DRAFT

ART Artificial Reasoning Toolkit

How To Use

Authors:
Marco Lamieri (lamieri@econ.unito.it)
Gianluigi Ferraris (ferraris@econ.unito.it)

20 December, 2004

Contents

1 Installing and running ART **3**
- 1.1 Requirements . . . 3
- 1.2 Directory tree . . . 3
- 1.3 Compiling and running Art . . . 3
  - 1.3.1 Compile classes . . . 3
  - 1.3.2 Run Art examples . . . 4
  - 1.3.3 Cleaning . . . 4

2 Genetic algorithm **5**
- 2.1 Genetic algorithm overview . . . 5
- 2.2 How to use ART for genetic algorithm . . . 9
  - 2.2.1 Solve an optimization problem . . . 9
  - 2.2.2 Use the genetic algorithm in a simulation model . . . 11

3 Classifier system **13**
- 3.1 Classifier system overview . . . 13
- 3.2 How to use ART for classifier system . . . 16

4 Parameters description and best practice **20**
- 4.1 generalParameters . . . 20
- 4.2 gaParameters . . . 21
- 4.3 genomeParameters . . . 22
- 4.4 csParameters . . . 23

References **26**

1 Installing and running ART

The ART (Artificial Reasoning Toolkit) is a pure Java library devoted to handle Genetic Algorithms and Classifier Systems.

It has been engineered in order to be used into Swarm or others agent based simulation's models, to easy obtain "minded" agents who are fully autonomous, able to decide their own behaviours and able to change it to fit in different environmental conditions. Another main usage of the algorithm is to search bounded optimal solutions in very wide solution spaces and for quite undefined problems.

1.1 Requirements

Art runs with **Sun JDK 1.3.x and 1.4.x.** For obtaining Sun JDK and the installation procedure refer to *http://java.sun.com*.

For compiling and running *art* applications in a easy way is suggested to use **Java Ant** from the Apache Group *http://ant.apache.org/*.

1.2 Directory tree

The Art project's directory tree contains the following subdirectories:

src/ with the source files of the current release

classes/ with the .class files related to src/ files and with the files used to create the art-library.jar.

lib/ with the external libraries (see credits.txt).

log/ with the log of the computations.

doc/ with javadoc of the ART 's api.

examples/ with some applications as example.

parameters/ with parameter's file for the examples.

1.3 Compiling and running Art

1.3.1 Compile classes

To compile Art classes:

1. open MS-DOS Command Line or a Linux shell;

2. from the directory of the project, type

```
ant compile
```

1.3.2 Run Art examples

To run art examples:

1. open MS-DOS Command Line or a Linux shell;
2. from the directory of the project, type:

```
ant -projecthelp
```

to obtain the list of the available examples.

If you do not have Apache Ant the list of the examples and a brief description of them can also be read from the file "parameters/examples.xml".

To run the examples type:

```
ant run
```

and than input the example number.

If you don have Apache Ant installed type:

```
examples.bat (Windows)
examples.sh  (Linux)
```

The trivial examples that can be used to understand how ART works are GaEasyTestCase (for the genetic algorithm) and CsEasyTestCase (for the classifier system).

To embed ART in your own application is sufficient to include in the classpath art-library.jar and import the library using *"import art.*;"* in your classes.

1.3.3 Cleaning

1. open MS-DOS Command Line or a Linux shell;
2. from the directory of the project, type

```
ant clean
```

2 Genetic algorithm

The application that can be created with art are of two type: genetic algorithm and classifer systems.

2.1 Genetic algorithm overview

Starting from "Survival of the fittest" (Darwin, 1859) the Genetic Algorithms (GA) are evolutionary programs that manipulate a population of individuals represented by fixed-format strings of information.

The background theory is the "artificial adaptation" discussed by Holland (Holland, 1975).

The GA are used to solve real-world optimization problems within a very large solution space and "hill defined" problems.

The GA works with some macro phases:

1. an initial population of individuals (solutions) is generated;individuals represent potential solutions to the given problem and are described as binary strings; each character in the individual's data string is called a gene and each possible value that the gene can take on is called an allele.

