Version 1 – Submitted December 11, 1998 for Evolution as Computation Workshop (EAC) at DIMACS to be held in Princeton, New Jersey on January 11 – 12 (Monday – Tuesday), 1999.

Genetic Programming: Biologically Inspired Computation that Creatively Solves Non-Trivial Problems

John R. Koza
Consulting Professor
Section on Medical Informatics
Department of Medicine
Medical School Office Building
Stanford University
Stanford, California 94305
koza@stanford.edu
http://www.smi.stanford.edu/people/koza/

Forrest H Bennett III
Chief Scientist
Genetic Programming Inc.
Box 1669
Los Altos, California 94023
forrest@evolute.com
http://www.genetic-programming.com

David Andre
Division of Computer Science
University of California
Berkeley, California 94720
dandre@cs.berkeley.edu

Martin A. Keane
Chief Scientist
Econometrics Inc.
111 E. Wacker Dr.
Chicago, Illinois 60601
makeane@ix.netcom.com

Abstract

This paper describes a biologically inspired domain-independent technique, called genetic programming, that automatically creates computer programs to solve problems. Starting with a primordial ooze of thousands of randomly created computer programs, genetic programming progressively breeds a population of computer programs over a series of generations using the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication, gene deletion, and certain mechanisms of developmental biology. The technique is illustrated by applying it to a non-trivial problem involving the automatic synthesis (design) of a lowpass filter circuit. The evolved results are competitive with human-produced solutions to the problem. In fact, four of the automatically created circuits exhibit human-level creativity and inventiveness, as evidenced by the fact that they correspond to four inventions that were patented between 1917 and 1936.

1. Introduction

One of the central challenges of computer science is to get a computer to solve a problem without explicitly programming it. In particular, it would be desirable to have a problem-independent system whose input is a high-level statement of a problem's requirements and whose output is a working computer program that solves the given problem. Paraphrasing Arthur Samuel (1959), this challenge concerns

> How can computers be made to do what needs to be done, without being told exactly how to do it?

As Samuel also explained (Samuel 1983),

> "The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence."

Three questions arise:

- Can computer programs be automatically created?
- Can automatically created programs be competitive with human-produced programs?
- Can the automatic process exhibit creativity and inventiveness?

This paper provides an affirmative answer to all three questions.

Section 2 describes genetic programming. Section 3 presents a problem involving the automatic synthesis (design) of an analog electrical circuit, namely a lowpass filter. Section 4 details the circuit-constructing functions used in applying genetic programming to the problem of analog circuit synthesis. Section 5 presents the preparatory steps required for applying genetic programming to the lowpass filter problem. Section 6 shows the results.

2. Background on Genetic Programming

Genetic programming is a biologically inspired, domain-independent method that automatically creates a computer program from a high-level statement of a problem's requirements.

John Holland's pioneering book *Adaptation in Natural and Artificial Systems* (1975) described a domain-independent algorithm, called the genetic algorithm, based on an evolutionary process involving natural selection, recombination, and mutation. In the most commonly used form of the genetic algorithm, each point in the search space of the given problem is encoded into a fixed-length string of characters reminiscent of a strand of DNA. The genetic algorithm then conducts a search in the space of fixed-length character strings to find the best (or at least a very good) solution to the problem by genetically breeding a population of character strings over a number of generations. Numerous practical problems can be solved using the genetic algorithm. Recent work in the field of genetic algorithms is described in Goldberg 1989, Michalewicz 1996, Mitchell 1996, Gen and Cheng 1997, and Back 1997.

Genetic programming is an extension of the genetic algorithm in which the population consists of computer programs. The goal of genetic programming is to provide a domain-independent problem-solving method that automatically creates a computer program from a high-level statement of a problem's requirements. Starting with a primordial ooze of thousands of randomly created computer programs, genetic programming progressively breeds a population of computer programs over a series of generations using the Darwinian principle of natural selection, recombination (crossover), mutation, gene duplication, gene deletion, and certain mechanisms of developmental biology. Work on genetic programming is described in Koza 1992; Koza and Rice 1992; Kinnear 1994; Koza 1994a; Koza 1994b; Angeline and Kinnear 1996; Koza, Goldberg, Fogel, and Riolo 1996; Koza et al 1997; Koza et al 1998; Banzhaf, Poli, Schoenauer, and Fogarty 1998, Banzhaf, Nordin, Keller, and Francone 1998; Spector, Langdon, O'Reilly, and Angeline 1999; Koza, Bennett, Andre, and Keane 1999, and on the World Wide Web at `www.genetic-programming.org`. The computer programs are compositions of functions (e.g., arithmetic operations, conditional operators, problem-specific functions) and terminals (e.g., external inputs, constants, zero-argument functions). The programs may be thought of as trees whose points are labeled with the functions and whose leaves are labeled with the terminals.

Genetic programming breeds computer programs to solve problems by executing the following three steps:

(1) Randomly create an initial population of individual computer programs.

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

(a) Assign a fitness value to each individual program in the population using the fitness measure.

(b) Create a new population of individual programs by applying the following three genetic operations. The genetic operations are applied to one or two individuals in the population selected with a probability based on fitness (with reselection allowed).

(i) *Reproduction*: Reproduce an existing individual by copying it into the new population.

(ii) *Crossover*: Create two new individual programs from two existing parental individuals by genetically recombining subtrees from each program using the crossover operation at randomly chosen crossover points in the parental individuals.

(iii) *Mutation*: Create a new individual from an existing parental individual by randomly mutating one randomly chosen subtree of the parental individual.

(3) Designate the individual computer program that is identified by the method of result designation (e.g., the *best-so-far* individual) as the result of the run of genetic programming. This result may represent a solution (or an approximate solution) to the problem.

Genetic programming starts with an initial population (generation 0) of randomly generated computer programs composed of the given primitive functions and terminals. The creation of this initial random population is a blind random search of the space of computer programs.

The computer programs in generation 0 of a run of genetic programming will almost always have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat more fit than others. These differences in performance are then exploited so as to direct the search into promising areas of the search space. The Darwinian principle of reproduction and survival of the fittest and the genetic operation of crossover (augmented by occasional mutation) are used to create a new population of offspring programs from the current population of computer programs.

The reproduction operation involves probabilistically selecting a computer program from the current population of programs based on fitness (i.e., the better the fitness, the more likely the individual is to be selected) and allowing it to survive by copying it into the new population.

The crossover operation creates new offspring computer programs from two parental programs selected probabilistically based on fitness. The parental programs in genetic programming are typically of different sizes and shapes. The offspring programs are composed of subexpressions (subtrees, subprograms) from their parents.

For example, consider the following computer program (presented here as a LISP S-expression):

`(+ (* 0.234 Z) (- X 0.789)),`

which one would ordinarily write as

$0.234\,Z + X - 0.789.$

This program takes two inputs (`X` and `Z`) and produces a floating point output.

Also, consider a second program:

`(* (* Z Y) (+ Y (* 0.314 Z))).`

One crossover point is randomly and independently chosen in each parent. Suppose that the crossover points are the `*` in the first parent and the `+` in the second parent. These two crossover fragments are the subexpressions rooted at the crossover points and are underlined in the above two parental computer programs.

The two offspring resulting from crossover are

`(+ (+ Y (* 0.314 Z)) (- X 0.789))`

and

`(* (* Z Y) (* 0.234 Z)).`

Crossover creates new computer programs using parts of existing parental programs. Because entire sub-trees are swapped, the crossover operation produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. The two offspring here are typical of the offspring produced by the crossover operation in that they are different from both of their parents and different from each other in size and shape. Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to regions of the search space whose programs contain parts of promising programs.

The mutation operation creates an offspring computer program from one parental program selected based on fitness. One mutation point is randomly and independently chosen and the subtree occurring at that point is deleted. Then, a new subtree is grown at that point using the same growth procedure as was originally used to create the initial random population.

