JGAP Frequently Asked Questions

[JGAP Home]

The following is a brief collection of questions and answers that come up periodically on the JGAP mailing lists. If you have suggestions for additional questions, please feel free to post them to the jgap-users mailing list!

1. Is there a way to create a fitness function that can evaluate all of the Chromosomes in a population at once?

   Yes, a bulk fitness-function can be used. The bulk fitness function is passed all of the Chromosomes in a population at once, and then is responsible for evaluating each of those Chromosomes and setting their fitness values. This can be especially useful when you want to evaluate Chromosomes in relationship to each other (rather than in isolation of each other), or if you want to make use of an external service to actually perform the evaluation.

   For more information on bulk fitness-functions, please see the javadocs for the org.jgap.BulkFitnessFunction class.

2. Does JGAP support real-valued fitness values or alleles?

   Yes, double typed fitness values have been introduced with release 1.1. Before that the type was int.

3. How can I select the top n performers of a population?

   During evolution, to select the best performing chromosomes for the next generation, use the `org.jgap.impl.BestChromosomesSelector`. See `org.jgap.impl.DefaultConfiguration` for an example.

   To select the top n performers after evolution, you could use `Genotype.getPopulation.toChromosomes()`. After that, sort the returned array of Chromosomes via `Arrays.sort(chromosomesArrayFromPopulation, yourComparator)`. `yourComparator` sorts the array of chromosomes by the fitness value of each Chromosome.

4. Does JGAP support multiple threads?

   JGAP does not currently offer any support for multi-threaded or distributed evaluation, but this is something we're considering for a future (after version 2.0) release.

5. I notice that sometimes the fittest chromosome doesn't get selected for the next generation. What's up with that?

   Natural selection in JGAP is statistical, so chromosomes that are more fit have a better statistical chance of being selected over chromosomes that are less fit, but it's not guaranteed. This is much like nature, where even the fittest of us can sometimes be unlucky!

   Incidentally, JGAP is designed to be very pluggable, and it's possible for you to write your

own natural selector and plug it into JGAP in place of the default version, if that better suits your needs.

6. Why aren't the chromosomes in my population ever candidates for natural selection?

   This should not happen if you properly use a NaturalSelector resp. if this selector is implemented correctly. The reproduction operator is deprecated in newer JGAP versions due to performance reasons and due to the fact that the current architecture delivers such functionality itself.

   Has anyone used JGAP in combination with the JOONE neural networks package?

   Yes, as a matter of a fact there is a whole open-source project dedicated to integration of JGAP with JOONE! You can find out more about the JOONEGAP project on their homepage.

   Additionally (among others), there is a dissertation about Genetic Algorithms playing together with Neural Nets to play Backgammon!

Copyright © 2002-2007 Klaus Meffert / Neil Rotstan. All rights reserved.

[JGAP Home]

Installing JGAP in 5 minutes

[JGAP Home]

Short introduction

JGAP is a framework written in Java. It is not an application you can run like an executable. But there is an example provided with JGAP you can run by using the command line (or a batch file, preferrably). As I am running Windows as operating system, the following description is somewhat orientated on that. But JGAP is a piece of Java and therefor able to be run on other systems as well!

Laying ground

1 First, download the JGAP package from the sourceforge site. You can use this link to get to the list of available JGAP files. Make sure you get the full installation (`jgap_xxx_full.zip`), not only the source code as the source would need to be compiled before executing anything! If you know how to compile Java source, then do it. This is a simple tutorial.

2 After that, extract the files to a directory of your choice, say to directory named `jgap-`

3 `current` somewhere on your harddisk.

Open this directory in a file manager of your choice, like Windows Explorer for Windows systems or another one for *NIX systems.
There is no need for registry entries, and no entry will be created in the start-menu of Windows.

4 In the directory you put JGAP in and opened in your explorer, there should be some files ending with `.jar`. These files contain the compiled JGAP distribution and that is all we need for now.

Running the example

5 Windows: Start the example by executing the batch file `start_example.bat`.

Other operating systems: Go to the command line and change your working directory to `jgap-current`. Now type the following:

```
java -classpath ".;jgap-examples.jar;jgap.jar"
      examples.MinimizingMakeChange 89
```

That's it! On your console there should appear something like (an example, your output may vary as a Genetic Algorithm involves randomness):

```
The best solution has a fitness value of 1599.0
It contained the following:
       3 quarters.
       1 dimes.
       0 nickels.
       4 pennies.
For a total of 89 cents in 8 coins.
```

Investigating the sources

6 If you want to code your own genetic algorithm by using JGAP, a good starting point is the examples provided with JGAP.

We suggest you have a look at the class `examples.simpleBoolean.SimpleExample`. Try to understand what happens there, modify and extend the example's code, run it and see what happens. Also have a look at the other examples shipped with JGAP.

Investigating the test cases

7 To get even more behind the scenes, investigate into the test cases. You find them under sub directory tests. It really might help studying the test code to understand how JGAP is supposed to work.

Still problems?

If you still have problems running the JGAP example or getting JGAP up and running in general, don't hesitate posting your question to the JGAP forum here! Or look at the main page of this site to get further contact information

Copyright © 2005-2007 Klaus Meffert. All rights reserved.

[JGAP Home]

A Brief Introduction to Genetic Algorithms

[JGAP Home]

What are Genetic Algorithms?

Genetic algorithms (GA's) are search algorithms that work via the process of natural selection. They begin with a sample set of potential solutions which then evolves toward a set of more optimal solutions. Within the sample set, solutions that are poor tend to die out while better solutions mate and propagate their advantageous traits, thus introducing more solutions into the set that boast greater potential (the total set size remains constant; for each new solution added, an old one is removed). A little random mutation helps guarantee that a set won't stagnate and simply fill up with numerous copies of the same solution.

In general, genetic algorithms tend to work better than traditional optimization algorithms because they're less likely to be led astray by local optima. This is because they don't make use of single-point transition rules to move from one single instance in the solution space to another. Instead, GA's take advantage of an entire set of solutions spread throughout the solution space, all of which are experimenting upon many potential optima.

However, in order for genetic algorithms to work effectively, a few criteria must be met:

- It must be relatively easy to evaluate how "good" a potential solution is relative to other potential solutions.
- It must be possible to break a potential solution into discrete parts that can vary independently. These parts become the "genes" in the genetic algorithm.
- Finally, genetic algorithms are best suited for situations where a "good" answer will suffice, even if it's not the absolute best answer.

Basic Mechanics of Genetic Algorithms

The basic operations of the genetic algorithm are simple and straight-forward:

Reproduction: The act of making a copy of a potential solution:

Crossover: The act of swapping gene values between two potential solutions, simulating the "mating" of the two solutions.

Mutation: The act of randomly altering the value of a gene in a potential solution.

Fitness Functions and Natural Selection

As mentioned earlier, it's necessary to be able to evaluate how "good" a potential solution is relative to other potential solutions. The "fitness function" is responsible for performing this evaluation and returning a positive integer number, or "fitness value", that reflects how optimal the solution is: the higher the number, the better the solution.

The fitness values are then used in a process of natural selection to choose which potential solutions will continue on to the next generation, and which will die out. It should be noted, however, that natural selection process does not merely choose the top *x* number of solutions; the solutions are instead chosen statistically such that it is more likely that a solution with a higher fitness value will be chosen, but it is not guaranteed. This tends to correspond to the natural world.

A common metaphor for the selection process is that of a large roulette wheel. Remembering that fitness values are positive integers, imagine that each potential solution gets a number of slots on the wheel equal to its fitness value. Then the wheel is spun and the solution on which it stops is selected. Statistically speaking, solutions with a higher fitness value will have a greater chance of being selected since they occupy more slots on the wheel, but even solutions with just a single slot still have a chance.

The above metaphor can also be useful for illustrating an additional point: relative fitness is measured as a *percentage*, not an absolute value. As a result, a solution with a fitness value of 255 has only about 0.4% chance of being selected over a solution with a fitness of 254. At higher fitness values, this problem is even more exaggerated. In order to keep your evolving solutions from stagnating at some point, it's important to implement your fitness function in such a way as to provide reasonable relative differences in solution fitness values at higher levels. A future version of JGAP may include an "auto-redistribution" feature to automatically exaggerate differences in near-optimal solutions, but for now a well-designed fitness function is required in these cases.

Learning more about Genetic Algorithms

To learn more about genetic algorithms, we highly recommend the book Genetic Algorithms in Search, Optimization, and Machine Learning by David Goldberg. To learn more about using JGAP, please see the tutorial: Getting Started with JGAP.

Copyright © 2002- 2007 Neil Rotstan / Klaus Meffert. All rights reserved.

[JGAP Home]

Getting Started With JGAP

[JGAP Home]

Preface: Please note that this documentation could be correlated to a former JGAP version. Therefor we recommend having a look at the examples provided with JGAP and then read this document in order to get an idea.

To use JGAP in an application, there are five basic things that you must do:

1. Plan your Chromosome.
2. Implement a "fitness function".
3. Setup a Configuration object.
4. Create a population of potential solutions.
5. Evolve the population!

Through the course of this tutorial, we'll build a simple example program that makes use of JGAP. The goal of our program will be to produce a user-specified amount of change in the fewest possible American coins (quarters, dimes, nickels, and pennies).

Step 1: Plan your Chromosome

At the heart of the genetic algorithm is the Chromosome. The Chromosome represents a potential solution and is divided into multiple genes. Genes in JGAP represent distinct aspects of the solution as a whole, just as human genes represent distinct aspects of individual people, such as their sex or eye color. During the JGAP evolution process, chromosomes are exposed to multiple genetic operators that represent mating, mutation, etc. and then are chosen for the next generation during a natural selection phase based upon their "fitness," which is a measure of how optimal that solution is relative to other potential solutions. The entire goal of the genetic algorithm is to mimic the natural process of evolution in order to produce superior solutions.

Step 1, therefore, is to decide on the makeup of your chromosomes, which includes how many genes you want and what those genes will represent. In our sample program, we want to create a pile of change that contains the fewest possible American coins that total up to the amount specified by the user. Since a chromosome represents a potential solution, in our sample it will represent a pile of change. We'll setup our genes to represent the different denominations of coins so that we'll have a total of four genes per chromosome (one each for quarters, dimes, nickels, and pennies).

Quarters	Dimes	Nickels	Pennies

Chromosome with four genes

We'll actually write the code to setup our Chromosome objects in step 3.

Step 2: Implementing a Fitness Function

JGAP is designed to do almost all of the evolutionary work for you in a relatively generic fashion. However, it has no knowledge of the specific problem you're actually trying to solve, and hence has no intrinsic way of deciding if one potential solution is any better than another potential solution for your specific problem. That's where the fitness function comes in: it's a single method that you must implement that accepts a potential problem solution and returns an integer value that indicates how good (or "fit") that solution is relative to other possible solutions. The higher the number, the better the solution. The lower the number (1 being the lowest legal fitness value), the poorer the solution. JGAP will use these fitness measurements to evolve the population of solutions toward a more optimal set of solutions.

Let's start with a fully-working example of a fitness function, the details of which will be explained below. Since the goal of our small program is to produce an amount of change in American coins equal to a target amount in the fewest possible number of coins, it seems reasonable that the measure of fitness for a particular solution would be a combination of the relative closeness of the amount of change it represented to the target amount of change, and the total number of coins represented by the solution (fewer being better).

```
package examples;

import org.jgap.Chromosome;
import org.jgap.FitnessFunction;

/**
 * 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).
 *
 * 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
 * answer virtually everytime.
 */
public class MinimizingMakeChangeFitnessFunction extends FitnessFunction
{
    private final int m_targetAmount;

    /**
     * Constructs this MinimizingMakeChangeFitnessFunction with the desired
     * amount of change to make.
     *
     * @param a_targetAmount The desired amount of change, in cents. This
     *                       value must be between 1 and 99 cents.
     */
    public MinimizingMakeChangeFitnessFunction( int a_targetAmount )
    {
        if( a_targetAmount < 1 || a_targetAmount > 99 )
        {
            throw new IllegalArgumentException(
                "Change amount must be between 1 and 99 cents." );
        }

        m_targetAmount = a_targetAmount;
    }
```

```java
/**
 * 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.
 *
 * @param a_subject: The Chromosome instance to evaluate.
 *
 * @return A positive integer reflecting the fitness rating of the given
 *         Chromosome.
 */
public double evaluate( Chromosome a_subject )
{
    // 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 calculate higher fitness values for amounts
    // closer to the target, and lower fitness values for amounts further
    // away from the target. If the amount equals the target, then we go
    // to step 2, which adjusts the fitness to a higher value for
    // solutions representing fewer total coins, and lower fitness
    // values for solutions representing a larger total number of coins.
    // ------------------------------------------------------------------
    int changeAmount = amountOfChange( a_subject );
    int totalCoins = getTotalNumberOfCoins( a_subject );
    int changeDifference = Math.abs( m_targetAmount - changeAmount );

    // Step 1: Determine the distance of the amount represented by the
    // solution from the target amount. Since we know the maximum amount
    // of change is 99 cents, we'll subtract from that the difference
    // between the solution amount and the target amount. That will give
    // the desired effect of returning higher values for amounts close
    // to the target amount and lower values for amounts further away
    // from the target amount.
    // ------------------------------------------------------------------
    double fitness = ( 99 - changeDifference );

    // Step 2: If the solution amount equals the target amount, then
    // we add additional fitness points for solutions representing fewer
    // total coins.
    // -----------------------------------------------------------------
    if( changeAmount == m_targetAmount )
    {
        fitness += 100 - ( 10 * totalCoins );
    }

    return fitness;
}


/**
 * Calculates the total amount of change (in cents) represented by
 * the given chromosome and returns that amount.
 *
 * @param a_potentialSolution The potential solution to evaluate.
 * @return The total amount of change (in cents) represented by the
 *         given solution.
 */
public static int amountOfChange( Chromosome a_potentialSolution )
{
    int numQuarters = getNumberOfCoinsAtGene( a_potentialSolution, 0 );
    int numDimes = getNumberOfCoinsAtGene( a_potentialSolution, 1 );
    int numNickels = getNumberOfCoinsAtGene( a_potentialSolution, 2 );
    int numPennies = getNumberOfCoinsAtGene( a_potentialSolution, 3 );

    return ( numQuarters * 25 ) + ( numDimes * 10 ) + ( numNickels * 5 ) +
           numPennies;
}
```

```java
    /**
     * Retrieves the number of coins represented by the given potential
     * solution at the given gene position.
     *
     * @param a_potentialSolution The potential solution to evaluate.
     * @param a_position The gene position to evaluate.
     * @return the number of coins represented by the potential solution
     *         at the given gene position.
     */
    public static int getNumberOfCoinsAtGene( Chromosome a_potentialSolution,
                                              int a_position )
    {
        Integer numCoins =
          (Integer) a_potentialSolution.getAllele( a_position ).getValue();

        return numCoins.intValue();
    }


    /**
     * Returns the total number of coins represented by all of the genes in
     * the given chromosome.
     *
     * @param a_potentialsolution The potential solution to evaluate.
     * @return The total number of coins represented by the given Chromosome.
     */
    public static int getTotalNumberOfCoins( Chromosome a_potentialsolution )
    {
        int totalCoins = 0;

        int numberOfGenes = a_potentialsolution.size();
        for( int i = 0; i < numberOfGenes; i++ )
        {
            totalCoins += getNumberOfCoinsAtGene( a_potentialsolution, i );
        }

        return totalCoins;
    }
}
```

Let's tackle our example fitness function bit by bit. To start, we define our own class and extend the org.jgap.FitnessFunction class. All fitness functions must extend the FitnessFunction class. We then define a constructor and an evaluate() method. The evaluate() method is a standard method that all fitness functions must implement. That is the method that will be called by the genetic engine when it needs to know the fitness value of a chromosome.

