Berlin Brown and Software Development

Sunday, April 19, 2009

Java Genetic Algorithms Package(JGAP) and Scala

This is example source for a non-idiomatic example with JGap and Scala. The goal of this code is to give an input amount of change, say 30 cents and let the program calculate the least number of American coins. This is mostly a source example, there are some resources for learning Scala and JGap in the resources section at the bottom of this blog entry.

```
[java] Running Make Change
  [java] Input amount of change => 30
  [java] Total evolution time: 2661 ms
  [java] The best solution has a fitness value of 296.0
  [java] It contained the following:
  [java]  1 quarters.
  [java]  0 dimes.
  [java]  1 nickels.
  [java]  0 pennies.
  [java] Done
```

On Scala and JGap:

"Scala goes further than all other well-known languages in fusing object- oriented and functional programming into a uniform language design.
In addition to being a pure object-oriented language, Scala is also a full-blown functional language." -- Programming in Scala Book

"JGap provides basic genetic mechanisms that can be easily used to apply evolutionary principles to problem solutions." -- JGap Website

The Scala example consists of one scala source file. The original Java consists of two classes. It uses non-idiomatic Scala code, it is really just a quick example to show syntax differences with the Java version of the source.

```
/*
 * This was based on an example from JGAP.
 *
 * JGAP offers a dual license model containing the LGPL as well as the MPL.
 *
 * For licensing information please see the file license.txt included with JGAP
 * or have a look at the top of class org.jgap.Chromosome which representatively
 * includes the JGAP license policy applicable for any file delivered with JGAP.
 *
 * Original Authors:
 *
 * @author Neil Rotstan
 * @author Klaus Meffert
 *
 * Converted to Java Genetic Algorithms Example (Jgap) and Scala - Berlin Brown, 4/14/2009
 */

package org.bnode.genetics

import org.jgap._;
import org.jgap._;
import org.jgap.data._;
import org.jgap.impl._;
import org.jgap.xml._;
import org.w3c.dom._;

import java.io._;

/**
 * This class provides an implementation of the classic "Make change" problem
 * using a genetic algorithm. The goal of the problem is to provide a specified
 * amount of change (from a cash purchase) in the fewest coins possible. This
 * example implementation uses American currency (quarters, dimes, nickels, and
 * pennies).
 * <p>
 * This example may be seen as somewhat significant because it demonstrates the
 * use of a genetic algorithm in a less-than-optimal problem space. The genetic
 * algorithm does best when there is a smooth slope of fitness over the problem
 * space towards the optimum solution. This problem exhibits a more choppy space
 * with more local optima. However, as can be seen from running this example,
 * the genetic algorithm still will get the correct (or a very close) answer
 * virtually everytime.
 *
 */
```

```
object MakeChange {

  val MAX_ALLOWED_EVOLUTIONS = 600;

  /**
   * Amount of input change we are targetting.
   */
  val a_targetChangeAmount   = 30
  val mainPopulationSize     = 300

  /* ////////////////////////////////////////////////////////////// */

  /**
   * Main Entry Point for the application.
   */
  def main(args: Array[String]): Unit = {
    println("Running Make Change")
    println("Input amount of change => " +  a_targetChangeAmount)
    makeChangeForAmount()
    println("Done")
  } // End of the method

  /* ////////////////////////////////////////////////////////////// */

  /**
   * Sample fitness function for the MakeChange example.
   */
  class MakeChangeFunction(atargetAmount:Int) extends FitnessFunction {

    var targetAmount = 0
    val MAX_BOUND    = 600

    targetAmount = atargetAmount

    def getTotalNumberOfCoins(a_potentialsolution:IChromosome) : Int ={
      val numberOfGenes = a_potentialsolution.size()
      var i = 0
      var totalCoins = 0
      while (i < numberOfGenes) {
        totalCoins += getNumberOfCoinsAtGene(a_potentialsolution, i)
        i += 1
      }
      return totalCoins
    }
```

```
/**
 * Calculates the total amount of change (in cents) represented by the given
 * potential solution and returns that amount.
 *
 */
def amountOfChange(a_potentialSolution:IChromosome) : Int = {
  val numQuarters = getNumberOfCoinsAtGene(a_potentialSolution, 0)
  val numDimes = getNumberOfCoinsAtGene(a_potentialSolution, 1)
  val numNickels = getNumberOfCoinsAtGene(a_potentialSolution, 2)
  val numPennies = getNumberOfCoinsAtGene(a_potentialSolution, 3)
  return (numQuarters * 25) + (numDimes * 10) + (numNickels * 5) + numPennies
}

def changeDifferenceBonus(a_maxFitness:Double, a_changeDifference:Double) : Double = {

  if (a_changeDifference == 0) {
    return a_maxFitness
  } else {
    if ((a_changeDifference * a_changeDifference) >= (a_maxFitness / 2)) {
      return 0.0
    } else {
      return a_maxFitness / 2.0 - a_changeDifference * a_changeDifference
    } // End of if - else

  } // End of if - else
}

