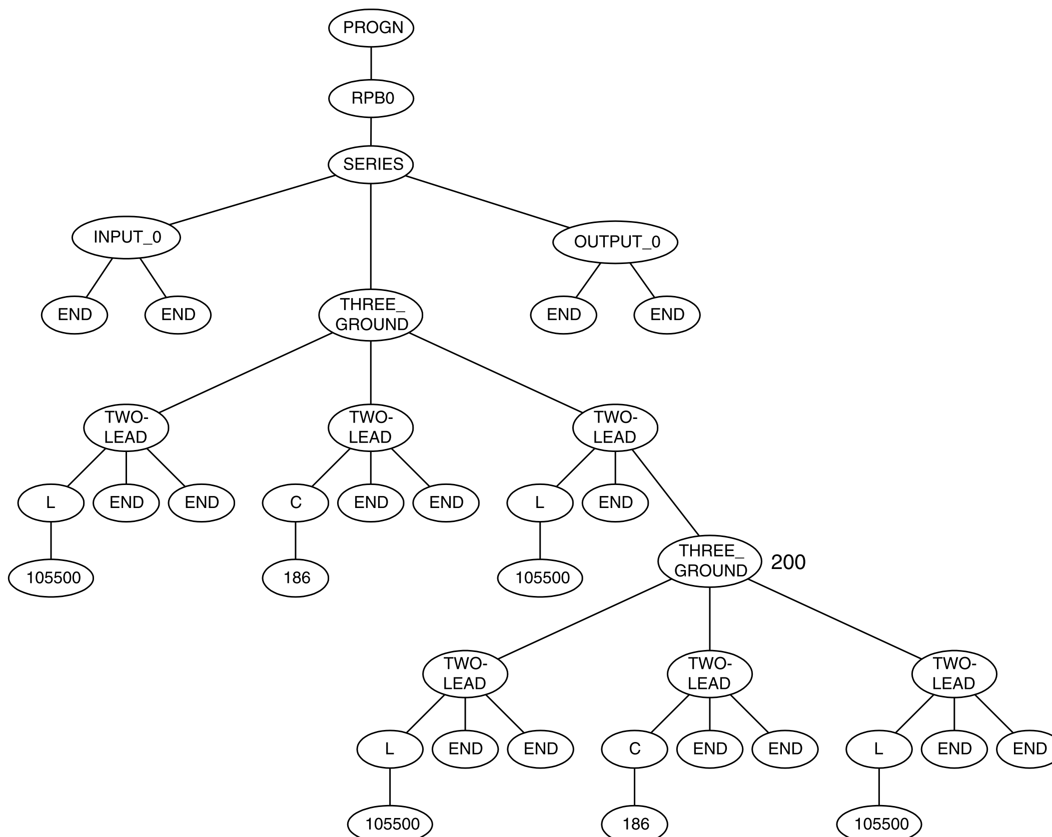

Figure 14.4 Circuit-constructing program tree that develops into two T-sections

3.2 Reuse by Means of Automatically Defined Functions

Subroutines (automatically defined functions) provide another mechanism by which genetic programming can automatically reuse code—either exactly or with different instantiations of dummy variables or formal parameters. Genetic programming is capable of automatically creating reusable subroutines along with main programs (result-producing branches) dynamically during a run. Figure 14.5 shows a circuit-constructing program tree that contains an automatically defined function (ADF0) that develops into the two T-sections shown in figure 14.3. ADF0 develops into one T-section. The result-producing branch (RPB0) invokes ADF0 twice, thereby creating a circuit consisting of two identical T-sections.