Our constructor isn't very exciting: it merely accepts a target change amount that the user desires, verifies that the amount meets our constraint of being between 1 and 99 cents, and then stores the amount in an instance variable for later use.

The interesting part of the whole class is the evaluate() method, which is where the work is done. The evaluate method is always passed in a Chromosome, which represents a potential solution. A Chromosome is made up of genes, each of which represents a respective part of the solution. In our example, the Chromosome represents an amount of change, while the genes represent the specific kinds of coins: quarters for the first gene, dimes for the second gene, nickels for the third gene, and pennies for the fourth gene. The value of a gene is called an allele. In our example, the allele would be the number of a given type of coin (for example, 2 pennies).

The first thing the evaluate() method does is invoke a couple of helper methods which conveniently return the total amount of change that is represented by the potential solution and the total number of coins represented by the solution. We'll take a closer look at how these work

later. It then subtracts the amount of change represented by the solution from the target amount, and takes the absolute value of the difference to measure how close the solution amount is to the target amount. We then set about calculating our fitness value.

As the comments indicate, we're going to calculate fitness in two stages. The first stage calculates an initial fitness value based on how far away the solution amount is from the target amount. Then, if the solution amount happens to equal the target amount, we go to stage two, which adjusts the fitness value based upon the total number of coins represented by the solution. In the end, we want to return high fitness values for solutions that match the target amount with very few coins, and return lower fitness values for solutions that are far away from the target amount or represent a large number of coins.

Moving beyond the evaluate() method, we encounter those helper methods we mentioned earlier. The amountOfChange() method calculates the total amount of change (in cents) represented by a Chromosome that is passed to it. Internally, it defers to the getNumberOfCoinsAtGene() method to actually extract the number of each type of coin. It then calculates the total amount of change and returns it.

The next convenience method, getNumberOfCoinsAtGene(), is responsible for determining the number of coins represented by a specific gene in the given Chromosome. As mentioned earlier, the value of each gene in the Chromosome is called an allele. This method gets the allele for the gene at the provided position in the Chromosome and then returns it as an int primitive.

Finally, there's a getTotalNumberOfCoins() method that determines the total number of coins represented by a given Chromosome. It simply tallies up the number of coins represented by each gene--using the getNumberOfCoinsAtGene() method--and then returns the tally.

And that's the end of the fitness function. If you're feeling a little bit overwhelmed, don't worry about it and take some comfort in the fact that it's all down hill from here! The fitness function is the hardest part of using JGAP and, after writing a few, you'll get the hang of it.

Step 3: Setup a Configuration Object

JGAP is designed to be very flexible and pluggable. If you want, you can create your own genetic operators, random number generators, natural selectors, and so on. To support all of this, JGAP uses a Configuration object that must be setup with all of the settings you want prior to using the genetic engine. Fortunately, we realize that most people will want to use the stock components, and so we include a DefaultConfiguration class that comes already setup with the most common settings. You just need to provide three extra pieces of information: what fitness function you want to use, how you want your Chromosomes to be setup, and how many Chromosomes you want in your population. Let's look at some sample code.

```
// Start with a DefaultConfiguration, which comes setup with the
// most common settings.
// -------------------------------------------------------------
Configuration conf = new DefaultConfiguration();

// Set the fitness function we want to use, which is our
// MinimizingMakeChangeFitnessFunction that we created earlier.
// We construct it with the target amount of change provided
// by the user.
// ------------------------------------------------------------
FitnessFunction myFunc =
    new MinimizingMakeChangeFitnessFunction( targetAmount );

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 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.
// --------------------------------------------------------------
Gene[] sampleGenes = new Gene[ 4 ];

sampleGenes[0] = new IntegerGene( 0, 3 );  // Quarters
sampleGenes[1] = new IntegerGene( 0, 2 );  // Dimes
sampleGenes[2] = new IntegerGene( 0, 1 );  // Nickels
sampleGenes[3] = new IntegerGene( 0, 4 );  // Pennies

Chromosome sampleChromosome = new Chromosome( sampleGenes );

conf.setSampleChromosome( sampleChromosome );

// Finally, we need to tell the Configuration object how many
// Chromosomes we want in our population. The more Chromosomes,
// the larger the number of potential solutions (which is good
// for finding the answer), but the longer it will take to evolve
// the population each round. We'll set the population size to
// 500 here.
// --------------------------------------------------------------
conf.setPopulationSize( 500 );
```

Hopefully most of the above code is pretty self-explanatory, with maybe the exception of setting up the sample Chromosome. Let's look at that bit in a little more detail.

As mentioned earlier, a Chromosome is made up of genes. JGAP lets you choose what Gene class to use to represent each gene in the Chromosome (for more information on creating custom Gene classes, please see the Creating Custom Genes document). That provides the most flexibility and convenience. If we wanted to, we could have actually written a separate Gene class for each coin type in our example, such as a QuarterGene, DimeGene, NickelGene, and PennyGene. In fact, we look at what a QuarterGene might look like in the *Creating Custom Genes* document that we just mentioned.

As it happens, we decided that the IntegerGene (which comes with JGAP) would suffice. You'll notice, however, that we did take advantage of the ability to specify different Gene implementations for each gene in the Chromosome by creating separate IntegerGenes with different lower and upper bounds for each coin type. We set the upper bounds to be the largest number of coins of that respective type that would appear in an optimal solution. Limiting the solution space this way helps JGAP arrive at better solutions with fewer evolutions.

So to get back to the code, we first create an array of Genes with a length of 4, since we want to represent 4 genes (one for each coin type). We then set each Gene in the array to an IntegerGene that is constructed with appropriate lower and upper bounds for that coin type. Finally, we construct a new Chromosome and pass it the array of Genes, and then set that sample Chromosome on the Configuration object.

The final part of this step is setting the population size, which is the number of Chromosomes we want in the population. A larger population size means more potential solutions to choose from and more genetic diversity, but it also means more work to evolve that population. Ultimately, you'll need to settle on a value that balances your need for a nice selection of potential solutions against how much time you're willing to spend waiting for your population to evolve. The population size is chosen higher than the number of possible solutions here (as 3*2*1*4 = 24). But as Genetic Algorithms is a stochastic system, the more tries the GA is

allowed to make, the higher the chances finding a good solution. Besides, the population size should not be correlated too specifically to a narrowed search space, but be usable in a more generic way. Meaning: if the 24 possible combinations mentioned above was raised, then the population wouldn't need to be adapted. And if the 24 is kept, the population size of 500 will not have a negative impact, because fewer evolutions are needed finding a good solutions than with smaller population size.

Step 4: Create a Population

Recall that each potential solution is represented by a Chromosome. A population of Chromosomes is called a Genotype, and that is the class we need to construct to create our population. If you want, you can construct each Chromosome individually and then pass them all into a new Genotype (much like we constructed each Gene and passed them into the sample Chromosome in step 3), but JGAP provides a much quicker and easier way of creating a random population. In fact, it only takes one line of code!

```
Genotype population = Genotype.randomInitialGenotype( conf );
```

The randomInitialGenotype() method takes in a Configuration object (which we setup in step 3) and returns a Genotype with the correct number of Chromosomes, each of which has its genes set to random values. In other words, it generates a random population. For most applications, this is all that's necessary to create your initial population of potential solutions.

Step 5: Evolve the Population!

Now that we've gotten everything setup and ready to go, it's time to start evolving the population until it contains some potential solutions that we're satisfied with. Evolving the population one cycle is another one-liner:

```
population.evolve();
```

Typically, after each evolution cycle, you'll want to check if the population contains any satisfactory solutions. The easiest way to do this is to invoke the getFittestChromosome() method on the population:

```
Chromosome bestSolutionSoFar = population.getFittestChromosome();
```

If the best solution so far is good enough for you, then you're done. If not, then you can evolve the population again. Alternatively, you may just choose to evolve the population a set number of times and then see what the best solution is that was produced at the end (or a combination thereof). For our example problem, we'll take this latter approach.

```
Chromosome bestSolutionSoFar;

for( int i = 0; i < MAX_ALLOWED_EVOLUTIONS; i++ )
{
    population.evolve();
}

System.out.println( "The best solution contained the following: " );

System.out.println(
    MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(
        bestSolutionSoFar, 0 ) + " quarters." );

System.out.println(
    MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(
```

```
        bestSolutionSoFar, 1 ) + " dimes." );

System.out.println(
    MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(
        bestSolutionSoFar, 2 ) + " nickels." );

System.out.println(
    MinimizingMakeChangeFitnessFunction.getNumberOfCoinsAtGene(
        bestSolutionSoFar, 3 ) + " pennies." );

System.out.println( "For a total of " +
    MinimizingMakeChangeFitnessFunction.amountOfChange(
        bestSolutionSoFar ) + " cents in " +
    MinimizingMakeChangeFitnessFunction.getTotalNumberOfCoins(
        bestSolutionSoFar ) + " coins." );
```

Now we've got ourselves a full-fledged genetic application! To view all of the code for this example, see the MinimizingMakeChange.java and MinimizingMakeChangeFitnessFunction.java files in the src/examples/ directory of the JGAP distribution.

Copyright © 2002-2007 Neil Rotstan / Klaus Meffert. All rights reserved.

[JGAP Home]

Genetic Programming with JGAP

[Documentation Index]

Genetic Programming (GP) is supposed to be introduced by John Koza. GP is sort of different to Genetic Algorithms (GA) as GP is more complex than GA. GP is about evolving programs with a given set of commands. Each command is assigned a number of parameters. That way, programs can be constructed. They are represented internally as a tree. But you do not have to cope with the internal representation. For you it is only important to know that you can write sort of programs. Each program contains only those commands and terminals (such as numbers or Boolean values) that you define to be used! It is even possible evolving full-blown Java programs (see the Robocode example).

GP was introduced with JGAP 3.0. This document presents some useful information about how to utilize JGAP for GP.

JGAP's Philosophy

The main philosophy that was setup before extending JGAP towards GP was to reach the goal by adding parts to GA to get GP. Thus: GA + x =GP, with x being the added parts.

A first start

To get a feeling of how to implement your GP-Program with JGAP, please have a look at the example located in class *examples.gp.MathProblem*. This example tries to find a formula for a given truth table (X/Y value pairs) and a given set of operators (such as +, -, * etc). The mentioned example class also contains a fitness function named *FormulaFitnessFunction*. A more complex example can be found in class *examples.gp.Fibonacci* and in class *examples.gp.anttrail.AntTrailProblem*.

The math example at a glance:

1. Create configuration

```
/**
 * Starts the example
 * @param args ignored
 * @throws Exception
 */
public static void main(String[] args) throws Exception {
  GPConfiguration config = new GPConfiguration();
  config.setGPFitnessEvaluator(new DeltaGPFitnessEvaluator());
  config.setMaxInitDepth(6);
  config.setPopulationSize(100);
  config.setFitnessFunction(new MathProblem.FormulaFitnessFunction());
  ... // continued below
```

2. Create initial Genotype (continued from above method *main*):

```
  GPGenotype gp = create(config);
}


public static GPGenotype create(GPConfiguration a_conf) throws
InvalidConfigurationException {
  Class[] types = {
    CommandGene.FloatClass};
  Class[][] argTypes = {
    {}
  };
  // Define the commands and terminals the GP is allowed to use.
  // -----------------------------------------------------------
  CommandGene[][] nodeSets = {
    {
    vx = Variable.create(a_conf, "X", CommandGene.FloatClass),
    new AddCommand(a_conf, CommandGene.FloatClass),
    new SubtractCommand(a_conf, CommandGene.FloatClass),
    new MultiplyCommand(a_conf, CommandGene.FloatClass),
    new DivideCommand(a_conf, CommandGene.FloatClass),
    new SinCommand(a_conf, CommandGene.FloatClass),
    new CosCommand(a_conf, CommandGene.FloatClass),
    new ExpCommand(a_conf, CommandGene.FloatClass),
    // Use terminal with possible value from 2.0 to 10.0 decimal
    new Terminal(conf, CommandGene.FloatClass, 2.0d, 10.0d, false),
    }
  };
```

```
  Random random = new Random();
  // Randomly initialize function data (X-Y table) for x^4+x^3+x^2+x.
  // This is problem-specific and not necessary in other cases.
  // ---------------------------------------------------------------
  for (int i = 0; i < 20; i++) {
    float f = 2.0f * (random.nextFloat() - 0.5f);
    x[i] = new Float(f);
    y[i] = f * f * f * f + f * f * f + f * f - f;
    System.out.println(i + ") " + x[i] + " " + y[i]);
  }
  // Create genotype with initial population.
  // Allow max. 100 nodes within one program.
  // ----------------------------------------
  return randomInitialGenotype(a_conf, types, argTypes, nodeSets, 100,
true);
}
```

3. Start the evolution and output the result: (continued in above method *main*):

```
// Do 100 evolutions in a row.
// ---------------------------
  gp.evolve(100);
// Output best solution found.
// ---------------------------
gp.outputSolution(gp.getAllTimeBest());

} // end of method "main"
```

4. Optional: Implement your own functions and terminals

Please have a look at packages *org.jgap.gp.function*, *org.jgap.gp.terminal* and examples.gp.anttrail for available GP functions and terminals. There you can see how to implement your own classes and use them in the above step 2 during setup.

You can benefit from looking at the examples mentioned above. Go for the code and you'll get a better feeling. It's very easy using JGAP for GP!

Copyright © 2006-2007 Klaus Meffert. All rights reserved.

[Documentation Index]

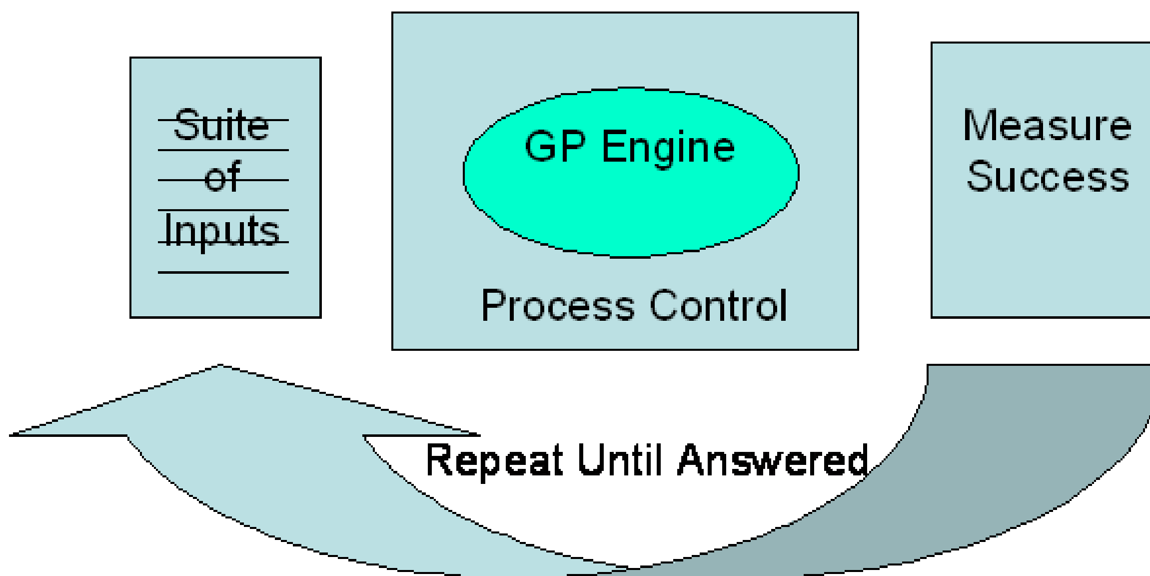
SOURCEFORGE.NET®

Solving Problems using Genetic Programming with JGAP

Identifying Candidate Problems

Genetic Programming has been applied to a wide variety of problems. In particular, Koza [TBD

reference] has demonstrated that many problems can be reformulated so that they share a similar software architecture and so that they can be solved with Genetic Programming techniques. This architecture consists of presenting the genetic programming component with a suite of inputs, running the inputs through a series of genetically generated programs, and then measuring the success of each program against the expected answer for the corresponding input. The process is repeated again and again until the success measure meets the exit criteria you've provided.



