On the theory of designing circuits using genetic programming and a minimum of domain knowledge

CONFERENCE PAPER · JUNE 1998

DOI: 10.1109/ICEC.1998.699489 · Source: IEEE Xplore

CITATIONS
9

READS
8

4 AUTHORS:

David Andre
University of California, Berkeley
107 PUBLICATIONS  2,867 CITATIONS
See Profile

Forrest H Bennett III
Palo Alto Neuroscience
63 PUBLICATIONS  1,693 CITATIONS
See Profile

John R. Koza
Stanford University
174 PUBLICATIONS  10,247 CITATIONS
See Profile

Martin A. Keane
198 PUBLICATIONS  2,596 CITATIONS
See Profile

All in-text references underlined in blue are linked to publications on ResearchGate, letting you access and read them immediately.

Available from: David Andre
Retrieved on: 25 March 2016
On the Theory of Designing Circuits using Genetic Programming and a Minimum of Domain Knowledge

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

Forrest H Bennett III  
Visiting Scholar  
Computer Science Dept.  
Stanford University  
Stanford, California, 94305  
forrest@evolute.com

John Koza  
Stanford University  
Stanford, California, 94305  
koza@genetic-programming.org

Martin A. Keane  
Keane Associates  
5733 West Grover  
Chicago, Illinois 60630  
makeane@ix.netcom.com

Abstract: The problem of analog circuit design is a difficult problem that is generally viewed as requiring human intelligence to solve. Considerable progress has been made in automating the design of certain categories of purely digital circuits; however, the design of analog electrical circuits has not proved to be as amenable to automation. When critical analog circuits are required for a project, skilled and highly trained experts are necessary. Previous work on applying genetic programming to the design of analog circuits has proved to be successful at evolving a wide variety of circuits, including filters, amplifiers, and computational circuits; however, previous approaches have required the specification of an appropriate embryonic circuit. This paper explores a method to eliminate even this small amount of problem specific knowledge, and, in addition, proves that the representation used is capable of producing all circuits.

1. Introduction

Designing analog circuits is a difficult process that usually requires the skills of highly trained experts. Although most practicing engineers focus on the digital arena, the analog domain remains critical to the success of many products. Considerable progress has been made in automating the design of certain categories of purely digital circuits; however, the design of analog electrical circuits has not proved to be as amenable to automation [8]. There has been a great deal of interest in recent years on applying evolutionary computation to the design of analog circuits [3,4,5,6,8].

In our past work, we have demonstrated that it is possible to evolve a wide variety of analog circuits using genetic programming, including amplifiers, filters, audio circuits, voltage reference circuits, source identification circuits, robotic controller circuits, temperature sensing circuits, and computational circuits (such as the cube root function). However, our previous work required the specification of an appropriate embryonic circuit upon which the evolving developmental programs would act. Although these embryonic circuits require only a small amount of domain knowledge, we were often forced to choose between multiple possible initial circuits, which would require additional problem specific knowledge. Thus, in the present work, we eliminate the step of choosing a problem-specific embryo. In addition, we also formally present the representation used in our circuit design work, and prove it to be capable of expressing any desired circuit.

2. Evolving circuits: the representation

Typically, genetic programming breeds a population of rooted, point-labeled trees with ordered branches. There is a considerable difference between the kind of trees evolved by genetic programming and the component graphs encountered in the world of electrical circuits.

Electrical circuits are component graphs where the nodes are the leads of components, and the arcs indicate which leads are connected to one another. For example, a simple circuit consisting of a battery and a lamp could be represented as a graph where there are 2 components, 4 nodes, and 2 connective wires (figure 1a). A slightly more complex circuit (figure 1b) uses a three-leaded component to connect the two inputs to the output. Each component in a circuit has some number of leads, a type, and zero or more values. For example, a resistor has two leads, is a resistor, and has some resistance.

2.1 Mapping trees to circuits

Genetic programming can be applied to circuits if a mapping is established between the point-labeled trees found in the world of genetic programming and the component graphs employed in the world of circuits. [2] described an innovative technique, called cellular encoding, in which genetic programming is used to concurrently evolve the architecture of a neural network, along with all weights, thresholds, and biases of the neurons in the network. In this paper we apply cellular encoding to the evolution of circuits.

