Chapter 11

Evolving Logic Gates

©2001-2003 by Dan Ashlock

In this chapter, we will be evolving logic gates. We will look at different ways of coding neural networks, at different mutation and crossover operators, and on their effect on speed of convergence. We will compare neural nets with genetic programming as a way of performing digital logic function induction. We assume the reader to be familiar with the logic of NOT, AND, OR, NAND, NOR, XOR, parity, and majority gates. When writing out logic functions, we will use the convention that NOT is the unary operation $\neg$, and AND and OR are the binary operations $\wedge$ and $\vee$, respectively. In the genetic programming section we will switch to printable characters (, *, and +) for these logic functions.

11.1 Introduction to Artificial Neural Nets

Artificial neural nets are structures inspired by biological nervous systems. In a living creature, neurons are connected to one another at *synapses*. The neuron is both excited and inhibited by impulses coming in from different synapses. When the total level of excitation passes a threshold, the neuron *fires* or *turns on*. A biological nervous system functions through the effects of many such firings in a net of interconnected neurons. It changes by changing its physical connections and their properties. This is accomplished both by changing the strength of synaptic connections and by changing the chemical environment. This chemical modification happens directly when neurons release biochemicals and indirectly when the neurons cause various organs to release biochemicals. The bandwidth and operation of a biological neural net often defies understanding or even adequate description.

An artificial neural net is simpler. It has neurons with inputs and outputs and intraneuron connections that are analogous to biological synapses. Each connection has a weight, usually a real number, and a direction. The direction establishes which neuron is sending the message along the connection and which neuron is receiving. The character of a neuron is determined by its *transfer function*. This function computes the neuron's output from the sum of its weighted inputs. When functioning, a neuron follows these steps: first, the inputs from connected neurons are multiplied by their connection weights and summed; the neuron then passes that sum through its transfer function; then, the result is sent along all the neuron's output connections. The neurons in a neural net can update their output values synchronously or asynchronously. The choice of updating method has a large effect on the behavior of the net. Graphs of several common transfer functions appear in Figure 11.1.

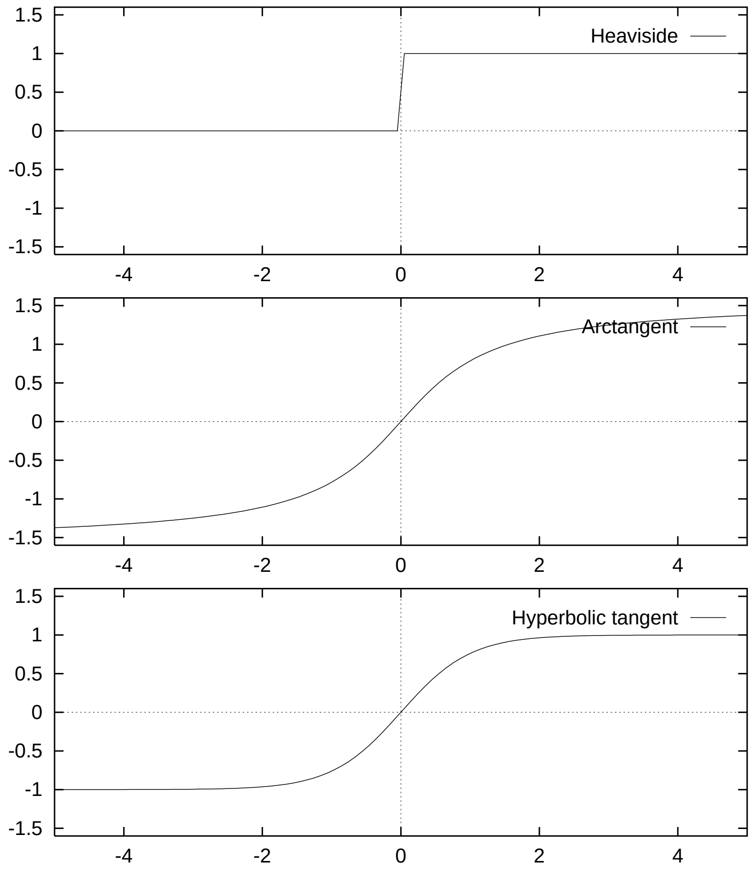

Figure 11.1: Heaviside, arc tangent, and hyperbolic tangent transfer functions for neurons.

When building a neural net, there are many choices to be made. What type of transfer function are you going to use? Should you allow the transfer functions to take parameters? What sort of connection topology are you going to use? How are you going to program your neural net?

Definition 11.1 *The* **connection topology** *of a neural net is a specification of which neurons are connected to which other other neurons. The connection topology is a directed*

2-input **XOR** Gate		
Input 1	Input 2	Output
0	0	0
0	1	1
1	0	1
1	1	0

3-input **AND** Gate			
Input 1	Input 2	Input 3	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Figure 11.2: Examples of truth tables for logic gates.

graph. If this graph contains no directed cycles, then the net is said to be **feed forward***, otherwise it is* **recurrent***.*

There are many ways to program neural nets. We will use the various evolutionary algorithms described in Section 1.2. For large net sizes and for some problems, evolutionary algorithm based programming of neural nets appears to be inferior to other methods. Where the evolutionary method shines is in the programming of small neural nets that have to do an ill-defined task and in programming systems of small neural nets that are later integrated together to do a complex task. We will study an example of the latter technique in Chapter 5.

The simplest transfer function used in neural nets is a step function. A neuron that uses this function is called a *Heaviside* neuron. It sums its weighted inputs, and it outputs 1 if the result is greater than a threshold value t, and it outputs 0 if the result is t or less. The first graph in Figure 11.1 is a Heaviside transfer function. These exceedingly simple neurons are more than adequate to the job of building neural nets that simulate logic gates. Logic gates are circuits that take n binary inputs and produce a single binary output. Examples of logic gates are: *AND* gates (which output 1 only if all their inputs are 1), *OR* gates (which output 1 if any of their inputs are 1, and 0 otherwise), and *NOT* gates (which has a single input and whose output is 1 minus its input). Logic gates are described with truth tables, examples of which appear in Figure 11.2. Truth tables list all possible combinations of inputs with their corresponding outputs.

There are two broad classes of neural nets, *feed forward* and *recurrent*. *Recurrent* nets have loops; *feed forward* nets don't have loops. What is a loop? Recall that all connections in a neural net have a direction. A *loop* is a sequence of connections, followed in the positive direction, that returns to its starting point. Other things being equal, a recurrent net is much more complex in its behavior than a feed forward net. Recurrent nets can, for example, develop long-term memory; the ability of feed forward nets to remember previous inputs is limited to the longest sequence of connections following an input.

