CHAPTER 11

Evolving Intelligence

Throughout this book you've seen a number of different problems, and in each case you used an algorithm that was suited to solve that particular problem. In some of the examples, you had to tweak the parameters or use optimization to search for a good set of parameters. This chapter will look at a different way to approach problems. Instead of choosing an algorithm to apply to a problem, you'll make a program that attempts to automatically build the best program to solve a problem. Essentially, you'll be creating an algorithm that creates algorithms.

To do this, you will use a technique called *genetic programming*. Since this is the last chapter in which you'll learn a completely new type of algorithm, I've picked a topic that is new, exciting, and being actively researched. This chapter is a little different from the others because it doesn't use any open APIs or public datasets, and because programs that can modify themselves based on their interactions with many people are an interesting and different kind of collective intelligence. Genetic programming is a very large topic about which many books have been written, so you'll only get an introduction here, but I hope it's enough to get you excited about the possibilities and perhaps to research and experiment on your own.

The two problems in this chapter are recreating a mathematical function given a dataset, and automatically creating an AI (artificial intelligence) player for a simple board game. This is just a very small sampling of the possibilities of genetic programming—computational power is really the only constraint on the types of problems it can be used to solve.

What Is Genetic Programming?

Genetic programming is a machine-learning technique inspired by the theory of biological evolution. It generally works by starting with a large set of programs (referred to as the *population*), which are either randomly generated or hand-designed and are known to be somewhat good solutions. The programs are then made to compete in some user-defined task. This may be a game in which the programs compete against

each other directly, or it may be an individual test to see which program performs better. After the competition, a ranked list of the programs from best to worst can be determined.

Next—and here's where evolution comes in—the best programs are replicated and modified in two different ways. The simpler way is *mutation*, in which certain parts of the program are altered very slightly in a random manner in the hope that this will make a good solution even better. The other way to modify a program is through *crossover* (sometimes referred to as *breeding*), which involves taking a portion of one of the best programs and replacing it with a portion of one of the other best programs. This replication and modification procedure creates many new programs that are based on, but different from, the best programs.

At each stage, the quality of the programs is calculated using a *fitness function*. Since the size of the population is kept constant, many of the *worst* programs are eliminated from the population to make room for the new programs. The new population is referred to as "the next generation," and the whole procedure is then repeated. Because the best programs are being kept and modified, it is expected that with each generation they will get better and better, in much the same way that teenagers can be smarter than their parents.

New generations are created until a termination condition is reached, which, depending on the problem, can be that:

- The perfect solution has been found.
- A good enough solution has been found.
- The solution has not improved for several generations.
- The number of generations has reached a specified limit.

For some problems, such as determining a mathematical function that correctly maps a set of inputs to an output, a perfect solution is possible. For others, such as a board game, there may not be a perfect solution, since the quality of a solution depends on the strategy of the program's adversary.

An overview of the genetic programming process is shown as a flowchart in Figure 11-1.

Genetic Programming Versus Genetic Algorithms

Chapter 5 introduced a related set of algorithms known as *genetic algorithms*. Genetic algorithms are an optimization technique that use the idea of evolutionary pressure to choose the best result. With any form of optimization, you have already selected an algorithm or metric and you're simply trying to find the best parameters for it.

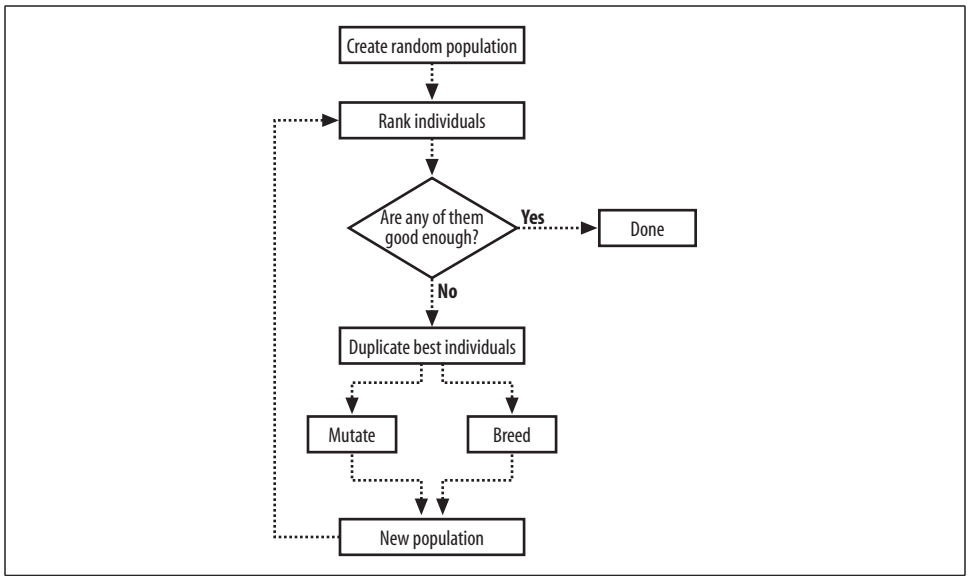

Figure 11-1. Genetic programming overview

Successes of Genetic Programming

Genetic programming has been around since the 1980s, but it is very computationally intensive, and with the computing power that was available at the time, it couldn't be used for anything more than simple problems. As computers have gotten faster, however, people have been able to apply genetic programming to sophisticated problems. Many previously patented inventions have been rediscovered or improved using genetic programming, and recently several new patentable inventions have been designed by computers.

The genetic programming technique has been applied in designing antennas for NASA, and in photonic crystals, optics, quantum computing systems, and other scientific inventions. It has also been used to develop programs for playing many games, such as chess and backgammon. In 1998, researchers from Carnegie Mellon University entered a robot team that was programmed entirely using genetic programming into the RoboCup soccer contest, and placed in the middle of the pack.

Like optimization, genetic programming requires a way to measure how good a solution is; but unlike optimization, the solutions are not just a set of parameters being applied to a given algorithm. Instead, the algorithm itself and all its parameters are designed automatically by means of evolutionary pressure.