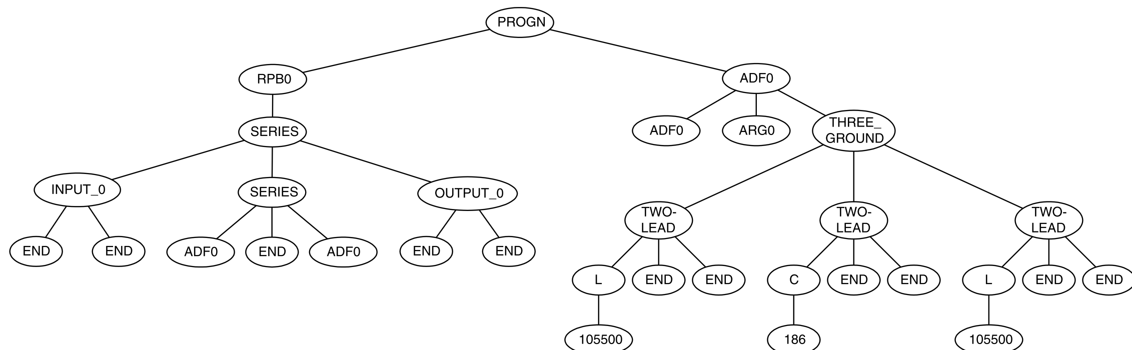

Figure 14.5 Circuit-constructing program tree containing automatically defined function ADF0 that develops into two T-sections

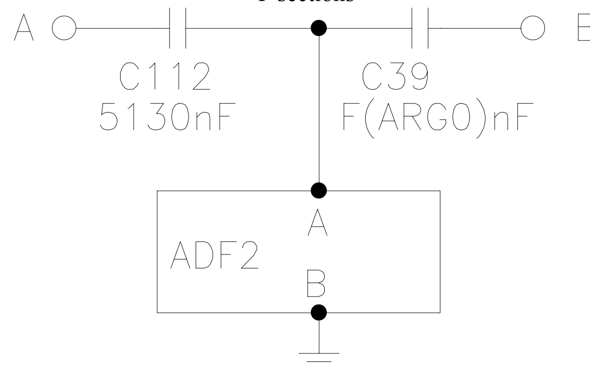

Figure 14.6 Subcircuit produced by three-ported automatically defined function ADF3

3.3 Parameterized Reuse

In genetic programming, different instantiations of a dummy variable (formal parameter) of an automatically defined function may be passed to an automatically defined function. This aspect of genetic programming can be illustrated by a portion of a genetically evolved circuit for a two-band crossover (woofer-tweeter) filter (described in detail in Koza, Bennett, Andre, and Keane 1999). Figure 14.6 shows the subcircuit that is created by the execution of three-ported automatically defined function. The execution of automatically defined function (ADF3) has the following three consequences.

- It inserts a (not noteworthy) fixed 5,130 nanofarad capacitor in the upper left of the figure.
- It inserts a (noteworthy) parameterized capacitor C39 whose component value is dependent on the dummy variable ARG0.

- It invokes an automatically defined function ADF2. The dummy variable ARG0 is passed to ADF2. In turn, ADF2 creates a (noteworthy) parameterized inductor whose component value is dependent on the dummy variable ARG0 that is passed to ADF2.

Parameters may be similarly passed into automatically defined iterations, automatically defined loops, and automatically defined recursions that possess dummy variables (as described in later sections of this paper).

3.4 Creation of Reuse by Means of the Architecture-Altering Operations

The genome of *Mycoplasma genitilium* contains only 470 different genes (Fraser et al. 1995), whereas the human genome contains approximately two orders of magnitude more. The question arises as to how new genes—that is, new biological functions—originate in nature? Occasionally, a gene may be duplicated, thereby creating two places on the chromosome that manufacture the same protein. After such a gene duplication, one of the two initially identical genes may remain intact and continue to manufacture the original protein (thus conferring the gene's presumably survival-related function on the organism). Meanwhile, over many generations, the second gene may harmlessly accumulate changes and diverge. Eventually the second gene may come to manufacture a new protein with an entirely new function. Thus, new biological functions emerge in nature as part of the evolutionary process (Ohno 1970).

