Automated Synthesis by Means of Genetic Programming of Human-Competitive Designs Employing Reuse, Hierarchies, Modularities, Development, and Parameterized Topologies

John R. Koza
Stanford University
koza@stanford.edu

Matthew J. Streeter
Genetic Programming Inc.
mjs@tmolp.com

Martin A. Keane
Econometrics Inc.
martinkeane@ameritech.net

Abstract

Genetic programming can be used as an automated invention machine to create designs. Genetic programming has automatically created designs that infringe, improve upon, or duplicate the functionality (in a novel way) of 16 previously patented inventions involving circuits, controllers, and mathematical algorithms. Genetic programming has also generated two patentable new inventions for which patent applications have been filed. Genetic programming has also generated numerous other human-competitive results, including the design of quantum computing circuits that are superior to those designed by human designers. Genetic programming has also designed antennae, networks of chemical reactions (metabolic pathways), and genetic networks. Genetic programming can automatically create hierarchies, automatically identify and reuse modularities, automatically determine program architecture, and automatically create parameterized topologies. When genetic programming is used to design complex structures, it is often advantageous to use a developmental process that enables syntactic validity and locality to be preserved under crossover.

1 Introduction

Genetic programming can be used as an automated invention machine to create designs. Genetic programming has automatically created structural entities that infringe, improve upon, or duplicate the functionality (in a novel way) of 16 previously patented inventions from the fields of circuits, controllers, and mathematical algorithms. Genetic programming has also generated numerous other human-competitive results, including the design of quantum computing circuits that are superior to those designed by human designers. Genetic programming has also designed antennae, networks of chemical reactions (metabolic pathways), and genetic networks. Genetic programming can automatically create hierarchies, automatically identify and reuse modularities, automatically determine program architecture, and automatically create parameterized topologies. When genetic programming is used to design complex structures, it is often advantageous to use a developmental process that preserves syntactic validity and locality.

Section 2 provides general background on genetic programming.

Section 3 describes eight features of genetic programming that are particularly relevant to automated design and computational synthesis, namely

- developmental genetic programming,
- reuse of useful substructures by means of the crossover (recombination) operation,
- reuse of useful substructures by means of subroutines (automatically defined functions),
- automatic creation of hierarchies, modularities, and reuse by means of architecture-altering operations,
- automatic creation of parameterized topologies,
- automatic creation of parameterized topologies containing conditional operators,
- passing parameters to substructures, and
- novelty-driven evolution.

Section 4 surveys 10 20th century patented circuits, controllers, and mathematical algorithms and six 21st-century patented circuits that have been automatically synthesized by means of genetic programming.

2 Genetic Programming

Genetic programming is an automatic method for solving problems. Specifically, genetic programming progressively breeds a population of computer programs over a series of generations. Genetic programming starts with a primordial ooze of thousands of randomly created computer programs and uses the Darwinian principle of natural selection and 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 many generations.

Genetic programming (Koza 1992; Koza and Rice 1992; Koza 1994a; Koza 1994b; Koza, Bennett, Andre, and Keane 1999; Koza, Bennett, Andre, Keane, and Brave 1999; Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003) breeds computer programs to solve problems by executing the following three steps:

(1) Generate an initial population of compositions (typically random) of functions and terminals appropriate to the problem.

(2) Iteratively perform the following substeps (a generation) on the population of programs until the termination criterion has been satisfied:

(A) Execute each program in the population and assign it a fitness value using the problem's fitness measure.

(B) Create a new population of programs by applying the following operations to program(s) selected from the population with a probability based on fitness (with reselection allowed).

(i) **Reproduction:** Copy the selected program to the new population.

(ii) **Crossover:** Create a new offspring program for the new population by recombining randomly chosen parts of two selected programs.

(iii) **Mutation:** Create one new offspring program for the new population by randomly mutating a randomly chosen part of the selected program.

