

Greg Paperin

<http://www.paperin.org>

Alex Michalas

[alex@michalas.org](mailto:alex@michalas.org)

<http://www.michalas.org>

Department of Computer Science  
University College London  
Gower Street  
London WC1E 6BT

## Abstract.

In this work we demonstrate some basic and more advanced approaches to genetic programming (GP) [1] on the example of a multiplexer.

We briefly discuss the topic of GP and how it can be applied to evolving boolean-valued functions, such as the multiplexer. We then consider various approaches to implementing GP and other genetic algorithms (GA) and briefly describe JAGA, a generic application programming interface (API) for Java [6] which we have developed for implementing various GAs [2]. In particular, we look at the implementation of the function tree approach to GP and describe our own implementation of a system for evolving a 6- and an 11-multiplexer. We discuss various algorithms and settings we tried on this problem and the way they affected the quality of the solution in terms of the percentage of problem instances solved by the emerging formulas, the size of the emerging formulas and the computational resources required by the GP system. Finally, we briefly outline unresolved questions and directions in which this work could be taken when more time is available.

## Acknowledgements.

This work was undertaken in February 2004 in the scope of a course in Evolutionary Computation, which is part of an Intelligent Systems MSc degree at the Department of Computer Science at University College London.

We thank Mark Herbster, the course lecturer, for the inspiration and hints that took this work into the right direction.

We also thank Cameron Angus, a research assistant in the Department, for his valuable personal and technical support during the development of the Java API for GA.

## Overview of Genetic Programming.

GP is a form of GA that automatically evolves computer programs [1]. GP typically starts with a random population of individuals (computer programs) and uses evolution, crossover, mutation and other biologically-inspired ideas to evolve a program that performs a desired task.

We usually tend to think of a computer program as a sequence of commands. If we wanted to develop this notion of a program through GP we would need to encode it in such a way that

standard GA operations can be performed on it. Such a representation could be a stack representation as proposed in [3].

Alternatively, a program can be abstracted as a function taking some input and returning some output and, in this work, we investigate the function notion of a computer program and focus in GP applied to evolving functions.

Consequently, we need a way to encode functions as individuals that will allow us to perform GA operations on them. Functions can be represented

by trees, so we considered using an approach of encoding trees in a chromosome like-manner like in [2].

### **The Java Genetic Algorithms (JAGA) Package.**

There is a large number of packages available which support implementation of GAs. Most of them are intended for use with scientific computation software (for example GAOT for Matlab [4]) or with C and C++ programming languages.

Scientific computation software has usually the advantage of providing a simple programming language, which does not require a lot of programming skills and is for scientists to learn and use. Such software also offers a wide range of mathematical and statistical functions, often not present in general purpose programming languages.

Unfortunately, this kind of software is not suited for implementation of general purpose programs. It often has drawbacks in run-time performance and does not provide for flexibility and scalability of software. The nature of GAs makes them suitable for parallel computations, but it is often impossible to optimise a program for distributed computation when using scientific computation software.

The C++ (or C) programming language evades these disadvantages. It allows for the writing of highly efficient, flexible and scalable software. A number of powerful packages for mathematical and statistical functions exist for C and various methods for implementing parallel distributed programs are well established (e.g. CORBA [5]). Unfortunately, C++ is difficult to learn and requires advanced programming skills to really take advantage of its powerful features, such as those mentioned above.

The Java programming language [6] combines the advantages of advanced programming languages and easy-to-learn scientific programming languages.

It is a widely used, modern object oriented programming language with all resulting advantages. It has built-in support for parallel computation on a multi-processor computer and in a distributed environment. A large number of open-source libraries for mathematical

computation exist for Java; and Java programs are at least as widely portable as those for scientific computation systems. At the same time, Java is much easier to learn and use than C++.

Various packages which support the development of GA and GP systems exist for Java (e.g. JGAP [7], Groovy Java Genetic Programming [8], GA Playground [9]). However, all these packages (and other freely available, similar packages) either concentrate on a specific aspect of GA or GP or provide only a loose framework without an efficient underlying implementation.

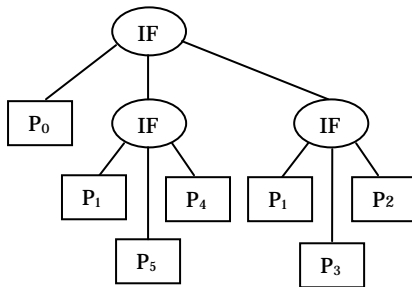
One of the authors of this work has many years of experience in software development and software systems engineering. We wanted to apply this expertise to develop a state-of-the-art API for developing GAs and GP applications in Java. The work on this package, called JAGA (Java API for Genetic Algorithms) is currently in progress, but this paper is based on the version 0.9 Beta of it. JAGA will soon appear under the GNU licence and will be available for download (we do not know the URL yet, but there will be a link from [10]); all code quoted in this paper will become part of JAGA.

## Evolving a Multiplexer.

### Representation.

It was attempted by various researchers to use GP to evolve a program which emulates a multiplexer. [1][11]

Following the approach of [2] we represent our solutions as a boolean function  $mux(p_0, \dots, p_n)$ . Such a function consists of boolean operators, e.g. AND, NOT, OR, IF and others, and of terminals  $p_0, \dots, p_n$  as well as the constants `true` and `false`. A function can be represented as a tree like this one: (here, using only the IF operator)



**Fig. 1:** Tree representation of a 6-Multiplexer function using only the tertiary IF operator.

Traditionally in GP, function trees like this are represented as strings like this [2]:

```
IFp0IFp1P5P4IFp1P3P4
```

This is done partly to save computer memory, partly to avoid processing of graphs. We feel, however, that memory is a cheap resource these days; and the JAGA package allows us to encapsulate the graph structures in such a way that they do not interfere with the development of the algorithms working on the formula representation. By processing graphs directly we can evaluate a particular formula much faster; we effectively trade memory for speed which allows us to run larger experiments using the available equipment (all results presented in this paper were achieved on a PC with an Athlon 1700+ processor and 512 Mb RAM, running Java 1.4 under Windows XP).

## Genetic Operators

We use two types of genetic operators on the tree graphs: mutation and crossover.

Mutation is a unary genetic operator. A random node of a formula-tree is selected. With a probability  $P_{mut}$ , this node (and its sub-tree) is replaced by a randomly generated sub-tree, otherwise the formula is not changed.

Crossover is a binary genetic operator on two “parents”  $X$  and  $Y$ . It is conducted with a probability  $P_{XOver}$ . If a “dice” falls outside  $P_{XOver}$ , the parents are simply copied to construct the “children”. Otherwise, the children are constructed as follows: For each parent, a random node  $X_{Cross}$  and  $Y_{Cross}$  is selected. Then,  $X_{Cross}$  and its sub-tree are removed from  $X$  and  $Y_{Cross}$  is inserted in its place together with the corresponding sub-tree. Similarly,  $Y_{Cross}$  is removed from  $Y$  and replaced by  $X_{Cross}$  and its sub-tree. The two resulting trees become the “children”.

We have found it useful to restrict the maximum depth of the trees (i.e. the maximum number of edges traversed between the root and any of the terminals). This prevents the trees growing indefinitely; such growing cannot be considered productive [11].

To prevent the trees exceeding the maximum depth, the genetic operators are modified as follows:

- Random tree creation: When constructing a graph, a random child node (operator or terminal) is attached to any node with a depth  $d$ , where  $0 \leq d < maxDepth - 1$ ; when  $d = maxDepth - 1$ , only a random terminal node can be attached.
- Mutation: After we selected a node with the depth  $d$  to be mutated, we set the maximum depth for the new sub-tree to be generated to  $maxDepth - d$ . This way the result cannot exceed  $maxDepth$ .
- Crossover: After the nodes for the crossover are selected, the depth of both children which would result from the crossover is calculated. If both children do not exceed  $maxDepth$ , the crossover is executed. Otherwise, two new nodes are randomly selected. This process is repeated  $maxAttempts$  times, where  $maxAttempts$  is a variable parameter. In all experiments described

in this paper, *maxAttempts* was constantly set to 15.

## Fitness

The evaluation of the fitness is crucial for successful GP.

When evaluating the fitness of a multiplexer, the number of inputs evaluated correctly (out of all possible inputs) is a good measure. In order to encourage the evolution of shorter formulas, some negative function of the number of nodes in the formula-tree can be incorporated.

In this work we used the following function:

$$\text{fitness}(X) = \text{round}(\text{correct}(X) - w \cdot \log_2(\text{size}(X))) + \text{correct}(X) / \text{inputSize}$$

, where:

- *X* is an individual;
- *round(k)* rounds *k* to the closest integer;
- *correct(X)* is the number of inputs correctly evaluated by *X*;
- *size(X)* is the number of nodes in *X*;
- *inputSize* is the total number of possible inputs to *X* (i.e.  $2^{(\text{num. of free parameters to } X)}$ );
- *w* is a variable parameter specifying how important the role of the formula-size shall be, for  $w = 0$  the size of the formula does not matter.

In all experiments described in this work, the multiplexers were always evaluated against all possible inputs.

## Conduct of Experiments.

### 3-Multiplexer

First, we tried to evolve the simplest type of multiplexer, the 3-multiplexer.

We used the standard GA-algorithm:

```
Create initial random population P;
```

```
While (termination condition does not apply) {  
    N = new empty population;  
    While (N.size < P.size) {  
        parents = selectionFunction(P);  
        children = XOver(parents);  
        children = mutation(children);  
    }  
    P = N;  
}
```

As termination condition, we used either the best individual in *P* reaching the maximum possible fitness or a certain number of passed generations (variable).

As selection function, we first used the standard roulette wheel selection. [2]

The parameters to the algorithms were:

```
Population size = 100;  
Shorter formulas weight  $w = 0$ ;  
 $P_{XOver} = 0.65$ ;  
 $P_{Mutation} = 0/1$ ;  
 $maxDepth = 8$ .
```

We allowed AND-, OR- and NOT-nodes, 3 types of terminals ( $p_0, p_1, p_2$ ) and no constants.

As expected, most of the GP-runs yielded correct solutions within 50 generations.

