John R. Koza and Forrest H Bennett III

The design of an electrical circuit entails creation of the circuit's topology, sizing, placement, and routing. Each of these tasks is either vexatious or computationally intractable. Design engineers typically perform these tasks sequentially– thus forcing the engineer to grapple with one vexatious or intractable problem after another. This chapter describes a holistic approach to the automatic creation of a circuit's topology, sizing, placement, and routing. This approach starts with a high-level statement of the requirements for the desired circuit and uses genetic programming to automatically and simultaneously create the circuit's topology, sizing, placement, and routing. The approach is illustrated with the problem of designing an analog lowpass filter circuit. The fitness measure for a candidate circuit considers the area of the fully laid-out circuit as well as whether the circuit passes or suppresses the appropriate frequencies. Genetic programming requires only about 1 1/2 orders of magnitude more computer time to create the circuit's topology, sizing, placement, and routing than to create the topology and sizing for this illustrative problem.

6.1 Introduction

The design process entails creation of a complex structure to satisfy user-defined requirements. Design is a major activity of practicing engineers. Since the design process typically involves tradeoffs between competing considerations, the end product is usually a satisfactory, as opposed to a perfect, design. The design process is usually viewed as requiring human intelligence. Consequently, the field of design is a source of challenging problems for automated techniques of machine learning and artificial intelligence.

The design process for electrical circuits begins with a high-level description of the circuit's desired behavior and entails creation of the circuit's topology, sizing, placement, and routing. The *topology* of a circuit involves specification of the gross number of components in the circuit, the type of each component (e.g., a capacitor), and a *netlist* specifying where each of a component's leads are to be connected. S*izing* involves specification of the values (typically numerical) of each component. *Placement* involves the assignment of each of the circuit's components to a particular physical location on a printed circuit board or silicon wafer. *Routing* involves the assignment of a particular physical location to the wires between the leads of the circuit's components.

All four of the above aspects of circuit design (topology, sizing, placement, and routing) are vexatious or computationally intractable for analog electrical circuits.

Specifically, until recently, there has been no general technique for automatically creating the topology and sizing for an entire analog electrical circuit from a high-level statement of the circuit's desired behavior. In describing the process of creating the topology and sizing of an analog circuit, Aaserud and Nielsen [1995] observed,

> 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.

The placement problem and the routing problem (for both analog and digital circuits) are examples of computationally intractable combinatorial optimization problems that require a computing effort that increases exponentially with problem size [Wong, Leong, and Liu 1988; Garey and Johnson 1979; Ullman 1984]. Since it is considered unlikely that efficient algorithms can be found for such problems, considerable effort has been expended on finding efficient approximate solutions to such problems [Wong, Leong, and Liu 1988; Cohn, Garrod, Rutenbar, and Carley 1994; Maziasz and Hayes 1992; Joobbani 1986; Sechen 1988]. The most important consideration in placement and routing is that there must be a wire connecting 100% of the nodes that must be connected. Wires cannot cross on a particular layer of a silicon chip or on a particular side of a printed circuit board. While it is possible to make connections (called *vias*) from another layer of the silicon chip or printed circuit board, the number of layers that are available and the type of components that can be located on each layer is strictly limited by the particular technology being used. For example, a printed circuit board or silicon chip may consist of one, two, or some other (small) number of layers. However, only one layer of currently used silicon chips can house components other than wires and at most two layers of a printed circuit boards can house anything other than wires. Once an acceptable placement and routing is discovered for a particular circuit, the next most important consideration is usually minimization of the area of the bounding rectangle of the laid-out circuit. Minimization of area has a substantial and direct economic impact in the manufacture of electronic circuitry. In addition, it is often desirable to minimize the length of wires connecting components and to minimize the number of vias.

As shown in figure 6.1, each of the above four vexatious or computationally intractable phases of circuit design (topology, sizing, placement, and routing) are currently tackled more or less sequentially.

The question arises as to whether all four aspects of circuit design can be combined into a unified automatic process. At first glance, it would appear that the four aspects are so different in character that such a combination would be impossible. Or, if such a combination is possible, it would appear that such a combination might require prohibitively large amounts of computer time. This chapter demonstrates that genetic programming can be used to automatically create the topology, sizing, placement, and routing of analog electrical circuits.

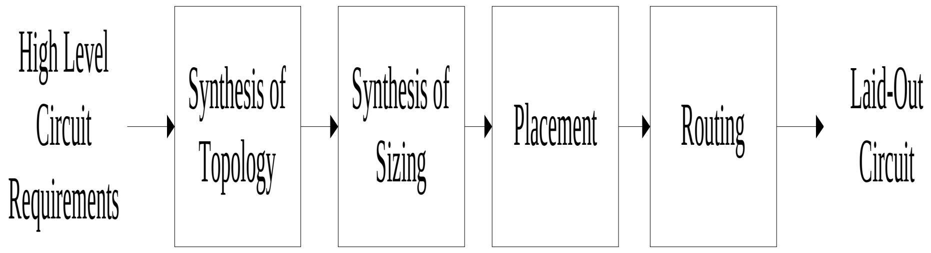

Figure 6.1
The process of designing a circuit includes creation of the circuit’s topology, sizing of the circuit’s components, placement of the components onto the substrate, and the routing of wires between the components.

Section 6.2 provides background. Section 6.3 presents our method. Section 6.4 describes an illustrative problem of designing an analog lowpass filter circuit. Section 6.5 details the preparatory steps required to apply our method to the illustrative problem. Section 6.6 presents the results. Section 6.7 discusses the amount of computer time required. Section 6.8 speculates on the role of genetic programming as an invention machine.

6.2 Automatic Creation of Circuit Topology and Sizing

There has been extensive previous work on the problem of automating various aspects of the design of electrical circuits using simulated annealing, artificial intelligence, and other optimization techniques (as detailed in Koza, Bennett, Andre, and Keane 1999), including genetic algorithms [Kruiskamp and Leenaerts 1995; Grimbleby 1995; Thompson 1996]. All of the existing techniques are limited to certain highly specialized types of circuits or address only one or two of the four aspects (topology, sizing, placement, and routing) of circuit design. Some of these techniques involve choosing pre-established alternative subcircuits for pre-established places within a pre-established overall circuit design. Many preexisting non-genetic techniques require the user to supply a reasonably good working circuit as a starting point (with the automated technique then merely adjusting the sizing of the components) or require repeated interactive intervention by the user.

Recently, a general technique using genetic programming has emerged for automatically creating the topology and sizing for an analog electrical circuit from a high-level statement of the circuit's desired behavior [Koza, Bennett, Andre, Keane, and Dunlap 1997; Koza, Bennett, Andre, and Keane 1999].

Genetic programming [Koza 1992; Koza and Rice 1992] is an extension of the genetic algorithm [Holland 1975] that automatically creates computer programs to solve problems. Genetic programming is also capable [Koza 1994a, 1994b] of evolving multi-part programs consisting of a main program and one or more reusable, parametrized, hierarchically-called automatically defined functions (subroutines). Architecture-altering operations [Koza 1995; Koza, Bennett, Andre, and Keane 1999] enable genetic programming to automatically determine the number of subroutines, the number of arguments that each possesses, and the nature of the hierarchical references, if any, among such automatically defined functions. They also enable genetic programming to automatically determine whether and how to use internal memory, iterations, and recursion in evolved programs. Additional information on current research in genetic programming can be found in Banzhaf, Nordin, Keller, and Francone 1998; Langdon 1998; Kinnear 1994; Angeline and Kinnear 1996; Koza, Goldberg, Fogel, and Riolo 1996; Koza, Deb, Dorigo, Fogel, Garzon, Iba, and Riolo 1997; Koza, Banzhaf, Chellapilla, Deb, Dorigo, Fogel, Garzon, Goldberg, Iba, and Riolo 1998; and Banzhaf, Poli, Schoenauer, and Fogarty 1998.

Numerous circuits have been designed using genetic programming, including lowpass, highpass, bandpass, bandstop, crossover, multiple bandpass, and asymmetric filters, amplifiers, computational circuits, a temperature-sensing circuit, a voltage reference circuit, a frequency-measuring circuit, and source identification circuits. The circuits evolved using genetic programming include eleven previously patented circuits.