Cellular encoding can be viewed as a developmental graph grammar. The program trees are viewed as grammar trees that specify in a recursive fashion how the circuit
will be developed. In other words, the program trees construct the circuit. The functions in the program trees are operations that act on some piece of the developing circuit and transform it into some number of new pieces, which will then be further developed by the arguments to the given function. For example, a two-argument function might act on a single component and split it into two identical components, connected in parallel. The development of the two newly created components would be controlled by the two arguments to the original function. The component of the circuit upon which each function will most directly act is pointed to by a writing head that points from the function to the piece of the developing circuit. The result of the developmental process is the topology of the circuit, the choice of components that are situated at each location within the topology, and the sizing of all the components.

Each program tree can contain (1) topology-modifying functions that change the topology of the circuit, (2) component-creating functions that insert components into locations within the topology of the circuit and that have arithmetic-performing subtrees to specify the numerical value (sizing) for each such component, and possibly (3) automatically defined functions, which are routines that can express a re-usable subcircuit.

Program trees conform to a constrained syntactic structure. Each component-creating function has zero, one, or more arithmetic-performing subtrees and zero, one, or more construction-continuing subtrees. Each topology-modifying function has one or more construction-continuing subtrees. The arithmetic-performing subtree(s) of each component-creating function consists of a composition of arithmetic functions and numerical constants that together yield the numerical value for the component. The construction-continuing subtree specifies how the development of the circuit is to be continued.

2.2 The functions that act on the circuit

Circuits can be quite complex. For example, one can imagine a perfectly valid circuit that has 23 components connected to one end of a resistor. However, designing functions to act on such circuits is complicated, as the functions may affect the components attached to a given component in various ways. Thus, we introduced a constraint that limits the degree of the nodes in the graph without limiting the completeness of the representation. To do this, we allow wires to serve as virtual components during development. The wires merely connect two nodes together, and are removed prior to simulation of the circuit. However, they serve an important role as both a developmental placeholder and as a method to obtain virtual nodes of high degree. We limit the degree of all nodes in the circuit to 3, i.e. one lead of a component can be connected to at most two other leads. By using wires, however, virtual nodes of arbitrary degree can be created (figure 2).

To use a minimum of domain knowledge, we define a simple set of functions that can represent any circuit. To start with, we have component-creating functions that change the component pointed to by the writing head to each of the components in our parts bin. For example, we have resistor-creating functions that change the pointed-to component into a resistor (with the value specified by the arithmetic performing subtree) and transistor creating functions that change the pointed-to component into a transistor. Component creating functions that create components with two leads have construction-continuing subtrees as well that continue the developmental process. When transistors and other components with more than two leads are created, they do not undergo further development. Additionally, we need some method of connecting to the various inputs, outputs, power supplies, and grounds that might be associated with a circuit. For these, we have functions that take the pointed-to component and turn it into a set of wires that is connected to both of the original leads of the component and also to the desired port. Finally, we need some functions to directly act on the topology of the circuit and enable us to create arbitrary circuits.

The parallel functions, pss and psl, provide the ability to create new wires and new topology. Shown in figure 3, they make a copy of the pointed-to component and connect the new and existing components together with wires. Both functions take four arguments, so that both

Figure 1. Two simple circuits. a) shows a complete circuit in simplified drawing, with 2 components, 2 connective wires, and 4 nodes. b) shows a circuit in a more traditional drawing, where the connective wires of the components are not explicitly shown. In b), there are 4 components, 6 leads, and 3 wires.
Figure 3. The pss function starts from the original circuit, and acts upon r1. The pss function duplicates the resistor, and connects the new component to the “smallest” of the component values at each node. Thus, c2 ends up connected to r7 (and not c3). The pss stands for “Parallel Smaller Smaller”, and psl stands for “Parallel Smaller Larger”. For psl, c2 ends up connected to r1 and not r7, because c3 is “larger” than c2.

components and both new wires can undergo further development. Pss and psl differ in that they connect up the new component differently depending on the number of the neighboring components. Components are numbered sequentially based on when they are created. Additionally, we need a function that can remove a component. The cut function removes a component from the circuit and ends the developmental pathway.

3 Completeness of the representation

It is important that the representation be able to encode all circuits. To show this, we will construct an arbitrary circuit as follows. First, build a grid-like graph out of wires large enough to encode all of the nodes in the desired circuit (i.e. all of the leads of all the components). This can easily be done by repeatedly executing parallel operators (pss or psl) on the appropriate wires of any embryo. Then, prior to placing the components in the circuit, the desired connectivity between the nodes is introduced by the procedure shown in figure 4. By performing a series of parallel functions and cuts, a series of connected wires can be created that connects any two points in the circuit. This entails that any arbitrary graph can be created. After the desired connectivity is in place, the components are inserted. It is important not to insert the components too early, as they can block the success of the procedure (figure 4).