While many of evolved solutions were optimally short (e.g.  $((p_2 \text{ AND } p_0) \text{ OR } (p_1 \text{ AND } (p_2 \text{ OR } !p_0)))$ ) or near that, about half of the results were very long.

Examples of typical results are:

- $((p_2 \text{ OR } !p_0) \text{ AND } ((!p_0 \text{ OR } (p_0 \text{ OR } p_2)) \text{ AND } (p_1 \text{ OR } p_0)))$
- $((p_2 \text{ OR } !p_0) \text{ AND } ((p_0 \text{ AND } ((!p_1 \text{ AND } p_0) \text{ OR } p_2) \text{ AND } p_2)) \text{ OR } p_1))$

Setting *shorter formulas weight*  $w = 0.3$  resulted in the majority of the evolved formulas becoming fairly short (20 or less nodes including the terminals), but a 100% correct solution was evolved within the first 50 generations only in 60% of runs.

We had very interesting results when only allowing IF-nodes for operators (with  $w = 0$ ):

A 100% correct solution was always found within 20 generations.

However, note that a single IF node is exactly a 3-multiplexer. Nevertheless, about half of the evolved formulas looked rather like this:

- IF (IF (IF (p0) THEN (p0) ELSE (p0)) THEN (p0) ELSE (IF (IF (p2) THEN (p1) ELSE (p0)) THEN (IF (p1) THEN (p2) ELSE (p1)) ELSE (IF (p2) THEN (p1) ELSE (p0)))) THEN (IF (p1) THEN (p2) ELSE (IF (p1) THEN (p2) ELSE (IF (p2) THEN (p2) ELSE (IF (p1) THEN (p1) ELSE (p1)))))) ELSE (IF (p2) THEN (IF (p0) THEN (p1) ELSE (p0)) ELSE (IF (p1) THEN (IF (p2) THEN (p2) ELSE (p1)) ELSE (IF (p1) THEN (p2) ELSE (p1))))))

The other half of the results were reasonably short and the best possible solution

- IF (p0) THEN (p2) ELSE (p1)
- was evolved in about 10% of cases.

Any boolean formula can be constructed from only NAND-nodes. An attempt to do so yielded quite surprising results: with *shorter formulas weight*  $w = 0$  the results contained far less nodes than for other node-type settings. Some of the results are:

- (((p1 NAND p0) NAND p1) NAND (p0 NAND p2))
- (((p2 NAND p0) NAND p2) NAND p2) NAND ((p0 NAND p0) NAND p1))
- ((p0 NAND p2) NAND (p1 NAND (p1 NAND p0)))

All of the NAND-solutions were evolved within 30 generations.

## 6-Multiplexer.

After such encouraging results, we attempted to evolve a 6-multiplexer.

Before describing the results, we should make an important remark about the fitness. A formula containing a single terminal node correctly solves 62.5% of cases, if that terminal corresponds to a data-line of the multiplexer. So, although presumably an average candidate should solve 50% of test cases, we have to consider any rate below 62.5% as “worse than trivial”.

Our initial setup for the 6-multiplexer was essentially as before, except that we had increased the population size to 1000.

Trying various node-type configurations, operator-probabilities and other parameter values did not yield any good results within 150 generations. Increasing the pressure for selection of shorter formulas results in trees with clearly less nodes but does not improve the rate of correct solutions.

To improve on the situation, we attempted a different selection function, the probabilistic tournament selection.

Our version of probabilistic tournament selection takes two parameters,  $S$  and  $P$  and works as follows:

1. Select  $S$  random individuals from the population and sort them in descending order according to their fitness in an array with indices  $1\dots S$ ;
2. Set  $i = 1$ ;
3. With probability  $P$ , return the individual at position  $i$ ;
4. If no individual was returned, set  $i = i - 1$ ; if  $i > S$ , set  $i = 0$ ;
5. Go to step 3.
6. Carry on until an individual is returned.

We tried various settings for  $S$  and  $P$ . As expected, small values do not provide enough selection pressure. A setting of  $S = 5$  and  $P = 0.9$  yields good results in most of the cases.

This selection method works better and faster for the multiplexer problem.

For the 6-multiplexer a 50 generations run with a population size of 500 is completed within ca. 20 seconds.

The reason for that is that the best individuals are more likely to be chosen and, therefore, the best individual in each population only rarely has a worse fitness than the best individual over all preceding populations.

Using 5-0.9-tournament-selection we achieved good (over 90%) solutions within 20 generations and 100% solutions within 50 generations in  $\frac{3}{4}$  of runs (other parameters as be).

However, the results were rather long, e.g.:

- (((p3 AND (p1 AND !p0)) OR ((p0 AND p0) AND (((!p3 OR p2) OR !p2) AND (p5 AND p1)))) OR (((p0 OR !!(p1 OR p2) OR ((p5 AND !!(p5 AND p0) AND !p0)) AND p4))) OR p1) AND !(p1 OR ((p1 AND p3) OR !p4) AND (!!p0 AND (((p5 AND !!(p1 OR p2) OR ((p5 AND !!(p3 OR p3) AND p0) AND !p0)) AND p4))) OR !(((p5 AND !!(p1 OR p3) OR ((p5 AND !!(p4 AND p0) AND !p0)) AND p4))) OR !!(p1 OR p0) OR ((p5 AND !!(p5 AND p0) AND !p0)) AND p4))) OR p1) OR p2) OR ((p5 AND !!(p5 AND p0) AND !p0)) AND p4))) OR p1) OR (p0 AND p4))))))

Most of results were even longer.

We have tried various approaches to get shorter results:

First, we tried setting *shorter formulas*  $w = 1$ . With this setting, in only a small proportion of runs a 100% solution evolved within 50 generations, most solutions solved 90%-95% percent of test cases correctly. However, 50%-60% of runs evolved a 100%-solution within 100 generations.

The second approach to evolve shorter formulas was to limit the maximum tree depth to 4. This did not help either – the trees became shallower, not more complete and the overall number of nodes per tree did not decrease.

Finally, we experimented with different node-type configurations. Allowing only IF-nodes yielded the best results: 9/10 of runs evolved a 100% solution with 25 generations.

The best possible solution evolved as early as generation 7 in a few runs:

- IF (p0) THEN (IF (p1) THEN (p5) ELSE (p4)) ELSE (IF (p1) THEN (p3) ELSE (p2))

Pure-NAND trees again provided some interesting results:

We tried allowing constant nodes (true/false) as a type of a terminal node. With this setting, the average time to evolve a 100% NAND-solution increased in comparison to runs without constants (ca 70 generations / ca 40 generations), but the length of the evolved formulas dramatically decreased. One of the solutions was:

- (((((p1 NAND p1) NAND ((p0 NAND (p0 NAND p4)) NAND p2)) NAND (true NAND ((p1 NAND ((p1 NAND (p0 NAND p0)) NAND true)) NAND (p1 NAND ((p5 NAND (true NAND p0)) NAND p5)))))) NAND ((p1 NAND p0) NAND ((p1 NAND p3) NAND (p4 NAND p0))))

in generation 66.

Finally, we tried using a larger number of possible node types:

- AND, OR, NOT, NAND, NOR, XOR, IF, EQUIV, IMPL.

This yields solutions similar to those evolved using only AND-, OR-, and NOT-nodes, but in a larger number of generations (70-130).

## Analysis of results and improvement strategy

In order to further improve the situation, we used a different GP-algorithm – Elitist algorithm with low edge cut-off.

The algorithm has two parameters – the elite proportion  $E$  and the cut-off proportion  $C$ . It is required that  $E, C > 0$  and  $E + C < 1$ . The algorithm works as follows:

```
Create initial random population P;
While ( !termination condition) {
    Sort P in ascending order of individual's
    fitnesses;

    P' = new empty population;
    cutSize = P.size * (1 - C);
    copy the first cutSize individuals from P to
    P';

    N = new empty population;
    eliteSize = P.size * E;
    copy the first cutSize individuals from P'
    to N;

    While (N.size < P.size) {
        parents = selectionFunction(P');
        children = XOver(parents);
        children = mutation(children);
    }
    P = N;
```

}

For  $E > 0$  this algorithm guaranties that the best individuals will never die; for  $C > 0$  it guaranties that the worst individuals will never survive.

We have investigated how the fitnesses of each generation evolve using this algorithm.

The JAGA package allows us to plot various fitness values in each generation on the y-axis against the generation number on the x-axis.

For all parameter settings, we have observed the same type of behaviour (see Fig. 2):

- The worst fitness of each generation alternates around the same point. The higher the cut-off proportion, the smaller the alternation frequency.
- The best solution found so far and the best individual in each generation are the same (because the best individual never dies)
- The best solution grows very similar to a log function. In early generations, it increases very fast, but with time, it starts to grow slower and slower. At some point, it stops increasing and stays on a constant level.
- The average fitness of each generation's population very slowly asymptotically converges towards the best solution. It behaves, however not monotone, but performs alternations of a small amplitude.
- The standard deviation of the individual fitnesses in each generation slightly increases while the average is growing, and stays approximately constant when the average has converged to an approximately stable value.

The standard deviation is an indirect indicator for the diversity of a population.

So, these observations suggest that the diversity in a population and the fitness growth are closely related. This is a very important result, as it suggests that the power of GP indeed comes from the crossover operator and is not just a type of gradual ascent.

These observations also explain why it appears to be relatively easy to get a very high quality

result (90%-95%), but quite hard to get a 100% solution.

From this, we learn that our GP setting must depend on the result we want to achieve: when we are looking for a high quality solution and have not found it within a relatively small number of generations, we should stop the run and re-start it. However, if we are looking for a 100%-solution, we need to wait longer in order to allow the fitness to converge.

While these rules are difficult to quantify in general, they can be experimentally determined for some given problem.

Anyhow, these results make us armed to approach a harder problem...





**Fig. 2:** Elitist algorithm with  $E = 0.3$ ,  $C = 0.22$  in generation 110;  $E = 0.3$ ,  $C = 0.22$  in generation 200;  $E = 0.3$ ,  $C = 0$  in generation 110; and  $E = 0.3$ ,  $C = 0$  in generation 200 (from top to bottom). X-axis is generation number, Y-axis is fitness. Red is the least fitness in a generation, green is top fitness in a generation, blue is best fitness up-to-date, black is average generation fitness, orange is average fitness + standard deviation.

