HELLO WORLD

Hands-On Introduction to Genetic Programming by Dmitry Batenkov

The idea to mimic the principles of Darwinian evolution in computing has been around at least since the 1950s, so long, in fact, that it has grown into the field called *evolutionary computing* (EC). In this tutorial, we'll learn the basic principles of EC and its offspring, genetic programming (GP), on a "toy problem" of *symbolic regression*. We'll also learn how to use **OpenBeagle**, a generic C++ object-oriented EC framework.

The Fittest Program Survives

EC can be regarded as a very general kind of optimization, where the solution to a given problem is selected from an evolving population of candidate solutions, or *individuals*, represented by their *genomes*. The selection is based on certain *fitness criteria*, which can just be a function operating on genomes.

The computation starts by choosing a random bunch of individuals—generation zero. Generation *n*+1 is the result of applying *evolution operators* to the individuals of generation *n*. The most used operators are *mutation* (random modification of a single individual's genome) and *crossover* (random mixing of genomes of two individuals). The individuals that produce "offspring" are chosen based on their fitness. The process ends when a certain stopping criteria are met (for example, some predefined number of generations).

GP takes these ideas one step further by performing the search in the space of *programs* (*algorithms*). A program's genome is usually represented as a tree of *primitives*, such as variables, arithmetical and logical operators, loops, conditionals, function calls, and so forth.

The very nature of EC and GP enables one to tackle problems without having the slightest idea how the solution should look. Indeed, this paradigm has been successfully applied in a broad range of applications, producing results that have even been patented as new inventions. On the other hand, success is never guaranteed.

Example

Let's demonstrate the principles outlined above on the classical "toy example" of symbolic regression. Among the several genetic programming tools available, I have chosen to use the OpenBeagle framework. Written in standard C++, OpenBeagle supports virtually any kind of EC through subclassing and polymorphism. Combined with portability and speed of C++, this approach is a good choice for many projects. It's also beautifully designed and a real pleasure to use. There are detailed instructions for downloading and installing the package at http://beagle.gel.ulaval.ca. The example below was largely taken from OpenBeagle documentation.

Let's say we are given some one-dimensional data samples $\{(x_i, y_i)\}_{i=1}^{N}$ and we would like to find a formula $y=f(x)$ which best fits the data. Suppose we decide to constrain the formula to consist only of arithmetic operations: addition, subtraction, multiplication, division. Then our genome trees will consist of these four primitives as intermediate nodes, together with the leaf node "x" (the function's argument). In OpenBeagle, we need to define an object of the type `GP::PrimitiveSet`. See **Listing 1a**.

The `GP::System` object is a kind of container holding the information about the evolutionary system. It contains objects of type `Beagle::Component`. (Note that all the framework objects have reference counting, and so the references are in fact smart pointers.) We define a component to hold our initial data, which will be used later on. See **Listing1b**.

Listing 1: (a) First, we define the primitive set in OpenBeagle. (b) Next, we define a component to hold the initial data.

a

```
1  GP::System::Handle create_system() {
2    GP::PrimitiveSet::Handle lSet = new GP::PrimitiveSet;
3    lSet->insert(new GP::Add);
4    lSet->insert(new GP::Subtract);
5    lSet->insert(new GP::Multiply);
6    lSet->insert(new GP::Divide);
7    lSet->insert(new GP::TokenT<Double>("x"));
8    return new GP::System(lSet);
9  }
```

b

```
1  class InitialData : public Component {
2  public:
3    std::vector<Double> X, Y;
4
5    InitialData (unsigned int npoints) : Component("InitialData")
6    {
7      srand((unsigned)time(0));
8      for(unsigned int i=0; i<npoints; i++)
9      {
10       X.push_back(-1.0+2.0*(double)rand()/RAND_MAX);
11       Y.push_back(-1.0-sin(2.0*X[i].getWrappedValue()));
12     }
13   }
14 };
```

Listing 2: Sample code is provided for (a) the evaluation operator and (b) the main program in OpenBeagle.

a

```
1  class SymbRegEvalOp : public GP::EvaluationOp
2  {
3  public:
4    SymbRegEvalOp() { }
5    virtual Fitness::Handle evaluate(
6      GP::Individual& inIndividual, GP::Context& ioContext)
7    {
8      InitialData &id = dynamic_cast<InitialData&>(
9        *(ioContext.getSystem().getComponent("InitialData")));
10     std::vector<Double> X = id.X;
11     std::vector<Double> Y = id.Y;
12     double lQErr = 0.; // square error
13     for(unsigned int i=0; i<X.size(); i++)
14     {
15       setValue("x",X[i],ioContext);
16       Double lResult;
17       inIndividual.run(lResult,ioContext);
18       double lError = Y[i]-lResult;
19       lQErr += (lError*lError);
20     }
21     return new FitnessSimple(1.0/
22       (std::sqrt(lQErr/X.size())+1.0));
23   }
24 };
```

b

```
1  #include <cmath>
2  #include <vector>
3  #include <beagle/GP.hpp>
4
5  using namespace Beagle;
6
7  int main(int argc, char *argv[]) {
8    GP::System::Handle lSystem = create_system();
9    InitialData::Handle id = new InitialData(20);
10   lSystem->addComponent(id);
11
12   SymbRegEvalOp::Handle lEvalOp = new SymbRegEvalOp;
13
14   GP::Evolver::Handle lEvolver = new GP::Evolver(lEvalOp);
15   GP::Vivarium::Handle lVivarium = new GP::Vivarium;
16
17   lEvolver->initialize(lSystem,argc,argv);
18   lEvolver->evolve(lVivarium);
19   return 0;
20 }
```

I have used the formula $y_i = -1 - \sin(2x_i)$ for this example. So in fact we will be approximating a trigonometric function by a rational one.