Genetic programming can be applied to circuit design by establishing a mapping between the rooted, point-labeled acyclic graphs (trees) with ordered branches used in genetic programming and the specialized type of cyclic graphs germane to electrical circuits. The creative work of Kitano [1990] on using developmental genetic algorithms to evolve neural networks, the innovative work of Gruau [1992] on using genetic programming to evolve neural networks (cellular encoding), and the principles of developmental biology suggest a method for mapping trees into electrical circuits by means of a growth process that begins with a simple embryo. For electrical circuits, we use an embryo consisting of one (and sometime more) modifiable wires. The embryo is embedded into a test fixture consisting of fixed (hard-wired) components (such as a source resistor and a load resistor) and certain fixed wires that provide connectivity to the circuit's external inputs and outputs and that enable the behavior of the evolving circuitry to be evaluated. Until the modifiable wires are modified by the developmental process, the circuit produces only trivial output. An electrical circuit is developed by progressively applying the functions in a circuit-constructing program tree (in the population being bred by genetic programming) to the modifiable wires of the original embryo and to the modifiable components and modifiable wires created during the developmental process.

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 function-defining branches 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 (strong typing). 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). This constrained syntactic structure is preserved using structure-preserving crossover with point typing. For details, see Koza, Bennett, Andre, and Keane 1999.

The foregoing method for automatically creating circuit topology and sizing does not address the problem of automatically placing and routing of components and wires at particular physical locations on a printed circuit board or silicon wafer.

6.3 Method for Automatic Creation of Circuit Topology, Sizing, Placement, and Routing

This section describes how all four aspects of circuit design (topology, sizing, placement, and routing) can be combined into a unified approach using genetic programming.

6.3.1 The Initial Circuit

An electrical circuit is created in a developmental process by executing a circuit-constructing program tree that contains various component-creating, topology-modifying, and development-controlling functions. The starting point of the developmental process for transforming a program tree in the population into a fully developed electrical circuit is an initial circuit consisting of an embryo and a test fixture.

All the wires and components of the initial circuit are located at specified physical locations on a printed circuit board or silicon wafer. A board or wafer has a limited number of layers that are available for wires and a limited number of layers (usually one for a wafer and one or two for a board) that are available for both wires and components.

The embryo is an electrical substructure consisting of at least one modifiable wire. The test fixture is a substructure composed of nonmodifiable wires and nonmodifiable electrical components. The test fixture provides access to the circuit's external input(s) and permits testing of the circuit’s behavior and probing of the circuit's output. An embryo has one or more ports that enable it to be embedded into a test fixture.

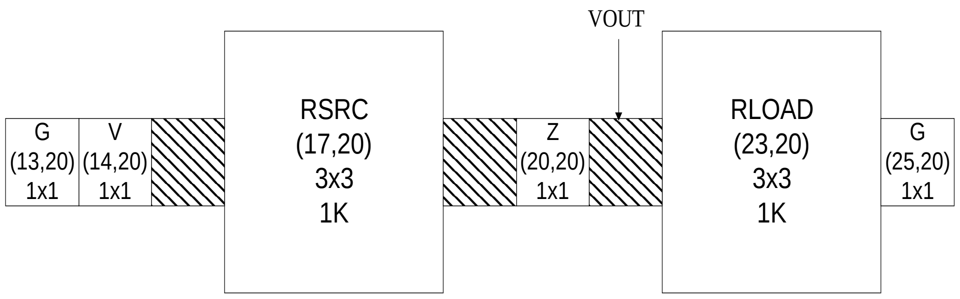

Figure 6.2
One-input, one-output initial circuit consisting of two ground points G, an incoming voltage signal V, a source resistor RSRC, a modifiable wire Z, a load resistor RLOAD, and three pieces of nonmodifiable wire. Each element is of a particular size and is located at a certain location. In addition, the two resistors have components values (1 kΩ each).

Figure 6.2 shows a one-input, one-output initial circuit located on one layer of a silicon wafer or printed circuit board. This initial circuit consists of the ground G (at the far left), the source V for the incoming voltage signal, a piece of nonmodifiable wire (hashed), a fixed 1 kilo-Ohm (kΩ) source resistor RSRC, another piece of nonmodifiable wire, a piece of modifiable wire Z, another piece of nonmodifiable wire, a fixed 1 kΩ load resistor RLOAD, and the ground G (at the far right). The output probe point VOUT is the place where the circuit's output voltage is measured.

Each element of a circuit resides at particular physical location on the circuit's two-dimensional substrate and occupies a particular amount of space. The two resistors each occupy a 3×3 area while all the other elements of this initial circuit each occupy a 1×1 area. The location of the 1×1 ground point G at the far left of the figure is (13, 20); the location of the incoming signal source V is (14,20); the location of the center of the 3×3 source resistor RSRC is (17, 20), and the location of the 1×1 modifiable wire Z is (20, 20). As will be seen momentarily, circuit elements typically change locations numerous times during the developmental process.

The embryo of this initial circuit consists of the single 1×1 piece of modifiable wire Z. All development originates from this embryo. The remaining elements (all fixed) of the initial circuit constitute the test fixture.

6.3.2 Circuit-Constructing Functions

A program tree may contain component-creating functions, topology-modifying functions, and development-controlling functions. Each of these three types of functions is associated with a modifiable wire or modifiable component in the developing circuit. The construction-continuing subtree(s), if any, of these functions point to a successor function or terminal in the circuit-constructing program tree.

A program tree may also contain arithmetic functions and constants. 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 -1.0 to +1.0). 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 details of this process are the same as used in Koza, Bennett, Andre, and Keane 1999.

6.3.2.1 Component-Creating Functions

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

The two-argument capacitor-creating `LAYOUT-C` function inserts a capacitor into a developing circuit in lieu of a modifiable wire (or modifiable component). Figure 6.3 shows a partial circuit containing four capacitors (C2, C3, C4, and C5) and a modifiable wire Z0. Each capacitor occupies a 3×3 area. Each piece of wire occupies a $1 \times n$ or an $n \times 1$ area. The modifiable wire Z0 occupies a 1×1 area and is located at (20, 20).

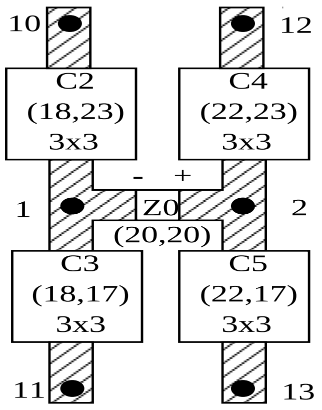

Figure 6.3
Partial circuit with a 1×1 piece of modifiable wire Z0 at location (20, 20) and four capacitors.

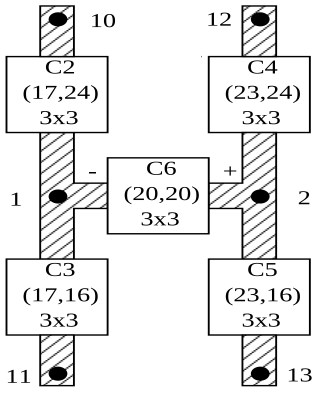

Figure 6.4
The application of the LAYOUT-C function to the modifiable wire Z0 of figure 6.3 causes a 3×3 capacitor C6 to be inserted at location (20, 20). The insertion of the new capacitor C6 forces a change in location for the other capacitors in the figure.

Figure 6.4 shows the result of applying the two-argument capacitor-creating LAYOUT-C function to the modifiable wire Z0 of figure 6.3. The newly created capacitor C6 occupies a 3×3 area and is centered at location (20, 20). The newly created component is larger in both directions than the 1×1 piece of modifiable wire that it replaces. Thus, its insertion affects the locations of all preexisting components in the developing circuit. In particular, preexisting capacitor C2 is pushed north and west by one unit thereby relocating it from (18, 23) to (17, 24). Similarly, preexisting capacitor C5 is pushed south and east by one unit thereby relocating it from (22, 17) to (23, 16). In our actual implementation, all adjustments in location are made after the completion of the entire developmental process; however, we will explain each circuit-constructing function as if the required adjustment is made at the time that the function is executed. The first argument of the capacitor-creating function is an arithmetic-performing subtree that specifies the value of the newly created capacitor in micro-Farads. The second argument of the capacitor-creating function is the construction-continuing subtree. The newly created capacitor C6 remains subject to subsequent modification.

Similarly, the two-argument inductor-creating LAYOUT-L function causes an inductor to be inserted into a developing circuit in lieu of a modifiable wire (or other modifiable component). The inductors in this chapter each occupy a 3×3 area; however, different components may, in general, have different dimensions. The value of the new inductor in micro-Henrys is specified by the arithmetic-performing subtree (the function's first argument). The function's second argument is the construction-continuing subtree.

