PerlGP - The Manual

Bob MacCallum
Stockholm Bioinformatics Center
Stockholm University
106 91 Stockholm
Sweden

February 3, 2003

Contents

1 Installation **5**
1.1 Versions and Disclaimer 5
1.2 Prerequisites 6
1.3 Unpacking 6
1.4 Environment Variables 6

2 Overview **7**
2.1 Aims of the Project 7
2.2 Naming and Font Conventions 8
2.3 Class/Object Hierarchy 9
2.4 How It All Works 9
2.4.1 Step-by-step: perlgp-run.pl 9
2.4.2 Step-by-step: Data input 10
2.4.3 Step-by-step: Tournaments I, Fitness Evaluation 11
2.4.4 Step-by-step: Tournaments II, Selection and Reproduction 12
2.5 What You Have to Implement 12

3 Class: Algorithm **13**
3.1 Methods 13
3.1.1 `new()` 13
3.1.2 `_init()` 13
3.1.3 `loadSet()*` 14
3.1.4 `loadData()` 14
3.1.5 `fitnessFunction()*` 14
3.1.6 `saveOutput()*` 15
3.1.7 `refresh()` 15
3.1.8 `run()` 15
3.1.9 `tournament()` 16
3.1.10 `calcFitnessFor()` 17
3.1.11 `makeFamilies()` 18
3.1.12 `crossoverFamily()` 18
3.1.13 `decideBetterFitness()` 19

3.1.14 stopCondition() 19
3.1.15 extraLogInfo() 19
3.1.16 parseExtraLogInfo() 19
3.2 Attributes & Variables 20
3.2.1 TrainingSet* 20
3.2.2 TestingSet* 20
3.2.3 FitnessDirection* 21
3.2.4 WorstPossibleFitness 21
3.2.5 Population 21
3.2.6 Tournament 21
3.2.7 Tournaments 22
3.2.8 TournamentsSinceBest 22
3.2.9 BestFitness 22
3.2.10 TournamentSize 22
3.2.11 TournamentParents 22
3.2.12 TournamentKillAge 23
3.2.13 AlwaysEvalFitness 23
3.2.14 MateChoiceRandom 23
3.2.15 TournamentLogFile 23
3.2.16 LogInterval 24
3.2.17 FitnessesFile 24
3.2.18 ComplexityLogFile 24
3.2.19 ComplexityInterval 24
3.2.20 RefreshInterval 24
3.2.21 RecentTrainingOutputFile 25
3.2.22 RecentTestingOutputFile 25
3.2.23 RecentCodeFile 25
3.2.24 BestTrainingOutputFile 25
3.2.25 BestTestingOutputFile 25
3.2.26 BestCodeFile 26
3.2.27 KeepBest 26
3.2.28 KeepBestDir 26
3.2.29 KeepMax 26
3.2.30 PopulationBackupInterval 26
3.2.31 EmigrateInterval 27
3.2.32 ImmigrateInterval 27
3.2.33 AlarmTime 27
3.2.34 ForkForEval 27

4 Class: Individual **28**
4.1 Methods 28
4.1.1 new() 28
4.1.2 _init() 28
4.1.3 reInitialise() 28
4.1.4 evalEvolvedSubs() 28
4.1.5 evolvedInit() 29
4.1.6 evaluateOutput()* 29
4.1.7 extraLogInfo() 30
4.1.8 crossover() 30
4.1.9 _start_crossover() 31