Problems

In the following problems, assume the neurons are Heaviside neurons. You select the threshold value t. Use the same t for all the neurons in a single net.

Problem 11.1 *Implement each of the following logic gates as a neural net:*

(i) *NOT*,

(ii) 2-input *AND*,

(iii) 2-input *OR*,

(iv) 2-input *XOR*.

Problem 11.2 *A* majority circuit *is a logic gate with an odd number of inputs whose output is equal to the majority of its inputs (majority vote). Construct neural nets that implement 3-input and 5-input majority circuits.*

Problem 11.3 *Compute the number of different n-input logic gates. Hint: count the number of ways to fill in the output column in a truth table.*

Problem 11.4 *Show that you can simulate any n-input logic gate using a single 2-input logic gate with multiple instances connected together. Give a neural net implementation of such a gate with the fewest possible neurons.*

Problem 11.5 *Give a bound on the number of neurons needed to implement an n-input* AND *gate, trying to make the bound as small as possible.*

Problem 11.6 *Suppose we are building an n-input logic gate which outputs 1 for exactly k of the possible inputs. Prove that we need at most $k + 1$ neurons to build the circuit. This bound is not tight. Give an example of a 4-input gate that outputs 1 for exactly 6 of its possible inputs and which uses fewer than 7 neurons.*

Problem 11.7 *Suppose you have an endless supply of 3-input majority circuits and* NOT *gates. What is the smallest number of gates you could connect together to make a 4-input* AND *gate? Assume that you have a input constant of 0 or 1 available. Advanced students: prove that the constant is necessary.*

Problem 11.8 Essay. *Given that evolution is the primary method we will use to find, design, or program the structures we are studying in this book, why is question 11.7 relevant?*

Problem 11.9 Essay. *The connections in a neural net hold the knowledge (or functionality) of the neural net. The pattern of connections and weights are both relevant. How much information can be packed into a single abstract real number? Be sure to* state *the definition of the term "information" you are using in your essay. Discuss the relevance of your answer to a neural net that might actually get built.*

Problem 11.10 *Construct a recurrent neural net with two inputs r and t that acts as follows. If t is 1, then the output of the net should be equal to r. If t is 0, then the output should be whatever it was the last time t was 1. This net is a very simple form of memory device: r is what is to be remembered, and t tells the device when to remember it.*

Problem 11.11 *Suppose you want to store in a computer unambiguous specifications of functions computed by logic gates. How many bits will you need to specify an n-input logic gate? Notice that the two examples in Figure 11.2 use 12 and 32 bits, respectively. (There are 12 or 32 1s and 0s in the truth tables.) These are* not *minimal representations.*

11.2 Evolving Logic Gates

In this section, we will be evolving logic gates that use the truth values: 0 = false and 1 = true. The neurons in our neural nets will be Heaviside neurons, making this assignment of truth values a natural one, as it is already the output type of the neurons.

An n-input logic gate can be thought of as a string over the alphabet $\{0, 1\}$ of length 2^n which is the output column of its truth table when inputs are listed in the standard binary order, the one that corresponds to counting in binary. The truth table of the AND function, shown in Table 11.2, would correspond to the string 0001, for example. Table 11.2 gives the output strings for several standard logic gates.

2-input AND gate		
Output	Input1	Input2
0	0	0
0	0	1
0	1	0
1	1	1

Table 11.1: AND gate truth table

Inputs	Gate	Output String
1	NOT	10
2	OR	0111
2	AND	0001
2	XOR	0110
2	NOR	1000
2	NAND	1110
3	Majority	00010111
3	Parity	01101001
3	AND	00000001
3	OR	01111111

Table 11.2: Output strings for various logic gates

In order to obtain a design for a logic gate as a neural net, we will evolve the output strings of logic gates to match a reference string that is the output string of the desired gate. How many neurons should we use? What connection topology should we use? In this section, we will simply choose to implement an n-input gate as a feed forward network with 3 layers of neurons, m neurons in the first layer, k neurons in the second layer, and 1 in the last. The real parameters of a gate will be an $n \times m$ matrix, F, that holds the connections of the inputs to the first layer of neurons, an $m \times k$ matrix, S, that holds the connections of the first layer to the second, and a $k \times 1$ matrix, T, that holds the connections of the second layer to the output neuron, together with $(m + k + 1)$ different threshold values for the neurons. This involves $nm + mk + m + 2k + 1$ total real values. For 2-input gates, we will find that $m = k = 2$ will suffice, which means we'll need 15 real parameters.

In addition to a model of evolution and some mutation operators taken from real function optimization, the pieces we will need to build a logic-gate evolver are as follows. First, we need a fitness function. Our initial fitness function will be the number of agreements of the output string of a neural net representing a logic gate (see Problem 11.14) with a reference string that is the output string of the desired logic gate. We also need a way of evaluating the neural net efficiently. If $\vec{v}$ is the vector of inputs to the net, then $\vec{v}F$ is the set of inputs to the first layer. Running this vector through the thresholds of the first layer gives us a new $0, 1$ vector $\vec{u}$ and $\vec{u}S$ is the vector of summed inputs to the second layer. Running this through the thresholds of the second layer produces a $0, 1$ vector $\vec{t}$ that is the inputs to the third layer and so $\vec{t}T$ is the number that the final neuron thresholds to give an output. From this discussion, one can see that running the neural net amounts to multiplying vectors by matrices and thresholding the resulting vectors. The last piece needed to evolve neural nets is a selection of crossover operators.

The structure of the neural net as 3 matrices and the individual neural thresholds is not naturally a string of real numbers. Each of the matrices could be peeled either by rows or by columns to make a string of real numbers. The 6 strings, 3 from the matrices and 3 from the thresholds of each layer, could be concatenated in any order. For simplicity, we will always use one of two ways of transforming the neural net into a string: row peeling of all 3 matrices or column peeling of all 3 matrices. Suppose we have a 2-input neural net with $m = k = 2$. If all the neurons have threshold values of 0.5, and if we have

$$F = \begin{bmatrix} 0.2 & 0.3 \\ -1 & 0.7 \end{bmatrix},$$

$$S = \begin{bmatrix} 0.6 & -0.2 \\ 0.9 & 0.8 \end{bmatrix},$$

$$T = \begin{bmatrix} 0.25 \\ 0.41 \end{bmatrix},$$

then the row peeling would yield the string of reals

$$(0.2, 0.3, -1, 0.7, 0.5, 0.5, 0.6, -0.2, 0.9, 0.8, 0.5, 0.5, 0.25, 0.41, 0.5)$$

while the column peeling would yield

$$(0.2, -1, 0.3, 0.7, 0.5, 0.5, 0.6, 0.9, -0.2, 0.8, 0.5, 0.5, 0.25, 0.41, 0.5).$$

In both row and column peeling, we put the matrix entries and the threshold values together in the order in which we use them in traversing the net. Once we have the neural net parameters peeled into a string of reals, we can use any of the crossover operators we have used in the past for real function optimization.

