*It would appear that we have reached the limits of what it is possible to achieve with computer technology, although one should be careful with such statements, as they tend to sound pretty silly in 5 years time.*

*—John Von Neumann 1949*

As you have learned, the architecture—or topology—of a neural network plays an important role in how effective it is. You've also learned that choosing the parameters for that architecture is more of an art than a science and usually involves an awful lot of hands-on tweaking. Although you can develop a "feel" for this, wouldn't it be great if your networks *evolved* to find the best topology along with the network weights? A network that is simple enough to learn whatever it is you want it to learn, yet not so simple that it loses its ability to generalize?

When using an evolutionary algorithm to evolve neural network topology, we can imagine an undulating fitness landscape where each point in search space represents a certain type of architecture. The goal of an EANN (Evolutionary Artificial Neural Network), therefore, is to traverse that landscape as best it can before alighting upon the global optima.

**NOTE**

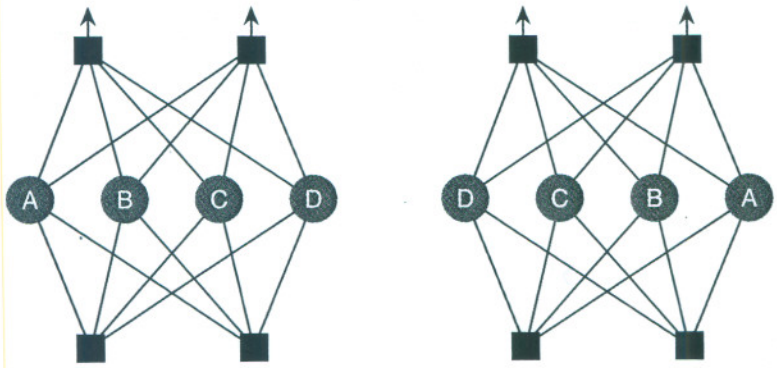
**This problem has been tackled in a few *non*-evolutionary ways. Researchers have attempted to create networks either *constructively* or *destructively*. A destructive algorithm commences with an oversized ANN with many neurons, layers, and links and attempts to reduce its size by systematically pruning the network during the training process. A constructive process is one that approaches the problem from the opposite end, by starting with a minimal network and adding neurons and links during training. However, these methods have been found to be prone to converging upon local optima and, what's more, they are still usually fairly restrictive in terms of network architecture. That is to say, only a fraction of the full spectrum of possible topologies is usually available for these techniques to explore.**

A fair amount of time and thought has been put into this problem by a number of different researchers, and I'm going to spend the first part of this chapter describing some of the many techniques available. The second part of the chapter will be spent describing a simple implementation of what I consider to be one of the better methods.

As with every other problem tackled with evolutionary algorithms, any potential solution has to figure out a way of encoding the networks, a way of assigning fitness scores, and valid operators for performing genome mutation and/or crossover. I say *or* crossover because a few methods dispose with this potentially troublesome operator altogether, preferring to rely entirely on mutation operators to navigate the search space. So before I describe some of the popular EANNs, let me show you why this operator can be so problematic.

# The Competing Conventions Problem

One of the main difficulties with encoding candidate networks is called the *competing conventions* problem—sometimes referred to as the structural-functional mapping problem. Simply put, this is where a system of encoding may provide several different ways of encoding networks that exhibit identical functionality. For example, imagine a simple encoding scheme where a network is encoded as the order in which the hidden neurons appear in a layer. Figure 11.1 shows a couple of examples of simple networks.




Network 1 Network 2

**Figure 11.1**
*A simple encoding scheme.*

Using the simple scheme I've just proposed, Network 1 may be encoded as:

**A B C D**

and Network 2 as:

**D C B A**

If you look carefully, you'll notice that, although the order of the neurons is different—and therefore the genomes are different—both networks are essentially identical. They will both exhibit exactly the same behavior. And this is where the problem lies, because if you now attempt to apply a crossover operator to these two networks, lets say at the midpoint of the genome, the resultant offspring will be:

**A B B A** or **D C C D**

This is an undesirable result because not only have both offspring inherited duplicated neurons, they have also lost 50% of the functionality of their parents and are unlikely to show a performance improvement. (Even if one of them did go on to produce such '70s classics as *Super Trooper* and *Dancing Queen.* <smile>).

Obviously, the larger the networks are, the more frequently this problem is encountered. And this results in a more negative effect on the population of genomes. Consequently, it is a problem researchers do their best to avoid when designing an encoding scheme.

**NOTE**

**There is an alternative camp of opinion to the competing convention problem. Some researchers believe that any steps taken to avoid this problem may actually create more problems. They feel it's preferable to simply ignore the problem and allow the evolutionary process to handle the disposal of the "handicapped" networks, or to ditch the crossover operator altogether and rely entirely on mutation to traverse the search space.**

# Direct Encoding

There are two methodologies of EANN encoding: *direct* encoding and *indirect* encoding. The former attempts to specify the exact structure of the network by encoding the number of neurons, number of connections, and so on, directly into the genome. The latter makes use of growth rules, which may even define the network structure recursively. I'll be discussing those in a moment, but first let's take a look at some examples of direct encoding.

## GENITOR

GENITOR is one of the simplest techniques to be found and is also one of the earliest. A typical version of this algorithm encodes the genome as a bit string. Each gene is encoded with nine bits. The first bit indicates whether there is a connection

between neurons and the rest represent the weight (-127 to 127). Given the network shown in Figure 11.2, the genome is encoded as:

**1**10010000 **0**00000010 **1**01000011 **0**00000101 **1**10000011

Where the bit in bold is the connectivity bit.

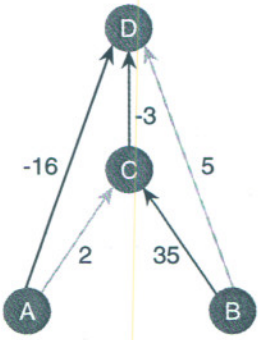


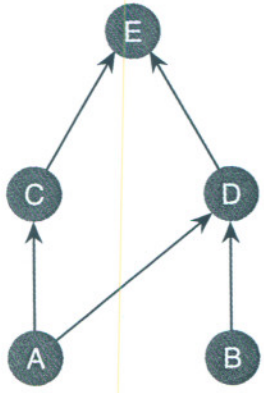

**Figure 11.2**

*GENITOR encoding. The light gray connectivity lines indicate disabled connections.*

The disadvantage of this technique, as with many of the encoding techniques developed to date, is that a maximal network topology must be designed for each problem addressed in order for all the potential connectivity to be represented within the genome. Additionally, this type of encoding will suffer from the competing conventions problem.

## Binary Matrix Encoding

One popular method of direct encoding is to use a *binary adjacency matrix*. As an example, take a look at the network shown in Figure 11.3.




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 1 | 0 |
| B | 0 | 0 | 0 | 1 | 0 |
| C | 0 | 0 | 0 | 0 | 1 |
| D | 0 | 0 | 0 | 0 | 1 |
| E | 0 | 0 | 0 | 0 | 0 |

**Figure 11.3**

*Binary matrix representation for a simple 5-node network*

As you can see, the connectivity for this network can be represented as a matrix of binary digits, where a 1 represents a connection between neurons and a 0 signifies no connection. The chromosome can then be encoded by just assigning each row (or column) of the matrix to a gene. Like so:

00110 00010 00001 00001 00000

However, because the network shown is entirely feedforward, this encoding is wasteful because half the matrix will always contain zeros. Realizing this, we can dispose of one-half of the matrix, as shown in Figure 11.4, and encode the chromosome as:

0110 010 01 1

which, I'm sure you will agree, is much more efficient!

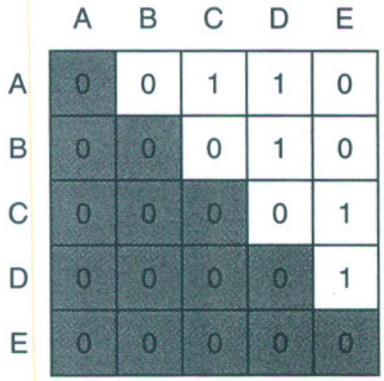
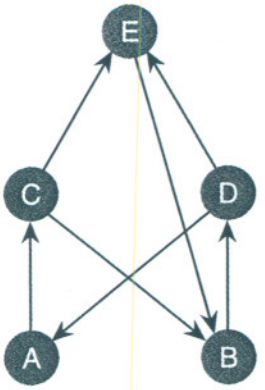



**Figure 11.4**

*The adjusted matrix.*

Once encoded, the bit strings may be run through a genetic algorithm to evolve the topologies. Each generation, the chromosomes are decoded and the resultant networks initialized with random weights. The networks are then trained and a fitness is assigned. If, for example, backprop is used as the training mechanism, the fitness function could be proportional to the error generated, with an additional penalty as the number of connections increases in order to keep the network size at a minimum.

Obviously, if your training approach can handle any form of connectivity, not just feedforward, then the entire matrix may be represented. Figure 11.5 shows an example of this. A genetic algorithm training approach would be okay with this type of network, but standard backpropagation would not.




| | A | B | C | D | E |
|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 0 | 0 |
| B | 0 | 0 | 0 | 1 | 0 |
| C | 0 | 1 | 0 | 0 | 1 |
| D | 1 | 0 | 0 | 0 | 1 |
| E | 0 | 1 | 0 | 0 | 0 |

**Figure 11.5**
*Network with recurrent connectivity.*

## Some Related Problems

It has been demonstrated that when using matrix encoding (and some other forms of direct encoding), performance deteriorates as the size of the chromosome increases. Because the size increases in proportion to the square of the number of neurons, performance deteriorates pretty quickly. This is known as the *scalability* problem. Also, the user still has to decide how many neurons will make up the maximal architecture before the matrix can be created. In addition, this type of representation does not address the competing conventions problem discussed earlier. It's very likely, when using this encoding, that two or more chromosomes may display the same functionality. If these chromosomes are then mated, the resultant offspring has little chance of being fitter than either parent. For this reason, it's quite common for the crossover operator to be dismissed altogether with this technique.

# Node-Based Encoding

Node-based encoding tackles the problem by encoding all the required information about each neuron in a single gene. For each neuron (or node), its gene will contain information about the other neurons it is connected to and/or the weights associated with those connections. Some node-based encoding schemes even go so far as to specify an associated activation function and learning rate. (A learning rate, don't forget, is used when the network is trained using a gradient descent method like backpropagation.)

Because the code project for this chapter uses node-based encoding, I'll be discussing this technique in a lot more detail later on, but for now, just so you get the idea, let's look at a simple example that encodes just the connectivity of a network.

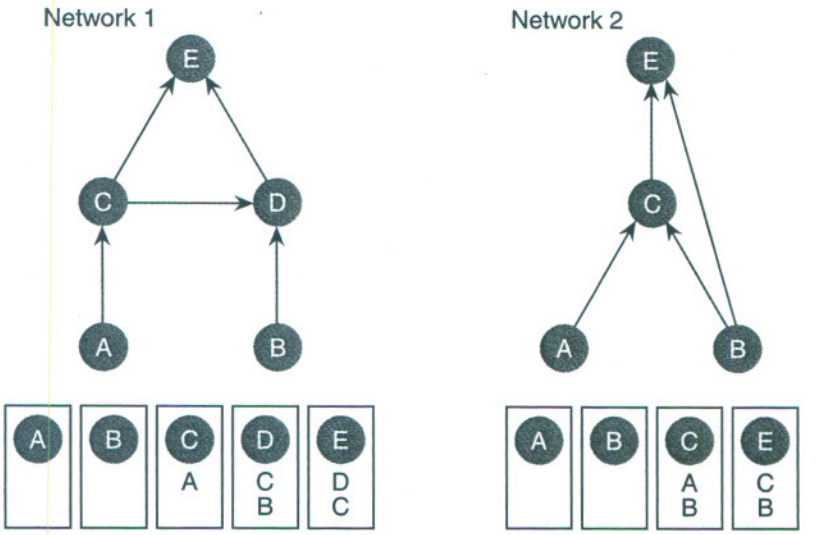



**Figure 11.6**
*Node-based encoding.*

Figure 11.6 shows two simple networks and their chromosomes. Each gene contains a node identifier and a list of incoming connections. In code, a simplified gene and genome structure would look something like this:

```
struct SGene
{
  int             NodeID;

  vector<Node*>   vecpNodes;
}

struct SGenome
{
  vector<SGene> chromosome;

  double        fitness;
};
```

Mutation operators using this sort of encoding can be varied and are simple to implement. They include such mutations as adding a link, removing a link, adding a node, or removing a node. The crossover operator, however, is a different beast altogether. Care must be taken to ensure valid offspring are produced and that neurons are not left stranded without any incoming and outgoing connections. Figure 11.7 shows the resultant offspring if the two chromosomes from Figure 11.6 are mated after the third gene (the "C" gene).