The next step is to define the fitness function which measures how close an individual *I* is to a perfect match. In our case, a good choice will be to calculate the deviation

$$D(I) = \sum_{i=1}^{N} (y_i - I(x_i))^2$$

and then set the fitness to be the normalized value [the algorithm maximizes the fitness, so a perfect match would have the maximal fitness]

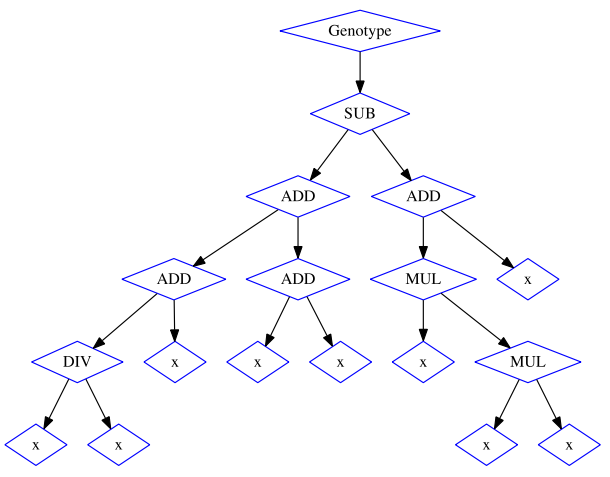
$$F(I) = \frac{1}{1 + \sqrt{\frac{D(I)}{N}}}$$

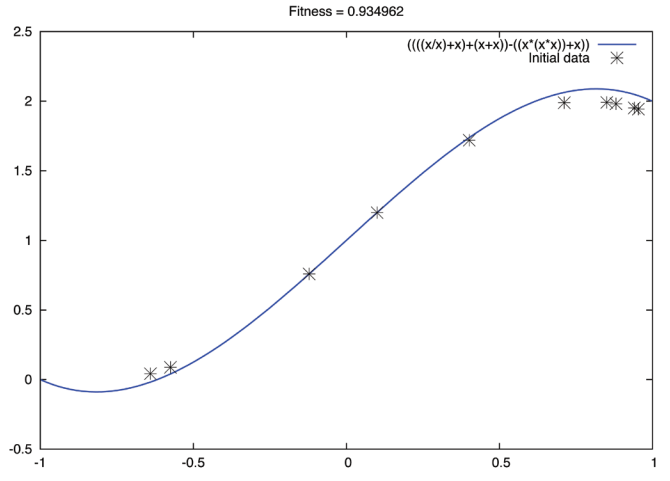
In OpenBeagle, the fitness calculation is encapsulated by objects of type `GP::EvaluationOp`. Ours would be coded as shown in **Listing 2a**.

Having defined these essential components of a GP system, now we only need to combine everything. There are two additional objects we need to be familiar with. The first is `GP::Vivarium`, which encapsulates all the individuals of all the generations throughout the whole evolution process, as well as statistical data. For example, it has a member of type `Beagle::HallOfFame` that holds the best individual. Finally, the entire process is controlled by a `GP::Evolver`.

{CONTINUED ON P.51}

Figure 1: The best individual is shown, including: (a) its complete genotype and (b) how well it approximates the initial data.





{CONTINUED FROM P.47}

It is responsible for selecting the initial population and applying the evolution operators, evaluating the individuals at each generation, until a termination criteria is met. The default implementation contains a pre-defined set of mutation and crossover operators. The main() function is shown in **Listing 2b**.

Compile and run the program. Two XML files will be produced: `beagle.log` and `beagle.obm.gz`. Exploring these files is highly recommended, as it provides many insights into the architecture and the inner workings of the framework.

Everything in OpenBeagle can be customized and extended. With almost no effort, I have added additional primitives to the system and extended the `InitialData` component to write the X and Y arrays to the log. Then I used this information to visually explore the best individual of a run, as depicted in **Figures 1a** and **1b**.

All the source code used in this example, along with instructions, can be downloaded from http://xrds.acm.org/code.cfm. Again, there's much more to OpenBeagle than presented here, so I encourage you to investigate!

RESOURCES & FURTHER READING

Wikipedia
Evolutionary computing entry
Genetic programming entry

ACM Special Interest Group on Genetic and Evolutionary Computation
www.sigevo.org

ACM SIG EVO Newsletter
www.sigevolution.org

A Field Guide to Genetic Programming
http://dces.essex.ac.uk/staff/rpoli/gp-field-guide

Professor John Koza:
a pioneer of modern GP
www.genetic-programming.com/johnkoza.html

Essentials of Metaheuristics:
free book
http://cs.gmu.edu/~sean/book/metaheuristics

Genetic-Programming.org
www.genetic-programming.org

FRAMEWORKS

PyEvolve
http://pyevolve.sourceforge.net

ECJ (Java)
http://cs.gmu.edu/~eclab/projects/ecj/

Gaul
http://gaul.sourceforge.net