2. using a fitness proportional approach parents and individuals that are going to survive to the next generation are selected;

3. the selected individuals are evolved by means of reproduction using two operators:

   (a) crossover,

   (b) mutation.

4. process go on until the population converges to a specific individual.

For instance consider an easy problem: compute the square root of 2. In this case the solution space is bounded between 0 and 1. We use a binary representation on 10 digits and there are 1024 numbers (2^10), starting from 0 and ending at 1023 ($2 * 10 - 1$).

binary	binary value	real equivalent
0 0 0 0 0 0 0 0 0 0	0	1 + (0/1023) = 1
0 0 0 0 0 0 0 0 0 1	1	1 + (1/1023)
0 0 0 0 0 0 0 0 1 0	2	1 + (2/1023)
⋮	⋮	⋮
1 1 1 1 1 1 1 1 1 1	1023	1 + (1023/1023) = 2

Figure 1: Binary representation of a real number on 10 digits

As first a population of solutions is generated randomly; for the square root problem, a fixed number of 10 character binary strings are generated.

Darwinian evolution of a population implies that the strongest individuals will probably survive. The fitness of an individual is a numerical assessment of that individual's ability to solve the problem, it is the ability of the individual to satisfy the requirements of the environment. In terms of the square root problem, the perfect individual is the numerical value approximated by 1.414213562373. In economic problems,the profit can be used to generate a fitness function. In this case the fitness function is reciprocal of the error function:

$$Error(x) = \frac{Abs(x^2 - 2)}{2} \qquad Fitness(x) = \frac{1}{Error(x)}$$

The selection process of parents and individual that are going to survive to the next generation use the roulette wheel technique.

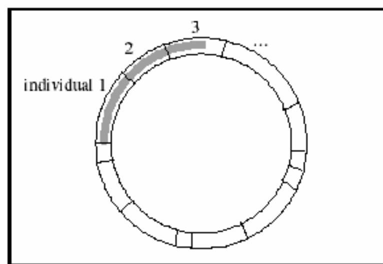

Figure 2: Roulette wheel

The roulette wheel implementation implicitly forces fitness-proportionate reproduction. Selection is divided in 2 steps:

1. Individuals that are going to survive to the next generation are selected;
2. Individuals that are going to reproduce are selected.

The Crossover swaps some of the genetic material of two individuals, creating two new individuals (children), who are possibly better than their parents.

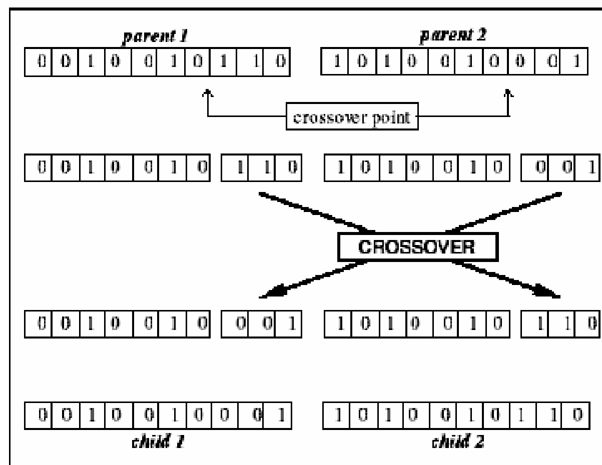

Figure 3: Crossover

In order to recover from this loss of genetic material, the individuals are allowed to change their genes randomly.

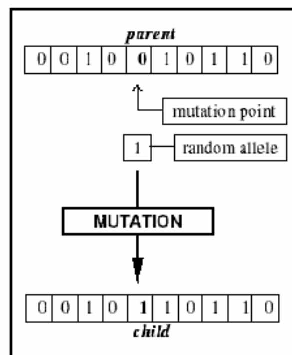

Figure 4: Mutation

John Holland's Schema Theorem (?, ?) is widely accepted as mathematical proof that the genetic algorithm, due to its fitness-proportionate reproduction, converges to better solutions. For this reason the convergence

of the population to a specific genetic set is used to understand when the solution is acceptable. Via the convergence method is possible to solve hill defined problems where the best solution is not known a priori.