For example, consider the following parental program (presented as a LISP S-expression) composed of Boolean functions and terminals:

`(OR (AND D2 D1 (NOR D0 D1))).`

Suppose that the `AND` is randomly chosen as the mutation point (out of the seven points in the program tree). The three-point subtree rooted at the `AND` corresponds to the underlined portion of the LISP S-expression above. The subtree rooted at the chosen mutation point is deleted. In this example, the subtree consists of the three points `(AND D2 D1)`. A new subtree, such as

`(AND (NOT D0) (NOT D1)),`
is randomly grown using the available functions and terminals and inserted in lieu of the subtree `(AND D2 D1)`. The result of the mutation operation is
`(OR ((AND (NOT D0) (NOT D1)) (NOR D0 D1))).`
The offspring here is typical of the offspring produced by the mutation operation in that they is different than its parent in size and shape.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation). Each individual in the new population of programs is then measured for fitness, and the process is repeated over many generations.

The dynamic variability of the computer programs that are created during the run are important features of genetic programming. It is often difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance.

Scalable automated programming requires some hierarchical mechanism to exploit, by reuse and parameterization, the regularities, symmetries, homogeneities, similarities, patterns, and modularities inherent in problem environments. Subroutines provide this mechanism in ordinary computer programs. Automatically defined functions (Koza 1994a, 1994b) implement this mechanism within the context of genetic programming. Automatically defined functions are implemented by establishing a constrained syntactic structure for the individual programs in the population. Each multi-part program in the population contains one (or more) automatically defined functions and one (or more) main result-producing branches. The result-producing branch usually has the ability to call one or more of the automatically defined functions. An automatically defined function may have the ability to refer hierarchically to other already-defined automatically defined functions.

The initial random generation is created so that every individual program in the population consists of automatically defined function(s) and result-producing branch(es) in accordance with the problem's constrained syntactic structure. Since a constrained syntactic structure is involved, crossover and mutation are performed so as to preserve this syntactic structure in all offspring.

Architecture-altering operations enhance genetic programming with automatically defined functions by providing a way to automatically determine the number of such automatically defined functions, the number of arguments that each automatically defined function possesses, and the nature of the hierarchical references, if any, among such automatically defined functions (Koza 1995). These operations include branch duplication, argument duplication, branch creation, argument creation, branch deletion, and argument deletion. The architecture-altering operations are motivated by the naturally occurring mechanism of gene duplication that creates new proteins (and hence new structures and new behaviors in living things) as described by Susumu Ohno in. *Evolution by Gene Duplication* (1970). Details are found in Koza, Bennett, Andre, and Keane 1999.

Genetic programming has been applied to numerous problems in fields such as system identification, control, classification, design, optimization, and automatic programming.

3. Statement of the Illustrative Problem

Design is a major activity of practicing engineers. The design process entails creation of a complex structure to satisfy user-defined requirements. Since the design process typically entails tradeoffs between competing considerations, the end product of the process is usually a satisfactory and compliant design as opposed to a perfect design. Design is usually viewed as requiring creativity and human intelligence. Consequently, the field of design is a source of challenging problems for automated techniques of machine intelligence. In particular, design problems are useful for determining whether an automated technique can produce results that are competitive with human-produced results.

The design (synthesis) of analog electrical circuits is especially challenging. The design process for analog circuits begins with a high-level description of the circuit's desired behavior and characteristics and entails creation of both the topology and the sizing of a satisfactory circuit. The topology comprises the gross number of components in the circuit, the type of each component (e.g., a capacitor), and a list of all connections between the components. The sizing involves specifying the values (typically numerical) of each of the circuit's components.

Although considerable progress has been made in automating the synthesis of certain categories of purely digital circuits, the synthesis of analog circuits and mixed analog-digital circuits has not proved to be as amenable to automation. There is no previously known general technique for automatically creating an analog circuit from a high-level statement of the design goals of the circuit. Describing "the analog dilemma," O. Aaserud and I. Ring Nielsen (1995) noted

> "Analog designers are few and far between. In contrast to digital design, most of the analog circuits are still handcrafted by the experts or so-called 'zahs' of analog design. The design process is

characterized by a combination of experience and intuition and requires a thorough knowledge of the process characteristics and the detailed specifications of the actual product.

"Analog circuit design is known to be a knowledge-intensive, multiphase, iterative task, which usually stretches over a significant period of time and is performed by designers with a large portfolio of skills. It is therefore considered by many to be a form of art rather than a science."

This paper focuses on one particular problem of analog circuit synthesis, namely the design of a lowpass filter circuit composed of capacitors and inductors. A simple *filter* is a one-input, one-output electronic circuit that receives a signal as its input and passes the frequency components of the incoming signal that lie in a specified range (called the *passband*) while suppressing the frequency components that lie in all other frequency ranges (the *stopband*). In particular, the goal is to design a lowpass filter that passes all frequencies below 1,000 Hertz (Hz) and suppresses all frequencies above 2,000 Hz.

The approach described in this paper has been applied to many other problems of analog circuit synthesis, including the design of a amplifiers, computational circuits, a temperature-sensing circuit, a voltage reference circuit, a time-optimal robot controller circuit, a difficult-to-design asymmetric bandpass filter, a crossover filter, a double passband filter, bandstop filter, frequency discriminator circuits, and a frequency-measuring circuit (as described in detail in Koza, Bennett, Andre, and Keane 1999).

4. Applying Genetic Programming to the Problem

Genetic programming can be applied to the problem of synthesizing circuits if a mapping is established between the program trees (rooted, point-labeled trees – that is, acyclic graphs – with ordered branches) used in genetic programming and the labeled cyclic graphs germane to electrical circuits. The principles of developmental biology provide the motivation for mapping trees into circuits by means of a developmental process that begins with a simple embryo. For circuits, the embryo typically includes fixed wires that connect the inputs and outputs of the particular circuit being designed and certain fixed components (such as source and load resistors). Until these wires are modified, the circuit does not produce interesting output. An electrical circuit is developed by progressively applying the functions in a circuit-constructing program tree to the modifiable wires of the embryo (and, during the developmental process, to new components and modifiable wires).

An electrical circuit is created by executing the functions in a circuit-constructing program tree. The functions are progressively applied in a developmental process to the embryo and its successors until all of the functions in the program tree are executed. That is, the functions in the circuit-constructing program tree progressively side-effect the embryo and its successors until a fully developed circuit eventually emerges. The functions are applied in a breadth-first order.

The functions in the circuit-constructing program trees are divided into five categories: (1) topology-modifying functions that alter the circuit topology, (2) component-creating functions that insert components into the circuit, (3) development-controlling functions that control the development process by which the embryo and its successors is changed into a fully developed circuit, (4) arithmetic-performing functions that appear in subtrees as argument(s) to the component-creating functions and specify the numerical value of the component, and (5) automatically defined functions that appear in the automatically defined functions and potentially enable certain substructures of the circuit to be reused (with parameterization).

Each branch of the program tree is created in accordance with a constrained syntactic structure. Each branch is composed of topology-modifying functions, component-creating functions, development-controlling functions, and terminals. Component-creating functions typically have one arithmetic-performing subtree, while topology-modifying functions, and development-controlling functions do not. Component-creating functions and topology-modifying functions are internal points of their branches and possess one or more arguments (construction-continuing subtrees) that continue the developmental process. The syntactic validity of this constrained syntactic structure is preserved using structure-preserving crossover with point typing. For details, see Koza, Bennett, Andre, and Keane 1999.

4.1. The Embryonic Circuit

An electrical circuit is created by executing a circuit-constructing program tree that contains various component-creating, topology-modifying, and development-controlling functions. Each tree in the population creates one circuit. The specific embryo used depends on the number of inputs and outputs.

Figure 1 shows a one-input, one-output embryonic (initial) circuit in which VSOURCE is the input signal and VOUT is the output signal (the probe point). The circuit is driven by an incoming alternating circuit source VSOURCE. There is a fixed load resistor RLOAD and a fixed source resistor RSOURCE in the embryo. In

addition to the fixed components, there are two modifiable wires, Z0 and Z1. All development originates from these modifiable wires.