Genetic programming uses architecture-altering operations (described in detail in Koza, Bennett, Andre, and Keane 1999) to automatically determine program architecture in a manner that parallels gene duplication, and the related operation of gene deletion, in nature. The architecture-altering operations are performed (usually with a low probability in the neighborhood of 1%) during each generation of a genetic programming run (along with the genetic operations of crossover, mutation, and reproduction). When the architecture-altering operations are used, the overall architecture and hierarchy of the evolved computer program are part of the output produced by genetic programming—not part of the input supplied by the human user.

The subroutine duplication operation duplicates a preexisting subroutine in an individual program, gives a unique new name to the copy, and randomly divides the preexisting calls to the old subroutine between the two. This operation changes the program architecture by broadening the hierarchy of subroutines in the overall program. As with gene duplication in nature, this operation preserves semantics when it first occurs. The two subroutines typically diverge later—sometimes yielding specialization.

The argument duplication operation duplicates one argument of a subroutine, randomly divides internal references to it, and preserves overall program semantics by adjusting all calls to the subroutine. This operation enlarges the dimensionality of the subspace on which the subroutine operates.

The subroutine creation operation can create a new subroutine from part of a main result-producing branch (main program), thereby deepening the hierarchy of references in the overall program, by creating a hierarchical reference between the main program and the new subroutine. ADF0 in figure 14.5, for example, could possibly have been created in this way. The subroutine creation operation can also create a new subroutine from part of an existing subroutine by creating a hierarchical reference between a preexisting subroutine and a new subroutine, thus creating a deeper and more complex overall hierarchy.

The subroutine deletion operation deletes a subroutine from a program thereby making the hierarchy of subroutines narrower or shallower.

The argument deletion operation deletes an argument from a subroutine, thereby (usually) reducing the amount of information available to the subroutine—a process that can be viewed as generalization.

The architecture-altering operations rapidly create an architecturally diverse population containing programs with different numbers of subroutines, different numbers of arguments, and, different hierarchical arrangements of the subroutines. Programs with architectures that are well-suited to the problem at hand will tend to grow and prosper in the competitive evolutionary process, while programs with inappropriate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure.

3.5 Reuse of Code with Iterations, Loops, and Recursions

Other architecture-altering operations add and delete iterations, loops, and recursions. Automatically defined iterations, automatically defined loops, and automatically defined recursions provide additional mechanisms that enable genetic programming to automatically reuse code (Koza, Bennett, Andre, and Keane 1999).

The circuit-constructing functions responsible for a useful subcircuit may reside in an automatically defined copy (ADC). An automatically defined copy is an iteration specialized to problems of circuit synthesis. Multiple occurrences of the subcircuit result when the copy body branch (CBB) is repeatedly invoked by the copy control branch (CCB) of the automatically defined copy (Koza, Bennett, Andre, and Keane 1999).

3.6 Reuse of the Results of Executing Code with Automatically Defined Stores

Automatically defined functions, automatically defined iterations, automatically defined loops, and automatically defined recursions provide a mechanism for reusing code at different levels of granularity in a complex system.

In contrast, automatically defined stores (memory) provide a way for automatically reusing the remembered results produced by the execution of code (Koza, Bennett, Andre, and Keane 1999).

4. AUTOMATIC SYNTHESIS OF PARAMETERIZED TOPOLOGIES

Genetic programming can automatically create, in a single run, a general (parameterized) solution to a problem in the form of a graphical structure whose nodes or edges represent components and where the values of the components are specified by mathematical expressions containing free variables. The free variables represent parameters of the design problem (e.g., the passband boundary frequency for a lowpass filter). By producing a graphical structure whose component values are specified by mathematical equations containing the free variables, genetic programming produces a general solution to the problem (e.g., a design for a collection of lowpass filters, each of which has a different passband boundary) as opposed to a design for a specific single problem (i.e., the design of a lowpass filter having a specific single passband boundary).