Some important remarks about this methodology:

- there is no ultimate goal or problem that must be solved by natural evolution;
- evolution itself does not guarantee the creation of fitter individuals;
- the GA use a fuzzy logic that not always lead to the best solution but to a good one;
- the algorithm is problem independent.

The genetic algorithm's implementation, starting from John Holland's work, introduces some extensions and innovations:

extended alphabet: each gene can be represented by up to 32000 values. In a standard representation the genes have a binary alphabet and so the genomes have to be explicitly translated into the various aspects composing the solution, which after some manipulation, as crossover or mutation, can become meaningless. With the extended alphabet each allele can be a meaningful part of the solution and the translation process is easier.

multi genome: each individual of the population is represented by a chromosome that could be composed by a variable number of genomes. Each genome of a chromosome represent a "substrategy" and the chromosome is the genetic algorithm's formalism for a "strategy" driving the actions of the simulated agent. The multi genome schema give a high degree of freedom to the user in formalizing problems in which coexist different binded aspects.

rescale fitness operator: the natural selection process has been modified in order to improve efficiency and manage negative fitness values. The technique utilized consist in rescale the fitness of all the chromosome.

univocal genome: using this option each value of the alphabet is unique within the genome, it means that in a genome there can not be two or more identical genes.

2.2 How to use ART for genetic algorithm

The ART library for GA can be used in two ways:

1. defining a fitness function that evaluate the solutions proposed by the GA. This approach is useful to solve static optimization problem in a fast way: the problem is embedded in the fitness function.

2. interacting with the GA directly getting each solution and evaluate it setting manually the reward. This second approach has been designed for the use of ART within Agent Based Simulation Models, where the evaluation of a specific solution (typically action of the agent) vary across the simulation time because the environment is dynamic.

In both case as first a configuration file specific for the analysed problem must be prepared. The file must be in XML format and must contain the parameters described in the section 4. An example parameter is ./parameters/examples.xml and can be used as template.

2.2.1 Solve an optimization problem

In order to develop a GA application using the first strategy it is required to create two classes:

1. a custom fitness class that implement the *Fitness* interface;

2. a custom class where *GaManager* is instantiated in order to manage the algorithm.

To explain the use of the library we present an examples defined "Basic Example". The problem is trivial and has no particular meaning but is useful to see how art works and how to use the art library.

The problem is identify a strategy composed by a single rule composed by 10 box (10 genes in the genetic formalism). Each box contain a binary (0 or 1) number in it. As the world starts all the box are filled with random numbers. The goal of the genetic algorithm is to obtain a solution with as many 1 as possible. Obviously the best solution is composed by ten 1 digits.

The rule defining how good is each strategy is defined fitness function, and give a reward (fitness) to each strategy. The algorithm is problem independent and the fitness function is the only part of code describing the problem.

The Fitness function return the fitness of a given solution.

As first import ART library, than define the class as implementation of the Fitness interface.

```java
import art.ga.*;
public class TestCaseFitness implements Fitness
{
 public double fitness=0;

 public double evaluateFitness(GaSolution strategy)
 {
  fitness =0;
  for(int i=0;i<strategy.getRule(0).length;i++)
  {
   fitness += (double) strategy.getRule(0)[i];
  }

  return fitness;
 }

}
```

The only method required is *double evaluateFitness(GaSolution strategy)* that return the fitness of the solution evaluated.

The fitness can be any positive or negative number (double); if negative value are used the option "rescaleFitness" must be set to "1" in the configuration file.

The Solution class has the method *Strategy.getRule(int)* that return an array of integer containing a rule of the solution. The idea is that each solution (chromosome or individual of the algorithm) is composed by different rules (genomes of the algorithm) that represent different aspect of the solution.

In our example the only rule present is the number 0 and contain the binary array to evaluate.