4.1.10 _crossover() 31
4.1.11 mutate() 32
4.1.12 point_mutate_shallow() 32
4.1.13 point_mutate_deep() 33
4.1.14 point_mutate() 33
4.1.15 _random_node() 33
4.1.16 macro_mutate() 34
4.1.17 replace_subtree() 34
4.1.18 insert_internal() 34
4.1.19 delete_internal() 35
4.1.20 copy_subtree() 35
4.1.21 swap_subtrees() 35
4.1.22 encapsulate_subtree() 36
4.1.23 simplify() 36
4.1.24 _xcopy_subtree() 37
4.1.25 _fix_nodes() 37
4.1.26 _get_subnodes() 37
4.1.27 _del_subtree() 37
4.1.28 _tree_type_size() 38
4.1.29 _display_tree() 38
4.1.30 saveCode() 38
4.1.31 saveTree() 38
4.1.32 save() 39
4.1.33 load() 39
4.1.34 tieGenome() 39
4.1.35 untieGenome() 40
4.1.36 retieGenome() 40
4.1.37 initTree() 40
4.1.38 _init_tree() 41
4.1.39 _grow_tree() 41
4.1.40 _expand_tree() 41
4.1.41 _tree_id() 41
4.1.42 initFitness() 42
4.1.43 eraseMemory() 42
4.1.44 memory() 42
4.1.45 getCode() 43
4.1.46 getSize() 43
4.1.47 Fitness() 43
4.1.48 Age() 43
4.1.49 _random_terminal() 44
4.1.50 _random_existing_terminal() 44
4.1.51 _random_function() 44
4.2 Attributes & Variables 45
4.2.1 NodeMutationProb 45
4.2.2 FixedMutations 45
4.2.3 FixedMutationProb 45
4.2.4 PointMutationFrac 45
4.2.5 NumericMutationFrac 46
4.2.6 NumericMutationRegex 46
4.2.7 NumericIgnoreNTypes 46

4.2.8 NumericAllowNTypes 46
4.2.9 NoNeutralMutations 47
4.2.10 PointMutationDepthBias 47
4.2.11 MacroMutationDepthBias 47
4.2.12 MacroMutationTypes 47
4.2.13 EncapsulateIgnoreNTypes 48
4.2.14 EncapsulateFracMax 48
4.2.15 UseEncapsTerminalsFrac 48
4.2.16 MutationLogFile 48
4.2.17 MutationLogProb 49
4.2.18 NodeXoverProb 49
4.2.19 FixedXovers 49
4.2.20 FixedXoverProb 49
4.2.21 XoverDepthBias 50
4.2.22 XoverSizeBias 50
4.2.23 XoverHomologyBias 50
4.2.24 AsexualOnly 50
4.2.25 XoverLogFile 51
4.2.26 XoverLogProb 51
4.2.27 MaxTreeNodes 51
4.2.28 MinTreeNodes 51
4.2.29 TreeDepthMax 52
4.2.30 TerminateTreeProb 52
4.2.31 TreeDepthMin 52
4.2.32 NewSubtreeDepthMean 52
4.2.33 NewSubtreeDepthMax 53
4.2.34 UseExistingTerminalsFrac 53
4.2.35 GetSizeIgnoreNTypes 53
4.2.36 DBFileStem 53
4.2.37 ExperimentId 53
4.2.38 Population 54
4.2.39 Functions 54
4.2.40 Terminals 54

5 Class: Population 54
5.1 Methods 54
5.1.1 new() 54
5.1.2 _init() 55
5.1.3 addIndividual() 55
5.1.4 findNewDBFileStem() 55
5.1.5 repopulate() 55
5.1.6 backup() 56
5.1.7 emigrate() 56
5.1.8 export_tar_file() 56
5.1.9 immigrate() 57
5.1.10 initFitnesses() 57
5.1.11 countIndividuals() 57
5.1.12 randomIndividual() 58
5.1.13 selectCohort() 58
5.2 Attributes & Variables 58

5.2.1 `Individuals` . . . 58
5.2.2 `PopulationSize` . . . 58
5.2.3 `MigrationSize` . . . 58
5.2.4 `PopulationDir` . . . 59
5.2.5 `ExperimentId` . . . 59

6 Universal Base Class: PerlGPObject **59**
6.1 Methods . . . 59
6.1.1 `new()` . . . 59
6.1.2 `AUTOLOAD()` (get and set attributes) . . . 59
6.1.3 optionalParams() . . . 60
6.1.4 compulsoryParams() . . . 60
6.2 Attributes & Variables . . . 61
6.2.1 `Class` . . . 61

