GEVA - Grammatical Evolution in Java (v2.0)

Michael O'Neill, Erik Hemberg, Conor Gilligan,
Eliott Bartley, James McDermott, Anthony Brabazon

Natural Computing Research & Applications Group
University College Dublin
Ireland

June 13, 2011

Abstract

GEVA is an open source implementation of Grammatical Evolution in Java developed at UCD's Natural Computing Research & Applications group. As well as providing the characteristic genotype-phenotype mapper of GE a search algorithm engine and a simple GUI are also provided. A number of sample problems and tutorials on how to use and adapt GEVA have been developed.

Contents

1 Introduction

Grammatical Evolution in Java (GEVA) was developed at UCD's Natural Computing Research & Applications group[1] [14]. It is an open source implementation of Grammatical Evolution (GE) [30, 28, 29, 39] released under GNU GPL v3.0, which provides a search engine framework in addition to a simple GUI and the genotype-phenotype mapper of GE.

This report serves as an introduction to the GEVA software providing guidelines on its installation and use. The software comes in two main parts, a simple GUI and the main GEVA package. The GUI provides a mechanism to explore the software and understand how it operates but use of the underlying GEVA can be accessed through a command line interface to allow for scripting. In this release some simple demonstration problems are provided to assist the user to gain an understanding of the code while reading the accompanying tutorials which are provided in this document and on the code's website [14].

Following a brief introduction to Grammatical Evolution in Section 2, we describe the design of GEVA in Section 3, provide installation instructions in Section 4, and an introductory tutorials on its use in Sections 6, 7, 8, 9, followed by more advanced programming tutorials in Section 10.

2 Grammatical Evolution

Grammatical Evolution (GE) (e.g., [30, 28, 4, 12, 13, 29, 39, 31, 19, 7, 10, 1, 32]) is a grammar-based form of Genetic Programming [38]. It marries principles from molecular biology to the representational power of formal grammars. GE's rich modularity gives a unique flexibility, making it possible to use alternative search strategies, whether evolutionary, deterministic or some other approach, and to radically change its behaviour by merely changing the grammar supplied. As a grammar is used to describe the structures that are generated by GE, it is trivial to modify the output structures by simply editing the plain text grammar. This is one of the main advantages that makes the GE approach so attractive. The genotype-phenotype mapping also means that instead of operating exclusively on solution trees, as in standard GP, GE allows search operators to be performed on the genotype (e.g., integer or binary chromosomes), in addition to partially derived phenotypes, and the fully formed phenotypic derivation trees themselves.

2.1 The Grammar

When tackling a problem with GE, a suitable BNF (Backus Naur Form) grammar definition must initially be defined. The BNF can be either the specification of an entire language or, perhaps more usefully, a subset of a language geared towards the problem at hand.

[1] http://ncra.ucd.ie

In GE, a BNF definition is used to describe the output language to be produced by the system. BNF is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of *terminals*, which are items that can appear in the language, e.g. binary boolean operators `and`, `or`, `xor`, and `nand`, unary boolean operators `not`, constants, `true` and `false` etc. and *non-terminals*, which can be expanded into one or more terminals and non-terminals.

For example the grammar below can be used to generate boolean expressions, and `<expr>` can be transformed into one of three rules. It can become either `( <expr> <biop> <expr> )`, `<uop> <expr>`, or `<bool>`. A grammar can be represented by the tuple $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. When there are a number of productions that can be applied to one element of N the choice is delimited with the '`|`' symbol. For example

```
N = { <expr>, <biop>, <uop>, <bool> }
T = { and, or, xor, nand, not,
      true, false, (, ) }
S = { <expr> }
```

And P can be represented as:

```
(A) <expr> ::= ( <expr> <biop> <expr> )
             | <uop> <expr>
             | <bool>

(B) <biop> ::= and
             | or
             | xor
             | nand

(C) <uop> ::= not

(D) <bool> ::= true
             | false
```

The code produced will consist of elements of the terminal set T. The grammar is used in a developmental approach whereby the evolutionary process evolves the production rules to be applied at each stage of a mapping process, starting from the start symbol, until a complete program is formed. A complete program is one that is comprised solely from elements of T.

In GEVA the BNF definition is comprised entirely of the set of production rules, with the definition of terminals and non-terminals implicit in these rules. The first non-terminal symbol is by default the start symbol.

As the BNF definition is a plug-in component of the system, it means that GE can produce code in any language thereby giving the system a unique flexibility. For the above BNF grammar, Table 1 summarises the production rules and the number of choices associated with each.

Table 1: The number of choices available from each production rule.

Rule Number	Choices
A	3
B	4
C	1
D	2

2.2 Genotype-Phenotype Mapping Process

The genotype is used to map the start symbol as defined in the Grammar onto terminals by reading codons to generate a corresponding integer value, from which an appropriate production rule is selected by using the following mapping function:

$$Rule = c \mod r$$

where `c` is the codon integer value, and `r` is the number of rule choices for the current non-terminal symbol.

Consider the following rule from the given grammar, i.e., given the non-terminal `<boolop>`, which describes the set of boolean operators that can be used, there are four production rules to select from. As can be seen, the choices are effectively labelled with integers counting from zero.

```
(B) <boolop> ::= and      (0)
               | or       (1)
               | xor      (2)
               | nand     (3)
```

If we assume the codon being read produces the integer 6, then

$$6 \mod 4 = 2$$

would select rule (2) `xor`. That is, `<boolop>` is replaced with `xor`. Each time a production rule has to be selected to transform a non-terminal, another codon is read. In this way the system traverses the genome.

During the genotype-to-phenotype mapping process, it is possible for individuals to run out of codons, and in this case the *wrap* operator is applied

which results in returning the codon reading head back to the first codon in the individual. As such codons are reused when wrapping occurs. This is quite an unusual approach in evolutionary algorithms as it is entirely possible for certain codons to be used two or more times. This technique of wrapping the individual draws inspiration from the gene-overlapping phenomenon that has been observed in many organisms [24]. GE works with or without wrapping, and wrapping has been shown to be useful on some problems [28], however, it does come at the cost of introducing functional dependencies between codons that would not otherwise arise.

In GE each time the same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of a different production rule. This feature is referred to as *intrinsic polymorphism*. What is crucial however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is the case because the same choices are made each time. It is possible that an incomplete mapping could occur, even after several wrapping events, and typically in this case the mapping process is aborted and the individual in question is given the lowest possible fitness value. The selection and replacement mechanisms then operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules repeatedly. For example, consider an individual with three codons, all of which specify rule 0 from below.

```
(A) <expr> ::= (<expr> <biop> <expr>) (0)
             | <uop> <expr>           (1)
             | <bool>                 (2)
```

Even after wrapping, the mapping process would be incomplete and would carry on indefinitely unless terminated. This occurs because the nonterminal `<expr>` is being mapped recursively by production rule 0, i.e., it becomes `( <expr> <biop> <expr> )`. Therefore, the leftmost `<expr>` after each application of a production would itself be mapped to a
`( <expr> <biop> <expr> )`, resulting in an expression continually growing as follows: `( ( <expr> <biop> <expr> ) <biop> <expr> )` followed by
`( ( ( <expr> <biop> <expr> ) <biop> <expr> )`
`<biop> <expr> )`
and so on.
Such an individual is dubbed invalid as it will never undergo a complete mapping to a set of terminals. For this reason an upper limit on the number of wrapping events that can occur is imposed. During the mapping process therefore, beginning from the left hand side of the genome codon integer values are generated and used to select rules from the BNF grammar, until one of the following situations arise:

1. A complete program is generated. This occurs when all the non-terminals

in the expression being mapped are transformed into elements from the terminal set of the BNF grammar.

2. The end of the genome is reached, in which case the *wrapping* operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue, unless an upper threshold representing the maximum number of wrapping events has occurred during this individual's mapping process.

3. In the event that a threshold on the number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual is assigned the lowest possible fitness value.

To reduce the number of invalid individuals being passed from generation to generation various strategies can be employed. Strong selection pressure could be applied, for example, through a steady state replacement. One consequence of the use of a steady state method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals. Alternatively, a repair strategy can be adopted, which ensures that every individual results in a valid program. For example, in the case that there are non-terminals remaining after using all the genetic material of an individual (with or without the use of wrapping) default rules for each non-terminal can be pre-specified that are used to complete the mapping in a deterministic fashion. Another strategy is to remove the recursive production rules that cause an individual's phenotype to grow, and then to reuse the genotype to select from the remaining non-recursive rules. Also, by adopting a ramped-half-and-half initialisation based on derivation trees, it is possible to ensure that all genotypes in the initial population are completely mapped sentences in the target language. From this start state it can be difficult for invalid genotypes to propagate during a run with strong selection pressure, and also genetic operators which manipulate the derivation tree [18] can be used to ensure the generation of completely mapped sentences.

There have been a number of extensions to GE in recent years in terms of the grammars adopted, the search engine and search operators employed, and even variants on the mapping process itself (e.g., see the following sources [31, 4, 13, 32, 12, 19, 5, 6, 20, 34, 21, 35, 26]). [2]

[2] Additional information on Grammatical Evolution is available from http://www.grammatical-evolution.org

3 GEVA Design Overview

The software comes in two main components, namely

1. GUI
2. GEVA

The GUI (`GUI.jar`) provides a simple interface to assist in an initial exploration of the software. In reality the GUI simply invokes the main GEVA package (`GEVA.jar`) using the selected parameter set. The more advanced user can simply skip the GUI and work directly with the command line interface with `GEVA.jar`, thereby accessing the full unrestricted suite of features that exist in the current release. The GUI also provides a simple graphing utility, which is provided for informational purposes to aid the user in understanding the behaviour of GEVA.

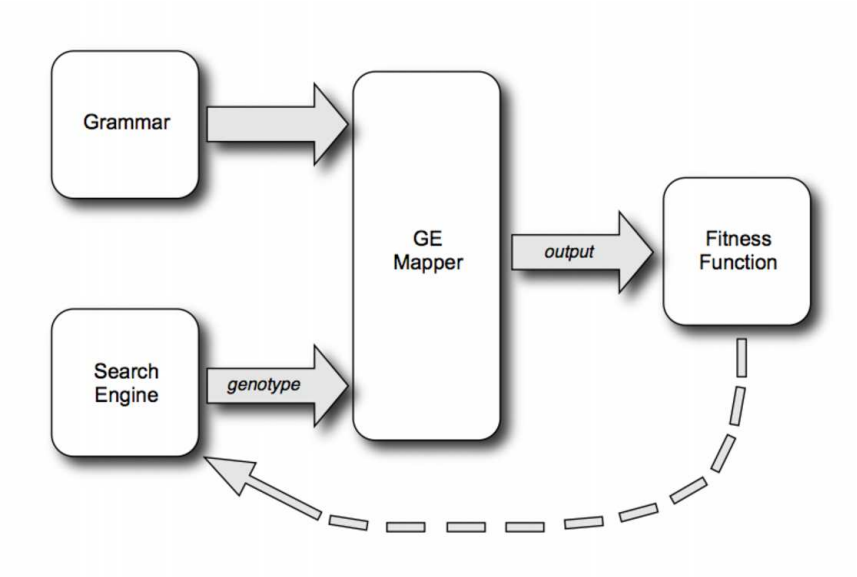

Figure 1: Modular components of Grammatical Evolution.