(iv) **Architecture-altering operations:** Select an architecture-altering operation from the available repertoire of such operations and create one new offspring program for the new population by applying the selected architecture-altering operation to the selected program.

(3) Designate the individual program that is identified by result designation (e.g., the individual with the best fitness) as the run's result. This result may be a solution (or approximate solution) to the problem.

3 Eight Features of Genetic Programming Relevant to Design

3.1 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). However, when genetic programming is used to design complex structures, it is often advantageous to use a developmental process that preserves syntactic validity and locality under the crossover (recombination) operation.

For example, electrical circuits are usually represented as labeled cyclic graphs. A developmental process can be used to establish a mapping between program trees and labeled 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-created circuit. A circuit is developed by progressively applying the functions in a circuit-constructing program tree (created by genetic programming) to the embryo's initial modifiable wire (and to succeeding modifiable wires and modifiable components). 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 1 shows a circuit-constructing program tree that develops into the circuit shown in figure 2. In figure 2, the incoming signal VSOURCE and its 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.

The fully developed circuit of figure 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 the circuit-constructing program tree of figure 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 2 and node 6 in figure 2 (an intermediate node created by the operation of the `THREE_GROUND` function). The first argument of the three-argument `THREE_GROUND` function is a `TWO_LEAD` function that creates the 105,500-micro-Henry inductor L1 connected between nodes 2 of figure 2 (one end of the embryo's now-replaced original modifiable wire) and node 6 of figure 2. The third argument of the `THREE_GROUND` function in figure 1 is a `TWO_LEAD` function that creates the 105,500-micro-Henry inductor L2 connected between nodes 6 and 3 of figure 2 (the other end of the embryo's now-replaced modifiable wire). The first argument of the three-argument `SERIES` topology-modifying function in figure 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 1 is a two-argument `OUPTUT_0` topology-modifying function that makes a connection between the circuit's output point and L2.

3.2 Reuse by Means of Crossover

Anyone who has ever looked at a floor plan of a building, a corporate organization chart, a musical score, a protein molecule, a city map, or an electrical circuit diagram 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 avoids reinventing the wheel on each occasion requiring a particular sequence of already-learned steps.

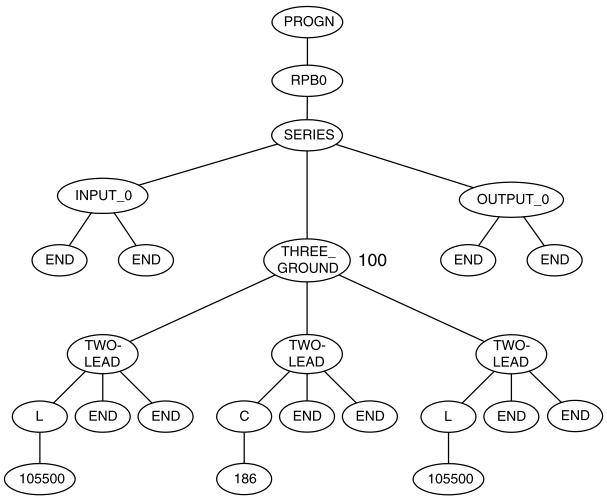

Figure 1 Circuit-constructing program tree that develops into one T-section

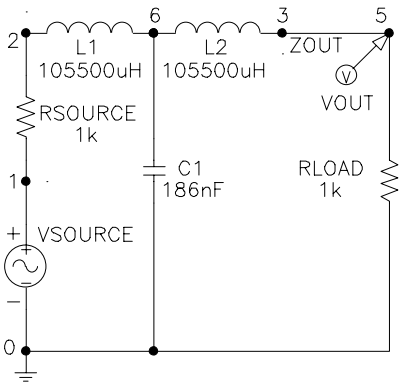

Figure 2 Circuit consisting of one T-section

We believe that reuse is an essential precondition for scalability in automated design.