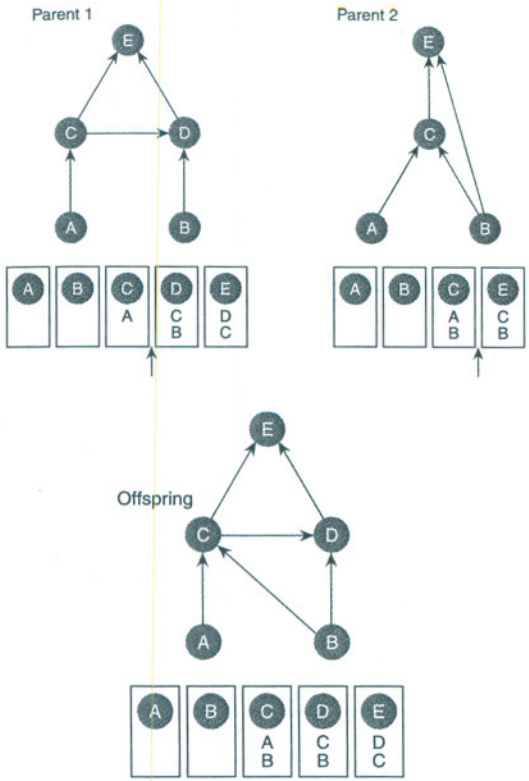



**Figure 11.7**

*Crossover in action.*

Once valid genetic algorithm operators have been defined, the neural networks encoded using the described scheme may be evolved as follows (assuming they are trained using a training set in conjunction with a gradient descent algorithm like backpropagation):

1. Create an initial random population of chromosomes.
2. Train the networks and assign a fitness score based on the overall error value of each network (target output – best trained output). It is also feasible to penalize the score as the networks grow in size. This will favor populations with fewer neurons and links.
3. Choose two parents using your favorite selection technique (fitness proportionate, tournament, and so on).
4. Use the crossover operator where appropriate.
5. Use the mutation operator/s where appropriate.
6. Repeat Steps 3,4, and 5 until a new population is created.
7. Go to Step 2 and keep repeating until a satisfactory network is evolved.

Later in the chapter, I'll be showing you how to use node-based encoding to evolve the topology *and* the connection weights at the same time.

## Path-Based Encoding

Path-based encoding defines the structure of a neural network by encoding the routes from each input neuron to each output neuron. For example, given the network described by Figure 11.8, the paths are:

$$1 \rightarrow A \rightarrow C \rightarrow 3$$

$$1 \rightarrow D \rightarrow B \rightarrow 4$$

$$1 \rightarrow D \rightarrow C \rightarrow 3$$

$$2 \rightarrow D \rightarrow C \rightarrow 3$$

$$2 \rightarrow D \rightarrow B \rightarrow 4$$

Because each path always begins with an input neuron and always ends with an output neuron, this type of encoding guarantees there are no useless neurons referred to in the chromosome. The operator used for recombination is two-point crossover. (This ensures the chromosomes are always bound with an input and output neuron). Several mutation operators are typically used:

- Create a new path and insert into the chromosome.
- Choose a section of path and delete.
- Select a path segment and insert a neuron.
- Select a path segment and remove a neuron.

Because the networks defined by this type of encoding are not restricted to feedforward networks (links can be recurrent), a training approach such as genetic algorithms must be used to determine the ideal connection weights.

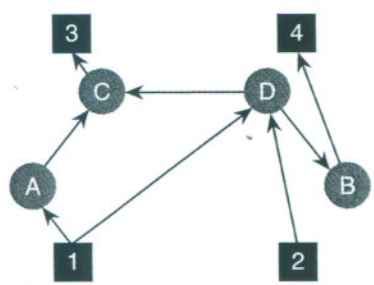



**Figure 11.8**

*Path-based encoding.*

# Indirect Encoding

Indirect encoding methods more closely mimic the way genotypes are mapped to phenotypes in biological systems and typically result in more compact genomes. Each gene in a biological organism does not give rise to a single physical feature; rather, the interactions between different permutations of genes are expressed. Indirect encoding techniques try to emulate this mechanism by applying a series of *growth rules* to a chromosome. These rules often specify many connections simultaneously and may even be applied recursively. Let's take a look at a couple of these techniques, so you get a feel for how they can work.

## Grammar-Based Encoding

This type of encoding uses a series of developmental rules that can be expressed as a type of grammar. The grammar consists of a series of left-hand side symbols (LHS) and right-hand side symbols (RHS). Whenever a LHS symbol is seen by the development process, it's replaced by a number of RHS symbols. The development process starts off with a *start symbol* (a LHS symbol) and uses one of the production rules to create a new set of symbols. Production rules are then applied to these symbols until a set of *terminal symbols* has been reached. At this point, the development process stops and the terminal symbols are expressed as a phenotype.

If you're anything like me, that last paragraph probably sounded like gobbledygook! This is a difficult idea to understand at first, and it's best illustrated with diagrams. Take a look at Figure 11.9, which shows an example of a set of production rules.

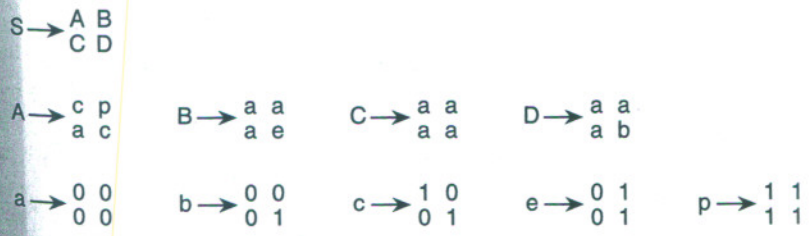



**Figure 11.9**

*Example production rules for grammar-based encoding.*

The **S** is the start symbol and the **1**s and **0**s are terminal symbols. Now examine Figure 11.10 to see how these rules are used to replace the start symbol **S** with more symbols in the grammar, and then how these symbols in turn are replaced by more symbols until the terminal symbols have been reached. As you can clearly see, what we have ended up with is a binary matrix from which a phenotype can be constructed. Cool, huh?

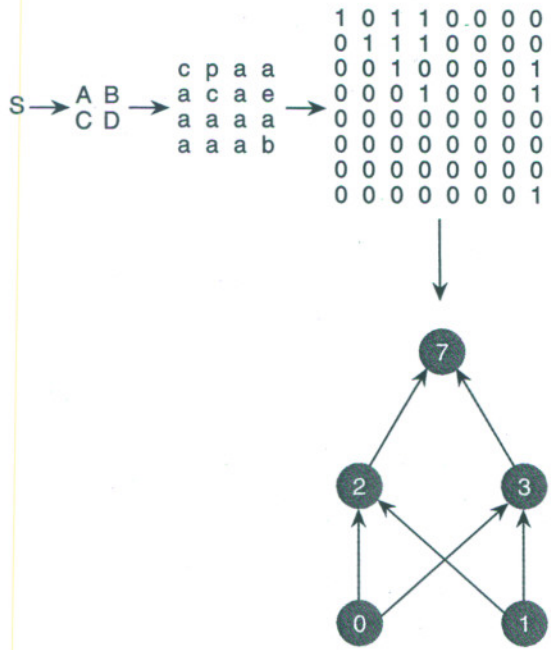



**Figure 11.10**

*Following the growth rules.*

A genetic algorithm is used to evolve the growth rules. Each rule can be expressed in the chromosome by four positions corresponding to the four symbols in the RHS of the rule. The actual position (its loci) of the rule along the length of the chromosome determines its LHS. The number of non-terminal symbols can be in any range. The inventors of this technique used the symbols **A** through **Z** and **a** through **p.** The rules that had terminal symbols as their RHS were predefined, so the chromosome only had to encode the rules consisting of non-terminal symbols. Therefore, the chromosome for the example shown in Figure 11.10 would be:

**ABCD cpac aaae aaaa aaab**

where the first four positions correspond to the start symbol **S**, the second four to the LHS symbol **A,** and so on.

## Bi-Dimensional Growth Encoding

This is a rather unusual type of encoding. The neurons are represented by having a fixed position in two-dimensional space, and the algorithm uses rules to actually *grow* axons, like tendrils reaching through the space. A connection is made when an axon touches another neuron. This is definitely a method best illustrated with a diagram, so take a look at Figure 11.11 to see what's going on.

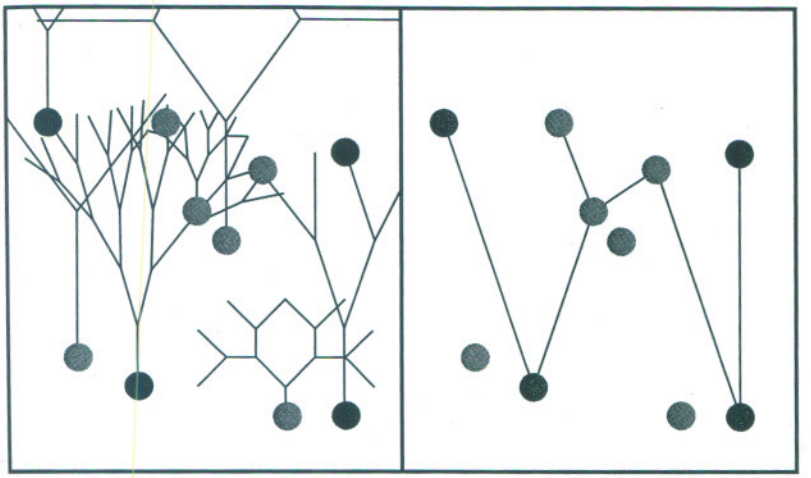


**Figure 11.11**

*Axons growing outward from neurons located in 2D space.*

The left-hand side of Figure 11.11 shows the neurons with all their axons growing outward, and the right-hand side shows where connections have been established.

The designers of this technique use a genome encoding which consists of 40 blocks, each representing a neuron. There are five blocks at the beginning of the genome to represent input neurons, five at the end to represent output neurons, and the remaining thirty are used as hidden neurons.

Each block has eight genes.

- Gene1 determines if the neuron is present or not.
- Gene2 is the X position of the neuron in 2D space.
- Gene3 is the Y position.
- Gene4 is the branching angle of the axon growth rule. Each time the axon divides, it divides using this angle.
- Gene5 is the segment length of each axon.
- Gene6 is the connection weight.
- Gene7 is the bias.
- Gene8 is a neuron type gene. This gene in the original experiment was used to determine which input the input neuron represented.

As you can imagine, this technique is tricky to implement and also pretty slow to evolve. So, although it's interesting, it's not really of much practical use.