An overview of the major components of Grammatical Evolution are provided in Fig. 1.

4 Installation Instructions & Running out-of-the-box

1. A working installation of Java is required (at least Java 1.5).
2. Download the latest version of GEVA [14] from `http://ncra.ucd.ie/geva`.
3. Unzip the distribution, and navigate into the GEVA-v2.0/ directory.
4. To run out-of-the-box (i.e., in GUI mode) either
   (a) Double-click the `GEVA_GUI.jar` or
   (b) in a shell type `java -jar GEVA_GUI.jar`

Figure 2: The GEVA Splash screen.

A GEVA splash screen will be displayed first (see Fig. 2) followed by the main GUI window which allows you to select the problem and associated parameters. A number of sample problems are provided in the release. Running out-of-the-box GEVA uses the HelloWorld example problem by default. A screenshot of what the user sees is presented in Fig. 3. In this problem the goal is to match all the characters in a target character string ("geva").

To run this default problem with the example settings simply click on `Run` and the following screen will appear (see Fig. 4). You will see a fitness plot generated as the run progresses, and in this problem a solution tends to be found quite quickly so more than likely you will just see the fitness plot graph of the finished run. It is possible to modify the graph by checking the `Visible`

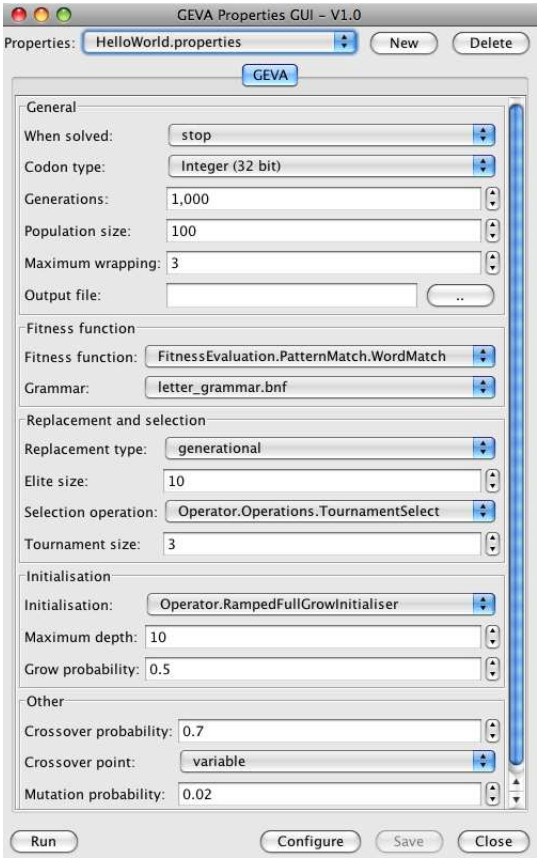

Figure 3: GEVA GUI screenshot for the default HelloWorld problem. The goal is to rediscover the string "*geva*"

box beside each data item, and it is also possible to change the colour associated with each data item. The GUI allows the user to plot a number of additional GE attributes by selecting the appropriate Tab (`Codon`, `Invalids` and `Other` are the current options). Under `Codon` it is possible to plot the average physical length of each individual in the population in terms of number of codons, and it is also possible to plot the average number of `Used`/expressed codons (i.e., codons that are actually used during the generation of the phenotype/solution). At any time it is possible to scale the graphs by right-clicking and dragging the graph along either axis. If you wish to save the resulting graph click on the `Save Image` button and a selection of file formats may be chosen from.

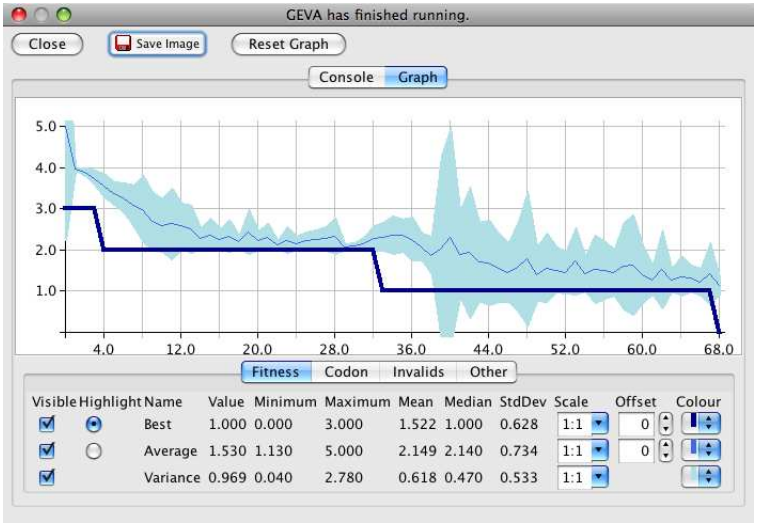

Figure 4: GEVA GUI screen-shot of the graphing feature for the default HelloWorld problem.

If you click on the `Console` tab you will see the following window, Fig. 5. This window simply captures the text dump that `GEVA.jar` generates upon execution and includes a print of the parameter settings, and the phenotype and fitness of the best solution ever found along with the total time GEVA was running for (in milliseconds) and the total number of generations.

4.1 Next Steps

A description of the demonstration problems is presented in Section 6. It is recommended that the new user moves directly to this section (Tutorial 0). To gain a deeper understanding of how to modify the grammar (Tutorial 1), add your own simple problem (Tutorial 2), and use the command line interface (Tutorial 3) the more adventurous user can skip to Sections 7, 8 and 9 respectively. If you want to jump straight into the deep end and get your hands dirty coding your

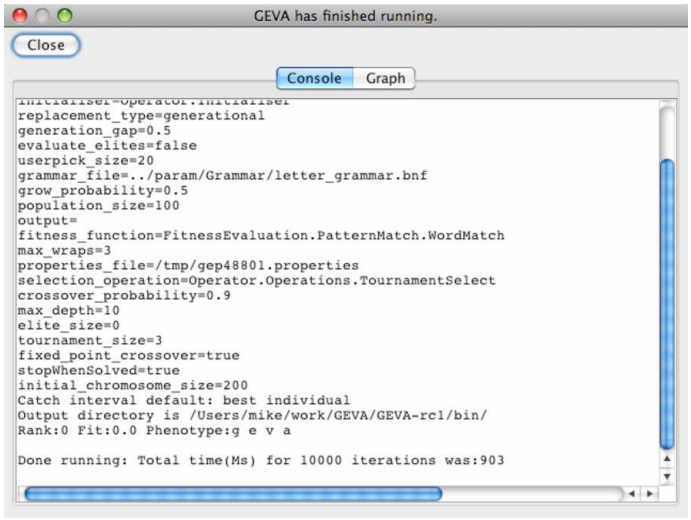

Figure 5: GEVA GUI screen-shot of the console feature for the default HelloWorld problem.

own search engine the more advanced user can visit the Advanced Tutorials in Section 10. Section 5 follows, which details the contents of the distribution.

5 Distribution Contents

In the main distribution directory you will see the following files:

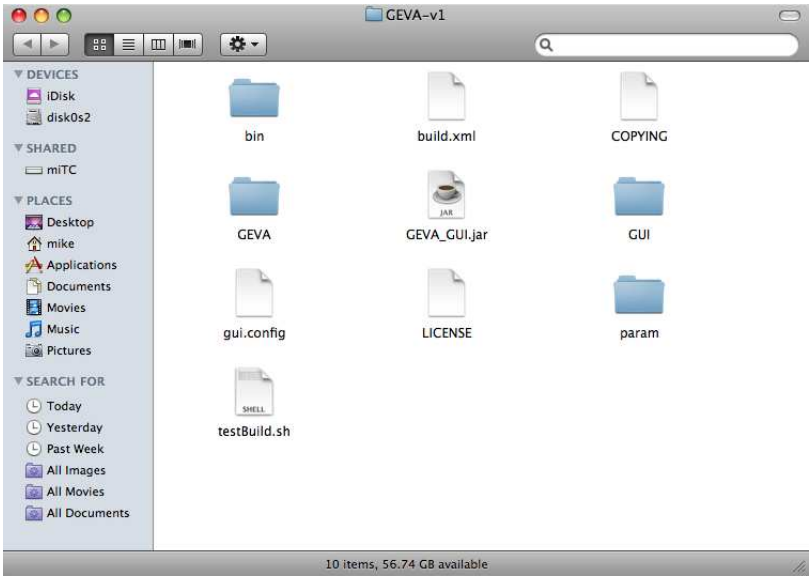

Figure 6: GEVA main directory listing.

`bin/` contains the executables `GEVA.jar` and `GUI.jar` and associated libraries.

`build.xml` is the build file used by `ant` to rebuild the entire distribution after modifying any of the source files. Seperate build files are available for the GUI and GEVA components in their respective directories.

`COPYING` contains the text of the GNU GPL version 3.

`GEVA/` contains the source code, documentation, and build files for the GEVA code.

`GEVA_GUI.jar` is the main point of entry to execute the GUI version of GEVA. It invokes `GUI.jar` which is contained in the `bin/` directory.

`GUI/` contains the source code, documentation, and build files for the GUI code.

`gui.config` is used to initialise the configuration of the GUI so that it knows where `GEVA.jar` and the variours property and grammar files are located.

`LICENSE` provides the Copyright notice and Licence information for the distribution.

`param/` contains two directories, `Grammar/` which contains the grammar files for the demonstration problems, and the corresponding example properties

files are contained in `Parameters/`. Also contained here are two additional configuration files for the GUI mode. These files tell the GUI what parameter and grammar files to associate with each demonstration problem (`ff.config`), and `graph.config` sets up some default settings for the graphing utility.

6 Tutorial 0: The Demo Problems

From the GUI using the drop down `Properties` menu item you can change the problem used. The appropriate grammar, fitness function and example parameter settings are automatically set for you when you change between a problem using the `Properties` menu.

When `GEVA_GUI.jar` is run for the first time it runs the `HelloWorld` problem by default. A description of this and the other demonstration problems now follow.

6.1 HelloWorld

In this simple problem the goal is to match the target string "geva". Fitness is simply a count of the number of characters of the candidate solution which match the target string. As we are using search based upon minimisation, smaller fitness values are better, so we adjust the match count by subtracting it from the total character count (`n`) of the target string.

$$Fitness = targetStringLength - \sum_{i=1}^{n}(targetChar[i] - predictedChar[i])$$

The example grammar (`letter_grammar.bnf`) adopted simply generates strings of characters. The grammar is as follows:

```
<string> ::= <letter>|<letter><string>
<letter> ::= <vowel>|<consonant>|<space>
<space> ::= _
<vowel> ::= a|o|u|e|i
<consonant> ::= q|w|r|t|y|p|s|d|f|g|h|j|k|l|z|x|c|v|b|n|m
```

A candidate solution is constructed beginning from the embryonic `<string>` symbol. A `<string>` can be replaced with either a `<letter>` or with `<letter><string>`. This means an output string can be anything from a single character to multiple characters.