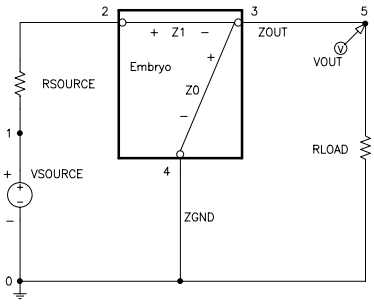

Figure 1 One-input, one-output embryonic (initial) circuit .

4.2. Component-Creating Functions

The component-creating functions insert a component into the developing circuit and assign component value(s) to the component.

Each component-creating function has a writing head that points to an associated highlighted component in the developing circuit and modifies that component in a specified manner. The construction-continuing subtree of each component-creating function points to a successor function or terminal in the circuit-constructing program tree.

The arithmetic-performing subtree of a component-creating function consists of a composition of arithmetic functions (addition and subtraction) and random constants (in the range -1.000 to +1.000). The arithmetic-performing subtree specifies the numerical value of a component by returning a floating-point value that is interpreted on a logarithmic scale as the value for the component in a range of 10 orders of magnitude (using a unit of measure that is appropriate for the particular type of component).

The two-argument resistor-creating R function causes the highlighted component to be changed into a resistor. The value of the resistor in kilo Ohms is specified by its arithmetic-performing subtree.

Figure 2 shows a modifiable wire Z0 connecting nodes 1 and 2 of a partial circuit containing four capacitors (C2, C3, C4, and C5). Figure 3 shows the result of applying the R function to the modifiable wire Z0 of figure 2..

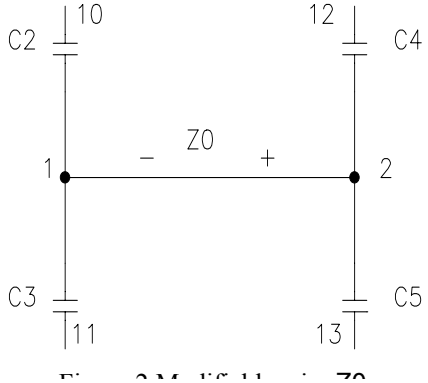

Figure 2 Modifiable wire Z0

Similarly, the two-argument capacitor-creating C function causes the highlighted component to be changed into a capacitor whose value in Micro-Farads is specified by its arithmetic-performing subtree. In addition, the two-argument inductor-creating L function causes the highlighted component to be changed into an inductor whose value in Micro-Henrys is specified by its arithmetic-performing subtree.

4.3. Topology-Modifying Functions

A topology-modifying function in a program tree points to an associated highlighted component and modifies the topology of the developing circuit.

The three-argument SERIES division function creates a series composition of the modifiable wire or modifiable component with which it is associated, a copy of the modifiable wire or modifiable component with which it is associated, one new modifiable wire (with a writing head), and two new nodes. Figure 4 shows the result of applying the SERIES function to the resistor R1 from figure 3. After execution of the SERIES function, resistors R1 and R7 and modifiable wire Z6 remain modifiable. All three are associated with the top-most function in one of the three construction-continuing subtrees of the SERIES function.

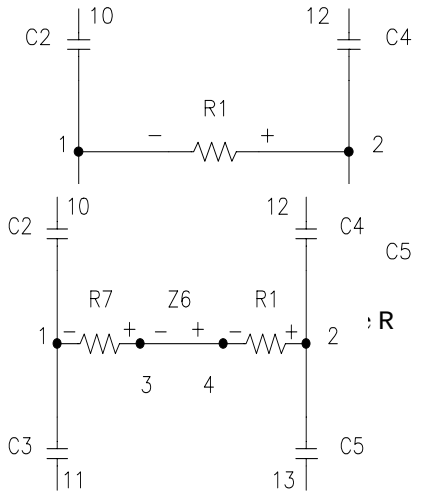

Figure 4 Result after applying the SERIES division function.

The reader is referred to Koza, Bennett, Andre, and Keane 1999 for a detailed description of all the circuit-constructing functions mentioned herein.

The four-argument PARALLEL0 parallel division function creates a parallel composition consisting of the modifiable wire or modifiable component with which it is associated, a copy of the modifiable wire or modifiable component with which it is associated, two new modifiable wires (each with a writing head), and two new nodes. There are actually two topologically distinct outcomes of a parallel division. Since we require the outcome of all circuit-constructing functions to be deterministic, there are two members (called PARALLEL0 and PARALLEL1) in the PARALLEL family of topology-modifying functions. The two functions operate differently depending on degree and numbering of the preexisting components in the developing circuit. The use of the two functions breaks the symmetry between

the potentially distinct outcomes.

The one-argument polarity-reversing FLIP function reverses the polarity of the highlighted component.

The two-argument TWO_GROUND ("ground") function enables any part of a circuit to be connected to ground. The TWO_GROUND function creates a new node and a composition of two modifiable wires and one nonmodifiable wire such that the nonmodifiable wire makes an unconditional connection to ground.

The eight two-argument functions in the TWO_VIA family of functions (called TWO_VIA0, ..., TWO_VIA7), each create a new node and a composition of two modifiable wires and one nonmodifiable wire such that the nonmodifiable wire makes a connection, called a *via*, to a designated one of eight imaginary numbered layers (0 to 7) of an imaginary silicon wafer on which the circuit resides. the TWO_VIA functions provide a way to connect distant parts of a circuits.

The zero-argument SAFE_CUT function causes the highlighted component to be removed from the circuit provided that the degree of the nodes at both ends of the highlighted component is three (i.e., no dangling components or wires are created).

4.4. Development-Controlling Functions

The one-argument NOOP ("No Operation") function has no effect on the modifiable wire or modifiable component with which it is associated; however, it has the effect of delaying activity on the developmental path on which it appears in relation to other developmental paths in the overall circuit-constructing program tree.

The zero-argument END function makes the modifiable wire or modifiable component with which it is associated non-modifiable (thereby ending a particular developmental path).

4.5. Example of Developmental Process

Figure 5 is an illustrative circuit-constructing program tree shown as a rooted, point-labeled tree with ordered branches. The overall program consists of two main result-producing branches joined by a connective LIST function (labeled 1 in the figure). The first (left) result-producing branch is rooted at the capacitor-creating C function (labeled 2). The second result-producing branch is rooted at the polarity-reversing FLIP function (labeled 3). This figure also contains four occurrences of the inductor-creating L function (at 17, 11, 20, and 12). The figure contains two occurrences of the topology-modifying SERIES function (at 5 and 10). The figure also contains five occurrences of the development-controlling END function (at 15, 25, 27, 31, and 22) and one occurrence of the development-controlling "no operation" NOP function (at 6). There is a seven-point arithmetic-performing subtree at 4 under the capacitor-creating C function at 4. Similarly, there is a three-point arithmetic-performing subtree at 19 under the inductor-creating L function at 11. There are also one-point arithmetic-performing subtrees (i.e.,

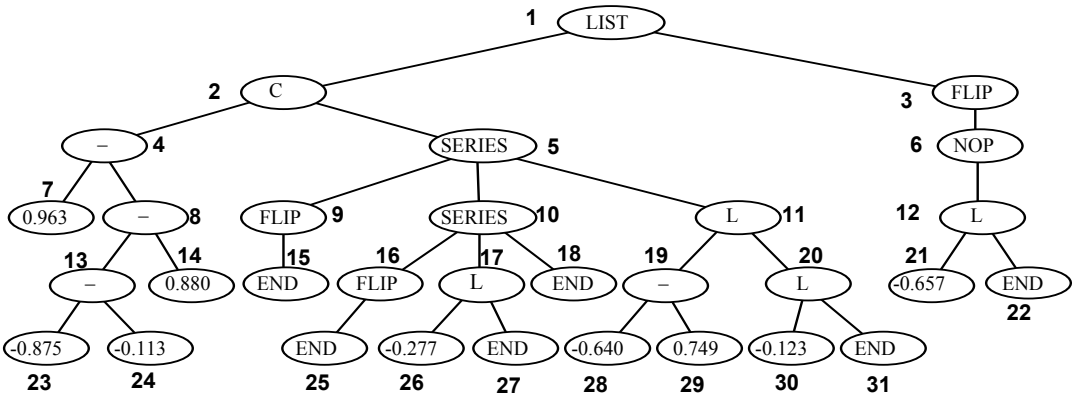

Figure 5 Illustrative circuit-constructing program tree.

constants) at 26, 30, and 21. Additional details can be found in Koza, Bennett, Andre, and Keane 1999.