And that ends your whistle-stop tour of encoding techniques. Next, I'll show you a fantastic way of using node-based encoding to grow your networks from scratch.

# NEAT

NEAT is short for *Neuro Evolution of Augmenting Topologies* and has been developed by Kenneth Stanley Owen and Risto Miikkulainen at the University of Texas. It uses node-based encoding to describe the network structure and connection weights, and has a nifty way of avoiding the competing convention problem by utilizing the historical data generated when new nodes and links are created. NEAT also attempts to keep the size of the networks it produces to a minimum by starting the evolution using a population of networks of minimal topology and adding neurons and connections throughout the run. Because nature works in this way—by increasing the complexity of organisms over time—this is an attractive solution and is partly the reason I've chosen to highlight this technique in this chapter.

There's quite a bit of source code required to implement this concept, so the related code is listed as I describe each part of the NEAT paradigm. This way (if I do it in the proper order <smile>), the source will help to reinforce the textual explanations and help you to grasp the concepts quickly. You can find all the source code for this chapter in the Chapter11/NEAT Sweepers folder on the CD.

First, let me describe how the networks are encoded.

## The NEAT Genome

The NEAT genome structure contains a list of *neuron genes* and a list of *link genes*. A link gene, as you may have guessed, contains information about the two neurons it is connected to, the weight attached to that connection, a flag to indicate whether the link is enabled, a flag to indicate if the link is recurrent, and an *innovation* number (more on this in a moment). A neuron gene describes that neuron's function within the network—whether it be an input neuron, an output neuron, a hidden neuron, or a bias neuron. Each neuron gene also possesses a unique identification number.

Figure 11.12 shows the gene lists for a genome describing a simple network.

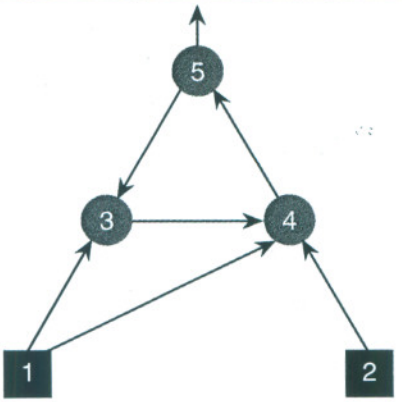
### SLinkGene

The link gene structure is called `SLinkGene` and can be found in genes.h. Its definition is listed here:

```
struct SLinkGene
{
  //the IDs of the two neurons this link connects
```

| Weight: 1.2 | Weight: -3 | Weight: 0.7 | Weight: -2.1 | Weight: 1.1 | Weight: 0.8 | Weight: -1 |
|---|---|---|---|---|---|---|
| From: 1 | From: 1 | From: 2 | From: 3 | From: 3 | From: 4 | From: 5 |
| To: 3 | To: 4 | To: 4 | To: 4 | To: 5 | To: 5 | To: 3 |
| Enabled: Y | Enabled: Y | Enabled: Y | Enabled: Y | Enabled: N | Enabled: Y | Enabled: Y |
| Recurrent: N | Recurrent: N | Recurrent: N | Recurrent: N | Recurrent: N | Recurrent: N | Recurrent: Y |
| Innovation: 1 | Innovation: 6 | Innovation: 2 | Innovation: 8 | Innovation: 3 | Innovation: 4 | Innovation: 7 |

Link Genes




| ID: 1 | ID: 2 | ID: 3 | ID: 4 | ID: 5 |
|---|---|---|---|---|
| Type: input | Type: input | Type: hidden | Type: hidden | Type: output |

Neuron Genes

**Figure 11.12**

*Encoding a network the NEAT way.*

```
int     FromNeuron,
        ToNeuron;

double  dWeight;

//flag to indicate if this link is currently enabled or not
bool    bEnabled;

//flag to indicate if this link is recurrent or not
bool    bRecurrent;

//I'll be telling you all about this value shortly
int     InnovationID;

SLinkGene(){}

SLinkGene(int    in,
          int    out,
```

```
            bool   enable,
            int    tag,
            double w,
            bool   rec = false):bEnabled(enable),
                                InnovationID(tag),
                                FromNeuron(in),
                                ToNeuron(out),
                                dWeight(w),
                                bRecurrent(rec)
  {}

  //overload '<' used for sorting(we use the innovation ID as the criteria)
  friend bool operator<(const SLinkGene& lhs, const SLinkGene& rhs)
  {
    return (lhs.InnovationID < rhs.InnovationID);
  }
};
```

## SNeuronGene

The neuron gene structure is called `SNeuronGene` and is found in genes.h. Here is its definition:

```
struct SNeuronGene
{
  //its identification number
  int       iID;

  //its type
  neuron_type NeuronType;
```

This is an enumerated type. The values are `input`, `hidden`, `bias`, `output`, and `none`. You will see how the `none` type is used when I discuss innovations in the next section.

```
  //is it recurrent?
  bool      bRecurrent;
```

A *recurrent neuron* is defined in NEAT as a neuron with a connection that loops back on itself. See Figure 11.13

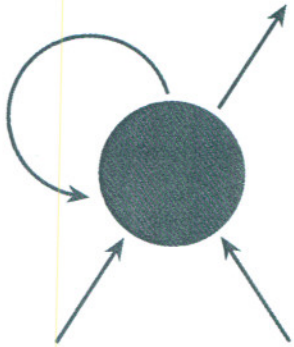


**Figure 11.13**

*A neuron with two incoming links: an outgoing link and a looped recurrent link.*

```
//sets the curvature of the sigmoid function
double    dActivationResponse;
```

In this implementation, the sigmoid function's activation response is also evolved separately for each neuron.

```
//position in network grid
double    dSplitY, dSplitX;
```

If you imagine a neural network laid out on a 2D grid, it's useful to know the coordinates of each neuron on that grid. Among other things, this information can be used to render the network to the display as a visual aid for the user.

When a genome is first constructed, all the neurons are assigned a `SplitX` and a `SplitY` value. I'll just stick to discussing the `SplitY` value for now, but the `SplitX` value is calculated in a similar way. Each input neuron is assigned a `SplitY` value of 0 and each output neuron a value of 1. When a neuron is added, it effectively splits a link, and so the new neuron is assigned a `SplitY` value halfway between its two neighbors. Figure 11.14 should help clarify this.

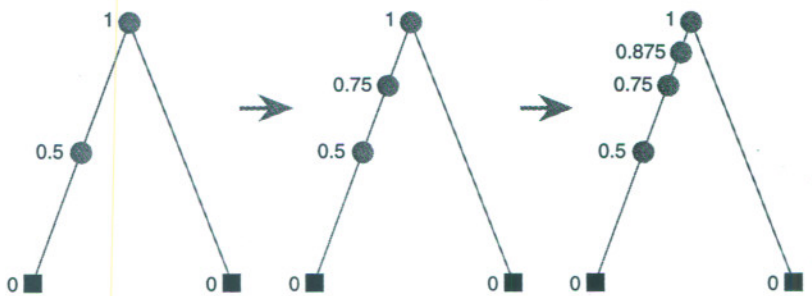



**Figure 11.14**

*Some example* `SplitY` *depths.*

As well as being used to calculate the display coordinates for the network render routine, this information is also invaluable for calculating the overall network depth and for determining if a newly created link is recurrent.

```
  SNeuronGene(neuron_type type,
              int         id,
              double      y,
              double      x,
              bool        r = false):iID(id),
                                     NeuronType(type),
                                     bRecurrent(r),
                                     pNeuronMarker(NULL),
                                     dSplitY(y),
                                     dSplitX(x)
  {}
};
```

## CGenome

Here's the definition of the genome class. There will be some methods and members you will not understand the purpose of just yet, but just take a quick glance at the class for now and move onto the next section.

(Please note, I have omitted the accessor methods for the sake of brevity).

```
class CGenome
{

private:

  //its identification number
  int                    m_GenomeID;

  //all the neurons which make up this genome
  vector<SNeuronGene>    m_vecNeurons;

  //and all the links
  vector<SLinkGene>      m_vecLinks;

  //pointer to its phenotype
```

```
  CNeuralNet*             m_pPhenotype;

  //its raw fitness score
  double                  m_dFitness;

  //its fitness score after it has been placed into a
  //species and adjusted accordingly
  double                  m_dAdjustedFitness;

  //the number of offspring this individual is required to spawn
  //for the next generation
  double                  m_dAmountToSpawn;

  //keep a record of the number of inputs and outputs
  int                     m_iNumInputs,
                          m_iNumOutPuts;

  //keeps a track of which species this genome is in (only used
  //for display purposes)
  int                     m_iSpecies;

  //returns true if the specified link is already part of the genome
  bool    DuplicateLink(int NeuronIn, int NeuronOut);

  //given a neuron id this function just finds its position in
  //m_vecNeurons
  int     GetElementPos(int neuron_id);

  //tests if the passed ID is the same as any existing neuron IDs. Used
  //in AddNeuron
  bool    AlreadyHaveThisNeuronID(const int ID);


public:

  CGenome();

  //this constructor creates a minimal genome where there are output &
  //input neurons and every input neuron is connected to each output neuron
```

```
CGenome(int id, int inputs, int outputs);

//this constructor creates a genome from a vector of SLinkGenes
//a vector of SNeuronGenes and an ID number
CGenome(int                     id,
        vector<SNeuronGene>     neurons,
        vector<SLinkGene>       genes,
        int                     inputs,
        int                     outputs);

~CGenome();

//copy constructor
CGenome(const CGenome& g);

//assignment operator
CGenome& operator =(const CGenome& g);

//create a neural network from the genome
CNeuralNet*         CreatePhenotype(int depth);

//delete the neural network
void                DeletePhenotype();

//add a link to the genome dependent upon the mutation rate
void                AddLink(double      MutationRate,
                            double      ChanceOfRecurrent,
                            CInnovation &innovation,
                            int         NumTrysToFindLoop,
                            int         NumTrysToAddLink);


//and a neuron
void                AddNeuron(double      MutationRate,
                              CInnovation &innovation,
                              int         NumTrysToFindOldLink);


//this function mutates the connection weights
void                MutateWeights(double  mut_rate,
```

```
                                    double  prob_new_mut,
                                    double  dMaxPertubation);

  //perturbs the activation responses of the neurons
  void              MutateActivationResponse(double mut_rate,
                                             double MaxPertubation);

  //calculates the compatibility score between this genome and
  //another genome
  double            GetCompatibilityScore(const CGenome &genome);

  void              SortGenes();

  //overload '<' used for sorting. From fittest to poorest.
  friend bool operator<(const CGenome& lhs, const CGenome& rhs)
  {
    return (lhs.m_dFitness > rhs.m_dFitness);
  }
};
```

## Operators and Innovations

Now that you've seen how a network structure is encoded, let's have a look at the ways a genome may be mutated. There are four mutation operators in use in this implementation of NEAT: a mutation to add a link gene to the genome, a mutation to add a neuron gene, a mutation for perturbing the connection weights, and a mutation that can alter the response curve of the activation function for each neuron. The connection weight mutation works very similarly to the mutation operators you've seen in the rest of the book, so I'll not show you the code. It simply steps through the connection weights and perturbs each one within predefined limits based on a mutation rate. There is one difference however, this time there is a probability the weight is *replaced with a completely new weight*. The chance of this occurring is set by the parameter `dProbabilityWeightReplaced`.

An *innovation* occurs whenever new structure is added to a genome, either by adding a link gene or by adding a neuron gene, and is simply a record of that change. A global database of all the innovations is maintained—each innovation having its own unique identification number. Each time a link or neuron addition occurs, the database is referenced to see if that innovation has been previously

created. If it has, then the new gene is assigned the existing innovation ID number. If not, a new innovation is created, added to the database, and the gene is tagged with the newly created innovation ID.

