epochX

genetic programming for research

Quickstart Guide

for version 1.0 beta

2009-11-04

Licensing

Introduction

What is EpochX?

EpochX is a Java API for evolving computer programs using tree based Genetic Programming[1] approach. When designing a Genetic Programming system there are many design decisions to be made, and there is very little coverage in the literature about what decision was made by an author. This makes it very difficult to reproduce another researcher's results and impossible to compare directly.

The Complete Guide makes transparent all design decisions made with EpochX, such that it should be possible to reproduce an equivalent system. Further, EpochX's sister project, the grammatical evolution system XGE, was constructed to an identical set of design decisions in order to make results from the two systems directly comparable.

Who is EpochX for?

EpochX is intended primarily for **researchers** who are working with Genetic Programming. This means that performance is not the main concern, rather it is designed with the intention of making statistical data about runs accessible in real time and to be easily modified and extended to mould to your research.

Who is this guide for?

This guide is for anyone who is looking to have a quick play with Genetic Programming or who is looking to try out the power of EpochX without getting too deep just yet. There is also a Complete Guide available which gives a much more in depth explanation of the components of EpochX and how to get the most out of them for your research.

1 See Genetic Programming by Koza for more information about the Genetic Programming algorithm.

Installation

You can download the latest version of EpochX from www.epochx.com.

EpochX is a Java API, and as such it contains no installer or executable. The download will contain the jar file *epochx-1.x.jar*. To use EpochX in your Java application, simply ensure this jar file is on your classpath. This will then give you complete programmatic access to a range of powerful features, instead of being limited to what can be specified in a properties file, as with most GP software.

Getting Started

The first thing you need to know to use EpochX, is about models. A **model** is simply a Java class written to control a GP run. Models provide the function and terminal sets, all sorts of run parameters and most importantly, the fitness function. All models implement the GPModel interface, which defines methods for all the information that the system needs to evolve your genetic programs. Typically a model will actually extend GPAbstractModel rather than GPModel though.

GPAbstractModel provides default values for parameters where possible to make it quick to throw a new model together to test some idea you may have. It also provides convenient setter methods for all parameters, so even when you do want to override the default parameters, you can do so with a simple setXXXParameter() method on construction. The final thing that GEAbstractModel gives us for free is the basic printing to the console of the statistics we request. This can all be easily overridden, but it handles the base case with no work.

So, lets get started with EpochX. The main entry point into EpochX is the static method:

```
GPController.run(GEModel model)
```

This single method call will start the evolution process, executing the model that is passed in as a parameter. GPController is found in the com.epochx.core package. The model object that is passed specifies all the control parameters, and the EpochX system will proceed according to the settings it defines.

Generic Typing

EpochX makes heavy use of Java's generic data typing. It allows us to create an extremely robust system which enforces data types throughout your genetic programs. A description of how to use generic types in Java can be found on Sun's Java language guide.

Most of the classes in EpochX require a generic type. In almost all cases this is simply the data-type of the genetic programs you're evolving. So, for problems like even parity or multiplexers this will be <Boolean> and for symbolic regression problems it will be <Double>. Only single data types are allowed within the programs evolved in EpochX, so these generic types define the required argument and return types of all functions and terminals in use. This will become clearer later.

Example #1: Built-in Models

EpochX comes with a number of built-in models which define suitable function and terminal sets and fitness functions for the common benchmark problems. To get started we will try and get one of these running, we'll use even-4 parity[2]. You can find the even-4 parity model in the com.epochx.models.parity package.

Create a new Java class, with a main method. Call the class whatever you wish. Inside the main method there are just three lines of code we need in order to start evolving some GP programs.

```
1. GPAbstractModel<Boolean> model = new Even4Parity();
2. model.setGenStatFields(new GenerationStatField[]{
          GenerationStatField.FITNESS_MIN,
          GenerationStatField.BEST_PROGRAM});
3. GPController.run(model);
```

Line 1 constructs the model, line 2 defines what statistics we would like to generate and line 3 executes our model. It is that simple. By default, a GPAbstractModel will print statistics to the console. We could have left out line 2 and our model would still have evolved, but there would not have been any output.