4 A bare minimum of domain knowledge

In our previous work, one of the preparatory steps for evolving a circuit with genetic programming was to choose an embryo that specified the test harness for the design problem being considered, the initial wires that would be modified, and also the connectivity among the initial wires and the test harness. Although sometimes it is desirable to specify an embryo precisely based on some problem specific knowledge, if no such knowledge is available, then it is desirable that the system function with less initial information. In the minimal case, only the number of inputs and outputs to the desired circuit need be specified. A problem-blind minimal circuit of two wires is used as the embryo, and new functions are added that allow the creation of connections to the inputs and outputs of the circuit.

4.1 Setting up the problem

To illustrate the approach, the problem of evolving a lowpass filter using a simple function set and a very simple embryo is considered. To apply genetic programming to the problem, we first must choose the function set. As we are evolving a simple filter, only capacitors and inductors are needed. Thus, the function set for the circuit-continuing subtrees is

\[ F_{circuit} = \{C, L, PSS, PSL, GND, INPUT, OUTPUT\} \]

and the terminal set is

\[ T_{circuit} = \{END\} \]

The C and L functions are those that replace a component with a capacitor or an inductor, respectively, and the GND, INPUT, and OUTPUT functions connect the component to the GND, INPUT, or OUTPUT nodes for the circuit. If more inputs or outputs were required, multiple input and output functions would be used. For the arithmetic subtrees that set the values of the components, the function set is

\[ F_{arithmetic} = \{+, -, \} \]

and the terminal set is

\[ T_{arithmetic} = \{\mathbb{R}\} \]

where \( \mathbb{R} \) represents floating-point random constants from -1.0 to +1.0. The embryo for this problem is the simplest embryo possible in our system, a pair of wires connected at each end, shown in figure 5 (a one wire embryo is not possible because we do not allow nodes of degree one). Thus, there are two main branches in each individual, one for each initial wire.

Note that all of the above could be applied to any one-input, one-output LC circuit. It is the fitness measure that directs the evolutionary process to a program tree that constructs the desired lowpass filter. In other words, circuit structure arises from fitness.
The evaluation of fitness for each individual circuit-constructing program tree in the population begins with its execution. This execution applies the functions in the program tree to the very simple embryonic circuit, thereby developing the embryonic circuit into a fully developed circuit. A netlist describing the circuit is then created. The netlist is an industry standard ASCII representation that identifies each component of the circuit, the nodes to which that component is connected, and the value of that component. It is at this point that all the wires used in development are compressed, as the netlist has no restriction on the degree of a node. Each circuit is then simulated to determine its behavior. The 217,000-line SPICE simulator was modified to run as a submodule within the genetic programming system.

SPICE (an acronym for Simulation Program with Integrated Circuit Emphasis) is a massive family of programs written over several decades at the University of California at Berkeley for the simulation of analog, digital, and mixed analog/digital electrical circuits [7]. Since we are designing a filter, the focus is on the behavior of the circuit in the frequency domain. SPICE is requested to perform an AC small signal analysis and to report the circuit's behavior for each of 101 frequency values chosen from the range between 10 Hz to 100,000 Hz (in equal increments on a logarithmic scale).

Fitness is measured in terms of the sum, over these 101 fitness cases, of the absolute weighted deviation between the actual value of the voltage that is produced by the circuit at the probe point \( U_{\text{out}} \) and the target value for voltage. The fitness measure does not penalize ideal values; it slightly penalizes every acceptable deviation; and it heavily penalizes every unacceptable deviation. Thus, for the lowpass filter, the output in the pass band is expected to be 1 volt, but voltages above 0.97 volts are considered acceptable deviations. In the stop band, the output is expected to be zero, but voltages of up to 0.001 are acceptable deviations. In the small area comprising 5 points between the stopband and the passband, the response is ignored, as this is a transition band.

Many of the circuits that are created in the initial random population and many that are created by the crossover and mutation operations cannot be simulated by SPICE. Such circuits are assigned a high penalty value of fitness \( 10^8 \).