As an example, imagine you are evolving a network that has two inputs and one output. The network on the left of Figure 11.15 describes the basic structure each member of the population possesses at the commencement of the run. The network on the right shows the result of a mutation that adds a neuron to the network. When neuron 4 is added, three innovations are created: an innovation for the neuron, and innovations for each of the new connections between neurons 1-4 and 4-3. (The old link gene between neurons 1 and 3 still exists in the genome, but it is disabled).

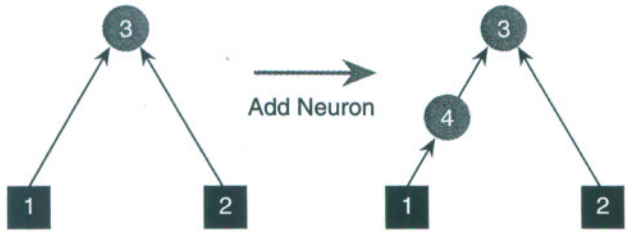



**Figure 11.15**

*Mutation to add a neuron.*

Each innovation is recorded in a `SInnovation` structure. The definition of this structure looks like this:

```
struct SInnovation
{
  //new neuron or new link?
  innov_type  InnovationType;

  int         InnovationID;

  int         NeuronIn;
  int         NeuronOut;

  int         NeuronID;

  neuron_type NeuronType;

  /*constructors and extraneous members omitted*/
};
```

The innovation type can be either `new_neuron` or `new_link`. You can find the definitions for `SInnovation` and the class `CInnovation`, which keeps track of all the innovations, in the file CInnovation.h.

Because NEAT grows structure by adding neurons and links, all the genomes in the initial population start off representing identical minimal topologies (but with different connection weights). When the genomes are created, the program automatically defines innovations for all the starting neurons and connections. As a result, the innovation database prior to the mutation shown in Figure11.15 will look a little like Table 11.1.

Input and output neurons are assigned a value of -1 for the in and out values to avoid confusion. Similarly, new links are assigned a neuron ID of -1 (because they're not neurons! <smile>).

After the addition of neuron 4, shown in Figure 11.15, the innovation database will have grown to include the new innovations shown in Table 11.2.

If at any time in the future a different genome stumbles across this identical mutation (adding neuron number 4), the innovation database is referenced and the correct innovation ID is assigned to the newly created gene. In this way, the genes contain a historical record of any structural changes. This information is invaluable for designing a valid crossover operator, as you shall see shortly.

Let me take you through the code for the `AddLink` and `AddNeuron` mutation operators.

**Table 11.1 Innovations Before the Neuron Addition**

| Innovation ID | Type | In | Out | Neuron ID | Neuron Type |
|---|---|---|---|---|---|
| 1 | new_neuron | -1 | -1 | 1 | input |
| 2 | new_neuron | -1 | -1 | 2 | input |
| 3 | new_neuron | -1 | -1 | 3 | output |
| 4 | new_link | 1 | 3 | -1 | none |
| 5 | new_link | 2 | 3 | -1 | none |

**Table 11.2 Innovations After the Neuron Addition**

| Innovation ID | Type | In | Out | Neuron ID | Neuron Type |
|---|---|---|---|---|---|
| 1 | new_neuron | -1 | -1 | 1 | input |
| 2 | new_neuron | -1 | -1 | 2 | input |
| 3 | new_neuron | -1 | -1 | 3 | output |
| 4 | new_link | 1 | 3 | -1 | none |
| 5 | new_link | 2 | 3 | -1 | none |
| 6 | new_neuron | 1 | 3 | 4 | hidden |
| 7 | new_link | 1 | 4 | -1 | none |
| 8 | new_link | 4 | 3 | -1 | none |

## CGenome::AddLink

This operator adds one of three different kinds of links:

- A forward link
- A recurrent link
- A looped recurrent link

Figure 11.16 shows an example of each type of link.

Here's the code for adding links to genomes. I've added additional comments where necessary.

```
void CGenome::AddLink(double       MutationRate,
                      double       ChanceOfLooped,
                      CInnovation  &innovation, //the database of innovations
                      int          NumTrysToFindLoop,
                      int          NumTrysToAddLink)
{
  //just return dependent on the mutation rate
  if (RandFloat() > MutationRate) return;

  //define holders for the two neurons to be linked. If we find two
  //valid neurons to link these values will become >= 0.
```

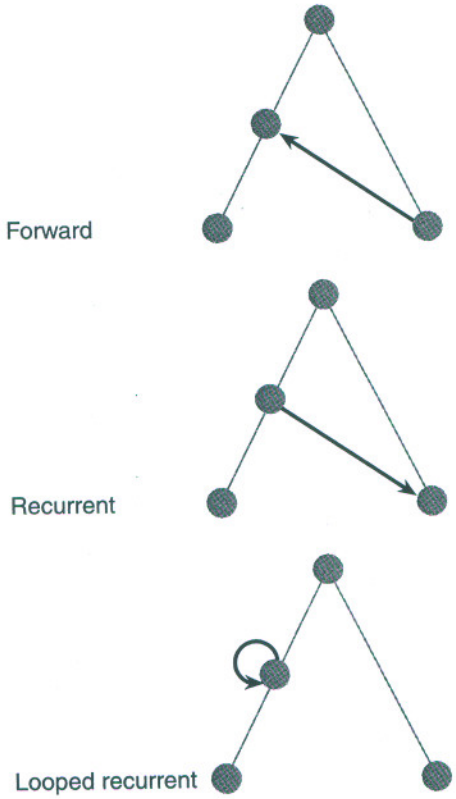



**Figure 11.16**
*Different types of links.*

```
int ID_neuron1 = -1;
int ID_neuron2 = -1;

//flag set if a recurrent link is selected to be added
bool bRecurrent = false;

//first test to see if an attempt should be made to create a
//link that loops back into the same neuron
if (RandFloat() < ChanceOfLooped)
{
  //YES: try NumTrysToFindLoop times to find a neuron that is not an
  //input or bias neuron and does not already have a loopback
  //connection
  while(NumTrysToFindLoop--)
  {
    //grab a random neuron
```

```
      int NeuronPos = RandInt(m_iNumInputs+1, m_vecNeurons.size()-1);

      //check to make sure the neuron does not already have a loopback
      //link and that it is not an input or bias neuron
      if (!m_vecNeurons[NeuronPos].bRecurrent &&
         (m_vecNeurons[NeuronPos].NeuronType != bias) &&
         (m_vecNeurons[NeuronPos].NeuronType != input))
      {
        ID_neuron1 = ID_neuron2 = m_vecNeurons[NeuronPos].iID;

        m_vecNeurons[NeuronPos].bRecurrent = true;

        bRecurrent = true;

        NumTrysToFindLoop = 0;
      }
    }
  }
```

First, the code checks to see if there is a chance of a looped recurrent link being added. If so, then it attempts `NumTrysToFindLoop` times to find an appropriate neuron. If no neuron is found, the program continues to look for two unconnected neurons.

```
  else
  {
    //No: try to find two unlinked neurons. Make NumTrysToAddLink
    //attempts
    while(NumTrysToAddLink--)
    {
```

Because some networks will already have existing connections between all its available neurons, the code has to make sure it doesn't enter an infinite loop when it tries to find two unconnected neurons. To prevent this from happening, the program only tries `NumTrysToAddLink` times to find two unlinked neurons. This value is set in CParams.cpp.

```
      //choose two neurons, the second must not be an input or a bias
      ID_neuron1 = m_vecNeurons[RandInt(0, m_vecNeurons.size()-1)].iID;

      ID_neuron2 =
```

```
      m_vecNeurons[RandInt(m_iNumInputs+1, m_vecNeurons.size()-1)].iID;

      if (ID_neuron2 == 2)
      {
        continue;
      }

      //make sure these two are not already linked and that they are
      //not the same neuron
      if ( !( DuplicateLink(ID_neuron1, ID_neuron2) ||
              (ID_neuron1 == ID_neuron2)))
      {
        NumTrysToAddLink = 0;
      }

      else
      {
        ID_neuron1 = -1;
        ID_neuron2 = -1;
      }
    }
  }

  //return if unsuccessful in finding a link
  if ( (ID_neuron1 < 0) || (ID_neuron2 < 0) )
  {
    return;
  }

  //check to see if we have already created this innovation
  int id = innovation.CheckInnovation(ID_neuron1, ID_neuron2, new_link);
```

Here, the code examines the innovation database to see if this link has already been discovered by another genome. `CheckInnovation` returns either the ID number of the innovation or, if the link is a new innovation, a negative value.

```
  //is this link recurrent?
  if (m_vecNeurons[GetElementPos(ID_neuron1)].dSplitY >
      m_vecNeurons[GetElementPos(ID_neuron2)].dSplitY)
```

```
  {
    bRecurrent = true;
  }
```

Here, the split values for the two neurons are compared to see if the link feeds forward or backward.

```
  if ( id < 0)
  {
    //we need to create a new innovation
    innovation.CreateNewInnovation(ID_neuron1, ID_neuron2, new_link);

    //now create the new gene
    int id = innovation.NextNumber() - 1;
```

If the program enters this section of code, then the innovation is a new one. Before the new gene is created, the innovation is added to the database and an identification number is retrieved. The new gene will be tagged with this identification number.

```
    SLinkGene NewGene(ID_neuron1,
                      ID_neuron2,
                      true,
                      id,
                      RandomClamped(),
                      bRecurrent);

    m_vecLinks.push_back(NewGene);
  }

  else
  {
    //the innovation has already been created so all we need to
    //do is create the new gene using the existing innovation ID
    SLinkGene NewGene(ID_neuron1,
                      ID_neuron2,
                      true,
                      id,
                      RandomClamped(),
                      bRecurrent);

    m_vecLinks.push_back(NewGene);
```

```
  }

  return;
}
```

## CGenome::AddNeuron

To add a neuron to a network, first a link must be chosen and then disabled. Two new links are then created to join the new neuron to its neighbors. See Figure 11.17.

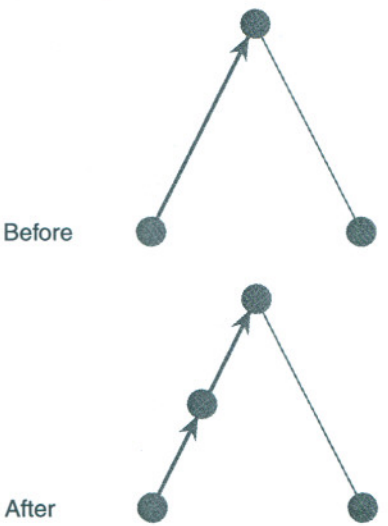



**Figure 11.17**

*Adding a neuron to a network.*

This means that every time a neuron is added, *three* innovations are created (or repeated if they have already been discovered): one for the neuron gene and two for the connection genes.

```
void CGenome::AddNeuron(double       MutationRate,
                        CInnovation  &innovations, //the innovation database
                        int          NumTrysToFindOldLink)
{
  //just return dependent on mutation rate
  if (RandFloat() > MutationRate) return;

  //if a valid link is found into which to insert the new neuron
  //this value is set to true.
```

```
bool bDone = false;

//this will hold the index into m_vecLinks of the chosen link gene
int  ChosenLink = 0;

//first a link is chosen to split. If the genome is small the code makes
//sure one of the older links is split to ensure a chaining effect does
//not occur. Here, if the genome contains less than 5 hidden neurons it
//is considered to be too small to select a link at random.
const int SizeThreshold = m_iNumInputs + m_iNumOutPuts + 5;

if (m_vecLinks.size() < SizeThreshold)
{
  while(NumTrysToFindOldLink--)
  {
    //choose a link with a bias towards the older links in the genome
    ChosenLink = RandInt(0, NumGenes()-1-(int)sqrt(NumGenes()));

    //make sure the link is enabled and that it is not a recurrent link
    //or has a bias input
    int FromNeuron = m_vecLinks[ChosenLink].FromNeuron;

    if ( (m_vecLinks[ChosenLink].bEnabled)    &&
         (!m_vecLinks[ChosenLink].bRecurrent) &&
         (m_vecNeurons[GetElementPos(FromNeuron)].NeuronType != bias))
     {
        bDone = true;

        NumTrysToFindOldLink = 0;
      }
  }

  if (!bDone)
  {
    //failed to find a decent link
    return;
  }
}
```

Early on in the development of the networks, a problem can occur where the same link is split repeatedly creating a chaining effect, as shown in Figure 11.18.