If you execute this class the console output should be a stream of 3 columns, something like:

```
0     5.0    OR(IF(OR(AND(OR(AND(D2 D3) AND(D2 D1)) …
1     5.0    OR(IF(OR(AND(OR(AND(D2 D3) AND(D2 D1)) …
2     3.0    IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) …
3     3.0    IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) …
4     2.0    IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) …
5     2.0    IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) …
6     2.0    IF(IF(NOT(IF(OR(IF(D3 D1 D3) NOT(D1)) …
      …      …      …
```

The first column is the generation number, where generation 0 is the result of initialisation. Then the second and third columns are the minimum fitness (i.e. the best fitness when using standardised fitness) and the best program, as requested on line 2 of our code. You could experiment with different stats fields available in the GenerationStatField enum.

If everything is running smoothly then you should see the minimum fitness value gradually dropping towards zero in the later generations.

Setting parameters

This is just the bare minimum to run a model, and currently we are using all the default parameter values as specified by GPAbstractModel, but now we want to provide our own. GPAbstractModel has many methods to set parameters, see the JavaDoc documentation for a full list, but we will use a couple of them now. Let's set the population size, the number of generations and the maximum program depth.

```
1. GPAbstractModel<Boolean> model = new Even4Parity();
2. model.setPopulationSize(500);
3. model.setNoGenerations(100);
4. model.setMaxProgramDepth(8);
5. model.setGenStatFields(new GenerationStatField[]{
          GenerationStatField.FITNESS_MIN,
          GenerationStatField.BEST_PROGRAM});
6. GPController.run(model);
```

Lines 2, 3 and 4 have been inserted to set these parameters. Another interesting parameter we can set is the number of runs (setNoRuns(int)) which makes performing 50 or 100 runs for the sake of experimentation very

2 For a description of the even parity problems, see Koza's first book - Genetic Programming.

easy.

Setting components

As well as the basic run parameters, many of the components in EpochX are configurable and changeable. In exactly the same way as we set the parameters, we can set these components. In the following code we will change the crossover operator, the program selector and the random number generator.

```
1. GPAbstractModel<Boolean> model = new Even4Parity();
2. model.setCrossover(new UniformPointCrossover<Boolean>(model));
3. model.setProgramSelector(new TournamentSelector<Boolean>(model, 7));
4. model.setRNG(new MersenneTwisterFast());
5. model.setGenStatFields(new GenerationStatField[]{
          GenerationStatField.FITNESS_MIN,
          GenerationStatField.BEST_PROGRAM});
6. GPController.run(model);
```

Lines 2, 3 and 4 have been changed to set the new components. The main difference is that we must pass in an instance of the component to use, and it is common for components to need access to the model that is being used, since it may define relevant parameters for the component. Some components will also have their own options, settable with arguments to the constructor or mutator methods, such as the new TournamentSelector we created which takes the tournament size.

Example #2: Creating a New Model

Whilst the benchmark models which are provided with EpochX define many of the problems which you may wish to execute, undoubtedly, it is necessary to be able to create your own. Creating a new model in EpochX is a simple process though. A 6-bit multiplexer model already exists in the built-in EpochX models, but for the purposes of demonstrating a simple model, we'll create a new one here.

Create a new Java class called Mux6, which extends GPAbstractModel using a generic type of Boolean. There are three abstract methods which you are required to define implementations for, they are:

- `public double getFitness(CandidateProgram<Boolean> program);`
- `public List<FunctionNode<Boolean>> getFunctions();`
- `public List<TerminalNode<Boolean>> getTerminals();`

The getFitness method is called to evaluate the fitness of a given program which is represented as a CandidateProgram object. This method will be called every time the EpochX system needs to know the fitness of a program, and it is expected that it will return a double representing a standardised fitness value, where 0.0 is the best score. How the method calculates the score is up to you, the model designer, to decide. Although typically it is obviously going to be based upon the quality of the program after executing it upon some inputs. We will come back to writing our fitness function shortly.