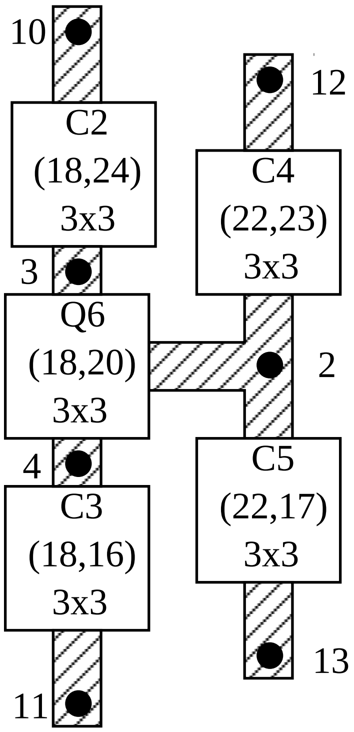

Figure 6.5
The application of `LAYOUT-CMOS-TRANSISTOR` function to the modifiable wire `Z0` of figure 6.3 causes a three-leaded transistor `Q6` occupying a 3×3 area to be inserted at location (18, 20).

Three-leaded components, such as transistors, may also be inserted into a developing circuit. Figure 5 shows the result of applying the one-argument transistor-creating `LAYOUT-CMOS-TRANSISTOR` function to the modifiable wire `Z0` of figure 3. The newly created transistor `Q6` occupies a 3×3 area and is located at (18, 20). The newly created component is larger than that which it replaces thereby affecting the locations of two preexisting components (`C2` and `C3`) in the developing circuit. Specifically, preexisting capacitor `C2` is pushed north by one unit thereby relocating it from (18, 23) to (18, 24). Similarly, preexisting capacitor `C3` is pushed south by one unit thereby relocating it from (18, 17) to (18, 16). The argument of the transistor-creating function is an arithmetic-performing subtree that specifies the transistor's width. The newly created transistor `Q6` is not subject to subsequent modification (and hence this function possesses only one argument and takes no construction-continuing subtree).

6.3.2.2 Topology-Modifying Functions

Each topology-modifying function modifies the topology of the developing circuit.

The two-argument SERIES-LAYOUT function creates a series composition consisting of the modifiable wire or modifiable component with which the function is associated and a copy of it. The function also creates two new nodes. Figure 6.6 shows a partial circuit containing six capacitors (C2, C3, C4, C5, C6, and C7). Figure 6.7 shows the result of applying the SERIES-LAYOUT function to modifiable capacitor C6 located at (20, 20) of figure 6.6. The SERIES-LAYOUT function creates a new capacitor C8 occupying a 3 × 3 area. The newly created capacitor C8 has the same values as modifiable capacitor C6. The function does not move the preexisting component (C6). Instead, room is made for the newly created capacitor C8 in the direction of a specified one of the two leads (the positive lead) of the preexisting component. Thus, C8 is located at (22, 20) to the east of C6 in this example. The addition of the four units horizontally to accommodate C8 affects the horizontal (but not vertical) location of the four preexisting capacitors. For example, preexisting capacitor C2 is pushed west by two units thereby relocating it from (17, 24) to (15, 24). Similarly, preexisting capacitor C5 is pushed east by two units thereby relocating it from (23, 16) to (25, 16). The addition of the four units horizontally to accommodate C8 also affects other parts of the developing circuit. For example, the wires to the east and west of preexisting capacitor C7 are lengthened (by two units each) to reflect the addition of the four horizontal units associated with the creation of C8. Both arguments of the SERIES-LAYOUT function are construction-continuing subtrees, so that both C6 and C8 remain subject to subsequent modification. New node 3 is located between preexisting capacitor C6 and new capacitor C8 at the original location(20, 20) of preexisting capacitor C6.

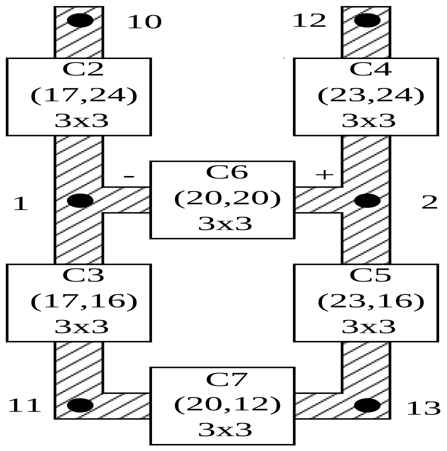

Figure 6.6
Partial circuit with a 3 × 3 modifiable capacitor C6 at location (20, 20) with five nearby capacitors.

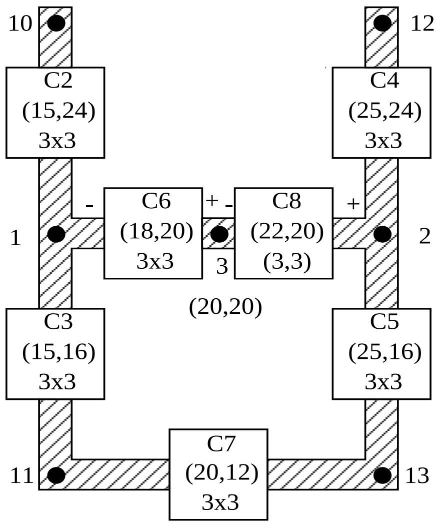

Figure 6.7
The application of the `SERIES-LAYOUT` function to the modifiable capacitor C6 of figure 6.6 causes a new 3×3 capacitor C8 to be inserted at location (22, 20) in series with C6.

Each of the two functions in the `PARALLEL-LAYOUT` family of four-argument functions creates a parallel composition consisting of two new modifiable wires, the preexisting modifiable wire or modifiable component with which the function is associated, and a copy of the modifiable wire or modifiable component. Each function also creates two new nodes. Figure 6.8 shows the result of applying the `PARALLEL-LAYOUT-LEFT` function to modifiable capacitor C6 located at (20, 20) of figure 6.4. The function does not change the location of the modifiable component or modifiable wire with which the function is associated. The function creates a new capacitor C7 occupying a 3×3 area with the same values as modifiable capacitor C6. The function positions the new capacitor C7 to the left of C6 (looking from the negative to positive lead of the modifiable component or modifiable wire with which the function is associated). The function does not affect the location of preexisting circuitry to the right of C6 (i.e., C3 and C5). The function inserts a new 1×1 modifiable wire Z9 at (23, 22) to the left of C6, a new 1×1 piece of wire between preexisting node 2 and new modifiable wire Z9, and a new 1×1 piece of wire between new node 4 and Z9. The function inserts a new 1×1 modifiable wire Z8 at (17, 22) to the left of C6, a new 1×1 piece of wire between preexisting node 1 and new modifiable wire Z8, and a new 1×1 piece of wire between new node 3 and Z8. The new capacitor C7 is located at (20, 24).

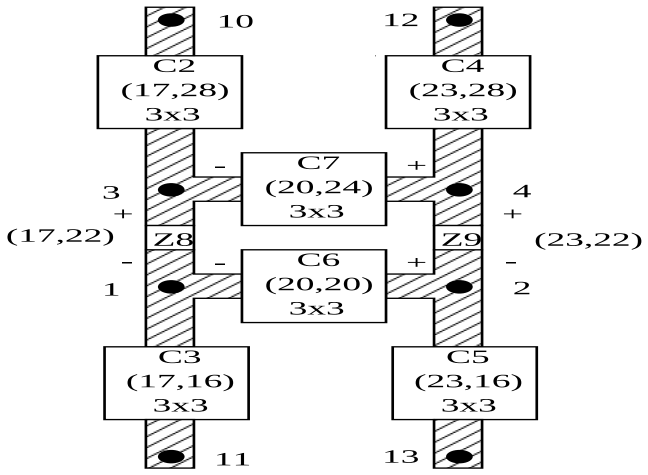

Figure 6.8
The application of the PARALLEL-LAYOUT-LEFT function to the modifiable capacitor C6 of figure 6.4 causes a new 3 × 3 capacitor C7 to be inserted at location (20, 24) in parallel with C6.