## 11-Multiplexer.

We used what we learned from working with the 3- and the 6-multiplexer to evolve an 11-multiplexer. Here are some selected results (note, that these results are achieved in the full-trace mode; time performance is about 20% faster when text and graph tracing is switched off).

Using only IF-nodes:

As expected, these do best – an IF-node is the most powerful building block, it is easiest to build a multiplexer from small multiplexers rather than simple logic gates.

This is the only configuration with which we evolved a solution, which solved all 2048 problem instances correctly.

We use  $maxDepth = 6$  here.

```
Selection: Tournament 6-0.9
pXOver: 0.8
pMutation: 0.15
Population: 1000
Elitist: 0.2, 0.1
Shortness: 3
Maximum fitness in generation 108 is
```

```
2032.0.
Run time: 01:37.97300
Total fitness evaluations: 28951
IF (p1) THEN (IF (p2) THEN (IF (p0)
THEN (p10) ELSE (p6)) ELSE (IF (p0)
THEN (p9) ELSE (IF (p1) THEN (p5) ELSE
(p9))) ELSE (IF (p2) THEN (IF (p0)
THEN (p8) ELSE (p4)) ELSE (IF (p0) THEN
(IF (IF (p0) THEN (p2) ELSE (p3)) THEN
(IF (IF (p5) THEN (p3) ELSE (p4)) THEN
(p9) ELSE (IF (p5) THEN (p10) ELSE
(p3))) ELSE (p7))) ELSE (IF (p2) THEN
(IF (IF (p10) THEN (p1) ELSE (p5)) THEN
(p5) ELSE (p4)) ELSE (IF (p0) THEN (p2)
ELSE (p3))))))
```

```
Selection: Tournament 6-0.8
pXOver: 0.8
pMutation: 0.15
Population: 1000
Elitist: 0.2, 0.1
Shortness: 2
Maximum fitness in generation 125 is
2038.0.
Run time: 02:19.139570
Total fitness evaluations: 33242
IF (IF (p1) THEN (p0) ELSE (p0)) THEN
(IF (p2) THEN (IF (p1) THEN (p10) ELSE
(p8)) ELSE (IF (IF (p1) THEN (p1) ELSE
(p1)) THEN (IF (p2) THEN (p10) ELSE
(p9) ELSE (IF (p1) THEN (IF (p2) THEN
(p4) ELSE (p5)) ELSE (IF (p2) THEN (p4)
ELSE (p7)))))) ELSE (IF (p1) THEN (IF
(p2) THEN (p6) ELSE (p5)) ELSE (IF (p2)
THEN (p4) ELSE (p3))))
```

```
Selection: Tournament 6-0.85
pXOver: 0.8
pMutation: 0.15
Population: 1000
Elitist: 0.2, 0.2
Shortness: 2.5
Maximum fitness in generation 158 is
2035.0.
Run time: 02:28.148223
Total fitness evaluations: 41536
IF (IF (p1) THEN (p0) ELSE (p0)) THEN
(IF (p2) THEN (IF (p1) THEN (p10) ELSE
(p8)) ELSE (IF (IF (p1) THEN (p1) ELSE
(p1)) THEN (IF (p2) THEN (p10) ELSE
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```
(p9)) ELSE (IF (p1) THEN (IF (p2) THEN
(p4) ELSE (p5)) ELSE (IF (p2) THEN (p4)
ELSE (p7)))) ELSE (IF (p1) THEN (IF
(p2) THEN (p6) ELSE (p5)) ELSE (IF (p2)
THEN (p4) ELSE (p3)))
```

Once, a 100%-solution with IF-nodes evolved as early as the 67<sup>th</sup> generation. Unfortunately, we did not log the parameters.

Using NAND- and OR-nodes:

<pre>Selection: Tournament 6-0.85 pXOver: 0.9 pMutation: 0.15 Population: 1000 Elitist: 0.1, 0.1 Shortness: 2</pre>
<pre>Maximum fitness in generation 200 is 1876.921875. Run time: 07:39.459071 Total fitness evaluations: 43209</pre>
<pre>((((p3 OR p2) NAND ((p4 OR p1) OR p0)) OR (((p6 NAND p2) NAND p1) OR ((p0 NAND p9) NAND p1)) NAND (((p1 NAND p2) NAND p8) NAND p0))) NAND (((((p5 OR p2) NAND (p1 OR p3)) NAND ((p2 NAND p2) OR (p2 NAND p9))) OR ((p5 OR (p5 OR ((p0 NAND p9) NAND p1))) NAND (p1 OR p3)) NAND (p7 NAND p0))) NAND ((p10 NAND (p8 OR p1)) OR (p0 NAND p0)) NAND p2)))</pre>

<pre>Selection: Tournament 6-0.85 pXOver: 0.85 pMutation: 0.1 Population: 1000 Elitist: 0.2, 0.1 Shortness: 2</pre>
<pre>Maximum fitness in generation 118 is 1903.9375. Run time: 06:15.375730 Total fitness evaluations: 23031</pre>
<pre>((((p0 NAND p1) NAND (p1 NAND p0)) NAND p9) NAND (((p1 NAND (p2 NAND p10)) NAND (p2 OR p7)) NAND p0) NAND ((p3 NAND p1) NAND ((p9 OR ((p0 NAND (((p0 OR p5) NAND p1) NAND (p1 NAND (p0 OR p2))) NAND p9) NAND ((p4 NAND p4) NAND p8))) OR (p4 OR p9))) NAND ((p2 OR p5) NAND (((((p0 NAND p8) NAND (p1 NAND p0)) NAND p9) NAND (((((p2 OR p7) NAND (p1 NAND p1)) NAND p8) NAND p8) NAND p0) NAND ((p1 NAND p1) NAND ((p9 OR ((p0 NAND (((p0 OR (p1 NAND p6)) NAND p1) NAND (p1 NAND (p0 OR p2))) NAND p9) NAND ((p4 NAND p1) NAND p8))) OR (p4 OR p9))) NAND ((p2 OR p2) NAND (p4 NAND</pre>

```
p4)) NAND (p3 OR p2)))) NAND ((p2 NAND
(((p1 NAND p1) NAND ((p9 OR ((p0 NAND
(((p4 NAND (p1 NAND (p0 OR
p2))) NAND p9) NAND ((p4 NAND p1) NAND
p8))) OR (p4 OR p9))) NAND (((p2 OR p8)
NAND (p4 NAND p4)) NAND (p3 OR p2)))
NAND ((p2 NAND (p1 NAND p6)) NAND (p5
OR p2))) NAND p6)) NAND (p5 OR p2))))
NAND p2)) NAND (p3 OR p2)))) NAND ((p2
NAND (p1 NAND (((p0 NAND p8) NAND (p1
NAND p0)) NAND p9) NAND (((p1 NAND (p2
NAND p10)) NAND (p2 OR p7)) NAND p0)
NAND ((p1 NAND (p1 NAND p0)) NAND ((p9
OR ((p0 NAND (((p0 OR p5) NAND p1)
NAND (p1 NAND (p0 OR p2))) NAND ((p2
OR p5) NAND (((p0 NAND p8) NAND (p1
NAND p0)) NAND p2) NAND (((((p2 OR p7)
NAND (p1 NAND p1)) NAND p8) NAND p8)
NAND p0) NAND ((p1 NAND p1) NAND ((p9
OR ((p0 NAND (((p0 OR (p1 NAND p6))
NAND p1) NAND (p1 NAND (p0 OR p2)))
NAND p9) NAND ((p4 NAND p1) NAND p8)))
OR (p4 OR p9))) NAND ((p2 OR p2) NAND
(p4 NAND p4)) NAND (p3 OR p2))) NAND
((p4 NAND p1) NAND p8))) OR (p4 OR
p9))) NAND ((p2 OR p5) NAND (((p0
NAND p8) NAND (p1 NAND p0)) NAND p2)
NAND (((((p2 OR p7) NAND (p1 NAND p1))
NAND p8) NAND p8) NAND p0) NAND ((p1
NAND p1) NAND ((p9 OR ((p0 NAND (((p0
OR (p1 NAND p6)) NAND p1) NAND (p1 NAND
(p0 OR p2))) NAND p9) NAND ((p4 NAND
p1) NAND p8))) OR (p4 OR p9))) NAND
(((p2 OR p2) NAND (p4 NAND p4)) NAND
(p3 OR p2)))) NAND p6))) NAND p4)) NAND
(p3 OR p2)))) NAND ((p2 NAND (p1 NAND
p6)) NAND (p5 OR p2)))))) NAND ((p0 OR
p5) OR p2))))
```

<pre>Selection: Tournament 6-0.85 pXOver: 0.85 pMutation: 0.15 Population: 1000 Elitist: 0.2, 0.1 Shortness: 4</pre>
<pre>Generation: 129 Fitness evaluations: 29933 Run time: 06:34.394877 fitness=1831.90625</pre>
<pre>((((((((((p6 NAND (p2 OR p4)) OR p0) NAND p1) NAND p2) NAND ((p10 NAND p0) NAND p2) NAND p1)) NAND p2) NAND p2) NAND ((p10 OR p9) OR p10) OR (((p3 NAND p2) NAND p3) OR p1) NAND p0) OR p10))) OR (((p0 NAND p9) NAND ((p5 NAND p1) OR p10)) OR p5) OR p2) NAND p1)) NAND (((p0 NAND (p6 NAND p2)) OR p2) OR p7) NAND (((p3 NAND p2) NAND</pre>

p3) OR p1) NAND (p0 NAND p8))) OR p1))
--

**Using AND-, OR- and NOT-nodes:**

Selection: Tournament 6-0.9 pXOver: 0.9 pMutation: 0.15 Population: 1000 Elitist: 0.2, 0.1 Shortness: 2
Fitness evaluations: 27353 Run time: 03:41.221248 Fitness = 1780.875 Generation = 122
((p7 OR (p2 OR p1)) AND ((p4 OR p1) AND ((p7 OR (p2 OR p1)) AND ((p4 OR p1) AND ((p2 AND p10) OR !p0)) OR !p2)) AND (((!p2 AND p0) OR p0) OR ((p2 AND p6) OR !p1))) OR !p2)) AND (((!p2 AND p0) OR ((p5 OR p5) AND ((p2 AND p10) OR !p0)) OR (p9 AND p0))) OR ((p2 AND p6) OR !p1)))

## Results

Besides the conclusions stated above we have two important results:

1) We could indeed evolve an 11-multiplexer made up of IF-nodes and various very good approximations made up of other nodes.