The `<letter>` symbol is used to determine which characters are included in the output string. These can be one of three items, a `<vowel>`, a `<consonant>` or a `<space>`. A space is represented by the underscore character _, and this is the only possible replacement for a `<space>` symbol. It is therefore not necessary to consult the genome for a codon value in that instance. There are however five possible replacement choices for a `<vowel>` and twenty one choices for a `<consonant>`.

Therefore, evolution has to search for a string of the correct length, and a string that contains the right sequence of characters to match the target string ("`geva`"). Click `Run` to see how GEVA performs on this problem.

6.2 L-systems

To run the L-system problem select the `LSystem.properties` option from the `Properties` menu, and the example settings for an L-system demo are provided. This is an interactive form of GE that presents the user with the current population of L-systems (with duplicate phenotypes overlaid to save space). The user then determines which L-systems will become parents to create the next generation and/or can decide to purge individuals from the population (see Fig. 7). To create the next generation, select only those individuals that you want to become a `Parent` or to `Purge`, and click on `Generate`. Individuals that are not selected for either becoming a parent or purging are not used in the creation of the next generation of solutions. `Purge` forces that individual to be deleted from the population.

To find out more about L-systems the reader is referred to Prusinkiewicz's Algorithmic Beauty of Plants [37].

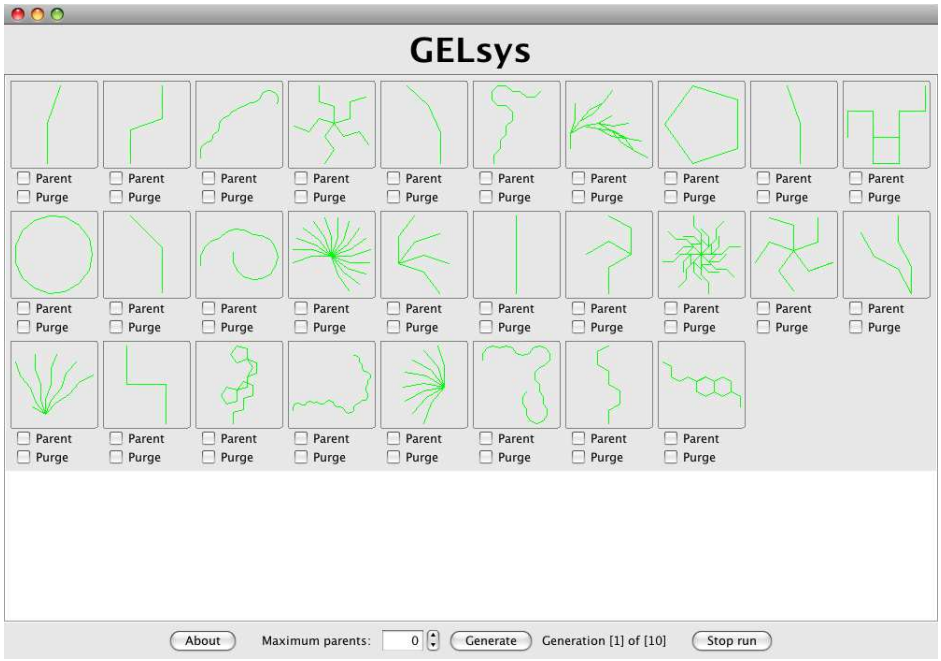

Figure 7: GEVA GUI screenshot of the L-system demo problem.

Evolution in this case is driven by the user where the user directly acts as the selection operator determining which L-systems becomes parents to create the next population of candidate solutions. The default grammar adopted is `LSys_1.bnf` and is detailed below:

```
<Sys>    ::=<n><angle><expr>
<n>      ::=2|3|4
```

```
<angle> ::= 20.0|22.5|25.0|27.5
<expr>  ::= <expr> <op> <expr> | [<expr> <op> <expr>] | <var>
<op>    ::= +|-
<var>   ::= F
```

The start symbol from which the L-system grammar is constructed is `<Sys>`. Evolutionary search determines the depth of expansion (`<n>`), the angle (`<angle>`) and production rules of each L-system. In this simple case we have restricted the depth of expansion to be either 2, 3 or 4, and limited the angles to four possibilities (20.0, 22.5, 25.0 and 27.5 degrees). When the symbol `F` is encountered a line is drawn. The symbols corresponding to `<op>` determine the angle the next line is drawn at.

A second L-system grammar file, `LSysEx_1.bnf`, is provided as an example. This is similar to the grammar adopted in the generation of the NCRA group logo [34]. To activate the use of this grammar simply select it from the `Grammar` drop down menu in the GUI (see the menu item highlighted in blue in Fig. 8).

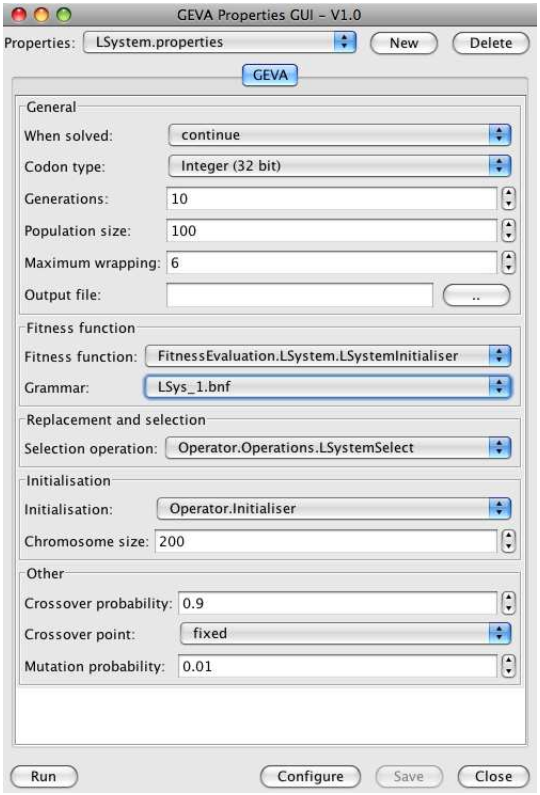

Figure 8: Parameter settings of the L-system demo problem with the grammar choice sub-menu highlighted in blue. `LSys_1.bnf` is currently in use. Select `LSysEx_1.bnf` to create slightly more complicated L-systems.

6.3 Paint

The `Paint` toy problem (`Paint.properties`), is provided which attempts to evolve the most *complex* colour picture possible where complexity refers to diversity of pixels. A screen dump of `Paint` in action is given in Fig. 9. The left-hand side of the picture continuously displays each individual in the population with the right-hand side presenting the most *complex* picture found to-date.

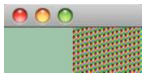

Figure 9: GEVA GUI screen-shot of the Paint demo problem.

The grammar adopted for the Paint problem is `paint.bnf` and is as follows:

```
<paint> ::= <intensity> <intensity> <intensity>
          | <paint> <intensity> <intensity> <intensity>
<intensity> ::= <digit><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Each `<digit>` represents a color, and each painting is comprised of at least six colours.

6.4 Even-5-Parity

This is the classic benchmark problem in which evolution attempts to find the five input even-parity boolean function. The grammar adopted here is `efp_grammar_gr.bnf`:

```
<prog> ::= <expr>
<expr> ::= <expr> <op> <expr> | ( <expr> <op> <expr> )
         | <var> | <pre-op> ( <var> )
<pre-op> ::= not
<op> ::= "|" | & | ^
<var> ::= d0 | d1 | d2 | d3 | d4
```

Select `EvenFiveParity.properties` from the `Properties` menu to run this problem. The optimal fitness is obtained when the correct output is generated for each of the 32 test cases.

6.5 Santa Fe Ant trail

Select `SantaFeAntTrail.properties` from the `Properties` menu. The objective here is to evolve a program to control the movement of an artificial ant on a toroidal grid of size 32 by 32 units. 89 pieces of food are located along a broken trail, and the ant has 600 units of energy to find all the food. A unit of energy is

consumed when the ant uses one of the following operations: `move()`, `right()` or `left()`. The ant also has the capability to look ahead into the square directly facing it to determine if there is food present or not. The example grammar adopted here, `sf_grammar_gr.bnf` is as follows:

```
prog> ::= <code>
<code> ::= <line> | <code> <line>
<line> ::= <condition>\n | <op>\n
<condition> ::= if(food_ahead()==1) { <opcode> } else { <opcode> }
<op> ::=  left(); | right(); | move();
<opcode> ::= <op> | <opcode> <op>
```

The ideal fitness is obtained when the ant has consumed all 89 pieces of food. In GEVA v1.2 some added trail problems were included, being the Los Altos Trail and the San Mateo Trail both of which use the Santa Fe Trail grammar above.

6.6 Symbolic Regression

Select `SymbolicRegression.properties` from the `Properties` menu. The classic quartic function is used here $x + x^2 + x^3 + x^4$ with 20 input-output test cases drawn from the range -1 to 1. Fitness is simply the sum of the errors, and an uncomplicated grammar adopted:

```
<expr> ::= ( <op> <expr> <expr> ) | <var>
<op>   ::= +|-|*
<var> ::= x0|1.0
```

In this particular instance prefix expressions are generated and then evaluated by the interpreter.

6.7 Max Problem

Select `MaxProblem.properties` from the `Properties` menu. This problem tries to generate the largest number possible from with a given terminal and functional set and with a depth limit. Despite seeming to be a simple problem the fitness landscape is deceptive and leads to suboptimal solutions. It is referenced in [40]

```
<prog> ::= <expr>
<expr> ::=  <op> <expr> <expr> | <var>
<op> ::= + | *
<var> ::= 0.5
```

Even though the problem tries to generate the largest number the fitness is still minimised

6.8 Royal Tree

Select `RoyalTree.properties` from the `Properties` menu. This problem is an adaptation of a Genetic Algorithm benchmark the Royal Road problem [42]. It is problem commonly used as a standard function for testing the effectiveness of Genetic Programming. It consists of a single base function that is specialized into as many cases as necessary, depending on the desired complexity of the resulting problem.