5. Preparatory Steps

Before applying genetic programming to a problem of circuit design, seven major preparatory steps are required: (1) identify the embryonic circuit, (2) determine the architecture of the circuit-constructing program trees, (3) identify the primitive functions of the program trees, (4) identify the terminals of the program trees, (5) create the fitness measure, (6) choose control parameters for the run, and (7) determine the termination criterion and method of result designation.

5.1. Embryonic Circuit

The embryonic circuit used on a particular problem depends on the circuit's number of inputs and outputs. A one-input, one-output embryo with two modifiable wires (figure 1) was used.

5.2. Program Architecture

Since there is one result-producing branch in the program tree for each modifiable wire in the embryo, the architecture of each circuit-constructing program tree depends on the embryonic circuit. Two result-producing branches were used for the filter problems.

The architecture of each circuit-constructing program tree also depends on the use, if any, of automatically defined functions. Automatically defined functions provide a mechanism enabling certain substructures to be reused and are described in detail in Koza, Bennett, Andre, and Keane 1999. Automatically defined functions and architecture-altering operations were not used here.

5.3. Function and Terminal Sets

The function set for each design problem depends on the type of electrical components that are to be used for constructing the circuit.

The function set included two component-creating functions (for inductors and capacitors), topology-modifying functions (for series and parallel divisions and for flipping components), one development-controlling function ("no operation"), functions for creating a via to ground, and functions for connecting pairs of points. That is, the function set, $\mathcal{F}_{ccs}$, for each construction-continuing subtree was

$\mathcal{F}_{ccs}$ = {`L`, `C`, `SERIES`, `PARALLEL0`, `PARALLEL1`, `FLIP`, `NOOP`, `TWO_GROUND`, `TWO_VIA0_`, `TWO_VIA1`, `TWO_VIA2`, `TWO_VIA3`, `TWO_VIA4`, `TWO_VIA5`, `TWO_VIA6`, `TWO_VIA7`}.

The terminal set, $\mathcal{T}_{ccs}$, for each construction-continuing subtree was

$\mathcal{T}_{ccs}$ = {`END`, `SAFE_CUT`}.

The terminal set, $\mathcal{T}_{aps}$, for each arithmetic-performing subtree consisted of

$\mathcal{T}_{aps} = \{\Re\}$,

where $\Re$ represents floating-point random constants from –1.0 to +1.0.

The function set, $\mathcal{F}_{aps}$, for each arithmetic-performing subtree was,

$\mathcal{F}_{aps}$ = {`+`, `-`}.

The terminal and function sets were identical for all result-producing branches for a particular problem.

5.4. Fitness Measure

The evolutionary process is driven by the *fitness measure*. Each individual computer program in the population is executed and then evaluated, using the fitness measure. The nature of the fitness measure varies with the problem. The high-level statement of desired circuit behavior is translated into a well-defined measurable quantity that can be used by genetic programming to guide the evolutionary process. The evaluation of each individual circuit-constructing program tree in the population begins with its execution. This execution progressively applies the functions in each program tree to an embryonic circuit, thereby creating a fully developed circuit. A netlist is created that identifies each component of the developed circuit, the nodes to which each component is connected, and the value of each component. The netlist becomes the input to our modified version of the 217,000-line SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program (Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994). SPICE then determines the behavior of the circuit. It was necessary to make considerable modifications in SPICE so that it could run as a submodule within the genetic programming system.

The desired lowpass filter has a passband below 1,000 Hz and a stopband above 2,000 Hz. The circuit is driven by an incoming AC voltage source with a 2 volt amplitude. In this problem, a voltage in the passband of exactly 1 volt and a voltage in the stopband of exactly 0 volts is regarded as ideal. The (preferably small) variation within the passband is called the *passband ripple*. Similarly, the incoming signal is never fully reduced to zero in the stopband of an actual filter. The (preferably small) variation within the stopband is called the *stopband ripple*. A voltage in the passband of between 970 millivolts and 1 volt (i.e., a passband ripple of 30 millivolts or less) and a voltage in the stopband of between 0 volts and 1 millivolts (i.e., a stopband ripple of 1 millivolts or less) is regarded as acceptable. Any voltage lower than 970 millivolts in the passband and any voltage above 1 millivolts in the stopband is regarded as unacceptable.

Since the high-level statement of behavior for the desired circuit is expressed in terms of frequencies, the voltage `VOUT` is measured in the frequency domain. SPICE performs an AC small signal analysis and reports the circuit's behavior over five decades (between 1 Hz and 100,000 Hz) with each decade being divided into 20 parts (using a logarithmic scale), so that there are a total of 101 fitness cases.

Fitness is measured in terms of the sum over these cases of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit at the probe point VOUT and the target value for voltage. The smaller the value of fitness, the better. A fitness of zero represents an (unattainable) ideal filter.

Specifically, the standardized fitness is

$$F(t) = \sum_{i=0}^{100} (W(d(f_i), f_i)d(f_i))$$

where f_i is the frequency of fitness case i; $d(x)$ is the absolute value of the difference between the target and observed values at frequency x; and $W(y,x)$ is the weighting for difference y at frequency x.

The fitness measure is designed to not penalize ideal values, to slightly penalize every acceptable deviation, and to heavily penalize every unacceptable deviation. Specifically, the procedure for each of the 61 points in the 3-decade interval between 1 Hz and 1,000 Hz for the intended passband is as follows:

- If the voltage equals the ideal value of 1.0 volt in this interval, the deviation is 0.0.
- If the voltage is between 970 millivolts and 1 volt, the absolute value of the deviation from 1 volt is weighted by a factor of 1.0.
- If the voltage is less than 970 millivolts, the absolute value of the deviation from 1 volt is weighted by a factor of 10.0.

The acceptable and unacceptable deviations for each of the 35 points from 2,000 Hz to 100,000 Hz in the intended stopband are similarly weighed (by 1.0 or 10.0) based on the amount of deviation from the ideal voltage of 0 volts and the acceptable deviation of 1 millivolts.

For each of the five "don't care" points between 1,000 and 2,000 Hz, the deviation is deemed to be zero.

The number of "hits" for this problem (and all other problems herein) is defined as the number of fitness cases for which the voltage is acceptable or ideal or that lie in the "don't care" band (for a filter).

Many of the random initial circuits and many that are created by the crossover and mutation operations in subsequent generations cannot be simulated by SPICE. These circuits receive a high penalty value of fitness (10^8) and become the worst-of-generation programs for each generation.

5.5. Control Parameters

The population size, M, was 320,000. The probability of crossover was approximately 89%; reproduction 10%; and mutation 1%. Our usual control parameters were used (Koza, Bennett, Andre, and Keane 1999, Appendix D).

5.6. Termination Criterion and Results Designation

The maximum number of generations, G, is set to an arbitrary large number (e.g., 501) and the run was manually monitored and manually terminated when the fitness of the best-of-generation individual appeared to have reached a plateau. The best-so-far individual is harvested and designated as the result of the run.

5.7. Implementation on Parallel Computer

The problem was run on a medium-grained parallel Parsytec computer system consisting of 64 80-MHz PowerPC 601 processors arranged in an 8 by 8 toroidal mesh with a host PC Pentium type computer. The distributed genetic algorithm (Andre and Koza 1996) with unsynchronized generations was used with a population size of Q = 5,000 at each of the D = 64 demes (semi-isolated subpopulations) for a total population, M, of 320,000. On each generation, four boatloads of emigrants, each consisting of B = 2% (the migration rate) of the node's subpopulation (selected on the basis of fitness) were dispatched to each of the four adjacent processing nodes.