Programs As Trees

In order to create programs that can be tested, mutated, and bred, you'll need a way to represent and run them from within your Python code. The representation has to lend itself to easy modification and, more importantly, has to be guaranteed to be an actual program—which means generating random strings and trying to treat them as Python code won't work. Researchers have come up with a few different ways to represent programs for genetic programming, and the most commonly used is a tree representation.

Most programming languages, when compiled or interpreted, are first turned into a *parse tree*, which is very similar to what you'll be working with here. (The programming language Lisp and its variants are essentially ways of entering a parse tree directly.) An example of a parse tree is shown in Figure 11-2.

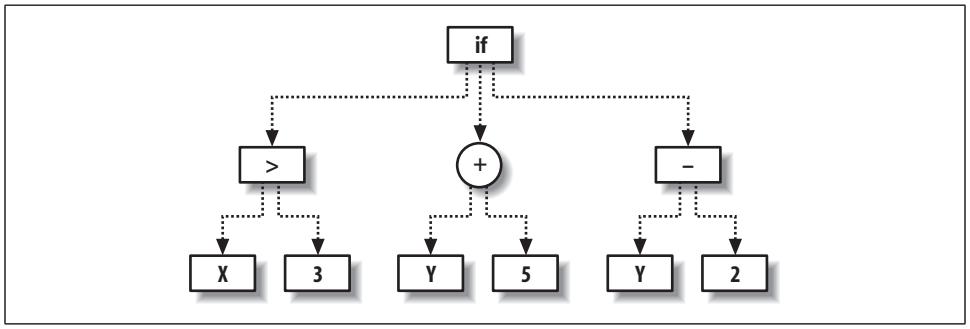

Figure 11-2. Example program tree

Each node represents either an operation on its child nodes or an endpoint, such as a parameter with a constant value. For example, the circular node is a sum operation on its two branches, in this case, the values Y and 5. Once this point is evaluated, it is given to the node above it, which in turn applies its own operation to its branches. You'll also notice that one of the nodes has the operation "if," which specifies that if its leftmost branch evaluates to true, return its center branch; if it doesn't, return its rightmost branch.

Traversing the complete tree, you can see that it corresponds to the Python function:

```
def func(x,y)
  if x>3:
    return y + 5
  else:
    return y - 2
```

At first, it might appear that these trees can only be used to build very simple functions. There are two things to consider here—first, the nodes that compose the tree can potentially be very complex functions, such as distance measures or

Gaussians. The second thing is that trees can be made recursive by referring to nodes higher up in the tree. Creating trees like this allows for loops and other more complicated control structures.

Representing Trees in Python

You're now ready to construct tree programs in Python. The trees are made up of nodes, which, depending on the functions associated with them, have some number of child nodes. Some of the nodes will return parameters passed to the program, others will return constants, and the most interesting ones will return operations on their child nodes.

Create a new file called *gp.py* and create four new classes called `fwrapper`, `node`, `paramnode`, and `constnode`:

```
from random import random,randint,choice
from copy import deepcopy
from math import log

class fwrapper:
  def __init__(self,function,childcount,name):
    self.function=function
    self.childcount=childcount
    self.name=name

class node:
  def __init__(self,fw,children):
    self.function=fw.function
    self.name=fw.name
    self.children=children

  def evaluate(self,inp):
    results=[n.evaluate(inp) for n in self.children]
    return self.function(results)

class paramnode:
  def __init__(self,idx):
    self.idx=idx

  def evaluate(self,inp):
    return inp[self.idx]

class constnode:
  def __init__(self,v):
    self.v=v
  def evaluate(self,inp):
    return self.v
```

The classes here are:

`fwrapper`
: A wrapper for the functions that will be used on function nodes. Its member variables are the name of the function, the function itself, and the number of parameters it takes.

`node`
: The class for function nodes (nodes with children). This is initialized with an `fwrapper`. When `evaluate` is called, it evaluates the child nodes and then applies the function to their results.

`paramnode`
: The class for nodes that only return one of the parameters passed to the program. Its `evaluate` method returns the parameter specified by `idx`.

`constnode`
: Nodes that return a constant value. The `evaluate` method simply returns the value with which it was initialized.

You'll also want some functions for the nodes to apply. To do this, you have to create functions and then give them names and parameter counts using `fwrapper`. Add this list of functions to *gp.py*:

```
addw=fwrapper(lambda l:l[0]+l[1],2,'add')
subw=fwrapper(lambda l:l[0]-l[1],2,'subtract')
mulw=fwrapper(lambda l:l[0]*l[1],2,'multiply')

def iffunc(l):
  if l[0]>0: return l[1]
  else: return l[2]
ifw=fwrapper(iffunc,3,'if')

def isgreater(l):
  if l[0]>l[1]: return 1
  else: return 0
gtw=fwrapper(isgreater,2,'isgreater')

flist=[addw,mulw,ifw,gtw,subw]
```

Some of the simpler functions such as `add` and `subtract` can be defined inline using lambda, while others require you to define the function in a separate block. In each case, they have been wrapped in an `fwrapper` with their names and the number of parameters required. The last line creates a list of all the functions so that later they can easily be chosen at random.

Building and Evaluating Trees

You can now construct the program tree shown in Figure 11-2 using the `node` class you just created. Add the `exampletree` function to *gp.py* to create the tree:

```
def exampletree( ):
  return node(ifw,[
                  node(gtw,[paramnode(0),constnode(3)]),
                  node(addw,[paramnode(1),constnode(5)]),
                  node(subw,[paramnode(1),constnode(2)]),
                  ]
              )
```

Start up a Python session to test your program:

```
>>> import gp
>>> exampletree=gp.exampletree( )
>>> exampletree.evaluate([2,3])
1
>>> exampletree.evaluate([5,3])
8
```

The program successfully performs the same function as the equivalent code block, so you've managed to build a mini tree-based language and interpreter within Python. This language can be easily extended with more node types, and it will serve as the basis for understanding genetic programming in this chapter. Try building a few other simple program trees to make sure you understand how they work.

Displaying the Program

Because you'll be creating program trees automatically and won't know what their structure looks like, it's important to have a way to display them so that you can easily interpret them. Fortunately the design of the `node` class means every node has a string representing the name of its function, so a display function simply has to return that string and the display strings of the child nodes. To make it easier to read, the display should also indent the child nodes so you can visually identify the parent-child relationships in the tree.