Genetic programming automatically identifies substructures that are to be reused. The crossover (recombination) operation is one mechanism by which genetic programming reuses 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 detailed specifications of the contemplated filter.

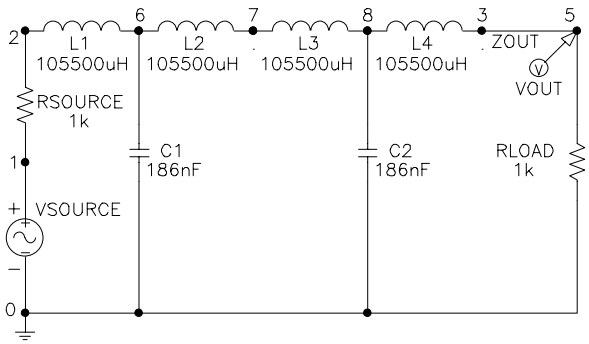

Figure 3 Circuit consisting of a cascade of two identical T-sections

A circuit (such as figure 2) consisting of one T-section is an extremely poor lowpass filter. A cascade consisting of two identical T-sections (figure 3) is a slightly better (but still poor) lowpass filter. The crossover operation may create a circuit-constructing program tree (such as the one shown in figure 4) by reusing genetic material from figure 1. The subtree rooted at the point labeled 200 in figure 4 is the subtree rooted at the `THREE_GROUND` function labeled 100 in figure 1. That is, the subtree that was responsible for the T-section and for the reasonably high fitness of the individual of figure 2 is now embedded inside another reasonably high-fitness individual that itself produces a T-section. The result is two identical T-sections.

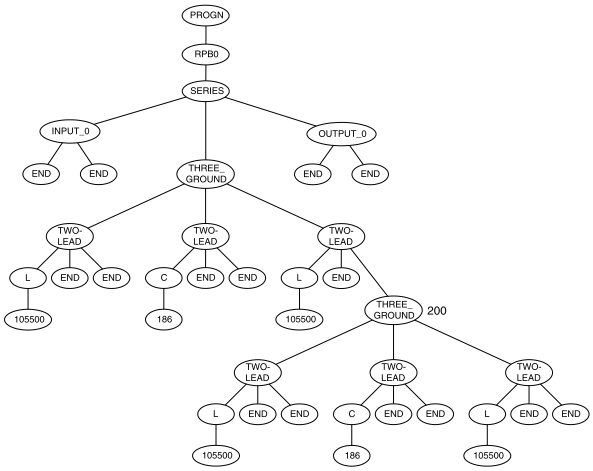

Figure 4 Circuit-constructing program tree that develops into two T-sections

3.3 Reuse by Means of a Subroutine

Subroutines (automatically defined functions) provide another (perhaps the most common) 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 automating creating reusable subroutines along with main programs (result-producing branches) dynamically during a run. Figure 5 shows a circuit-constructing program tree containing an automatically defined function (`ADF0`) that develops into the two T-sections shown in figure 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.

3.4 Automatic Creation of Hierarchies, Modularities, and Reuse by Means of Architecture-Altering Operations

The genome of the simplest currently-known living organism manufactures only 470 different proteins, whereas the human genome manufactures about 35,000 proteins. Since mutation and crossover modify only a preexisting gene, the question arises as to how do 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.

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.

Thus, the architecture, hierarchy, size, and content 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 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 5 could, for example, 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 reducing the amount of information available to the subroutine—a process that can be viewed as generalization.

The architecture-altering operations add and delete iterations, loops, recursions, and memory. Automatically defined iterations, automatically defined loops, and automatically defined recursions provide additional mechanisms that enable genetic programming to automatically reuse code. Automatically defined stores provide a way for automatically reusing the results produced by the execution of code.

The architecture-altering operations rapidly create an architecturally diverse population containing programs with different numbers of subroutines, arguments, iterations, loops, recursions, and different amounts of memory and, also, different hierarchical arrangements of these elements. 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 inadequate architectures will tend to wither away under the relentless selective pressure of the problem's fitness measure.