The first challenge is identifying an interesting problem and determining that the selected problem has a reasonable chance for success. The key characteristic about finding a genetic programming solution to a problem is that you are trading computer processing effort against human development effort. There are problems that can be developed with traditional programming that can be also solved with genetic programming. However, you may find in some cases that the effort to create the genetic programming software architecture above is equal to or greater than the effort to create the solution with traditional programming.

However, you might receive by creating a genetic solution to a previously solved problem or a simple problem. I definitely learned about using JGAP by trying to solve a simple problem and to repeat a research result. This tutorial will start by identifying how to use JGAP to find programs using a previously solved genetic problem.

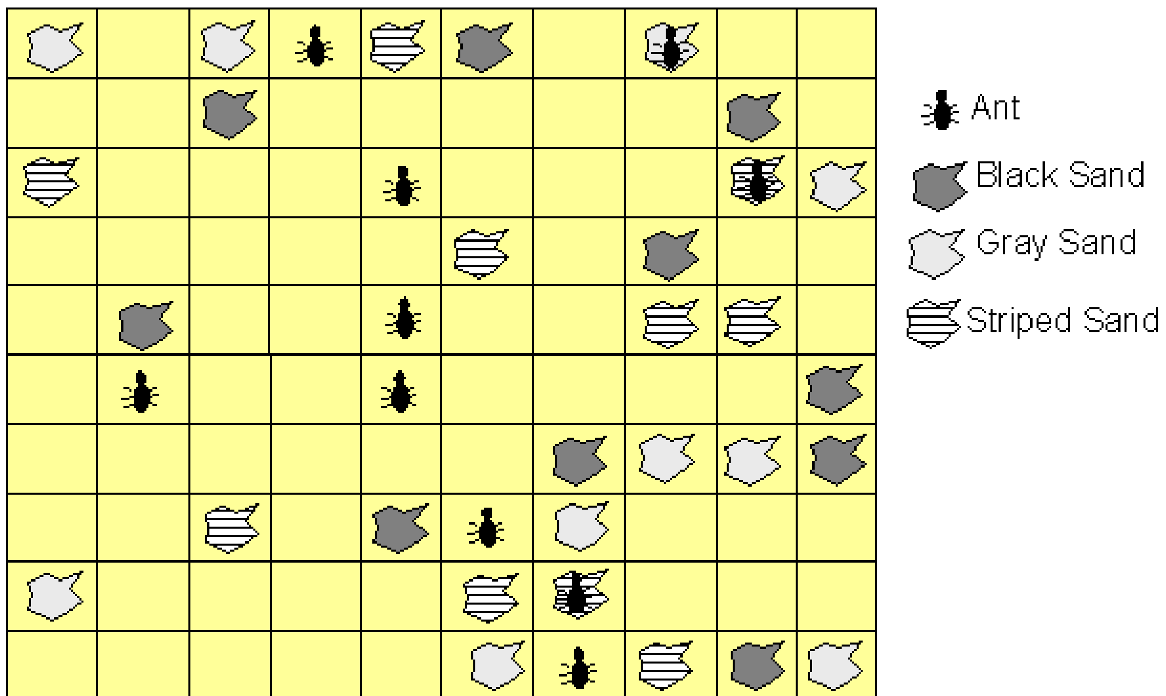
Problem Statement

For this tutorial, we'll repeat the solution that Koza found in his paper, "**Evolution of Emergent Cooperative Behavior using Genetic Programming".** This paper can be found at [TBD]. We'll first solve this problem by carefully repeating the elements that Koza used in the JGAP system. This means that we'll program Java classes that emulate the Lisp functions that Koza used in his paper. The intent of starting with this approach is that we can learn how to use JGAP with a problem that has a known solution. Once we've learned lessons about using JGAP by repeating this solution, then we can proceed to solving the problem with a more modern approach. At the current time, I've only

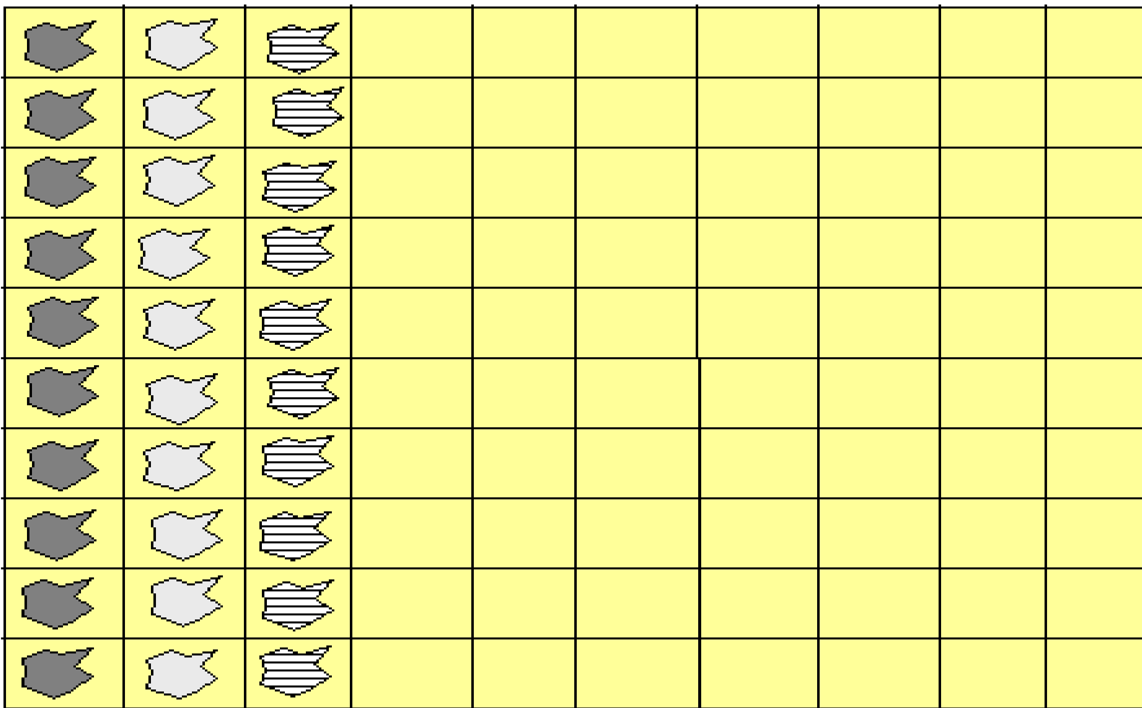
implemented the lisp-like solution and it conversion of this solution to a more modern approach may be a exercise for the reader.

Painted Desert Ant Problem

The problem consists of 10 ants in a 10x10 toroidal grid with 10 black sand grains, 10 gray sand grains, and 10 striped sand grains. The goal is to find a single program that when run on each ant, moves the grains of sands into the first three columns.



Starting map


Final Map with Sand aligned into columns

Selecting the Function Set

For a problem with an unknown solution, we need to identify a set of functions that are sufficient to solve the problem. For our problem, we are using the function set identified by Koza.

Terminals = {X, Y, Carrying, SandColor, GO_N, GO_E, GO_S, GO_W, MoveRandom, Pickup}

Functions = {IfLessThanOrEqual, IfLessThanZero, IfDrop}

For the first solution, we'll be mimicking the Lisp programming language by having each function return an integer value according to the rules in the paper.

Creating the Domain classes

I chose to implement the business logic in two classes. The AntMap contains the knowledge of the locations and the set of ants. It is responsible for loading the problem from a file location that contains the sand and ant locations. It is also responsible for presenting the current state of the map on System.out. The ant class is responsible for any ant specific behavior. The approach of using the domain classes allowed unit testing the behavior outside of the JGAP framework and subsequently creating JGAP based unit tests to compare against the domain behavior.

The domain classes can easily be modified to a more modern approach by changing the return values from int's to void's. These procedures can then be mapped to appropriate genetic classes.

Creating the Function Set

Now that we know the functions that we want to try for our program, we need to convert the functions into appropriate JGAP compatible classes. For each terminal and each function listed above there is a corresponding Java class. You can browse the classes in the paintedDesert package. For this tutorial, we'll focus on the key concepts to integrate with the JGAP framework.

Key JGAP concepts

Class constructor – Each function class needs a constructor that extends from CommandGene. Just like the JGAP anttrail example, we've created an AntCommand class to provide the common behavior related to retrieving the AntMap. Somewhere in the construction of the object, the instance must identify the number of arguments and the return type. The number of arguments is the second argument in the CommandGene constructor and passed in as the a_arity of the argument. The third argument in the constructor is the return type. You should use the CommandGene static definitions of types; BooleanClass, IntegerClass, LongClass, FloatClass, and VoidClass. I must admit that I don't currently understand how the last two arguments work: `final int a_subReturnType, final int[] a_childSubTypes`.

Function Operations

The return type is used to determine which operation to call on the function class. A function class may support more than one return type. As mentioned above, the return type is identified in the constructor. The available functions are execute_boolean, execute_int, execute_long, execute_float, execute_double, execute_void and execute_object. For our Lisp-like sample solution, only execute_int is used.

Arguments

The allowable set of argument types are defined in the operation getChildType(IGPProgram a_ind, int a_chromNum). The a_chrom_Num is the key argument. It identifies the 0-based index of the arguments. If you want to support combinations of argument types, then you'll need to return the appropriate types. For example, if the first argument is a VoidClass and the second argument needs to be an IntegerClass, you would want a method like:

```
   if(a_chromNum == 0){
```

```
    return CommandGene.VoidClass;

  }else if(a_chromNum == 1){

    Return CommandGene.IntegerClass;

  }
```

String Representations for the Function Class

There are two mechanisms where the system creates a string representation of the class. The first is the getName() function that returns the name of the function. This version is utilized in the various Exceptions thrown by the CommandGene class.

The second mechanism provides feedback on the genetically derived function. The IGPProgram instance function of toStringNorm(int a_startNode) is used to return a string representing the program the JGAP system has created. It hierarchically calls the toString() operation to return the program representation.

Putting it all Together

We've analyzed our problem and created our functions that are hopefully sufficient to solve the problem, now we need to add the logic that I called the "Process Control" in the first diagram.

We'll extend from the GPProblem class to create our PaintedDesertProblem class. One of the key methods is the GPGenotype create() method. I must admit that I don't fully understand the various options for controlling the learning process. I'd like to identify the items that I'd like to learn more about. What is the advantage of the array of CommandGene array in the nodeSets variable help us? I only had a one dimensional array.

Running our Test

The next step is to create a batch command or other mechanism to invoke the PaintedDesertProblem's main subprogram. However, if your experience is like mine, then the program runs and it doesn't solve the problem as you had hoped.

Creating Unit Tests

Like all programming exercises, it's important to create unit tests. You're hoping to find the reason why JGAP didn't solve the problem as you expected. You could have made a mistake in the sufficiency test. Can the available functions be combined to solve the problem? Is your fitness function returning the proper results? What actually is happening in that magic JGAP black box? Is JGAP putting the program together in a way that you expected? Did you find a bug in JGAP?

Or you could have made a traditionally programming mistake? In the Painted Desert problem, the domain object approach allows for traditional junit testing on the Ant and AntMap classes. Certainly, I had my mistakes in the domain classes. The advantage of the existing solution approach is that in addition to the unit tests, I could write a program that demonstrated that the solution works.

I'd like to add a note about the kinds of mistakes you can make. I've identified that mistakes were made in the domain classes. I put together some elemental unit test examples to work through the issues. Certainly, the unit test coverage is not complete, but it provides an idea that may help you with your challenge. You might notice that one test case, OneElementGenElementsTest, compares the result of the hand coded solution against a solution using the JGAP function classes. One other confession to note, as simple as this problem is, I made a key mistake in programming the IfDrop class. It did not drop the sand. Therefore, the genetic program could never solve the problem. It was a simple oversight that cost too many hours of frustration and exploration, but helped me understand more about JGAP.

Writing Unit Tests in JGAP

JGAP is a Java program just like your traditional programming approaches. It is possible to create a unit test using the function classes that we created.

```
public void testGO_W()

throws Exception {

    GPProgram prog = new GPProgram(m_gpconf, 3);

    m_antMap.resetMap();

    prog.setApplicationData(m_antMap);





    ProgramChromosome pc1 = new ProgramChromosome(m_gpconf, 50, prog);

    pc1.getFunctions()[0] = new GO_W(m_gpconf);
```

```
        pc1.redepth();

        prog.setChromosome(0, pc1);


        Object[] noargs = new Object[0];

        int answer = prog.execute_int(0, noargs);

        assertEquals(1, m_antMap.getAnt().getXpos());

        assertEquals(1, m_antMap.getAnt().getYpos());


        answer = prog.execute_int(0, noargs);

        assertEquals(0, m_antMap.getAnt().getXpos());

        assertEquals(1, m_antMap.getAnt().getYpos());


        answer = prog.execute_int(0, noargs);

        assertEquals(2, m_antMap.getAnt().getXpos());

        assertEquals(1, m_antMap.getAnt().getYpos());


    }
```

The second argument in `new GPProgram(m_gpconf, 3)` identifies the number of programs that you are going to add to the GPProgram instance. One trick is that you must run the redepth() subprogram before you add the program to the GPProgram instance using setChromosome(…).

Another minor trick is how you handle nesting subfunctions with arguments. The testSolution subprogram of the OneAntGenElementsTest class is an example of how nesting works.

```
//          IFLTE (GO-W) (IF-DROP COLOR (IFLTE (GO-W) X (GO-S) (PICK-UP))) (IFLTE X

//          COLOR COLOR (PICK-UP)) (GO-S)

        ProgramChromosome pc1 = new ProgramChromosome(m_gpconf, 50, prog);
```

```
        pc1.getFunctions()[0] = new IfLessThanOrEqual(m_gpconf,
CommandGene.IntegerClass);

        pc1.getFunctions()[1] = new GO_W(m_gpconf);

        pc1.getFunctions()[2] = new IfDrop(m_gpconf, CommandGene.IntegerClass);

        pc1.getFunctions()[3] = new SandColor(m_gpconf);

        pc1.getFunctions()[4] = new IfLessThanOrEqual(m_gpconf,
CommandGene.IntegerClass);

        pc1.getFunctions()[5] = new GO_W(m_gpconf);

        pc1.getFunctions()[6] = new X(m_gpconf);

        pc1.getFunctions()[7] = new GO_S(m_gpconf);

        pc1.getFunctions()[8] = new Pickup(m_gpconf, "First Pickup");

        pc1.getFunctions()[9] = new IfLessThanOrEqual(m_gpconf,
CommandGene.IntegerClass);

        pc1.getFunctions()[10] = new X(m_gpconf);

        pc1.getFunctions()[11] = new SandColor(m_gpconf);

        pc1.getFunctions()[12] = new SandColor(m_gpconf);

        pc1.getFunctions()[13] = new Pickup(m_gpconf, "Second Pickup");

        pc1.getFunctions()[14] = new GO_S(m_gpconf);



        pc1.redepth();

        prog.setChromosome(0, pc1);
```

If the function class requires one or more arguments, then the classes are added in the following array elements and are used by the nested function class before continuing to add arguments to the parent function class.