6. Results

The creation of the initial random population is a blind random search of the search space of the problem. The best circuit from generation 0 has a fitness of 61.7 and scores 52 hits (out of 101).

Figures 6, 7, 8, and 9 show the behavior of the best circuits from generation 0, 10, 15, and 49, respectively, of one run of genetic programming. The horizontal axis represents five decades of frequencies from 1 Hz to 100,000 Hz on a logarithmic scale. The vertical axis represents output voltage on a linear scale. Excluding the fixed source and load resistors of the test fixture of the embryonic circuit, the best-of-generation circuit from generation 0 consists of only a lone 358 nF capacitor that shunts the incoming signal to ground. A good filter cannot be created by a single capacitor. However, even a single capacitor differentially passes higher frequencies to ground and performs a certain amount of filtering. Figure 6 shows that the best circuit from generation 0 bears some resemblance to the desired lowpass filter in that it passes frequencies up to about 70 Hz at nearly a full volt and it almost fully suppresses frequencies near 100,000 Hz. However, its transition region is exceedingly leisurely. Nonetheless, in the valley of the blind, the one-eyed man is king. Moreover, as will be seen momentarily, this

modest beginning serves as a building block that will become incorporated in the 100%-compliant lowpass filter that will eventually be evolved.

The evolutionary process produces better and better individuals as the run progresses. For example, the best circuit from generation 10 has inductors in series with the incoming signal as well as a single capacitor shunted to ground. Figure 7 shows that that the frequencies up to about 200 Hz are passed at nearly full voltage and that frequencies above 10,000 Hz are almost fully suppressed. Figure 8 shows that the best circuit from generation 15 (with two inductors in series with the incoming signal and three capacitors shunted to ground) comes closer to meeting the requirements of this design problem.

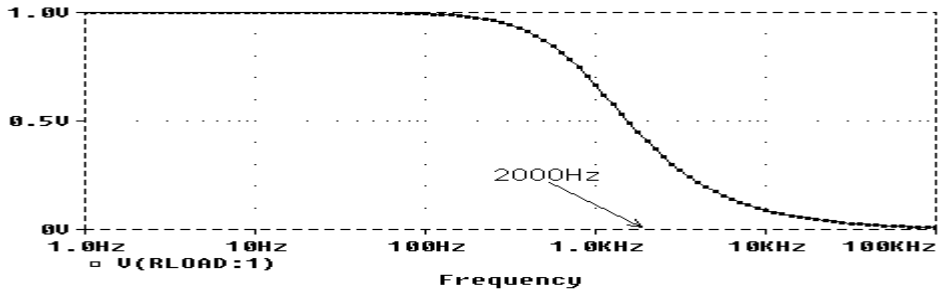

Figure 6 Frequency domain behavior of the best circuit of generation 0.

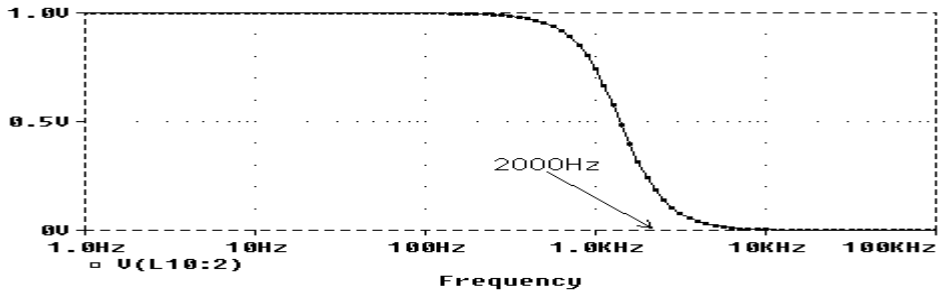

Figure 7 Frequency domain behavior of the best circuit of generation 10.

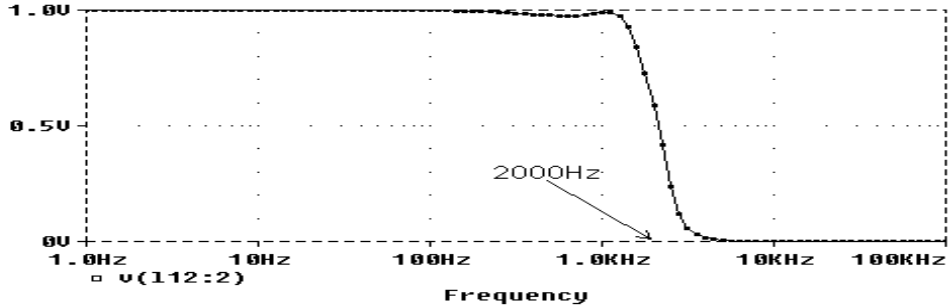

Figure 8 Frequency domain behavior of the best circuit of generation 15.

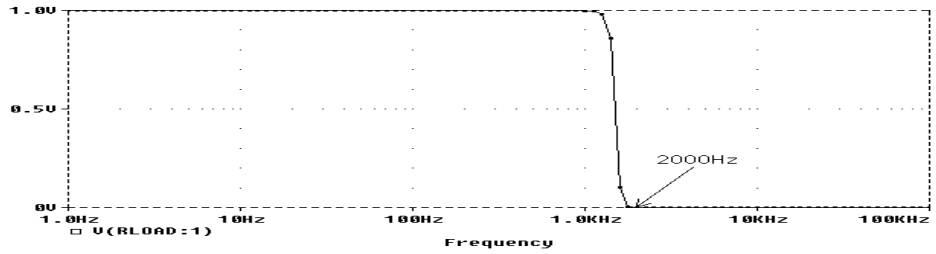

Figure 9 Frequency domain behavior of 100%-compliant seven-rung ladder circuit from generation 49.

6. 1 Campbell 1917 Ladder Filter Patent

The best circuit (figure 10) of generation 49 from this run is 100% compliant with the problem's design requirements in the sense that it scored 101 hits (out of 101). It has a near-zero fitness of 0.00781 (about five orders of magnitude better than the best circuit of generation 0). As can be seen, this evolved circuit consists of seven inductors (L5, L10, L22, L28, L31, L25, and L13) arranged horizontally across the top of the figure "in series" with the incoming signal VSOURCE and the source resistor RSOURCE. It also contains seven capacitors (C12, C24, C30, C3, C33, C27, and C15) that are each shunted to ground. This circuit is a classical ladder filter with seven rungs (Williams and Taylor 1995).

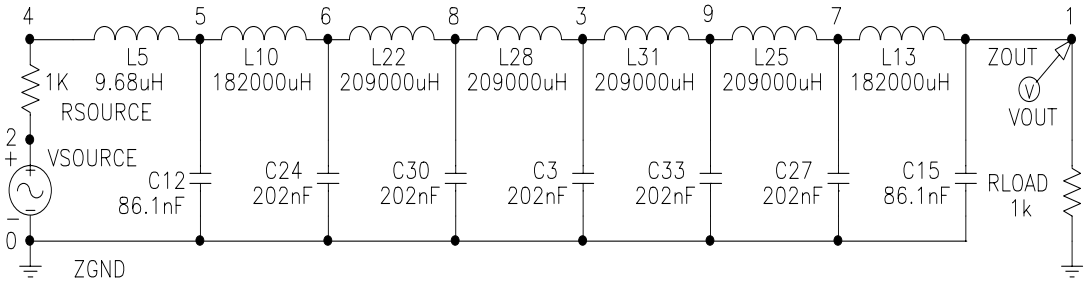

Figure 10 Evolved seven-rung ladder lowpass filter.

approximately equal capacitors are replaced by 12 equal capacitors with capacitance equal to their average value (98.5 nF). The behavior of the evolved circuit is only slightly changed by these substitutions.