The main class that manage the GA is the following:

```java
import art.ga.*;

public class GaEasyTestCase extends Object {

 // Main  method
 public static void main(String args[])
 {
  // Save initial time to monitor performance
  long initialTime = System.currentTimeMillis();
  GaSolution bestSolution;
  GaManager myGaManager = new GaManager("parameters/examples.xml", "
      basicExamples");
  Fitness aFitness = new TestCaseFitness();
  bestSolution = myGaManager.computeBestSolution(aFitness);
  bestSolution.print();
  long elapsed = (((System.currentTimeMillis()) - initialTime));
  System.out.println("Elapsed time in millisec "+elapsed);
 }
}
```

The main steps are:

- Create an instance of GaManager giving the configuration file "parameters/examples.xml" and the correct project from which read parameters "basicExamples".

```
GaManager myGaManager = new GaManager("parameters/examples.xml", "
    basicExamples");
```

- create an instance of the fitness function defined by the user (TestCaseFitness.java).

```
Fitness aFitness = new TestCaseFitness();
```

- compute the best solution using GaManager as a black box, the only parameters needed are the previously defined fitness function.

```
bestSolution = myGaManager.computeBestSolution(aFitness);
```

- Print to screen the best solution achieved, as is easy to understand that it must be 1.1.1.1.1.1.1.1.1.1

```
 bestSolution.print();
```

2.2.2 Use the genetic algorithm in a simulation model

In this case is sufficient a main class that use the GaManager. This is the same example as before but using a simulation approach.

```
import art.ga.*;
public class AgentBasedTestCase
{
 // Main method
 public static void main(String[] args)
 {
  GaManager myGaManager = new GaManager("parameters/examples.xml", "
      basicExamples");

  GaSolution currentStrategy=null;
  int fitness =0;
  int worldLife=10000;
```

```
  for (int tick=0;tick<worldLife;tick++)
  {
   currentStrategy=myGaManager.getNextSolution();

   fitness=0;
   for(int i=0;i<10;i++)
   {
    fitness += (double) (currentStrategy.getRule(0))[i];
   }
   System.out.println(currentStrategy+" evaluated at tick "+tick+" - 
       generation "+myGaManager.getGenerationNumber());

   myGaManager.setFitness(currentStrategy, fitness);
  }
  System.out.println("\nThe last solution evaluated is:");
  currentStrategy.print();
 }
}
```

The steps are:

- create an instance of GaManager giving the configuration file ”parameters/examples.xml” and the correct project from which read parameters ”basicExamples”.

```
GaManager myGaManager = new GaManager("parameters/examples.xml", "
    basicExamples");
```

- initialize the parameters;

```
GaSolution currentStrategy=null;
int fitness =0;
int worldLife=10000;
```

- manage simulated time with a for loop

```
for (int tick=0;tick<worldLife;tick++)
```

- ask to GaManager a solution, in a real agent based simulation the solution could describe the behaviour of the agents

```
  currentStrategy=myGaManager.getNextSolution();
```

- evaluate the fitness of each solution, as the number of 1 contained (the solution is composed by 10 binary digits, than max fitness will be 10 and will never be negative)

```
for(int i=0;i<10;i++)
{
 fitness += (double) (currentStrategy.getRule(0))[i];
}
```

- tell to the GaManager the fitness of the current solution. When all the strategies have been evaluated an evolution is automaticlly executed by the ga. Pay attention: until each specific rule has been evaluated trought GaManager.setFitness(fitness) you will always get the same rule with GaManager.getNextRule()

```
myGaManager.setFitness(currentStrategy, fitness);
```

- when the world finish the last solution is shown, hopefully it is the best one (1.1.1.1.1.1.1.1.1.1)

```
System.out.println("\nThe last solution evaluated is:");
currentStrategy.print();
```

3 Classifier system

The classifier system implementation starts from Holland and Goldberg's work (Booker, Goldberg, & Holland, 1989).

3.1 Classifier system overview

Classifier system is an induction self-learning system based on a set of simple logical rules called classifiers. Each rule has the following structure: *"if condition then action"*. Classifier system is being optimized by using learning rule called "bucket brigade" and evolutionary algorithms (genetic algorithms). During learning process rules priorities (strengths) are changed. In case of success current and previous activated rules are encouraged. Evolutionary methods are used for new rules searching.

According to Holland Booker et al. (1989) and Wilson and Goldberg (1989), a non-learning classifier system consists of four principal components:

- List of classifiers (population of classifiers).
- List of messages that plays the role of a "message board" for communications and short term memory.
- Input interface (detector) that represents the environment state.
- Output interface (effector) that ensures interaction with the environment or its change.