Create a new method in the `node` class called `display`, which shows a string representation of the tree:

```
def display(self,indent=0):
  print (' '*indent)+self.name
  for c in self.children:
    c.display(indent+1)
```

You'll also need to create a `display` method for the `paramnode` class, which simply prints the index of the parameter it returns:

```
def display(self,indent=0):
  print '%sp%d' % (' '*indent,self.idx)
```

And finally, one for the `constnode` class:

```
def display(self,indent=0):
  print '%s%d' % (' '*indent,self.v)
```

Use these methods to print out the tree:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> exampletree=gp.exampletree( )
>>> exampletree.display( )
if
 isgreater
  p0
  3
 add
  p1
  5
 subtract
  p1
  2
```

If you've read Chapter 7, you'll notice that this is similar to the way in which decision trees were displayed in that chapter. Chapter 7 also shows how to display those trees graphically for a cleaner, easier-to-read output. If you feel so inclined, you can use the same idea to build a graphical display of your tree programs.

Creating the Initial Population

Although it's possible to hand-create programs for genetic programming, most of the time the initial population consists of a set of random programs. This makes the process easier to start, since it's not necessary to design several programs that almost solve a problem. It also creates much more *diversity* in the initial population—a set of programs designed by a single programmer to solve a problem are likely to be very similar, and although they may give answers that are almost correct, the ideal solution make look quite different. You'll learn more about the importance of diversity shortly.

Creating a random program consists of creating a root node with a random associated function and then creating as many random child nodes as necessary, which in turn may have their own associated random child nodes. Like most functions that work with trees, this is most easily defined recursively. Add a new function, `makerandomtree`, to *gp.py*:

```
def makerandomtree(pc,maxdepth=4,fpr=0.5,ppr=0.6):
  if random( )<fpr and maxdepth>0:
    f=choice(flist)
    children=[makerandomtree(pc,maxdepth-1,fpr,ppr)
              for i in range(f.childcount)]
    return node(f,children)
  elif random( )<ppr:
    return paramnode(randint(0,pc-1))
  else:
    return constnode(randint(0,10))
```

This function creates a node with a random function and then looks to see how many child nodes this function requires. For every child node required, the function calls itself to create a new node. In this way an entire tree is constructed, with branches ending only if the function requires no more child nodes (that is, if the function returns a constant or an input variable). The parameter `pc`, used throughout this chapter, is the number of parameters that the tree will take as input. The parameter `fpr` gives the probability that the new node created will be a function node, and `ppr` gives that probability that it will be a `paramnode` if it is not a function node.

Try out this function in your Python session to build a few programs, and see what sort of results you get with different variables:

```
>>> random1=gp.makerandomtree(2)
>>> random1.evaluate([7,1])
7
>>> random1.evaluate([2,4])
2
>>> random2=gp.makerandomtree(2)
>>> random2.evaluate([5,3])
1
>>> random2.evaluate([5,20])
0
```

If all of a program's terminating nodes are constants, the program will not actually reference the input parameters at all, so the result will be the same no matter what input you pass to it. You can use the function defined in the previous section to display the randomly generated trees:

```
>>> random1.display( )
p0
>>> random2.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

You'll see that some of the trees get quite deep, since each branch will keep growing until it hits a zero-child node. This is why it's important that you include a maximum depth constraint; otherwise, the trees can get very large and potentially overflow the stack.

Testing a Solution

You would now have everything you'd need to build programs automatically, if you could just generate random programs until one is correct. Obviously, this would be ridiculously impractical because there are infinite possible programs and it's highly unlikely that you would stumble across a correct one in any reasonable time frame. However, at this point it is worth looking at ways to test a solution to see if it's correct, and if it's not, to determine how close it is.

A Simple Mathematical Test

One of the easiest tests for genetic programming is to reconstruct a simple mathematical function. Imagine you were given a table of inputs and an output that looked like Table 11-1.

Table 11-1. Data and result for an unknown function

X	Y	Result
26	35	829
8	24	141
20	1	467
33	11	1215
37	16	1517

There is some function that maps X and Y to the result, but you're not told what it is. A statistician might see this and try to do a regression analysis, but that requires guessing the structure of the formula first. Another option is to build a predictive model using k-nearest neighbors as you did in Chapter 8, but that requires keeping all the data. In some cases, you just need a formula, perhaps to codify in another much simpler program or to describe to other people what's going on.

I'm sure you're in suspense, so I'll tell you what the function is. Add `hiddenfunction` to *gp.py*:

```
def hiddenfunction(x,y):
    return x**2+2*y+3*x+5
```

You're going to use this function to build a dataset against which you can test your generated programs. Add a new function, `buildhiddenset`, which creates the dataset:

```
def buildhiddenset( ):
  rows=[]
  for i in range(200):
    x=randint(0,40)
    y=randint(0,40)
    rows.append([x,y,hiddenfunction(x,y)])
  return rows
```

And use this to create a dataset in your Python session:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> hiddenset=gp.buildhiddenset( )
```

Of course, you know what the function used to generate the dataset looks like, but the real test is whether genetic programming can reproduce it without being told.

Measuring Success

As with optimization, it's necessary to come up with a way to measure how good a solution is. In this case, you're testing a program against a numerical outcome, so an easy way to test a program is to see how close it gets to the correct answers for the dataset. Add `scorefunction` to *gp.py*:

```
def scorefunction(tree,s):
  dif=0
  for data in s:
    v=tree.evaluate([data[0],data[1]])
    dif+=abs(v-data[2])
  return dif
```

This function checks every row in the dataset, calculating the output from the function and comparing it to the real result. It adds up all the differences, giving lower values for better programs—a return value of 0 indicates that the program got every result correct. You can now test some of your generated programs in your Python session to see how they stack up:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.scorefunction(random2,hiddenset)
137646
>>> gp.scorefunction(random1,hiddenset)
125489
```

Since you only generated a few programs and they were generated completely randomly, the chance that one of them is actually the correct function is vanishingly small. (If one of your programs is the correct function, I suggest that you put the book down and go buy yourself a lottery ticket.) However, you now have a way to test how well a program performs on predicting a mathematical function, which is important for deciding which programs make it to the next generation.