3.5 Automatic Creation of Parameterized Topologies

Genetic programming automatically can 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 value of the components are specified by mathematical expressions containing free variables.

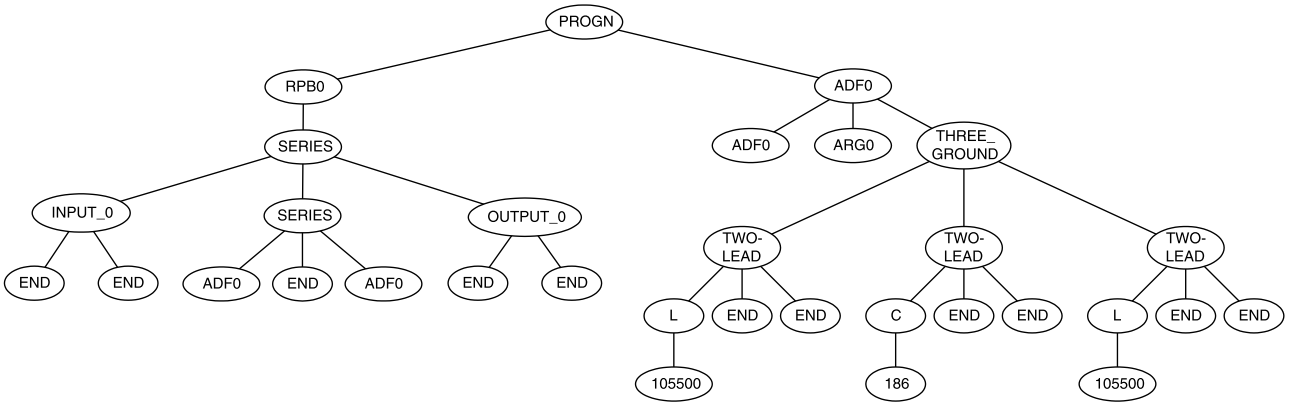

Figure 5 Circuit-constructing program tree containing automatically defined function `ADF0` that develops into two T-sections

The genetically evolved individual represents a complex structure (e.g., an electrical circuit, a controller, a network of chemical reactions, an antenna, a genetic network). In the automated 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. The components may be transistors, resistors, and capacitors in a circuit. They may be integrators, differentiators, gain blocks, adders, and subtractors in a controller. 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. Genetic programming does all of the above in an automated way in a single run.

As an example, suppose the goal is to design a circuit to feed the woofer speaker of a hi-fi system. That is, the desired circuit is intended to pass signals up to a certain frequency at full power into the woofer but to suppress all higher-frequency signals. Furthermore, the goal is to create a general solution to this design problem—that is, a solution that works for any value of f (not just a solution that works for, say, just 1,250 Hertz). The general solution produced by genetic programming includes the circuit's topology. The general solution has nine components. Four of the nine components are inductors and five are capacitors. The nine components are connected to each other in a particular way. The genetically evolved solution is general because the numerical values (capacitance) of the five capacitors and the numerical values (inductances) of the five inductors are not constant, but, instead, are functions of the free variable f. That is, the solution produced by genetic programming includes nine different mathematical expressions—each containing the free variable f—that establish the nine component values. One of the nine mathematical expressions is, for example,

$$C2 = \frac{1.6786 \times 10^5}{f}.$$

When all nine mathematical expressions are instantiated with a particular value of the free variable, f, the resulting circuit is a lowpass filter with a passband boundary of f. That is, genetic programming produced a general solution to the problem (not just a solution to a single instance of the problem).

The capability of genetic programming to create parameterized topologies for design problems is also illustrated by the automatic creation of a general-purpose non-PID controller (figure 6) whose blocks are parameterized by mathematical expressions containing the problem's four free variables, namely the plant's time constant, T_r, ultimate period, T_u, ultimate gain, K_u, dead time, L.