At any time the Classifier list can contain zero or more Classifier. Each Classifier consists of a string of fixed length and binary alphabet.

A classifier list consists of a set of classifiers looking as follows:

condition1,condition2,...,conditionN:action

For instance consider the CsEasyTestCase example, available in the "./examples/csBasicExample/" folder. This examples compute the integer part of the remainder of a range of numbers divided by a given denominator. If the range is 15 and the denominator is 4 (booth of them can be passed as command line argument) it mean that the classifier find the integer part of the rest of all the numbers between 0 and 15 divided by 4.

dividend	**divisor**	**dividend/divisor**	**dividend%divisor**
0	4	0	0
1	4	0,25	1
2	4	0,5	2
3	4	0,75	3
4	4	1	0
5	4	1,25	1
6	4	1,5	2
7	4	1,75	3
8	4	2	0
9	4	2,25	1
10	4	2,5	2
11	4	2,75	3
12	4	3	0
13	4	3,25	1
14	4	3,5	2
15	4	3,75	3

If we consider $6/4 = 1.5$ the integer part of the remainder is $6\%4 = 2$. In a classifier it will be codified as binary in this way:

$$\overbrace{0110}^{Condition}\ \overbrace{0010}^{Action}$$

$$Condition = 6 \ \ Action = 2$$

When the condition part of the classifier matches the input message, activation of the classifier occurs, i.e. the classifier puts one or more messages on the message list.

In the situation in which the input message does not meet any classifier the covering procedure is employed: an input message is picked and some of its symbols are replaced with symbol # (wildcard).

The detector is subprogram that generates the input messages (that are matched with the conditions). In the CsEasyTestCase example the Detector create the numerator of the division.

The output interface is a device or subprogram that receives action messages and on their basis performs manipulations with the environment. In the example the Effector return to the user the action to be evaluated.

The wildcard, marked as #, is used to means "every value match" in a specific position. Rules with many wildcard must be considered less relevant than rules more "specific". Specificity is the number of non # symbols in the condition part divided by its length. The specificity is used in the Bucket Brigade algorithm. The more # symbols a classifier has, the more specific it is. The value of specificity varies from 0 through 1 and is calculated by formula:

$$Specificity = \frac{(Length - WildcardNumber)}{Length}$$

The computations of a CS are the follows:

1. add messages obtained with the help of the input interface to the input message list;

2. compare all the messages on the message list with the condition parts of all classifiers and remember all the classifiers for which coincidence has been observed;

3. select by a auction some classifier actions;

4. pass the messages obtained through the output interface;

5. replace the contents of the message list with new messages from the environment;

6. if a classifier chain is implemented add to message list also the classifier's action;

7. auction's bid is paied using the bucket brigade algorithm.

The bucket brigade algorithm was first introduced by Holland. It is based on a simplified model of economics. The algorithm consists of two principal components: auction and clearing house.

When condition parts of classifiers coincide with the input message, they send their messages not directly but after an auction. At the auction the classifiers bid proportionally to their strength. A classifier with a greater bid has bid competition to win against the other classifiers.

After a classifier has been chosen, it has to pay through the clearing house to all participants. It pays to the classifiers thanks to which it was able to participate in the auction at the preceding step. At each step all the classifiers pay taxes.

As a result of this manipulation, chains of activated classifiers emerge. The last activated classifier in the chain sends its message to the environment and in case of successful action it receives a reward from the environment. Multiple reward of the given chain will lead to a constant growth in the strength of the classifiers entering the chain. This, in turn, will raise the probability of the selection of these classifiers at the auction.

The learning part of the CS is used to vary the classifier's list using a genetic algorithm.