7 Grammar definition **61**
7.1 Tree-as-Hash-Table Genotype Representation . . . 62
7.2 Grammar Specification . . . 62
7.3 Random Initialisation of Programs . . . 63

8 Utility Scripts **64**
8.1 perlgp-run.pl . . . 64
8.2 plot-tlog.pl . . . 65
8.3 perlgp-wipe-expt.pl . . . 65
8.4 perlgp-rand-prog.pl . . . 65
8.5 perlgp-sample-pop.pl . . . 65
8.6 perlgp-show-prog.pl . . . 66
8.7 perlgp-mrun.pl . . . 66
8.8 perlgp-avg-logs.pl . . . 66

9 Demos **67**
9.1 Approximation of pi . . . 67
9.1.1 Problem definition . . . 67
9.1.2 PerlGP approach . . . 67
9.2 Symbolic regression of sine function . . . 68
9.2.1 Problem definition . . . 68
9.2.2 PerlGP approach . . . 68
9.3 Compound interest . . . 69
9.3.1 Problem definition . . . 69
9.3.2 PerlGP approach . . . 69

1 Installation

1.1 Versions and Disclaimer

This is version 1.001 of the PerlGP manual, which refers to version 1.0.0 of PerlGP, released on 03 Feb 2003. While every effort has been made to keep the manual up to date and accurate, it is not guaranteed to be free of errors and omissions. The author is not responsible for any losses or inconvenience

caused by such errors. All reports of errors and suggestions for improvement or clarification are welcome. This manual is copyright Bob MacCallum 2003.

1.2 Prerequisites

You are strongly advised to install the plotting program gnuplot on your system (`http://www.gnuplot.info`). That's all you need I think, apart from Perl of course. Perl 5.6.1 or higher is recommended, but 5.6.0 works too (you may experience memory leaks).

PerlGP is currently only tested under Linux, but it should work on all Unix-like systems (it used to run on IRIX a few years ago, and in MacPerl even longer ago).

1.3 Unpacking

After obtaining the file perlgp-X.Y.Z.tar.gz (where X, Y and Z are version numbers), unpack it with one of the following command sequences on most Unix-like systems:

```
  linux> tar xzf perlgp-X.Y.Z.tar.gz
or
  unix1> gtar xzf perlgp-X.Y.Z.tar.gz
or
  unix2> gunzip perlgp-X.Y.Z.tar.gz
  unix2> tar xf perlgp-X.Y.Z.tar
```

Then cd into the newly created directory named perlgp-X.Y.Z and follow the instructions in the README file for starting to work with the PerlGP system. Before you download, you may take a look at this README file in the following section.

1.4 Environment Variables

Here follows a copy of the README file to give you an idea of the preparation needed to get PerlGP ready for use. I hope you agree that it is not much work. *For the latest instructions, always refer to the README file in the latest distribution.*

```
-----------------
The PerlGP system  -  author/copyright: Bob MacCallum 2002
-----------------

To run the demos, first set the following environment variables
(preferably in your shell startup file)

##############################
# PerlGP environment variables (bash style)

# where you unpacked the PerlGP package
PERLGP_BASE=~/perlgp
```

```
# probably don't need to change these two
PERLGP_LIB=$PERLGP_BASE/lib
PERLGP_BIN=$PERLGP_BASE/bin

## for best results, add $PERLGP_BIN to your PATH ##

# a LOCAL temporary directory (you'll need to clean up manually afterwards)
PERLGP_SCRATCH=/scratch

# a network directory for storing 'checkpoint/restart' populations and
# migrants between machines.  leave empty if you're not doing this
PERLGP_POPS=

# if you use a queueing system, give the name of the environment
# variable which contains the job id
PERLGP_JOBID=

#
###############################


then you simply go into a demo directory and type

perlgp-run.pl

(preferably backgrounding it)

for more, see the README files in the demo directories.
```

2 Overview

2.1 Aims of the Project

One early aim in this project was to create a GP system using some of Perl's most convenient and well developed features (namely hash tables and regular expressions) rather than make a Perl clone of existing implementations. The initial goal was to evolve string manipulation code for protein secondary structure prediction, but through time the emphasis has shifted more towards making a robust GP system for general use.