Mutating Programs

After the best programs are chosen, they are replicated and modified for the next generation. As mentioned earlier, mutation takes a single program and alters it slightly. The tree programs can be altered in a number of ways—by changing the function on a node or by altering its branches. A function that changes the number of required child nodes either deletes or adds new branches, as shown in Figure 11-3.

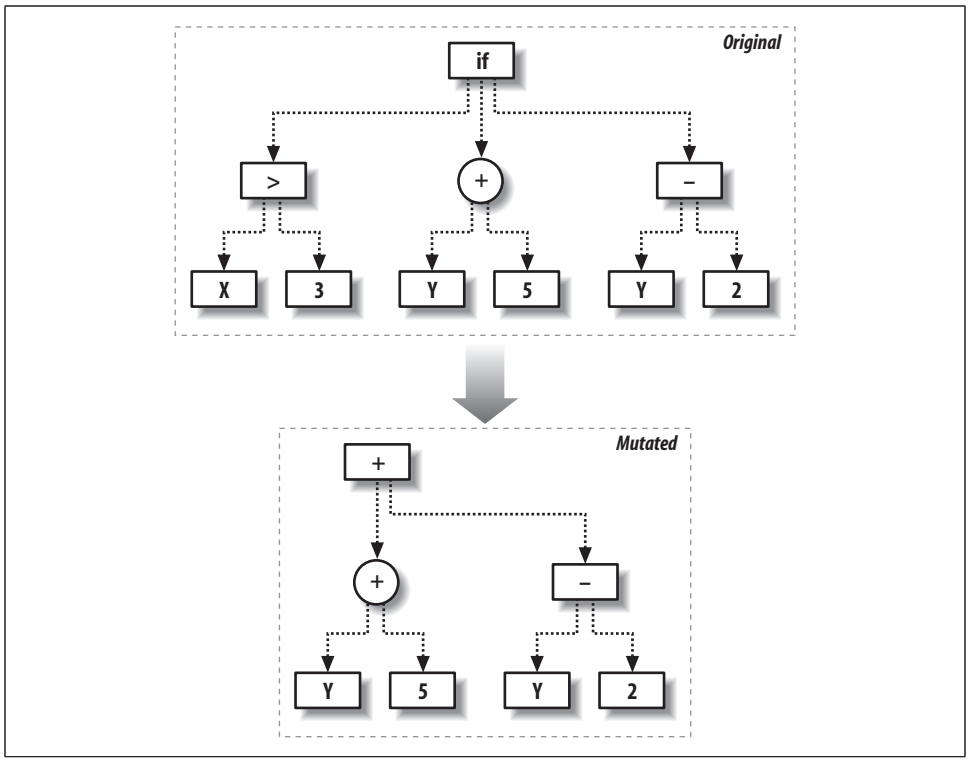

Figure 11-3. Mutation by changing node functions

The other way to mutate is by replacing a subtree with an entirely new one, as shown in Figure 11-4.

Mutation is not something that should be done too much. You would not, for instance, mutate the majority of nodes in a tree. Instead, you can assign a relatively small probability that any node will be modified. Beginning at the top of the tree, if a random number is lower than that probability, the node is mutated in one of the ways described above; otherwise, the test is performed again on its child nodes.

To keep things simple, the code given here only performs the second kind of mutation. Create a new function called `mutate` to perform this operation:

```
def mutate(t,pc,probchange=0.1):
  if random( )<probchange:
    return makerandomtree(pc)
  else:
    result=deepcopy(t)
    if isinstance(t,node):
      result.children=[mutate(c,pc,probchange) for c in t.children]
    return result
```

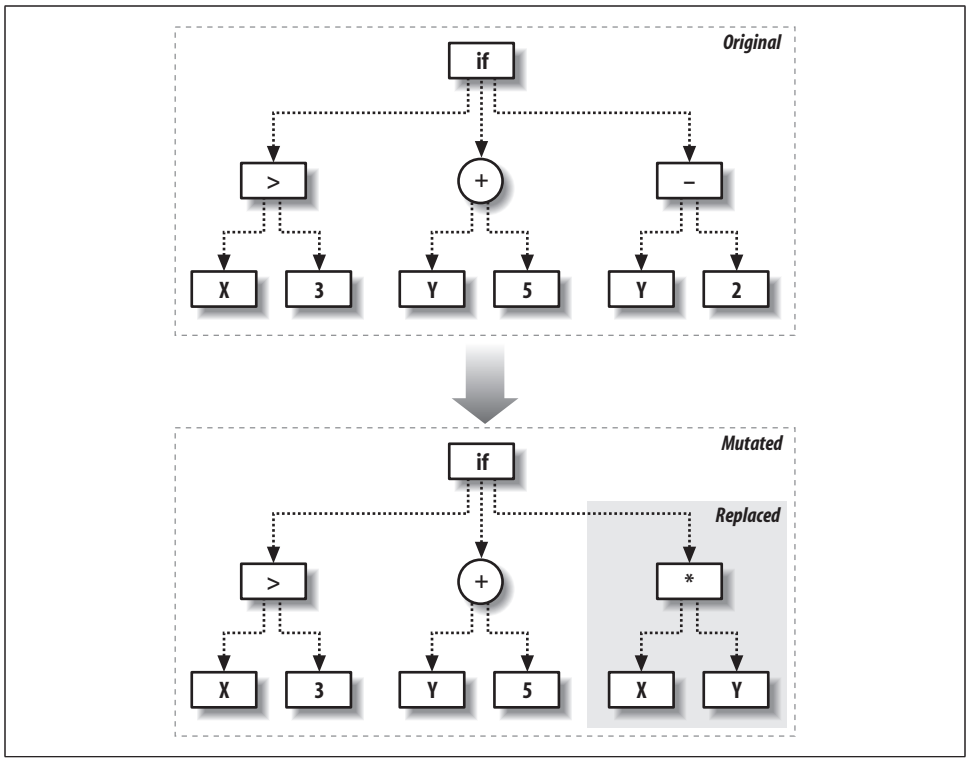

Figure 11-4. Mutation by replacing subtrees

This function begins at the top of the tree and decides whether the node should be altered. If not, it calls `mutate` on the child nodes of the tree. It's possible that the entire tree will be mutated, and it's also possible to traverse the entire tree without changing it.