3.2 How to use ART for classifier system

To embed a CS in your application few lines of code are sufficient. From a logical point of view the steps required are:

1. import art package in your classes;

2. create a parameter's file;

3. create a custom class that implement the interface *Detector* defining the method *getCondition()*;

4. create an instance of the CsManager class, that is the main class accessed by the user to manage all the basic operations for a standard use of the Classifier System;

5. pass the environment message to the Classifier System;

An example of Detector realized to generate message for the classifier as binary string starting from integer on base 10 is the following.

```
package csBasicExamples;
import art.cs.Detector;

public class CsBasicDetector implements Detector {
 int num, length;
 public CsBasicDetector() {
  super();
 }

 public int[] getMessage() {
  int[] msg = new int[length];
  char[] msgChar1= new char[length];
  char[] msgChar2=Integer.toBinaryString(num).toCharArray();
  int pad=length-msgChar2.length;
  for(int i=0;i<pad;i++)
   msg[i]=0;
  for (int i=0; i<msgChar2.length; i++)
  {
   msg[i+pad]=((int)msgChar2[i])-48;  // Shift the ascii code of 0 (48)
            // and 1(49) to desired value
  }
  return  msg;
 }

 public void setNum(int numerator, int length){
  this.num=numerator;
  this.length=length;
 }
}
```

The only required method is getMessage() and return a numerator "num" in binary format. The CS use this value to compute num%denum and "num" is intended as a message from the environment.

The method *setNum(int numerator, int length)* set the numerator in base 10 to be converted in binary.

The main class that manage the CS is this one:

```
package csBasicExamples;
import art.cs.*;
import java.util.*;

public class CsEasyTestCase extends Object {
 static int num, denum, length;

 //  Main  method
 public static void main(String args[])
 {
    // denumerator
  denum = Integer.parseInt(args[0]);
  // range of numerators (0-domain)
```

```
    int domain=Integer.parseInt(args[1]);
    // number of cycle to be computed
    int cycle=Integer.parseInt(args[2]);
    // length of the message generated by the detector
    length=4;

    Random r = new Random(1234);
    // Create an instance of the detector to generate the environment message.
    CsBasicDetector detect = new CsBasicDetector();
    // Create an instance of the CsManager to manage all the CS operation.
    CsManager myCsManager = new CsManager("parameters/examples.xml", "
        basicExamples",detect);
    // Get the effector in order to obtain the computed actions from the CS
    Effector eff = myCsManager.getEffector();
    // reward
    int reward;
    // solution
    int sol;
    int integ;
    for(int i=0;i<cycle;i++){
     num= r.nextInt(domain);
     sol= num % denum;
     // create a message via detector
     detect.setNum(num, length);
     // give the message to the classifier
     myCsManager.buildMessage();
     // start a cycle of elaboration
     myCsManager.go();
     // evaluate all the actions proposed by the classifier
     while(eff.hasNext())
     {
         // get an action
       int[] in=eff.getNextAction();
       String str="";
       // evaluate
       for(int ii=0;ii<in.length;ii++){
        str+=in[ii];
       }
       integ=Integer.parseInt(str, 2);
       if (integ==sol)
        reward=1;
       else
        reward =0;
       // set the reward of the action
       myCsManager.setReward(reward);
     }
     // every 100 cycle print current situation
     if(i%100==0)
         System.out.println("Winners at "+i+" are "+myCsManager.getMAWinners()
             +"%");
    }
    System.out.println("Final winners are "+myCsManager.getMAWinners()+"%");
    // after the elaboration print the final classifier list
    myCsManager.printClassifierList();
    // save the final classifier list to file
    myCsManager.saveClassifierList("log/classifierList.txt");
  }
}
```

In order to import ART in your project is sufficient use the statement:

```
import art.cs.*;
import art.util.*;
```

In order to define a configuration file look to parameters description and use the axample file ”examples.xml” as template to customize your parameters.

In order to create an instance of CsManager two steps are required:

1. pass to the constructor of the class the configuration file name and the project name;
2. create an instance of the implemented Detector class and pass it to the constructor.

The statements are:

```
DetectorImpl detectorImpl = new DetectorImpl();
CsManager myCsManager = new CsManager(filename.xml, projectname,
    detectorImpl);
```

Once the CsManager is created call the *CsManager.BuildMessage()* method to invoke the class implementing the Detector and create a message. From an internal point of view this method invoke the *getMessage()* method of the interface *Detector*.

When the desired number of message are created call *CsManager.go()* to make the CS compute the action(s).

The statements are:

```
myCsManager.BuildMessage();
myCsManager.go();
```

After the actions are computed use *CsManager.getEffector()* to obtain an instance of Effector and extracting the actions using *Effector.hasNext()* and *Effector.getNextAction()*.