2) We run various runs with only IF-nodes, population size 1000,  $maxDepth = 6$  and different combinations of other parameters (while the other parameters only affect the computation, the two stated parameters influence the computational resources required). We have averaged results of all those runs to construct a cumulative averaged generation-number  $\rightarrow$  evolved-solution-quality graph. To this graph we fitted a function.

This allows us to approximately predict, how many generations do we need to wait to get a solution of desired quality, given the computational resources are similar to the ones used in the research (see above).

This is not an amazing, but nevertheless useful result.

The fitted formula for expected quality  $Q$  and generation number  $G$  is:

$$Q = 50 + \log_K(G)$$

$Q$  is measured in per cent.

So the prediction formula is:

$$G = pred_K(Q) = K^{(Q-50)}$$

Here  $K$  is a constant with  $1.1 \leq K \leq 1.3$ , which depends on the parameters and probability. We experimentally determined that in 90% of cases the observed  $G$  will be in the interval  $[pred_{1.1}(Q), pred_{1.3}(Q)]$ .

## Future Prospects.

In the short time available for undertaking this work we have been satisfied with the performance of our GP system and the JAGA API. We have, nonetheless, identified a series of possible ideas on more elaborately assessing and improving their effectiveness, which we believe

are also very interesting research ideas in themselves. We outline the most important of these in this section.

### Harder selection criteria with generation age.

We believe that by gradually increasing the difficulty of our various selection criteria as the generation age increases, we can achieve a fine-graining effect on our population. Examples of this include gradually increasing the selection parameter in tournament selection and gradually increasing the elitist rate so that a bigger proportion of fitter individuals are left unchanged in new generations as the generation age increases. We have performed initial experiments on these and we believe their effectiveness should be further evaluated.

### Variable Population Size.

Vandiver [12] has proposed varying the population size with time. We recognize that this is an idea that brings GP closer to real-life human evolution and could potentially offer performance and/or quality advantages to a GP algorithm. We believe our framework could be used to conduct further experiments on this. We are particularly interested in seeing how a decreasing population size setting would perform.

### More Elaborate Elitist Selection System.

The simplest form of an elitist selection system is, for example, producing identical copies of the top 10% fitter individuals and evolving the rest (90%) of the population. There seems to be a lot more to elitist selection than this though, as a refinement of this model would be to, additionally, disallow the bottom 10% less fit individuals from reproducing. Even more importantly, though, we carried out some experiments that showed us that an elitist selection system can bring improvements even when it favours parts of the population other than the fitter individuals.

### Stack Implementation.

We consider the stack representation of programs [3] to be the strongest alternative to the tree representation we have implemented. As a means of assessing the effectiveness of this choice we would wish to implement the stack method and perform some back to back tests with the tree representation.

### **Digital Circuit Multiplexer Implementation.**

A multiplexer is an important component in digital circuit design. Considerable human thinking in digital circuit architecture has lead to

the knowledge that a simple 6-multiplexer can be constructed out of two NOT gates, four 3-input AND gates and one 4-input OR gate.

We believe that, based on this knowledge, there is a way to further investigate the effectiveness of our system which is also quite an interesting undertaking in itself. We propose to expand the system to allow for 3-input AND functions and 4-input OR functions and then undertake the GP procedure. If the system can then evolve the same solution as the “human” one outlined above and, furthermore, if it can do so in relatively low computational time (small number of generations, small initial populations size, etc.), then we believe this would constitute an important achievement.

## References

- [1] Koza, J.R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, MA: MIT Press, 1992.
- [2] Mitchell, M., *An Introduction to Genetic Algorithms*, MIT Press, 1996.
- [3] Perkis, T., *Stack-Based Genetic Programming*, Proceedings of the 1994 IEEE World Congress on Computational Intelligence.
- [4] Houck et al, *A Genetic Algorithm for Function Optimization: A Matlab Implementation*, North Carolina State University, 1996.
- [5] <http://www.corba.org/>
- [6] <http://java.sun.com/>
- [7] <http://jgap.sourceforge.net/>
- [8] <http://jgprog.sourceforge.net/>
- [9] <http://www.aridolan.com/ga/gaa/gaa.html>
- [10] <http://www.paperin.org/>
- [11] W.Langdon, *Quadratic Bloat in Genetic Programming*, Proceedings of the GECCO 2000.
- [12] Alex Vandiver, *Genetic Programming and the effect of Variable Population Size*, 1999

## Appendix.

Here is the source code of the parts of JAGA 0.9 Beta most relevant to this work. The full package will be available soon. The time did unfortunately not allow fully commenting the code. This will be caught up on for the release of the package.

The classes are listed in the following order:

```
com.gregPaperin.ga.multiplexer.fitnessSelection.MultiplexerFitness
com.gregPaperin.ga.multiplexer.fitnessSelection.TournamentSelection
com.gregPaperin.ga.multiplexer.representation.BooleanFormulaTree
com.gregPaperin.ga.multiplexer.representation.BooleanFormulaTreeFactory
com.gregPaperin.ga.multiplexer.reproduction.FunctionTreeMutation
com.gregPaperin.ga.multiplexer.reproduction.FunctionTreeXOver
com.gregPaperin.ga.simpleImplementation.fitnessSelection.RouletteWheelSelection
com.gregPaperin.ga.simpleImplementation.SimpleGA
com.gregPaperin.ga.multiplexer.ElitistGA
com.gregPaperin.ga.multiplexer.MultiplexerEvolution
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

**MultiplexerFitness.java**

```
package
com.gregPaperin.ga.multiplexer.fitnessSelection;

import com.gregPaperin.ga.definitions.*;
import
com.gregPaperin.ga.simpleImplementation.fitnessSe
lection.AbsoluteFitness;
import
com.gregPaperin.ga.multiplexer.representation.*;

public class MultiplexerFitness implements
FitnessEvaluationAlgorithm {

    private class TestEntry {
        private boolean [] testVals = null;
        private boolean refResult = false;
        private TestEntry() {assert false : "Dont
use!";}
        public TestEntry(Multiplexer refMulti, int
encodedVals) {
            testVals = new
boolean[refMulti.getTotalLines()];
            for (int i = refMulti.getTotalLines() -
1; i >= 0; i--) {
                testVals[i] = (0 != (encodedVals &
0x1));
            }
            encodedVals >>= 1;
            refResult = refMulti.evaluate(testVals);
        }
        public boolean [] getTestVals() {
            return testVals;
        }
        public boolean getReferenceResult() {
            return refResult;
        }
        public String toString() {
            StringBuffer s = new StringBuffer("[");
            for (int i = 0; i < testVals.length;
i++) {
                if (i > 0)
                    s.append(", ");
                s.append(testVals[i] ? "1" : "0");
            }
            s.append("] => ");
            s.append(refResult ? "1" : "0");
            return s.toString();
        }

        private TestEntry [] tests = null;
        private double pressForShortResultFactor =
0.0;

        public MultiplexerFitness() {
            initTests(new Multiplexer());
        }

        public MultiplexerFitness(Multiplexer
referenceMultiplexer) {
            if (null == referenceMultiplexer)
                throw new
NullPointerException("referenceMultiplexer may
not be null");
            initTests(referenceMultiplexer);
        }

        public MultiplexerFitness(double
pressForShortFactor) {
            this.pressForShortResultFactor =
pressForShortFactor;
            initTests(new Multiplexer());
        }

        public double getPressForShortFactor() {
            return this.pressForShortResultFactor;
        }

        public void setPressForShortFactor(double
pressForShortFactor) {
            this.pressForShortResultFactor =
pressForShortFactor;
        }

        private void initTests(Multiplexer
referenceMultiplexer) {
            int testCount = (int) Math.round(Math.pow(2,
referenceMultiplexer.getTotalLines()));
            tests = new TestEntry[testCount];
            for (int encodedTstVals = 0; encodedTstVals <
testCount; encodedTstVals++) {
                tests[encodedTstVals] = new
TestEntry(referenceMultiplexer, encodedTstVals);
            }
        }

        public Fitness evaluateFitness(Individual
individual, int age, Population population,
GAPParameterSet params) {
            BooleanFormulaTree formula = (BooleanFormulaTree)
individual;
            int correct = 0;
            for (int i = 0; i < tests.length; i++) {
                boolean expected =
tests[i].getReferenceResult();
                boolean [] args = tests[i].getTestVals();
                boolean actual = formula.evaluate(args);
                if (expected == actual)
                    ++correct;
            }
            double fract = ((double) correct) / ((double)
tests.length);
            double whole = (double) correct;

            // double fitVal = whole + fract;

            double fitVal = Math.round(whole -
pressForShortResultFactor *
Math.log(formula.getNodeCount()) / Math.log(2)
) + fract;

            AbsoluteFitness fitness = new
AbsoluteFitness(fitVal);
            return fitness;
        }

        /*
        //TEST:
        public static void main(String[] args) {
            Multiplexer mult = new Multiplexer(4);
            MultiplexerFitness fitEval = new
MultiplexerFitness(mult);
            for (int t = 0; t < fitEval.tests.length; ++t)
                System.out.println(fitEval.tests[t]);
        }
    }
}
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```
*/
}