This genetically evolved controller outperforms PID controllers tuned using the widely used 1942 Ziegler-Nichols tuning rules and the recently developed 1995 Astrom and Hagglund tuning rules on an industrially representative set of plants. The authors have applied for a patent on this new non-PID controller (and other genetically evolved controllers and PID tuning rules).

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 figure 6 has a gain of

$$\left| \log \left| T_r - T_u + \log \left| \frac{\log\left(|L|^L\right)}{T_u + 1} \right| \right| \right| \qquad [31]$$

and that gain block 760 of figure 6 has a gain of

$$\left| \log \left| T_r + 1 \right| \right| \qquad [34].$$

The genetically evolved mathematical expressions for the other blocks in this controller are detailed in Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003.

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

- a 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,
- a parameterized controller for controlling a three-lag plant whose time constant is specified by a free variable, and
- a parameterized controller for controlling plants belonging to two families.

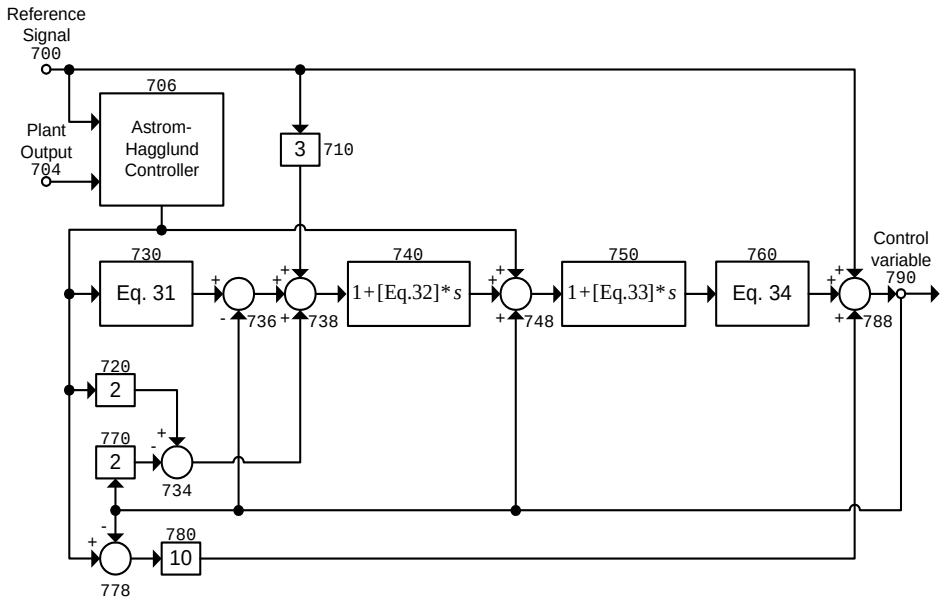

Figure 6 G**enetically evolved general-purpose controller.**

3.6 Automatic Creation of Parameterized Topologies Containing Conditional Operators

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

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

- a lowpass filter with a variable passband boundary determined by the free variables F1 and F2 (figure 7), or
- a highpass filter with a variable passband boundary by the free variables F1 and F2 (figure 8).

Automatically created mathematical expressions parameterize the components in these automatically created circuits.

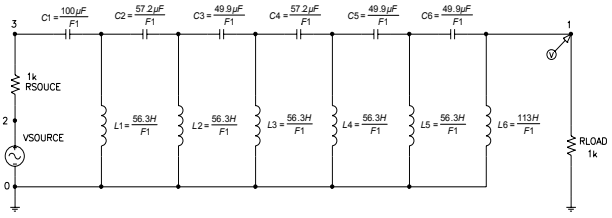

Figure 7 Genetically evolved generalized circuit when free variables call for a highpass filter (that is, F1 > F2**)**