Fourth, we note also that the six non-trivial inductors (L10, L22, L28, L31, L25, and L13) are approximately equal. Suppose, for sake of argument, that these six approximately equal inductors are replaced by six equal inductors with inductance equal to their average value (200,000 μH). Again, the behavior of the evolved circuit is only slightly changed by these substitutions.

The behavior in the frequency domain of the circuit resulting from the above four changes is almost the same as that of the evolved circuit of Figure 10. In fact, the modified circuit is 100%-compliant (i.e., scores 101 hits). The modified circuit can be viewed as what is now known as a cascade of six identical symmetric π-sections. Each π-section consists of an inductor of inductance L (where L equals 200,000 μH) and two equal capacitors of capacitance $C/2$ (where C equals 197 nF). In each π-section, the two 98.5 nF capacitors constitute the vertical legs of the π and the one 200,000 μH inductor constitutes the horizontal bar across the top of the π. Such π-sections are characterized by two key parameters. The first parameter is the characteristic resistance (impedance) of the π-section. This characteristic resistance should match the circuit's fixed load resistance RLOAD (1,000 Ω). The second parameter is the nominal cutoff frequency which separates the filter's passband from its stopband. This second parameter should lie somewhere in the transition region between the end of the passband (1,000 Hz) and the beginning of the stopband (2,000 Hz). The characteristic resistance, R, of each of the π-sections is given by the formula $\sqrt{(L/C)}$. This formula yields a characteristic resistance, R, of 1,008 Ω. This value is very close to the value of the 1,000 Ω load resistance of this problem. The nominal cutoff frequency, f_c, of each of the π-sections of a lowpass filter is given by the formula $1/(\pi\sqrt{LC})$. This formula yields a nominal cutoff frequency, f_c, of 1,604 Hz (i.e., roughly in the middle of the transition region between the passband and stopband of the desired lowpass filter).

The legal criteria for obtaining a U. S. patent are that the proposed invention be "new" and "useful" and

> ... the differences between the subject matter sought to be patented and the prior art are such that the subject matter as a whole would [not] have been obvious at the time the invention was made to a person having ordinary skill in the art to which said subject matter pertains. (35 *United States Code* 103a).

George Campbell was part of the renowned research team of the American Telephone and Telegraph Corporation. He received a patent for his filter in 1917 because his idea was new in 1917, because it was useful, and because satisifed the above statutory test for unobviousness. The fact that genetic programming rediscovered an electrical circuit that was unobvious "to a person having ordinary skill in the art" establishes that this evolved result satisfies Arthur Samuel's criterion (Samuel 1983) for artificial intelligence and machine learning, namely

> "The aim [is] ... to get machines to exhibit behavior, which if done by humans, would be assumed to involve the use of intelligence."

6.2 Zobel 1925 "*M*-Derived Half Section" Patent

In another run of this same problem, a 100%-compliant circuit was evolved in generation 34. This evolved circuit is roughly equivalent to what is now known as a cascade of three symmetric T-sections and an *M*-derived half section (Johnson 1950). To see this, we modify this evolved circuit from generation 34 in three ways.

First, we insert wires in lieu of two 0.138 μH inductors (whose value is about six orders of magnitude smaller than the value of the other inductors in the circuit). The behavior of this slightly modified evolved circuit (figure 11) is not noticeably affected by these changes for the frequencies of interest in this problem.

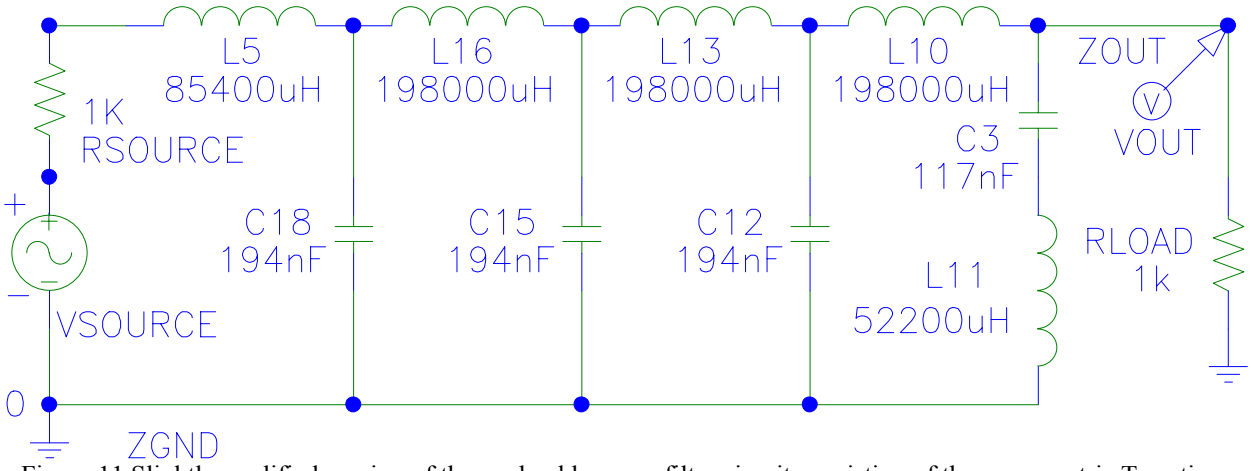

Figure 11 Slightly modified version of the evolved lowpass filter circuit consisting of three symmetric T-sections and an *M*-derived half section.

Second, we replace each of the three 198,000 μH inductors in the figure (L16, L13, and L10) with a series composition of two 99,000 μH inductors. Since the inductance of two inductors in series is equal to the sum of their inductances, this change does not affect the behavior of the circuit at all. The circuit can now be viewed as having one incoming 85,400 μH inductor (L5) and six 99,000 μH inductors in series horizontally at the top of the figure.

Third, we note also that the values of the (now) seven inductors in series horizontally across the top of the figure are approximately equal. Suppose, for sake of argument, that each of these seven approximately equal inductors are replaced by an inductor with inductance equal to their average value (97,000 μH). This change does not appreciably affect the behavior of the circuit for the frequencies of interest.

After the above changes, the evolved lowpass filter can be viewed as consisting of a cascade of three identical symmetric T-sections and an "*M*-derived half section." In particular, each T-section consists of an incoming inductor of inductance *L/2* (where *L* equals 194,000 μH), a junction point from which a capacitor of capacitance *C* (where *C* equals 194 nF) is shunted off to ground, and an outgoing inductor of inductance *L/2*. The two inductors are the horizontal arms of the "T." The final half section (so named because it has only one arm of a "T") has one incoming inductor of inductance *L/2* and a junction point from which a capacitive-inductive shunt (C3 and L11) is connected to ground.

The first three symmetric T-sections are referred to as "constant K" filter sections (Johnson l950, page 331). Such filter sections are characterized by two key parameters. The characteristic resistance, *R*, of each of each of the three T-sections is given by the formula $R = \sqrt{(L/C)}$. When the inductance, *L*, is 194,000 μH and the capacitance, *C*, is 194 nF, then the characteristic resistance, *R*, is 1,000 Ω according to this formula (i.e., equal to the value of the actual load resistor). The nominal cutoff frequency, f_c, of each of the three T-sections of a lowpass filter is given by the formula $f_c = 1/(\pi\sqrt{LC})$. This formula yields a nominal cutoff frequency, f_c, of 1,641 Hz (which is near the middle of the transition band for the desired pass filter).

In other words, both of the key parameters of the three T-sections are very close to the canonical values of "constant K" sections designed with the aim of satisfying this problem's design requirements.