Every action must be evaluated and the reward of the action must be setted one by one using *GaManager.setReward(double)*.

```
Effector eff = myCsManager.getEffector();
while(eff.hasNext()){
   int[] action=eff.getNextAction();
}
myCsManager.setReward(reward);
```

4 Parameters description and best practice

As template is possible to use the file "./parameters/examples.xml"

The configuration file must have a fixed structure:

```
<?xml version="1.0"?>
<art-configuration>
 <project name="projectName" type="type">
   ...
 </project>
</art-configuration>
```

Each project must be defined using this a project free project name an a project type. The project type can be

ga for genetic algorithm;

cs for classifier system.

4.1 generalParameters

randomSeed this is the seed of the random number generator that allow to control It is of type "long" and it is used to initialize all the random number generators of the application. In a more detailed way there is a main random number generator that, by MyRandomMaker.class, produce many different random number generators, one for each object that needs it (for example each gene has its own random generator), initialising each produced random number generator with the nextLong() of the main generator controlled by the randomSeed parameter.

verboseness it is the desired level of verbosity during the run. The options alowed are:

- **0** = silent mode (standard value);
- **1** = verbose;
- **2** = very verbose;
- **3** = debug mode;

logStatistics This option enable or disable logging of some statistics on the evolution process. The chromosome statistics are written in file: "./log/chromosomeStatistics.txt" and the population statistics are written in file: "./log/populationStatistics.txt". The alowed values are:

- **0** = no logging (standard value);

1 = log population statistics only (population name, generation, average fitness, standard deviation, max fitness and min fitness in the population);

2 = log population statistics and chromosome statistics (chromosome name, belonging population, age fitness)

Chromosome statistics slow down much computation, use value "2" only for very small cases.

plotGraph if equal 1 some dynamic graphics have to be plotted. Values are:

0 = do not plot graph;

1 = plot graph (standard value).

4.2 gaParameters

Those parameters are specific for the genetic algorithm but they must be defined also when a learning classifier system is used because the learnin part evolve using the ga implementation.

The parameters are:

populationSize it is the size of population, that means the number of chromosomes managed. Type Integer.

A standard value can be about 100 for a GA and 1.5xMaxRulesNumber for a CS.

stoppingMethod when the ga has to stop evolutions:

0 = never stop (standard value for CS and for use in AB simulations of GA);

1 = stop at a given convergence (standard value for GA);

2 = stop at a given generation;

3 = stop when a given fitness is reached.

stoppingValue value to stop evolution at. Depend on stopping method:

if stoppingMethod = 1 it is the convergence value to stop at(double bounded 0-1);

if stoppingMethod = 2 it is the generation number to stop at ($integer > 1$);

if stoppingMethod = 3 it is the fitness to stop at (double).

chromosomeToReturn select if computation of the best strategy return:

- **1** = return most diffused chromosome (standard value);
- **2** = return fittest chromosome.

normalizeFitness define if the fitness has to be rescaled. This option is useful to manage negative fitness and it also improve GA performance. Options are:

- **0** = do not rescale fitness (standard value for CS);
- **1** = rescale fitness (standard value for GA).

mutateAdults if mutation are applied also to chromosome born 1 or more generation ago (adults).

This option is usually off (value =0) and can be useful only in special cases). Valid vales are:

- **0** = do not mutate adults, mutate only child (standard value);
- **1** = mutate adults.

fittestNeverDie the fittest individual will always be selected to survive to the next generation.Valid vales are:

- **0** = do not force selection for survive of the fittest chromosome (standard value);
- **1** = force selection for survive of the fittest chromosome.

turnoverRate part of the population that is going to become parents in order to produce the next generation. It is a double.

Standard value is 0.8 for GA and 0.3 for CS.

chromosomeLength number of genomes in each chromosome.

The value user defined for the GA and must be 2 for the CS.

4.3 genomeParameters

All those parameters are related to the genome. It is required to specify them for each genome that compose a chromosome. The number of the different genome type is equal to the total number of genomes in each chromosome. Each genome is a tag `<genome>` block in configuration file.

Those parameters must be defined also for CS projects.