Try running `mutate` a few times on the randomly generated programs you built earlier, and see how it modifies the trees:

```
>>> random2.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   2
```

```
>>> muttree=gp.mutate(random2,2)
>>> muttree.display( )
subtract
 7
 multiply
  isgreater
   p0
   p1
  if
   multiply
    p1
    p1
   p0
   p1
```

See if the result of `scorefunction` has changed significantly, for better or worse, after the tree has been mutated:

```
>>> gp.scorefunction(random2,hiddenset)
125489
>>> gp.scorefunction(muttree,hiddenset)
125479
```

Remember that the mutations are random, and they aren't necessarily directed toward improving the solution. The hope is simply that some will improve the result. These changes will be used to continue, and over several generations the best solution will eventually be found.

Crossover

The other type of program modification is crossover or breeding. This involves taking two successful programs and combining them to create a new program, usually by replacing a branch from one with a branch from another. Figure 11-5 shows an example of how this works.

The function for performing a crossover takes two trees as inputs and traverses down both of them. If a randomly selected threshold is reached, the function returns a copy of the first tree with one of its branches replaced by a branch in the second tree. By traversing both trees at once, the crossover happens at approximately the same level on each tree. Add the `crossover` function to *gp.py*:

```
def crossover(t1,t2,probswap=0.7,top=1):
  if random( )<probswap and not top:
    return deepcopy(t2)
  else:
    result=deepcopy(t1)
    if isinstance(t1,node) and isinstance(t2,node):
      result.children=[crossover(c,choice(t2.children),probswap,0)
                       for c in t1.children]
    return result
```

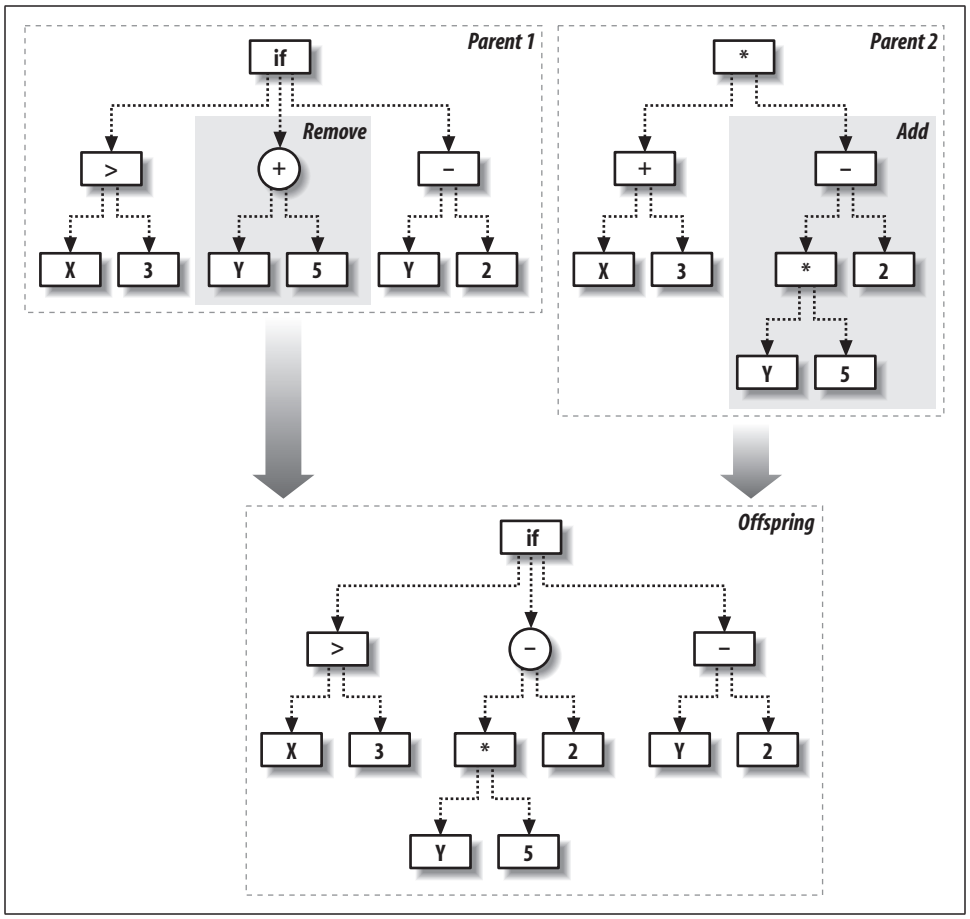

Figure 11-5. Crossover operation

Try `crossover` on a few of the randomly generated programs. See what they look like after the crossover, and see if crossing over two of the best programs occasionally leads to a better program:

```
>>> random1=gp.makerandomtree(2)
>>> random1.display( )
multiply
 subtract
  p0
  8
 isgreater
  p0
  isgreater
   p1
   5
```

```
>>> random2=gp.makerandomtree(2)
>>> random2.display( )
if
 8
 p1
 2
>>> cross=gp.crossover(random1,random2)
>>> cross.display( )
multiply
 subtract
  p0
  8
 2
```

You'll probably notice that swapping out branches can radically change what the program does. You may also notice that programs may be close to being correct for completely different reasons, so merging them produces a result that's very different from either of its predecessors. Again, the hope is that some crossovers will improve the solution and be kept around for the next generation.

Building the Environment

Armed with a measure of success and two methods of modifying the best programs, you're now ready to set up a competitive environment in which programs can evolve. The steps are shown in the flowchart in Figure 11-1. Essentially, you create a set of random programs and select the best ones for replication and modification, repeating this process until some stopping criteria is reached.