Notice that we are simplifying matters by leaving the neuron's internal parameter (the threshold value) constant, rather than permitting it to evolve.

Experiment 11.1 *Create or obtain software for an evolutionary algorithm to operate on a gene containing an $n \times m$ matrix, an $m \times k$ matrix, and a $k \times 1$ matrix, coding a neural logic gate as described in the text. Let all the neurons have the same threshold value α. Take, as your fitness function, the number of positions on which the output string of the evolving gate agrees with the output string of the desired gate. Use single tournament selection with tournament size 4 and single point uniform real mutation with mutation size $\epsilon = 0.2$. Use column peeling to generate the strings for crossover. Set $n = k = m = 2$, take $\alpha = 0.5$, and perform exactly one mutation on each gene generated by crossover.*

With a population size of 200, run an evolutionary algorithm 100 times for at most 400 generations on the task of locating weights for an XOR gate. Let the initial connection weights be in the range $-1 < x < 1$, and do not permit the threshold numbers to change. Save the number of generations to solution. Repeat the experiment with $\alpha = -0.5$. In your write up, use the normal approximation to the binomial at the time half of one set of runs are completed to decide if the choice of α makes a significant difference.

The next experiment is for the suspenders-and-belt student. It is intuitive that the choice between column peeling and row peeling isn't all that important. The only way to tell *for sure* is to do the experiment.

Experiment 11.2 *Modify the software from Experiment 11.1 to use row peeling instead of column peeling. Run the same simulations and compare the time-to-solution results in the same fashion, both within this experiment and between experiments.*

Definition 11.2 *Call a logic gate* **irreducible** *if, for each input i, there is some combination of values for the other inputs for which changing the value of i will change the value of the output.*

What distinguishes interesting logic gates from boring ones? The definition of irreducibility is an attempt to formally define that property. (For more about irreducibility, see Problems 11.15, 11.16, and 11.17.) One quite simple and interesting gate is the *parity* gate on n inputs. If an odd number of its inputs are 1, the parity gate returns 1, otherwise it returns 0. Notice that XOR is 2-input parity. Because the n-input parity function is so interesting, it is a standard test case for systems that are supposed to induce logic gates.

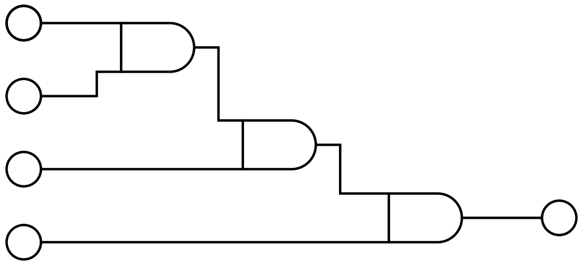

Figure 11.3: A cascade of 3 AND gates

A *cascade* of (2-input) gates is a diagram of the sort shown in Figure 11.3. Very often, a cascade of gates extends the functionality of the gate to a larger number of inputs. A cascade of n 2-input AND or OR gates yields an $(n + 1)$-input AND or OR circuit, for example. Likewise, a cascade of $(n - 1)$ XOR gates yields n-input parity. This implies that, for an evolutionary algorithm that works on large logic gates, some sort of variation operator that creates cascades would be a good thing.

At the moment, we want to look at a small, meaningful subunit of any neural logic gate. One neuron connected to two inputs can simulate a 2-input gate. Such a net-fragment is depicted in Figure 11.4. For a fixed threshold value t, we can determine the gate's behavior as a function of its two input weights, a and b. There are 16 possible behaviors, corresponding to the 16 2-input logic gates, only some of which are possible. We find which we have by applying the 4 possible combinations of zero-one inputs (00, 11, 01, and 10) and examining

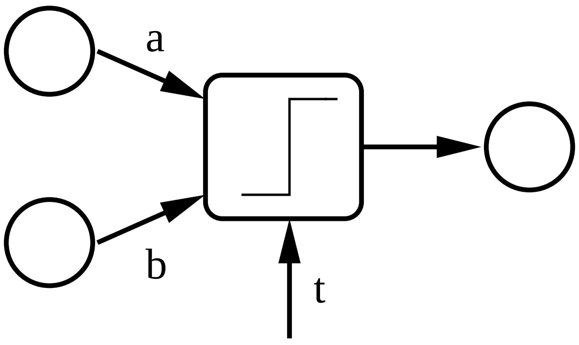

Figure 11.4: A minimal fragment of a nerual net capable of representing a gate

the output behavior. The result is a "phase diagram," indexed by the values of a and b. A pair of such phase diagrams for threshold values $t = 0.5$ and $t = -0.5$ are given in Figure 11.5.

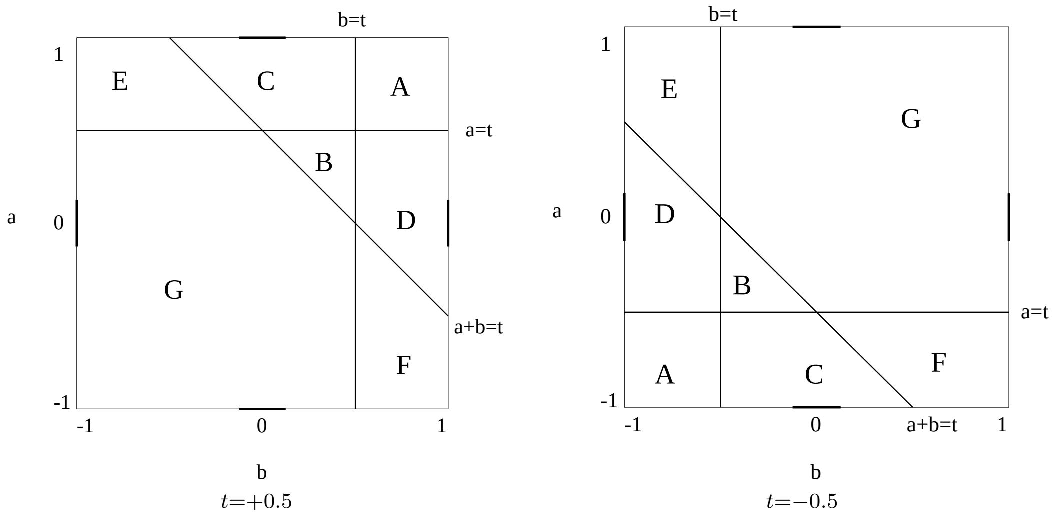

Figure 11.5: Phase diagram of logic in a gate fragment from Figure 11.4 in terms of connection weights a and b (For positive threshold t, the logical behaviors are A - $a \vee b$, B - $a \wedge b$, C - a, D - b, E - $a \wedge (\neg b)$, F - $(\neg a) \wedge b$, G - $false$. For negative threshold t, the logical behaviors are A - $\neg(a \vee b)$, B - $\neg(a \wedge b)$, C - $\neg a$, D - $\neg b$, E - $a \vee (\neg b)$, F - $(\neg a) \vee b$, G - $true$.)