TournamentSelection.java

package
com.gregPaperin.ga.multiplexer.fitnessSelection;

import com.gregPaperin.ga.definitions.*;
import
com.gregPaperin.ga.simpleImplementation.fitnessSe
lection.AbsoluteFitness;
import java.util.Arrays;

public class TournamentSelection implements
SelectionAlgorithm {

    private static final Class
applicableFitnessClass = AbsoluteFitness.class;

    private int competitorsNum = 2;
    private double betterCandidateProbability =
0.7;

    public TournamentSelection() {
    }

    public TournamentSelection(int oneOutOf,
double betterCandProb) {
        setCompetitors(oneOutOf);
    }

    public int getCompetitors() {
        return this.competitorsNum;
    }

    public void setCompetitors(int oneOutOf) {
        if (oneOutOf < 1)
            throw new
IllegalArgumentException("Tournament must include
at least 1 competitor");
        this.competitorsNum = oneOutOf;
    }

    public double getBetterCandidateProbability()
{
        return this.betterCandidateProbability;
    }

    public void
setBetterCandidateProbability(double
betterCandProb) {
        if (betterCandProb < 0 || 1 <
betterCandProb)
            throw new
IllegalArgumentException("Better candidate
selection "
+
"probability must be in [0, 1]");
        this.betterCandidateProbability =
betterCandProb;
    }

    public Class getApplicableFitnessClass() {
        return applicableFitnessClass;
    }

    public Individual select(Population
population, GAParameterSet params) {
        RandomGenerator rnd =
params.getRandomGenerator();
```

```
int popSize = population.getSize();
Individual [] competitors = new
Individual[competitorsNum];
for (int i = 0; i < competitorsNum; i++) {
    int p = rnd.nextInt(0, popSize);
    competitors[i] = population.getMember(p);
}
Arrays.sort(competitors, new
AbsoluteFitnessIndividualComparator());

int ci = competitorsNum - 1;
for ( ; ; ) {
    double dice = rnd.nextDouble();
    if (dice < betterCandidateProbability)
        return competitors[ci];
    if (--ci < 0)
        ci = competitorsNum - 1;
}
}

public Individual[] select(Population population,
int howMany, GAParameterSet params) {
    Individual [] selection = new
Individual[howMany];
    for (int i = 0; i < howMany; i++) {
        selection[i] = select(population, params);
    }
    return selection;
}
}

BooleanFormulaTree.java

package com.gregPaperin.ga.multiplexer.representation;

import com.gregPaperin.ga.definitions.*;
import
com.gregPaperin.ga.multiplexer.representation.nodes.*;
import java.util.HashMap;
import java.util.Iterator;

public class BooleanFormulaTree implements Individual {

    // remember the max-depth constrain

    private int numberOfParameters = 0;
    private Fitness fitness = null;
    private HashMap nodes = new HashMap();
    private BooleanFormulaTreeNode root = null;
    private long nextHandleToGenerate = -Long.MIN_VALUE;

    private BooleanFormulaTree() {
        throw new
java.lang.UnsupportedOperationException("Use the other
constructor");
    }

    public BooleanFormulaTree(int numberOfParameters) {
        this.numberOfParameters = numberOfParameters;
        root = new FalseNode();
        addToNodeList(root, 0);
    }

    private Long generateNewHandle() {
        if (Long.MAX_VALUE == nextHandleToGenerate)
            throw new
RuntimeException("nextHandleToGenerate="
+ nextHandleToGenerate +
", which is too big; ")
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

                                + " In this
version, a formula tree cannot "
                                + "generate that
much handles");
    Long handle = new
Long(nextHandleToGenerate++);
    if (0 == nextHandleToGenerate)
        nextHandleToGenerate = 1;
    return handle;
}

public Fitness getFitness() {
    return this.fitness;
}

public void setFitness(Fitness fitness) {
    this.fitness = fitness;
}

public int getNumberOfParameters() {
    return this.numberofParameters;
}

public Long selectRootNode() {
    return root.getHandle();
}

public int getNodeCount() {
    return this.nodes.size();
}

public Long selectRandomNode(GAPParameterSet
params) {
    int nodeCount = getNodeCount();
    if (0 == nodeCount)
        return new Long(0);
    int rndNum =
params.getRandomGenerator().nextInt(0,
nodeCount);
    Iterator key = getHandlersIterator();
    Object keyVal = key.next();
    int i = 0;
    while (i++ < rndNum)
        keyVal = key.next();
    return (Long) keyVal;
}

public Iterator getHandlersIterator() {
    return nodes.keySet().iterator();
}

public BooleanFormulaTreeNode exportNode(Long
handle) {
    BooleanFormulaTreeNode node =
getNode(handle);
    BooleanFormulaTreeNode clone =
(BooleanFormulaTreeNode) node.clone();
    return clone;
}

public int getNodeDepth(Long handle) {
    BooleanFormulaTreeNode node =
getNode(handle);
    return node.getDepth();
}

public int getNodeHeight(Long handle) {
    BooleanFormulaTreeNode node =
getNode(handle);
    return node.getHeight();
}

private BooleanFormulaTreeNode getNode(Long
handle) {
    if (0 == handle.longValue())

```

```

        throw new IllegalArgumentException("The handle
0 is illegal");
        BooleanFormulaTreeNode node =
(BooleanFormulaTreeNode) nodes.get(handle);
        assert null != node : "Must be an invalid handle
(" + handle.longValue() + ")";
        return node;
    }

    public void replaceNode(Long oldNodeHandle,
BooleanFormulaTreeNode newNode) {

        if (null == newNode)
            throw new NullPointerException("Cannot replace
a node by a null-pointer");

        BooleanFormulaTreeNode toReplace =
getNode(oldNodeHandle);

        if (root == toReplace) { // if we are replacing
the root:
            try {
                nodes.clear();
                root = newNode;
            } catch (OutOfMemoryError e) {
                e.printStackTrace(); throw new
RuntimeException(e.getMessage());
            }
            try {
                addToNodeList(newNode, 0);
            } catch (OutOfMemoryError e) {
                e.printStackTrace(); throw new
RuntimeException(e.getMessage());
            }
            try {
                root.recalcHeight();
            } catch (OutOfMemoryError e) {
                e.printStackTrace(); throw new
RuntimeException(e.getMessage());
            }
        } else { // any other node:
            removeFromNodeList(toReplace);
            OperatorNode parent = toReplace.getParent();
            assert null != parent : "Node has no parent!";
            int chInd = parent.findChild(toReplace);
            assert chInd >= 0 : "Need debug !!!!";
            parent.setChild(chInd, newNode);
            addToNodeList(newNode, parent.getDepth() + 1);
            parent.recalcHeight();
        }
    }

    public boolean evaluate(boolean [] parameters) {
        return root.evaluate(parameters);
    }

    public boolean
isIndividualValid(IndividualConstraint[] constraints) {
        for (int i = 0; i < constraints.length; i++) {
            if
(!constraints[i].getApplicableClass().isInstance(this))
                throw new
IllegalArgumentException("Constraint number " + i
+ " can only be
applied to "
+
constraints[i].getApplicableClass().getName()
+ " and does not
support this class ("
+
this.getClass().getName() + ")");
            if (!constraints[i].isIndividualLegal(this))
                return false;
        }
        return true;
    }

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```
}

public String toString() {
    return toString(true);
}

public String toString(boolean infix) {
    StringBuffer s = new
StringBuffer("{formula=");
    s.append(root.toStringBuffer(infix));
    s.append(", ");
    if (null == fitness) {
        s.append("fitness not known");
    } else {
        s.append("fitness=");
        s.append(fitness.toString());
        s.append("}");
    }
    return s.toString();
}

private void
removeFromNodeList(BooleanFormulaTreeNode node) {
    Long key = node.getHandle();
    nodes.remove(key);
    if (node instanceof OperatorNode) {
        OperatorNode parent = (OperatorNode)
node;
        for (int i = 0; i <
parent.getRequiredChildrenNumber(); i++) {
            BooleanFormulaTreeNode kid =
parent.getChild(i);
            if (null != kid)
                removeFromNodeList(kid);
        }
        node.setHandle(new Long(0));
    }
}

private void
addToNodeList(BooleanFormulaTreeNode node, int
depth) {
    Long handle = null;

    try {
        node.setDepth(depth);
    } catch (OutOfMemoryError e) {
        e.printStackTrace(); throw new
RuntimeException(e.getMessage());
    }

    try {
        handle = generateNewHandle();
    } catch (OutOfMemoryError e) {
        e.printStackTrace(); throw new
RuntimeException(e.getMessage());
    }

    try {
        nodes.put(handle, node);
    } catch (OutOfMemoryError e) {
        System.out.println(nodes.size());
        e.printStackTrace(); throw new
RuntimeException(e.getMessage());
    }

    try {
        node.setHandle(handle);
    } catch (OutOfMemoryError e) {
        e.printStackTrace(); throw new
RuntimeException(e.getMessage());
    }

    if (node instanceof OperatorNode) {
        try {
            OperatorNode parent = (OperatorNode)
node;
```

```
        for (int i = 0; i <
parent.getRequiredChildrenNumber(); i++) {
            BooleanFormulaTreeNode kid =
parent.getChild(i);
            if (null != kid)
                addToNodeList(kid, depth + 1);
        }
    } catch (OutOfMemoryError e) {
        e.printStackTrace(); throw new
RuntimeException(e.getMessage());
    }
}
}
```

---

**BooleanFormulaTreeFactory.java**

```
package com.gregPaperin.ga.multiplexer.representation;

import com.gregPaperin.ga.definitions.*;
import
com.gregPaperin.ga.multiplexer.representation.nodes.*;

public class BooleanFormulaTreeFactory implements
IndividualsFactory {

    private static final Class [] stoppersWithConsts =
new Class[] { TrueNode.class,
FalseNode.class,
TerminalNode.class };

    private static final Class [] stoppersWithoutConsts
= new Class[] { TerminalNode.class };

    private int maxTreeDepth = 20;
    private int numberOfParameters = 6;
    private Class [] allowedNodeTypes = new Class[] {
ANDNode.class, ORNode.class,
NOTNode.class,
TerminalNode.class };
    private boolean allowConstants = true;

    public BooleanFormulaTreeFactory() {}

    public BooleanFormulaTreeFactory(int maxTreeDepth,
int numberOfParameters,
Class [] allowedNodeTypes,
boolean allowConstants) {
        this.numberOfParameters = numberOfParameters;
        if (null == allowedNodeTypes)
            this.allowedNodeTypes = new Class[0];
        else
            this.allowedNodeTypes = allowedNodeTypes;
        this.allowConstants = allowConstants;
        this.maxTreeDepth = maxTreeDepth;
    }

    public int getMaxTreeDepth() {
        return this.maxTreeDepth;
    }

    public void setMaxTreeDepth(int maxDepth) {
        if (maxDepth < 1)
            throw new IndexOutOfBoundsException("Max. tree
depth may not be < 1");
        this.maxTreeDepth = maxDepth;
    }

    public int getNumberOfParameters() {
        return this.numberOfParameters;
    }
}
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