The parameters of each genome are the following:

genomeLength number of genes for this genome.

domain alphabet used by this genome. The domain is the number of states in witch a gene can be. The alphabet start from 0 to domain-1. Usually a standard ga has a binary alphabet, than the domain will be 2 (that means 2 possible modes: 0 and 1). Standard value is 2.

For CS the values must be 3 for the condition part (binary plus wildcard) and 2 for the action part (binary).

genomeType define if the gene within the genome must be random (default value for most problems) or univocal (only for specific problem like the quiz example). In a univocal genome each gene can be present only once, no duplicated gene are allowed within the same genome.

The CS allow only random genomes.

Value are:

0 = random genome (standard value);

1 = univocal genome.

crossOverPointNumber number of cross over point for each genome every evolution.

Standard value is 1.

crossOverRate this rate identify the probability of cross over of each chromosome of the population.

Standard value is 0.5 for GA and 1 for CS.

mutationRate this rate identify the probability of random mutation applied to each gene.

Standard value is 0.001.

4.4 csParameters

For the classifier system there is a set of specific parameters that are not need in "ga" projects.

The parameters are:

evolutionRate rate of evolution used to decide when perform an evolution.

Standard value is 0.01.

evolutionBrake if the reward of the action is correct (reward ¿= 1) for the number of cycle declared by this parameter the evolution is excluded in order to avoid good classifier's die. Standard value is 100. If set at 0 the brake is disabled, this configuration is suggested when the memoList is used (memoListSize¿0).

coverDetectorStrategy the cover detector is a function called when arrive from the environment a message that do not mach any classifier. The cover detector modify the similar and less fitted classifier forcing the match.

The standard value is 1.

0 = the cover detector replace the classifier's positions having values different from the message with the message values. With this strategy you obtain specific classifiers after few cycle. (Standard value).

1 = the cover detector replace the classifier's positions having values different from the message with wildcard #. This strategy create low specificity classifier

selectionForDieStrategy There are diferent algorithm for the selection for die of a classifier.

The standard value is 0;

0 = Fast and efficient algorithm that select a predefined number of individual similar to the child (differentiationRate*populationSize) and kill the lower fitted. (Standard value)

1 = Goldberg's selection algorithm. Quite good results but very slow.

2 = differentiation algorithm, this approach as the aim to differentiate the population as much as possible and has to be used only in particular cases.

differentiationRate parameters used by algorithm 0 of selectionForDieStrategy.

Standard value 0.5.

initialFitness initial fitness of the population.

Standard value 10.

wildCardRate probability to have a wildcard in every gene of the initial population.

Standard value 0.2.

lifeTax rate of the fitness paid at every cycle by every classifier.

Standard value 0.005.

bidTax rate of the fitness paid to make a bid.

Standard value 0.25.

baseBidRate bid not proportional to the specificity.

Standard value 0.25

proportionalBidRate bid proportional to the specificity.

Standard value 0.50

randomBidBias random bias on the bid to avoid bid to be equals.

Standard value 0.001.

maxAuctionWinners max number of classifier winning the auction.

Standard value 1.

maxActions max number of actions returned to the environment. If maxAuctionWinners¡maxActions a chain is implemented.

Standard value 1.

movingAverageLag lag to be used to compute the moving average plotted in the win gaph.

Standard value 100.

memoListSize this is an improvement to the method used to simulate human memory on different situation. This is usefull when some specific conditions are harder to solve than the others: the fitness is normalized in respect to the max, min and average fitness of the specific condition. If the action is evaluated above max fitness in past action for the same condition the reward is set to 2, if is below the average fitness of previous action for this condition the reward is set to 0 and if it is between average and maximum the reward is 1. Standard value 0 (disable this option).

References

Booker, L. B., Goldberg, D. E., & Holland, J. H. (1989). Classifier systems and genetic algorithms. *Artificial Intelligence*.

Darwin, C. (1859). *The origin of species*. Murray.

Holland, J. (1975). Adaptation in natural and artificial systems. *University of Michigan Press*.

Wilson, S. W., & Goldberg, D. E. (1989). A critical review of classifier systems. *Proceedings of the third international conference on Genetic algorithms*.