```
<tree> ::= <leaf> | <intnode>
<leaf> ::= x
<intnode> ::= ( <A> <tree> )
            | ( <B> <tree> <tree> )
    | ( <C> <tree> <tree> <tree> )
            | ( <D> <tree> <tree> <tree> <tree> )
<A> ::= A
<B> ::= B
<C> ::= C
<D> ::= D
```

6.9 GUI Features

In addition to changing between the different demonstration problems using the main GUI window it is also possible to:

- modify the common parameter settings for all problems including the choice of grammar,
- save parameter settings to a new file,
- delete a parameter settings file,
- or load in previously saved parameter settings.

6.10 Next Steps

The interested user can now gain a deeper understanding of how to modify the grammar (Tutorial 1), add your own simple problem (Tutorial 2), and use the command line interface (Tutorial 3) by reading Sections 7, 8 and 9 respectively.

7 Tutorial 1: Modifying Grammar Files

After modifying parameter setting using the GUI, the simplest task is to alter the grammar for one of the demonstration problems. In GUI mode it is not currently possible to directly edit the grammar files provided. To modify a grammar:

1. Open the relevant grammar file in a text editor from the `<GEVA_ROOT>/param/Grammar/` directory and be sure to save the modified file to the same location and give it a new name.
2. Return to the GUI and using the drop down menu for Grammar switch to the grammar file with the new name.
3. Run with the new grammar.

We now walk the interested reader through a specific example of altering the grammar adopted for the default HelloWorld problem.

7.1 Modifying the HelloWorld Grammar

The grammar for the HelloWorld demo problem is located in the directory called `<GEVA_ROOT>/param/Grammar/`. Open the file `letter_grammar.bnf` in your favourite text editor. You will see the following:

```
<string> ::= <letter>|<letter><string>
<letter> ::= <vowel>|<consonant>|<space>
<space> ::= _
<vowel> ::= a|o|u|e|i
<consonant> ::= q|w|r|t|y|p|s|d|f|g|h|j|k|l|z|x|c|v|b|n|m
```

By default the *Start Symbol* of the grammar is the first non-terminal symbol that appears in the file. In this case `<string>` is the Start Symbol from which the mapping process will commence. In this grammar there are 2 possible replacements for `<string>`. It can become `<letter>` or it will be replaced with `<letter><string>`. This means that an output sentence (solution) from this grammar will be comprised of one or more `<letter>`s.

In the above example a `<letter>` can be replaced with any one of `<vowel>`, `<consonant>` or `<space>`, where a `<space>` will become a terminal symbol '_', a `<vowel>` can become any one of the five terminal symbols 'a', 'o', 'u', 'e', or 'i', and finally a `<consonant>` can become any one of the 21 terminal symbols 'q', 'w', 'r', ..., 'm'.

The grammar can be easily modified such that a string could print additional characters. For example, this could be achieved by adding a new production rule for `<letter>`. The new rule might look like:

```
<letter> ::= <vowel>|<consonant>|<space>|<symbol>
```

We then need to define the terminal symbol for the non-terminal `<symbol>`. This is achieved by adding a new set of productions rule to the grammar for `<symbol>`. The new grammar could now look like:

```
<string> ::= <letter>|<letter><string>
<letter> ::= <vowel>|<consonant>|<space>|<symbol>
<space> ::= _
<vowel> ::= a|o|u|e|i
<consonant> ::= q|w|r|t|y|p|s|d|f|g|h|j|k|l|z|x|c|v|b|n|m
<symbol> ::= $ | \Euro
```

To use this new grammar, start up GEVA as before. First navigate to the main GEVA directory (`<GEVA_ROOT>/`) and then either:

1. Double-click the `GEVA_GUI.jar` or
2. in a shell type `java -jar GEVA_GUI.jar`

Now select the new grammar file from the `Grammar` drop-down menu, and click on `Run`.

7.2 Next Steps

The interested user should now move on to Tutorial 2 to learn how to adapt the existing demo problems to create your own. Tutorial 2 walks through how you can modify your GEVA distribution to include an additional bonus demo problem, Battleship.

8 Tutorial 2: Adding the Bonus Battleship Problem

The Advanced Tutorials described later explain in some detail how to build your own algorithms in GEVA. However, it is also possible (and much easier) to re-use the built-in algorithms to solve new problems. In this tutorial we will make some simple customisations to just 3 files and write a new fitness function, in order to solve a new problem: a simplified version of the game of Battleship. We will ignore questions of good style and software engineering practice.

8.1 Battleship

Battleship is a paper-and-pencil game for two players, each possessing a grid representing a map of the sea, with the player's own ships distributed throughout the grid. The aim is to fire at and sink the opponent's ships, by guessing what cells they occupy.

In this tutorial, the aim will be to evolve an optimal "firing pattern" for a fixed grid of enemy ships. The firing pattern will consist of multiple salvos, and each salvo will consist of multiple adjacent shots. This description should give a hint as to the type of grammar we will use. The aim will be to sink as many as possible of the ships using as few salvos and as few shots as possible. Each firing pattern will correspond to a single (evolutionary) individual, and after each individual is run, the grid will be reset with the ships intact and in the same places as before. Therefore, our version of Battleship will omit the in-game feedback and two-player aspects of the real game.

8.2 Fitness Function

GEVA's fitness evaluation functions are represented as classes which live in `<GEVA_ROOT>/GEVA/src/FitnessEvaluation/` (where `<GEVA_ROOT>` represents the root directory of your GEVA download). Create a new directory under that location, named `Battleship`, and copy the file `Battleship.java` in there. This file contains not only the fitness evaluation code, but also (in our case) some hard-coded game logic and a method of visualising the success of an individual. You can find the file `Battleship.java` at `http://ncra.ucd.ie/geva/Battleship.java`.

`Battleship.java` was created firstly by copying an existing fitness evaluation function, `<GEVA_ROOT>/GEVA/src/FitnessEvaluation/PatternMatch/WordMatch.java`. It may be instructive to view the differences between the two: they consist largely of logic specific to the two problem instances. The same copy-and-edit approach may be the easiest way to create new fitness functions of your own devising.

The essential parts of the fitness evaluation class are that it imports `FitnessEvaluation.FitnessFunction`, it implements `FitnessFunction`, it has a constructor, it over-rides `setProperties()`, and it returns a (`double`) fitness value from `evaluateString()`. `evaluateString()` is where the real work

happens, and its structure is shown below. The first line converts the input phenotype into a string (without adding superfluous spaces). The body of the function adds to fitness every time it detects that a ship has been hit, with a bonus for each complete sinking, and subtracts for every salvo and every shot. Note that GEVA always minimises fitness, so in our example we take a safe reciprocal (`1 / (1 + x)`).

```
public double evaluateString(Phenotype p) {
    String pS = p.getStringNoSpace();
    double fitness;
    // parse pS and evaluate...
    return fitness;
}
```

Much of `Battleship.java` is hard-coded game logic, and is written in a rather C-like style which will offend native Java programmers. `evaluateString()` contains some code for printing (on the console) a visualisation of each individual's success. The code is commented out by default, since it will slow the program down somewhat. This is regarded as a quick and dirty way of accomplishing this task.

8.3 Grammar

Have a look at some of the existing grammars, which all have the suffix `.bnf` and live in `<GEVA_ROOT>/param/Grammar/`. They are written in Backus-Naur form, as explained elsewhere. In every case, the start symbol will be that on the left-hand side of the rule on the first line in the file. In our case, as hinted above, an individual (a "firing-pattern") consists of multiple salvos separated by semi-colons, and each salvo consists of x- and y-coordinates and a size. Each of these is an integer with a limited range, as you can see in the code below. Paste it into a new file `<GEVA_ROOT>/param/Grammar/Battleship.bnf`.

```
<pattern> ::= <salvo> | <salvo>;<pattern>
<salvo> ::= <pos>,<pos>,<size>
<pos> ::= <GECodonValue[0, 9]>
<size> ::= <GECodonValue[1, 5]>
```

Note the special `GECodonValue` non-terminals in this grammar. This is an easy way to put numerical values — in this case, integer values — into the phenotype. The current codon in the genome is mapped to produce a numerical value in the range given. For example, the fourth rule above is equivalent to `<size> ::= 1|2|3|4|5`. Double values can be produced by specifying the end-points of the range with a decimal point or in scientific notation. The range may be closed (i.e. include the end-points), as above, or may be open (using round brackets, e.g. `<GECodonValue(0, 6)>`), or half-open at either end. There is also a special notation for producing a string from a given set: `<GECodonValue{x, y, z}>`.

8.4 GEVA Bureaucracy

For the Battleship problem to run from the GEVA GUI, there are two more small tasks. Open the file `<GEVA_ROOT>/param/ff.config` and add the line

```
FitnessEvaluation.Battleship.Battleship,Battleship.bnf
```

just before the `-jar` line near the end. This tells the GEVA GUI that our new fitness function exists and corresponds to our new grammar.

Next, copy the file `<GEVA_ROOT>/param/Parameters/HelloWorld.properties` to a new file in the same directory, `Battleship.properties`. Change the `grammar_file` and `fitness_function` entries as follows:

```
grammar_file=../param/Grammar/Battleship.bnf
fitness_function=FitnessEvaluation.Battleship.Battleship
```

Also change the number of generations and population size as follows, so that initial experiments with Battleship will be quite short:

```
generations=20
population_size=20
```

This new `.properties` file gives all the configuration necessary for a single experiment with GEVA. When we start GEVA, this set of properties will be available as a preset experiment in the GUI (it can also be loaded as a preset on the command-line).

8.5 Building and running

As you have modified some of the source code you now need to rebuild the jar files. To do this go to `<GEVA_ROOT>/` and type `ant`

```
<GEVA_ROOT>$ ant
```

Note that you need to have `ant` [2] installed to do this! Otherwise you could use your favourite IDE to rebuild the jar files. The `build.xml` file is used by `ant` to recompile the modified source files and build fresh jar files. The `<GEVA_ROOT>/build.xml` file rebuilds all the jar files.

You should see a successful build happening. Java `jar` files will be placed in `<GEVA_ROOT>/GEVA_GUI.jar`, `<GEVA_ROOT>/bin/GEVA.jar` and `<GEVA_ROOT>/bin/GUI.jar`. Now your program can be used from the command-line or from its own GUI:

GUI usage Go to `<GEVA_ROOT>/` and type `java -jar GEVA_GUI.jar`. In the first drop-down menu, select `Battleship.properties` (note that the configuration will change to reflect the `.properties` file we edited earlier), and click "run". The program should again run with output in the GUI's console.

Command-line usage Go to `<GEVA_ROOT>/bin` and type

```
java -jar GEVA.jar -main_class Main.Run -properties_file
../param/Parameters/Battleship.properties
```

and you should see the program running.

If you wish to see visualisations of the game grid, uncomment the print statements in battleship.java, rebuild and re-run. In the console you will now see the ships gradually being sunk more and more successfully as the generations go by.

8.5.1 Windows

Here are some notes especially for Windows users:

1. Download latest 'ant' (Binary Edition).
2. Dowload latest JDK (e.g., `jdk1.6.0_10`), jdk version <1.5 does not work.
3. Add path of ant and JDK: Go to 'My Computer > Properties > Advanced Tab > Environmental Variables > Systematic Variables > Path > click Edit >' add under this path add the following: `<ANT_ROOT>\bin;<JAVA_ROOT>\jdk1.6.0_10\bin` after this, we can run from windows command line `<GEVA_ROOT>\ant`, in order to build the file;

8.6 Next Steps

The reader who feels that the fitness could have been calculated differently is certainly right and is invited to experiment. However, a more interesting project might involve altering the grammar and using the fitness function to encourage the evolution of re-use. If the concept of horizontal salvos with variable sizes had to be discovered by evolution, rather than being built-in as it is here, this game might provide a neat demonstration of the usefulness of re-use and automatically-defined functions.

A description of how to use GEVA's command line interface is now presented in Tutorial 3 (Section 9).

9 Tutorial 3: Command Line Access

For those users who want more direct control and the ability to use GEVA with scripts a command-line interface is provided. To access this interface `GEVA.jar`, which is contained in the `bin/` directory, should be used. In a shell navigate into `<GEVA_ROOT>/bin`, and you can run GEVA by typing