The other two methods – getFunctions() and getTerminals() - are simply intended to return the function and terminal sets. In EpochX, the program trees are constructed from Node objects, which can each have children. There are 2 main types of Node – FunctionNodes and TerminalNodes, then there are also Variables which are a type of TerminalNode (they extend from it). These two methods must simply return a list of these two types of nodes, which the EpochX system will later create copies of to construct program trees.

The getFunctions() method implementation:

```
public List<FunctionNode<Boolean>> getFunctions() {
      List<FunctionNode<Boolean>> functions =
                          new ArrayList<FunctionNode<Boolean>>();
      functions.add(new IfFunction());
      functions.add(new AndFunction());
      functions.add(new OrFunction());
      functions.add(new NotFunction());

      return functions;
}
```

The getTerminals() implementation is subtly different since we need to keep a copy of all the variables in a field so we can change their values to each of our test inputs during fitness evaluation.

```
public List<TerminalNode<Boolean>> getTerminals() {
      List<TerminalNode<Boolean>> terminals =
                          new ArrayList<TerminalNode<Boolean>>();
      terminals.add(variables.get("D3"));
      terminals.add(variables.get("D2"));
      terminals.add(variables.get("D1"));
      terminals.add(variables.get("D0"));
      terminals.add(variables.get("A1"));
      terminals.add(variables.get("A0"));

      return terminals;
}
```

So this implementation assumes that we have already created Variable (TerminalNode) nodes in a hashmap called variables. This is convenient because it then allows us to refer to each variable by name.

So, back to the getFitness method. The fitness of a program in our 6-bit multiplexer will be 64 minus the number of correct outputs after trying all 64 input combinations, so a fitness of 0.0 will mean success on all inputs. Assuming we have initialised a boolean[][] field, inputs, that contains all the possible combinations of inputs, and outputs, that contains the corresponding correct output response, the structure of our getFitness function will be:

```
public double getFitness(CandidateProgram<Boolean> program) {
	double score = 0;

	for (int i=0; i<inputs.length; i++) {
		// Set the variables.
		variables.get("A0").setValue(inputs[i][0]);
		variables.get("A1").setValue(inputs[i][1]);
		variables.get("D0").setValue(inputs[i][2]);
		variables.get("D1").setValue(inputs[i][3]);
		variables.get("D2").setValue(inputs[i][4]);
		variables.get("D3").setValue(inputs[i][5]);

		Boolean result = program.evaluate();

		if (result == outputs[i]) {
			score++;
		}
	}

	return 64 - score;
}
```

To evaluate our program with each set of variables, we set the value of each of our Variable objects which we have held a copy of in a HashMap. Then we simply call the evaluate() method on the program to trigger an evaluation of the program tree.

Once we've completed our model, we can then execute it in the same way as in Example #1, by passing an instance of it into the GPController.run(GPModel) method.

The complete code for this example can be found in Appendix B.

Conclusions

This is as far as we are going to go in the Quickstart Guide, hopefully it was enough to demonstrate some of the power of EpochX and the ease with which you can wield it. It is only possible to scratch the surface of EpochX with such a short guide, so we recommend taking a look at the Complete Guide for more detailed information. Some of the features which have not been discussed here but which are covered in the Complete Guide, include:

- Life Cycle Events: EpochX includes a mechanism for monitoring and modifying execution in real time as a run progresses, these are part of the Life Cycle system.
- Statistics: We have seen generation stats that are generated at the end of each generation, but EpochX also has run stats, crossover stats and mutation stats, for retrieving statistical information about each of these events as they happen. We can also do a lot more with them than just print them to the console.
- Writing custom components: All genetic operators, as well as many other components of the EpochX system are pluggable; they can be easily changed, and writing your own crossover operator or initialiser is relatively simple.