Summary

Genetic programming with JGAP can work if you've set your problem up correctly. Debugging a program that the system automatically created offers a significant challenge. To combat the delegation of control to JGAP, consider writing unit tests that check your

domain logic and seriously consider checking your genetic functions with JGAP unit tests.

JGAP Philosophy and Visions

[JGAP Home]

Philosophy of JGAP

A complex subject such as one lying behind Evolutionary Algorithms allows to create a huge software package containing everything and the world. To keep things practical, we setup some leading records for the development of JGAP. These are:

- Deliver stable and easy to maintain code by extensively providing unit tests for each class. Currently, there are nearly 1000 unit tests!
- Optimize for performance and probably neglect logical correspondence to biological archetypes.
- Make JGAP easy-to-use by providing examples (how-tos), easy-to-understand class names and architecture.
- Allow for easy extensibility by carefully building up the architecture
- Make use of interfaces and abstract base classes wherever useful (historical growth of the classes makes this a difficult target when it comes to older entities).
- Document the more important things, like upgrading to newer JGAP versions.
- Make JGAP worth using by steering towards distributed computing and showing how to solve real-world examples. For the latter, Genetic Programming will be introduced, especially, which is very time-consuming and forces distributed computing.

Visions

Currently, JGAP undergoes heavy enhancement. Just have a look at the feature requests. Astonishingly few bugs are reported, although JGAP is used by many people. The current visions for JGAP are:

- Introduce Distributed Computing.
- Enhance monitoring capabilities.
- Evolve a great Robocode robot with RobocodeJGAP

- Add more graphical support (e.g. statistics, configurability).
- Make JGAP more popular by providing adapters to significant systems, steering towards symbiosis.

What we do to keep the quality of JGAP high

As it is impossible avoiding logical and conceptual errors, we rely on several effective means to keep JGAP's quality still high:

- JGAP's documentation is extensive. You can find how-tos and tutorials on the homepage, many comments in the source code and a self-documentation by the unit tests provided. Please compare JGAP's code documentation with that of other projects. Tell us if you find a better documented project!
- Detect code changes and program errors by unit tests. There are far over 1000 JUnit tests packaged with JGAP. They are part of our code basis and with every release new tests are added. In case a bug is reported, we add a test to reveal it.
- Before we publish a release, all unit tests are run. Additional reference implementations and the examples are also run to see that JGAP behaves as we expect it. Furthermore we use code coverage analysis (see homepage for a Clover report) to verify the test coverage of the productive classes delivered.
- Feedback from users (i.e. from you) is taken very seriously. We try to find a solution for you. In case of a bug by fixing it, in case of a feature request by working on it, and in case of a question by answering it. Try it out!
- Nothing is constant except change. Thus we try to react very fact to new requirements, try to release new versions as soon as the code basis is stable enough to do so. Additionally we keep JGAP's architecture up-to-date and don't take the paradigms and concepts of the current version as the final solution.
- JGAP's homepage is updated whenever necessary (please allow some delays sometimes, as we are talking about an Open Source project the whole time!).
- And lastly, we want you to contribute to JGAP. Either by asking questions, sending feedback, trying out the package. Or by becoming a developer who wants to add value to the project (see our homepage for contact)!

Copyright © 2005-2007 Klaus Meffert. All rights reserved.

Creating Custom Genes

[JGAP Home]

Preface: Please note that this documentation could be correlated to a former JGAP version. Therefore we recommend having a look at the examples provided with JGAP and then read this document in order to get an idea.

Genes represent the discrete components of a problem solution (chromosome) that can vary independently of each other throughout the evolution process. Just as genes control various independent features within our own genetic makeup, such as sex or eye color, so too do genes in JGAP control various aspects of the overall makeup of a solution. JGAP comes with a couple of standard Gene implementations that are useful for genes that are to be represented with boolean or integer values. But many applications can benefit from more specialized representations, and that's where custom gene implementations can come in handy. They're also easy to create, as we'll soon see.

How Genes are Used

First, let's talk about what genes are and how genes are created and used by JGAP. Simply put, a gene represents some characteristic of a solution that can be varied indepedently from the other characteristics of that solution. For example, let's assume that a solution (chromosome) represents a pile of American change. Then the genes in that solution might represent the various coin denominations that make up the pile, such as quarters, dimes, nickels, and pennies. The value of those genes (called alleles) would be the actual number of coins of each respective denomination present in that particular solution. In JGAP, genes are represented by classes that implement the Gene interface. Each gene in a chromosome may use a different Gene implementation to manage its values, but those implementations must be consistent across chromosomes. For example, if gene number 1 in chromosome A is using a particular implementation, say a QuarterGene, then gene number 1 in all other chromosomes in that population must also use a QuarterGene. But Gene number 2 in all of those chromosomes may be using some completely different implementation, such as a DimeGene. Of course, all of the genes in a Chromosome may also use the exact same Gene implementation if that's desirable.

You convey your desired Gene settings to JGAP by creating a sample Chromosome instance that is setup just the way you want all of the chromosomes in your population setup. You then pass that sample Chromosome to the active Configuration object. Each time JGAP needs to create a new Chromosome, it will reference the sample Chromosome to see how it's set up, and then will setup the new Chromosome exactly like it. In fact, JGAP actually takes advantage of a newGene() method in each of the Gene instances in the sample Chromosome to create the new Gene instances for the new Chromosome. So to create the Gene for gene number 1 in the new Chromosome, JGAP will invoke the newGene() method on gene number 1 of the sample chromosome. For gene number 2, it will invoke the newGene() method on gene number 2 of the sample Chromosome, and so on. This guarantees that the new Genes can be setup exactly like the Genes in the sample Chromosome.

QuarterGene: an Example

As an example of a custom Gene implementation, we'll create a QuarterGene class which could be used to represent the number of quarters in a pile of change. Before diving in, we should say that if we were implementing this in the real world, we would have extended the IntegerGene class to save ourselves most of the effort. But since this is supposed to be a demonstrative

example, we'll implement it here from scratch. First the code:

```
package hypothetical.examples;

import org.jgap.*;

import java.util.StringTokenizer;


/**
 * A Gene that can be used to represent the quarters in a pile of change.
 * An optional maximum number of quarters may be provided, in
 * which case the value (allele) of this Gene will never exceed that
 * maximum.
 */
public class QuarterGene extends BaseGene implements Gene, java.io.Serializable
{
    private static final String TOKEN_SEPARATOR = ":";

    private int m_maxNumberOfQuarters;
    private Integer m_numberOfQuarters;

    /**
     * Constructs a new QuarterGene with no maximum number of quarters that
     * can be represented (other than Integer.MAX_VALUE, of course).
     */
    public QuarterGene()
    {
        m_maxNumberOfQuarters = Integer.MAX_VALUE;
    }


    /**
     * Constructs a new QuarterGene with a constraint on the maximum number
     * of quarters that may be represented by this QuarterGene.
     *
     * @param a_maxNumberOfQuarters The maximum number of quarters that this
     *                              QuarterGene may represent. This value
     *                              must be non-negative.
     *
     * @throws IllegalArgumentException if the given maximum value is negative.
     */
    public QuarterGene( int a_maxNumberOfQuarters )
    {
        // Make sure the given maximum is non-negative.
        // ---------------------------------------------
        if( a_maxNumberOfQuarters < 0 )
        {
            throw new IllegalArgumentException(
                "The maximum number of quarters must be non-negative." );
        }

        m_maxNumberOfQuarters = a_maxNumberOfQuarters;
    }


    /**
     * Provides an implementation-independent means for creating new Gene
     * instances. The new instance that is created and returned should be
     * setup with any implementation-dependent configuration that this Gene
     * instance is setup with (aside from the actual value, of course). For
     * example, if this Gene were setup with bounds on its value, then the
     * Gene instance returned from this method should also be setup with
     * those same bounds. This is important, as the JGAP core will invoke this
     * method on each Gene in the sample Chromosome in order to create each
     * new Gene in the same respective gene position for a new Chromosome.
     *
     * It should be noted that nothing is guaranteed about the actual value
```

```java
     * of the returned Gene and it should therefore be considered to be
     * undefined.
     *
     * @return A new Gene instance of the same type and with the same
     *         setup as this concrete Gene.
     */
    public Gene newGene( )
    {
        // We construct the new QuarterGene with the same maximum number
        // of quarters that this Gene was constructed with.
        // -----------------------------------------------------------------
        return new QuarterGene( m_maxNumberOfQuarters );
    }


    /**
     * Sets the value of this Gene to the new given value. The actual
     * type of the value is implementation-dependent.
     *
     * @param a_newValue the new value of this Gene instance.
     */
    public void setAllele( Object a_newValue )
    {
        m_numberOfQuarters = (Integer) a_newValue;
    }


    /**
     * Retrieves the value represented by this Gene. The actual type
     * of the value is implementation-dependent.
     *
     * @return the value of this Gene.
     */
    public Object getAllele()
    {
        return m_numberOfQuarters;
    }


    /**
     * Sets the value of this Gene to a random legal value for the
     * implementation. This method exists for the benefit of mutation and other
     * operations that simply desire to randomize the value of a gene.
     *
     * @param a_numberGenerator The random number generator that should be
     *                          used to create any random values. It's important
     *                          to use this generator to maintain the user's
     *                          flexibility to configure the genetic engine
     *                          to use the random number generator of their
     *                          choice.
     */
    public void setToRandomValue( RandomGenerator a_numberGenerator )
    {
        // Pick a random number between 0 and the maximum number
        // of quarters we're allowed to represent.
        // ---------------------------------------------------------
        m_numberOfQuarters = new Integer(
            a_numberGenerator.nextInt( m_maxNumberOfQuarters) );
    }


    /**
     * Retrieves a string representation of the value of this Gene instance
     * that includes any information required to reconstruct it at a later
     * time, such as its value and internal state. This string will be used to
     * represent this Gene instance in XML persistence. This is an optional
     * method but, if not implemented, XML persistence and possibly other
     * features will not be available. An UnsupportedOperationException should
     * be thrown if no implementation is provided.
```

```java
     *
     * @return A string representation of this Gene's current state.
     * @throws UnsupportedOperationException to indicate that no implementation
     *         is provided for this method.
     */
    public String getPersistentRepresentation()
                  throws UnsupportedOperationException
    {
        // We want to represent both the maximum number of quarters that
        // can be represented by this Gene and its actual current value.
        // We'll separate the two with colons.
        // ----------------------------------------------------------------
        return new Integer( m_maxNumberOfQuarters ).toString() +
                   TOKEN_SEPARATOR + m_numberOfQuarters.toString();
    }


    /**
     * Sets the value and internal state of this Gene from the string
     * representation returned by a previous invocation of the
     * getPersistentRepresentation() method. This is an optional method but,
     * if not implemented, XML persistence and possibly other features will not
     * be available. An UnsupportedOperationException should be thrown if no
     * implementation is provided.
     *
     * @param a_representation the string representation retrieved from a
     *                         prior call to the getPersistentRepresentation()
     *                         method.
     *
     * @throws UnsupportedOperationException to indicate that no implementation
     *         is provided for this method.
     * @throws UnsupportedRepresentationException if this Gene implementation
     *         does not support the given string representation.
     */
    public void setValueFromPersistentRepresentation( String a_representation )
                throws UnsupportedOperationException,
                       UnsupportedRepresentationException
    {
        // We're expecting to find the maximum number of quarters that this
        // Gene can represent, followed by a colon, followed by the actual
        // number of quarters currently represented.
        // ------------------------------------------------------------------
        StringTokenizer tokenizer = new StringTokenizer( a_representation,
                                                         TOKEN_SEPARATOR );
        // Make sure there are exactly two tokens.
        // ---------------------------------------
        if( tokenizer.countTokens() != 2 )
        {
            throw new UnsupportedRepresentationException(
                "Unknown representation format: Two tokens expected." );
        }

        try
        {
            // Parse the two tokens as integers.
            // ---------------------------------
            m_maxNumberOfQuarters = Integer.parseInt( tokenizer.nextToken() );
            m_numberOfQuarters = new Integer( tokenizer.nextToken() );
        }
        catch( NumberFormatException e )
        {
            throw new UnsupportedRepresentationException(
                "Unknown representation format: Expecting integer values." );
        }
    }


    /**
     * Executed by the genetic engine when this Gene instance is no
```

```java
     * longer needed and should perform any necessary resource cleanup.
     */
    public void cleanup()
    {
        // There's no cleanup necessary for this implementation.
        // -----------------------------------------------------
    }


    /**
     * Compares this Gene with the specified object for order.  Returns a
     * negative integer, zero, or a positive integer as this object is less
     * than, equal to, or greater than the specified object. The given object
     * must be a QuarterGene.
     *
     * @param   a_otherQuarterGene the Object to be compared.
     * @return  a negative integer, zero, or a positive integer as this object
     *              is less than, equal to, or greater than the specified object.
     *
     * @throws ClassCastException if the specified object's type prevents it
     *         from being compared to this Object.
     */
    public int compareTo( Object a_otherQuarterGene )
    {
        // If the other allele is null, we're bigger.
        // ------------------------------------------
        if( a_otherQuarterGene == null )
        {
            return 1;
        }

        // If our allele is null, then we're either the same as the given
        // QuarterGene if its allele is also null, or less than it if
        // its allele is not null.
        // -------------------------------------------------------------
        if( m_numberOfQuarters == null )
        {
            if ( ((QuarterGene) a_otherQuarterGene).m_numberOfQuarters == null )
            {
                return 0;
            }
            else
            {
                return -1;
            }
        }

        // Otherwise, we just take advantage of the Integer.compareTo()
        // method.
        // ------------------------------------------------------------
        return m_numberOfQuarters.compareTo(
            ( (QuarterGene) a_otherQuarterGene ).m_numberOfQuarters );
    }


    /**
     * Determines if this QuarterGene is equal to the given QuarterGene.
     *
     * @return true if this QuarterGene is equal to the given QuarterGene,
     *         false otherwise.
     */
    public boolean equals( Object otherQuarterGene )
    {
        return otherQuarterGene instanceof QuarterGene &&
               compareTo( otherQuarterGene ) == 0;
    }


    /**
```

```
     * Calculates the hash-code for this QuarterGene.
     *
     * @return the hash-code of this QuarterGene
     */
    public int hashCode()
    {
        return m_numberOfQuarters;
    }

    public Object getInternalValue() {
        return m_numberOfQuarters;
    }
}
```

Hopefully most of the methods and code in the example above are pretty self-explanatory once the javadocs and comments are consulted. We start off with a couple of constructors, one of which lets the user specify an upper bound on the number of quarters that this specific QuarterGene instance may represent. We then move onto the newGene() method, which was mentioned earlier, but deserves a second look.

The most important thing to note about the newGene() method is that it should return a new Gene that is setup with the same internal configuration as this Gene. The actual value of the returned Gene doesn't matter, but you'll notice that we make sure to pass on the same upper bound as our current QuarterGene instance. This is worth noting because JGAP uses this method to create new Genes for new Chromosomes, and it's important that the Genes in the new Chromosome are setup exactly the same way as the Genes in earlier Chromosomes. By invoking the newGene() method on each of the Genes in a Chromosome, JGAP can be sure that the Genes in the new Chromosome will be setup the same way each time.