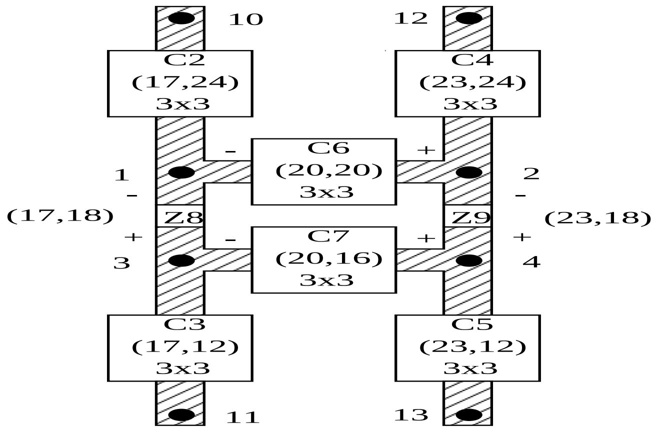

Figure 6.9
The application of the PARALLEL-LAYOUT-RIGHT function to the modifiable capacitor C6 of figure 6.4 causes a new 3 × 3 capacitor C7 to be inserted at location (20, 16) in parallel with C6.

The function also relocates the preexisting circuitry to the left of C6. Specifically, preexisting capacitor C2 is pushed to the north from (17, 24) to (17, 28) and preexisting capacitor C4 is pushed to the north from (23, 24) to (23, 28).

Figure 6.9 shows the result of applying the PARALLEL-LAYOUT-RIGHT function to modifiable capacitor C6 located at (20, 20) of figure 6.4. This function operates in a manner similar to the PARALLEL-LAYOUT-LEFT function, except that new capacitor C7 is located to the right of C6 and the location of preexisting circuitry to the right of C6 is pushed south.

The one-argument polarity-reversing FLIP function reverses the polarity of the modifiable component or modifiable wire with which the function is associated.

All of the foregoing circuit-constructing functions operate in a plane. However, most practical circuits are not planar. Vias provide a way to connect distant points of a circuit. Each of the four functions in the VIA-TO-GROUND-LAYOUT family of three-argument functions creates a T-shaped composition consisting of the modifiable wire or modifiable component with which the function is associated, a copy of it, two new modifiable wires, and a via to ground. The function also creates two new nodes.

Figure 6.10 shows the result of applying the VIA-TO-GROUND-NEG-LEFT-LAYOUT function to modifiable capacitor C6 located at (20, 20) of figure 6.4. The function creates a new node 3 at the location (20, 20) of the modifiable component or modifiable wire with which the function is associated and also creates a new 2 × 1 area at the negative end of the modifiable component or modifiable wire with which the function is associated and a new 4 × 1 area to the left.

The new 4 × 1 area consists of a new 1 × 1 piece of wire perpendicular and to the left of the modifiable component or modifiable wire with which the function is associated (facing from the negative to positive lead of the modifiable component or modifiable wire with which the function is associated), a new 1 × 1 modifiable wire Z9 at location (20, 22) beyond the new 1 × 1 piece of wire, a new node 4 at (20, 23) beyond Z9 and the new 1 × 1 piece, and a via to ground G at (20, 24) beyond node 4, Z9, and the new 1 × 1 piece.

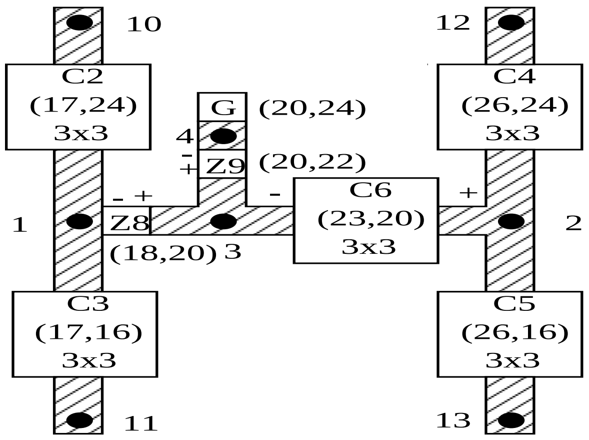

Figure 6.10
The application of the VIA-TO-GROUND-NEG-LEFT-LAYOUT function to the modifiable capacitor C6 of figure 6.4 causes a new 1 × 1 connection to ground G to be inserted at location (20, 24).

The new 2 × 1 area consists of a new 1 × 1 piece of wire at (19, 20) at the negative lead of the modifiable component or modifiable wire with which the function is associated and a new modifiable wire Z8 at (18,20). Since the VIA-TO-GROUND-NEG-LEFT-LAYOUT function creates a new 2 × 1 area at the negative end of preexisting capacitor C6 and a new 1 × 1 node 3, capacitor C6 is pushed to the east by three units so that C6 becomes centered at (23, 20). Consequently, preexisting capacitor C4 is pushed east from (23, 24) to (26, 24) and preexisting capacitor C5 is pushed east from (23, 16) to (26, 16).

The three other members of this family of functions are named to reflect the fact that they create the new 2 × 1 area at the positive (instead of negative) end of the modifiable component or modifiable wire with which the function is associated and that they create the new 4 × 1 area to the right (instead of left).

If desired, similar families of three-argument functions can be defined to allow direct connections to a positive power supply or a negative power supply.

In addition, if desired, numbered vias can be created to provide connectivity between two different parts of a circuit. A distinct four-member family of three-argument functions is used for each via. For example, VIA-2-NEG-LEFT-LAYOUT makes connection with an imaginary layer numbered 2 of the imaginary multi-layered silicon wafer (or multi-layered printed circuit board) on which the circuit resides.

The initial circuit complies with the requirement that wires cannot cross on a particular layer of a silicon chip or on a particular side of a printed circuit board and with the requirement that there must be a wire connecting 100% of the leads. Each of the component-creating and topology-modifying functions preserves compliance with these two mandatory requirements for successful placement and routing, so any sequence of such functions yields a fully laid-out circuit that complies with these two requirements.

6.3.2.3 Development-Controlling Functions

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

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 the developmental process on the particular path on which it appears.

6.3.3 The Developmental Process

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 embryonic circuit 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 embryonic circuit and its successors until a fully developed circuit eventually emerges. The functions are applied in a breadth-first order.

6.4 Statement of the Illustrative Problem

The method will be illustrated on the problem of creating the topology, sizing, placing, and routing for a lowpass filter. A simple *filter* is a one-input, one-output circuit that 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*). The desired lowpass filter is to pass all frequencies below 1,000 Hertz (Hz) and to suppress all frequencies above 2,000 Hz. The circuit is to be constructed on a two-sided printed circuit board whose top side contains discrete components (capacitors and inductors) that are connected by perpendicularly intersecting metallic wires and whose bottom side is devoted to connections to ground.

6.5 Preparatory Steps

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

6.5.1 Initial Circuit

The one-input, one-output initial circuit consisting of an embryo with one modifiable wire and a test fixture as shown in figure 6.2 is suitable for this problem.

6.5.2 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 has one result-producing branch. Neither automatically defined functions nor architecture-altering operations are used in this chapter.

6.5.3 Function and Terminal Sets

The function set, $\mathcal{F}_{\text{CCS}}$, for each construction-continuing subtree is

$\mathcal{F}_{\text{CCS}}$ = {`C-LAYOUT`, `L-LAYOUT`, `SERIES-LAYOUT`, `PARALLEL-LAYOUT-LEFT`, `PARALLEL-LAYOUT-RIGHT`, `FLIP`, `NOOP`, `VIA-TO-GROUND-NEG-LEFT-LAYOUT`, `VIA-TO-GROUND-NEG-RIGHT-LAYOUT`, `VIA-TO-GROUND-POS-LEFT-LAYOUT`, `VIA-TO-GROUND-POS-RIGHT-LAYOUT`}.

These functions possess 2, 2, 1, 4, 4, 1, 1, 3, 3, 3, and 3 arguments, respectively.

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

$$\mathcal{T}_{\text{ccs}} = \{\texttt{END}\}.$$

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

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

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

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

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

6.5.4 Fitness Measure

The high-level statement of requirements for the desired circuit is translated into a well-defined measurable quantity (the fitness measure) that is used by genetic programming to guide the evolutionary search for a satisfactory solution.

The fitness measure for this problem is multiobjective. It is expressed in terms of minimization of area of the bounding rectangle around the fully laid-out circuit, suppression of frequencies in the stopband of the desired filter, and passage at full power of frequencies in the passband of the desired filter.