    }

    public void setNumberOfParameters(int val) {
        if (numberOfParameters < 0)
            throw new
IndexOutOfBoundsException("Number of parameters
may not be < 0");
        this.numberOfParameters = val;
    }

    public Class [] getAllowedNodeTypes() {
        return this.allowedNodeTypes;
    }

    public void setAllowedNodeTypes(Class [] val)
{
    if (null == allowedNodeTypes)
        this.allowedNodeTypes = new Class[0];
    else
        this.allowedNodeTypes = val;
}

    public boolean getAllowConstants() {
        return this.allowConstants;
    }

    public void setAllowConstants(boolean val) {
        this.allowConstants = val;
    }

    public Individual
createDefaultIndividual(GAPParameterSet params) {
        BooleanFormulaTree formula = new
BooleanFormulaTree(numberOfParameters);
        return formula;
    }

    public Individual
createRandomIndividual(GAPParameterSet params) {
        BooleanFormulaTree formula = null;
        Long root = null;
        BooleanFormulaTreeNode node = null;
        try {
            formula = new
BooleanFormulaTree(numberOfParameters);
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException("1 " + e.getMessage());
        }
        try {
            root = formula.selectRootNode();
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException("2 " + e.getMessage());
        }
        try {
            node = createRandomNode(1, params);
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException("3 " + e.getMessage());
        }
        try {
            formula.replaceNode(root, node);
            return formula;
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException("4 " + e.getMessage());
        }
    }

    public Individual
createSpecificIndividual(Object init,
GAPParameterSet params) {
        if (init instanceof BooleanFormulaTree)

```

```

        return clone((BooleanFormulaTree) init,
params);
        throw new ClassCastException("Initialisation
value for BooleanFormulaTree "
+ "must be of type Integer
(but is "
+ init.getClass() + ")");
    }

    public BooleanFormulaTree clone(BooleanFormulaTree
template, GAPParameterSet params) {
        try {
            BooleanFormulaTree copy = new
BooleanFormulaTree(template.
getNumberOfParameters());
            copy.setFitness(template.getFitness());
            Long trh = template.selectRootNode();
            BooleanFormulaTreeNode nodeClone =
template.exportNode(trh);
            Long crh = copy.selectRootNode();
            copy.replaceNode(crh, nodeClone);
            return copy;
        } catch (OutOfMemoryError e) {
            e.printStackTrace();
            throw new RuntimeException(e.getMessage());
        }
    }

    public Class getSupportedConstraintClass() {
        return null;
    }

    public BooleanFormulaTreeNode createRandomNode(int
thisDepth, GAPParameterSet params) {

        BooleanFormulaTreeNode node = null;

        try {
            if (thisDepth >= this.maxTreeDepth)
                return createRandomStopNode(params);
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException(e.getMessage());
        }
        try {
            node =
createOneNodeOfRandomType(allowedNodeTypes, params);
        } catch (OutOfMemoryError e) {
            e.printStackTrace(); throw new
RuntimeException(e.getMessage());
        }

        for (int i = 0; i <
node.getRequiredChildrenNumber(); i++) {
            BooleanFormulaTreeNode child = null;
            try {
                child = createRandomNode(thisDepth + 1,
params);
            } catch (OutOfMemoryError e) {
                e.printStackTrace(); throw new
RuntimeException(e.getMessage());
            }
            try {
                ((OperatorNode) node).setChild(i, child);
            } catch (OutOfMemoryError e) {
                e.printStackTrace(); throw new
RuntimeException(e.getMessage());
            }
        }

        return node;
    }
}

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```
private BooleanFormulaTreeNode
createRandomStopNode(GAPParameterSet params) {
    if (allowConstants)
        return
createOneNodeOfRandomType(stoppersWithConsts,
params);
    else
        return
createOneNodeOfRandomType(stoppersWithoutConsts,
params);
}

private BooleanFormulaTreeNode
createOneNodeOfRandomType(Class [] nodeTypes,
GAPParameterSet params) {

    BooleanFormulaTreeNode node = null;

    int rndType =
params.getRandomGenerator().nextInt(0,
nodeTypes.length);
    Class type = nodeTypes[rndType];

    try {
        node = (BooleanFormulaTreeNode)
type.newInstance();
    } catch (InstantiationException e1) {
        throw new Error("Dodgy: can't
instantiate " + type.getName()
+ ". (" + e1.getMessage() +
")");
    } catch (IllegalAccessException e2) {
        throw new Error("Dodgy: can't
instantiate " + type.getName()
+ ". (" + e2.getMessage() +
")");
    } catch (OutOfMemoryError e3) {
        throw new Error("Dodgy: can't
instantiate " + type.getName()
+ ". (" + e3.getMessage() +
")");
    }

    assert null != node;

    if (node instanceof TerminalNode) {
        int rndTerm =
params.getRandomGenerator().nextInt(0,
this.numberofParameters);
        ((TerminalNode)
node).setTerminalIndex(rndTerm);
    }

    return node;
}

/*
//TEST:
public static void main(String[] args) {
    GAPParameterSet params = new
com.gregPaperin.ga.simpleImplementation.DefaultPa
rameterSet();
    BooleanFormulaTreeFactory fact = new
BooleanFormulaTreeFactory();
    fact.setMaxTreeDepth(6);
    fact.setAllowConstants(false);
    for (int i = 0; i < 5; i++) {
        Individual form =
fact.createRandomIndividual(params);
        System.out.println(i + " " + form);
    }
}
*/
```

```
}
```

---

**FunctionTreeMutation.java**

```
package com.gregPaperin.ga.multiplexer.reproduction;

import com.gregPaperin.ga.definitions.*;
import java.util.Iterator;
import
com.gregPaperin.ga.multiplexer.representation.nodes.*;
import com.gregPaperin.ga.multiplexer.representation.*;
import
com.gregPaperin.ga.simpleImplementation.reproduction.Mu
tation;

public class FunctionTreeMutation extends Mutation {

    private static final Class applicableClass =
BooleanFormulaTree.class;

    public FunctionTreeMutation() {
        super();
    }

    public FunctionTreeMutation(double mutProb) {
        super(mutProb);
    }

    public Class getApplicableClass() {
        return applicableClass;
    }

    public Individual[] reproduce(Individual[] parents,
GAPParameterSet params) {

        Individual [] kids = new
Individual[parents.length];
        for (int i = 0; i < kids.length; i++) {
            if
(!getApplicableClass().isInstance(parents[i]))
                throw new ClassCastException("Incompatible
parent class: must be "
+
getApplicableClass().getName()
+ ", but is " +
parents.getClass().getName());
            kids[i] = mutate((BooleanFormulaTree)
parents[i], params);
        }
        return kids;
    }

    private BooleanFormulaTree mutate(BooleanFormulaTree
parent, GAPParameterSet params) {

        BooleanFormulaTree kid =
createSpecificIndividual(parent, params);

        if (params.getRandomGenerator().nextDouble() <
getMutationProbability()) {
            final BooleanFormulaTreeFactory fact =
fetchFactory(params);
            final Long handle =
kid.selectRandomNode(params);
            int depth = kid.getNodeDepth(handle);
            BooleanFormulaTreeNode replacement =
fact.createRandomNode(depth, params);
            kid.replaceNode(handle, replacement);
            kid.setFitness(null);
        }

        return kid;
    }
}
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```
    }

    private BooleanFormulaTree
createSpecificIndividual(Object init,
GAPParameterSet params) {
    final BooleanFormulaTreeFactory fact =
fetchFactory(params);
    final Individual form =
(BooleanFormulaTree)
fact.createSpecificIndividual(init, params);
    return (BooleanFormulaTree) form;
}

private BooleanFormulaTreeFactory
fetchFactory(GAPParameterSet params) {
    return (BooleanFormulaTreeFactory)
params.getIndividualsFactory();
}
}
```

### FunctionTreeXOver.java

```
package
com.gregPaperin.ga.multiplexer.reproduction;

import com.gregPaperin.ga.definitions.*;
import
com.gregPaperin.ga.simpleImplementation.reproduct
ion.XOver;
import
com.gregPaperin.ga.multiplexer.representation.*;
import
com.gregPaperin.ga.multiplexer.representation.nod
es.BooleanFormulaTreeNode;

public class FunctionTreeXOver extends XOver {

    private static final Class applicableClass =
BooleanFormulaTree.class;

    public FunctionTreeXOver() {
        super();
    }

    public FunctionTreeXOver(double xOverProb) {
        super(xOverProb);
    }

    public Class getApplicableClass() {
        return this.applicableClass;
    }

    public Individual[] reproduce(Individual[]
parents, GAPParameterSet params) {

        if (null == parents || parents.length !=
getRequiredNumberOfParents())
            throw new IllegalArgumentException("Must
have " + getRequiredNumberOfParents()
+ "
parents");

        RandomGenerator rand =
params.getRandomGenerator();
        Individual [] kids = copyParents(parents,
params);

        if (getXOverProbability() >
rand.nextDouble())
            return kids;
```

```
        BooleanFormulaTree mum = (BooleanFormulaTree)
kids[0];
        BooleanFormulaTree dad = (BooleanFormulaTree)
kids[1];

        int maxDepth = ((BooleanFormulaTreeFactory)
params.getIndividualsFactory()).getMaxTreeDepth();
        int attempts = 0;
        boolean kidsAreValid = false;
        while (!kidsAreValid && attempts <=
params.getMaxBadReproductionAttempts()) {