```
java -jar GEVA.jar -main_class Main.Run
```

By default the `HelloWorld` demo problem will run with the default grammar and parameter settings. Output to stdout will be generated as follows:

```
Loading properties from file system: ../param/Parameters/HelloWorld.properties
mutation_probability=0.02
initialiser=Operator.RampedFullGrowInitialiser
generations=1000
replacement_type=generational
generation_gap=0.5
crossover_operation=Operator.Operations.SinglePointCrossover
evaluate_elites=false
userpick_size=20
grammar_file=../param/Grammar/letter_grammar.bnf
grow_probability=0.5
population_size=100
output=
fitness_function=FitnessEvaluation.PatternMatch.WordMatch
max_wraps=3
selection_operation=Operator.Operations.TournamentSelect
crossover_probability=0.7
mutation_operation=Operator.Operations.IntFlipMutation
max_depth=10
elite_size=10
word=geva
tournament_size=3
fixed_point_crossover=false
stopWhenSolved=true
Catch interval default: best individual
Output directory is /Users/mike/Desktop/GEVA-v1/bin/
#---Data---
Gen FitEvals Time(ms) Invalids BestFit AveFit VarFit AveUsedGenes AveLength
   0     0     44    0  3.000  4.980    2.900  9.690  9.000
   1     0     13    2  3.000  4.122    0.454  6.082  6.000
   2     0      7    4  2.000  3.792    0.186  5.833  7.000
   3     0      8    0  2.000  3.530    0.289  7.010  9.000
   4     0      5    2  2.000  3.265    0.317  8.051  9.000
   5     0      4    0  2.000  3.120    0.346  8.720 11.000
   6     0      6    4  2.000  3.063    0.579  8.958 11.000
   7     0      5    0  2.000  2.740    0.532  8.860 12.000
   8     0      7    0  2.000  2.610    0.498  8.820 13.000
   9     0      5    0  1.000  2.420    0.504  9.550 13.000
  10     0      6    2  1.000  2.643    0.821  9.439 14.000
  11     0      5    0  1.000  2.320    0.818  9.770 15.000
  12     0      4    0  1.000  2.350    0.988  9.380 16.000
  13     0      6    0  1.000  2.040    0.918  9.460 19.000
  14     0      9    0  1.000  1.770    0.837 10.690 20.000
  15     0      7    0  0.000  1.630    0.753 11.380 21.000
Rank:0 Fit:0.0 Phenotype:g e v a

Done running: Total time(Ms) for 1000 generations was:219
```

As can be seen above the output includes the list of parameters and their settings, followed by a number of statistics for each generation of the run, and the fitness and phenotype of the best solution found. Finally the total runtime of GEVA in milliseconds is output.

9.1 Changing Parameters

To change the parameters for the out-of-the-box GEVA there are two options.

1. Command line switches.
2. Modifying the Properties file.

9.1.1 Command Line

In a shell you can manually override the default parameter settings. To see the standard parameters which can be changed use:

```
java -jar GEVA.jar  -main_class Main.Run -h
```

For convenience the output of the help switch is provided here:

```
Command line arguments
-h for help
-v for version
-mutation_probability
-tail_percentage
-initialiser
-generations
-replacement_type
-generation_gap
-main_class
-crossover_operation
-evaluate_elites
-derivation_tree
-userpick_size
-grammar_file
-grow_probability
-population_size
-output
-fitness_function
-max_wraps
-selection_operation
-crossover_probability
-mutation_operation
-max_depth
-elite_size
-word
-tournament_size
-fixed_point_crossover
-stopWhenSolved
-initial_chromosome_size
```

- To change the mutation probability to 0.1%

```
java -jar GEVA.jar -main_class Main.Run -mutation_probability 0.001
```

- To change the number of wrapping events

```
java -jar GEVA.jar -main_class Main.Run -max_wraps 5
```

- To change the name of the output file to which run statistics are dumped

```
java -jar GEVA.jar -main_class Main.Run -output stats
```

Note that a timestamp will be concatenated onto the filename you choose (e.g., in this case you might see something like `stats11223421.dat`).
An example output statistics file is illustrated here:

```
#bestFitness averageFitness averageUsedGeneLength time invalids varFitness aveLength
3.0 4.9 9.82 98 0 3.1500000000000012 9.0
3.0 3.878787878787879 6.03030303030303 19 1 0.20752984389348092 6.0
3.0 3.7 5.82 8 0 0.29 6.0
2.0 3.38 5.79 9 0 0.25559999999999966 6.0
2.0 3.29 5.85 10 0 0.24590000000000017 7.0
2.0 3.14 6.06 6 0 0.3004000000000002 7.0
2.0 2.91 6.65 7 0 0.4618999999999992 8.0
2.0 2.52 7.73 10 0 0.3495999999999999 9.0
2.0 2.5 7.62 6 0 0.49 10.0
2.0 2.47 7.67 9 0 0.5090999999999998 11.0
1.0 2.46 7.58 5 0 0.5084000000000001 12.0
1.0 2.38 8.24 8 0 0.4155999999999997 14.0
1.0 2.38 8.22 6 0 0.49559999999999976 14.0
1.0 2.11 8.94 8 0 0.39790000000000025 14.0
1.0 2.05 9.12 21 0 0.507500000000001 14.0
1.0 2.01 9.96 5 0 0.6899 14.0
1.0 2.19 10.01 8 0 0.8538999999999993 14.0
1.0 2.06 10.06 4 0 0.7163999999999995 15.0
1.0 2.19 10.44 4 0 1.0938999999999997 14.0
1.0 2.08 10.02 30 0 0.7736000000000008 13.0
1.0 2.08 10.16 5 0 0.9136000000000007 14.0
1.0 1.79 10.59 5 0 0.7659000000000001 16.0
1.0 1.82 10.69 4 0 0.6875999999999999 17.0
1.0 1.74 10.52 18 0 0.6923999999999998 17.0
1.0 1.64 11.28 4 0 0.5304000000000002 19.0
1.0 1.78 11.28 7 0 0.8115999999999997 19.0
1.0 1.81 10.88 6 0 0.9738999999999991 21.0
1.0 1.65 11.11 3 0 0.6874999999999991 22.0
1.0 1.59 10.93 3 0 0.6419000000000002 24.0
0.0 1.55 11.14 5 0 0.6874999999999997 29.0
```

9.1.2 Properties File

To change parameter settings via the properties files, navigate to the directory containing the property files: `<GEVA_ROOT>/param/Parameters/`. Using your favourite text editor open the file of interest. To create a new properties file it is recommended to copy an existing file, or the `TemplateProperties.properties` file to a new file. When GUI version of GEVA is restarted the new property file will be automatically available from within the `Properties` drop-down menu.

To call a specific properties file from the command line using the `properties_file` switch. Go to `<GEVA_ROOT>/bin` and type

```
java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/<yourfilenamehere>.properties
```

9.2 Changing the Problem/Fitness Function

GEVA includes out-of-the-box a number of example problems that can be easily switched between from the command line or the properties file. To work with new problems please refer to the Advanced Tutorials for an example of how this can be achieved.

The simplest approach for any of the demo problems is to use all of the predefined parameter settings (which include the appropriate grammar) by using the correct properties file:

HelloWorld: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/HelloWorld.properties`

EvenFiveParity: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/EvenFiveParity.properties`

L-system: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/LSystem.properties`

Paint: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/Paint.properties`

SantaFeAnt: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/SantaFeAntTrail.properties`

SymbolicRegression: `java -jar GEVA.jar -main_class Main.Run -properties_file ../param/Parameters/SymbolicRegression.properties`

To select specific fitness functions and grammar files associated with some of the demo problems here are some further examples using the available problems from the command line.

- HelloWorld

```
java -jar GEVA.jar
     -main_class Main.Run
     -fitness_function FitnessEvaluation.PatternMatch.WordMatch
     -grammar_file ../param/Grammar/letter_grammar.bnf
```

In the case of the HelloWorld problem it is possible to modify the target string to something other than "geva" without modifying the code itself. Simply set a value for the parameter called `word`. For example, from the command line:

```
java -jar GEVA.jar -main_class Main.Run  -word helloworld
```

Here are examples for a selection of the other demo problems.

- Even Five Parity

```
java -jar GEVA.jar
     -main_class Main.Run
     -fitness_function FitnessEvaluation.ParityProblem.BooleanInterpreter
     -grammar_file ../param/Grammar/efp_grammar_gr.bnf
```

- Santa Fe Ant

```
java -jar GEVA.jar
     -main_class Main.Run
     -fitness_function FitnessEvaluation.SantaFeAntTrail.SantaFeAntTrailInterpreter
     -grammar_file ../param/Grammar/sf_grammar_gr.bnf
```

- Symbolic Regression ($x + x^2 + x^3 + x^4$)

```
java -jar GEVA.jar
     -main_class Main.Run
     -fitness_function FitnessEvaluation.SymbolicRegression.SymbolicRegressionInterpreter
     -grammar_file ../param/Grammar/sr_grammar_sch.bnf
```

9.3 Next Steps

The advanced user who now wants to jump straight into the deep end and get their hands dirty coding their own search engine should visit the Advanced Tutorials in Section 10.

10 Advanced Tutorials

In this series of tutorials the user gets their hands dirty with Java code. We gradually expose how to construct an evolutionary algorithm with GEVA starting by simply initialising a population in Tutorial 4. Tutorial 5 then demonstrates how this population can be manipulated in an evolutionary loop by setting up the pipeline of operators that can be employed (e.g., selection, crossover and mutation). Tutorial 6 then outlines how a new fitness function can be created.

10.1 Tutorial 4: How to Initialise a Population

To work through this tutorial navigate to `<GEVA_ROOT>/GEVA/src/Main/Tutorials/` and open `Tutorial4.java` in your favourite code editor. The parameter settings for this tutorial are contained in `<GEVA_ROOT>/param/Parameters/Tutorials/Tutorial4.properties`.

```
package Main.Tutorials;

import Algorithm.MyFirstSearchEngine;
import Algorithm.Pipeline;
import Algorithm.SimplePipeline;
import Main.AbstractRun;
import Mapper.GEGrammar;
import Util.Constants;
import Util.Random.MersenneTwisterFast;
```

Figure 10: Tutorial 4 header details.

Fig. 10 details the package and import statements necessary to setup a basic search algorithm in GEVA. Key components here are the use of the classes belonging to `Algorithm` to set up the basic architecture of the search engine itself. The `Mapper` class implements GE's characterisitc genotype-phenotype mapper. The remaining statements import utilities such as the random number generator.

Fig 11 outlines the signature of the `Tutorial4` class, and it extends `AbstractRun`. A `State` can be considered a container class which is used to setup, initialise and run the algorithm. `AbstractRun` adds the parameter reading functionality to `State`. The constructor method `Tutorial4()` begins by setting up the pseudo-random number generator and then points to the location of the parameter file for this example.