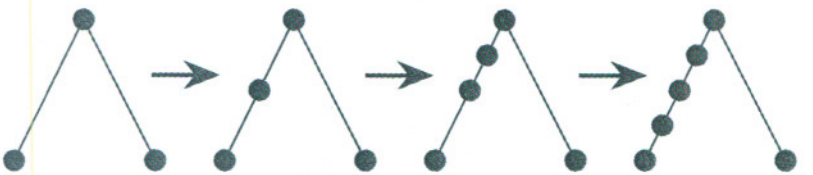


**Figure 11.18**
*The chaining effect.*

Obviously, this is undesirable, so the following code checks the number of neurons in the genome to see if the structure is below a certain size threshold. If it is, measures are taken to ensure that older links are selected in preference to newer ones.

```
else
{
  //the genome is of sufficient size for any link to be acceptable
  while (!bDone)
  {
    ChosenLink = RandInt(0, NumGenes()-1);

    //make sure the link is enabled and that it is not a recurrent link
    //or has a BIAS input
    int FromNeuron = m_vecLinks[ChosenLink].FromNeuron;

    if ( (m_vecLinks[ChosenLink].bEnabled) &&
         (!m_vecLinks[ChosenLink].bRecurrent) &&
         (m_vecNeurons[GetElementPos(FromNeuron)].NeuronType != bias))
    {
      bDone = true;
    }
  }
}

//disable this gene
m_vecLinks[ChosenLink].bEnabled = false;

//grab the weight from the gene (we want to use this for the weight of
//one of the new links so the split does not disturb anything the
//NN may have already learned
double OriginalWeight = m_vecLinks[ChosenLink].dWeight;
```

When a link is disabled and two new links are created, the old weight from the disabled link is used as the weight for one of the new links, and the weight for the other link is set to 1. In this way, the addition of a neuron creates as little disruption as possible to any existing learned behavior. See Figure 11.19.

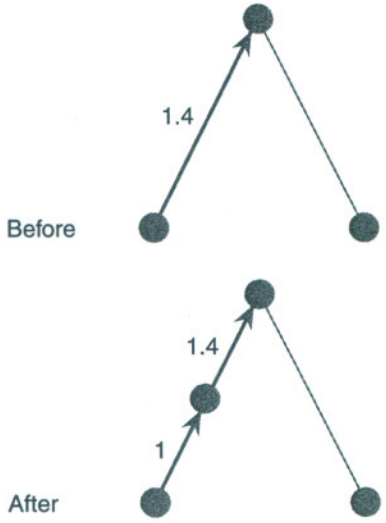



**Figure 11.19**

*Assigning weights to the new link genes.*

```
//identify the neurons this link connects
int from =  m_vecLinks[ChosenLink].FromNeuron;
int to   =  m_vecLinks[ChosenLink].ToNeuron;

//calculate the depth and width of the new neuron. We can use the depth
//to see if the link feeds backwards or forwards
double NewDepth = (m_vecNeurons[GetElementPos(from)].dSplitY +
                   m_vecNeurons[GetElementPos(to)].dSplitY) /2;

double NewWidth = (m_vecNeurons[GetElementPos(from)].dSplitX +
                   m_vecNeurons[GetElementPos(to)].dSplitX) /2;

//Now to see if this innovation has been created previously by
//another member of the population
int id = innovations.CheckInnovation(from,
                                     to,
```

```
                                        new_neuron);

  /*it is possible for NEAT to repeatedly do the following:

      1. Find a link. Lets say we choose link 1 to 5
      2. Disable the link,
      3. Add a new neuron and two new links
      4. The link disabled in Step 2 may be re-enabled when this genome
         is recombined with a genome that has that link enabled.
      5  etc etc

  Therefore, the following checks to see if a neuron ID is already being used.
  If it is, the function creates a new innovation for the neuron. */
  if (id >= 0)
  {
    int NeuronID = innovations.GetNeuronID(id);

    if (AlreadyHaveThisNeuronID(NeuronID))
    {
      id = -1;
    }
  }
```

`AlreadyHaveThisNeuronID` returns true if (you guessed it) the genome already has a neuron with an identical ID. If this is the case, then a new innovation needs to be created, so `id` is reset to `-1`.

```
  if (id < 0)  //this is a new innovation
  {
    //add the innovation for the new neuron
    int NewNeuronID = innovations.CreateNewInnovation(from,
                                                      to,
                                                      new_neuron,
                                                      hidden,
                                                      NewWidth,
                                                      NewDepth);


    //Create the new neuron gene and add it.
    m_vecNeurons.push_back(SNeuronGene(hidden,
```

```
                                        NewNeuronID,
                                        NewDepth,
                                        NewWidth));

    //Two new link innovations are required, one for each of the
    //new links created when this gene is split.

    //---------------------------------first link

    //get the next innovation ID
    int idLink1 = innovations.NextNumber();

    //create the new innovation
    innovations.CreateNewInnovation(from,
                                    NewNeuronID,
                                    new_link);


    //create the new gene
    SLinkGene link1(from,
                    NewNeuronID,
                    true,
                    idLink1,
                    1.0);

    m_vecLinks.push_back(link1);

    //---------------------------------second link

    //get the next innovation ID
    int idLink2 = innovations.NextNumber();

    //create the new innovation
    innovations.CreateNewInnovation(NewNeuronID,
                                    to,
                                    new_link);


    //create the new gene
    SLinkGene link2(NewNeuronID,
```

```
                    to,
                    true,
                    idLink2,
                    OriginalWeight);

  m_vecLinks.push_back(link2);
}

else     //existing innovation
{
  //this innovation has already been created so grab the relevant neuron
  //and link info from the innovation database
  int NewNeuronID = innovations.GetNeuronID(id);

  //get the innovation IDs for the two new link genes
  int idLink1 = innovations.CheckInnovation(from, NewNeuronID, new_link);
  int idLink2 = innovations.CheckInnovation(NewNeuronID, to, new_link);

  //this should never happen because the innovations *should* have already
  //occurred
  if ( (idLink1 < 0) || (idLink2 < 0) )
  {
    MessageBox(NULL, "Error in CGenome::AddNode", "Problem!", MB_OK);

    return;
  }

  //now we need to create 2 new genes to represent the new links
  SLinkGene link1(from, NewNeuronID, true, idLink1, 1.0);
  SLinkGene link2(NewNeuronID, to, true, idLink2, OriginalWeight);

  m_vecLinks.push_back(link1);
  m_vecLinks.push_back(link2);

  //create the new neuron
  SNeuronGene NewNeuron(hidden, NewNeuronID, NewDepth, NewWidth);

  //and add it
```

```
    m_vecNeurons.push_back(NewNeuron);
  }

  return;
}
```

## How Innovations Help in the Design of a Valid Crossover Operator

As I discussed at the beginning of this chapter, the crossover operator for EANNs can often be more trouble than it's worth. In addition to ensuring that crossover does not produce invalid networks, care must also be taken to avoid the competing conventions problem. The designers of NEAT have managed to steer clear of both these evils by using the innovation IDs as historical gene markers. Because each innovation has a unique ID, the genes can be tracked chronologically, which means similar genes in different genomes can be aligned prior to crossover. To see this clearly, take a look at Figure 11.20.

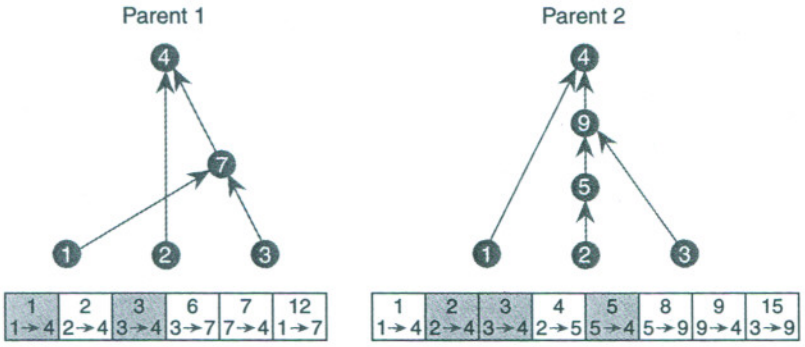



**Figure 11.20**

*Two phenotypes with different innovations. The gray genes are disabled. The number at the top of each gene is that gene's innovation number.*

The genes shown are the link genes for each phenotype. As you can see, the phenotypes have very different topologies, yet we can easily create an offspring from them by matching up the innovation numbers of the genomes before swapping over the appropriate genes, as shown in Figure 11.21.

Those genes that do not match in the middle of the genomes are called *disjoint* genes, whereas those that do not match at the end are called *excess* genes. Crossover proceeds a little like multi-point crossover, discussed earlier in the book. As the operator iterates down the length of each genome, the offspring inherits matching genes randomly. Disjoint and excess genes are only inherited from the fittest parent.

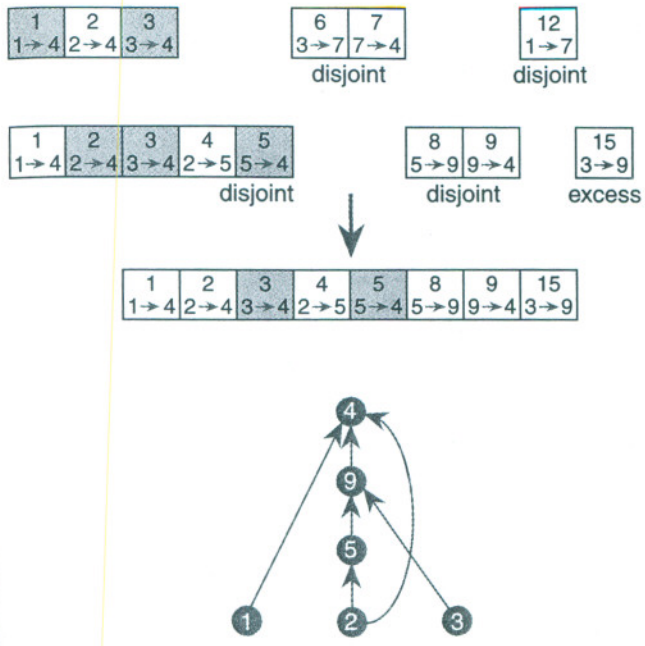



**Figure 11.21**

*The crossover operator in action.*

This way, NEAT ensures only valid offspring are created and that the competing convention problem is avoided. Neat, huh? (sorry, couldn't resist! <smile>)