Next come the setAllele() and getAllele() calls, which aren't very interesting. Values (alleles) in Genes are represented as Objects, in order to be as flexible as possible, and so we use an Integer object in our QuarterGene implementation to represent the number of quarters. Following these is the setRandomValue() method, which sets the value of our QuarterGene to a legal random value--in this case, an Integer between zero and the upper bound.

The next two methods deal with XML marshalling and are optional. (For more information on XML marshalling, please see the Marshalling Populations to XML document.) We've implemented them here to provide an example. The getPersistentRepresentation() method returns a String that represents not only the current value of our QuarterGene, but also the upper bound, so that this QuarterGene can be later reconstructed with exactly the same state that it has currently. To do this, we just convert our two integer values to strings and separate them with a colon. Simple enough! The setValueFromPersistentRepresentation() method does the opposite: it takes a String representation, parses it, and then sets the internal state of this QuarterGene accordingly. If there's anything wrong with the representation, an UnsupportedRepresentationException is thrown.

The final three methods are cleanup(), which is invoked by JGAP when this Gene is no longer needed; compareTo(), which compares this QuarterGene to another one and determines which is the "greater" of the two; equals(), which compares this QuarterGene to another one and determines if they are equal; and, finally, hashCode(), which generates a hash-code. All of these should be self-explanatory. Note that it's very important to include working equals() and hashCode() methods, and not to simply rely on the default implementations in java.lang.Object.

A Note about Serialization

As mentioned in the previous section, JGAP supports marshalling and unmarshalling genetic components to/from XML. It also supports the standard Java serialization mechanism, allowing

Genotypes and Chromosomes to be serialied. In order for this to work properly, two things are required of your Gene implementation: first, it must implement the java.io.Serializable interface. Second, it must declare any fields transient that are not an actual part of the Gene state. For example, if your Gene maintains a reference to the active Configuration object, then this reference must be declared transient. Ideally, your Gene implementations should simply avoid instance fields that aren't a part of the Gene state.

Conclusion

As can be seen from the above example, creating custom genes is an easy and straight-forward process. We hope that you'll create your own Gene implementations whenever you feel the stock genes aren't exactly what you need. And hey--if you create a custom gene that you think might be useful to others, please feel free to post it on the jgap-users mailing list so that others can benefit from your work!

Copyright © 2002-2007 Neil Rotstan / Klaus Meffert. All rights reserved.

[JGAP Home]

Creating Custom Genetic Operators

[JGAP Home]

Preface: This document corresponds to an earlier JGAP version. The introduced ReproductionOperator is no longer in use. Now, reproduction takes place via natural selectors.

It may be you want to experiment around with new or different genetic operators to assess their impact upon the performance of the genetic algorithm. Or maybe you wish to replace the random number generator with a more rigorous implementation. Customizing JGAP is intended to be easy and straight-forward. JGAP supports a Configuration mechanism that easily allows you to add your own genetic operators to the mix, as well as replace certain components such as the natural selector, the random number generator, and the event manager.

Creating a new genetic operator is probably the second-most common form of JGAP customization, after the creation of custom Genes. We'll look at an existing operator, the ReproductionOperator, as an example of how to create your own genetic operator. As you'll see, it's extremely simple.

Step 1: Implement GeneticOperator

To create your own genetic operator, you'll need to implement the org.jgap.GeneticOperator interface. This interface only has one method, operate(), which is the method that will be

invoked by the genetic engine when it's time for your genetic operator to do its work. Let's look at the source for the ReproductionOperator:

```
package org.jgap.impl;

import org.jgap.Chromosome;
import org.jgap.Configuration;
import org.jgap.GeneticOperator;

import java.util.List;

/**
* The reproduction operator makes a copy of each Chromosome in the
* population and adds it to the list of candidate chromosomes. This
* essentially guarantees that each Chromosome in the current
population
* remains a candidate for selection for the next population.
*/
public class ReproductionOperator implements GeneticOperator
{
    /**
* The operate method will be invoked on each of the genetic operators
* referenced by the current Configuration object during the evolution
* phase. Operators are given an opportunity to run in the order that
* they are added to the Configuration. Implementations of this method
* may reference the population of Chromosomes as it was at the
beginning
* of the evolutionary phase or the candidate Chromosomes, which are
the
* results of prior genetic operators. In either case, only Chromosomes

* added to the list of candidate chromosomes will be considered for
* natural selection. Implementations should never modify the original
* population.
*
* @param a_population the population of chromosomes from the current
* evolution prior to exposure to any genetic operators
* @param a_candidateChromosomes the pool of chromosomes that are
candidates
* for the next evolved population. Any chromosomes that are modified
by this
* genetic operator that should be considered for natural selection
should be
* added to the candidate chromosomes
*/
    public void operate( final Population a_population,
                         final List a_candidateChromosomes )
    {
        // Just loop over the chromosomes in the population, make a
copy of
        // each one, and then add that copy to the candidate
chromosomes
        // pool so that it'll be considered for natural selection
during the
        // next phase of evolution.
```

```
        //
-----------------------------------------------------------------
        int len = a_population.size();
        for ( int i = 0; i < len; i++ )
        {
            a_candidateChromosomes.add(
a_population.getChromosome(i).clone() );
        }
    }
}
```

The purpose of this operator is simply to create a copy of each of the Chromosomes in the population. This is done to ensure that the original Chromosomes will have a chance to be selected, in addition to their offspring. As can be seen from the code, the operate() method accepts three parameters: the active Configuration object, an array of Chromosomes that make up the current population, and a List of candidate chromosomes that will ultimately go to the natural selection phase. The population array should never be modified--it is intended to represent the population as it was before any genetic operators have been invoked. If you wish to alter a Chromosome, you should first make a copy of it via the clone() method and then alter the copy. Any altered Chromosomes that you wish to go on to the natural selection phase must be added to the list of candidate chromosomes.

Step 2: Add the Operator to the Configuration

Now that we've written our operator, we need to tell JGAP to use it. This just requires a single additional call to the Configuration object during your normal setup phase:

```
Configuration conf = new DefaultConfiguration();
conf.addGeneticOperator( new ReproductionOperator() );
```

The genetic engine will run the population of Chromosomes through each of the active genetic operators in the order in which they were added to the Configuration object. It's worth noting that there is no way to remove an operator from the Configuration object, which means that if you want to replace an existing operator with your own implementation, or place your genetic operator "in front" of an existing operator, you won't be able to rely on the DefaultConfiguration convenience class, since it automatically adds all of the stock genetic operators. Instead, you'll need to setup your Configuration from scratch, which isn't all that daunting a task. Just have a look at the DefaultConfiguration class to see what it does. If you're just adding new genetic operators (and not replacing existing ones) and are content with them being at the end of the operator chain, then you should be able to continue using the DefaultConfiguration class.

Important Note! When using your own set of genetic operators, please be aware that if you do not include a reproduction operator, then the chromosomes in your population will not automatically become candidates for natural selection. It is the reproduction operator that guarantees that each of the chromosomes in the current population become candidates for the next generation.

Other Customizations

As mentioned above, it's also possible to replace the JGAP natural selector, random number generator, and event manager. Details of these customizations are beyond the scope of this document, but performing them follows a similar process to creating a new genetic operator: first, the new implementation must be written and implement the appropriate interface; and second, the Configuration object must be setup to use your new implementation. Unlike genetic operators, the Configuration object allows these other components to be replaced, so you can

continue using the DefaultConfiguration convenience class if desired.

The appropriate interfaces to implement for the various customizations are as follows:

- To create your own natural selector, implement the `org.jgap.NaturalSelector` interface.
- To create your own random number generator, implement the `org.jgap.RandomGenerator` interface.
- To create your own event manager, subclass the `org.jgap.event.EventManager` class.
- To create your own Gene implementations, subclass `org.jgap.BaseGene`.

Please see the Javadocs for each of the above for further information on what is required for each implementation.

Copyright © 2002- 2007 Neil Rotstan / Klaus Meffert. All rights reserved.

Marshalling Populations to XML

For many applications, some sort of marshalling mechanism can be very useful. Perhaps it's desirable to save the current population to disk for later so that evolution can continue at another time, or perhaps it's necessary to send populations over a network as part of a distributed application. JGAP comes with facilities for representing genotypes, chromosomes, and even individual genes as XML documents and/or elements. This process is called marshalling. And, of course, JGAP can later unmarshall those objects from their respective XML representations. This document discusses the org.jgap.xml.XMLManager class, which is responsible for these facilities, and provides an example of its use.

Parser-specific Implementations

As a bit of an aside, it should be noted that, while the creation and traversing of an XML document object (DOM) has been standardized by the W3C, the actual generation of an XML file or other representation from a DOM object will not be standardized until the DOM Level 3 specification is finalized. Until then, each XML parser provides its own implementation-specific mechanism. As a result, JGAP will generate a DOM object for you from a genetic entity, but the actual process of rendering that DOM object to a file or other medium is up to you. An example of this process is provided in this document using the Xerces Java 2 parser, which is a free, open-source XML parser provided by the Apache foundation.

An Example

Let us say, for our example, that we are creating a chess program that, as one of its many selling points, will evolve its strategy to match any opponent. To implement this in our hypothetical program, we have of course decided to use JGAP to create a genetic algorithm that manages the chess AI. It's not enough, however, for us to simply evolve the AI strategy to some acceptable point at the factory and then ship out the program with a frozen strategy. Since we've promised that our AI will adapt to its opponents, it's necessary for the AI strategy to continue evolving as the game is played. This inevitably means that we need some mechanism for storing our current genetic state in between program runs. Fortunately, JGAP provides the XMLManager class to help us out with this. Even more fortunate is that it's very simple to use. Here's the code:

```
/**
 * Saves the given population to the given filename as an XML
 * file. The population can later be reconstructed from the
 * XML file through a call to the getSavedStrategyState() method.
 *
 * @param a_currentPopulation The population to save.
 * @param a_saveToFilename The full path to the file where the
 *                         XML representation should be saved.
 */
public static saveStrategyState( Genotype a_currentPopulation,
                                 string a_saveToFilename )
              throws Exception
{
    // Convert the Genotype to a DOM object.
    // -------------------------------------
    Document xmlRepresentation =
        XMLManager.representGenotypeAsDocument( a_currentPopulation );

    // Now write that DOM to disk as an XML file. This is all
    // parser-specific. We're assuming the use of the Xerces Java 2
    // parser for this example.
    // -----------------------------------------------------------
    Writer documentWriter = new FileWriter( a_saveToFilename );

    OutputFormat formatting =
        new OutputFormat( genotypeDoc, "UTF-8", true );

    SerializerFactory factory =
        SerializerFactory.getSerializerFactory( Method.XML );

    Serializer genericSerializer =
        factory.makeSerializer( documentWriter, formatting );

    DOMSerializer documentSerializer =
        genericSerializer.asDOMSerializer();

    documentSerializer.serialize( genotypeDoc );
    documentWriter.close();
}


/**
 * Recreates a population that was earlier saved to an XML file through
 * the saveStrategyState() method.
 *
 * @param a_activeConfiguration The current active Configuration object.
 * @param a_savedToFilename The full path of the file where the XML
 *                          representation was saved.
 *
 * @return A new Genotype instance that is populated from the XML
 *         representation found in the given file.
 */
public static Genotype getSavedStrategyState(
```

```
                               Configuration a_activeConfiguration,
                               String a_savedToFilename )
                           throws Exception
{
    // Read the saved xml file into a DOM object.
    // ------------------------------------------
    Reader documentReader = new FileReader( a_savedToFilename );
    InputSource documentSource = new InputSource( documentReader );
    DocumentBuilder builder =
        DocumentBuilderFactory.newInstance().newDocumentBuilder();
    Document genotypeDocument = builder.parse( documentSource );


    // Convert the DOM object back to a Genotype
    // -----------------------------------------
    return XMLManager.getGenotypeFromDocument( a_activeConfiguration,
                                               genotypeDocument );
}
```

As can be seen from the above, interaction with the JGAP XMLManager to create a DOM object from a Genotype--or recreate a Genotype from a DOM object--can typically be confined to a line or two of code. Most of the work involves getting that DOM to disk (and back again), which may be easier or more difficult depending on which parser you use. When the DOM Level 3 standard is finalized, it is likely that the facilities of the XMLManager class will be expanded to encompass this entire process.

Copyright © 2002- 2006 Neil Rotstan / Klaus Meffert. All rights reserved.

[JGAP Home]

Supergenes

[JGAP Home]

The problem

In some tasks, you may have several closely dependent genes that cannot mutate in arbitrary way. Instead, the valid set of values for each gene-component depends from the current value of the related genes - part of the same system. Such group of the related genes can make only part of the whole genome. Using default JGAP mechanism means that a mutation, breaking the internal rules of the interdependent gene group, will lead to discarding the whole such chromosome. This finally slows the evolution, reducing the system performance.

The solution: Supergenes

In biology, *Supergene* represents several genes, which control closely related aspects of the phenotype. The living cell is a complex machinery, and the problem of gene dependency is frequent. For example, part of the plant photosynthetic system is a so-called ATP synthase.

This is a complex formation, consisting of several proteins - components. If some components mutate, becoming useless, the ATP synthase cannot longer be created. The most surprising is that *the other* proteins that have not been touched by mutation, are also suppressed. Probably having a damaged ATP synthase is worse than having no ATP synthase at all. So, the gene group, responsible for ATP synthase, mutates as a single unit: ATP synthase absent/ATP synthase present.

It is possible to suggest an example from mathematics. Imagine that we use JGAP to find the values *a*, *b, c* and *d,* fitting the experimental data into real function

$$\sqrt{a-\frac{b}{x}} + \sin(d \cdot x + c)$$

If we only restrict x > 0, we must also restrict *a* >= *b*, but *c* and *d* are not dependent from this condition. So *a* and *b* can be placed in a supergene, setting validity condition *a* >= *b*.

The old MinimizeMakeChange example can also be extended to make supergenes useful: lets suppose that the number of pennies and nickels must be *both* even or *both* odd.

JGAP Supergene mutates only in such way, that the allele combination remains valid. Mutations, that make allele combination invalid, are rejected inside the method, responsible for creating the mutated version. For the efficient use, you must combine several genes and supergenes inside the same chromosome. Supergene components can also be a Supergenes, creating the tree-like structures in this way.

Using the supergenes with JGAP

For better understanding of this chapter, we recommend to read about the MinimizeMakeChange example first.

In JGAP, the supergene support is implemented in package org.jgap.supergenes. The simpliest way to create a supergene is to derive it from the *abstractSupergene* class, overriding *isValid()* method. For example, if we need a supergene, holding the two *IntegerGenes* (nickels and pennies) and ensuring condition *nickels mod 2 = pennies mod 2:*

```
import org.jgap.impl.IntegerGene;
import org.jgap.supergenes.abstractSupergene;
import org.jgap.Gene;

/** Supergene to hold quarters and nickels. Valid if the number of
 * nickels and pennies is either both odd or both even. */

public class NickelsPenniesSupergene extends abstractSupergene {
 /* It is important to provide these two constructors: */
 public NickelsPenniesSupergene() {}
 public NickelsPenniesSupergene( Gene[] a_genes )
 {
 super(a_genes);
 }

 /* Now just check nickels mod 2 = pennies mod 2: */
 public boolean isValid(Gene [] genes, Supergene s)
 {
 IntegerGene nickels = (IntegerGene) genes[0];
 IntegerGene pennies = (IntegerGene) genes[1];
 boolean valid = nickels.intValue() % 2 == pennies.intValue() % 2;
 return valid;
 }
```

```
}
```

Now, the sample gene array, consisting of one supergene and two ordinary genes for quarters and dimes, is created in the following way:

```
Gene[] sampleGenes = new Gene[3];
sampleGenes[DIMES] = getDimesGene ();
sampleGenes[QUARTERS] = getQuartersGene ();

sampleGenes[2] = new NickelsPenniesSupergene(
  new Gene[] { getNickelsGene (), getPenniesGene (),
}
```

Compare this with the non-supergene version:

```
Gene[] sampleGenes = new Gene[4];
sampleGenes[DIMES] = getDimesGene();
sampleGenes[NICKELS] = getNickelsGene();
sampleGenes[QUARTERS] = getQuartersGene();
sampleGenes[PENNIES] = getPenniesGene();
```

It is trivial to create a chromosome from the gene array and to get a solution of the task (see example class org.jgap.supergenes.SupergeneSampleApplicationTest included with the tests in the JGAP distribution).