```
/**
 * Tutorial4 main class.
 * This class is derived from the class State. State is conceptually the outer level
 * of a program created using GEVA, it is a container class for the algorithm and it
 * used to setup initialise and run the algorithm. AbstractRun provides parameter
 * reading functionality. You should try to familiarise yourself with GEVA's parameterisation
 * mechanism(Covered in a later tutorial)  as it will make your life much easier :)
 *
 * When Implementing a algorithm with GEVA it makes sense to extend AbstractRun,
 * and implement the method setup(String[] args).
 * @author erikhemberg
 */
public class Tutorial4 extends AbstractRun {

    /** Creates a new instance of Tutorial4 */
    public Tutorial4() {
        this.rng = new MersenneTwisterFast();
        super.propertiesFilePath = Constants.DEFAULT_PARAM_ROOT
                                 + "Paramaters/Tutorials/Tutorial4.properties";
    }
```

Figure 11: Tutorial 4 class signature and constructor details.

The `setup(String[]args)` method is where all the magic happens in this example, see Fig. 12. The first order of business is to read in the command line arguments (if there are any). Next we setup the grammar which is read in from the current state of properties. The current state was read in from the properties file specified in the constructor method initially and the command line arguments modify this state. So this can be determined by either the properties file or overridden by using the appropriate command line switch.

Now we start to setup the search algorithm engine itself by adopting the `Algorithm` interface. We then create the `Initialiser` module from the grammar, pseudo-random number generator and remaining properties. This module is later passed into the algorithm pipeline.

This means we now need to create the pipeline for the algorithm. There are two pipelines assocated with an algorithm in GEVA. The first pipeline (`pipelineInit`) is used during the initialisation phase, the second pipeline (`pipelineRun`) during the main body of the algorithm (e.g. the evolutionary loop in the case of an Evolutionary Algorithm). The pipeline determines what actions take place in the execution of the algorithm, and of course their order of execution. `SimplePipeline()` is used to initialise the new pipeline, and we then tell the pipeline that it is the initialisation pipeline using `alg.setInitPipeline(pipelineInit);`.

Once we have the pipeline established we can add modules to it. Earlier we created the initialistion module which contains all the details required to setup the first population of candidate solutions.

Finally in this case we tell GEVA what algorithm to use with `this.algorithm=alg;`.

```
/**
 * Setup the algorithm. Read the properties. Create the modules(Operators) and operations
 * @param args The command line arguments
 */
public void setup(String[] args) {
    /* Read properties
     * You can configure many different aspects of GEVA
     * through the use of a properties file. While it is possible to configure
     * the modules using get/set methods we encourage you to learn how to use the
     * properties file and associated properties object to configure the system as
     * this makes it much easier to set up and change parameters for a run.
     * These properties can also be specified or overridden using the command line
     * arguments
     *
     * When you pass in an argument or an argument is read from the properties file
     * they are placed into a properties object. This object is passed to all modules'
     * constructor methods and each module is responsible for reading its own arguments.
     * This makes parameterising easy by reducing the need for parameter setting in the
     * code. In fact it is possible to set up and configure different types of algorithms
     * using the supplied modules using only a properties file.
     */

    // Creates properties from the command line arguments
    this.readProperties(args);

    /*
     * There are various steps to create an algorithm in GEVA. In this tutoral we only
     * intitalise a population. To get a better idea of what is going on, look at the
     * associated properties file Parameter/Tutorial4.properties
     */

    //Grammar
    GEGrammar grammar = new GEGrammar(this.properties);

    //State has a data member Algorithm alg; Algorithm is an interface and
    //MyFirstSearchEngine is a simple but surprisingly flexible implementation.
    MyFirstSearchEngine alg = new MyFirstSearchEngine();

    //Initialiser - One of he features of AbstractRun is that you can,
    //by using reflection, specify modules as parameters.
    //The method getInitialiser will return an initialiser as specified in the properties object.
    initialiser = getInitialiser(grammar, this.rng, this.properties);

    /*
     * Init
     * Here we create the initialisation pipeline. Algorithms derived from the base
     * class AbstractAlgorithm have two pipelines, pipelineInit and pipelineRun.
     * pipelineInit is run once at the start of an algorithm, so any initialisation
     * modules need to go on that pipeline.
     * In this tutorial we are simply going to create an instance of SimplePipeline(),
     * which as the suggests is a simple implementation of a pipeline. Pipelines are
     * at the heart of the way GEVA builds algorithms and they will be covered in more
     * detail in later tutorials. For now we simply need to add our initialiser module
     * to the initialisation pipeline.
     */
    Pipeline pipelineInit = new SimplePipeline();

    alg.setInitPipeline(pipelineInit);

    //Add modules to pipeline
    pipelineInit.addModule(initialiser);

    // Finally we set the algorithm
    this.algorithm = alg;
}
```

Figure 12: Tutorial 4 setup() method details.

```
/**
 * Run the state
 * @param args The command line arguments
 */
public static void main(String[] args) {
    try{
        //Create the Tutorial4 object
        Tutorial4 mfs = new Tutorial4();

        //Read the command-line arguments
        if(mfs.commandLineArgs(args)) {
            //Setup the algorithm. This calls the above setup method
            mfs.setup(args);

            //Initialize the algorithm. This runs the initialisation pipeline
            mfs.init();
            System.out.println("Well done running: Tutorial4, now look at Tutorial5");
        }
    } catch(Exception e) {
        System.err.println("Exception: "+e);
        e.printStackTrace();
    }
}
```

Figure 13: Tutorial 4 main method details.

The last step in this example is to write the main method to execute our pipeline. The code is detailed in Fig. 13. We start by calling the Tutorial4 constructor method, reading in the command-line arguments and finally calling the `init()` method to initialise our algorithm.

10.1.1 Running the code

Now to test your code. Navigate into the `<GEVA_ROOT>` directory, and use `ant` [2] to rebuild the code. That is, assuming `ant` is installed type `ant` in your favourite shell. The `build.xml` file is used by `ant` to recompile the modified source files and build fresh jar files. The `<GEVA_ROOT>/build.xml` file rebuilds all the jar files. Once finished the build, navigate to the `<GEVA_ROOT>/bin` and type the following:

```
java -classpath GEVA.jar Main.Tutorials.Tutorial4
```

The following will be output if successfully executed:

```
Loading properties from file system: ../param/Parameters/Tutorials/Tutorial4.properties
initial_chromosome_size=200
initialiser=Operator.Initialiser
max_wraps=1
grammar_file=../param/Grammar/HelloWorld_grammar.bnf
population_size=100
Well done running: Tutorial4, now look at Tutorial5
```

10.2 Tutorial 5: How to Construct a Simple Evolutionary Algorithm

In Tutorial 4 we had a look at a simple algorithm that only initilised a population. In this tutorial we will show how to construct a simple algorithm.

In GEVA algorithms are built by combining different modules into a pipeline. A module is a self-contained algorithm building block, by stacking modules an algorithm is created (You could create your entire algorithm in one module, but that is not what GEVA is designed for). Another tutorial will deal with creating your own modules, this tutorial only deals with the existing GEVA modules.

A module should perform a single part of the algorithm, e.g mutation or selection. In this tutorial we will create a fairly standard evolutionary algorithm using tournament selection, int-flip mutation and single point crossover, the newly created individuals will replace the worst individuals in the population.

This class is derived from the class State. State is conceptually the outer level of a program created using GEVA, it is a container class for the algorithm and it used to setup, initialise and run the algorithm. AbstractRun provides parameter reading functionality. You should try to familiarise yourself with GEVA's parameterisation mechanism (covered in a later tutorial) as it will make your life much easier. When implementing an algorithm with GEVA it makes sense to extend AbstractRun, and implement the method `setup(String[]args)`.

The `setup` method is identical to Tutorial4 up to setting up the `Initialiser` module. The first new item is the creation of a crossover module as detailed in Fig. 14.

```
/*
 * CROSSOVER
 * To implement crossover. First a crossover operation needs to be chosen.
 * Here it is single point crossover. To the operation a reference to the
 * random number generator is passed as well as the properties. The crossover
 * operation class will identifie which of the properties it will consider
 * for its settings, the rest will be ignored.
 *
 * After the operation is instanciated a CrossoverModule object is created,
 * which takes a reference to the crossover operation and sets it as the
 * operation performed by the module.
 */
CrossoverOperation singlePointCrossover = new SinglePointCrossover(this.rng, this.properties);
CrossoverModule crossoverModule = new CrossoverModule(this.rng, singlePointCrossover);
```

Figure 14: Tutorial 5 crossover module details.

In this case a one-point crossover operator is employed (`singlePointCrossover`. To create a crossover module the actual crossover type used (`CrossoverOperation`) and a pseudo-random number generator are required.

The next step sees the creation of a mutation module, which adopts an integer-based mutation. Codons in GEVA are integers by default. The integer mutation randomly picks an integer to replace the current codon value. You will notice a strong similarity to the creation of the crossover module (see Fig. 15).

```
/*
 * MUTATION
 * The same as for crossover, except that a mutation operation
 * and a mutation module are created.
 */
 IntFlipMutation mutation = new IntFlipMutation(this.rng, this.properties);
 MutationOperator mutationModule = new MutationOperator(this.rng, mutation);
```

Figure 15: Tutorial 5 mutation module creation details.

Fig. 16 outlines the creation of the selection and replacement modules, which is achieved in a similar fashion to crossover and mutation. The difference in this case is the choice of strategy in each case is read from the properties state (these were set either from the properties file or through the command-line). The choice of mutation and crossover are effectively hard-coded in this tutorial example.

```
/*
 * SELECTION
 * A selection operation is created by calling getSelectionOperation, to,
 * via reflection, create a SelectionOperation. The SelectionOperation
 * class to instanciate is specified in the properties file.  As with
 * crossover this is passed to the SelectionScheme module.
 */
 SelectionOperation selectionOperation = getSelectionOperation(this.properties, this.rng);
 SelectionScheme selectionScheme = new SelectionScheme(this.rng, selectionOperation);

/*
 * REPLACEMENT
 * Again similar procedure to crossover when creating the replacement
 * operation.  For the module creation reflection is used. This is
 * because there are different ways of joining the new and old
 * populations.  Another important thing to notice for the replacement
 * is that it needs to know which population it will join. Here it takes
 * the reference from selectionScheme.getPopulation(), in other words
 * the selected population.
 */
 ReplacementOperation replacementOperation = new ReplacementOperation(this.properties);
 JoinOperator replacementStrategy =
        this.getJoinOperator(this.properties, this.rng,
                             selectionScheme.getPopulation(), replacementOperation);
```

Figure 16: Tutorial 5 the creation of the selection and replacement modules.

Now we create the pipelines. The initialisation pipeline is created in an identical manner to Tutorial1. The difference in this tutorial is that we setup the `LoopPipeline`, and the details are provide in Fig. 17.

Up to this point we have established the various operators (modules) that will comprise the search algorithm. The next step is to associate the relevant population with the different modules (see Fig. 18). The selection module takes the initial population, and the replacement module also takes the initial popula-

```
/*
 * LOOP
 * Here we create the loop pipeline, this is what will be run after the
 * initialisation. It is to this pipeline that all the modules should be
 * attached. It is important to consider the execution order of the modules,
 * and which population they will be working on.
 */
 Pipeline pipelineLoop = new SimplePipeline();
 alg.setLoopPipeline(pipelineLoop);
```

Figure 17: The creation of the LoopPipeline in Tutorial 5.

tion in addition to the population output by population created by the selection scheme. The initial population is equivalent to the current population in an EA and the selection scheme population is equivalent to a temporary population of children which is used to create the next generation in combination with the current population members. The crossover and mutation modules then take the selection scheme population.

