ECJ Tutorial - Making custom GUI charts

Robert Hovden

July 15, 2005

0.1 Intro

The ECJ GUI allows real-time visualization of genetic algorithms which appear in the statistics tab of the 'console'. This tutorial will show you how to impliment existing chart statistics along with creating and displaying custom chart types.

1 Make the parameters file

1.1 make a directory and basic parameter file

Go into the ec/app directory and create a new directory called tutorialgui1. In this directory we will need to create a "tutorialgui1.params". The majority of our parameter file will run our program in a manar similar to many of the problems and tutorials packaged with ECJ. We will not go into the details of this part as they are covered in other tutorial on the ECJ documentation web site. The source code to this tutorial should also be available for you to look at.

1.2 Add more statistics children

Here is where we add our charts. The statistics charts are just a statistics class, so they are added to the parameter file the same way (see tutorial 3). We will add statistics children and then specify their statistics class:

```
stat.num-children = 2

stat.child.0 = ec.display.chart.SimpleXYSeriesChartStatistics
stat.child.0.title = Best of Generation - Fitness
stat.child.0.x-axis-label = generation
stat.child.0.y-axis-label = fitness value


stat.child.1 = ec.app.tutorialgui1.SimpleBarChartStatistics
stat.child.1.title = Best of Generation - Gene Values
stat.child.1.x-axis-label = event number
stat.child.1.y-axis-label = start time
```

These specify the class for each specified child, its corresponding title, x-axis-label, and y-axis-label. The title and labels are nothing more than arbitrary strings that will be displayed on the chart.

1.3 optional parameter file organization

You may want to keep the chart statistics parameters seperate from the rest of your parameter file. Like described in previous tutorials, this can be done by specifying the parent parameter file. To better demonstrate this, you could create a second parameter file, lets call it tutorialgui1-second.params, and put your chartable statistics children in this new file. You must, however, also specify the parent parameter file. With this example, the parent parameter would be specified by adding the following to the top of tutorialgui1-second.params:

parent.0 = ../coevolve2/coevolve2.params

Why would you want to separate your parameter files? Not only is this good organization, but it also separates the GUI console dependent parameters from the standard ECJ parameters. With the two separated, you can be sure to safely evolve your problem with or without the GUI console.

2 Define the problem

The problem defined for this tutorial will be simple event scheduling optimization. Although this particular scheduling problem can be solved quite easily, more complex scheduling problems are extremely difficult to optimize (NP-Hard). It is also a tutorial, so we'll keep it simple. This problem will involve a given number of tasks that should be scheduled sequentially, one following right after the other. Each event only knows that it should come after the one before it. We will keep the event start times as integers and the duration of each event will be one. So in effect, event one should be followed by event two, which should followed by event three, etc. I like to call this a staircase scheduling problem.

2.1 Create a new class

Make a new file called "staircase.java". In this file add:

```
package ec.app.missionplan;

import ec.*;
import ec.simple.*;
import ec.vector.*;

public class SimpleStaircase extends Problem implements SimpleProblemForm
    {
    public void evaluate(final EvolutionState state,
                         final Individual ind,
                         final int threadnum)
        {
        if (ind.evaluated) return;

        if (!(ind instanceof IntegerVectorIndividual))
            state.output.fatal("Whoa! It's not a IntegerVectorIndividual!",null);

        IntegerVectorIndividual ind2 = (IntegerVectorIndividual)ind;
```

Now we are going to go through each gene (an integer) in the genome and determine if it's value is greater than the one before it. If this is true then we will say a constraint is met and add 1 to "constraintsMet", otherwise we add 1 to "constraintsBroke". After looping through the genome, it will have determined the total constraints met and broke.

```
        //---START FITNESS EVALUATION---//
        int constraintsMet = 0;
        int constraintsBroke = 0;

        for(int x=1; x<ind2.genome.length; x++){
         if (ind2.genome[x] > (ind2.genome[x-1])) constraintsMet++;
         else constraintsBroke++;
        }
```