            Long mumHandle = mum.selectRandomNode(params);
            Long dadHandle = dad.selectRandomNode(params);

            int mumHeight = mum.getNodeHeight(mumHandle);
            int dadHeight = dad.getNodeHeight(dadHandle);
            int mumDepth = mum.getNodeDepth(mumHandle);
            int dadDepth = dad.getNodeDepth(dadHandle);

            if (mumDepth + dadHeight <= maxDepth ||
dadDepth + mumHeight <= maxDepth) {

                BooleanFormulaTreeNode mumNode =
mum.exportNode(mumHandle);
                BooleanFormulaTreeNode dadNode =
dad.exportNode(dadHandle);

                mum.replaceNode(mumHandle, dadNode);
                dad.replaceNode(dadHandle, mumNode);

                kidsAreValid = true;
                mum.setFitness(null);
                dad.setFitness(null);

            } else { // i.e. if mum + dad > maxDepth
                ++attempts;
            }

        }

        return kids;

    }

    private Individual [] copyParents(Individual[]
parents, GAPParameterSet params) {
        BooleanFormulaTreeFactory factory =
(BooleanFormulaTreeFactory)
params.getIndividualsFactory();
        Individual [] clones = new
Individual[parents.length];
        for (int i = 0; i < clones.length; i++) {
            clones[i] =
factory.createSpecificIndividual(parents[i], params);
        }
        return clones;
    }
}
```

### RouletteWheelSelection.java

```
package
com.gregPaperin.ga.simpleImplementation.fitnessSelectio
n;

import com.gregPaperin.ga.definitions.*;
import java.math.BigDecimal;

public class RouletteWheelSelection implements
SelectionAlgorithm {
```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

private static final Class
applicableFitnessClass = AbsoluteFitness.class;
public static final double MIN_FITNESS_LIMIT =
-Double.MAX_VALUE;

private double minFitness = MIN_FITNESS_LIMIT;

public RouletteWheelSelection() {}

public RouletteWheelSelection(double
minFitness) {
    setMinFitness(minFitness);
}

public double getMinFitness() {
    return this.minFitness;
}

public void setMinFitness(double minFitness) {
    if (Double.isNaN(minFitness)
        || Double.NEGATIVE_INFINITY ==
minFitness
        || Double.POSITIVE_INFINITY ==
minFitness)
        throw new
IllegalArgumentException("Minumum fitness is an
illegal "
                                + "value or
larger then maximum "
                                + " fitness
(" + minFitness + ")");
    this.minFitness = minFitness;
}

public Class getApplicableFitnessClass() {
    return applicableFitnessClass;
}

public Individual select(Population
population, GAPParameterSet params) {
    double cumSum =
calculateCumulativeFitness(population);
    return spinRoulette(cumSum, population,
params);
}

public Individual [] select(Population
population, int howMany, GAPParameterSet params) {
    double cumSum =
calculateCumulativeFitness(population);
    Individual [] selection = new
Individual[howMany];
    for (int i = 0; i < howMany; i++) {
        selection[i] = spinRoulette(cumSum,
population, params);
    }
    return selection;
}

private double
calculateCumulativeFitness(Population population)
{
    double sum = 0;
    for (int i = 0; i < population.getSize();
i++) {
        double fitVal =
getRelativeFitnessValue(population.getMember(i));
        sum += fitVal;
    }
    if (sum == Double.POSITIVE_INFINITY ||
Double.isNaN(sum))
        throw new
IllegalStateException("\nCumulative fitness is "
+ sum

```

```

+ ", which is over the
bound. "
+ "Maybe lower fitness
bound is "
+ "set too low? (" +
this.minFitness + ")");
    return sum;
}

private Individual spinRoulette(double cumSum,
Population pop, GAPParameterSet params) {

    int popSize = pop.getSize();

    if (0 == cumSum)
        return
pop.getMember(params.getRandomGenerator().nextInt(0,
popSize));

    double roulette =
params.getRandomGenerator().nextDouble(0, cumSum);

    double loB = 0.0;
    double upB = 0.0;
    for (int i = 0; i < popSize; i++) {
        Individual indiv = pop.getMember(i);
        double fval = getRelativeFitnessValue(indv);
        upB += fval;
        if (loB <= roulette && roulette < upB)
            return indiv;
        loB = upB;
    }

    throw new Error("Something is dodgy, this line
should never be executed!");
}

private double getRelativeFitnessValue(Individual
indv) {
    Fitness f = indiv.getFitness();
    assert null != f : "Individuall has null
fitness";
    AbsoluteFitness fit = (AbsoluteFitness) f;
    double rval = fit.getValue() - this.minFitness;
    if (0 > rval)
        throw new
IllegalArgumentException("\nIndividual has a fitness
value smaller "
                                + "then the boundary (" +
fit.getValue() + " < " + this.minFitness);
    return rval;
}
}

```

---

### SimpleGA.java

```

package com.gregPaperin.ga.simpleImplementation;

import com.gregPaperin.ga.definitions.*;
import java.util.ArrayList;
import java.util.Iterator;
import com.gregPaperin.ga.simpleImplementation.hooks.*;

public class SimpleGA implements GeneticAlgorithm {

    private ArrayList hooks = null;

    public SimpleGA() {
    }

    public GAResult exec(GAPParameterSet params) {

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

    if (null == params)
        throw new
NullPointerException("Parameters to a GA may not
be null");

    int age = 0;
    Population pop =
createInitialPopulation(params);
    FittestIndividualResult result =
(FittestIndividualResult) createResult();
    for (int i = 0; i < pop.getSize();
updateIndividualFitness(pop.getMember(i++), pop,
age, params));
    checkForBetterResult(result, pop, params);
    notifyInitialisationDone(pop, age, result,
params);

    while (! terminationConditionApplies(pop,
age, result, params)) {

        Individual best =
result.getFittestIndividual();

        Population nextPop =
generateNextPopulation(pop, age, result, params);

        pop = nextPop;
        age++;
        notifyGenerationChanged(pop, age,
result, params);

        if (result.getFittestIndividual() !=
best)
            notifyFoundNewResult(pop, age,
result, params);
    }

    notifyTerminationConditionApplies(pop, age,
result, params);

    return result;
}

protected Population
generateNextPopulation(Population oldPop, int
age,
                                GAResult
result, GAPParameterSet params) {
    FittestIndividualResult res =
(FittestIndividualResult) result;
    Population newPop =
createEmptyPopulation(params);
    while (newPop.getSize() <
params.getPopulationSize()) {

        Individual [] parents =
selectForReproduction(oldPop, params);
        notifySelectedForReproduction(parents,
oldPop, age, result, params);

        Individual [] children =
haveSex(parents, params);
        for (int i = 0; i < children.length;
i++) {
            if (null != children[i].getFitness())
                continue;
            updateIndividualFitness(children[i],
oldPop, age, params);
            if
(children[i].getFitness().isBetter(res.getBestFit
ness()))

                res.setFittestIndividual(children[i]);
        }
    }
}

```

```

        notifyReproduced(children, parents, oldPop,
age, result, params);
        newPop.addAll(children);
    }

    return newPop;
}

protected GAResult createResult() {
    return new FittestIndividualResult();
}

protected boolean checkForBetterResult(GAResult
oldResult, Population newPop, GAPParameterSet params) {

    FittestIndividualResult result =
(FittestIndividualResult) oldResult;
    Fitness best = result.getBestFitness();

    final int size = newPop.getSize();
    for (int i = 0; i < size; i++) {
        Fitness f = newPop.getMember(i).getFitness();
        if (f.isBetter(best)) {
            best = f;
        }
    }

    result.setFittestIndividual(newPop.getMember(i));
}

return (result.getBestFitness() != best);
}

protected Population
createInitialPopulation(GAPParameterSet params) {
    Population pop = createEmptyPopulation(params);
    while (pop.getSize() <
params.getPopulationSize()) {
        Individual ind;
        try {
            ind =
params.getIndividualsFactory().createRandomIndividual(p
arams);
        } catch (OutOfMemoryError e) {
            e.printStackTrace();
            throw new RuntimeException("p-size=" +
pop.getSize() + " // " + e.getMessage());
        }
        try {
            pop.add(ind);
        } catch (OutOfMemoryError e) {
            e.printStackTrace();
            throw new RuntimeException("p-size=" +
pop.getSize() + " // " + e.getMessage());
        }
    }
    return pop;
}

protected Population
createEmptyPopulation(GAPParameterSet params) {
    Population pop = new
SimpleCollectionOfIndividuals();
    return pop;
}

protected boolean
terminationConditionApplies(Population pop, int genNum,
GAResult result,
GAPParameterSet params) {
    return genNum >= params.getMaxGenerationNumber();
}

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

protected Individual []
selectForReproduction(Population pop,
GAPParameterSet params) {

    int selCount =
params.getReproductionAlgorithm().getRequiredNumber
ofParents();
    if (0 > selCount)
        selCount = 2;

    SelectionAlgorithm selector =
params.getSelectionAlgorithm();
    Individual [] selection =
selector.select(pop, selCount, params);

    return selection;
}

protected Individual [] haveSex(Individual []
parents, GAPParameterSet params) {
    ReproductionAlgorithm cosyPlace =
params.getReproductionAlgorithm();
    Individual [] oops =
cosyPlace.reproduce(parents, params);
    return oops;
}

protected void
updateIndividualFitness(Individual indiv,
Population pop,
                        int genNum,
GAPParameterSet params) {
    FitnessEvaluationAlgorithm tester =
params.getFitnessEvaluationAlgorithm();
    Fitness fitness =
tester.evaluateFitness(indiv, genNum, pop,
params);
    indiv.setFitness(fitness);
    updateFitnessCalculated(indiv, pop, genNum,
params);
}

public boolean addHook(SimpleGAHook hook) {
    if (null == hook)
        return false;
    else if (null == this.hooks)
        this.hooks = new ArrayList();
    else if (this.hooks.contains(hook))
        return false;
    this.hooks.add(hook);
    return true;
}

public boolean removeHook(SimpleGAHook hook) {
    if (null == hook)
        return false;
    else if (null == this.hooks)
        return false;
    else if (!this.hooks.contains(hook))
        return false;
    this.hooks.remove(hook);
    if (0 == this.hooks.size())
        this.hooks = null;
    return true;
}

protected void
notifyInitialisationDone(Population pop, int age,
GAPParameterSet params) {
    if (!params.getUseMainAlgorithmHooks())
        return;
    if (null == hooks)
        return;

    for (Iterator hook = hooks.iterator();
hook.hasNext();
        ((SimpleGAHook)
hook.next()).initialisationDone(this, pop, age, result,
params)
        );
    }

    protected void notifyFoundNewResult(Population pop,
int age,
                                        GAResult result,
GAPParameterSet params) {
        if (!params.getUseMainAlgorithmHooks())
            return;
        if (null == hooks)
            return;
        for (Iterator hook = hooks.iterator();
hook.hasNext();
            ((SimpleGAHook)
hook.next()).foundNewResult(this, pop, age, result,
params)
            );
        }

    protected void notifyGenerationChanged(Population
pop, int age,
                                        GAResult result,
GAPParameterSet params) {
        if (!params.getUseMainAlgorithmHooks())
            return;
        if (null == hooks)
            return;
        for (Iterator hook = hooks.iterator();
hook.hasNext();
            ((SimpleGAHook)
hook.next()).generationChanged(this, pop, age, result,
params)
            );
        }

    protected void
notifyTerminationConditionApplies(Population pop, int
age,
                                    GAResult result,
GAPParameterSet params) {
        if (!params.getUseMainAlgorithmHooks())
            return;
        if (null == hooks)
            return;
        for (Iterator hook = hooks.iterator();
hook.hasNext();
            ((SimpleGAHook)
hook.next()).terminationConditionApplies(this, pop,
age, result, params)
            );
        }

    protected void
notifySelectedForReproduction(Individual []
selectedParents, Population pop,
int age, GAResult
result, GAPParameterSet params) {
        if (!params.getUseMainAlgorithmHooks())
            return;
        if (null == hooks)
            return;
        for (Iterator hook = hooks.iterator();
hook.hasNext();
            ((SimpleGAHook)
hook.next()).selectedForReproduction(this,
selectedParents,
                                        POP,
age, result, params)
            );
        }
}

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

protected void notifyReproduced(Individual []
children, Individual [] parents, Population pop,
int age, GAResult
result, GAPParameterSet params) {
    if (!params.getUseMainAlgorithmHooks())
        return;
    if (null == hooks)
        return;
    for (Iterator hook = hooks.iterator();
hook.hasNext();
        ((SimpleGAHook)
hook.next()).reproduced(this, children, parents,
pop, age,
result, params)
    );
}

protected void
updateFitnessCalculated(Individual updated,
Population pop,
int age,
GAPParameterSet params) {
    if (!params.getUseMainAlgorithmHooks())
        return;
    if (null == hooks)
        return;
    for (Iterator hook = hooks.iterator();
hook.hasNext();
        ((SimpleGAHook)
hook.next()).fitnessCalculated(this, updated,
pop,
age, params)
    );
}