Let me show you the code for the crossover operator, so you can check out the complete process.

```
CGenome Cga::Crossover(CGenome& mum, CGenome& dad)
{
  //first, calculate the genome we will using the disjoint/excess
  //genes from. This is the fittest genome. If they are of equal
  //fitness use the shorter (because we want to keep the networks
  //as small as possible)
  parent_type best;

  if (mum.Fitness() == dad.Fitness())
  {
    //if they are of equal fitness and length just choose one at
    //random
    if (mum.NumGenes() == dad.NumGenes())
    {
      best = (parent_type)RandInt(0, 1);
```

```
    }

    else
    {
      if (mum.NumGenes() < dad.NumGenes())
      {
        best = MUM;
      }

      else
      {
        best = DAD;
      }
    }
  }

  else
  {
    if (mum.Fitness() > dad.Fitness())
    {
      best = MUM;
    }

    else
    {
      best = DAD;
    }
  }

  //these vectors will hold the offspring's neurons and genes
  vector<SNeuronGene>  BabyNeurons;
  vector<SLinkGene>    BabyGenes;

  //temporary vector to store all added neuron IDs
  vector<int> vecNeurons;

  //create iterators so we can step through each parents genes and set
  //them to the first gene of each parent
  vector<SLinkGene>::iterator curMum = mum.StartOfGenes();
```

```
vector<SLinkGene>::iterator curDad = dad.StartOfGenes();

//this will hold a copy of the gene we wish to add at each step
SLinkGene SelectedGene;

//step through each parents genes until we reach the end of both
while (!((curMum == mum.EndOfGenes()) && (curDad == dad.EndOfGenes())))
{

  //the end of mum's genes have been reached
  if ((curMum == mum.EndOfGenes())&&(curDad != dad.EndOfGenes()))
  {
    //if dad is fittest
    if (best == DAD)
    {
      //add dads genes
      SelectedGene = *curDad;
    }

    //move onto dad's next gene
    ++curDad;
  }

  //the end of dad's genes have been reached
  else if ( (curDad == dad.EndOfGenes()) && (curMum != mum.EndOfGenes()))
  {
    //if mum is fittest
    if (best == MUM)
    {
      //add mums genes
      SelectedGene = *curMum;
    }

    //move onto mum's next gene
    ++curMum;
  }

  //if mums innovation number is less than dads
  else if (curMum->InnovationID < curDad->InnovationID)
```

```
{
  //if mum is fittest add gene
  if (best == MUM)
  {
    SelectedGene = *curMum;
  }

  //move onto mum's next gene
  ++curMum;
}

//if dad's innovation number is less than mum's
else if (curDad->InnovationID < curMum->InnovationID)
{
  //if dad is fittest add gene
  if (best = DAD)
  {
    SelectedGene = *curDad;
  }

  //move onto dad's next gene
  ++curDad;
}

//if innovation numbers are the same
else if (curDad->InnovationID == curMum->InnovationID)
{
  //grab a gene from either parent
  if (RandFloat() < 0.5f)
  {
    SelectedGene = *curMum;
  }

  else
  {
    SelectedGene = *curDad;
  }

  //move onto next gene of each parent
```

```
      ++curMum;
      ++curDad;
    }

    //add the selected gene if not already added
    if (BabyGenes.size() == 0)
    {
      BabyGenes.push_back(SelectedGene);
    }

    else
    {
      if (BabyGenes[BabyGenes.size()-1].InnovationID !=
          SelectedGene.InnovationID)
      {
        BabyGenes.push_back(SelectedGene);
      }
    }

    //Check if we already have the neurons referred to in SelectedGene.
    //If not, they need to be added.
    AddNeuronID(SelectedGene.FromNeuron, vecNeurons);
    AddNeuronID(SelectedGene.ToNeuron, vecNeurons);

  }//end while

  //now create the required neurons. First sort them into order
  sort(vecNeurons.begin(), vecNeurons.end());

  for (int i=0; i<vecNeurons.size(); i++)
  {
    BabyNeurons.push_back(m_pInnovation->CreateNeuronFromID(vecNeurons[i]));
  }

  //finally, create the genome
  CGenome babyGenome(m_iNextGenomeID++,
                     BabyNeurons,
                     BabyGenes,
                     mum.NumInputs(),
```

```
                    mum.NumOutputs());

  return babyGenome;
}
```

## Speciation

When structure is added to a genome, either by adding a new connection or a new neuron, it's quite likely the new individual will be a poor performer until it has a chance to evolve and establish itself among the population. Unfortunately, this means there is a high probability of the new individual dying out before it has time to evolve any potentially interesting behavior. This is obviously undesirable—some way has to be found of protecting the new innovation in the early days of its evolution. This is where simulating speciation comes in handy...

Speciation, as the name suggests, is the separation of a population into species. The question of what exactly *is* a species, is still one the biologists (and other scientists) are arguing over, but one of the popular definitions is:

> *A species is a group of populations with similar characteristics that are capable of successfully interbreeding with each other to produce healthy, fertile offspring, but are reproductively isolated from other species.*

In nature, a common mechanism for speciation is provided by changes in geography. Imagine a widespread population of animals, let's call them "critters", which eventually come to be divided by some geographical change in their environment, like the creation of a mountain ridge, for example. Over time, these populations will diversify because of different natural selection pressures and because of different mutations within their chromosomes. On one side of the mountain, the critters may start growing thicker fur to cope with a colder climate, and on the other, they may adapt to become better at avoiding the multitude of predators that lurk there. Eventually, the two populations will have changed so much from each other that if they ever did come into contact again, it would be impossible for them to mate successfully and have offspring. It's at this point they can be considered two different species.

NEAT simulates speciation to provide evolutionary niches for any new topological change. This way, similar individuals only have to compete among themselves and not with the rest of the population. Therefore, they are protected somewhat from premature extinction. A record of all the species created is kept in a class called—wait for it—`CSpecies`. Each epoch, every individual is tested against the first member in each species and a *compatibility distance* is calculated. If the compatibility distance

is within certain boundaries, then the individual is added to that species. If the individual is incompatible with all the current species, then a new species is created and the individual is added to that.

## Testing for Compatibility

The compatibility distance is calculated by measuring how diverse the genomes of two individuals are. Once again, the innovation numbers come in handy here because we can simply match up the genes, as we did for crossover, and count the number of excess and disjoint genes. The higher this count, the greater the diversity. In addition, the weights of the connections are also compared and a total of the absolute value of differences is recorded. Consequently, we have three criteria:

- The number of excess genes (E)
- The number of disjoint genes (D)
- The difference in connection weights (W)

Once these values have been determined, the final compatibility distance is calculated using the formula:

$$C.Dist = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 W$$

where N is the number of genes in the larger genome (to normalize for size) and $c_1$, $c_2$, and $c_3$ are coefficients used to tweak the final value accordingly. If this final value is below the compatibility threshold, the genomes are said to be of the same species. If it is higher, the genomes represent different species. The method used to calculate the compatibility distance is `CGenome::GetCompatibilityScore` and it looks like this:

```
double CGenome::GetCompatibilityScore(const CGenome &genome)
{
  //travel down the length of each genome counting the number of
  //disjoint genes, the number of excess genes and the number of
  //matched genes
  double  NumDisjoint = 0;
  double  NumExcess   = 0;
  double  NumMatched  = 0;

  //this records the summed difference of weights in matched genes
  double  WeightDifference = 0;

  //indexes into each genome. They are incremented as we
```

```
//step down each genomes length.
int g1 = 0;
int g2 = 0;

while ( (g1 < m_vecLinks.size()-1) || (g2 < genome.m_vecLinks.size()-1) )
{
  //we've reached the end of genome1 but not genome2 so increment
  //the excess score
  if (g1 == m_vecLinks.size()-1)
  {
    ++g2;
    ++NumExcess;

    continue;
  }

  //and vice versa
  if (g2 == genome.m_vecLinks.size()-1)
  {
    ++g1;
    ++NumExcess;

    continue;
  }

  //get innovation numbers for each gene at this point
  int id1 = m_vecLinks[g1].InnovationID;
  int id2 = genome.m_vecLinks[g2].InnovationID;

  //innovation numbers are identical so increase the matched score
  if (id1 == id2)
  {
    ++g1;
    ++g2;
    ++NumMatched;

    //get the weight difference between these two genes
    WeightDifference += fabs(m_vecLinks[g1].dWeight -
                             genome.m_vecLinks[g2].dWeight);
```

```
    }

    //innovation numbers are different so increment the disjoint score
    if (id1 < id2)
    {
      ++NumDisjoint;
      ++g1;
    }

    if (id1 > id2)
    {
      ++NumDisjoint;
      ++g2;
    }

  }//end while

  //get the length of the longest genome
  int longest = genome.NumGenes();

  if (NumGenes() > longest)
  {
    longest = NumGenes();
  }

  //these are multipliers used to tweak the final score.
  const double mDisjoint = 1;
  const double mExcess   = 1;
  const double mMatched  = 0.4;

  //finally calculate the scores
  double score = (mExcess   * NumExcess / ( double)longest) +
                 (mDisjoint * NumDisjoint / (double)longest) +
                 (mMatched  * WeightDifference / NumMatched);


  return score;
}
```

## The CSpecies Class

Once an individual has been assigned to a species, it may only mate with other members of the same species. However, speciation alone does not protect new innovation within the population. To do that, we must somehow find a way of adjusting the fitnesses of each individual in a way that aids younger, more diverse genomes to remain active for a reasonable length of time. The technique NEAT uses to do this is called *explicit fitness sharing*.

As I discussed in Chapter 5, "Building a Better Genetic Algorithm," fitness sharing is a way of retaining diversity by sharing the fitness scores of individuals with similar genomes. With NEAT, fitness scores are shared by members of the same species. In practice, this means that each individual's score is divided by the size of the species before any selection occurs. What this boils down to is that species which grow large are penalized for their size, whereas smaller species are given a "foot up" in the evolutionary race, so to speak.

**NOTE**

**In the original implementation of NEAT, the designers incorporated inter-species mating although the probability of this happening was set very low. Although I have never observed any noticeable performance increase when using it, it may be a worthwhile exercise for you to try this out when you start fooling around with your own implementations.**

In addition, young species are given a fitness boost prior to the fitness sharing calculation. Likewise, old species are penalized. If a species does not show an improvement over a certain number of generations (the default is 15), then it is killed off. The exception to this is if the species contains the best performing individual found so far, in which case the species is allowed to live.

I think the best thing I can do to help clarify all the information I've just thrown at you is to show you the method that calculates all the fitness adjustments. First though, let me take a moment to list the `CSpecies` class definition:

```
class CSpecies
{

private:

  //keep a local copy of the first member of this species
```

```
  CGenome           m_Leader;

  //pointers to all the genomes within this species
  vector<CGenome*>  m_vecMembers;

  //the species needs an identification number
  int               m_iSpeciesID;

  //best fitness found so far by this species
  double            m_dBestFitness;

  //average fitness of the species
  double            m_dAvFitness;

  //generations since fitness has improved, we can use
  //this info to kill off a species if required
  int               m_iGensNoImprovement;

  //age of species
  int               m_iAge;

  //how many of this species should be spawned for
  //the next population
  double            m_dSpawnsRqd;


public:

  CSpecies(CGenome &FirstOrg, int SpeciesID);

  //this method boosts the fitnesses of the young, penalizes the
  //fitnesses of the old and then performs fitness sharing over
  //all the members of the species
  void    AdjustFitnesses();

  //adds a new individual to the species
```

```
  void    AddMember(CGenome& new_org);

  void    Purge();

  //calculates how many offspring this species should spawn
  void    CalculateSpawnAmount();

  //spawns an individual from the species selected at random
  //from the best CParams::dSurvivalRate percent
  CGenome Spawn();


  //-------------------------------------accessor methods
  CGenome  Leader()const{return m_Leader;}

  double   NumToSpawn()const{return m_dSpawnsRqd;}

  int      NumMembers()const{return m_vecMembers.size();}

  int      GensNoImprovement()const{return m_iGensNoImprovement;}

  int      ID()const{return m_iSpeciesID;}

  double   SpeciesLeaderFitness()const{return m_Leader.Fitness();}

  double   BestFitness()const{return m_dBestFitness;}

  int      Age()const{return m_iAge;}


  //so we can sort species by best fitness. Largest first
  friend bool operator<(const CSpecies &lhs, const CSpecies &rhs)
  {
    return lhs.m_dBestFitness > rhs.m_dBestFitness;
  }
};
```

And now for the method that adjusts the fitness scores:

```
void CSpecies::AdjustFitnesses()
```

```
{
  double total = 0;

  for (int gen=0; gen<m_vecMembers.size(); ++gen)
  {
    double fitness = m_vecMembers[gen]->Fitness();

    //boost the fitness scores if the species is young
    if (m_iAge < CParams::iYoungBonusAgeThreshhold)
    {
      fitness *= CParams::dYoungFitnessBonus;
    }

    //punish older species
    if (m_iAge > CParams::iOldAgeThreshold)
    {
      fitness *= CParams::dOldAgePenalty;
    }

    total += fitness;

    //apply fitness sharing to adjusted fitnesses
    double AdjustedFitness = fitness/m_vecMembers.size();

    m_vecMembers[gen]->SetAdjFitness(AdjustedFitness);

  }
}
```

## The Cga Epoch Method

Because the population is speciated, the epoch method for the NEAT code is somewhat different (and a hell of a lot longer!) than the epoch functions you've seen previously in this book. `Epoch` is part of the `Cga` class, which is the class that manipulates all the genomes, species, and innovations.