The genetically evolved individual represents a complex structure (e.g., electrical circuit, controller, network of chemical reactions, antenna, genetic network). In the automated design process, genetic programming determines the gross size of the graphical structure (i.e., the number of nodes in the graph) as well as the graph’s connectivity (i.e., a specification of the nodes that are connected to each other). Genetic programming also assigns component types to various edges (or nodes) of the graphical structure. In a circuit, the components may be transistors, resistors, and capacitors. In a controller, they may be integrators, differentiators, gain blocks, leads, lags, delays, adders, and subtractors. Genetic programming also creates

mathematical expressions that establish the parameter values for the components (e.g., the capacitance of a capacitor in a circuit, the amplification factor of a gain block in a controller). The free variables in the mathematical expressions confer generality on the genetically evolved solution. The free variables enable a single genetically evolved graphical structure to represent a parameterized solution to an entire category of problems. In creating parameterized topologies, genetic programming can do all of the above in an automated way in a single run.

The capability of genetic programming to create parameterized topologies for design problems is illustrated by the automatic creation of a general-purpose controller (figure 14.7). The purpose of a controller is to force a system—conventionally called the *plant*—to produce an output signal that matches a specified *reference signal*. This genetically evolved controller outperforms PID controllers tuned using the widely used 1942 Ziegler-Nichols tuning rules (Ziegler and Nichols 1942) and the recently developed 1995 Astrom and Hagglund tuning rules on an industrially representative set of plants in terms of the criteria described in Astrom and Hagglund 1995. Some of the blocks of this genetically evolved controller are parameterized by constants whereas others are parameterized by mathematical expressions containing the problem's four free variables. These free variables are readily measurable behavioral characteristics of plants. In particular, they are the plant's time constant, T_r, ultimate period, T_u, ultimate gain, K_u, and dead time, L. This controller's overall topology consists of three adders, three subtractors, four gain blocks parameterized by a constant, two gain blocks parameterized by non-constant mathematical expressions containing free variables, and two lead blocks parameterized by non-constant mathematical expressions containing free variables. For purposes of illustration, we mention that gain block 730 of the figure has a gain of

$$\left|\log\left|T_r - T_u + \log\left|\frac{\log\left(|L|^L\right)}{T_u + 1}\right|\right|\right|$$

and gain block 760 of the figure has a gain of

$$\left|\log\left|T_r + 1\right|\right| .$$

The genetically evolved mathematical expressions for the other blocks in this controller and other details of this problem can be found in Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003.

The techniques above have been applied to a variety of additional problems (Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003), including synthesis of

- a parameterized circuit-constructing program tree containing free variables that yields a Zobel network,
- a parameterized circuit-constructing program tree that yields a passive third-order elliptic lowpass filter whose modular angle is specified by a free variable,
- a parameterized circuit-constructing program tree that yields a passive lowpass filter whose passband boundary is specified by a free variable,
- a parameterized circuit-constructing program tree that yields an active lowpass filter whose passband boundary is specified by a free variable, and
- a parameterized controller for controlling a three-lag plant whose time constant is specified by a free variable.

Figure 14.7 Genetically evolved general-purpose controller.

5. AUTOMATIC SYNTHESIS OF PARAMETERIZED TOPOLOGIES CONTAINING CONDITIONAL OPERATORS

If the genetically evolved program for a parameterized topology contains conditional developmental operators as well as free variables, a different graphical structure may be produced for different instantiations of the free variables. That is, the genetically evolved program for the parameterized topology may operate as a genetic switch. Each genetically evolved program has inputs, namely the problem's free variables. Depending on the values of the free variables, different graphical structures may result from the execution of the program.

For example, a single genetically evolved circuit-constructing program tree with free variables (representing two frequencies, `F1` and `F2`) and conditional operators may yield either

- a lowpass filter with a variable passband boundary determined by the free variables (figure 14.8), or
- a highpass filter with a variable passband boundary determined by the free variables (figure 14.9).

Automatically created mathematical expressions parameterize the capacitors and inductors in these automatically created circuits.

