Quickstart Guide

for version 1.1

2010-04-15

Licensing

EpochX is licensed under the GNU General Public License.

EpochX is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version. EpochX is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

A copy of the GPL license should have been included in the download with EpochX.

Introduction

What is EpochX?

EpochX is a Java framework for studying the evolution of computer programs generated using Genetic Programming[1] algorithms. Version 1.1 has 3 fully supported representations – strongly-typed tree GP, context-free grammar GP[2] and Grammatical Evolution[3]. However, EpochX's extendible structure makes it possible to implement entirely new representations within the bounds of the evolutionary framework. A wide range of implementations are provided to perform selection, initialisation, crossover, mutation... as well as many of the common benchmark problems that GP is frequently tested upon.

Who is EpochX for?

EpochX is intended primarily for **researchers** who are working on Genetic Programming theory. The sort of person who might benefit from using EpochX are those who are interested in the distribution of depth/length/diversity etc or a large range of other statistical data about their run. Or those who wish to have a truly dynamic system where statistics are accessible in real time and parameters can be updated as a run progresses. Since EpochX is a Java framework, a prerequisite is a basic ability to program in Java.

Who is EpochX not for?

Raw speed junkies. EpochX is built sensibly to avoid performance issues, and has support for caching and threading. But, it is not aiming to be the fastest GP system. Being extendible, flexible and providing easy access to data is of far more importance here.

Who is this guide for?

This guide is for anyone just starting out with the EpochX framework. It will outline the essential information needed to get started. A more in-depth Complete Guide will be available soon which will provide a full explanation of the components of EpochX and how they can be used to support your research.

1 See Genetic Programming by Koza for more information about the Genetic Programming algorithm.
2 CFG-GP is outlined by Whigham in his paper on Grammatically-based Genetic Programming
3 As discussed in various publications by O'Neill and Ryan.

Installation

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

EpochX contains no installer or executable, instead it provides a programmatic interface which you can harness from your own Java code. The download will contain the jar file *epochx-1.1.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 it is with most GP software.

Getting Started with Models

The key to using EpochX is in the use of models. A model is simply a Java class which defines how execution for a problem should proceed; it provides the run parameters, the fitness function and other configurable options. All models extend from the abstract Model class, which provides default values for many standard parameters. Typically a new model will actually extend from one of GPModel, GRModel or GEModel (for tree GP, CFG-GP and GE respectively), which provide further defaults for each of the representation's parameters. Models are provided pre-constructed for a number of standard problems for each representation. Overriding a model's default value for a parameter can be performed simply by calling the associated setter method.

Triggering execution of evolutionary runs with your model is simply a case of calling the model's run method. Where the SantaFeTrail class is one of the built in models:

```
new SantaFeTrail().run();
```

This single line is all that is required to start evolving solutions to the Santa Fe trail problem. However, that's not much good without some output and EpochX provides two powerful mechanisms to get your hands on all manner of outputs.

Obtaining statistics

Each model has a StatsManager which is retrievable with the model's getStatsManager() method. The components of the EpochX framework will put data and statistics into the executing model's StatsManager as runs progress. That data is then retrievable at any point from your code. Where possible the statistics are calculated lazily only when requested.

Each statistic and piece of data is associated with an event – a run, generation, crossover, mutation, etc. and an individual item of data can be requested with the getXxxStat(String) methods, where Xxx is the event and the parameter is the field to retrieve. The field string used must match the field string a statistic was inserted with. The fields of data that the framework provides are given as constants in the GPStatField, GRStatField and GEStatField classes for each of the representations. The comments for those constants indicate the instance type of the Object that is returned. Alternatively, a sequence of stats fields can be requested at once with the getXxxStats(String[]) which will return an array of the statistics retrieved. The StatsManager also provides a series of methods printXxxStats(String[]) that will print those statistics to the console.

```
model.getLifeCycleManager().printRunStats(…);
```

Having statistics available for retrieval about generations, crossovers etc. is only useful if we have the facility to obtain the statistics at each of these events. For this we have an event handling system.