Create a new function called `evolve` to carry out this procedure:

```
def evolve(pc,popsize,rankfunction,maxgen=500,
           mutationrate=0.1,breedingrate=0.4,pexp=0.7,pnew=0.05):
  # Returns a random number, tending towards lower numbers. The lower pexp
  # is, more lower numbers you will get
  def selectindex( ):
    return int(log(random( ))/log(pexp))

  # Create a random initial population
  population=[makerandomtree(pc) for i in range(popsize)]
  for i in range(maxgen):
    scores=rankfunction(population)
    print scores[0][0]
    if scores[0][0]==0: break

    # The two best always make it
    newpop=[scores[0][1],scores[1][1]]
```

```
    # Build the next generation
    while len(newpop)<popsize:
      if random( )>pnew:
        newpop.append(mutate(
                      crossover(scores[selectindex( )][1],
                                scores[selectindex( )][1],
                               probswap=breedingrate),
                        pc,probchange=mutationrate))
      else:
      # Add a random node to mix things up
        newpop.append(makerandomtree(pc))

    population=newpop
  scores[0][1].display( )
  return scores[0][1]
```

This function creates an initial random population. It then loops up to `maxgen` times, each time calling `rankfunction` to rank the programs from best to worst. The best program is automatically passed through to the next generation unaltered, which is sometimes referred to as *elitism*. The rest of the next generation is constructed by randomly choosing programs that are near the top of the ranking, and then breeding and mutating them. This process repeats until either a program has a perfect score of 0 or `maxgen` is reached.

The function has several parameters, which are used to control various aspects of the environment. They are:

`rankfunction`
: The function used on the list of programs to rank them from best to worst.

`mutationrate`
: The probability of a mutation, passed on to `mutate`.

`breedingrate`
: The probability of crossover, passed on to `crossover`.

`popsize`
: The size of the initial population.

`probexp`
: The rate of decline in the probability of selecting lower-ranked programs. A higher value makes the selection process more stringent, choosing only programs with the best ranks to replicate.

`probnew`
: The probability when building the new population that a completely new, random program is introduced. `probexp` and `probnew` will be discussed further in the upcoming section "The Importance of Diversity."

The final thing you'll need before beginning the evolution of your programs is a way to rank programs based on the result of `scorefunction`. In *gp.py*, create a new function called `getrankfunction`, which returns a ranking function for a given dataset:

```
def getrankfunction(dataset):
  def rankfunction(population):
    scores=[(scorefunction(t,dataset),t) for t in population]
    scores.sort( )
    return scores
  return rankfunction
```

You're ready to automatically create a program that represents the formula for your mathematical dataset. Try this in your Python session:

```
>>> reload(gp)
>>> rf=gp.getrankfunction(gp.buildhiddenset( ))
>>> gp.evolve(2,500,rf,mutationrate=0.2,breedingrate=0.1,pexp=0.7,pnew=0.1)
16749
10674
5429
3090
491
151
151
0
add
 multiply
  p0
  add
   2
   p0
 add
  add
   p0
   4
  add
   p1
   add
    p1
    isgreater
     10
     5
```

The numbers change slowly, but they should decrease until they finally reach 0. Interestingly, the solution shown here gets everything correct, but it's quite a bit more complicated than the function used to create the dataset. (It's very likely that the solution you generated will also seem more complicated than it has to be.) However, a little algebra shows us that these functions are actually the same—remember that `p0` is X and `p1` is Y. The first line is the function represented by this tree:

```
 (X*(2+X))+X+4+Y+Y+(10>5)
= 2*X+X*X+X+4+Y+Y+1
= X**2 + 3*X + 2*Y + 5
```

This demonstrates an important property of genetic programming: the solutions it finds may well be correct or very good, but because of the way they are constructed, they will often be far more complicated than anything a human programmer would design. There will often be large sections of a program that don't do anything or that represent a complicated formula that returns the same value every time. Notice in the above example that the node (`10>5`) is just an odd way of saying 1.

It is possible to force the programs to remain simple, but in many cases this will make it more difficult to find a good solution. A better way to deal with this issue is to allow the programs to evolve to a good solution and then remove and simplify unnecessary portions of the tree. You can do this manually, and in some cases you can do it automatically using a pruning algorithm.

The Importance of Diversity

Part of the `evolve` function ranks the programs from best to worst, so it's tempting to just take two or three of the programs at the top and replicate and modify them to become the new population. After all, why would you bother allowing anything less than the best to continue?

The problem is that choosing only a couple of the top solutions quickly makes the population extremely homogeneous (or inbred, if you like), containing solutions that are all pretty good but that won't change much because crossover operations between them lead to more of the same. This problem is called reaching a *local maxima*, a state that is good but not quite good enough, and one in which small changes don't improve the result.

It turns out that having the very best solutions combined with a large number of moderately good solutions tends to lead to better results. For this reason, the `evolve` function has a couple of extra parameters that allow you to tune that amount of diversity in the selection process. By lowering the `probexp` value, you allow weaker solutions into the final result, turning the process from "survival of the fittest" to "survival of the fittest and luckiest." By increasing the `probnew` value, you allow completely new programs to be added to the mix occasionally. Both of these values increase the amount of diversity in the evolution process but won't disrupt it too much, since the very worst programs will always be eliminated eventually.

A Simple Game

A more interesting problem for genetic programming is building an AI for a game. You can force the programs to evolve by having them compete against each other and against real people, and giving the ones that win the most a higher chance of making it to the next generation. In this section, you'll create a simulator for a very simple game called Grid War, which is depicted in Figure 11-6.