The evaluation of each individual circuit-constructing program tree in the population begins with its execution. This execution progressively applies the functions in the program tree to the embryo of the circuit, thereby creating a fully developed (and fully laid out) 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 SPICE (Simulation Program with Integrated Circuit Emphasis) simulation program [Quarles, Newton, Pederson, and Sangiovanni-Vincentelli 1994]. SPICE is a 217,000-line industrial-strength electrical circuit simulator written over a period of several decades at the University of California at Berkeley. The SPICE simulator (and its many spinoffs) dominates the field of simulation for general-purpose electrical circuits. There are several hundred thousand copies in daily use by practicing electrical engineers throughout the world [Perry 1998]. Our modified version of SPICE (described in Koza, Bennett, Andre, and Keane 1999) then determines the circuit's behavior. Since the high-level statement of the behavior for the desired circuit is expressed (in part) in terms of frequencies, the output voltage VOUT is measured in the frequency domain. SPICE performs an AC small signal analysis and report 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 (sampled frequencies).

Since the developmental process for creating the fully developed circuit in this problem includes the actual physical placement of components and the actual physical routing of wires between the components, the area of the bounding rectangle for the fully developed circuit is easily computed.

The desired lowpass filter has a passband below 1,000 Hz and a stopband above 2,000 Hz. Each circuit is driven by an incoming AC voltage source with a 2 volt amplitude. Each circuit is tested by a test fixture containing a 1 kΩ source (internal) resistor RSRC and a kΩ load resistor RLOAD. There should be a sharp drop-off from 1 to 0 Volts in the transitional (“don't care”) region between 1,000 Hz and 2,000 Hz.

The *attenuation* of the filter is defined in terms of the output signal relative to the reference voltage (half of 2 volts here). A *decibel* is a unitless measure of relative voltage that is defined as 20 times the common (base 10) logarithm of the ratio between the voltage at a particular probe point and a reference voltage (1 volt).

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 filer. 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.

A fifth-order *elliptic* (*Cauer*) *filter* with a modular angle Θ of 30 degrees (i.e., the arcsin of the ratio of the boundaries of the passband and stopband) and a reflection coefficient ρ of 24.3% can satisfy the above design goals [Williams and Taylor 1995].

Fitness is defined for this problem using one or two of the following terms.

The first term is the sum, over the 101 fitness cases (sampled frequencies), 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 (0 or 1 volts). Specifically, this term 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 second term is the area of the bounding rectangle for the fully developed circuit divided by 100,000 square units of area.

The term of the fitness measure involving the filter's frequency response is designed to not penalize ideal voltage values, to slightly penalize every acceptable voltage deviation, and to heavily penalize every unacceptable voltage 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 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.

The term involving the bounding rectangle is much smaller than the term involving the filter's frequency response until a circuit scores 101 (or near 101) hits. For individuals not scoring the maximum number (101) of hits, fitness is the sum of the two terms. For individuals scoring the maximum number of hits, fitness is only the area-based term. In any event, the smaller the value of fitness, the better. A value of zero is unattainable because no actual circuit occupies zero area and because no actual analog filter can perfectly satisfy the problem's requirements in the frequency domain.

The SPICE simulator is remarkably robust; however, it cannot simulate every conceivable circuit. In particular, many circuits that are randomly created for the initial population of a run of genetic programming and many circuits that are created by the crossover and mutation operations in later generations are so pathological that SPICE cannot simulate them. These circuits receive a high penalty value of fitness (10^8) and become the worst-of-generation programs for each generation.

6.5.5 Control Parameters

The population size, *M*, is 1,120,000. A maximum size of 600 points (functions and terminals) was established for each circuit-constructing program tree. Our usual control parameters are used [Koza, Bennett, Andre, and Keane 1999, Appendix D].

6.5.6 Termination Criterion and Results Designation

The maximum number of generations, *G*, is set to an arbitrary large number and the run was manually monitored and manually terminated when the fitness of the best-of-generation individual appeared to have reached a plateau.

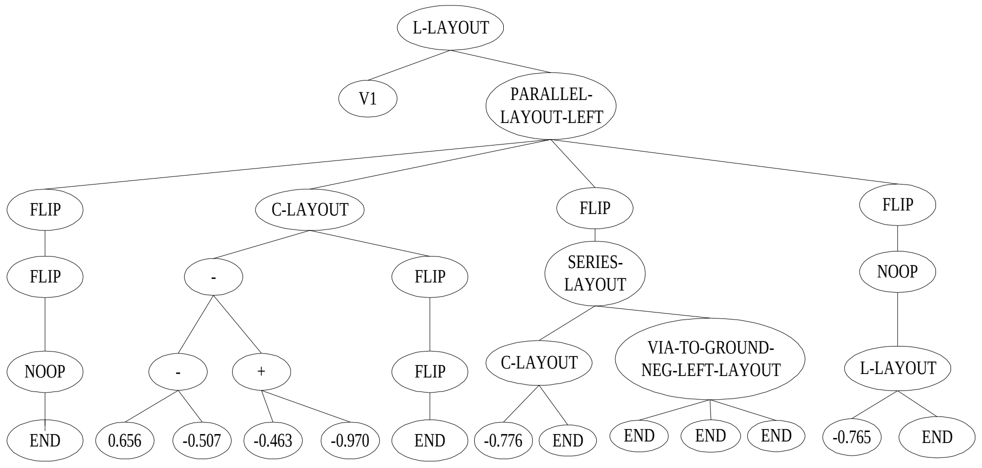

Figure 6.11
Best individual circuit-constructing program tree of generation 0. The program tree is a composition of component-creating, topology-modifying, and development-controlling functions.

6.5.7 Implementation on Parallel Computer

This problem was run on a home-built Beowulf-style [Sterling, Salmon, Becker, and Savarese 1999] parallel cluster computer system consisting of 56 processing nodes (each containing a 533-MHz DEC Alpha microprocessor and 64 megabytes of RAM) arranged in a two-dimensional 7×8 toroidal mesh. The system has a DEC Alpha computer as host. The processing nodes are connected with a 100 megabit-per-second Ethernet. The processing nodes and the host use the Linux operating system. The distributed genetic algorithm with unsynchronized generations and semi-isolated subpopulations [Andre and Koza 1996] was used. There was a subpopulation size of Q = 20,000 at each of D = 56 demes. 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.

6.6 Results

A run of genetic programming starts with the random creation of an initial population of circuit-constructing program trees composed of the functions and terminals identified in the previous section. The initial random population (generation 0) of a run of genetic programming is a blind random search of the search space of the problem.

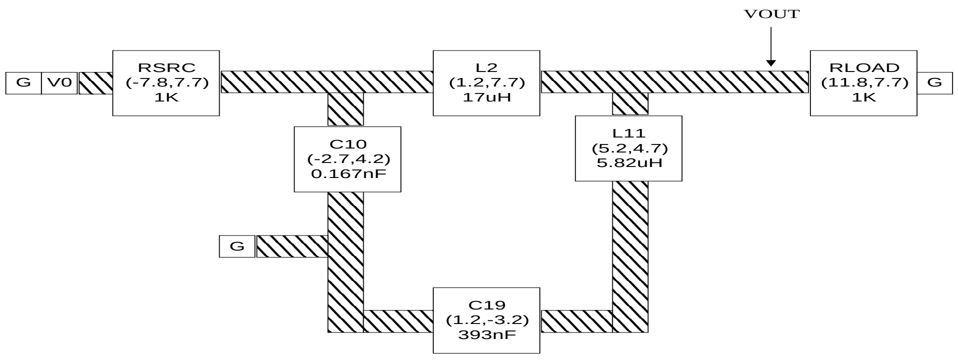

Figure 6.12
Best circuit of generation 0 containing two inductors (L2 and L11) and two capacitors (C10 and C19) in addition to all of the nonmodifiable elements of the original test fixture of the initial circuit of figure 6.2.

Generation 0 provides a baseline for comparing the results of subsequent generations. About a quarter of the circuits in generation 0 are so pathological that the SPICE simulator cannot simulate them (compared to 2% to 5% for later generations). We use the best circuit-constructing program tree (figure 6.11) from generation 0 to illustrate the developmental process used to convert a tree into a fully developed and fully laid-out electrical circuit.

This circuit-constructing program tree is shown below in the style of a LISP S-expression:

```
(L-LAYOUT
  V1
  (PARALLEL-LAYOUT-LEFT
    (FLIP (FLIP (NOOP END)))
    (C-LAYOUT
       (– (– 0.656 –0.507) (+ -0.463 –0.970))
       (FLIP (FLIP END)))
    (FLIP
       (SERIES-LAYOUT
          (C-LAYOUT –0.776 END)
          (VIA-TO-GROUND-NEG-LEFT-LAYOUT END END END)))
    (FLIP
       (NOOP (L-LAYOUT 0.765 END)))))
```

The program begins with a two-argument inductor-creating L-LAYOUT function. The value of the new inductor is established by the first argument of this L-LAYOUT function. Since this particular argument is a large arithmetic-performing subtree composed of addition, subtraction, and floating-point random constants, it is abbreviated and labeled V1 in figure 6.11 and this program. The second argument (construction-continuing subtree) of this L-LAYOUT function is a PARALLEL-LAYOUT-LEFT function.