Are supergenes useful?

In other words, can they increase the application performance in a real JGAP applications? Maybe this conception is interesting just from the point of theoretical biology? To answer this question, we did a performance tests. From these tests we concluded the following:

> The supergene conception can significantly (up to 3 times in the most boundary cases) increase the evolution speed. It can also increase the accuracy of solution for the small populations. However these effects are dependent from the chosen population size and maximal number of iterations. While in some cases the use of supergenes can be definitely sensible, we would recommend to try both supergene and non-supergene versions for your specific task.

[Documentation Index]

This page is copyright © 2004-2006 Klaus Meffert and Audrius Meskauskas, GNU free documentation license

Testing the *Supergene* conception

[Documentation Index]

The test

The test task: given amount [20..99] in pennies. Express this value in pennies, nickels (5 pennies), dimes (10 pennies) and quarters (25 pennies), additional requiring, that the number of pennies and nickels should be both even or both odd. For some amounts the ideal solution does not exists, and the system should suggest an alternative with a slightly different change amount.

In the supergene version, the chromosome, representing the solution of the task, consisted of 2 regular genes (quarters and dimes) and one supergene, enclosing two subgenes (nickels and pennies). The supergene validity condition was set to nickels *mod* 2 = pennies *mod* 2. The alternative algorithm used chromosome with 4 independent genes and estimation function, including the mentioned condition.

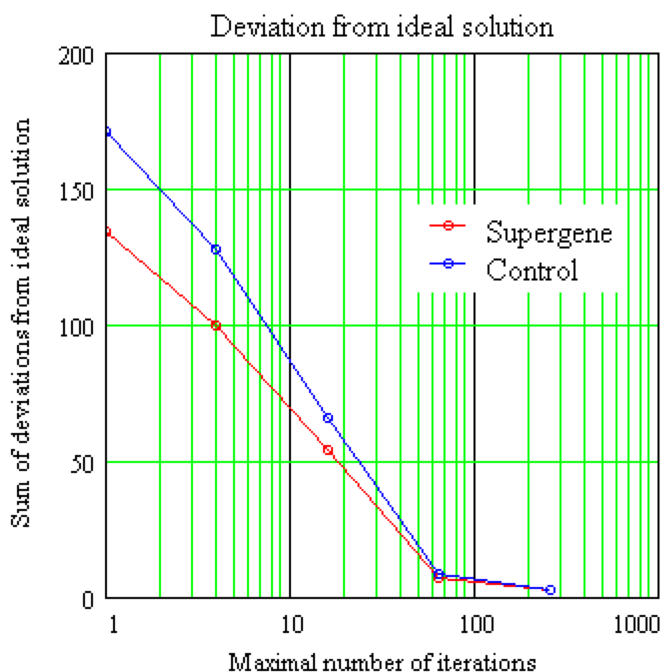
Conditions

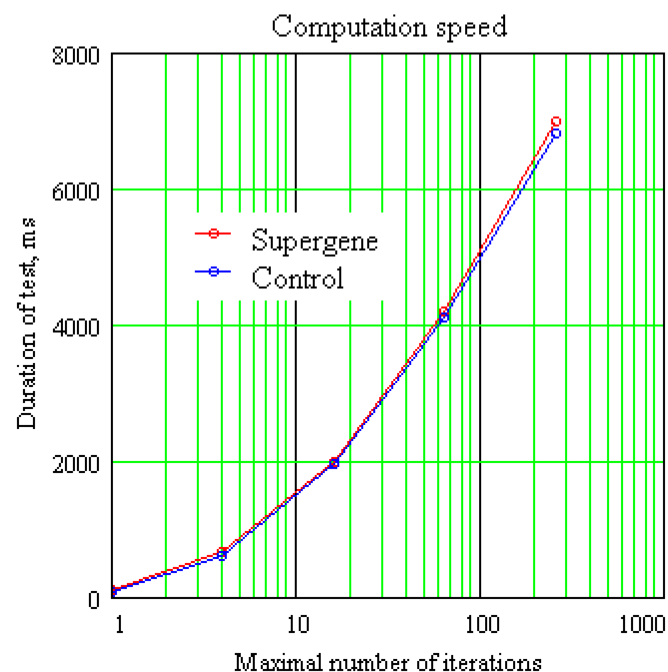
The test program computed the change for all values from 20 to 99, measuring the computation speed and deviation from the ideal solution (exact change amount and the satisfied additional condition). The test was repeated 10 times, averaging the data. The test program was executed on computer, having Intel Pentium 4 2.5 Mhz processor, 512 Mb RAM, Windows XP SP 2.0, Java 1.4.2 b-28.

Results

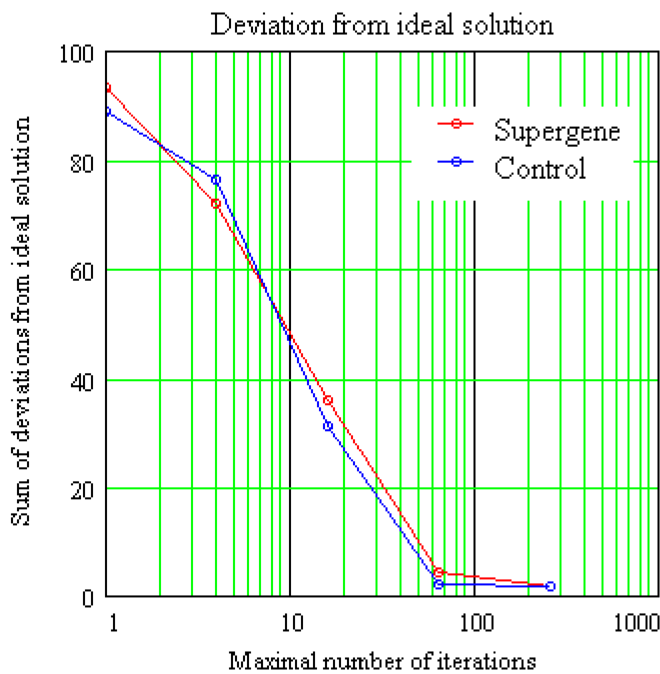
The results were dependent from the population size and maximal allowed number of iterations.

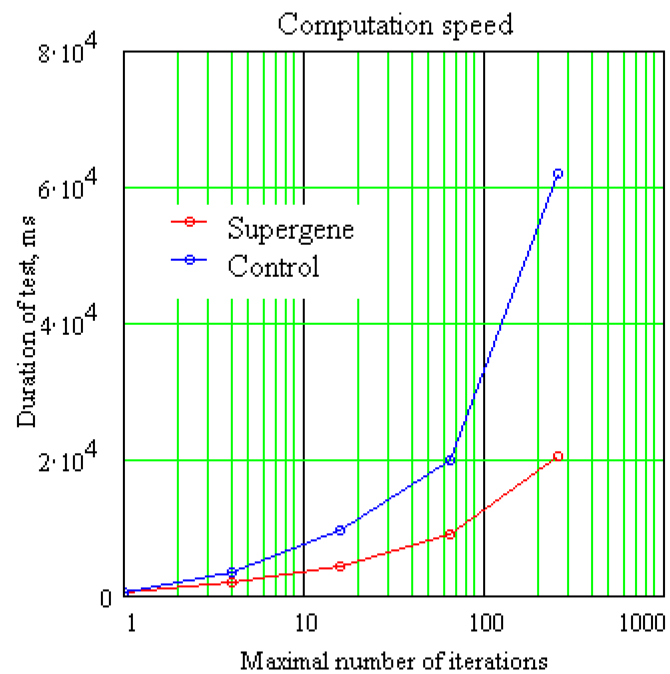
For the small population size (64) and small number of iterations the supergene method is a little more accurate, running roughly at the same speed as the simplified version. However this advantage decreases when incrasing the number of allowed iterations.



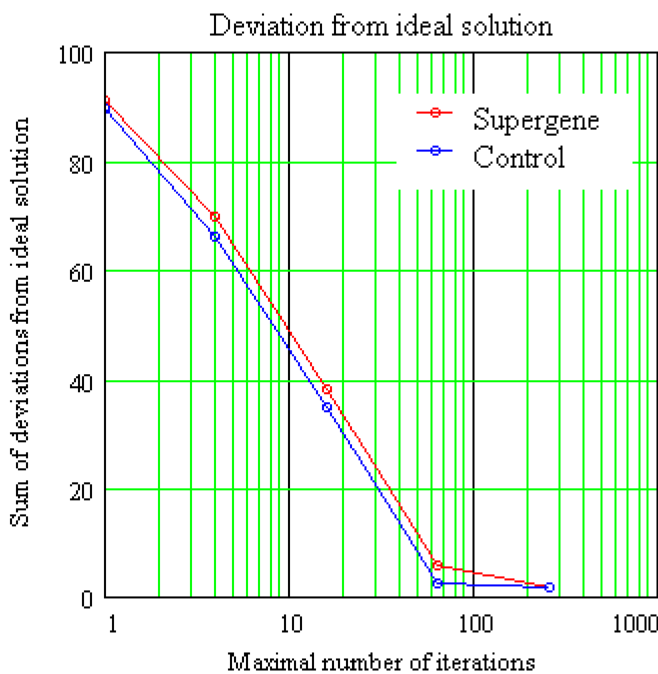


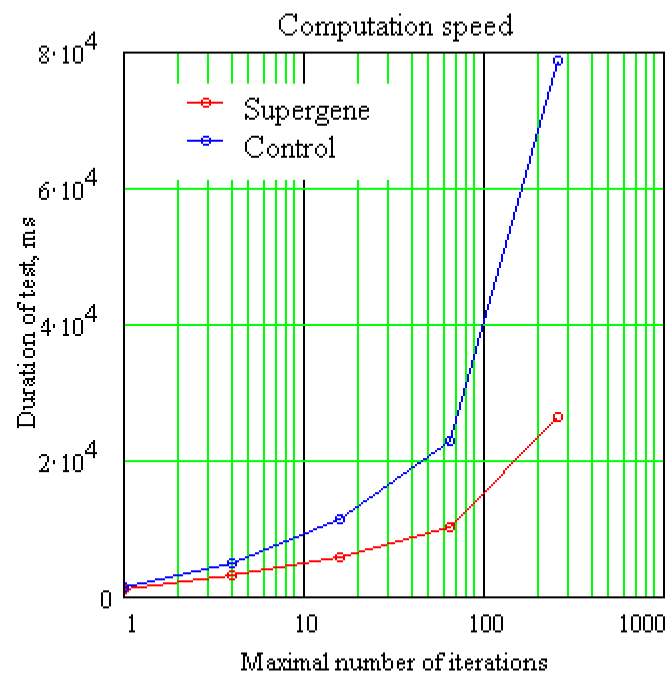
For the moderate population size (512) the supergene method is significantly (up to three times) faster than the simplified version.





For a large populations (1024) the supergene method may be slightly less accurate. However this difference seems minor, and the method is still up to three times faster than the simplified version.





Conclusion

The supergene conception can significantly (up to 3 times) increase the evolution speed. It can also increase the accuracy of solution for the small populations. However these effects are dependent from the chosen population size and maximal number of iterations. While in some cases the use of supergenes can be definitely sensible, we would recommend to try both supergene and non-supergene versions for your specific task.

This testing work has been performed by Audrius Meskauskas

This page is copyright © 2004, 2005 Audrius Meskauskas, GNU free documentation license

SOURCEFORGE.NET®

The traveling salesman problem

[Documentation Index]

The traveling salesman problem is the following: given a finite number of 'cities' along with the cost of travel between each pair of them, find the cheapest way of visiting all the cities and returning to your starting point. This task has deserved a serious attention from mathematicians, and a lot of literature is available.

The number of the possible routes increases rapidly when increasing the number of cities to visit, and the method of force become inappropriate. Genetic methods cannot find surely the best solution, but they can find the comparatively good solution in an acceptable time. JGAP implements the solution of this task using swap mutations and greedy crossover algorithm, as it is described by (Grefenstette et al, 1985).

To solve this task with JGAP, you must define the specific of your task by deriving class from org.jgap.impl.Salesman.

The most critical information you need to specify is how to measure the travel cost between the two cities. JGAP uses subclasses of its *Gene* to represent a city. For example, if you use the *IntegerGene* to store the number-city reference then you can query some local structure of even remote database to get the travel costs or distance between the cities. To specify the needed algorithm, you must override the method Salesman.**distance**:

```
abstract double distance(Genea_from, Genea_to)
```

You also need to specify the cities we need to visit, and from which city the journey should begin. This is done by representing a "sample solution". This solution must be formally correct, but need not be even close to the optimal one. The sample solution must be presented in as a *Chromosome* that can be easily constructed from the gene array.

```
abstract Chromosome createSampleChromosome(Objectan_initial_data)
```

For more precise control on the solution you may also need to override several other methods and use the parameter *an_initial_data.* Read the org.jgap.impl.salesman.Salesman documentation for details.

After you create a working descendant of the *Salesman* by overriding the two mentioned methods, the task solution can be obtained just by calling

```
Chromosome findOptimalPath(Objectan_initial_data)
```

The parameter *an_initial_data* is later passed to **createSampleChromosome.** This is how you can solve multiple tasks with the same class, creating the different sample solution from the data, stored in your object.

The result is also returned in a form of *Chromosome*, where you can easily access the individual genes.

You can actually do more, but other details are obvious from the code documentation page. If you have some comments, questions or suggestions for this implementation, please submit a bug or rfe (feature request).

See also the complete java source code for a simple example, where the distance between the cities is equal to the absolute difference between they given numbers.

This page is copyright © 2004, 2005 Audrius Meskauskas, GNU free documentation license
Style updated 2006 by Klaus Meffert

Literature

- J. Grefenstette, R. Gopal, R. Rosmaita, and D. Gucht. *Genetic algorithms for the traveling salesman problem*. In Proceedings of the Second International Conference on Genetic Algorithms. Lawrence Eribaum Associates, Mahwah, NJ, 1985.
- J. L Sushil, L. Gong (explanatory material on the web)

Brief Summary of Coding Style and Practices used in JGAP

The following is a short list of guidelines to follow when contributing code or patches to the JGAP project. Before we commit any code to the CVS repository, we need to make it follow these guidelines (for protection of our own sanity), so it's a lot easier on us if your code simply adheres to the guidelines when you submit it :-) Thanks!

A sort of ironic compilation of *dont do's* can be found in Roedy Green's article, it's worth a read.