Handling events

As well as a StatsManager, models have their own LifeCycleManager which can be obtained with their getLifeCycleManager() method. The LifeCycleManager maintains a set of listeners for life cycle events such as the start and end of runs, generations, crossovers etc. Listeners can be added to the LifeCycleManager to listen for these events. Typically this would be done with an anonymous implementation of one of the event adapters. Any action could be performed on occurrence of these events but in particular it is convenient to make use of the statistics. For example:

```
model.getLifeCycleManager().addGenerationListener(new
GenerationAdapter(){
```

```
            public void onGenerationEnd() {
                  model.getStatsManager().printGenerationStats(…);
    }
});
```

Example #1: Built-in Models

EpochX comes with a number of built-in models which define suitable setups and fitness functions for the common benchmark problems. To get started we will get one of these running, we'll use even-4 parity[4] using tree GP. You can find the even-4 parity model in the org.epochx.gp.models.parity package.

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

```
1. GPModel model = new Even4Parity();
2. model.getLifeCycleManager().addGenerationListener(){
3.     public void onGenerationEnd() {
4.         model.getStatsManager().printGenerationStats(GEN_NUMBER,
5.                                                      GEN_FITNESS_MIN,
6.                                                      GEN_FITTEST_PROGRAM);
7.     }
8. });
9. model.run();
```

Line 1 constructs the model, lines 2-8 defines what statistics we would like to print each generation and line 7 executes our model. It is that simple. We could have left out lines 2-8 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 lines 5 and 6 of our code. You could experiment with different stats fields available in the GPStatField class.

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 Model and GPModel classes, but now we want to provide our own. GPModel 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. GPModel model = new Even4Parity();
2. model.setPopulationSize(500);
3. model.setNoGenerations(100);
4. model.setMaxProgramDepth(8);
5. model.getLifeCycleManager().addGenerationListener(){
6.     public void onGenerationEnd() {
7.         model.getStatsManager().printGenerationStats(GEN_NUMBER,
8.                                                      GEN_FITNESS_MIN,
```

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

```
9.                                                  GEN_FITTEST_PROGRAM);
10.     }
11.});
12.model.run();
```

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 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. GPModel model = new Even4Parity();
2. model.setCrossover(new UniformPointCrossover(model));
3. model.setProgramSelector(new TournamentSelector(model, 7));
4. model.setRNG(new MersenneTwisterFast());
5. model.getLifeCycleManager().addGenerationListener(
                                        new GenerationAdapter(){
6.     public void onGenerationEnd() {
7.         model.getStatsManager().printGenerationStats(GEN_NUMBER,
8.                                                 GEN_FITNESS_MIN,
9.                                                GEN_FITTEST_PROGRAM);
10.    }
11.});
12.model.run();
```

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 setter methods, such as the new TournamentSelector we created which takes the tournament size as an argument.

See **Appendix A** for the full code listing of this example.

Example #2: Creating a New GP 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 GPModel. There are two main things that must be provided to implement our new model:

- An appropriate fitness function.
- The function set
- The terminal set.

Providing the fitness function is by implementing the abstract getFitness method inherited from the Model class. The getFitness method receives an instance of CandidateProgram as an argument and must determine a double fitness score for that program. EpochX uses standardised fitness so the fitness score that is returned should reflect that with a lower value signifying a better program. How the method calculates the score is up to you, the model designer, to decide. Typically the fitness would 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 tasks of providing the function and terminal sets are actually performed as one in EpochX, since functions and terminals are not distinguished. A terminal is merely a Node with an arity of zero, whereas a function is a Node with an arity of one or more. We refer to the combined set of the functions and terminals as the syntax. We can set the syntax that is to be used using the GPModel's setSyntax method which passes a List of instances of all the Nodes that the system can use in constructing program trees. We only want to do this once so inside the constructor is a good place to do this.

```
public Mux6() {
      // Construct the variables into fields.
      d3 = new BooleanVariable("d3");
      d2 = new BooleanVariable("d2");
      d1 = new BooleanVariable("d1");
      d0 = new BooleanVariable("d0");
      a1 = new BooleanVariable("a1");
      a0 = new BooleanVariable("a0");

      List<Node> syntax = new ArrayList<Node>();
      // Functions.
      syntax.add(new IfFunction());
      syntax.add(new AndFunction());
      syntax.add(new OrFunction());
      syntax.add(new NotFunction());
      // Terminals.
      syntax.add(d3);
      syntax.add(d2);
      syntax.add(d1);
      syntax.add(d0);
      syntax.add(a1);
      syntax.add(a0);

      setSyntax(syntax);
}
```

We need to keep a reference to all the variables in fields so we can change their values to each of our test

inputs during fitness evaluation. If we had a lot of inputs we might choose to use a HashMap or other collection to store them.