```

### ElitistGA.java

```

package com.gregPaperin.ga.multiplexer;

import
com.gregPaperin.ga.multiplexer.fitnessSelection.A
bsoluteFitnessIndividualComparator;
import
com.gregPaperin.ga.simpleImplementation.FittestIn
dividualResult;
import
com.gregPaperin.ga.simpleImplementation.ReusableS
impleGA;
import com.gregPaperin.ga.definitions.*;
import java.util.Arrays;

public class ElitistGA extends ReusableSimpleGA {

    private double eliteProportion =
0.1;//propotion
    private double badProportion = 0;

    public ElitistGA(double eliteProportion) {
        setEliteProportion(eliteProportion);
    }

    public ElitistGA(double eliteProportion,
double badProportion) {
        setEliteProportion(eliteProportion);
        setBadProportion(badProportion);
    }

    public ElitistGA(GAPParameterSet parameters,
double eliteProportion) {
        super(parameters);

```

```

        setEliteProportion(eliteProportion);
    }

    public ElitistGA(GAPParameterSet parameters, double
eliteProportion, double badProportion) {
        super(parameters);
        setEliteProportion(eliteProportion);
        setBadProportion(badProportion);
    }

    public double getEliteProportion() {
        return this.eliteProportion;
    }

    public void setEliteProportion(double
eliteProportion) {
        if (eliteProportion < 0 || 1 < eliteProportion)
            throw new IllegalArgumentException("Elite
proportion must be in [0, 1]");
        this.eliteProportion = eliteProportion;
    }

    public double getBadProportion() {
        return this.badProportion;
    }

    public void setBadProportion(double badProportion) {
        if (badProportion < 0 || 1 < badProportion)
            throw new IllegalArgumentException("Bad
proportion must be in [0, 1]");
        this.badProportion = badProportion;
    }

    protected Population
generateNextPopulation(Population oldPop, int age,
GAResult result,
GAPParameterSet params) {

        FittestIndividualResult res =
(FittestIndividualResult) result;
        Population newPop =
createEmptyPopulation(params);
        final IndividualsFactory fact =
params.getIndividualsFactory();

        // Cut bad:

        Individual [] pop = oldPop.getAllMembers();
        Arrays.sort(pop, new
AbsoluteFitnessIndividualComparator());
        int cutSize = (int) ((double) pop.length * (1.0 -
badProportion));
        Individual [] cutPop = new Individual[cutSize];
        System.arraycopy(pop, pop.length - cutSize,
cutPop, 0, cutSize);
        // the population we want is cutPop
        // Copy elite:

        int eliteSize = (int) ((double)
params.getPopulationSize() * getEliteProportion());
        int p = cutSize - 1;
        while (newPop.getSize() < eliteSize) {
            Individual kid =
fact.createSpecificIndividual(cutPop[p], params);
            kid.setFitness(cutPop[p].getFitness());
            newPop.add(kid);
            if (--p < 0)
                p = cutSize - 1;
        }

        // Copy rest:

        while (newPop.getSize() <
params.getPopulationSize()) {

```

Greg Paperin. Alex Michalas.  
Evolving a Multiplexer using Genetic Programming.

---

```

        Individual [] parents =
selectForReproduction(oldPop, params);
        notifySelectedForReproduction(parents,
oldPop, age, result, params);

        Individual [] children =
haveSex(parents, params);
        for (int i = 0; i < children.length;
i++) {
            if (null != children[i].getFitness())
                continue;
            updateIndividualFitness(children[i],
oldPop, age, params);
            if
(children[i].getFitness().isBetter(res.getBestFit
ness()))

                res.setFittestIndividual(children[i]);
        }

        notifyReproduced(children, parents,
oldPop, age, result, params);
        newPop.addAll(children);
    }

    return newPop;
}
}

```

**MultiplexerEvolution.java**

```

package com.gregPaperin.ga.multiplexer;

import com.gregPaperin.ga.definitions.*;
import com.grepPaperin.ga.simpleImplementation.*;
import
com.gregPaperin.ga.simpleImplementation.reproduct
ion.*;
import
com.gregPaperin.ga.simpleImplementation.fitnessSe
lection.*;
import
com.gregPaperin.ga.simpleImplementation.hooks.*;
import
com.gregPaperin.ga.multiplexer.fitnessSelection.*
;
import
com.gregPaperin.ga.multiplexer.representation.*;
import
com.gregPaperin.ga.multiplexer.representation.nod
es.*;
import
com.gregPaperin.ga.multiplexer.reproduction.*;

public class MultiplexerEvolution {

    public MultiplexerEvolution() {}

    public void exec() {

        GAParameterSet params = new
DefaultParameterSet();
        params.setPopulationSize(1500);
        Multiplexer multiplexer = new
Multiplexer(8);
        params.setFitnessEvaluationAlgorithm(new
MultiplexerFitness(multiplexer, 2));
        CombinedReproductionAlgorithm repAlg = new
CombinedReproductionAlgorithm();
        repAlg.insertReproductionAlgorithm(0, new
FunctionTreeXOver(0.7));

```

```

        repAlg.insertReproductionAlgorithm(1, new
FunctionTreeMutation(0.1));
        params.setReproductionAlgorithm(repAlg);
        params.setMaxGenerationNumber(200);
        params.setSelectionAlgorithm(new
TournamentSelection(5, 0.9));
            //new RouletteWheelSelection(-5));
        BooleanFormulaTreeFactory factory = new
BooleanFormulaTreeFactory();
        factory.setAllowConstants(false);
        factory.setMaxTreeDepth(7);
        factory.setNumberOfParameters(11);
        final int attempts = 1;

        factory.setAllowedNodeTypes(new Class[] {
TerminalNode.class,
                                ANDNode.class,
                                ORNode.class,
                                NOTNode.class,
                                //NANDNode.class,
                                //NORNode.class,
                                //XORNode.class,
                                //IFNode.class,
                                //EQUIVNode.class,
                                //IMPLNode.class,
                                });

        params.setIndividualsFactory(factory);

        //ReusableSimpleGA ga = new
ReusableSimpleGA(params);
        ElitistGA ga = new ElitistGA(params, 0.2, 0.17);
// params, elite, bad
        AnalysisHook hook = new AnalysisHook();
        hook.setLogStream(System.out);
        hook.setUpdateDelay(500);
        ga.addHook(hook);
        GAResult [] allResults = new GAResult[attempts];

        for (int i = 0; i < attempts; i++) {

            System.out.println("\n ===== STARTING RUN
" + (i+1) + ". =====\n");
            hook.reset();

            GAResult result = new
FittestIndividualResult();
            try {
                result = ((ReusableSimpleGA) ga).exec();
            } catch (OutOfMemoryError e) {
                e.printStackTrace();
            }

            System.out.println("\nDONE.\n");
            System.out.println("Total fitness evaluations:
" + hook.getFitnessCalculations());
            System.out.println("Result is: " + result);
            System.out.println("(Fitness: " +
((FittestIndividualResult) result).getBestFitness() +
")");
            allResults[i] = result;

            factory.setAllowConstants(!factory.getAllowConstants
());
        }

        System.out.println("\nALL DONE.\n");
        for (int i = 0; i < attempts; i++) {
            System.out.println("Result " + i + " is: " +
allResults[i]);
            System.out.println("(Fitness " + i + " is:" +
((FittestIndividualResult)
allResults[i]).getBestFitness() + ")");
        }

```

```
    }  
    public static void main(String [] unusedArgs)  
    {  
        MultiplexerEvolution multiplexerEvolution =  
new MultiplexerEvolution();  
        multiplexerEvolution.exec();  
    }  
}
```