We can now evaluate the fitness. For simplicity, it is just the negative value of the total constraints broke. The fitness will be maximized by the Genetic Algorithm, therefor the total broken constraints should be minimized.

```
        if (!(ind2.fitness instanceof SimpleFitness))
            state.output.fatal("Whoa!  It's not a SimpleFitness!!!",null);
        ((SimpleFitness)ind2.fitness).setFitness(state,
                                   // normalize the fitness for genome length
                                   // so it's within float range
                                   (float)((double)(-constraintsBroke)),
                                   ///... is the individual ideal?  Indicate here...
                                   false);
        ind2.evaluated = true;
        }
     //---END FITNESS EVALUATION---//

    public void describe(final Individual ind,
                         final EvolutionState state,
                         final int threadnum, final int log,
                         final int verbosity)
        {
        }
    }
```

2.2 Looking further at the parameter file

Now that the problem has been created, you can make sense of what the genome represents. In order to make the problem reasonable, lets make the genome size 20, and the min max gene 0 and 20. This will correspond to the following in your parameter file:

pop.subpop.0.species.genome-size = 20
pop.subpop.0.species.min-gene = 0
pop.subpop.0.species.max-gene = 20

3 Make the Charts

In the parameter file we have declared our chartable statistics, now we must create the corresponding statistics classes. To refresh your memory, we declared the classes in our parameter file by stating the following:

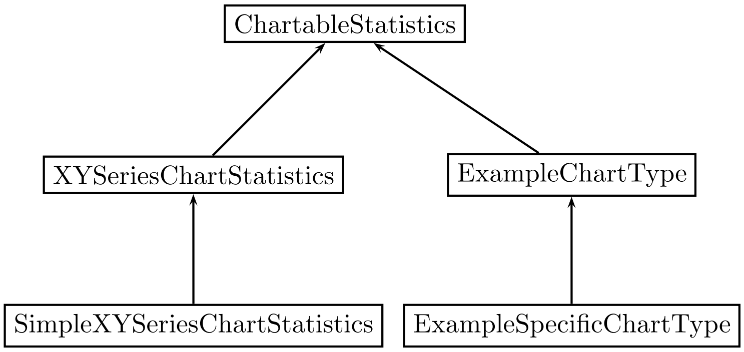
stat.child.0 = ec.display.chart.SimpleXYSeriesChartStatistics
stat.child.1 = ec.display.chart.SimpleBarChartStatistics

3.1 The class structure of ec.display.chart

The ChartableStatistics class is located in the ec.display.chart package. As you might expect, ChartableStatistics is a subclass of Statistics. ChartableStatistics is the class that the various chart types will extend.

When the console runs the Genetic Algorithm, it will call ec.display.StatisticsChartPane.createCharts() which will check to see if each of the statistics objects are instances of ChartableStatistics. If the statistics object you defined is a ChartableStatistic, it will set up a nice chart panel and pane tab for your chart, make a ChartableStatistics object called chartStats, and then call chartStats.makeChart().

Chartable statistics is an abstract class, and simply needs to be extended so we can start making our own chart classes. Fortunately, ECJ comes packaged with a ChartableStatistics subclass: XYSeriesChartStatistics. It is subclassed by SimpleXYSeriesChartStatistics which gives us our fitness value graph. Such a graph is very general and therefore is packaged into ec.display.chart.



3.2 Create a chart type class

We are going to make a new chart type, so we will extend ChartableStatistics with our new BarChartStatistics. Make the new BarChartStatistics.java. Now add:

```
package ec.app.tutorialgui1;

import org.jfree.chart.ChartFactory;
import org.jfree.chart.JFreeChart;
import org.jfree.chart.plot.PlotOrientation;
import org.jfree.data.category.DefaultCategoryDataset;
```

```
import ec.EvolutionState;
import ec.util.Parameter;

public class BarChartStatistics
    extends ChartableStatistics {

    public DefaultCategoryDataset dataset;
```

In the setup we must make a dataset for the Bar Chart.

```
    public void setup(EvolutionState state, Parameter base) {
        super.setup(state, base);
        dataset = new DefaultCategoryDataset();

    }
```

Here we will create makeChart(), telling it to create a bar chart and use the dataset.

```
    public JFreeChart makeChart(){
     JFreeChart chart = ChartFactory.createBarChart(this.title,
      this.xlabel,this.ylabel, this.dataset, PlotOrientation.VERTICAL,
      true, true, false);

        return chart;
    }
```