The first argument of the four-argument PARALLEL-LAYOUT-LEFT function executes two one-argument polarity-reversing FLIP functions and one one-argument NOOP ("No Operation") function before reaching a development-terminating zero-argument END function.

The second argument of the PARALLEL-LAYOUT-LEFT function executes a capacitor-creating two-argument C-LAYOUT function whose value is established by a seven-point arithmetic-performing subtree (shown in its entirety in the figure) and whose construction-continuing subtree contains two polarity-reversing FLIP functions and one NOOP function.

The third argument of the PARALLEL-LAYOUT-LEFT function is a one-argument FLIP function whose construction-continuing subtree consists of a two-argument SERIES-LAYOUT function. The first construction-continuing subtree of the SERIES-LAYOUT function executes a second capacitor-creating C-LAYOUT function. The value of the second capacitor is established (in the manner described earlier) by the one-point arithmetic-performing subtree consisting of the floating-point random constant -0.7763983. The second construction-continuing subtree of the SERIES-LAYOUT function executes a three-argument VIA-TO-GROUND-NEG-LEFT-LAYOUT function.

The fourth argument of the PARALLEL-LAYOUT-LEFT function is a FLIP function whose construction-continuing subtree executes a one-argument NOOP function which, in turn, causes execution of a second two-argument inductor-creating L-LAYOUT function. The value of the second inductor is established by the one-point arithmetic-performing subtree consisting of the floating-point constant 0.7648563.

When this circuit-constructing program tree for the best-of-generation circuit of generation 0 is executed, it yields a fully laid-out circuit (figure 6.12) with two created inductors (L2 and L11) and two created capacitors (C10 and C19). The four components, the connecting wires, the ground points, and the source of the incoming signal are all assigned a precise physical location in this fully laid-out circuit. Notice that all of the nonmodifiable elements of the original test fixture of the initial circuit of figure 6.2 (including, in particular, the incoming signal V, the source resistor RSRC, the load resistor RLOAD) survive in this fully developed and fully laid-out circuit (albeit in different locations). In addition, notice that this fully laid-out circuit complies with the requirement that wires cannot cross on a particular layer of a silicon chip or on a particular side of a printed circuit board and with the requirement that there must be a wire connecting 100% of the leads.

This best-of-generation circuit from generation 0 scores 53 hits and has a fitness of 57.961037. The incoming signal V first passes through source resistor RSRC located at position (-7.8, 7.7) at the top left of the figure. The signal passes into inductor L2 located at position (1.2, 7.7) to load resistor RLOAD located at position (11.8, 7.7) and to ground G (at the top right of the figure). In addition, capacitor C10 located at (-2.7, 4.2) is connected to the ground G (in the middle left of the figure). Also, the series composition of inductor L11 and capacitor C19 is also connected to the same ground point G in the middle left of the figure. The area-based term of the fitness measure is only 0.003710 at this at this early stage of the run.

Figure 6.13 shows the behavior in the frequency domain of the best circuit of generation 0. The horizontal axis represents five decades of frequency on a logarithmic scale. The vertical axis represents voltage linearly. As can be seen, the behavior of this circuit bears very little resemblance to the desired lowpass filter. The circuit delivers a full volt for all frequencies up to about 50 Hz. The output approximates 0 volts only for a few frequencies near 100,000 Hz. There is a very large and leisurely transition region.

Both the average fitness of all individuals in the population and the fitness of the best individual in the population improve over successive generations. The best circuit of generation 8 (figure 6.16) scores 82 hits and has a fitness of 9.731077 (of which only 0.008138 is contributed by the area-based term of the fitness measure). The result-producing branch of its circuit-constructing program tree contains 165 points. The circuit has five inductors and three capacitors. The incoming signal V passes through source resistor RSRC (at the bottom left of the figure) and is fed into a parallel-series composition of inductors L13, L2, and L12 (which are together electrically equivalent to one inductor). In a lowpass filter, the capacitors that are connected to ground are called *shunts* and the inductors positioned in series (along the bottom of this figure) between the source resistor and load resistor are called *series* inductors [Williams and Taylor 1995]. When all the parallel and series compositions of like components are combined, this circuit is equivalent to a first series inductor (the L13, L2, and L12 combination), a first capacitive shunt (C18), a second series inductor (L11), a second capacitive shunt (the C16 and C19 combination), and a third series inductor (L16). The series inductors and capacitive shunts of a lowpass filter are typically drawn on paper to resemble a ladder. The capacitive shunts correspond to the rungs of the ladder. The series inductors correspond to one side of the ladder. The second side of the ladder is a common ground wire to which each capacitive shunt is connected. This circuit is a two-rung ladder. The best-of-run circuit that emerges below in generation 138 is a four-rung ladder.

Figure 6.14 shows the behavior in the frequency domain of the best circuit of generation 8. This circuit contains five inductors and three capacitors. As can be seen, the behavior of this circuit bears some resemblance to the desired lowpass filter. It delivers 1 volt for all frequencies up to about 900 Hz and about 0 volts for all frequencies above 10,000 Hz. However, there is a very leisurely transition from the frequencies that are passed to the frequencies that suppressed.

The first circuit scoring 101 hits (out of 101) appears in generation 25 (figure 6.17). It has a fitness of 0.01775. The result-producing branch of its circuit-constructing program tree contains 548 points. The circuit has five capacitors and 11 inductors. The incoming signal V passes through source resistor RSRC (at the bottom left of the figure) and is fed into a composition of inductors along the straight line connecting the incoming signal V to the load resistor RLOAD (at the bottom right of the figure). Along the way, there is a series composition of L2, L12, the L11/L23 parallel combination, L10, L26, the L9/L33 parallel combination, L32, and L31. There are four shunts to ground G. Three of these shunts consist of one capacitor each (C19, C29, and C40). The fourth shunt consists of inductor L16 and the C17/C19 parallel combination. As can be seen, this figure 6.occupies a considerable area.

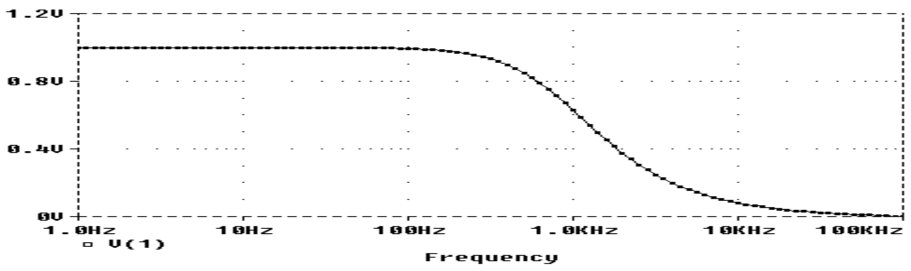

Figure 6.13
Frequency domain behavior of best circuit from generation 0.

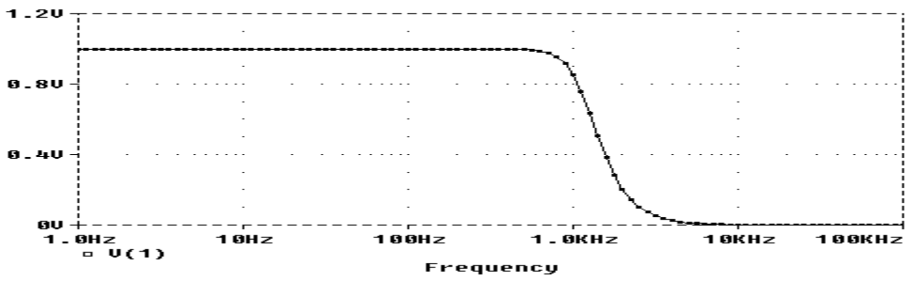

Figure 6.14
Frequency domain behavior of best circuit from generation 8.