Formatting of Code

1. All new code files must have a copyright statement at the top that reads:

```
/*
 * This file is part of JGAP.
 *
 * JGAP offers a dual license model containing the LGPL as well as the MPL.
 *
 * For licencing 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.
 */
```

2. Add the lines

```
/** String containing the CVS revision. Read out via reflection!*/
private static final String CVS_REVISION = "$Revision: 1.9 $";
```

after each class declaration line at top of the class file. For interfaces omit the "private" statement.

3. When importing classes with the `import` statement, use the generic form and don't import single classes:

```
import java.util.*;    // correct
          import java.util.List; // WRONG
```

4. Do not format javadocs by adding unnecessary spaces or indentation (javadoc output will do this!)

5. Do not start a javadoc for parameters with a capital letter. Do not end such text with a period.

6. In methods: Order of javadoc elements should be:

   - description of method
   - add a blank line
   - parameters (if existent)
   - return value (if not void)
   - throws (if existent)
   - add a blank line
   - author
   - since (here you should enter the next JGAP version to be released, i.e. "2.4" (without quotation marks)).

7. All classes, methods, and fields require javadocs, including private methods and fields and the main method.

8. All in-line comments should be "underlined." For example:

```
// This is an in-line comment.
// Notice that it is underlined.
// -----------------------------
doSomeWork();
```

9. Indent 4 spaces. No tabs, please.

10. Lines of code should not exceed 79 characters in length. Please take this very seriously!

11. Curly braces go on the same line, no extra indentation. For example:

```
public void foo() {
  doSomeWork();
}
```

12. Always but the statements after an if or else-statement in curly braces, like:

```
if (condition) { // this curly brace is required
  statement1;
  statement2;
}
```

13. Class names always start with a capital letter and contain no underscores. THIS IS VERY IMPORTANT!

14. Class and instance variables, other than constants, should start with `m_`. For example:

```
private int m_myField;
```

15. Method arguments should always begin with "a_". For example,

```
public void foo(int a_someValue);
```

16. Constants should be in all capitals. For example,

```
public static final int MY_CONSTANT_VALUE = 1;
```

17. In interfaces, don't use the key word *public* for declared methods.

Programming

1. Favor clean, readable, maintainable code over optimized code. Let a profiler determine where the bottlenecks exist and then optimize as little as possible to achieve desired performance. When you do optimize, please leave a very detailed comment explaining that the code has been optimized and what it is doing.

2. Currently, we are *not* using Java 1.4 assertions in the JGAP code. We may begin doing so in the future, but for now please leave them out.

3. If you end up creating a new package, please also create a `package.html` file that explains the purpose of that package. Javadoc will use this file to generate a package description.

4. In general, favor descriptive method and variable names over in-line comments. For example, instead of:

```
// Calculate area of circle
// ------------------------
x = 2 * PI * r * r;
```

   Do:

```
areaOfCircle = getAreaOfCircle( circleRadius );
```

5. When adding a new class or new method, you could add the RFE number in the javadoc comment.

Test cases

1. For each new class introduced there should be one new (JUnit) test class created extending `junit.framework.TestCase` (see the JGAP unit tests for examples)

2. Each test case class is added to a test suite class containing a reference to all unit tests in the package the test suite class is lying. See `org.jgap.impl.AllImplTests` for example.

3. Test clase and test suite classes are lying in a different directory branch than the source code classes. Please see the JGAP CVS: `src` is the toplevel directory for source code and `test` is the toplevel directory for test code. Please notice, that the `src` and `test` directory both have the same package hierarchy!

4. For test cases, the class heading javadoc should write (e.g. for class StringGene):

```
/*
* Tests the StringGene class
* ...
* @author ...
 */
```

5. Don't add any outputs to the console (`System.out` or `System.err`) in your test cases. In nearly any case this could be avoided completely by using explanatory messages with exceptions or the JUnit `fail(...)` method.

6. It might be useful looking at the existing unit tests to get some hints about how to do exception handling etc. Also see Test-Driven Development: Useful Techniques, Resources.

7. The blog entry Intentionally failing JUnit tests could be a useful resource.

8. If you don't feel sure when to write your tests (before, during or after coding), the blog entry When TDD is not optimal might be useful.

CVS

1. For developers, it's ok to commit in-progress code to CVS that doesn't work, but please try to avoid committing code that doesn't compile or that completely breaks everything. If you're going to be doing a lot of work that will likely break everything, we can consider creating a branch for you to work on until your code is at least semi-stable.

2. It is better to check in quite often than to wait too long with committing your work.

3. When checking in something to the CVS, you should do this per class as you could then add a useful comment. It is not forbidden adding the RFE number (the request number from the JGAP sourceforge RFE page) in the comment!

4. If using WinDiff: It's much more comfortable using an external viewer (such as WinMerge) than using the internal textual representation.

In general

1. Finally, you may find JGAP code that doesn't adhere to one of the rules. That doesn't

mean it's ok to disregard them in your code! ;-)

2. If you think an important functionality is missing or there is a design flaw in JGAP, feel free to post this as a RFE.

Copyright © 2002-2007 Klaus Meffert / Neil Rotstan. All rights reserved.

Upgrade from versions prior to 3.0

Introduction

Some JGAP versions (such as 2.4 and 3.0) introduced many concepts as well as many architectural changes. This was seen necessary to evolve the structure of the product.

If you work with a JGAP version prior to 3.0 (i.e. 1.1 or 1.0), and want to upgrade to a newer version , you should read this document. It will guide you thru the process of upgrading to version 2.0 and later upcoming versions.

Hint: If you try to upgrade from a version lower than 2.0 to a version higher than 2.0, first upgrade to 2.0, then test your application. After that, upgrade to 2.4 (or a later version) and test again! The same holds true for upgrading from lower 2.6 to 3.0: First advance to 2.6, then to 3.0

Upgrade Instructions

Reproduction Operator

The ReproductionOperator class is no longer needed. Reproduction takes place via implementations of NaturalSelector's. Main reason: Performance increase.
As this information has been added quite a time after releasing version 2.0, it could be that it applies not for version 2.0. But it applies for the latest JGAP versions for sure!

Central Configuration Object

With version 2.0, the holder of the org.jgap.Configuration object has been made a Singleton.

That means you don't have to give it into any method as it was necessary earlier. After you set up the configuration object in the used manner, you only need to state:

```
Configuration myConfiguration;
//Set up the configuration object
...
//Register the configuration object
Genotype.setConfiguration(myConfiguration);
```

As you see, the holder of the `Configuration` object is org.jgap.Genotype.

With version 3.0, the configuration object is passed to any object needing it via each object's constructor. The access via *Genotype.getConfiguration()* and Genotype.setConfiguration(org.jgap.Configuration) is deprecated and no longer supported by most of the objects.

```
Configuration myConfiguration;
//Set up the configuration object
...
//Register the configuration object with each JGAP class used, e.g.:
IChromosome sampleChromosome = new Chromosome(gaConf,
            new BooleanGene(gaConf),
            chromeSize);
```

Fitness value now of type double

The type of the fitness value has changed from `int` to `double`. The fitness value can be obtained via

```
double fitnessValue = Chromosome.getFitnessValue();
```

Just change your implementation to work with `double`-typed variables instead of `int`-typed ones. There is no performance loss as our extended tests show!

Fitness evaluator in Configuration Object

If you don't construct your `Configuration` object via something like

```
Configuration myConfig = new DefaultConfiguration()
```

you need to make sure you set the `FitnessEvaluator`. You can easily do this by

```
myConfig.setFitnessEvaluator(new DefaultFitnessEvaluator());
```

The fitness evaluator is responsible for deciding when a fitness value is better and when it's not. If you are interpreting the fitness value as a defect rate then the lower the better. If your fitness value is a performace indicator, it should be higher to be better.

NaturalSelector now an abstract class

Change class declaration

The `org.jgap.NaturalSelector` interface has become an abstract class. The interface definition has moved to `org.jgap.INaturalSelector`.

So instead of writing

```
class MySelector implements NaturalSelector {...}
```

you should now write

```
class MySelector extends NaturalSelector {...}
```

Implement new method returnsUniqueChromosomes()

Interface org.jgap.INaturalSelector now holds method returnsUniqueChromosomes:

```
/**
 * @return true: The implementation of the NaturalSelector only returns
 * unique Chromosome's (example: BestChromosomesSelector).
 * false: Also doublettes could be returned (example: WeightedRouletteSelector)
 */
boolean returnsUniqueChromosomes();
```

Just implement something like

```
public boolean returnsUniqueChromosomes() {
   return true;
}
```

in your implementation of org.jgap.NaturalSelector.

Signature of method select changed

The new signature of the select method in org.jgap.NaturalSelector (declared in INaturalSelector) is as follows:

```
 /**
  * Select a given number of Chromosomes from the pool that will move on
  * to the next generation population. This selection should be guided by
  * the fitness values, but fitness should be treated as a statistical
  * probability of survival, not as the sole determining factor. In other
  * words, Chromosomes with higher fitness values should be more likely to
  * be selected than those with lower fitness values, but it should not be
  * guaranteed.
  *
  * @param a_howManyToSelect the number of Chromosomes to select
  * @param a_from_population the population the Chromosomes will be selected from
  * @param a_to_population the population the Chromosomes will be added to
  */
void select(int a_howManyToSelect, Population a_from_population, Population
a_to_population);
```

Normally, calls to this method should only be made by org.jgap.Genotype in method applyNaturalSelectors. If you call the method yourself, adapt the parameters as shown above or have a look at the implementation in Genotype.

Signature of method newGene changed

The new signature of the select method in interface org.jgap.Gene (used in all implementations, as in StringGene) is as follows:

```
/**
   * Provides an implementation-independent means for creating new Gene
   * instances. The new instance that is created and returned should be
   * setup with any implementation-dependent configuration that this Gene
   * instance is setup with (aside from the actual value, of course). For
   * example, if this Gene were setup with bounds on its value, then the
   * Gene instance returned from this method should also be setup with
   * those same bounds. This is important, as the JGAP core will invoke this
   * method on each Gene in the sample Chromosome in order to create each
   * new Gene in the same respective gene position for a new Chromosome.
   * <p>
   * It should be noted that nothing is guaranteed about the actual value
   * of the returned Gene and it should therefore be considered to be
   * undefined.
   *
   * @return a new Gene instance of the same type and with the same
   * setup as this concrete Gene
   *
   * @since 1.0
   */
Gene newGene();
```

Previous signatures included an input parameter of type `org.jgap.Configuration`. The configuration is now set as described in this document (see above).

Method add(Chromosome a_chromosomeToAdd) is private now

As method `add(Chromosome a_chromosomeToAdd)` is private now, you need to adapt your implementations to reflect this: Just change the `public` keyword in your implementation to `protected`.

If you call the `add`-method, remove these calls. They are obsolete as the select method (see above) now calls the `add`-method itself.

Since 2.2: New abstract base class for all Gene implementations

The abstract base class `org.jgap.BaseGene` should be used as a base class for all `org.jgap.Gene` implementations. With that you get all default logic for free.

Since 2.4: toString() of classes implementing Gene interface changed

Please consider that the result of the `toString()` methods in the classes implementing the `Gene` interfaces (`DoubleGene`, `StringGene`, `CompositeGene` etc.).

In the new implementation, the class name is preceeding the rest of the output (e.g. "CompositeGene(...)" or "BooleanGene=true").

Since 2.4: New Method keepPopulationSizeConstant(boolean) in class org.jgap.Configuration

The new method `keepPopulationSizeConstant(boolean)` allows to keep the population size constant after each single evolution, even if no appropriate `NaturalSelector` has been registered with the `Configuration` to be executed as postprocessor. The population size will be kept constant to the size specified with `Configuration.setPopulationSize(int)`.

Since 2.4: Extended interface org.jgap.Gene

In interface org.jgap.Gene two new methods have been introduced: Object getApplicationData() and boolean isCompareApplicationData(). You could easiest adapt your existing Gene implementations without further modifications if you extended them from org.jgap.BaseGene.

Since 2.4: Gene implementations: m_value is private

For all implementations of interface org.jgap.Gene their variable Object m_gene was made private (it was protected before). Now use the more sophisticated methods getAllele() and setAllele(Object) instead. Reason for this change was the need of registering when the allele value changed..

Since 2.6: Supergene-related classes capitalized

All names of classes related to supergenes have been capitalized. Each class name should begin with a capital letter!

Since 2.6: Configuration must be set before calling Gene.setAllele or applyMutation

As setAllele and applyMutation (and possibly other methods) in Gene classes may call mapToValueWithinBounds, which in turn accesses the configuration via Genotype.getConfiguration, the configuration must be set before the call to setAllele, applyMutation and possibly other methods. However, this should be good practice!

Since 2.6: Several fields private instead of protected

These fields are Genotype.m_population, Chromosome.m_genes, Genotype.m_activeConfiguration, CrossoverOperator.m_crossoverRate, MutationOperator.m_mutationRate, IntegerGene.m_upperBounds, IntegerGene.m_lowerBounds, DoubleGene.m_upperBounds and DoubleGene.m_lowerBounds. Use setters resp. constructors instead!

Since 2.6: Method(s) Genotype.randomInitialGenotype

The above method now uses the sample Chromosome provided with the Configuration to determine the class of new Chromosomes to be created. The method with signature (Configuration, Class) has been removed for that!

Since 2.6: Gene's new method newGeneInternal()

Now the newGene() method is a template method in BaseGene calling protected method newGeneInternal. The latter needs to be implemented in all offsprings of BaseGene.

Since 2.6: Interface IGeneConstraintChecker changed

For method verify there are two new paramaters: a_chromosome and a_geneIndex.

Since 2.6: Removed method getNoFitnessValue() in class FitnessFunction

Use the constant `FitnessFunction.NO_FITNESS_VALUE` instead.

Final remarks

It should be quite straight forward upgrading from version 1.1 to 2.0. With version 1.0 it could be a bit more difficult, although realistic.

If you have ***any*** problems accomplishing this or think this documentation is incomplete or inconsistent, feel yourself required to post a message to the forum!