The final section of the evolved circuit closely approximates a section now called an "*M*-derived half section". This final section is said to be "derived" because it is derived from the foregoing three identical "constant K" prototype sections. In the derivation, *m* is a real constant between 0 and 1. Let *m* be 0.6 here. In a canonical "*M*-derived half section" that is derived from the above "constant K" prototype section, the value of the capacitor in the vertical shunt of the half section is given by the formula *mC* (116.4 nF). The actual value of C3 in the evolved circuit is 117 nF. The value of the inductor in the vertical shunt of an "*M*-derived half section" is given by the formula $L(1-m^2)/4m$. This formula yields a value of 51,733. The actual value of L5 in the evolved circuit is 52,200 μH. The frequency, f_∞, where the attenuation first becomes complete, is given by the formula $f_\infty = f_c/\sqrt{(1-m^2)}$. This formula yields a value of f_∞ of 2,051 Hz (i.e., near the beginning of the desired stopband).

Taken as a whole, the topology and component values of the evolved circuit are reasonably close to the canonical values for the three identical symmetric T-sections and a final "*M*-derived half section" that is designed with the aim of satisfying this problem's design requirements.

Otto Zobel of American Telephone and Telegraph Company invented the idea of adding an "*M*-derived half section" to one or more "constant K" sections. As Zobel (1925) explains in U. S. patent 1,538,964,

> The principal object of my invention is to provide a new and improved network for the purpose of transmitting electric currents having their frequency within a certain range and attenuating currents of frequency within a different range. . . . Another object of my invention is to provide a wave-filter with recurrent sections not all of which are alike, and having certain advantages over a wave-filter with all its sections alike.

The advantage of Zobel's approach is a "sharper transition" in the frequency domain behavior of the filter. Claim 1 of Zobel's 1925 patent covers,

> A wave-filter having one or more half-sections of a certain kind and one or more other half-sections that are M-types thereof, M being different from unity.

Claim 2 covers,

> A wave-filter having its sections and half-sections so related that they comprise different M-types of a common prototype, M having several values for respectively different sections and half-sections.

Claim 3 goes on to cover,

> A wave-filter having one or more half-sections of a certain kind and one or more half-sections introduced from a different wave-filter having the same characteristic and the same critical frequencies and a different attenuation characteristic outside the free transmitting range.

Viewed as a whole, the evolved circuit here infringes the claims of Zobel's 1925 patent.

6.3 Johnson 1926 "Bridged T" Patent

In another run, a 100% compliant recognizable "bridged T" arrangement was evolved. The "bridged T" filter topology was invented and patented by Kenneth S. Johnson of Western Electric Company in 1926 (Johnson 1926).

The "bridged T" was invented by Kenneth S. Johnson of Western Electric Company and patented in 1926. As U. S. patent 1,611,916 (Johnson 1926) states,

> In accordance with the invention, a section of an artificial line, such as a wave filter, comprises in general four impedance paths, three of which are arranged in the form of a T network with the fourth path bridged across the transverse arms of the T. The impedances of this network, which for convenience, will be referred to as a bridged T network, bear a definite relationship to a network of the series shunt type, the characteristics of which are well known.
>
> In the forms of the invention described herein, the arms of the bridged T network consist of substantially pure reactances. Its most useful forms are found to be wave filter networks in which there is a substantially infinite attenuation at a frequency within the band to be suppressed and the network may be designed so that this frequency is very near the cut-off frequency of the filter, thus producing a very sharp separation between the transmitted and suppressed bands.

Claim 1 of patent 1,611,916 covers,

> An electrical network comprising a pair of input terminals and a pair of output terminals, an impedance path connected directly between an input terminal and an output terminal, a pair of impedance paths having a common terminal and having their other terminals connected respectively to the terminals of said first path, and a fourth impedance path having one terminal connected to said common terminal and having connections from its other terminal to the remaining input terminal and output terminal, each of said paths containing a substantial amount of reactance, the impedances of said network having such values that said network is the equivalent of a series-shunt network having desired transmission characteristics.

The "bridged T" of figure 12 involves L14, C3, C15, and L11. In particular, L14 is the "impedance path connected directly between an input terminal and an output terminal" that is referred to later as "the first path." The junction of C3, C15, and L11 is the "common terminal." C3 and C15 are the "pair of impedance paths having a common terminal and having their other terminals connected respectively to the terminals of said first path." L11 is the "fourth impedance path having one terminal connected to said common terminal and having connections from its other terminal to the remaining input terminal and output terminal" (namely, the input and output terminal of the section that are both grounded).

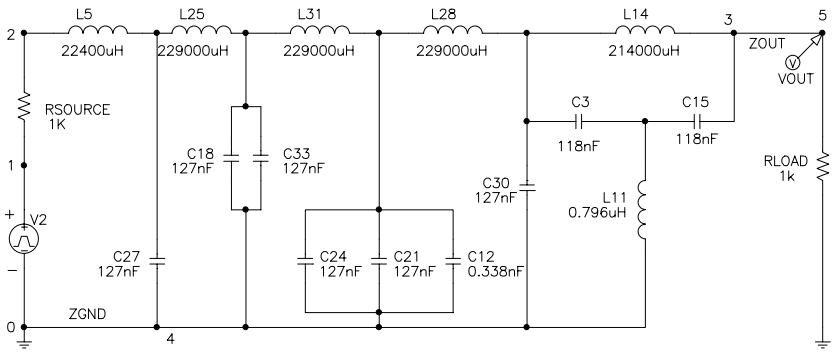

Figure 12 "Bridged T" circuit from generation 64.

6.4 Cauer 1934 – 1936 Elliptic Patents

In a run of this same problem using automatically defined functions (described in Koza, Bennett, Andre, and Keane 1999), a 100% compliant circuit emerged in generation 31. After all of the pairs and triplets of series inductors in the evolved circuit are consolidated (as shown in figure 13), it can be seen that the circuit has the equivalent of six inductors horizontally across the top of the circuit and five vertical shunts. Each vertical shunt consists of an inductor and a capacitor.

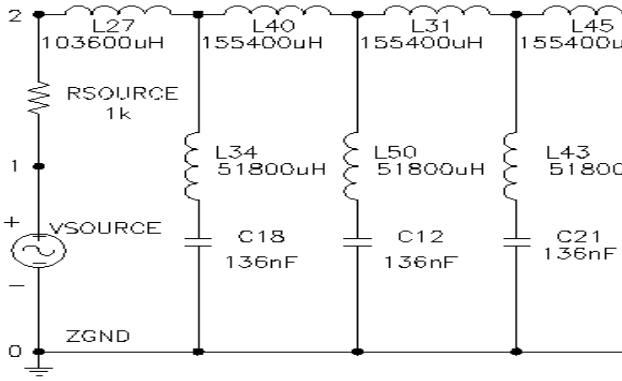

Figure 13 Evolved Cauer (elliptic) filter topology from generation 31.

This circuit has the recognizable elliptic topology that was invented and patented by Wilhelm Cauer (1934, 1935, 1936). The Cauer filter was a significant advance (both theoretically and commercially) over the earlier filter designs of Campbell, Zobel, Johnson, Butterworth, and Chebychev. For example, for one commercially important set of specifications for telephones, a fifth-order elliptic filter matches the behavior of a 17th-order Butterworth filter or an eighth-order Chebychev filter. The fifth-order elliptic filter has one less component than the eighth-order Chebychev filter. As Van Valkenburg (1982, page 379) relates in connection with the history of the elliptic filter:

> Cauer first used his new theory in solving a filter problem for the German telephone industry. His new design achieved specifications with one less inductor than had ever been done before. The world first learned of the Cauer method not through scholarly publication but through a patent disclosure, which eventually reached the Bell Laboratories. Legend has it that the entire Mathematics Department of Bell Laboratories spent the next two weeks at the New York Public library studying elliptic functions. Cauer had studied mathematics under Hilbert at Goettingen, and so elliptic functions and their applications were familiar to him.

Genetic programming did not, of course, study mathematics under Hilbert or anybody else. Instead, the elliptic topology invented and patented by Cauer emerged from this run of genetic programming as a natural consequence of the problem's fitness measure and natural selection – not because the run was primed with domain knowledge about elliptic functions or filters or electrical circuitry. Genetic programming opportunistically *reinvented* the elliptic topology because necessity (fitness) is the mother of invention.