The technique above has also been applied to additional problems (Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003), including the synthesis of

- a circuit-constructing program tree containing conditional developmental operators and free variables that yields either a quadratic or cubic function, and
- a circuit-constructing program tree containing conditional developmental operators and free variables that yields either a 40 dB or 60 dB amplifier.

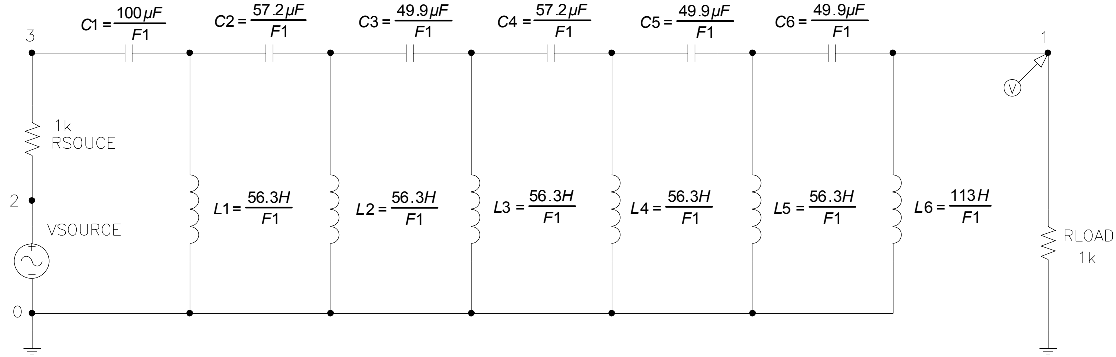

Figure 14.8 Genetically evolved generalized circuit when free variables call for a highpass filter (that is, F1 > F2)

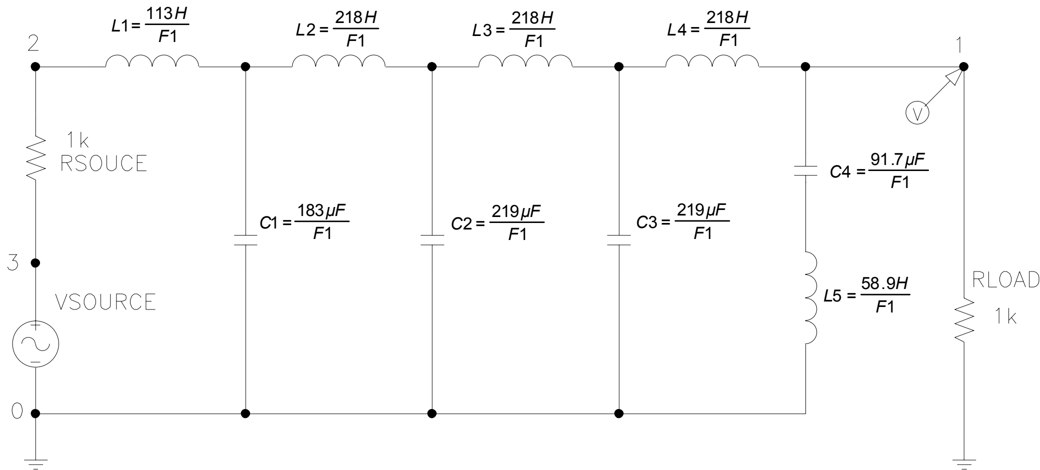

Figure 14.9 Genetically evolved generalized circuit when free variables call for a lowpass filter.

6. DISCUSSION

In this paper we have discussed how genetic programming can be used to synthesize a variety of complex structures. Our approach to the synthesis of complex structures has itself evolved over the years, with changes to methodology, representations, and fitness functions made primarily based on practical experience, intuition, and judgments based on the theoretical foundations of genetic algorithms. Examples include our decision to use representations that we would expect to preserve building blocks in the face of crossover. In some cases—notably our design of the architecture-altering operations and our use of a developmental process for growing electrical circuits—the technique enhancements found their inspiration in biology. However, there is a clear need for more rigorous guidance and theory. Given that a representation is "crossover-friendly" (i.e., there exist within the representation local structures that contribute to fitness), the question naturally arises as to how can we further characterize the effectiveness of a representation. To what extent is the scalability of optimization algorithms affected by their ability to exploit opportunities for reuse? When is the use of higher-level constructs, such as iterations, loops, and recursions, essential for efficient solution of a problem? We believe the answers to these and similar questions will be essential to the development of truly competent genetic programming (in the sense advocated by David Goldberg in his 2002 book *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*).