---

Figure 4. 1) A grid large enough to contain all of the nodes of the desired circuit. Arbitrary connections can then be created as follows. 2) Use a parallel operator to jump across the first gap, without crossing the wires. 3) Use a parallel operation to jump to the next square, using a crossing parallel. Whether the right function is pss or psl will depend on the number of the neighboring components. 4) Jump to the long vertical wire using a crossing parallel function. 5) Using parallel functions, continue creating a path to B. 6) Remove all unnecessary wires. The circles represent the two nodes that make up the virtual nodes A and B.

Figure 5. A simple two wire embryo. Both \( Z_1 \) and \( Z_2 \) have writing heads pointing to them.
The problems presented in this paper were run on a medium-grained parallel Parsytec computer system consisting of 64 80-MHz Power PC 601 processors arranged in a toroidal mesh with a host PC Pentium type computer. Details of the parallel implementation of genetic programming followed [1].

4.2 Results

The best individual from generation 0 from one run of this problem had a fitness of 145.3 and was a very simple individual, using only one capacitor. This is a simple low quality low-pass filter, and achieves the frequency response shown in figure 6.

On generation 92, a circuit that satisfied all of the specifications emerged. This circuit is shown in figure 7, with its frequency response in figure 8. Thus, even though a very simple embryo was used, the system was able to evolve a circuit to the desired specifications. It is an interesting question to consider the computational expense of using less problem specific knowledge. Although extensive comparisons have not been performed, on all the runs we observed, it took approximately 1.5 times as long to evolve satisfactory filters when only the simple two-wire embryo was used as opposed to the more connected embryo discussed in section 6 (and which is similar to the embryos used in our previous work).

5 Starting with a connected, basic embryo

Although using a bare minimum of problem specific knowledge has its importance, for practical reasons, it is often beneficial to include some limited domain knowledge in order to speed up the run. In this section, we use an embryo that assumes that the initial wires are connected to the input, the output, and the ground ports. This embryo is shown in figure 9.

For this problem, the input and output functions were removed from the function and terminal set, as they are included in the embryo. The ground function was left in the function set, as it is often beneficial to add grounds at various points in a circuit. Thus, a small amount of domain knowledge went into this embryo and function set. However, the function set used herein contained only the two basic topology creating operations and not the extensive set used in [4,5].

On one run of this problem, a circuit satisfying all of the design criteria emerged on generation 66. The evolved circuit is shown in figure 10. The frequency behavior is similar to that shown in figure 8.

6 A few more functions

For most of our research on this topic, we use an embryo similar in spirit to figure 9, where the initial wires are connected to the inputs and outputs to the circuit. We have also included a variety of supplemental functions that are not necessary but speed the time required to solve the problem. For example, we include a set of via functions that allow quick ‘jumps’ to other points in the circuit. We also include a series function (which can, of course, be built from the parallel functions and cut). Also, we include a NOP function for simplifying the relative developmental timing of different subtrees, a FLIP function that switches the polarity of a component (such as a diode), and any components necessary for the given problem, such as transistors.
Figure 9. A slightly more complex embryo.

Using this model, genetic programming has evolved circuits for a variety of problems, including a two-output crossover filter (woofer/tweeter filter), a difficult-to-design bandpass filter, comb filters, and a source identification circuit. The system is not limited to passive circuits; amplifiers, computational circuits, temperature sensing circuits, voltage reference circuits, and robotic control circuits have also been evolved [4].

An important characteristic of the approach is that highly desirable problem specific sub-circuits can be included as basic components in the system. For example, filters or op-amps can be included as primitive functions. For the op-amp, which can have 4 inputs and 1 output (depending on the model of the opamp being used), it is necessary when inserting the opamp into the circuit to replace just one component, but 2 adjacent components. This facility fits seamlessly into the genetic programming system for evolving circuits.

7 Conclusions and Future work

The genetic programming system for evolving analog circuits for a set of desired specifications was presented. The representation used by the system was shown to be capable of developing arbitrary circuits. As an example, results of using the system with a bare minimum of problem specific knowledge were presented for the low pass filter problem. Results of using the system with marginally more domain knowledge were reviewed. Starting with an embryonic circuit, a bag of components, connective functions to glue the components together, and an analog circuit simulator, genetic programming is capable of evolving circuits for a set of desired specifications. The technique requires only a bare minimum of problem specific knowledge, and thus is applicable to a wide variety of circuit design problems.

Acknowledgments

The first author is supported by a National Defense Science and Engineering Grant.

Bibliography