def computeCoinNumberPenalty(a_maxFitness:Double, a_coins:Int) : Double = {
  if (a_coins == 1) {
    return 0
  } else {
    return Math.min(a_maxFitness, a_coins * a_coins)
  }
}

/**
 * Determine the fitness of the given Chromosome instance. The higher the
 * return value, the more fit the instance. This method should always return
 * the same fitness value for two equivalent Chromosome instances.
 *
 */
def evaluate(a_subject:IChromosome) : Double = {

  val defaultComparation = a_subject.getConfiguration().getFitnessEvaluator().isFitter(2, 1)
```

```
    // The fitness value measures both how close the value is to the
    // target amount supplied by the user and the total number of coins
    // represented by the solution. We do this in two steps: first,
    // we consider only the represented amount of change vs. the target
    // amount of change and return higher fitness values for amounts
    // closer to the target, and lower fitness values for amounts further
    // away from the target. Then we go to step 2, which returns a higher
    // fitness value for solutions representing fewer total coins, and
    // lower fitness values for solutions representing more total coins.
    val changeAmount = amountOfChange(a_subject)
    val totalCoins = getTotalNumberOfCoins(a_subject)
    val changeDifference = Math.abs(targetAmount - changeAmount)

    var fitness = 0.0
    if (defaultComparation) {
      fitness = 0.0
    } else {
      fitness = MAX_BOUND / 2.0
    }

    if (defaultComparation) {
      fitness += changeDifferenceBonus(MAX_BOUND / 2, changeDifference)
    } else {
      fitness -= changeDifferenceBonus(MAX_BOUND / 2, changeDifference)
    }

    if (defaultComparation) {
      fitness -= computeCoinNumberPenalty(MAX_BOUND / 2.0, totalCoins.asInstanceOf[Int])
    } else {
      fitness += computeCoinNumberPenalty(MAX_BOUND / 2.0, totalCoins.asInstanceOf[Int])
    }

    return Math.max(1.0, fitness)
  }

  def getNumberOfCoinsAtGene(a_potentialSolution:IChromosome, a_position:Int) : Int = {
    val numCoins = a_potentialSolution.getGene(a_position).getAllele()
    return numCoins.asInstanceOf[Int]
  }

} // End of Class

/* ////////////////////////////////////////////////////////////// */

/**
```

```
 * Executes the genetic algorithm to determine the minimum number of coins
 * necessary to make up the given target amount of change. The solution will
 * then be written to System.out.
 */
def makeChangeForAmount() {

  val conf = new DefaultConfiguration();
  conf.setPreservFittestIndividual(true)
  val myFunc = new MakeChangeFunction(a_targetChangeAmount)
  conf.setFitnessFunction(myFunc)

  // Now we need to tell the Configuration object how we want our
  // Chromosomes to be setup. We do that by actually creating a
  // sample Chromosome and then setting it on the Configuration
  // object. As mentioned earlier, we want our Chromosomes to each
  // have four genes, one for each of the coin types. We want the
  // values (alleles) of those genes to be integers, which represent
  // how many coins of that type we have. We therefore use the
  // IntegerGene class to represent each of the genes. That class
  // also lets us specify a lower and upper bound, which we set
  // to sensible values for each coin type.

  val sampleGenes = new Array[IntegerGene](4);

  // Quarters
  sampleGenes(0) = new IntegerGene(conf, 0, 3 * 10)
  // Dimes
  sampleGenes(1) = new IntegerGene(conf, 0, 2 * 10)
   // Nickels
  sampleGenes(2) = new IntegerGene(conf, 0, 1 * 10)
  // Pennies
  sampleGenes(3) = new IntegerGene(conf, 0, 4 * 10)

  val sampleChromosome = new Chromosome(conf.asInstanceOf[Configuration],
                                        sampleGenes.asInstanceOf[Array[Gene]])
  conf.setSampleChromosome(sampleChromosome);
  conf.setPopulationSize(mainPopulationSize)
  val population = Genotype.randomInitialGenotype(conf)

  var i = 0;
  val startTime = System.currentTimeMillis()
  while (i < MAX_ALLOWED_EVOLUTIONS) {
    population.evolve();
    i += 1;
  } // End of While
```

```
    val endTime = System.currentTimeMillis()
    println("Total evolution time: " + (endTime - startTime) + " ms")

    val builder = DataTreeBuilder.getInstance()
    val doc2 = builder.representGenotypeAsDocument(population)

    // Write the XML document but do not read it.
    val docbuilder = new XMLDocumentBuilder()
    val xmlDoc = docbuilder.buildDocument(doc2)

    XMLManager.writeFile(xmlDoc.asInstanceOf[Document], new File("JGAPExample26.xml"))
    val bestSolutionSoFar = population.getFittestChromosome()

    // Print the best solutions
    println("The best solution has a fitness value of " + bestSolutionSoFar.getFitnessValue())
    println("It contained the following: ")
    println("\t" + myFunc.getNumberOfCoinsAtGene(bestSolutionSoFar, 0) + " quarters.")
    println("\t" + myFunc.getNumberOfCoinsAtGene(bestSolutionSoFar, 1) + " dimes.")
    println("\t" + myFunc.getNumberOfCoinsAtGene(bestSolutionSoFar, 2) + " nickels.")
    println("\t" + myFunc.getNumberOfCoinsAtGene(bestSolutionSoFar, 3) + " pennies.")

    true
  } // End of Method


} // End of Object
```