Make sure that Figure 11.5 makes sense to you. It seems that the two-connection-weights-and-one-neuron fragment shown in Figure 11.4 can produce very different sets of logical functions depending on whether the threshold is positive of negative. In particular, the two functions from which any other function can be built, $\neg(a \wedge b)$ (NAND) and $\neg(a \vee b)$ (NOR), appear only when the threshold is negative. This sheds light on the outcome of

Experiment 11.1. It also suggests a second experiment to see what thresholds evolution will select, left to its own devices.

Experiment 11.3 *Modify the software from Experiment 11.1 to make the threshold weight of each neuron evolvable and initialized in the range (-1,1). Rerun the experiment, documenting whether solution time with floating thresholds is significantly different from that with fixed positive or negative threshold values. Also, plot the fraction of neuron thresholds that are positive and negative in the final generation of each population. (The instructor may wish to assign Problem 11.23 with or before this experiment.)*

Experiment 11.3 is intended to check whether the theoretical suggestions of Figure 11.5 are correct. If they are, then most populations should have at least one negative threshold weight. This question is explored in more detail in Problem 11.23, but be sure to at least consider the question of what outcomes in terms of distribution of negative thresholds would be interesting or suggestive.

At this point, we suggest an experiment designed to verify something hinted at other places: XOR is hard. In earlier experiments, we used an evolutionary algorithm to locate XOR gates. Now, we will locate some other gates as well for comparison.

Experiment 11.4 *Modify the software from Experiment 11.1 to search for a NOR (output string 1000), an AND (output string 0001) and an "a" gate 0101 (it just echoes one of its inputs). Save the time-to-solution data and decide which of these gates is significantly easier to locate than (i) the others and (ii) the XOR gate located in Experiment 11.1. Notice that all 3 of the logic functions for which you are locating gates in this experiment can be implemented as a gate fragment of the type shown in Figure 11.4.*

To conclude this section, we will increase the size of the neural logic gates for which we are searching. All the experiments thus far have been on 5-neuron nets with 2 inputs. In the remainder of the chapter, we will work with larger gates, and, so, this experiment will provide a basis for comparison.

Experiment 11.5 *Modify the software from Experiment 11.1 to search for 3-input AND and parity gates. This means that $n = 3$ and we will take $m = k = 3$ as well. Set the neuron thresholds to -0.3 and perform 100 simulations for each gate, leaving the simulation parameters the same. Save the times-to-solution and compare them for these two gates.*

In this section, we have built code for evolving 3-layer feed-forward neural nets with all possible connections between layers. In the next section, we will retool our representation of neural nets to permit us to evolve connections as well as the weights involved.

Problems

Problem 11.12 *Compute the output string for the following logic gates. Assume that the inputs are listed in the usual binary counting or lexicographic order.*

(i) 5-input parity

(ii) 3-out-of-5 majority

(iii) A cascade of 4 NAND gates

(iv) A cascade of 4 NOR gates.

Problem 11.13 *Give a logical function that will yield the following output strings. Call your inputs $x_1, x_2, \ldots, x_n$ and use only AND, OR, and NOT to build your logical expressions. Assume x_1 is the lowest order bit of the input(the one that changes the most often).*

(i) **0110**,

(ii) **01101001**,

(iii) **00000001**,

(iv) **11111110**,

(v) **01110000**.

Problem 11.14 *Suppose you have an n-input function called GATE that somehow implements a logic gate L. Give pseudo-code for a function that produces the output string of L. Do* not *nest n loops; this is inelegant.*

Problem 11.15 *Give an algorithm for detecting irreducibility (see Definition 11.2).*

Problem 11.16 *Explain why irreducible logic gates are interesting.*

Problem 11.17 *Prove that n-input parity, n-input AND, and n-input OR gates are all irreducible.*

Problem 11.18 *Suppose that, for some reason, the inputs of a logic gate are on the vertices of a regular pentagon. Give the output string for a logic gate that outputs 1 when no symmetry of the pentagon (rotation or flip) takes the pattern of inputs to itself, and outputs 0 otherwise. Hint: if we had said square instead of pentagon the answer would be 0110100110010110.*

Problem 11.19 *Suppose we generalize Problem 11.18 to an n-gon. We call such a gate an n-input asymmetry detector. Prove that the output string of the gate must be a palindrome.*

Problem 11.20 *Examine Figure 11.5. The choice of the neuron threshold in the 1-neuron 2-input gate fragment creates a good deal of variation in which logic gates are possible. Each of the negative threshold gates is a logical inversion of one of the positive threshold gates, and 7 gates appear in each diagram. Since there are 16 total gates, two are missing. Which two?*

Problem 11.21 *Examine Figure 11.5. If the two weights a and b are generated uniformly at random between 1 and -1 then what is the probability of each of the gates (i) for* $\alpha = +0.5$ *and (ii) for* $\alpha = -0.5$*?*

Problem 11.22 *Examine Figure 11.5. For what value of the negative threshold would the NAND* ($\neg(a \wedge b)$) *and NOR* ($\neg(a \vee b)$) *have equal probability of appearing? Assume that the selection of* a *and* b *is made uniformly at random on the interval* $-1 \leq x \leq 1$. *Reading (or doing) Problem 11.21 may clarify this question.*

Problem 11.23 Essay. *A fact of digital logic is that the only 2-input gates from which you can construct any other are the NAND or the NOR. Because of this, the information in Figure 11.5 suggests that a negative threshold is probably a good thing. Experiment 11.3 tests how often negative thresholds evolve. Here is the essay question: what evidence from Experiment 11.3 demonstrates the value of negative weights and why? Possible answers include: overabundance or appearance at least once in each gate. Be sure to state* and defend *your conclusions.*

11.3 Selecting the Net Topology

Even with a fixed number of inputs, some logic gates require more neurons than others. Compare, for example, minimal neural-net implementations of a 3-input AND and a 3-input parity gate. In addition, a net done by layers with all possible connections present will often have connections it does not need. When we do not know ahead of time what the good number of neurons or connections is, it would be nice to let evolution select, not only the connection weights, but the topology (wiring diagram, layout) of the neural net. One way to do this is to fix the number of neurons and evolve a list of weighted connections between neurons. Neuron number can, to some degree, be evolved by noticing when evolution has chosen to connect a neuron in a fashion that makes it irrelevant to the output of the gate.