This GP implementation has always tried to follow the evolution of biological organisms. It does not start from a baseline, such as Koza's work, but uses many borrowed and a few original ideas. The first release of this software contains code for running tournament based genetic algorithm on a population of tree-encoded programs. However, the object-oriented design allows exchange of components, for example a generational GA or Monte Carlo approach for the search algorithm, and other representations of individuals. Object method overloading lets you easily customise the algorithms to suit your needs.

2.2 Naming and Font Conventions

I have tried to be consistent with naming, particularly during the recent reorganisation in preparation for going open source. However, at this stage the project was around 4 years old and there are some exceptions to the rules.

The following table should explain all the font usage in this manual:

Thing	Example	Description	Exceptions
Perl program or package	perlgp-run.pl	normal font, with trailing .pl or .pm	
File or directory name	'results/best.pl'	normal font, in single quotes	see text
Environment variable	PERLGP_LIB	normal capitals, no $	
Public object method	`loadSet()`	typewriter font, trailing (), no underscores, capitalised words, except first word, which is usually a verb	`Fitness()`, `Age()`, mutation operators
Private object method	`_init_tree()`	typewriter font, leading underscore, usually all lowercase with underscores between words	
Mutation operator	`copy_subtree()`	typewriter font, lowercase with underscores, trailing ()	
Other Perl functions	`eval()`	typewriter font, trailing ()	
Object attribute	`AlarmTime`	typewriter font, no underscores, capitalised words	don't forget the get and set forms: `$self->KeepBest()` and `$self->KeepBest(5)`
Variables	*fitness*	italics	
Data types	NUMBER	small capitals	

Finally, some subroutines and methods take a hash table as their argument. This is a way of having "named" arguments, which don't have to be given in

any particular order (and can be left out if not required). In the manual, this is presented as:

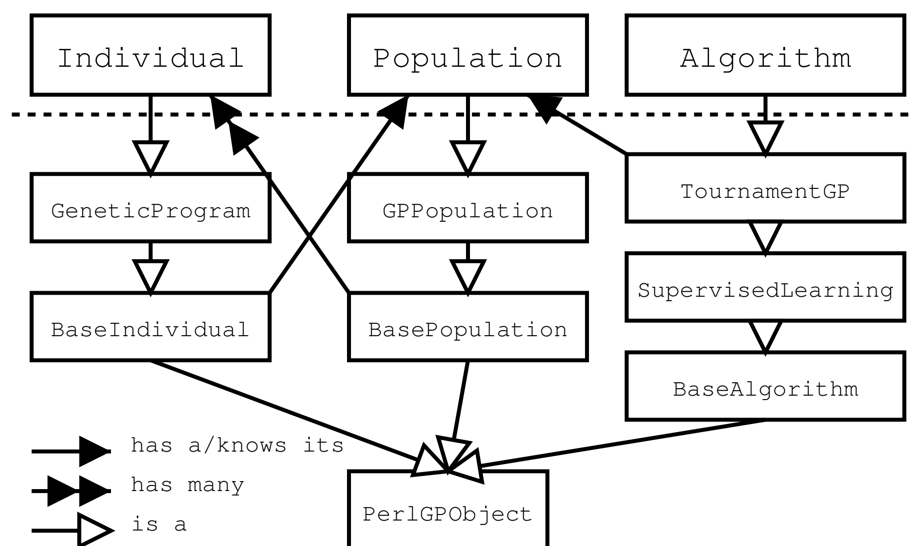
Arguments: `ArgName1 =>` TYPE1, `ArgName2 =>` TYPE2

Be extremely careful typing these argument names, because if you spell them wrong, it will probably silently ignore it.

The type called ATTRIBUTE-HASH is just a hash where the keys are valid Attribute names, and the values are the values you want them to have.