7. The Illogical Nature of Creativity and Evolution

Many computer scientists and mathematicians unquestioningly assume that every problem-solving technique must be logically sound, deterministic, logically consistent, and parsimonious. Accordingly, most conventional methods of artificial intelligence and machine learning are constructed so as to possess these characteristics. However, in

spite of this strong predisposition by computer scientists and mathematicians, the features of logic do not govern two of the most important types of complex problem solving processes, namely the invention process performed by creative humans and the evolutionary process occurring in nature.

A new idea that can be logically deduced from facts that are known in a field, using transformations that are known in a field, is not considered to be an invention. There must be what the patent law refers to as an "illogical step" (i.e., an unjustified step) to distinguish a putative invention from that which is readily deducible from that which is already known. Humans supply the critical ingredient of "illogic" to the invention process. Interestingly, everyday usage parallels the patent law concerning inventiveness: People who mechanically apply existing facts in well-known ways are summarily dismissed as being uncreative. Logical thinking is unquestionably useful for many purposes. It usually plays an important role in setting the stage for an invention. But, at the end of the day, logical thinking is the antithesis of invention and creativity.

Recalling his invention in 1927 of the negative feedback amplifier, Harold S. Black of Bell Laboratories (1977) said,

> Then came the morning of Tuesday, August 2, 1927, when the concept of the negative feedback amplifier came to me in a flash while I was crossing the Hudson River on the Lackawanna Ferry, on my way to work. For more than 50 years, I have pondered how and why the idea came, and I can't say any more today than I could that morning. All I know is that after several years of hard work on the problem, I suddenly realized that if I fed the amplifier output back to the input, in reverse phase, and kept the device from oscillating (singing, as we called it then), I would have exactly what I wanted: a means of canceling out the distortion of the output. I opened my morning newspaper and on a page of *The New York Times* I sketched a simple canonical diagram of a negative feedback amplifier plus the equations for the amplification with feedback.

Of course, inventors are not oblivious to logic and knowledge. They do not thrash around using blind random search. Black did not try to construct the negative feedback amplifier from neon bulbs or doorbells. Instead, "several years of hard work on the problem" set the stage and brought his thinking into the proximity of a solution. Then, at the critical moment, Black made his "illogical" leap. This unjustified leap constituted the invention.

The design of complex entities by the evolutionary process in nature is another important type of problem-solving that is not governed by logic. In nature, solutions to design problems are discovered by the probabilistic process of evolution and natural selection. There is nothing logical about this process. Indeed, inconsistent and contradictory alternatives abound. In fact, such genetic diversity is necessary for the evolutionary process to succeed. Significantly, the solutions evolved by evolution and natural selection almost always differ from those created by conventional methods of artificial intelligence and machine learning in one very important respect. Evolved solutions are not brittle; they are usually able to grapple with the perpetual novelty of real environments.

Similarly, genetic programming is not guided by the inference methods of formal logic in its search for a computer program to solve a given problem. When the goal is the automatic creation of computer programs, all of our experience has led us to conclude that the non-logical approach used in the invention process and in natural evolution are far more fruitful than the logic-driven and knowledge-based principles of conventional artificial intelligence and machine learning. In short, "logic considered harmful."

8. Conclusion

We illustrated genetic programming by applying it to a non-trivial problem, namely the synthesis of a design for a lowpass filter circuit. The results were competitive with human-produced solutions to the problem. The results exhibited creativity and inventiveness and correspond to four inventions that were patented between 1917 and 1936.

References

Aaserud, O. and Nielsen, I. Ring. 1995. Trends in current analog design: A panel debate. *Analog Integrated Circuits and Signal Processing*. 7(1) 5-9.

Andre, David and Koza, John R. 1996. Parallel genetic programming: A scalable implementation using the transputer architecture. In Angeline, P. J. and Kinnear, K. E. Jr. (editors). 1996. Advances in Genetic Programming 2. Cambridge: MIT Press.

Angeline, Peter J. and Kinnear, Kenneth E. Jr. (editors). 1996. *Advances in Genetic Programming 2*. Cambridge, MA: The MIT Press.

Back, Thomas. (editor). 1997. *Genetic Algorithms: Proceedings of the Seventh International Conference*. San Francisco, CA: Morgan Kaufmann.

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

Banzhaf, Wolfgang, Poli, Riccardo, Schoenauer, Marc, and Fogarty, Terence C. 1998. *Genetic Programming: First European Workshop. EuroGP'98. Paris, France, April 1998 Proceedings. Paris, France. April l998*. Lecture Notes in Computer Science. Volume 1391. Berlin, Germany: Springer-Verlag.

Black, Harold S. 1977. Inventing the negative feedback amplifier. *IEEE Spectrum*. December 1977. Pages 55 – 60.

Campbell, George A. 1917. *Electric Wave Filter*. Filed July 15, 1915. U. S. Patent 1,227,113. Issued May 22, 1917.

Cauer, Wilhelm. 1934. *Artificial Network*. U. S. Patent 1,958,742. Filed June 8, 1928 in Germany. Filed December 1, 1930 in United States. Issued May 15, 1934.

Cauer, Wilhelm. 1935. *Electric Wave Filter*. U. S. Patent 1,989,545. Filed June 8, 1928. Filed December 6, 1930 in United States. Issued January 29, 1935.

Cauer, Wilhelm. 1936. *Unsymmetrical Electric Wave Filter*. Filed November 10, 1932 in Germany. Filed November 23, 1933 in United States. Issued July 21, 1936.

Gen, Mitsuo and Cheng, Runwei. 1997. *Genetic Algorithms and Engineering Design*. New York: John Wiley and Sons.

Goldberg, David E. l989a. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

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

Johnson, Kenneth S. 1926. *Electric-Wave Transmission*. Filed March 9, 1923. U. S. Patent 1,611,916. Issued December 28, 1926.

Johnson, Walter C. 1950. *Transmission Lines and Networks*. New York: NY: McGraw-Hill.

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

Koza, John R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* 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. 1995. Evolving the architecture of a multi-part program in genetic programming using architecture-altering operations. In McDonnell, John R., Reynolds, Robert G., and Fogel, David B. (editors). 1995. *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*. Cambridge, MA: The MIT Press. Pages 695–717.

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 L. (editors). *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of Wisconsin, Madison, Wisconsin*. San Francisco, CA: Morgan Kaufmann.

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., Deb, Kalyanmoy, Dorigo, Marco, Fogel, David B., Garzon, Max, Iba, Hitoshi, and Riolo, Rick L. (editors). 1997. *Genetic Programming 1997: Proceedings of the Second Annual Conference* San Francisco, CA: Morgan Kaufmann.

Koza, John R., Goldberg, David E., Fogel, David B., and Riolo, Rick L. (editors). 1996. *Genetic Programming 1996: Proceedings of the First Annual Conference*. Cambridge, MA: The MIT Press.

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

Michalewicz, Zbigniew. 1996. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag. Third edition.

Mitchell, Melanie. 1996. *An Introduction to Genetic Algorithms*. Cambridge, MA: The MIT Press.

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

Quarles, Thomas, Newton, A. R., Pederson, D. O., and Sangiovanni-Vincentelli, A. 1994. *SPICE 3 Version 3F5 User's Manual*. Department of Electrical Engineering and Computer Science, University of California, Berkeley, CA. March 1994.

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

Samuel, Arthur L. 1983. AI: Where it has been and where it is going. *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*. Los Altos, CA: Morgan Kaufmann. Pages 1152 – 1157.

Spector, Lee, Langdon, William B., O'Reilly, Una-May, and Angeline, Peter (editors). 1999. *Advances in Genetic Programming 3*. Cambridge, MA: The MIT Press.

Van Valkenburg, M. E. 1982. *Analog Filter Design*. Fort Worth, TX: Harcourt Brace Jovanovich.

Williams, Arthur B. and Taylor, Fred J. 1995. *Electronic Filter Design Handbook*. Third Edition. New York, NY: McGraw-Hill.