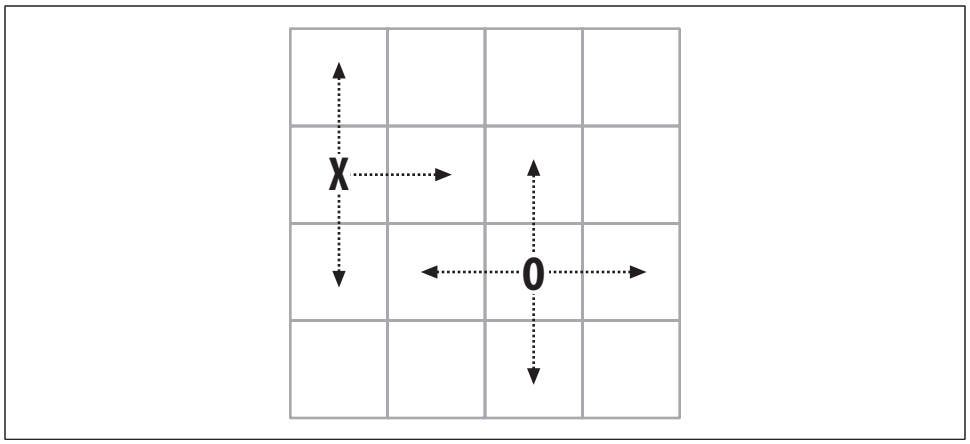

Figure 11-6. Grid War example

The game has two players who take turns moving around on a small grid. Each player can move in one of four directions, and the board is limited so if a player attempts to move off one side, he forfeits his turn. The object of the game is to capture the other player by moving onto the same square as his on your turn. The only additional constraint is that you automatically lose if you try to move in the same direction twice in a row. This game is very basic but since it pits two players against each other, it will let you explore more competitive aspects of evolution.

The first step is to create a function that uses two players and simulates a game between them. The function passes the location of the player and the opponent to each program in turn, along with the last move made by the player, and takes the return value as the move.

The move should be a number from 0 to 3, indicating one of four possible directions, but since these are random programs that can return any integer, the function has to handle values outside this range. To do this, it uses *modulo 4* on the result. Random programs are also liable to do things like create a player that moves in a circle, so the number of moves is limited to 50 before a tie is declared.

Add `gridgame` to *gp.py*:

```
def gridgame(p):
  # Board size
  max=(3,3)

  # Remember the last move for each player
  lastmove=[-1,-1]

  # Remember the player's locations
  location=[[randint(0,max[0]),randint(0,max[1])]]

  # Put the second player a sufficient distance from the first
  location.append([(location[0][0]+2)%4,(location[0][1]+2)%4])
```

```
  # Maximum of 50 moves before a tie
  for o in range(50):

    # For each player
    for i in range(2):
      locs=location[i][:]+location[1-i][:]
      locs.append(lastmove[i])
      move=p[i].evaluate(locs)%4

      # You lose if you move the same direction twice in a row
      if lastmove[i]==move: return 1-i
      lastmove[i]=move
      if move==0:
        location[i][0]-=1
        # Board limits
        if location[i][0]<0: location[i][0]=0
      if move==1:
        location[i][0]+=1
        if location[i][0]>max[0]: location[i][0]=max[0]
      if move==2:
        location[i][1]-=1
        if location[i][1]<0: location[i][1]=0
      if move==3:
        location[i][1]+=1
        if location[i][1]>max[1]: location[i][1]=max[1]

      # If you have captured the other player, you win
      if location[i]==location[1-i]: return i
  return -1
```

The program will return 0 if player 1 is the winner, 1 if player 2 is the winner, and –1 in the event of a tie. You can try building a couple of random programs and having them compete:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> p1=gp.makerandomtree(5)
>>> p2=gp.makerandomtree(5)
>>> gp.gridgame([p1,p2])
1
```

These programs are totally unevolved, so they probably lose by moving in the same direction twice in a row. Ideally, an evolved program will learn not to do this.

A Round-Robin Tournament

In keeping with collective intelligence, you would want the programs to test their fitness by playing against real people, and force their evolution that way. This would be a great way to capture the behavior of thousands of people and use it to develop a more intelligent program. However, with a large population and many generations,

this could quickly add up to tens of thousands of games, and most of them would be against very poor opponents. That's impractical for our purposes, so you can first have the programs evolve by competing against each other in a tournament.

The `tournament` function takes a list of players as its input and pits each one against every other one, tracking how many times each program loses its game. Programs get two points if they lose and one point if they tie. Add `tournament` to *gp.py*:

```
def tournament(pl):
  # Count losses
  losses=[0 for p in pl]

  # Every player plays every other player
  for i in range(len(pl)):
    for j in range(len(pl)):
      if i==j: continue

      # Who is the winner?
      winner=gridgame([pl[i],pl[j]])

      # Two points for a loss, one point for a tie
      if winner==0:
        losses[j]+=2
      elif winner==1:
        losses[i]+=2
      elif winner==-1:
        losses[i]+=1
        losses[i]+=1
        pass

  # Sort and return the results
  z=zip(losses,pl)
  z.sort( )
  return z
```

At the end of the function, the results are sorted and returned with the programs that have the fewest losses at the top. This is the return type needed by `evolve` to evaluate programs, which means that `tournament` can be used as an argument to `evolve` and that you're now ready to evolve a program to play the game. Try it in your Python session (this may take some time):

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> winner=gp.evolve(5,100,gp.tournament,maxgen=50)
```

As the programs evolve, notice that the loss numbers don't strictly decrease like they did with the mathematical function. Take a minute to think about why this is—after all, the best player is always allowed into the next generation, right? As it turns out, since the next generation consists entirely of newly evolved programs, the best program in one generation might fare a lot worse in the next.

Playing Against Real People

Once you've evolved a program that performs well against its robotic competitors, it's time to battle against it yourself. To do this, you can create another class that also has an evaluate method that displays the board to the user and asks what move they want to make. Add the `humanplayer` class to *gp.py*:

```
class humanplayer:
  def evaluate(self,board):

    # Get my location and the location of other players
    me=tuple(board[0:2])
    others=[tuple(board[x:x+2]) for x in range(2,len(board)-1,2)]

    # Display the board
    for i in range(4):
      for j in range(4):
        if (i,j)==me:
          print 'O',
        elif (i,j) in others:
          print 'X',
        else:
          print '.',
      print

    # Show moves, for reference
    print 'Your last move was %d' % board[len(board)-1]
    print ' 0'
    print '2 3'
    print ' 1'
    print 'Enter move: ',

    # Return whatever the user enters
    move=int(raw_input())
    return move
```

In your Python session, you can take on your creation:

```
>>> reload(gp)
<module 'gp' from 'gp.py'>
>>> gp.gridgame([winner,gp.humanplayer()])
. O . .
. . . .
. . . .
. . . X
Your last move was -1
 0
2 3
 1
Enter move:
```

Depending on how well your program evolved, you may find it easy or difficult to beat. Your program will almost certainly have learned that it can't make the same move twice in a row, since that leads to instant death, but the extent to which it has mastered other strategies will vary with each run of `evolve`.