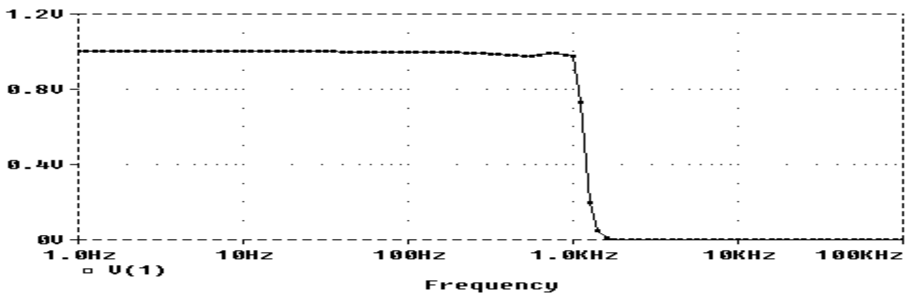

Figure 6.15
Frequency domain behavior of 100%-compliant circuit from generation 25.

Figure 6.15 shows the behavior in the frequency domain of the 100%-compliant 16-component circuit of generation 25. As can be seen, this circuit delivers approximately a full volt for all frequencies up to about 1,000 Hz and about 0 volts for all frequencies above 2,000 Hz. The drop-off in the transition region between the passband

and stopband is far sharper than that of the best-of-generation circuits from generations 0 and 8. This circuit occupies an area of 1775.2. This circuit is indeed a lowpass filter.

In generation 30, the best-of-generation circuit (figure 6.18) has a fitness of 0.00950. It also scores 101 hits. This circuit has 10 inductors and five capacitors. This 100%-compliant 15-component circuit (occupying an area of 950.3) occupies only 54% of the area of the 100%-compliant 16-component circuit from generation 25.

The best-of-run circuit (figure 6.19) appears in generation 138. It has a near-zero fitness of 0.00359 (more than five orders of magnitude better than the fitness of the best circuit of generation 0). The result-producing branch of its circuit-constructing program tree contains 463 points. This circuit has four inductors and four capacitors (half as many components as the best circuit from generation 25). The circuit has a compact layout. There are no series or parallel compositions of capacitors or inductors that redundantly connect the same two points of the circuit. The topology of this best-of-run circuit is a four-rung ladder.

This best-of-run circuit occupies an area of 359.4 (only 20% of the area of the 100%-compliant best circuit from generation 25 in figure 6.17). Note that, for reasons of space, figures 6.17, 6.18, and 6.19 are not drawn to scale.

Table 6.1 shows the number of capacitors, the number of inductors, the number of capacitive or inductive shunts to ground, the area in terms of the number of square units of the bounding rectangle, the frequency-based term of fitness for the 101 fitness cases, and the total fitness for the three best-of-generation circuits (each scoring 101 hits) from generations 25, 30, and 138.

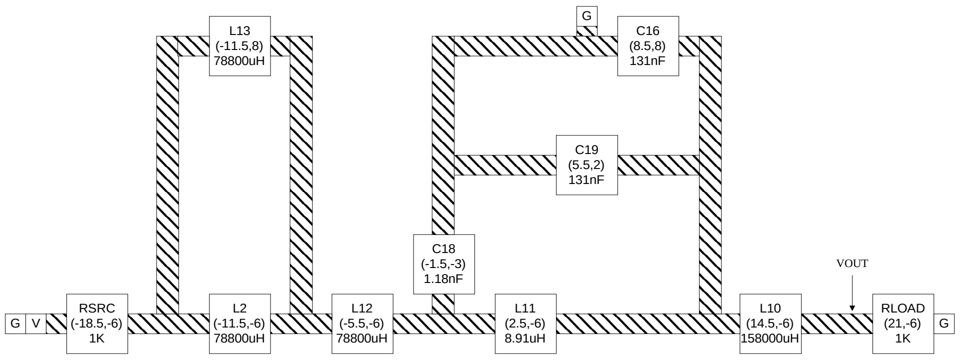

Figure 6.16
Best circuit of generation 8 containing five inductors and three capacitors.

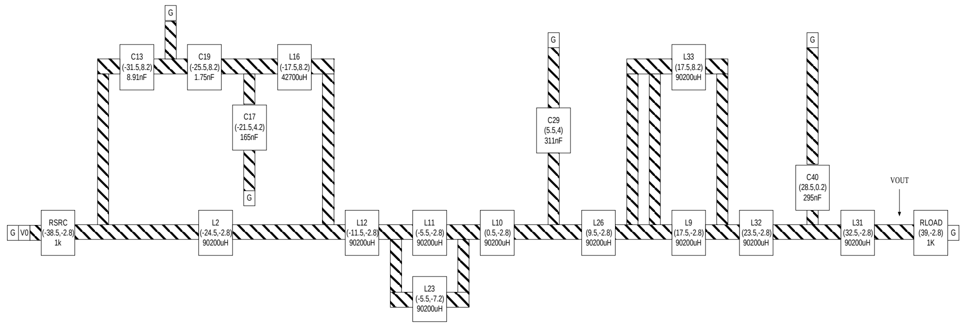

Figure 6.17
100%-compliant best circuit of generation 25 containing five capacitors and 11 inductors (total of 16 components) occupying an area of 1775.2.

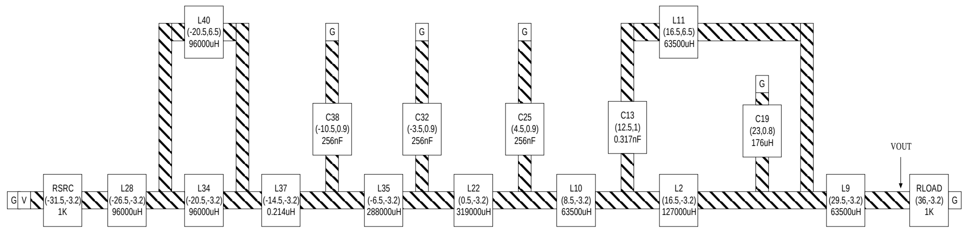

Figure 6.18
100%-compliant best circuit of generation 30 containing 10 inductors and five capacitors (total of 15 components) occupying an area of 950.3.

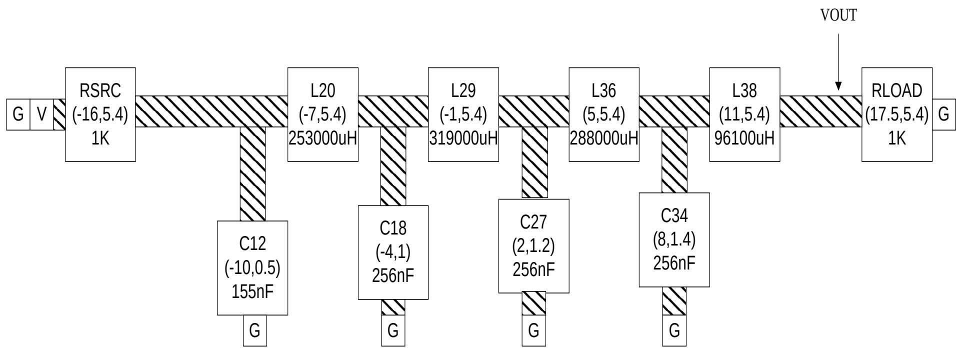

Figure 6.19
100%-compliant best-of-run circuit of generation 138 containing four inductors and four capacitors (total of only eight components) occupying an area of 359.4.

As can be seen in table 6.1, all three of these best-of-generation circuits have the same number (four) of capacitive or inductive shunts to ground. That is, they solve the problem with more or less the same approach. However, the best-of-generation circuits from the two earlier generations (25 and 30) each have a total of 16 inductors and capacitors, while the best-of-run circuit from generation 138 has only eight. Moreover, the best-of-run circuit from generation 138 has only one fifth of the area of the best-of-generation circuit from generation 25. The frequency-based term of the fitness measure (all of which are near zero) is shown for reference only.

Table 6.1
Comparison of three best-of-generation circuits scoring 101 hits.

Generation	Number of Capacitors	Number of Inductors	Number of Shunts	Area	Frequency-Based Term	Fitness
25	5	11	4	1775.2	0.264698	0.01775
30	10	5	4	950.3	0.106199	0.00950
138	4	4	4	359.4	0.193066	0.00359

6.7 Computer Time

This run involving a population of 1,120,000 required 1.55×10^8 fitness evaluations and took 28.4 hours (1.02×10^5) seconds on the 56-node parallel computer system described above. The 56 533-MHz processors operate at an aggregate rate of 2.98×10^{10} Hz, so that the run consumes a total of 3.04×10^{15} computer cycles (about 3 petacycles).