[Documentation Index]

This page is copyright © 2005-2006 Klaus Meffert, GNU free documentation license

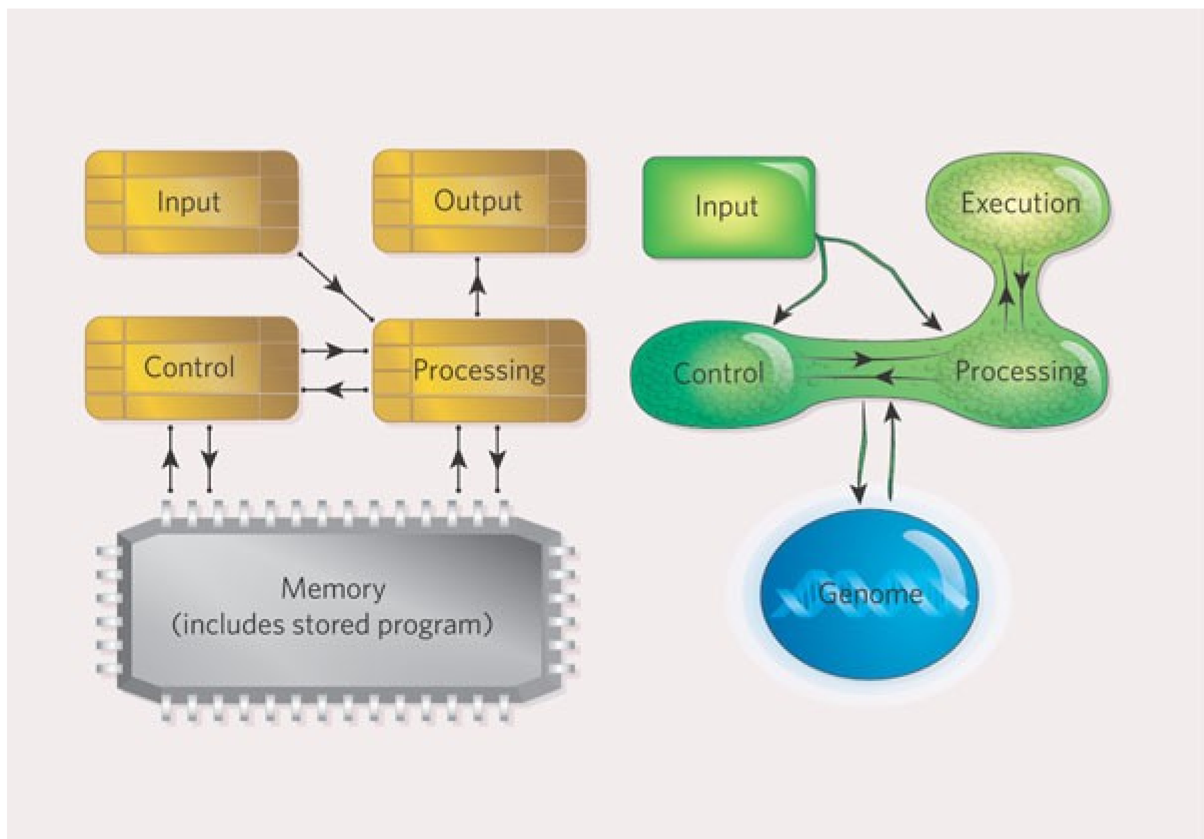
Differences between Nature and a Genetic Algorithm

[JGAP Home]

A Genetic Algorithm is an idealized model of processes and structures occurring in nature. It is by no means meant to follow or copy nature completely. To clarify this **partial** plagiarism, here are some few examples.

General Architecture

We are referring to the interesting article 2020 Computing: Can computers help to explain biology?, from which the following picture was taken.



Mitosis and Meiosis

Mitosis is the process where a cell separates. Meiosis occurs for eukaryotes and transforms one diploid cell into four haploid cells. Both processes are important for the sexual reproduction of humans.

Mitosis and meiosis are complex processes containing sub-processes. In nature, especially errors in mitosis can occur as well as in meiosis. A computer-run Genetic Algorithm does not have problems producing offsprings (asexually) with 100% correctness.

Wobble-Hypothesis

Following the Wobble hypothesis, amino acids are encoded asymmetrically by three codons. Some amino acids are more common than others. Stop codons mark the end of a protein synthesis. As stop codons are not present with Genetic Algorithms, effects depending on them (such as interpreting errors) could not occur.

Mendelian inheritance

In nature, the phenotypic expression of genotypic characteristics allows for intermediary inheritance. For Genetic Algorithms, this holds only true for special implementations of interpretations of the current genotype.

Barr-bodies

Inactive X-chromosome. For women and all female mammals owning two homologous X-chromosomes, one is always inactivated.

Conclusion

As it is not intended copying nature transparently, we are free in implementing a Genetic Algorithm as it seems most reasonable. Aspects involved in judging what could be reasonable are ease of use, capability and performance (time, memory, convergence). One example for a difference between JGAP and other Genetic Algorithms packages is the *Genotype.evolve()* method. In JGAP, we don't use a pool of candidate chromosomes any more, although it is an established concept in other packages. We have not noticed any losses because of this, but instead a speedup.

Copyright © 2005-2007 Klaus Meffert. All rights reserved.

[JGAP Home]

Build a JGAP Grid using your LAN

[JGAP Home]

What is a Grid?

In the case of JGAP, a grid is a distributed computing environment where as many independent computers as possible work on a set of problems. These computers are called workers. Each worker receives tasks from a server, the master. The master gets fed in problems by a client. After that, the master splits those problems into handy units where each single unit will be processed on a single worker. After a worker has done its work for one unit, the result is sent back to the master. The master in turn uses the result to advance the solution, or in the best case stop because a satisfactory solution has been found. In the end, the client receives the solution(s). computed by the workers.

Getting started with your Grid

JGAP provides an example for a grid consisting of the components Server (=master), Worker, and Client. Please see package *examples.grid.fitnessDistributed*. Package *examples.grid* hosts a file *readme.txt* that provides instructions for setting up the grid. The example distributes fitness computations to workers and does the evolution (genetic operations such as mutation and selection) on the client. Other scenarios, such as distributing evolution, are possible as well. JGAP's grid architecture is very flexible because a lot of strategy classes offer a bunch of configuration possibilities.

To use the grid, do the following in the given order:

1. Start the server on a machine of your choice (called SERVER):

```
java -cp ".;jgap-examples.jar;lib\log4j.jar;jgap.jar;lib\commons-cli-
1.0.jar;lib\jcgrid.jar" org.jgap.distr.grid.JGAPServer
```

Ensure that the given jar-files reside in the directories you specify! Each required jar-file is shipped with JGAP!

2. Start as many workers as you like on machines of your choice:

```
java -cp ".;jgap-examples.jar;lib\log4j.jar;jgap.jar;lib\commons-cli-
1.0.jar;lib\jcgrid.jar" org.jgap.distr.grid.JGAPWorker -s SERVER -n myworker1
```

Ensure that the given jar-files reside in the directories you specify! Each required jar-file is shipped with JGAP!
Each worker must have a unique name --> specify it after the "-n" option.
Each worker must point to the SERVER's ip address --> specify it after the "-s" option instead of the above "SERVER".

3. Start a client on a machine of your choice:

```
java -cp ".;jgap-examples.jar;lib\log4j.jar;jgap.jar;lib\commons-cli-
1.0.jar;lib\jcgrid.jar" org.jgap.distr.grid.JGAPClient
examples.grid.fitnessDistributed.GridConfiguration -s SERVER
```

The client must point to the SERVER's ip address --> specify it after the "-s" option instead of the above "SERVER".

After starting the client the server receives a task from the client.
After that the work is split into tasks and the started workers are asked by the server to compute them.
After a worker has computed a task, the result is sent back to the server.
The server in turn returns each solution to the client.
The client can process the result.

To make your GA work with a JGAP grid, especially have a look at the classes *ExampleClient and MyGAWorker* provided in the example package *examples.grid.fitnessDistributed*. Most of the necessary stuff is done by JGAP for you.

Copyright © 2006/2007 Klaus Meffert. All rights reserved.

[JGAP Home]

Upgrade from versions prior to 3.0

[Documentation Index]

Introduction

Some JGAP versions (such as 2.4 and 3.0) introduced many concepts as well as many architectural changes. This was seen necessary to evolve the structure of the product.

If you work with a JGAP version prior to 3.0 (i.e. 1.1 or 1.0), and want to upgrade to a newer version , you should read this document. It will guide you thru the process of upgrading to version 2.0 and later upcoming versions.

Hint: If you try to upgrade from a version lower than 2.0 to a version higher than 2.0, first upgrade to 2.0, then test your application. After that, upgrade to 2.4 (or a later version) and test again! The same holds true for upgrading from lower 2.6 to 3.0: First advance to 2.6, then to 3.0

Upgrade Instructions

Reproduction Operator

The ReproductionOperator class is no longer needed. Reproduction takes place via implementations of NaturalSelector's. Main reason: Performance increase.
As this information has been added quite a time after releasing version 2.0, it could be that it applies not for version 2.0. But it applies for the latest JGAP versions for sure!

Central Configuration Object

With version 2.0, the holder of the org.jgap.Configuration object has been made a Singleton. That means you don't have to give it into any method as it was necessary earlier. After you set up the configuration object in the used manner, you only need to state:

```
Configuration myConfiguration;
//Set up the configuration object
...
//Register the configuration object
Genotype.setConfiguration(myConfiguration);
```

As you see, the holder of the `Configuration` object is org.jgap.Genotype.

With version 3.0, the configuration object is passed to any object needing it via each object's constructor. The access via *Genotype.getConfiguration()* and Genotype.setConfiguration(org.jgap.Configuration) is deprecated and no longer supported by most of the objects.

```
Configuration myConfiguration;
//Set up the configuration object
...
//Register the configuration object with each JGAP class used, e.g.:
IChromosome sampleChromosome = new Chromosome(gaConf,
            new BooleanGene(gaConf),
            chromeSize);
```

Fitness value now of type double

The type of the fitness value has changed from `int` to `double`. The fitness value can be

obtained via

```
double fitnessValue = Chromosome.getFitnessValue();
```

Just change your implementation to work with `double`-typed variables instead of `int`-typed ones. There is no performance loss as our extended tests show!

Fitness evaluator in Configuration Object

If you don't construct your `Configuration` object via something like

```
Configuration myConfig = new DefaultConfiguration()
```

you need to make sure you set the `FitnessEvaluator`. You can easily do this by

```
myConfig.setFitnessEvaluator(new DefaultFitnessEvaluator());
```

The fitness evaluator is responsible for deciding when a fitness value is better and when it's not. If you are interpreting the fitness value as a defect rate then the lower the better. If your fitness value is a performace indicator, it should be higher to be better.

NaturalSelector now an abstract class

Change class declaration

The `org.jgap.NaturalSelector` interface has become an abstract class. The interface definition has moved to `org.jgap.INaturalSelector`.

So instead of writing

```
class MySelector implements NaturalSelector {...}
```

you should now write

```
class MySelector extends NaturalSelector {...}
```

Implement new method returnsUniqueChromosomes()

Interface `org.jgap.INaturalSelector` now holds method `returnsUniqueChromosomes`:

```
/**
 * @return true: The implementation of the NaturalSelector only returns
 * unique Chromosome's (example: BestChromosomesSelector).
 * false: Also doublettes could be returned (example: WeightedRouletteSelector)
 */
boolean returnsUniqueChromosomes();
```

Just implement something like

```
public boolean returnsUniqueChromosomes() {
   return true;
}
```

in your implementation of org.jgap.NaturalSelector.

Signature of method select changed

The new signature of the select method in org.jgap.NaturalSelector (declared in INaturalSelector) is as follows:

```
 /**
  * Select a given number of Chromosomes from the pool that will move on
  * to the next generation population. This selection should be guided by
  * the fitness values, but fitness should be treated as a statistical
  * probability of survival, not as the sole determining factor. In other
  * words, Chromosomes with higher fitness values should be more likely to
  * be selected than those with lower fitness values, but it should not be
  * guaranteed.
  *
  * @param a_howManyToSelect the number of Chromosomes to select
  * @param a_from_population the population the Chromosomes will be selected from
  * @param a_to_population the population the Chromosomes will be added to
  */
void select(int a_howManyToSelect, Population a_from_population, Population
a_to_population);
```

Normally, calls to this method should only be made by org.jgap.Genotype in method applyNaturalSelectors. If you call the method yourself, adapt the parameters as shown above or have a look at the implementation in Genotype.

Signature of method newGene changed

The new signature of the select method in interface org.jgap.Gene (used in all implementations, as in StringGene) is as follows:

```
/**
   * Provides an implementation-independent means for creating new Gene
   * instances. The new instance that is created and returned should be
   * setup with any implementation-dependent configuration that this Gene
   * instance is setup with (aside from the actual value, of course). For
   * example, if this Gene were setup with bounds on its value, then the
   * Gene instance returned from this method should also be setup with
   * those same bounds. This is important, as the JGAP core will invoke this
   * method on each Gene in the sample Chromosome in order to create each
   * new Gene in the same respective gene position for a new Chromosome.
   * <p>
   * It should be noted that nothing is guaranteed about the actual value
   * of the returned Gene and it should therefore be considered to be
   * undefined.
   *
   * @return a new Gene instance of the same type and with the same
   * setup as this concrete Gene
   *
   * @since 1.0
   */
Gene newGene();
```

Previous signatures included an input parameter of type org.jgap.Configuration. The configuration is now set as described in this document (see above).

Method add(Chromosome a_chromosomeToAdd) is private now

As method `add(Chromosome a_chromosomeToAdd)` is private now, you need to adapt your implementations to reflect this: Just change the `public` keyword in your implementation to `protected`.

If you call the `add`-method, remove these calls. They are obsolete as the select method (see above) now calls the `add`-method itself.

Since 2.2: New abstract base class for all Gene implementations

The abstract base class `org.jgap.BaseGene` should be used as a base class for all `org.jgap.Gene` implementations. With that you get all default logic for free.

Since 2.4: toString() of classes implementing Gene interface changed

Please consider that the result of the `toString()` methods in the classes implementing the `Gene` interfaces (`DoubleGene`, `StringGene`, `CompositeGene` etc.).

In the new implementation, the class name is preceeding the rest of the output (e.g. "CompositeGene(...)" or "BooleanGene=true").

Since 2.4: New Method keepPopulationSizeConstant(boolean) in class org.jgap.Configuration

The new method `keepPopulationSizeConstant(boolean)` allows to keep the population size constant after each single evolution, even if no appropriate `NaturalSelector` has been registered with the `Configuration` to be executed as postprocessor. The population size will be kept constant to the size specified with `Configuration.setPopulationSize(int)`.

Since 2.4: Extended interface org.jgap.Gene

In interface `org.jgap.Gene` two new methods have been introduced: `Object getApplicationData()` and `boolean isCompareApplicationData()`. You could easiest adapt your existing Gene implementations without further modifications if you extended them from `org.jgap.BaseGene`.

Since 2.4: Gene implementations: m_value is private

For all implementations of interface `org.jgap.Gene` their variable `Object m_gene` was made `private` (it was `protected` before). Now use the more sophisticated methods `getAllele()` and `setAllele(Object)` instead. Reason for this change was the need of registering when the allele value changed..

Since 2.6: Supergene-related classes capitalized

All names of classes related to supergenes have been capitalized. Each class name should begin with a capital letter!

Since 2.6: Configuration must be set before calling Gene.setAllele or applyMutation

As `setAllele` and `applyMutation` (and possibly other methods) in Gene classes may call `mapToValueWithinBounds`, which in turn accesses the configuration via `Genotype.getConfiguration`, the configuration must be set before the call to `setAllele`, `applyMutation` and possibly other methods. However, this should be good practice!

Since 2.6: Several fields private instead of protected

These fields are Genotype.m_population, Chromosome.m_genes, Genotype.m_activeConfiguration, CrossoverOperator.m_crossoverRate, MutationOperator.m_mutationRate, IntegerGene.m_upperBounds, IntegerGene.m_lowerBounds, DoubleGene.m_upperBounds and DoubleGene.m_lowerBounds. Use setters resp. constructors instead!

Since 2.6: Method(s) Genotype.randomInitialGenotype

The above method now uses the sample Chromosome provided with the Configuration to determine the class of new Chromosomes to be created. The method with signature (Configuration, Class) has been removed for that!

Since 2.6: Gene's new method newGeneInternal()

Now the `newGene()` method is a template method in `BaseGene` calling protected method `newGeneInternal`. The latter needs to be implemented in all offsprings of `BaseGene`.

Since 2.6: Interface IGeneConstraintChecker changed

For method `verify` there are two new paramaters: `a_chromosome` and `a_geneIndex`.

Since 2.6: Removed method getNoFitnessValue() in class FitnessFunction

Use the constant `FitnessFunction.NO_FITNESS_VALUE` instead.

Final remarks

It should be quite straight forward upgrading from version 1.1 to 2.0. With version 1.0 it could be a bit more difficult, although realistic.

If you have ***any*** problems accomplishing this or think this documentation is incomplete or inconsistent, feel yourself required to post a message to the forum!

This page is copyright © 2005-2006 Klaus Meffert, GNU free documentation license

SOURCEFORGE.NET®