Further Possibilities

This chapter is just an introduction to genetic programming, which is a huge and rapidly advancing field. You've used it so far to approach simple problems in which programs are built in minutes rather than days, but the principles can be extended to much more complex problems. The number of programs in the populations here have been very small compared to those used in more complex problems—a population of thousands or tens of thousands is more typical. You are encouraged to come up with more difficult problems and try larger population sizes, but you may have to wait hours or days while the programs run.

The following section outlines a few ways in which the simple genetic programming model can be extended for different applications.

More Numerical Functions

We have used a very small set of functions to construct the programs so far. This limits the scope of what a simple program can do—for more complicated problems, it's necessary to greatly increase the number of functions available to build a tree. Here are some possible functions to add:

- Trigonometric functions like sine, cosine, and tangent
- Other mathematical functions like power, square root, and absolute value
- Statistical distributions, such as a Gaussian
- Distance metrics, like Euclidean and Tanimoto distances
- A three-parameter function that returns 1 if the first parameter is between the second and third
- A three-parameter function that returns 1 if the difference between the first two parameters is less than the third

These can get as complicated as you like, and they are often tailored to specific problems. Trigonometric functions may be a necessity when working in a field like signal processing, but they are not much use in a game like the one you built in this chapter.

Memory

The programs in this chapter are almost entirely reactive; they give a result based solely on their inputs. This is the right approach for solving mathematical functions, but it doesn't allow the programs to work from a longer-term strategy. The chasing game passes the programs the last move they made—mostly so the programs learn they can't make the same move twice in a row—but this is simply the output of the program, not something they set themselves.

For a program to develop a longer-term strategy, it needs a way to store information for use in the next round. One simple way to do this is to create new kinds of nodes that can store and retrieve values from predefined slots. A *store* node has a single child and an index of a memory slot; it gets the result from its child and stores it in the memory slot and then passes this along to its parent. A *recall* node has no children and simply returns the value in the appropriate slot. If a store node is at the top of the tree, the final result is available to any part of the tree that has the appropriate recall node.

In addition to individual memory, it's also possible to set up shared memory that can be read and written to by all the different programs. This is similar to individual memory, except that there are a set of slots that all the programs can read from and write to, creating the potential for higher levels of cooperation and competition.

Different Datatypes

The framework described in this chapter is for programs that take integer parameters and return integers as results. It can easily be altered to work with float values, since the operations are the same. To do this, simply alter `makerandomtree` to create the constant nodes with a random float value instead of a random integer.

Building programs that handle other kinds of data will require more extensive modification, mostly changing the functions on the nodes. The basic framework can be altered to handle types such as:

Strings
: These would have operations like concatenate, split, indexing, and substrings.

Lists
: These would have operations similar to strings.

Dictionaries
: These would include operations like replacement and addition.

Objects
: Any custom object could be used as an input to a tree, with the functions on the nodes being method calls to the object.

An important point that arises from these examples is that, in many cases, you'll require the nodes in the tree to process more than one type of return value. A substring operation, for example, requires a string and two integers, which means that one of its children would have to return a string and the other two would have to return integers.

The naïve approach to this would be to randomly generate, mutate, and breed trees, simply discarding the ones in which there is a mismatch in datatypes. However, this would be computationally wasteful, and you've already seen how you can put a constraint on the way trees are constructed—every function in the integer trees knows how many children it needs, and this can be easily extended to constrain the types of children and their return types. For example, you might redefine the `fwrapper` class like the following, where `params` is a list of strings specifying datatypes that can be used for each parameter:

```
class fwrapper:
  def __init__(self,function,params,name):
    self.function=function
    self.childcount=param
    self.name=name
```

You'd also probably want to set up `flist` as a dictionary with return types. For example:

```
flist={'str':[substringw,concatw],'int':[indexw,addw,subw]}
```

Then you could change the start of `makerandomtree` to something like:

```
def makerandomtree(pc,datatype,maxdepth=4,fpr=0.5,ppr=0.5):
  if random( )<fpr and maxdepth>0:
    f=choice(flist[datatype])
    # Call makerandomtree with all the parameter types for f
    children=[makerandomtree(pc,type,maxdepth-1,fpr,ppr)
              for type in f.params]
    return node(f,children)
etc...
```

The `crossover` function would also have to be altered to ensure that swapped nodes have the same return type.

Ideally, this section has given you some ideas about how genetic programming can be extended from the simple model described here, and has inspired you to improve it and to try automatically generating programs for more complex problems. Although they may take a very long time to generate, once you find a good program, you can use it again and again.

Exercises

1. *More function types*. We started with a very short list of functions. What other functions can you think of? Implement a Euclidean distance node with four parameters.
2. *Replacement mutation*. Implement a mutation procedure that chooses a random node on the tree and changes it. Make sure it deals with function, constant, and parameter nodes. How is evolution affected by using this function instead of the branch replacement?
3. *Random crossover*. The current crossover function chooses branches from two trees at the same level. Write a different crossover function that crosses any two random branches. How does this affect evolution?
4. *Stopping evolution*. Add an additional criteria to `evolve` that stops the process and returns the best result if the best score hasn't improved within X generations.
5. *Hidden functions*. Try creating other mathematical functions for the programs to guess. What sort of functions can be found easily, and which are more difficult?
6. *Grid War player*. Try to hand-design your own tree program that does well at Grid War. If you find this easy, try to write another completely different one. Instead of having a completely random initial population, make it *mostly* random, with your hand-designed programs included. How do they compare to random programs, and can they be improved with evolution?
7. *Tic-tac-toe*. Build a tic-tac-toe simulator for your programs to play. Set up a tournament similar to the Grid War tournament. How well do the programs do? Can they ever learn to play perfectly?
8. *Nodes with datatypes*. Some ideas were provided in this chapter about implementing nodes with mixed datatypes. Implement this and see if you can evolve a program that learns to return the second, third, sixth, and seventh characters of a string (e.g., "genetic" becomes "enic").