Evolving Intelligence

Björn Lijnema
Overview

- Genetic programming
- Program representation
- Creating populations
- Testing solutions
- Mutation and crossover
- Selection
- An example
Genetic programming

• Compared with genetic algorithms
  – Genetic algorithms only search for suitable parameters for a pre-defined algorithm
  – Genetic programming searches for a suitable algorithm
Program representation

• Neural networks
• Linear genetic programming
• Trees
Programs as trees

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

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)
Programs as trees
addw=fwrapper(lambda x:x[0]+x[1],2,'add')
subw=fwrapper(lambda x:x[0]-x[1],2,'subtract')
mulw=fwrapper(lambda x:x[0]*x[1],2,'multiply')

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

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

flist=[addw,mulw,ifw,gtw,subw]
def exampeltree():
    return node(ifw,[
        node(gtw,[paramnode(0),constnode(3)]),
        node(addw,[paramnode(1),constnode(5)]),
        node(subw,[paramnode(1),constnode(2)]),
    ]
)
Programs as trees

```
if
 ├──>
 │   ├── X
 │   └── 3
 └── +
     ├── Y
     └── 5
 └── –
     ├── Y
     └── 2
```
Displaying the trees

For the node class

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

For the paramnode class

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

For the constnode class

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

```python
>>> tree=gp.exampletree()
>>> tree.display()
if
    isgreater
    p0
    3
    add
    p1
    5
    subtract
    p1
    2
```
Creating populations

• Automated is preferred over manual
  – Efficient
  – Ensures diversity
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))
Testing solutions

• Evaluate
• Ranking using a cost function
A simple test

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

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

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

• Mutation
  – Changes within one member of the population
  – For maintaining diversity

• Crossover
  – Changes using parts from different population members
  – For fitness improvement
Mutation

- Replacing
- Moving
- Swapping
- Scrambling
Node replacement
Branch replacement
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
Mutation

- Replacing
- Moving
- Swapping
- Scrambling
Crossover
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
Selection

• Proportionate selection
• Truncation selection
• Tournament selection
Population sizing

• Too small
  – Premature (suboptimal) convergence

• Too large
  – Computationally inefficient

• Solutions
  – Estimation
  – Experimental results
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))
    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]
The hidden function

def hiddenfunction(x,y):
    return x**2+2*y+3*x+5
An example: Grid War
Grid War

• Rules:
  – Players take turns moving
  – Players may not move twice in the same direction
  – Moving into a wall keeps the player stationary
  – A player wins when moving to the square inhabited by the opposing player
  – A tie is declared after 50 turns
Possible extensions

• Complex function nodes
  – Trigonometric and other mathematical functions
  – Statistical distributions
  – Distance metrics
  – Memory storage nodes
Possible extensions

• Different data types
  – Strings
  – Lists
  – Dictionaries
  – Objects
Conclusions

- Powerful mechanism
- Suited for complex problems