In order to ensure that we get a sensible feed-forward net, we need to impose some restrictions on the possible connections (read Section 11.1 for terminology). Our evolutionary algorithm will operate on a list of connections. Each connection will be of the form: (a_i, b_i, ω_i), meaning that the output of neuron a_i is connected to the input of neuron b_i with connection weight ω_i. An example of such a neural net is given in Figure 11.6. From that example, it is clear we need the following restrictions. First, the inputs and neurons are numbered consecutively and all connections go from smaller numbers to larger numbers ($a_i < b_i$). This ensures that the net is feed-forward. Second, all b_i must be large enough *not* to be inputs. Third, the output is taken off of the highest numbered neuron. We will call this scheme for specifying a neural net a *connection-list specified neural net.*

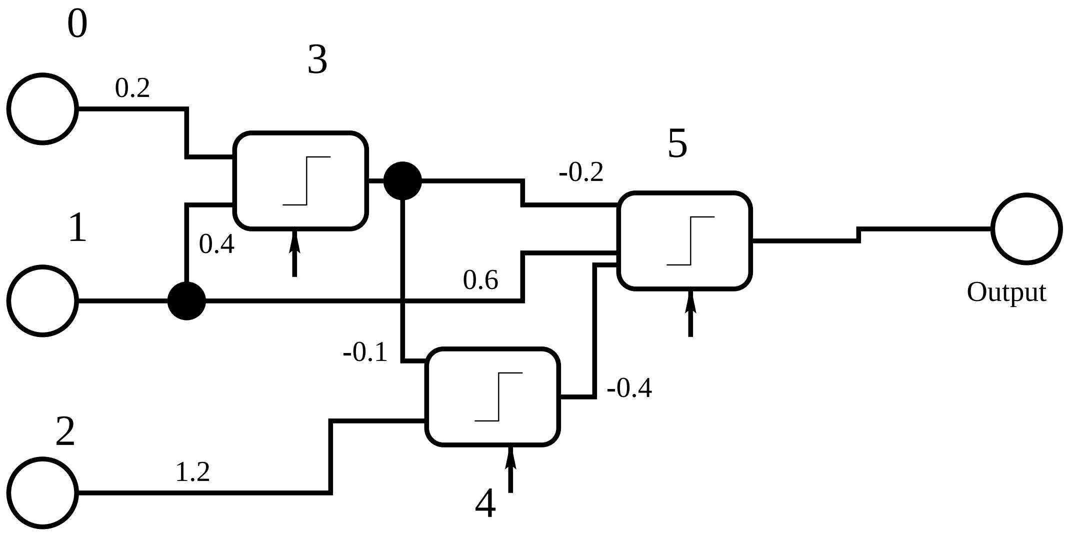

Figure 11.6: Neural net derived from the connection list: $(0, 3, 0.2)$, $(1, 3, 0.4)$, $(1, 5, 0.6)$, $(2, 4, 1.2)$, $(3, 4, -0.1)$, $(3, 5, -0.2)$, $(4, 5, -0.4)$ (Notice inputs are numbered and treated as neurons.)

There are some restrictions that seem natural, but are unnecessary. For example it would be natural to demand that, when $i \neq j$, $a_i \neq a_j$ or $b_i \neq b_j$. In other words, no double connections. Instead, we will simply interpret a double connection as a connection whose strength is the sum of the individual connections. This gives evolution the freedom to reduce the number of connections in a net by superimposing them. Implicit in this is a need to simplify an evolved net, once we have it in hand.

Evaluating a Connection-List Specified Neural Net

Once we have a list of connections that specify a neural net, we have the problem of actually evaluating the list and producing a net. What follows is one possible method. Suppose our neurons are Heaviside neurons with a known weight threshold and that, including inputs, we have n neurons. From the list, we will derive a *weight matrix* as follows. First, initialize an $n \times n$ matrix M to all zeros. Traversing the connection list, add ω_i to $M[a_i, b_i]$. Now, all the connections are stored in M.

Evaluating the net is done recursively. Initialize a boolean n-vector v to [*input*], that is to say put true in $v[i]$ for i equal to the index of an input. Keep, also, an output n-vector of reals, m, that holds the output values of each neuron. It is initialized to hold the known input values. These vectors are used by a recursive routine that returns the output value of any desired neuron, given values for the input neurons, by summing the appropriate row of M times the outputs of relevant neurons connected to the desired neuron. When the output of a given neuron is known, it is in m and the corresponding entry of v is true. Otherwise, a recursive call is used. Code for this evaluation routine is given in Figure 11.7.

Notice that the algorithm given in Figure 11.7 does not compute a neuron's output value until it needs it. This is called *lazy evaluation* and is a standard method of reducing computational work. Before we define evolutionary algorithms for connection-list neural nets, let's do a sampling experiment that will permit us to implement and debug our evaluator.

Experiment 11.6 *Implement or obtain software for generating and evaluating connection-list specified neural nets. The number of inputs and neurons should be easy to change. Generate 1000 different 3-input nets with 5 neurons other than the inputs and 15 connections. Use Heaviside neurons with a threshold value of* $\alpha = -0.5$. *Evaluate them, determining their output string, and tally how many times each output string appears. Report the empirical probability of each of the 256 possible gates. Which was more likely: AND or parity?*

To expand the software from Experiment 11.6 into an evolutionary algorithm for evolving neural gate specifications, we must specify variation operators for connection-list specified neural nets. Crossover is not difficult. If we treat the triples (a_i, b_i, ω_i) that specify connections as atomic objects, then we can do crossover on the list as if it were a string, treating connections as characters.

There are two different types of objects inside a connection that should be modifiable by mutation. Thus, we need two distinct notions of point mutation. A *topological mutation* replaces the values of a_i and b_i with new, semantically correct values, i.e., $a_i < b_i$ and b_i is not an input neuron. A *weight mutation* is a real point mutation of the weight α of the parameter ω_i. We will use Gaussian real point mutations with mean zero and specified variance. These add a zero mean Gaussian to the weight ω. The formula for Gaussian random numbers is given in Equation 3.1.