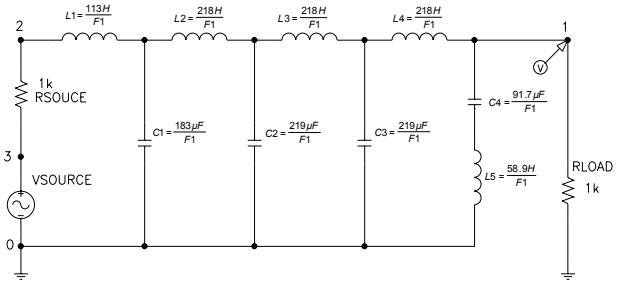

Figure 8 Genetically evolved generalized circuit when free variables call for a lowpass filter.

The techniques above have also been applied (Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003) to additional problems, including 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.

3.7 Passing Parameters to Substructures

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 9 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 a 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.

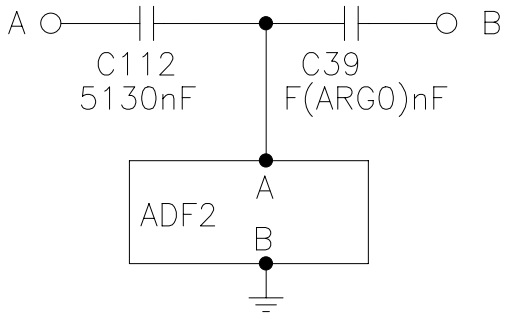

Figure 9 Subcircuit produced by three-ported automatically defined function ADF3

3.8 Novelty-Driven Evolution

One may have a scientific interest in producing novel solutions to challenging design problems. Or, one may be interested in patenting a novel design for commercial advantage. Alternatively, one may want to avoid infringing an existing patent (either to avoid paying royalties or because the patent holder is unwilling to license a competitor).

In any of the above three situations, a fitness measure may be formulated so as to incorporate the degree to which a candidate satisfies the problem's technical design requirements and the degree to which it avoids characteristics that read on prior art. That is, the evolutionary process can be directed to focus on finding solutions that are novel and that also satisfy the problem's technical design requirements.

Because circuits can be conveniently represented by labeled graphs, a graph isomorphism algorithm can be applied to the candidate circuit and various template graphs representing key characteristics of the relevant prior art. The measure of similarity can be based on the size of the maximal common subgraph between a candidate circuit and a template. For details, see Koza, Bennett, Andre, and Keane 1999.

4 Other Patented Designs

As an additional indicator of the ability of genetic programming to automatically synthesize designs, table 1 shows 10 additional 20th century patented circuits, controllers, and mathematical algorithms and six 21st-century patented circuits that have been automatically synthesized by means of genetic programming (Koza, Bennett, Andre, and Keane 1999; Koza, Keane, Yu, Bennett, and Mydlowec 2000; Koza, Keane, Streeter, Mydlowec, Yu, and Lanza 2003).

5 Conclusions

This paper has described the following eight features of genetic programming that are particularly relevant to problems of automated design:

- developmental genetic programming,
- reuse of useful substructures by crossover,
- reuse of useful substructures by subroutines,
- architecture-altering operations,
- parameterized topologies,
- parameterized topologies containing conditional operators,
- passing parameters to substructures, and
- novelty-driven evolution.

References

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

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

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

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

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. 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., Bennett III, Forrest H, Andre, David, Keane, Martin A., and Brave Scott. 1999. *Genetic Programming III Videotape: Human-Competitive Machine Intelligence*. 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. In press.

Koza, John R., Keane, Martin A., Yu, Jessen, Bennett, Forrest H III, and Mydlowec, William. 2000. Automatic creation of human-competitive programs and controllers by means of genetic programming. *Genetic Programming and Evolvable Machines*. (1) 121 - 164.

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

Table 1 Sixteen patented circuits, controllers, or mathematical algorithms that have been reinvented by means of genetic programming

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