A bar chart will be made (or refreshed) by a method which takes in a seriesID plus an array of values (one for each bar) and then stores them in the dataset. When data is changed or added to the dataset, JFreeChart will redraw the chart. We will allow makeBar to take in both double[] and int[].

```
    public void makeBar(int seriesID, double[] genes){
     for (int i = 0; i < genes.length; i++){
         dataset.setValue(genes[i], "Genome "+seriesID, String.valueOf(i));
     }
    }
    public void makeBar(int seriesID, int[] genes){
     for (int i = 0; i < genes.length; i++){
         dataset.setValue(genes[i], "Genome "+seriesID, String.valueOf(i));
     }
    }

    }
```

3.3 Create a specific chart class

Now we must make a subclass of BarChartStatistics which we will call SimpleBarChartStatistics. This is our specific chart class because it will tell what the bar chart is going to represent. In this case it will represent the genome of the current best individual of the current generation. Make SimpleBarChartStatistics.java and add:

```
package ec.app.tutorialgui1;
```

```
import ec.EvolutionState;
import ec.Fitness;
import ec.util.Parameter;
import ec.vector.*;
```

We need to make a seriesID array to hold an ID for each subpop. We will get the number of subpop out of the parameter file and create the seriesID array;

```
public class SimpleBarChartStatistics
    extends BarChartStatistics {

    private int[] seriesID;

    public void setup(EvolutionState state, Parameter base) {
        super.setup(state, base);

        int numSubPops =
state.parameters.getInt(new Parameter("pop.subpops"));

        seriesID = new int[numSubPops];

        for (int i = 0; i < numSubPops; ++i) {
            seriesID[i] = i; //series id identifies each sub pop
            }
    }
```

After the evaluation of the generation we need to loop through every individual and determine if that individual is the best so far. If it is then we will chart its genome using makeBar().

```
    public void postEvaluationStatistics(EvolutionState state) {

    super.postEvaluationStatistics(state);

    for (int subPop=0; subPop < state.population.subpops.length; ++subPop) {
        Fitness bestFit=state.population.subpops[subPop].individuals[0].fitness;
        for (int i=1; i < state.population.subpops[subPop].individuals.length; ++i) {
            Fitness fit=state.population.subpops[subPop].individuals[i].fitness;
            if (fit.betterThan(bestFit)){
                bestFit = fit;

                //Best individual is found, make a bar graph
               makeBar(seriesID[subPop], ((IntegerVectorIndividual)
     state.population.subpops[subPop].individuals[i]).genome);
                }
            }
        }
    }

    }
```

4 Run the program

Now the fun part, running the program. With the addition of real time charts, it will be much more enjoyable to watch the evolution.

4.1 Start up ECJ's console

Fire up the console by running ec.display.Console. To do this at the prompt you can type:

java ec.display.Console

Now the console should appear beautifully on the screen, giving you a plethera of things to click, manipulate, and change. You should also see multiple tabs and a file menu.

4.2 Load the parameter file

Now you need to load the parameter file into the console. Do this by first locating the file menu at the top. Simply go to:

-File
-Load Parameters . . .
-Now browse for the parameter file you wish to load (for this tutorial, it is tutorialgui1.params)

With the parameter file loaded, you can browse the parameter file settings throughout the console. Many of these settings can also be changed using the console.

4.3 Press Play

Here is the best part: press play. The genetic algorithm should run immediately displaying the current generation at the bottom. You can also view the charts in the statistics tab. There should be two charts, one for each type we specified in our parameter file. The first chart will display the best fitness of each generation. If there are multiple subpopulations, uniquely colored lines will represent each subpop. The second chart will show a bar chart of the genome of the current best individual of the current generation. If there are multiple subpopulations, uniquely colored bars will represent each subpop. You can now watch how the genome evolves graphically.

5 The staircase problem

The staircase problem is should be challenging for such a generalized genetic algorithm to solve. It is very easy for it to become stuck in a local minimum, especially as the number of events grows. You will probably notice that the solution settles on multiple smaller staircases, failing to satisfy every constraint. Different parameters and methods will get different results, have fun with it.