Let me talk you through the `Epoch` method so you understand exactly what's going on at each stage of the process:

```
vector<CNeuralNet*> Cga::Epoch(const vector<double> &FitnessScores)
{
```

```
//first check to make sure we have the correct amount of fitness scores
if (FitnessScores.size() != m_vecGenomes.size())
{
  MessageBox(NULL,"Cga::Epoch(scores/ genomes mismatch)!","Error", MB_OK);
}

ResetAndKill();
```

First of all, any phenotypes created during the previous generation are deleted. The program then examines each species in turn and deletes all of its members apart from the best performing one. (You use this individual as the genome to be tested against when the compatibility distances are calculated). If a species hasn't made any fitness improvement in `CParams::iNumGensAllowedNoImprovement` generations, the species is killed off.

```
//update the genomes with the fitnesses scored in the last run
for (int gen=0; gen<m_vecGenomes.size(); ++gen)
{
  m_vecGenomes[gen].SetFitness(FitnessScores[gen]);
}

//sort genomes and keep a record of the best performers
SortAndRecord();

//separate the population into species of similar topology, adjust
//fitnesses and calculate spawn levels
SpeciateAndCalculateSpawnLevels();
```

`SpeciateAndCalculateSpawnLevels` commences by calculating the compatibility distance of each genome against the representative genome from each live species. If the value is within a set tolerance, the individual is added to that species. If no species match is found, then a new species is created and the genome added to that.

When all the genomes have been assigned to a species `SpeciateAndCalculateSpawnLevels` calls the member function `AdjustSpeciesFitnesses` to adjust and share the fitness scores as discussed previously.

Next, `SpeciateAndCalculateSpawnLevels` calculates how many offspring each individual is predicted to spawn into the new generation. This is a floating-point value calculated by dividing each genome's adjusted fitness score with the average adjusted fitness score for the entire population. For example, if a genome had an adjusted fitness score of 4.4 and the average is 8.0, then the genome should spawn 0.525

offspring. Of course, it's impossible for an organism to spawn a fractional part of itself, but all the individual spawn amounts for the members of each species are summed to calculate an overall spawn amount for that species. Table 11.3 may help clear up any confusion you may have with this process. It shows typical spawn values for a small population of 20 individuals. The epoch function can now simply iterate through each species and spawn the required amount of offspring.

To continue with the `Epoch` method...

```
//this will hold the new population of genomes
vector<CGenome> NewPop;

//request the offspring from each species. The number of children to
//spawn is a double which we need to convert to an int.
int NumSpawnedSoFar = 0;

CGenome baby;

//now to iterate through each species selecting offspring to be mated and
//mutated
for (int spc=0; spc<m_vecSpecies.size(); ++spc)
{
  //because of the number to spawn from each species is a double
  //rounded up or down to an integer it is possible to get an overflow
  //of genomes spawned. This statement just makes sure that doesn't
  //happen
  if (NumSpawnedSoFar < CParams::iNumSweepers)
  {
    //this is the amount of offspring this species is required to
    // spawn. Rounded simply rounds the double up or down.
    int NumToSpawn = Rounded(m_vecSpecies[spc].NumToSpawn());

    bool bChosenBestYet = false;

    while (NumToSpawn--)
    {
      //first grab the best performing genome from this species and transfer
      //to the new population without mutation. This provides per species
      //elitism
      if (!bChosenBestYet)
      {
```

```
      baby = m_vecSpecies[spc].Leader();

      bChosenBestYet = true;
    }

    else
    {
      //if the number of individuals in this species is only one
      //then we can only perform mutation
      if (m_vecSpecies[spc].NumMembers() == 1)
      {
        //spawn a child
        baby = m_vecSpecies[spc].Spawn();
      }

      //if greater than one we can use the crossover operator
      else
      {
        //spawn1
        CGenome g1 = m_vecSpecies[spc].Spawn();

        if (RandFloat() < CParams::dCrossoverRate)
        {

          //spawn2, make sure it's not the same as g1
          CGenome g2 = m_vecSpecies[spc].Spawn();

          // number of attempts at finding a different genome
          int NumAttempts = 5;

          while ( (g1.ID() == g2.ID()) && (NumAttempts--) )
          {
            g2 = m_vecSpecies[spc].Spawn();
          }

          if (g1.ID() != g2.ID())
          {
            baby = Crossover(g1, g2);
          }
```

## Table 11.3 Species Spawn Amounts

**Species 0**

| Genome ID | Fitness | Adjusted Fitness | Spawn Amount |
|---|---|---|---|
| 88 | 100 | 14.44 | 1.80296 |
| 103 | 99 | 14.3 | 1.78493 |
| 94 | 99 | 14.3 | 1.78493 |
| 61 | 92 | 13.28 | 1.65873 |
| 106 | 37 | 5.344 | 0.667096 |
| 108 | 34 | 4.911 | 0.613007 |
| 107 | 32 | 4.622 | 0.576948 |
| 105 | 11 | 1.588 | 0.198326 |
| 104 | 7 | 1.011 | 0.126207 |

**Total offspring for this species to spawn: 9.21314**

**Species 1**

| Genome ID | Fitness | Adjusted Fitness | Spawn Amount |
|---|---|---|---|
| 112 | 43 | 7.980 | 0.99678 |
| 110 | 43 | 7.985 | 0.99678 |
| 116 | 42 | 7.8 | 0.973599 |
| 68 | 41 | 7.614 | 0.950419 |
| 111 | 37 | 6.871 | 0.857695 |
| 115 | 37 | 6.871 | 0.857695 |
| 113 | 17 | 3.157 | 0.394076 |

**Total offspring for this species to spawn: 6.02704**

**Species 2**

| Genome ID | Fitness | Adjusted Fitness | Spawn Amount |
|---|---|---|---|
| 20 | 59 | 25.56 | 3.19124 |
| 100 | 14 | 6.066 | 0.757244 |
| 116 | 9 | 3.9 | 0.4868 |

**Total offspring for this species to spawn: 4.43529**

Because the number of individuals in a species may be small and because only the best 20% (default value) are retained to be parents, it is sometimes impossible (or slow) to find a second genome to mate with. The code shown here tries five times to find a different genome and then aborts.

```
    }

    else
    {
      baby = g1;
    }
  }

  ++m_iNextGenomeID;

  baby.SetID(m_iNextGenomeID);

  //now we have a spawned child lets mutate it! First there is the
  //chance a neuron may be added
  if (baby.NumNeurons() < CParams::iMaxPermittedNeurons)
  {
    baby.AddNeuron(CParams::dChanceAddNode,
                   *m_pInnovation,
                   CParams::iNumTrysToFindOldLink);
  }

  //now there's the chance a link may be added
  baby.AddLink(CParams::dChanceAddLink,
               CParams::dChanceAddRecurrentLink,
               *m_pInnovation,
               CParams::iNumTrysToFindLoopedLink,
               CParams::iNumAddLinkAttempts);

  //mutate the weights
  baby.MutateWeights(CParams::dMutationRate,
                     CParams::dProbabilityWeightReplaced,
                     CParams::dMaxWeightPerturbation);

  //mutate the activation response
```

```
        baby.MutateActivationResponse(CParams::dActivationMutationRate,
                                      CParams::dMaxActivationPerturbation);
      }

      //sort the babies genes by their innovation numbers
      baby.SortGenes();

      //add to new pop
      NewPop.push_back(baby);

      ++NumSpawnedSoFar;

      if (NumSpawnedSoFar == CParams::iNumSweepers)
      {
        NumToSpawn = 0;
      }

    }//end while

  }//end if

}//next species


//if there is an underflow due to a rounding error when adding up all
//the species spawn amounts, and the amount of offspring falls short of
//the population size, additional children need to be created and added
//to the new population. This is achieved simply, by using tournament
//selection over the entire population.
if (NumSpawnedSoFar < CParams::iNumSweepers)
{
  //calculate the amount of additional children required
  int Rqd = CParams::iNumSweepers - NumSpawnedSoFar;

  //grab them
  while (Rqd--)
  {
    NewPop.push_back(TournamentSelection(m_iPopSize/5));
  }
```

```
  }

  //replace the current population with the new one
  m_vecGenomes = NewPop;

  //create the new phenotypes
  vector<CNeuralNet*> new_phenotypes;

  for (gen=0; gen<m_vecGenomes.size(); ++gen)
  {
    //calculate max network depth
    int depth = CalculateNetDepth(m_vecGenomes[gen]);

    CNeuralNet* phenotype = m_vecGenomes[gen].CreatePhenotype(depth);

    new_phenotypes.push_back(phenotype);
  }

  //increase generation counter
  ++m_iGeneration;

  return new_phenotypes;
}
```

# Converting the Genome into a Phenotype

Well, I've covered just about everything except how a genome is converted into a phenotype. We're nearly there now! Phenotypes use different neuron and link structures than the genome. They can be found in phenotype.h, and look like this:

## The SLink Structure

The structure for the links is very simple. It just has pointers to the two neurons it connects and a connection weight. The bool value, `bRecurrent`, is used by the drawing routine in `CNeuralNet` to help render a network into a window.

```
struct SLink
{
  //pointers to the neurons this link connects
```

```
  CNeuron*  pIn;
  CNeuron*  pOut;

  //the connection weight
  double  dWeight;

  //is this link a recurrent link?
  bool    bRecurrent;

  SLink(double dW, CNeuron* pIn, CNeuron* pOut, bool bRec):dWeight(dW),
                                                           pIn(pIn),
                                                           pOut(pOut),
                                                           bRecurrent(bRec)

  {}
};
```

## The SNeuron Structure

The neuron defined by `SNeuron` contains much more information than its little brother `SNeuronGene`. In addition, it holds the values for the sum of all the inputs × weights, this value after it's been put through the activation function (in other words, the output from this neuron), and two `std::vectors`—one for storing the links into the neuron, and the other for storing the links out of the neuron.

```
struct SNeuron
{
   //all the links coming into this neuron
  vector<SLink> vecLinksIn;

  //and out
  vector<SLink> vecLinksOut;

  //sum of weights x inputs
  double        dSumActivation;

  //the output from this neuron
  double        dOutput;

  //what type of neuron is this?
```

```
  neuron_type   NeuronType;

  //its identification number
  int           iNeuronID;

  //sets the curvature of the sigmoid function
  double        dActivationResponse;

  //used in visualization of the phenotype
  int           iPosX,   iPosY;
  double        dSplitY, dSplitX;

  //-- ctors
  SNeuron(neuron_type type,
          int         id,
          double      y,
          double      x,
          double      ActResponse):NeuronType(type),
                                   iNeuronID(id),
                                   dSumActivation(0),
                                   dOutput(0),
                                   iPosX(0),
                                   iPosY(0),
                                   dSplitY(y),
                                   dSplitX(x),
                                   dActivationResponse(ActResponse)
  {}
};
```