```
/*
 * POPULATIONS
 * Here the populations that the different modules will work on is set.
 * Remember that we set the population which would be join when we
 * constructed the module. Here the modules work on either the
 * initialised population or the selected population. Finally both
 * populations are in the replacementStrategy module.
 */
 selectionScheme.setPopulation(initialiser.getPopulation());
       // The selectionScheme takes one population and splits it
 replacementStrategy.setPopulation(initialiser.getPopulation());
       // The replacement takes two populations and joins them
       // The population that will be joined is specified in the constructor.
 crossoverModule.setPopulation(selectionScheme.getPopulation());
       // Crossover will be performed on the selected population
 mutationModule.setPopulation(selectionScheme.getPopulation());
       // Mutation will be performed on the selected population
```

Figure 18: Associating populations with the different modules from the pipelines.

Finally we add the modules to the loop pipeline in the order in which we wish them to be executed. This is outlined in Fig. 19. Then we create our main method, which is the same as in the previous tutorial except we can run the loop pipeline on this occasion as can be seen in Fig. 20 using the `run()` method.

10.2.1 Running the code

Now to test your code follow the same steps as for the last tutorial. Navigate into the `<GEVA_ROOT>` directory, and use `ant` [2] to rebuild the code. That is, assuming `ant` is installed type `ant` in your favourite shell. The `build.xml` file is

```
/*
 * PIPELINE
 * Here the modules are added in the desired order to the loop pipeline.
 * First selection will be performed. After that crossover will be
 * performed, remember that the crossover module will operate on the
 * selected population. This was set above, when the setPopulation was
 * called.  Then the selected population will be mutated. Finally the
 * replacementStartegy modul will replace the old population
 * with the selected population.
 */
pipelineLoop.addModule(selectionScheme);
    //Select the population according to the in the properties
    //file specified criteria, here tournament selection
pipelineLoop.addModule(crossoverModule);
    //Perform crossover on the setPopulation, here the selected
    //population will be subjected to single point crossover
pipelineLoop.addModule(mutationModule);
    //Mutate the setPopulation. Here int-flip muatation is
    //performed on the selected population
pipelineLoop.addModule(replacementStrategy);
    //Replace the old population with the selected population.
```

Figure 19: Adding modules to the pipeline in Tutorial 5.

```
   /**
    * Run the state
    * @param args The command line arguments
    */
   public static void main(String[] args) {
       try{
           //Create the Tutorial5 object
           Tutorial5 mfs = new Tutorial5();

           //Read the command-line arguments
           if(mfs.commandLineArgs(args)) {
               //Setup the algorithm. This calls the above setup method
               mfs.setup(args);

               //Initialize the algorithm. This runs the initialisation pipeline
               mfs.init();

               // Run the algorithm
               int its = mfs.run(); //Returns the number of iterations performed
               System.out.println("Well done running: Tutorial5 for "+its+",
                                   now look at Tutorial6");
           }
       } catch(Exception e) {
           System.err.println("Exception: "+e);
           e.printStackTrace();
       }
   }
```

Figure 20: The main method for Tutorial 5 which executes the loop pipeline.

used by `ant` to recompile the modified source files and build fresh jar files. The `<GEVA_ROOT>/build.xml` file rebuilds all the jar files. Once finished the build, navigate to the `<GEVA_ROOT>/bin` and type the following:

```
java -classpath GEVA.jar Main.Tutorials.Tutorial5
```

The following will be output if successfully executed:

```
Loading properties from file system: ../param/Parameters/Tutorials/Tutorial5.properties
initialiser=Operator.Initialiser
selection_operation=Operator.Operations.TournamentSelect
tournament_size=3
grammar_file=../param/Grammar/HelloWorld_grammar.bnf
replacement_type=steady_state
fixed_point_crossover=true
crossover_probability=0.9
initial_chromosome_size=200
mutation_probability=0.05
population_size=100
max_wraps=1
Well done running: Tutorial5 for 5000, now look at Tutorial6
```

10.3 Tutorial 6: How to Add a New Fitness Function

In Tutorial 5 we walked through an example of how a simple Genetic Algorithm could be implemented using GEVA without any fitness evaluation. In this tutorial we will show how to add a simple symbolic regression fitness evaluation to the algorithm.

In the demonstration problems outlined earlier in this document a Symbolic Regression example was introduced in which fitness was calculated using the JScheme interpreter. JScheme is a dialect of scheme with a simple java interface [22]. That is, solutions to the demo Symbolic Regression problem were output in the JScheme language.

One of the advantages of adopting the Grammatical Evolution approach to Genetic Programming is the flexibility of language choice that it allows. Effectively you can output code in any language and use a compiler for that language to create an executable which can be executed to calculate a fitness. Alternatively, if it is an interpreted language you can use the interepeter for fitness calculation. In this Tutorial we outline an alternative approach to the demo problem where we generate symbolic expressions in the Java language and use the Bean Scripting Framework [8] and Groovy [17] to dynamically evaluate them.

10.3.1 A new fitness function

A fitness function which uses the Bean Scripting Framework is provided in `<GEVA_ROOT>/GEVA/src/FitnessEvaluation/externalInterpreters/SymbolicRegression/SymbolicRegressionBSF.java`. The code is reproduced in Fig. 21. You can see that this class extends the `InterpretedFitnessEvaluationBSF` class which can be found in `<GEVA_ROOT>/GEVA/src/FitnessEvaluation/InterpretedFitnessEvaluationBSF.java`

In the `createCode` method contained in `SymbolicRegressionBSF.java` we basically write a Java class as a string and dynamically load in the evolved individual using the `code.append(p.getString());` statement. This Java code in the form of a string is then used in the `runFile` method in the `InterpretedFitnessEvaluationBSF.java` class. This string (Java code) is then passed to the Bean Scripting Framework and Groovy to evaluate its fitness.

Now that we have our new fitness function which evaluates code output in Java, we can now add this to the algorithm from the main method of this tutorial.

```
package FitnessEvaluation.externalInterpreters.SymbolicRegression;

import FitnessEvaluation.InterpretedFitnessEvaluationBSF;
import Individuals.Phenotype;

import java.util.Properties;

/**
 * Evaluates the fitness for the SymbolicRegressionExperiment class. The help class SymRegFunk
 * is used to evaluate the arithmetic expressions.
 * @author jonatan
 */
public class SymbolicRegressionBSF extends InterpretedFitnessEvaluationBSF {

    /** Creates a new instance of SymbolicRegression */
    public SymbolicRegressionBSF() {
    }

    public void setProperties(Properties p) {
    }

    public String createCode(Phenotype p) {
        StringBuffer code = new StringBuffer();
        //Header
        code.append("package FitnessEvaluation.SymbolicRegression;\n");
        code.append("public class Test extends SymRegFunkBSF {\n");
        code.append("\tpublic Test() {}\n");
        code.append("\tpublic double expr(double X) {\n\t\treturn ");
        //Input
        code.append(p.getString());
        //Tail
        code.append("\n\t}\n}\n");
code.append("test = new Test()\ntest.getFitness()");
        return code.toString();
    }

}
```

Figure 21: A simple Fitness function which adopts the Bean Scripting Framework to dynamically evaluate Java code.

10.3.2 Joining the Dots

As in Tutorials 4 and 5, again we start with the `setup()` method with the difference this time that we add in code to identify which fitness function to use. Fig. 22 outlines the statements necessary to achieve this where the fitness function to adopt is specified from the properties object whose state has been determined by either the properties file, or command line arguments, or both.

In GEVA fitness evaluation is peformed by an operation called `fitnessEvaluationOperation`. The `FitnessEvaluationOperation` uses the `fitnessFunction` to calculate fitness by calling the getFitness method of a FitnessFunction object.

The `FitnessEvaluator` is then used to pass the fitness function to the algo-

```
/*
 * FITNESS FUNCTION
 * The fitness function is determined dynamically by getFitnessFunction.
 * Where the fitness function is specified in the properties file.
 * A fitness function implements the interface FitnessFunction, this
 * interface requires it to implement getFitness(Individual i).
 * It is there that the fitness evaluation should be written. In this
 * example the fitness evaluation consists of matching the string
 * that an individual maps to a predifined string. This tutorial
 * demonstrates a simple case which does not involve compiling.
 *
 * The FitnessEvaluationOperation takes the fitness function as an
 * argument and calls the getFitness method in the FitnessFunction.
 * This operation checks if the individual is already evaluated,
 * mapped and valid. In this tutorial only unevaluated individuals with
 * valid mappings will be sent to the fitness function.
 * In GE the individual can be invalid if the there are non-terminals
 * still present in the developing solution after we have read all
 * the codons on the genome.
 *
 * A FitnessEvaluator module contains the operation and is what is
 * attached to the pipeline.
 */
FitnessFunction fitnessFunction = getFitnessFunction(this.properties);
FitnessEvaluationOperation fitnessEvaluationOperation =
           new FitnessEvaluationOperation(fitnessFunction);
FitnessEvaluator fitnessEvaluator =
           new FitnessEvaluator(this.rng, fitnessEvaluationOperation);
```

Figure 22: Tutorial 6 code to state which FitnessFunction to adopt.

```
FitnessEvaluator fitnessEvaluatorInit =
         new FitnessEvaluator(this.rng, fitnessEvaluationOperation);
fitnessEvaluatorInit.setPopulation(initialiser.getPopulation());
```

Figure 23: Addition of the FitnessEvaluator to the initialisation pipeline.

rithm pipelines. We can see the code making the addition to the initialisation pipeline in Fig. 23.

Before we add the FitnessEvaluator to the main loop pipeline we must first tell it which population to operate on (see Fig. 24) and then add it to the loop pipeline in Fig. 25.

```
fitnessEvaluator.setPopulation(selectionScheme.getPopulation());
// Fitness evaluation will be performed on the selected population
```

Figure 24: Tell the FitnessEvaluator which population to operate on in the main loop pipeline.

```
pipelineLoop.addModule(fitnessEvaluator);
//The population will be assigned new fitness
```

Figure 25: Addition of the FitnessEvaluator to the main loop pipeline.

10.3.3 Collecting Statistics

In the example code we have also included the ability to capture some statistics about the evolving population (see Fig. 26).