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(GPCandidateProgram program) {
      double score = 0;

       for (int i=0; i<inputs.length; i++) {
            // Set the variables.
            a0.setValue(inputs[i][0]);
            a1.setValue(inputs[i][1]);
            d0.setValue(inputs[i][2]);
            d1.setValue(inputs[i][3]);
            d2.setValue(inputs[i][4]);
            d3.setValue(inputs[i][5]);

            Boolean result = (Boolean) 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 BooleanVariable objects which we have held a reference to. 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 calling its run() method:

```
new Mux6().run();
```

See **Appendix B** for the full code listing of this example.

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 once it's completed for more detailed information.

For the latest version of EpochX and any guides and documentation take a look at www.epochx.org. A discussion group is also available on the website where you can ask questions and are likely to get a quick response.

Appendix A: Full Code for Example #1

```
import org.epochx.gp.model.GPModel;
import org.epochx.gp.model.parity.Even4Parity;
import org.epochx.gp.op.crossover.UniformPointCrossover;
import org.epochx.life.GenerationAdapter;
import org.epochx.op.selection.TournamentSelector;
import org.epochx.tools.random.MersenneTwisterFast;

import static org.epochx.gp.stats.GPStatField.*;

public class Example1 {

	public static void main(String[] args) {
		// Construct the model.
		final GPModel model = new Even4Parity();

		// Set parameters.
		model.setPopulationSize(500);
		model.setNoGenerations(100);
		model.setMaxProgramDepth(8);

		// Set operators and components.
		model.setCrossover(new UniformPointCrossover(model));
		model.setProgramSelector(new TournamentSelector(model, 7));
		model.setRNG(new MersenneTwisterFast());

		// Request statistics every generation.
		model.getLifeCycleManager().addGenerationListener(new GenerationAdapter(){
			@Override
			public void onGenerationEnd() {
				model.getStatsManager().printGenerationStats(GEN_NUMBER,

		GEN_FITNESS_MIN,

		GEN_FITTEST_PROGRAM);
			}
		});

		// Run the model.
		model.run();
	}
}
```

Appendix B: Full Code for Example #2

```
import java.util.*;

import org.epochx.gp.model.GPModel;
import org.epochx.gp.representation.*;
import org.epochx.gp.representation.bool.*;
import org.epochx.representation.CandidateProgram;
import org.epochx.tools.util.BoolUtils;


public class Mux6 extends GPModel {

	private BooleanVariable d3;
	private BooleanVariable d2;
	private BooleanVariable d1;
	private BooleanVariable d0;
	private BooleanVariable a1;
	private BooleanVariable a0;

	// The boolean inputs/outputs that we will test solutions against.
	private boolean[][] inputs;
	private boolean[] outputs;

	public Mux6() {
		// Construct the variables into fields.
		d3 = new BooleanVariable("d3");
		d2 = new BooleanVariable("d2");
		d1 = new BooleanVariable("d1");
		d0 = new BooleanVariable("d0");
		a1 = new BooleanVariable("a1");
		a0 = new BooleanVariable("a0");

		List<Node> syntax = new ArrayList<Node>();
		// Functions.
		syntax.add(new IfFunction());
		syntax.add(new AndFunction());
		syntax.add(new OrFunction());
		syntax.add(new NotFunction());
		// Terminals.
		syntax.add(d3);
		syntax.add(d2);
		syntax.add(d1);
		syntax.add(d0);
		syntax.add(a1);
		syntax.add(a0);

		setSyntax(syntax);

		// Generate set of test inputs and corresponding correct output.
		inputs = BoolUtils.generateBoolSequences(6);
		outputs = generateOutputs(inputs);
	}

	@Override
	public double getFitness(CandidateProgram p) {
		GPCandidateProgram program = (GPCandidateProgram) p;

		double score = 0;
		for (int i=0; i<inputs.length; i++) {
			// Set the variables.
		a0.setValue(inputs[i][0]);
		a1.setValue(inputs[i][1]);
		d0.setValue(inputs[i][2]);
		d1.setValue(inputs[i][3]);
		d2.setValue(inputs[i][4]);
		d3.setValue(inputs[i][5]);

		Boolean result = (Boolean) program.evaluate();
```

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

            return 64 - score;
        }

        /*
         * Generates the correct outputs for the 6-bit multiplexer from
         * the given inputs to test against.
         */
        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) {
            new Mux6().run();
        }

}
```