For comparison, you can see that some of the Java constructs look similar to the Scala version:

```
    public double evaluate(IChromosome a_subject) {

        boolean defaultComparation = a_subject.getConfiguration().getFitnessEvaluator().isFitter(2, 1);

        int changeAmount = amountOfChange(a_subject);
        int totalCoins = getTotalNumberOfCoins(a_subject);
        int changeDifference = Math.abs(m_targetAmount - changeAmount);
        double fitness;
        if (defaultComparation) {
            fitness = 0.0d;
        } else {
            fitness = MAX_BOUND / 2;
        }
```

```
        if (defaultComparation) {
            fitness += changeDifferenceBonus(MAX_BOUND / 2, changeDifference);
        } else {
            fitness -= changeDifferenceBonus(MAX_BOUND / 2, changeDifference);
        }

        if (defaultComparation) {
            fitness -= computeCoinNumberPenalty(MAX_BOUND / 2, totalCoins);
        } else {
            fitness += computeCoinNumberPenalty(MAX_BOUND / 2, totalCoins);
        }

        return Math.max(1.0d, fitness);
    }
```

Resources

This example was tested with Scala 2.7.3 and JGap 3.4.3

http://jgap.sourceforge.net/

http://www.scala-lang.org/

Full Source

The full source is hosted as a google code project, jvmnotebook. You can browse through the source or perform a svn checkout.

http://jvmnotebook.googlecode.com/svn/trunk/scala/jgap

About Me

Berlin Brown

He is a software developer with a diverse background in a multitude of different environments. He has worked with the CDC/SAIC, Geographic Information Systems (GIS) and now works for a Financial Services firm. You can find him freenode as blbrown and also visit botlist and botnode.com. Berlin can be found in Atlanta, Georgia. Also see botnode.com and on twitter. Please copyright any work to me (Berlin Brown) but you are free to do anything you like with it. All text is placed under a Creative Commons license. All code is placed under a New BSD license (unless noted otherwise).

View my complete profile