We estimate that about 1 1/2 orders of magnitude more computer time is required by genetic programming to automatically create the circuit's topology, sizing, placement, and routing than is required to merely create the circuit's topology and sizing. The basis for this rough estimate is as follows. In *Genetic Programming III* [Koza, Bennett, Andre, and Keane 1999, section 54.2], genetic programming automatically created the topology and sizing (but not placement and routing) of a lowpass filter circuit with the same specification as used in this chapter. Based on 64 runs of the simpler version of this problem with a population size of 30,000, a computational effort of E = 4,683,183 (30,000 $\times$ 43 generations $\times$ 3.63 runs) was required to yield a solution with 99% probability. When the population size was other than 30,000 for this problem in *Genetic Programming III*, a greater computational effort was found to be necessary to yield a solution to the simpler form of this problem (based on several groups of about 64 runs each). Moreover, none of the numerous additional isolated runs of the simpler form of this problem in *Genetic Programming III* required as few as 4,683,183 fitness evaluations. Although we cannot say with certainty that 30,000 is the optimal population size or that 4,683,183 fitness evaluations is the minimal computation effort required for the simpler form of this problem, we use 4,683,183 fitness evaluations as the best available estimate of the computational effort for the simpler form of this problem.

Each fitness evaluation (in both the simpler form of the problem and the run described in this chapter involving automatic creation of the topology, sizing, placement, and routing) entails a SPICE simulation consuming an average of 2.3×10^7 cycles. Thus, it is reasonable to say that the simpler form of this problem can be solved with a total of 1.08×10^{14} cycles with 99% probability.

Although it is not practical to make 64 28.4-hour runs of the more difficult version of this problem described in this chapter (involving automatic creation of the topology, sizing, placement, and routing) and although we cannot say that 1,120,000 is the optimal population size for this version of this problem, we use 3.04×10^{15} cycles as the best available estimate of the computational effort for the more difficult version of this problem. With these qualifications, we can say that the automatic creation of the topology, sizing, placement, and routing of a lowpass filter takes about 28 times more computer cycles than the automatic creation of merely the circuit's topology and sizing.

There are 14 instances in *Genetic Programming III*: *Darwinian Invention and Problem Solving* [Koza, Bennett, Andre, and Keane 1999, table 61.1] where genetic programming produced results that are competitive with human-produced results. Eleven of these involve the rediscovery by genetic programming of a previously patented invention. The runs that yielded these 14 instances consumed an average of 1.5×10^{15} computer cycles. Thus, the single run described in this chapter (involving automatic creation of the topology, sizing, placement, and routing of a circuit) took 2.03 times more cycles than the average of the 14 results in *Genetic Programming III*.

6.8 Genetic Programming as an Invention Machine

The best-of-run circuit from generation 138 (figure 6.19) has the recognizable features of the circuit for which George Campbell of American Telephone and Telegraph received U. S. patent 1,227,113 in 1917 [Campbell 1917]. Claim 2 of Campbell's patent covered,

> An electric wave filter consisting of a connecting line of negligible attenuation composed of a plurality of sections, each section including a capacity element and an inductance element, one of said elements of each section being in series with the line and the other in shunt across the line, said capacity and inductance elements having precomputed values dependent upon the upper limiting frequency and the lower limiting frequency of a range of frequencies it is desired to transmit without attenuation, the values of said capacity and inductance elements being so proportioned that the structure transmits with practically negligible attenuation sinusoidal currents of all frequencies lying between said two limiting frequencies, while attenuating and approximately

> extinguishing currents of neighboring frequencies lying outside of said limiting frequencies.

An examination of the evolved circuit of figure 6.19 shows that it indeed consists of “a plurality of sections” (specifically, four). Also, as can be seen in the figure, “Each section include[es] a capacity element and an inductance element.” Specifically, the first of the four sections consists of inductor L20 and capacitor C12; the second section consists of inductor L29 and capacitor C18; and so forth. Moreover, “one of said elements of each section [is] in series with the line and the other in shunt across the line.” As can be seen in the figure, inductor L20 of the first section is indeed “in series with the line” and capacitor C12 is “in shunt across the line.” This is also the case for the remaining three sections of the evolved circuit. In addition, the topology of the circuit in figure 6.19 of this chapter exactly matches the topology of the circuit in figure 7 in Campbell’s 1917 patent. Finally, this circuit’s 100%-compliant frequency domain behavior confirms the fact that the values of the inductors and capacitors are such as to transmit “with practically negligible attenuation sinusoidal currents” of the passband frequencies “while attenuating and approximately extinguishing currents” of the stopband frequencies. In short, the circuit created by genetic programming has all the features contained in claim 2 of Campbell’s 1917 patent.

Campbell received a patent for his invention of the ladder filter because it satisfied the legal criteria for obtaining a U. S. patent in that his filter was “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].

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.

Interestingly, the remainder of 35 *United States Code* 103a (quoted in part above) goes on to state,

> Patentability shall not be negatived by the manner in which the invention was made.

This wording suggests the permissibility of using genetic programming as an “invention machine” to produce patentable inventions. Unencumbered by preconceptions that may channel human thinking along well-trodden paths, genetic programming starts each run as a new adventure that is free to innovate in any manner that satisfies the requirements of the problem. Genetic programming is a search that is guided by the necessities articulated by the fitness measure of the particular problem at hand. Genetic programming approaches each new problem in terms of “what needs to be done” - not “how to do it.”

6.9 Conclusion

This chapter establishes the principle that genetic programming is capable of automatically creating the topology, sizing, placement, and routing of an analog electrical circuit. Specifically, the chapter starts with a high-level statement of the requirements for an analog lowpass filter and creates the topology, sizing, placement, and routing of a satisfactory circuit.

Bibliography

Aaserud, O. and Nielsen, I. R. 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, MA: The MIT Press.

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

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.

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

Cohn, John M., Garrod, David J., Rutenbar, Rob A., and Carley, L. Richard. 1994. *Analog Device-Level Layout Automation*. Boston: Kluwer.

Grimbleby, J. B. 1995. Automatic analogue network synthesis using genetic algorithms. *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*. London: Institution of Electrical Engineers. Pages 53–58.

Garey, Michael R. and Johnson, David S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY: W. H. Freeman.

Gruau, Frederic. 1992. *Cellular Encoding of Genetic Neural Networks*. Technical report 92-21. Laboratoire de l'Informatique du Parallélisme. Ecole Normale Supérieure de Lyon. May 1992.

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

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

Kitano, Hiroaki. 1990. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*. 4(1990) 461–476.

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

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

Koza, John R. 1994b. *Genetic Programming II Videotape: The Next Generation*. Cambridge, MA: The 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. (editors). 1998. *Genetic Programming 1998: Proceedings of the Third Annual Conference, July 22-25, 1998, University of 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., Bennett III, Forrest H, Andre, David, Keane, Martin A, and Dunlap, Frank. 1997. Automated synthesis of analog electrical circuits by means of genetic programming. *IEEE Transactions on Evolutionary Computation*. 1(2). Pages 109 – 128.

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: The MIT Press.

Kruiskamp Marinum Wilhelmus and Leenaerts, Domine. 1995. DARWIN: CMOS opamp synthesis by means of a genetic algorithm. *Proceedings of the 32nd Design Automation Conference*. New York, NY: Association for Computing Machinery. Pages 433–438.

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

Maziasz, Robert L. and Hayes, John P. 1992. *Layout Minimization of CMOS Cells*. Boston: Kluwer.

Perry, Tekla S. 1998. Donald O. Pederson - The Father of SPICE. *IEEE Spectrum*. 35(6) 22 – 27. June 1998.

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. 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.

Sechen, Carl. 1988. *VLSI Placement and Global Routing using Simulated Annealing*. Boston, MA: Kluwer.

Sterling, Thomas L., Salmon, John, and Becker, Donald J., and Savarese. 1999. *How to Build a Beowulf: A Guide to Implementation and Application of PC Clusters*. Cambridge, MA: The MIT Press.

Thompson, Adrian. 1996. Silicon evolution. In 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.

Thompson, Adrian. 1998. *Hardware Evolution: Automatic Design of Electronic Circuits in Reconfigurable Hardware by Artificial Evolution*. Conference of Professors and Heads of Computing / British Computer Society Distinguished Dissertation series. Berlin: Springer-Verlag.

Ullman, Jeffrey D. 1984. *Computational Aspects of VLSI*. Rockville, MD: Computer Science Press.

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

Wong, D. F., Leong, H. W., and Liu. C. L. 1988. *Simulated Annealing for VLSI Design*. Boston, MA: Kluwer.

Version 7 - Submitted December 26, 1998 for *Advances in Genetic Programming 3* book.

Automatic Synthesis, Placement, and Routing of Electrical Circuits by Means of Genetic Programming

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

Indexing Information