Non absolute file names are always relative to the current "experiment directory", which is the directory which contains the Algorithm.pm, Individual.pm and so on.

2.3 Class/Object Hierarchy



The classes above the dotted line are the "user servicable parts", you don't need to edit the classes below this, but you might need to look at them from time to time (they live in PERLGP_LIB).

2.4 How It All Works

Here I will take you through the perlgp-run.pl script and the sine curve fitting example, also discussed in Section 9.2. Some code is excluded for clarity. This script is always run from the "experiment directory", the one containing your locally modified Algorithm, Individual and Population classes (see Section 2.5).

2.4.1 Step-by-step: perlgp-run.pl

```
#!/usr/bin/perl -w

use lib '.', $ENV{PERLGP_LIB} ||  die "
  PERLGP_LIB undefined
  please see README file concerning shell environment variables\n\n";
```

```
use Population;
use Individual;
use Algorithm;
use Cwd;

my ($exptid) = cwd() =~ m:([^/]+)$:;
```

Here we make sure that the current directory and PERLGP_LIB are in the Perl include path. Then the three main classes are loaded and the experiment id is extracted from the trailing part of the current directory. In this case `$exptid` would be 'sin'.

```
my $population = new Population( ExperimentId => $exptid );
$population->repopulate();
```

Then we make a new Population object with this experiment id, and ask it to repopulate itself from disk (only does something if this is a restarted run).

```
while ($population->countIndividuals() < $population->PopulationSize()) {
  $population->addIndividual(new Individual( Population => $population,
                                              ExperimentId => $exptid,
                       DBFileStem => $population->findNewDBFileStem()));
}
```

This bit fills up the Population with brand new Individuals, until it is full. Note that they are told the identity of their parent Population, but this isn't used by the Individuals in the standard PerlGP system.

```
my $algorithm = new Algorithm( Population => $population );
```

Now we create an Algorithm object which does need to know about a Population, because this is the Population that the (genetic) Algorithm will manipulate.

```
$algorithm->run();
```

Then we ask the algorithm object to run itself, and this is the last thing that perlgp-run.pl does. So you can see that perlgp-run.pl is just a wrapper which constructs some objects and tells them to get on with it...

2.4.2 Step-by-step: Data input

Now into TournamentGP.pm we go... The `run()` method is basically a `for` loop around calls to the `tournament()` method. But first, the data is loaded if needed with

```
  $self->loadData() unless ($self->TrainingData());
```

In other words, if the `TrainingData` attribute is undefined, then `loadData()` is called (which as you will see, will set both `TrainingData` and `TestingData`).

`loadData()` is just a wrapper to call the user-defined `loadSet()` method on training and testing data. In the sine curve example, the data is read from files

TrainingSet and TestingSet (there is no real need for this, the data could be generated on-the-fly, but it is more transparent).

If you look at loadSet() in Algorithm.pm in the sin demo directory, you'll see that an array is filled as the input file is read. Each element represents one training point and an 'x' and "known" 'y' value are stored in an anonymous hash (wasteful but transparent). loadSet() returns a reference to this array, and this scalar reference will be stored in the attribute TrainingData or TestingData. We have now loaded up the data structures for training and testing data into the Algorithm object.

```
sub loadSet {
  my ($self, $file) = @_;
  my @set;
  open(FILE, $file) || die "can't read data set $file\n";
  while (<FILE>) {
    my ($x, $y) = split;
    push @set, { 'x'=>$x, 'y'=>$y };
  }
  close(FILE);
  return \@set;
}
```

2.4.3 Step-by-step: Tournaments I, Fitness Evaluation

So remember that run() calls tournament() many times. The most important steps in tournament() are outlined below.

First a group of Individuals is selected from the Population at random:

```
  my @cohort = $self->Population()->selectCohort($self->TournamentSize());
```

Then for each Individual in the cohort, the fitness on the training examples is calculated (simplified):

```
    my $fitness = $ind->Fitness();
    if (not defined $fitness) { # fitness is not cached
      $ind->reInitialise();     # evaluate evolved subroutines
      $fitness = $self->calcFitnessFor($ind, $self->TrainingData());
      $ind->Fitness($fitness);  # set new fitness
    }
```

The method reInitialise() calls evalEvolvedSubs() which expands the code tree into Perl code and evaluates it, which redefines object methods for the Individual including, importantly, evaluateOutput().