Experiment 11.7 *Using the connection-list specified neural net routines from Experiment 11.6 and the variation operators described above, build or obtain software for an evolutionary algorithm to evolve neural logic gates. Use the fitness function, population size, and model of evolution from Experiment 11.1. Perform two point crossover of the list of connections and mutate each new gene 0-2 times with the number of mutations selected uniformly at random. Do a topological mutation one time in four and a weight mutation with a variance of 0.2 three times in four. Use 3 inputs and 6 non-input neurons with 20 connections.*

Run 100 populations for at most 400 generations, attempting to locate a 3-input AND and a 3-input parity gate. Compare time-to-solution data. Compute and record the average

```
//global definitions
int v[n];      //is neuron output i known?
double m[n];   //value of neuron output i, if known
int i;

[...]
   //prepare to call recursive net evaluator
   for(i=0;i<n;i++){
      v[i]=(i<inputs);
      if(v[i])m[i]=INPUT[i];else m[i]=-1;  //load known input values
   }
   return(receval(n-1));
[...]

int receval(int index){  //return the output value of neuron index

double scratch;
int i;

   if(v[index])return(m[index]);  //if value is known, return it
   //otherwise compute it
   scratch=0;
   for(i=0;i<index-1;i++)if(M[i,index]!=0)scratch+=M[i,index]*recval(i);
   v[index]=1;  //value now known
   if(scratch<alpha)m[index]=0;else m[index]=1;  //Heaviside computation
   return(m[index]);
}
```

Figure 11.7: Code for recursive evaluation of a neural net, given the connection matrix M ("Alpha" is the Heaviside threshold. "Inputs" is the number of input neurons.)

number of non-input neurons used *in correct solutions (computing the number of neurons used should be built into your evaluator).*

It may be that neural gates with fewer parameters are easier to locate. Rather than trying several different numbers of neurons, the next experiment employs lexical fitness to encourage the evolutionary algorithm to not use some of its neurons.

Experiment 11.8 *Modify the software from Experiment 11.7 to put "number of neurons used" in a lexical fitness function. For two gates that get the same number of positions correct in the output string, break ties by judging the gate that used fewer of its neurons more fit. Do the same simulations as in Experiment 11.7 and compare the results.*

It is possible that we will want a neural net with more than one output. It's not hard to code such an object, but we need to modify (a little) the coding of our connection-list specified neural nets and (somewhat more) our fitness functions. A common 2-output object is the 2-bit adder which, as its name suggests, adds 2 bits. The outputs are the sum and carry of the 2 bits. To make an adder for binary natural numbers, we also need to accept carry bits into the adder. The adders can then be cascaded to add any number of bits. The truth table for a binary adder is given in Figure 11.8.

Binary Adder with Carry In and Out				
Outputs		Inputs		
Carry	Sum	Bit1	Bit2	Carry
0	0	0	0	0
0	1	0	0	1
0	1	0	1	0
1	0	0	1	1
0	1	1	0	0
1	0	1	0	1
1	0	1	1	0
1	1	1	1	1

Figure 11.8: 2-bit adder with carry, input, and output

The structure of the connection list is modified as follows. The value of a_i may never be that of an output neuron. In the past, we had only one output neuron. The binary adder shown in Figure 11.8 has two output neurons. This means that the two largest neuron indices are off-limits for values of a_i, where before only the largest was.

The fitness function is modified by changing the definition of the output string. The binary adder has 3 inputs, and, so, has an 8-character output string, but this character string is over the alphabet $\{00, 01, 10, 11\}$. We will turn this into a 16-character output string by running through the sum bits, then the carry bits. That makes the output string for the binary adder a 16-character string, to wit: 0110100100010111. To generate the output string for an evolving gate, we generate, for the combinations of 3 inputs in lexicographical order, the string of sum-outputs followed by the string of carry-outputs. This is compared to the correct output string, and fitness is the number of bits in agreement.

Experiment 11.9 *Modify the software from Experiment 11.7 as outlined to permit the evolutionary algorithm to search for a binary adder with carry. Use the same simulation parameters as Experiment 11.7, save that there be 8 non-input neurons and 28 connections. Perform 100 simulations. Save solution times and compare to the difficulty of locating a 3-bit (3-input) parity gate.*

A *neutral mutation operator* is one that does not modify the fitness of an organism, but that does modify its gene. After a creature has undergone a neutral mutation, it may have different children than it otherwise would. For a connection-list specified neural net, there is a natural choice of neutral mutation. The connections appear in the list in some order, and that order is irrelevant to the functioning of the net. Our neutral mutation is some means of reordering the list of connections. In the tradition of keeping mutations small, we will implement neutral mutation by swapping a pair of list elements selected uniformly at random.

Experiment 11.10 *Modify the software from Experiment 11.7 to use neutral mutations half the time on new genes, in addition to and separate from other mutation. Using the same simulation parameters, perform 100 simulations. Save solution times and compare with the results not using the neutral mutation. Do a second set of simulations in which the rate of neutral mutations drops smoothly from 1.0 after the first generation to 0.0 in generation 200. Compare with the other simulations. Does neutral mutation help? Does it disrupt convergence near the end of evolution? Option: add to the evaluator the ability to compute the number of non-zero connections. Does neutral mutation affect this statistic?*

In the next section, we will compare neural nets to genetic programming as a method of performing logic function induction.

Problems

Problem 11.24 *Suppose we have n input and k non-input neurons. If there are no multiple or repeated connections, then what is the maximum number of connections possible in the connection-list scheme used in this section?*

Problem 11.25 *Suppose we have n inputs and k non-input neurons. If there are no multiple or repeated connections, then what is the maximum number of net connection topologies possible? Ignore weight values when answering the question.*

Problem 11.26 *Reread Problem 11.25. A* connection topology *is a list of connections with unknown weights. Find two connection topologies that are different, in the sense of Problem 11.25, but neither of which can be used to implement a logic gate the other cannot.*

Problem 11.27 Short Essay. *Reread problem 11.26. Call two topologies* different *if one can, by selecting weights, implement a logic gate in one that no selection of weights will implement in the other. Is it difficult to count the number of different connection topologies?*

Problem 11.28 *Compute the minimal number of Heaviside neurons, with threshold $\alpha = 0.5$, needed to implement a 3-input AND and a 3-input parity gate. Prove that your solutions are minimal.*

Problem 11.29 *On page 297, it is asserted that the restriction $a_i < b_i$ in a connection list forces the neural net to be feed-forward. Prove this assertion.*