BIBLIOGRAPHY

Astrom, Karl J. and Hagglund, Tore. 1995. *PID Controllers: Theory, Design, and Tuning.* Second Edition. Research Triangle Park, NC: Instrument Society of America.

Banzhaf, Wolfgang, Nordin, Peter, Keller, Robert E., and Francone, Frank D. 1998. *Genetic Programming – An Introduction*. San Francisco, CA: Morgan Kaufmann and Heidelberg: dpunkt.

Barnum, H., Bernstein, H.J. and Spector, Lee. 2000. Quantum circuits for OR and AND of ORs. *Journal of Physics A: Mathematical and General*. 33(45):8047–8057. November 17, 2000.

Fraser, C. M., Gocayne, J. D., White, O., Adams, M. D., Clayton, R. A., Fleischmann, R. D., Bult, C. J., Kerlavage, A. R. , Sutton, G., Kelley, J. M., et al. 1995. The minimal gene complement of Mycoplasma genitalium. *Science.* 270(5235):397-403. Oct 20, 1995.

Goldberg, David E. 2002. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Boston: Kluwer Academic Publishers.

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. Second edition. Cambridge, MA: The MIT Press 1992.

Keane, Martin A., Koza, John R., and Streeter, Matthew J. 2002. *Improved General-Purpose Controllers*. U.S. patent application filed July 12, 2002.

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

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

Koza, John R., Bennett III, Forrest H, Andre, David, and Keane, Martin A. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. San Francisco, CA: Morgan Kaufmann.

Koza, John R., Keane, Martin A., Streeter, Matthew J., Mydlowec, William, Yu, Jessen, and Lanza, Guido. 2003. *Genetic Programming IV. Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

Langdon, William B. 1998. *Genetic Programming and Data Structures: Genetic Programming + Data Structures = Automatic Programming!* Amsterdam: Kluwer.

Langdon, William B. and Poli, Riccardo. 2002. *Foundations of Genetic Programming*. Springer-Verlag.

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

Spector, Lee, Barnum, Howard, and Bernstein, Herbert J. 1998. Genetic programming for quantum computers. In Koza, John R., Banzhaf, Wolfgang, Chellapilla, Kumar, Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max H., Goldberg, David E., Iba, Hitoshi, and Riolo, Rick. (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference*. San Francisco, CA: Morgan Kaufmann. Pages 365–373.

Spector, Lee, Barnum, Howard, and Bernstein, Herbert J. 1999. Quantum computing applications of genetic programming. In Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). *Advances in Genetic Programming 3*. Cambridge, MA: The MIT Press. Pages 135–160.

Spector, Lee, Barnum, Howard, Bernstein, Herbert J., and Swamy, N. 1999. Finding a better-than-classical quantum AND/OR algorithm using genetic programming. In IEEE. *Proceedings of 1999 Congress on Evolutionary Computation*. Piscataway, NJ: IEEE Press. Pages 2239–2246.

Spector, Lee, and Bernstein, Herbert J. 2003. Communication capacities of some quantum gates, discovered in part through genetic programming. In Shapiro, Jeffery H. and Hirota, Osamu (editors). *Proceedings of the Sixth International Conference on Quantum Communication, Measurement, and Computing*. Princeton, NJ: Rinton Press. Pages 500-503.

Ziegler, J. G. and Nichols, N. B. 1942. Optimum settings for automatic controllers. *Transactions of ASME*. (64)759–768.