## Putting the Bits Together

The method that actually creates all the `SLinks` and `SNeurons` required for a phenotype is `CGenome::CreatePhenotype`. This function iterates through the genome and creates any appropriate neurons and all the required links required for pointing to those neurons. It then creates an instance of the `CNeuralNet` class. I'll be discussing the `CNeuralNet` class immediately after you've had a good look at the following code.

```
CNeuralNet* CGenome::CreatePhenotype(int depth)
{
```

```
//first make sure there is no existing phenotype for this genome
DeletePhenotype();

//this will hold all the neurons required for the phenotype
vector<SNeuron*>  vecNeurons;

//first, create all the required neurons
for (int i=0; i<m_vecNeurons.size(); i++)
{
  SNeuron* pNeuron = new SNeuron(m_vecNeurons[i].NeuronType,
                                 m_vecNeurons[i].iID,
                                 m_vecNeurons[i].dSplitY,
                                 m_vecNeurons[i].dSplitX,
                                 m_vecNeurons[i].dActivationResponse);

  vecNeurons.push_back(pNeuron);
}

//now to create the links.
for (int cGene=0; cGene<m_vecLinks.size(); ++cGene)
{
  //make sure the link gene is enabled before the connection is created
  if (m_vecLinks[cGene].bEnabled)
  {
    //get the pointers to the relevant neurons
    int element         = GetElementPos(m_vecLinks[cGene].FromNeuron);
    SNeuron* FromNeuron = vecNeurons[element];

    element           = GetElementPos(m_vecLinks[cGene].ToNeuron);
    SNeuron* ToNeuron = vecNeurons[element];

    //create a link between those two neurons and assign the weight stored
    //in the gene
    SLink tmpLink(m_vecLinks[cGene].dWeight,
                  FromNeuron,
                  ToNeuron,
                  m_vecLinks[cGene].bRecurrent);

    //add new links to neuron
```

```
        FromNeuron->vecLinksOut.push_back(tmpLink);
        ToNeuron->vecLinksIn.push_back(tmpLink);
      }
    }

    //now the neurons contain all the connectivity information, a neural
    //network may be created from them.
    m_pPhenotype = new CNeuralNet(vecNeurons, depth);

    return m_pPhenotype;
}
```

## The CNeuralNet Class

This class is pretty simple. It contains a `std::vector` of the neurons that comprise the network, a method to update the network and retrieve its output, and a method to draw a representation of the network into a user-specified window. The value `m_iDepth` is the depth of the network calculated from the `splitY` values of its neuron genes, as discussed earlier. You'll see how this value is used in a moment. The enumerated type, `run_type`, is especially important because this is how the user chooses *how* the network is updated. I'll elaborate on this after you've taken a moment to look at the class definition.

```
class CNeuralNet
{

private:

  vector<SNeuron*>  m_vecpNeurons;

  //the depth of the network
  int               m_iDepth;


public:


  CNeuralNet(vector<SNeuron*> neurons,
```

```
            int             depth);

  ~CNeuralNet();

  //you have to select one of these types when updating the network
  //If snapshot is chosen the network depth is used to completely
  //flush the inputs through the network. active just updates the
  //network each time-step
  enum run_type{snapshot, active};

  //update network for this clock cycle
  vector<double>  Update(const vector<double> &inputs, const run_type type);

  //draws a graphical representation of the network to a user specified window
  void            DrawNet(HDC &surface,
                          int cxLeft,
                          int cxRight,
                          int cyTop,
                          int cyBot);
};
```

Up until now, all the networks you've seen have run the inputs through the complete network, layer by layer, until an output is produced. With NEAT however, a network can assume any topology with connections between neurons leading backward, forward, or even looping back on themselves. This makes it next to impossible to use a layer-based update function because there aren't really any layers! Because of this, the NEAT update function runs in one of two modes:

**active**: When using the `active` update mode, each neuron adds up all the activations calculated during the *preceeding time-step* from all its incoming neurons. This means that the activation values, instead of being flushed through the entire network like a conventional ANN each time-step, only travel from one neuron to the next. To get the same result as a layer-based method, this process would have to be repeated as many times as the network is deep in order to flush all the neuron activations completely through the network. This mode is appropriate to use if you are using the network dynamically (like for controlling the minesweepers for instance).

**snapshot**: If, however, you want NEAT's `update` function to behave like a regular neural network update function, you have to ensure that the activations are flushed *all the way through* from the input neurons to the output neurons. To facilitate this,

`Update` iterates through all the neurons as many times as the network is deep before spitting out the output. This is why calculating those `splitY` values was so important. You would use this type of update if you were to train a NEAT network using a training set. (Like we used for the mouse gesture recognition program in Chapter 9, "A Supervised Training Approach").

Here is the code for `CNeuralNet::Update`, which should help clarify the process.

```
vector<double> CNeuralNet::Update(const vector<double> &inputs,
                                  const run_type        type)
{
  //create a vector to put the outputs into
  vector<double>  outputs;

  //if the mode is snapshot then we require all the neurons to be
  //iterated through as many times as the network is deep. If the
  //mode is set to active the method can return an output after
  //just one iteration
  int FlushCount = 0;

  if (type == snapshot)
  {
    FlushCount = m_iDepth;
  }
  else
  {
    FlushCount = 1;
  }

  //iterate through the network FlushCount times
  for (int i=0; i<FlushCount; ++i)
  {
    //clear the output vector
    outputs.clear();

    //this is an index into the current neuron
    int cNeuron = 0;

    //first set the outputs of the 'input' neurons to be equal
    //to the values passed into the function in inputs
```

```
while (m_vecpNeurons[cNeuron]->NeuronType == input)
{
  m_vecpNeurons[cNeuron]->dOutput = inputs[cNeuron];

  ++cNeuron;
}

//set the output of the bias to 1
m_vecpNeurons[cNeuron++]->dOutput = 1;

//then we step through the network a neuron at a time
while (cNeuron < m_vecpNeurons.size())
{
  //this will hold the sum of all the inputs x weights
  double sum = 0;

  //sum this neuron's inputs by iterating through all the links into
  //the neuron
  for (int lnk=0; lnk<m_vecpNeurons[cNeuron]->vecLinksIn.size(); ++lnk)
  {
    //get this link's weight
    double Weight = m_vecpNeurons[cNeuron]->vecLinksIn[lnk].dWeight;

    //get the output from the neuron this link is coming from
    double NeuronOutput =
    m_vecpNeurons[cNeuron]->vecLinksIn[lnk].pIn->dOutput;

    //add to sum
    sum += Weight * NeuronOutput;
  }

  //now put the sum through the activation function and assign the
  //value to this neuron's output
  m_vecpNeurons[cNeuron]->dOutput =
  Sigmoid(sum, m_vecpNeurons[cNeuron]->dActivationResponse);

  if (m_vecpNeurons[cNeuron]->NeuronType == output)
  {
    //add to our outputs
```

```
            outputs.push_back(m_vecpNeurons[cNeuron]->dOutput);
          }

          //next neuron
          ++cNeuron;
        }

      }//next iteration through the network

      //the network outputs need to be reset if this type of update is performed
      //otherwise it is possible for dependencies to be built on the order
      //the training data is presented
      if (type == snapshot)
      {
        for (int n=0; n<m_vecpNeurons.size(); ++n)
        {
          m_vecpNeurons[n]->dOutput = 0;
        }
      }

      //return the outputs
      return outputs;
    }
```

Note that the outputs of the network must be reset to zero before the function returns if the `snapshot` method of updating is required. This is to prevent any dependencies on the *order* the training data is presented. (Training data is usually presented to a network sequentially because doing it randomly would slow down the learning considerably.)

For example, imagine presenting a training set consisting of a number of points lying on the circumference of a circle. If the network is not flushed, NEAT might add recurrent connections that make use of the data stored from the previous update. This would be okay if you wanted a network that simply mapped inputs to outputs, but most often you will require the network to generalize.

## Running the Demo Program

To demonstrate NEAT in practice, I've plugged in the minesweeper code from Chapter 8, "Giving Your Bot Senses." I think you'll be pleasantly surprised by how

NEAT performs in comparison! You can either compile it yourself or run the executable NEAT Sweepers.exe straight from the relevant folder on the CD.

As before, the F key speeds up the evolution, the R key resets it, and the B key shows the best four minesweepers from the previous generation. Pressing the keys 1 through 4 shows the minesweeper's "trails".

This time there is also an additional window created in which the phenotypes of the four best minesweepers are drawn, as shown in Figure 11.22.

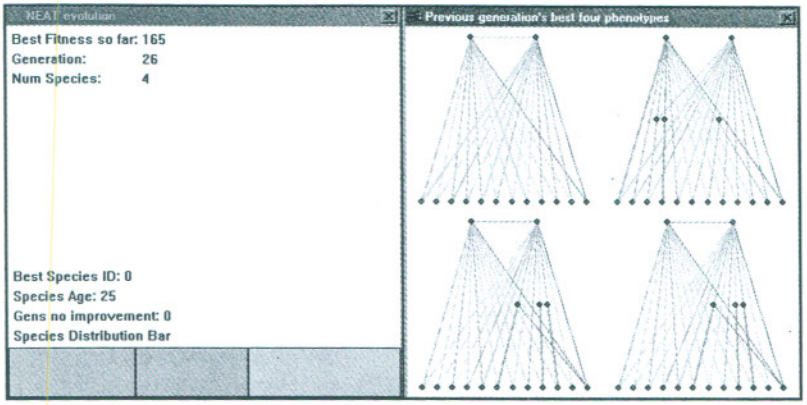



**Figure 11.22**

*NEAT Sweepers in action.*

Excitory forward connections are shown in gray and inhibitory forward connections are shown in yellow. Excitory recurrent connections are shown in red and inhibitory connections are shown in blue. Any connections from the bias neuron are shown in green. The thickness of the line gives an indication of the magnitude of the connection weight.

Table 11.4 lists the default settings for this project:

# Summary

You've come a long way in this chapter, and learned a lot in the process. To aid your understanding, the implementation of NEAT I describe in this chapter has been kept simple and it would be worthwhile for the curious to examine Ken Stanley and Risto Miikkulainen's original code to gain a fuller insight into the mechanisms of NEAT. You can find the source code and other articles about NEAT via Ken's Web site at:

http://www.cs.utexas.edu/users/kstanley/

## Table 11.4 Default Project Settings for NEAT Sweepers

### Parameters for the Minesweepers

| Parameter | Setting |
|---|---|
| Num sensors | 5 |
| Sensor range | 25 |
| Num minesweepers | 50 |
| Max turn rate | 0.2 |
| Scale | 5 |

### Parameters Affecting Evolution

| Parameter | Setting |
|---|---|
| Num ticks per epoch | 2000 |
| Chance of adding a link | 0.07 |
| Chance of adding a node | 0.03 |
| Chance of adding a recurrent link | 0.05 |
| Crossover rate | 0.7 |
| Weight mutation rate | 0.2 |
| Max mutation perturbation | 0.5 |
| Probability a weight is replaced | 0.1 |
| Probability the activation response is mutated | 0.1 |
| Species compatibility threshold | 0.26 |
| Species old age threshold | 50 |
| Species old age penalty | 0.7 |
| Species youth threshold | 10 |
| Species youth bonus | 1.3 |

## Stuff to Try

1. Add code to automatically keep the number of species within user-defined boundaries.
2. Have a go at designing some different mutation operators.
3. Add interspecies mating.
4. Have a go at coding one of the alternative methods for evolving network topology described at the beginning of the chapter.