Problem 11.30 *Examine the code given in Figure 11.7. When computing the output value of the neural net for a given set of inputs, does it evaluate the output value of every neuron? If so, explain why it needs to; if not, explain how the list of neurons it does use might be useful.*

Problem 11.31 *Compute the truth table of the logic gate given in Figure 11.6. Assume the neurons are Heaviside neurons with a threshold of $\alpha = -0.5$.*

Problem 11.32 *For the following list of connections, for 3 input and 3 non-input neurons, give the neural net which is coded and compute its truth table. Assume the neurons are Heaviside neurons with a threshold of $\alpha = -0.5$. Remember that the output of the gate is computed from the highest numbered neuron. The input neurons are numbered 0-2, the others 3-5.*

(a,b, ω)
(0,3,-0.41)
(3,5,-0.43)
(1,4,-0.32)
(2,4,-0.39)
(4,5,-0.44)
(1,3,-0.34)

Problem 11.33 *Give an algorithm that takes as its input a list of connections for specifying a neural net and outputs a reduced list of connections and the numbers of neurons actually used in the output. This reduced list should be minimal; explain why yours is. This type of software is called a target compiler - it takes a specification for a digital object, trims out the unneeded parts, and gives a minimal functional object as output.*

Problem 11.34 *Design, by hand, a feed-forward neural logic gate to perform the binary-adder-with-carry task given in Figure 11.8. Use as few neurons and connections as you can.*

Problem 11.35 Short Essay. *In the first two sections of this chapter we insist that our neural nets be feed-forward. Explain why.*

Problem 11.36 Essay. *In Experiment 11.7, we use a somewhat arbitrary ratio, 1:3, of topological to weight mutations. Explain the names of these mutations in terms of their functions. The evolutionary algorithm is searching both for a neural net connection topology and its weights. Of these two sorts of mutations, which one makes more changes in the output string, on average? Justify your answer mathematically or experimentally. Would varying the ratio of the two mutation types over evolutionary time be a good idea? Why?*

11.4 GP-Logics

In the last two sections, we have tried various neural network technologies to search for logic gates. Another evolutionary computation technique that is used to search for logic functions is genetic programming. There are pros and cons. Typically, we start with a complete set of logic operations as the foundation of the genetic programming environment. On the other hand, genetic programs are organized as trees and so have a fan-out for their operations of one. Fan-out is an electrical engineering term. In a circuit, the *fan-out* of a device is the number of inputs of other devices to which the output of the device is connected. Neuron 3 in Figure 11.6 has a fan-out of two, for example. One thing that helps with this problem is that the inputs to the logic gate are terminals and so have an arbitrarily large fan-out.

Before proceeding with this section, you should review Sections 8.1 and 8.2. We will need a genetic programming language that operates on logical variables with logical operators. The tree manipulation routines given in Section 8.2 will be needed, respecialized to a logical language. We will begin by specifying a logical GP-language and testing it. The logical language we will use is given in Table 11.3.

Experiment 11.11 *Write or obtain software for a parse-tree language that implements the logical GP-language given in Table 11.3. Make sure that the number of input variables in a given tree is easy to specify on the fly. Once the parse tree routines are written and debugged,*

Language Element	Type	Symbol	Semantics
True	terminal	**T**	constant - logical true
False	terminal	**F**	constant - logical false
x_i	terminal	x_i	input variable
NOT	unary operation	$\sim$	inverts argument
OR	binary operation	$+$	true, if either argument is true
AND	binary operation	$*$	true, if both arguments are true

Table 11.3: Logical language for genetic programming of logic gates

compute the output strings of 1000 random parse trees with 12 nodes and 3 input variables. Compute the empirical probability of AND and of 3-input parity, comparing the result with the results of Experiment 11.6.

Notice that parse trees form a drop-in replacement for the neural nets we used in the earlier sections of this chapter. The definition of "output string" and with it the fitness functions we used in the last chapter remain the same. As we have radically changed the coding used for the logic functions, the time to convergence for various logic functions, all other things being as close to equal as possible, may well be different. The next experiment checks this.

Experiment 11.12 *Using the parse tree routines from Experiment 11.11 and the variation operators described in Section 8.1, build an evolutionary algorithm to evolve parse-tree logic gates. Use the fitness function, population size, and model of evolution from Experiment 11.1. Use 3 input variables and trees with at most 16 nodes. Let initial trees have from 6-16 nodes with the number of nodes selected uniformly at random. Run 100 populations for at most 400 generations, attempting to locate a 3-input AND gate and a 3-input parity gate. Compare the time-to-solution of these two tasks, and, also, compare to the results obtained in Experiment 11.7. Option: try performing these experiments again* without *subtree crossover and see what the effect is on solution time.*

The next experiment mirrors the tension between topological mutation and weight mutation in the last section. Mutation of a GP-tree consists of deleting a subtree and replacing it with another. Another mutation operator sometimes used is a *leaf mutation*, which picks a terminal of the tree uniformly at random and replaces it with another terminal. In Problem 11.40, we ask you to suggest why leaf mutation might be helpful in getting a population to use cascading.

Experiment 11.13 *Modify the software from Experiment 11.12 to incorporate leaf mutation. Perform 100 simulations, modified as follows. Start trying to evolve a 2-input parity*

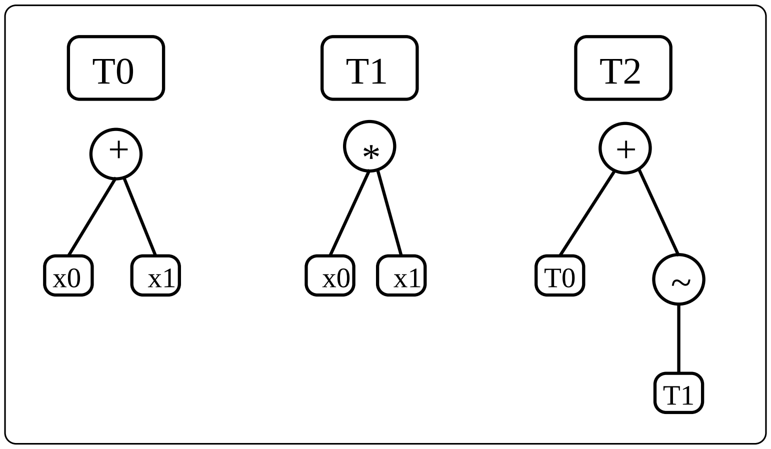

Figure 11.9: A 3-tree logic gate (The output of trees with smaller number is available as input terminals to trees with larger numbers.)

gate. 10 generations after a 2-input parity gate is found, modify the rate of the mutation operators from 100% subtree replacement to half-and-half subtree replacement and leaf mutation. At the same time, change the fitness function to search for 3-input parity. Save time-to-solution and compare with the results of Experiment 11.12.