Appendix A: Full Code for Example #1

```
import com.epochx.core.*;
import com.epochx.model.parity.*;
import com.epochx.op.crossover.*;
import com.epochx.op.selection.*;
import com.epochx.random.*;
import com.epochx.stats.*;


public class Test {

	public static void main(String[] args) {
		GPAbstractModel<Boolean> model = new Even4Parity();
		model.setPopulationSize(500);
		model.setNoGenerations(100);
		model.setMaxProgramDepth(8);
		model.setCrossover(new UniformPointCrossover<Boolean>(model));
		model.setProgramSelector(new TournamentSelector<Boolean>(model, 7));
		model.setRNG(new MersenneTwisterFast());

		model.setGenStatFields(new GenerationStatField[]{
				GenerationStatField.FITNESS_MIN,
				GenerationStatField.BEST_PROGRAM});
		GPController.run(model);
	}

}
```

Appendix B: Full Code for Example #2

```java
import java.util.*;

import com.epochx.core.*;
import com.epochx.representation.*;
import com.epochx.representation.bool.*;
import com.epochx.util.BoolUtils;


public class Mux6 extends GPAbstractModel<Boolean> {

        private Map<String, Variable<Boolean>> variables;
        private boolean[][] inputs;
        private boolean[] outputs;

        public Mux6() {
                variables = new HashMap<String, Variable<Boolean>>();

                // Define variables.
                variables.put("D3", new Variable<Boolean>("D3"));
                variables.put("D2", new Variable<Boolean>("D2"));
                variables.put("D1", new Variable<Boolean>("D1"));
                variables.put("D0", new Variable<Boolean>("D0"));
                variables.put("A1", new Variable<Boolean>("A1"));
                variables.put("A0", new Variable<Boolean>("A0"));

                inputs = BoolUtils.generateBoolSequences(6);
                outputs = generateOutputs(inputs);
        }

        @Override
        public double getFitness(CandidateProgram<Boolean> program) {
                double score = 0;

                for (int i=0; i<inputs.length; i++) {
                        // Set the variables.
                        variables.get("A0").setValue(inputs[i][0]);
                        variables.get("A1").setValue(inputs[i][1]);
                        variables.get("D0").setValue(inputs[i][2]);
                        variables.get("D1").setValue(inputs[i][3]);
                        variables.get("D2").setValue(inputs[i][4]);
                        variables.get("D3").setValue(inputs[i][5]);

                        Boolean result = program.evaluate();

                     if (result == outputs[i]) {
                        score++;
                        }
                }

                return 64 - score;
        }

        @Override
        public List<FunctionNode<Boolean>> getFunctions() {
                List<FunctionNode<Boolean>> functions =
                                          new ArrayList<FunctionNode<Boolean>>();
                functions.add(new IfFunction());
                functions.add(new AndFunction());
                functions.add(new OrFunction());
                functions.add(new NotFunction());

                return functions;
        }

        @Override
        public List<TerminalNode<Boolean>> getTerminals() {
                List<TerminalNode<Boolean>> terminals =
                                          new ArrayList<TerminalNode<Boolean>>();
                terminals.add(variables.get("D3"));
                terminals.add(variables.get("D2"));
                terminals.add(variables.get("D1"));
                terminals.add(variables.get("D0"));
                terminals.add(variables.get("A1"));
                terminals.add(variables.get("A0"));

                return terminals;
        }

        /*
         * Generates the correct outputs for the 6-bit multiplexer from
         * the given inputs.
         */
        private boolean[] generateOutputs(boolean[][] in) {
                boolean[] out = new boolean[in.length];

                for (int i=0; i<in.length; i++) {
                        if(in[i][0] && in[i][1]) {
                       out[i] = in[i][2];
                     } else if(in[i][0] && !in[i][1]) {
                        out[i] = in[i][3];
                     } else if(!in[i][0] && in[i][1]) {
                        out[i] = in[i][4];
                     } else if(!in[i][0] && !in[i][1]) {
                        out[i] = in[i][5];
                     }
                }
```

```
                return out;
        }

        public static void main(String[] args) {
                GPController.run(new Mux6());
        }
}
```