```
//Statistics
StatCatcher stats =
  new StatCatcher(Integer.parseInt(this.properties.getProperty("generations")));
IndividualCatcher indCatch = new IndividualCatcher(this.properties);
stats.addTime(startTime);
//Set initialisation time for the statCatcher (Not completly accurate here)
StatisticsCollectionOperation statsCollection =
  new StatisticsCollectionOperation(stats, indCatch, this.properties);
Collector collector = new Collector(statsCollection);

...
collector.setPopulation(initialiser.getPopulation());
//set population that collector works on
pipelineInit.addModule(collector);
// add collector to the initialisation pipeline

...
pipelineLoop.addModule(collector);
// add collector to the main loop pipeline
```

Figure 26: Addition of the statistics collector.

10.3.4 Running the code

Now to test your code follow the same steps as for the other tutorials. Navigate into the `<GEVAROOT>` directory, and use `ant` [2] to rebuild the code. That is, assuming `ant` is installed type `ant` in your favourite shell. The `build.xml` file is used by `ant` to recompile the modified source files and build fresh jar files. The `<GEVA_ROOT>/build.xml` file rebuilds all the jar files. Once finished the build, navigate to the `<GEVA_ROOT>/bin` and type the following:

```
java -classpath GEVA.jar Main.Tutorials.Tutorial6
```

Something along the lines of the following will be output if successfully executed:

```
Loading properties from file system: ../param/Parameters/Tutorials/Tutorial6.properties
fitness_function=FitnessEvaluation.SymbolicRegression.SymbolicRegressionBSF
initialiser=Operator.Initialiser
selection_operation=Operator.Operations.TournamentSelect
tournament_size=3
```

```
grammar_file=../param/Grammar/sr_grammar_gr.bnf
replacement_type=steady_state
fixed_point_crossover=true
crossover_probability=0.9
initial_chromosome_size=200
mutation_probability=0.05
population_size=10
max_wraps=1
generations=2
Catch interval default: best individual
#---Data---
Gen FitEvals Time(ms) Invalids BestFit AveFit VarFit AveUsedGenes AveLength AveDTDepth
   0     0    554    7 12.767 19.792   24.676  3.333 200.000  3.667
   5     0    332    2 12.767 18.035   27.760  4.000 200.000  4.000
  10     0    441    0 12.767 13.820    9.994  5.600 200.000  4.800
Rank:0 Fit:12.766599999999995 Phenotype:sin ( sin ( X ) )

Well done running: Tutorial6 for 10, now look at Tutorial7
```

10.3.5 Next Steps

This tutorial demonstrated how you can write your own fitness function to determine the fitness of a candidate solution regardless of the language in which it is written. The original demo Symbolic Regression instance used a dialect of Scheme to represent the solutions and this tutorial output simple Java expressions which were then dynamically evaluated using Groovy and the Bean Scripting Framework. For symbolic regression it is often the case that we wish to optimise the speed at which we can evaluate solutions, so it is common to code your own interpreter for simple prefix expressions for example. As a next step the user interested in Symbolic Regression should consider writing their own specialised fitness function with a built-in interpreter to suit their specific needs.

11 Conclusions

GEVA is available for download from the UCD NCRA group website `http://ncra.ucd.ie/geva` or `http://www.grammatical-evolution.org`. Included in the release are instructions on how to run GEVA out-of-the-box, and more detailed tutorials for those who wish to modify the software for new purposes. We also welcome feedback on the software as we plan to actively maintain the code, releasing new versions as features are added. A GEVA Google group has been set up to facilitate communication amongst the GEVA community [15]. We hope that GEVA will be a useful resource for the EC community and beyond.

Acknowledgments

We would like to thank past and present members of the UCD Natural Computing Research & Applications group, especially Tiberiu Simu, Jonathan Huggoson, and Jeff Wright for initial testing and some additions to the code. GEVA was beta-tested by the students of *COMP30290 Natural Computing*, an elective offered by UCD School of Computer Science & Informatics [27], from September to November 2007, and again from September to November 2008, and by Patrick Middleburgh who developed the seed of the LSystem interface as part of his Final Year Computer Science project at UCD. We are grateful to Edgar Galvan, Jonathan Byrne, Wei Cui, Jing Dang, John Mark Swafford, Uy Nguyen and Kai Fan for their comments on an earlier version of this document. We would also like to thank Miguel Nicolau for many interesting discussions that helped inform the design of GEVA, in particular the genotype-phenotype mapper which draws upon the design adopted in Miguel's libGE C++ library. This publication has emanated from research conducted with the financial support of Science Foundation Ireland under Grant No. 06/RFP/CMS042, and UCD Research Seed Funding.

References

[1] Amarteifio S. (2005). Interpreting a Genotype-Phenotype Map with Rich Representations in XMLGE, Masters Thesis, University of Limerick. Available from `http://ncra.ucd.ie/downloads/pub/SaoirseMScThesis.pdf`

[2] Apache Ant. `http://ant.apache.org/manual/index.html`.

[3] Berarducci P., Jordan D., Martin D., Seitzer J. (2004). GEVOSH: Using Grammatical Evolution to Generate Hashing Functions. In Proceedings of Genetic and Evolutionary Computation Conference (GECCO2004) workshop program.

[4] Brabazon, A., O'Neill, M. (2006). Biologically Inspired Algorithms for Financial Modelling. Springer.

[5] Brabazon A., O'Neill M. (2006). Credit Classification Using Grammatical Evolution. Informatica, pp.325-335 Vol.30 No.3.

[6] Brabazon A., O'Neill M. (2008). Bond Rating with πGrammatical Evolution. Knowledge-Driven Computing, pp.17-30 Springer.

[7] Brabazon A., O'Neill M., Dempsey I. (2008). An Introduction to Evolutionary Computation in Finance. IEEE Computational Intelligence Magazine, November, pp. 42-55, IEEE Press.

[8] Bean Scripting Framework. `http://jakarta.apache.org/bsf/`

[9] Cebrian M., Alfonseca M., Ortega A. (2007). Automatic generation of benchmarks for plagiarism detection tools using grammatical evolution. GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation, Vol. 2, pp. 2253-2253, ACM Press.

[10] Cleary R. (2005). Extending Grammatical Evolution with Attribute Grammars: An Application to Knapsack Problems, Masters Thesis, University of Limerick. Available from `http://ncra.ucd.ie/downloads/pub/thesisExtGEwithAGs-CRC.pdf`

[11] de la Puente A.O., Alfonso R.S., Moreno M.A. (2002). Automatic composition of music by means of grammatical evolution. Proceedings of the 2002 conference on APL, pp. 148-155, ACM Press.

[12] Dempsey I. (2007). Grammatical Evolution in Dynamic Environments, PhD Thesis, University College Dublin.

[13] Dempsey I., O'Neill M., Brabazon A. (2007). Constant Creation with Grammatical Evolution. International Journal of Innovative Computing and Applications, pp.23-38 Vol.1 No.1

[14] GEVA - Grammatical Evolution in Java. (2008). `http://ncra.ucd.ie/geva/`

[15] GEVA user group. `http://groups.google.com/group/geva`
Email: geva@googlegroups.com

[16] `http://www.grammatical-evolution.org`

[17] Groovy. `http://groovy.codehaus.org/`

[18] Harper R., Blair A. (2005). A structure preserving crossover in Grammatical Evolution. In Proceedings of IEEE Congress on Evolutionary Computation, pp.2537-2544, IEEE Press.

[19] Hemberg E., Gilligan C., O'Neill M., Brabazon A. (2007). A Grammatical Genetic Programming Approach to Modularity in Genetic Algorithms. In Ebner M., O'Neill M., Ekart A., Vanneschi L., and Esparcia Alcazar A. (eds.) *EuroGP (Tenth European Conference on Genetic Programming)*, Valencia, Spain. Springer.

[20] Hemberg E., O'Neill M., Brabazon A. (2008). Altering Search Rates of the Meta and Solution Grammars in the mGGA. In LNCS 4971 O'Neill M., Vanneschi L., Gustafson S., Esparcia-Alcazar A.I., De Falco I., Della Cioppa A., Tarantino E. (Eds.) Proceedings of EuroGP 2008, pp.362-373. Naples. Springer.

[21] Hemberg E., O'Neill M., Brabazon A. (2008). Grammatical Bias and Building Blocks in Meta-Grammar Grammatical Evolution. In Proceedings of IEEE World Congress on Computational Intelligence, pp. , Hong Kong, IEEE Press.

[22] JScheme `http://jscheme.sourceforge.net/jscheme/main.html`.

[23] Karpuzcu U.R. (2005). Automatic Verilog Code Generation through Grammatical Evolution. In Proceedings of Genetic and Evolutionary Computation Conference (GECCO2005) workshop program, pp. 394-397, ACM Press.

[24] Lewin B. (2000). Genes VII. Oxford University Press.

[25] Moore J.M., Hahn L.W. (2003). Petri net modeling of high-order genetic systems using grammatical evolution. BioSystems, 72(1-2), pp.177-186.

[26] Murphy J.E., Carr H., O'Neill M. (2008). Grammatical Evolution for Gait Retargeting. In Proceedings of Sixth Theory and Practice of Computer Graphics 2008 Conference TPCG08 Eurographics. University of Manchester, UK.

[27] O'Neill, M. (2007). COMP30290 Natural Computing. `http://ncra.ucd.ie/COMP30290/`

[28] O'Neill, M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick.

[29] O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation.* 2001.

[30] O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers.

[31] O'Neill, M., Brabazon, A. (2005). Recent Adventures in Grammatical Evolution. In Proceedings of CMS 2005 Computer Methods and Systems. Vol.1, pp. 245-253, November 2005, Krakow, Poland.

[32] O'Neill, M. and Brabazon, A. (2006). Grammatical Swarm: The Generation of Programs by Social Programming. Natural Computing, pp. 443-462 Vol.5 No.4

[33] O'Neill, M. and Brabazon, A. (2006). Grammatical Differential Evolution. International Conference on Artificial Intelligence (ICAI'06), pp. 231-236 CSEA Press Las Vegas, Nevada.

[34] O'Neill, M., Brabazon, A. (2008). Evolving a Logo Design using Lindenmayer Systems, Postscript and Grammatical Evolution. In Proceedings of the IEEE World Congress on Evolutionary Computation, pp. 3788-3794. Hong Kong, 1-6 June 2008. IEEE Press.

[35] O'Neill M., Brabazon A., Hemberg E. (2008). Subtree Deactivation Control with Grammatical Genetic Programming. In Proceedings of IEEE World Congress on Computational Intelligence, pp. , Hong Kong, IEEE Press.

[36] O'Reilly U-M., Hemberg M. (2007). Integrating generative growth and evolutionary computation for form exploration. Genetic Programming and Evolvable Machines, 8(2), pp.163-186.

[37] Prusinkiewicz, P. (1990). The Algorithmic Beauty of Plants. Springer-Verlag. Also available from `http://algorithmicbotany.org/papers/#abop`.

[38] Poli, R., Langdon, W.B., McPhee, N.F. (2008). A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`.

[39] Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, pp. 83-95, Springer-Verlag.

[40] An analysis of the MAX problem in genetic programming, Langdon, W.B. and Poli, R., Genetic Programming,222–230,1997

[41] Elitism reduces bloat in genetic programming,Poli, R. and McPhee, N.F. and Vanneschi, L.,Proceedings of the 10th annual conference on Genetic and evolutionary computation, 1343–1344, 2008, ACM New York, NY, USA

[42] The royal tree problem, a benchmark for single and multi-population genetic programming, Punch, WF and Zongker, D. and Goodman, ED, Advances in genetic programming, 299–316, 1996

[43] Tsoulos I.G., Lagaris I.E. (2006). Solving differential equations with genetic programming. Genetic Programming and Evolvable Machines, 7(1), pp. 33-54.