The method calcFitnessFor() is a wrapper which calls evaluateOutput() on the Individual object and then calls the Algorithm method fitnessFunction() on the output.

Let's take a closer look at evaluateOutput(). You can get a copy by either looking at Grammar.pm or running perlgp-rand-prog.pl, which will generate a random version from the grammar.

```
sub evaluateOutput {
  my ($self, $data) = @_;
```

```
  my ($x, $y, $z, @output);
  foreach $input (@$data) {
    $x = $input->{x};
    # begin evolved bit

    $y = (2 + ($x - pdiv(5,2)));

    # end evolved bit
    push @output, { 'y'=> $y };
  }
  return \@output;
}
```

Note that this has some similarities to `loadSet()`; mainly in the way it fills an array with anonymous hashes and returns a reference to it. This argument passed to this method is the data structure that `loadSet()` returned. The data structure returned by `evaluateOutput()` is then passed to the fitness function.

In most cases, `fitnessFunction()` also requires the "correct answer" too. You will see that in this example, `loadSet()` loads the "known" y value into the input data structure. It is critical that the y value is not accessed by the evolving code, otherwise the trivial result `$y = $y` would occur, and we would be wasting our time. So long as y is not mentioned in the grammar, this will not occur. The main point here is that the input data structure is not purely input in nature, it will often also contain the known or observed output too. The use of the word "input" here is not perfect, and I apologise.

The only requirement that PerlGP makes on the data structures passed between these methods is that it must be a scalar variable. You will usually use a reference to an array or a hash, but look in the pi demo (Section 9.1) where you will see that the scalar is a number.

2.4.4 Step-by-step: Tournaments II, Selection and Reproduction

Now that we have a fitness value for every Individual in the tournament, the rest is standard. The tournament members are sorted on fitness and the best ones get a chance to reproduce (by calling the Individual method `crossover()`) and replace the worst ones. The offspring are mutated.

2.5 What You Have to Implement

Data I/O Algorithm.pm: `loadSet()`, `saveOutput()`

Fitness Algorithm.pm: `fitnessFunction()`

Grammar Grammar.pm: Functions and Terminals to produce Perl code which defines `Individual::evaluateOutput()` or other object methods or subroutines. All explained in Section 7.

Functions Individual.pm: Perhaps you need to define some special functions, like protected division. If your grammar didn't produce a definition for `evaluateOutput()` then you need to define it here.

3 Class: Algorithm

The following sections refer to the `TournamentGP` implementation of the Algorithm object, unless otherwise stated.

3.1 Methods

Methods marked with * *must* be implemented in Algorithm.pm for each new application/experiment. These are the routines for reading/writing data, fitness function etc.

3.1.1 `new()`

Arguments: `Population =>` OBJECT, ATTRIBUTE-HASH (optional)
Return value: OBJECT
Defined in: PerlGPObject.pm
Mainly called by: perlgp-run.pl
Usually calls: `_init()`
Relevant attributes: all

This is the constructor, but you probably don't need to call it directly, except in specialist applications. A Population object must be given in the argument hash. Other attributes may also be specified in the argument hash, but note that the "usual" way to set these attributes is to edit Algorithm.pm in the experiment directory.

3.1.2 `_init()`

Arguments: ATTRIBUTE-HASH
Return value: void
Defined in: Algorithm.pm, TournamentGP.pm, SupervisedLearning.pm
Mainly called by: constructor
Usually calls: `_init()` cascade (see below)
Relevant attributes: all, *and in particular:* `BestFitness`, `WorstPossibleFitness`, `FitnessDirection`, `TournamentLogFile`, `Tournament`

This method sets the attributes. You should customise the object attributes by editing the `_init()` method in Algorithm.pm. Default settings can be found in TournamentGP.pm and SupervisedLearning.pm. The `_init()` methods are called in a cascade in the following order `Algorithm::_init()`, `TournamentGP::_init()`, `SupervisedLearning::_init()`. Attributes set at the beginning of the cascade override those defined at the end.