Now we will introduce a multi-tree variation on the GP-tree approach that reduces the fan-out problem. It is based on an idea, MIPs nets or *Multiple Independent Program systems*, invented by Peter Angeline. An example of such a system is shown in Figure 11.9. In a MIPs net, multiple parse trees appear. The output of some trees are made available to other trees as input terminals. This is similar to the automatically defined functions (ADFs) used in other genetic programming systems, but there is no master-slave relationship between the trees.

We still want to have a feed-forward structure for searching. In a feed-forward MIPs net, we number the trees, and larger trees have access to the input of smaller trees. To do this, we will use "other tree" terminals which are numbered 0-255. In a given tree, number i, we take the index of the "other tree" terminal, mod i. This ensures that we will always access the output value of a tree with a smaller number and does not presume any particular number of trees. Tree0 will get a result of false from its other tree terminals. During evaluation, we evaluate Tree0 first. After evaluation, we know its output value, and this is then made available to the other trees. Tree1 is evaluated next, and so on, until the highest number tree is evaluated, and its output is the output value of the net.

A MIPs net does not have any particular number of trees, rather it has a limit on its total number of nodes. The modifications of our parse tree software needed to use MIPs nets are as follows. First, add a new type of terminal with 256 flavors, T_0-T_{255}. Second, change the basic structure to be a vector of pointers, large enough to permit a generous number of trees. We will require new mutation operators: add a new random tree from a list (*enlargement*), and delete one tree selected at random from a list of trees (*contraction*). MIPs nets do not use crossover, the system uses only 3 types of mutations.

Experiment 11.14 *Write or obtain software for an evolutionary algorithm to evolve MIPs nets as described. Assume a net has at most 24 nodes and at most 8 trees. When generating the initial population generate from 1-4 trees in each MIPs net with from 3-6 nodes in each tree. Use a steady-state evolutionary algorithm in which a pair of nets are selected and compared. A mutant of the better net is copied over the worse one. The output of a MIPs net is the output of its highest numbered tree; use agreement with the desired output string as the fitness function.*

Use 1-2 mutations on each new tree of which 25% pick a tree at random and do a standard GP-mutation, 25% pick a tree and do a leaf mutation, 25% are enlargement (add a tree the same size as one of the initial random trees) and 25% are contraction (delete one tree). When a MIPs net has 8 trees, perform contraction in place of enlargement. When a MIPs net has 1 tree, perform enlargement in place of contraction. When a MIPs net exceeds the total node boundary, perform contractions unless it has 1 tree with too many nodes, in which case use a chop operation.

Run 100 simulations on a population of 200 MIPs nets for at most 400 generations, looking for a 3-input parity gate. Compare with other experiments searching for a 3-input parity gate.

One of the mechanisms that biological evolution uses is to duplicate a gene, freeing one copy of that gene to change without disrupting critical biological function. Following some ideas from Chapter 10 on GP-automata deciders, we note that MIPS nets have the potential to exploit this type of evolutionary mechanism. Call this a *gene duplication* and implement it by picking a tree uniformly at random and copying it over another picked uniformly at random. Let us experimentally check the effect of such a mutation operator.

Experiment 11.15 *Modify the software from Experiment 11.14 to include gene duplication. Use the 4 mutation operators from Experiment 11.14 80% of the time and use gene duplication the remaining 20% of the time. Perform the same simulations and compare the time-to-solution with and without gene duplication.*

Problems

Problem 11.37 *Compute the number of 12-node parse trees in the language given in Table 11.3. Assume 3 input variables. You may want to look at Problem 8.8.*

Problem 11.38 *Prove or disprove: all 3-input logic functions can be realized with 12-node parse trees in the language given in Table 11.3. There are 256 of these, so you must use symmetries to represent entire classes of functions with single examples.*

Problem 11.39 *Compute the output string for each of the following parse trees.*

(i) $(+\ (\sim x_0)\ (+\ (\sim x_1)\ (\sim x_2)))$,

(ii) $(+\ (*\ (\sim x_0)\ (*\ x_1\ (\sim x_2)))\ (*\ x_0\ (*\ (\sim x_1)\ (\sim x_2))))$,

(iii) $(+\ (+\ x_1\ x_0)\ (\sim (+\ x_0\ x_1)))$, and

(iv) $(+\ (*\ (\sim x_2)\ (*\ x_0\ (\sim x_1)))\ (*\ x_1\ (*\ (\sim x_2)\ (\sim x_0))))$.

Problem 11.40 *Show, pictorially, how GP-crossover makes it easier to discover cascades of gates. Discuss, in a few sentences, the problem with getting the terminals right. A cascade of AND gates is given in Figure 11.3.*

Problem 11.41 *Construct, by hand, a GP-tree logic gate to compute the logic function with output string: 0110100110010110.*

Problem 11.42 *What logic function does the MIPs net given in Figure 11.9 compute? Give the truth table and name it if it has a name.*

Problem 11.43 *In Figure 11.8, a 2-output logic gate (a binary adder with carry in and out) is described. In a few sentences, explain how to use a MIPs net to code for multiple output logic gates, and illustrate your technique by constructing a MIPs net that computes the binary adder function.*

Problem 11.44 *When doing genetic programming, there is no need to limit ourselves to binary operations. Both the AND and OR functions are defined for any number of inputs. Describe how to modify the GP-software used in this section to accommodate arbitrary-input AND and OR operations. Be sure to explain how mutation (or crossover) can change the number of arguments a given operator uses.*

Problem 11.45 *If we use AND, OR, and NOT then what is the minimum number of operations needed to compute (i) 3-input parity, (ii) 4-input parity, and (iii) 5-input parity.*

Problem 11.46 *We can change the function computed by a GP-logic tree by modifying its terminals. True or false: if two GP-trees have the same operations (possibly connected differently), then each has the same number of functions it could code for under various modifications of its terminals. In essence, you are asked to establish if some trees are inherently more diverse than others. Prove your answer.*

Problem 11.47 Essay. *Give the design of an experiment used to compute the optimum ratio of leaf mutations to subtree mutations when evolving GP-logic trees as in Experiment 11.13.*

Problem 11.48 *In Chapter 10, we discovered that storing internal state information was key to performing well on the Tartarus task. Suppose we modify MIPs nets evaluation so that the output value of each tree in the last time-step is stored and is the value of the terminal referring to that tree in the current time-step. In essence, this is a one time-step delay. Can a MIPs net evaluated in this manner store internal state information? Justify your answer in a few sentences.*