`TournamentGP::_init()` performs some other initialisation. If `WorstPossibleFitness` has not already been defined in Algorithm.pm, it is set to zero or an arbitrarily large number (1e99) according to the `FitnessDirection`. The attribute/variable `BestFitness` is used to store the fitness of the fittest Individual during the run, and is initialised to `WorstPossibleFitness`. If the file 'results/tournament.log' file exists (for example after a restart), `Tournament` (the current tournament number) and `BestFitness` are updated from the values in last line in this file. If the 'results'

directory does not exist it is created, likewise the directory `KeepBestDir` if required.

3.1.3 `loadSet()*`

Arguments: STRING *filename*
Return value: SCALAR (usually a reference to training/testing data structure)
Defined in: Algorithm.pm
Mainly called by: `loadData()`
Usually calls:
Relevant attributes:

You must redefine this method to load up training *or* testing data into a suitable data structure, and return a scalar variable (which is usually going to be a reference to a more complex data structure). If your data is in files then you can use *filename*; see Section 3.1.4 for more details, and see the sine curve fitting demo in Section 9.2 for an example. You don't have to read from a file of course; see the pi demo in Section 9.1 for an example.

3.1.4 `loadData()`

Arguments: none
Return value: void
Defined in: SupervisedLearning.pm
Mainly called by: `run()`, `refresh()`
Usually calls: `loadSet()`
Relevant attributes: `TrainingSet`, `TestingSet`, `TrainingData`, `TestingData`

A simple wrapper to call `loadSet()` twice, once each on the training and testing data. The filenames passed to `loadSet()` are the attributes `TrainingSet` and `TestingSet`. The two return values (usually references to data structures) are stored in the attributes `TrainingData` and `TestingData`.

3.1.5 `fitnessFunction()*`

Arguments: `Input =>` SCALAR, `Output =>` SCALAR, `TimeTaken =>` NUMBER, `CodeSize =>` NUMBER
Return value: NUMBER
Defined in: Algorithm.pm
Mainly called by: `calcFitnessFor()`
Usually calls:
Relevant attributes: `FitnessDirection`

Calculates the fitness given the Input data (which should also contain the desired/true output value), and the Output data (which is generated by `Individual::evaluateOutput()`). The Input and Output scalars will usually be references to data structures containing multiple data points, which you will have to loop through (see Section 9.2 for an example). You may ignore CodeSize and TimeTaken. The fitness value may go "up" or "down" as you wish, but you must set the attribute `FitnessDirection` accordingly.

3.1.6 `saveOutput()*`

Arguments: `Filename =>` FILENAME, `Input =>` SCALAR, `Output =>` SCALAR, `Individual =>` OBJECT (optional)
Return value: void
Defined in: Algorithm.pm
Mainly called by: `tournament()`
Usually calls:
Relevant attributes:

This method saves the output (usually from the best-of-tournament Individual) in FILENAME for further offline analysis, for example plotting a regression curve. Both Input and Output data are required as in Section 3.1.5, so that input variables, output variables and the desired/true outputs can be saved. If the Individual object is passed to this function, you can print out some information about that too. See the sine curve demo in Section 9.2 for an example of how to implement this method.

This method should perhaps have been put in the Individual class, but it was put in the Algorithm class alongside `loadSet()` which also deals with data formatting.

3.1.7 `refresh()`

Arguments: none
Return value: void
Defined in: TournamentGP.pm
Mainly called by: `run()`
Usually calls:
Relevant attributes: `RefreshInterval`, `BestFitness`

Redefine this method in Algorithm.pm if you want to re-read or modify the training/testing data at certain intervals (see attribute `RefreshInterval`). The sine demo in Section 9.2 uses this technique. The default `refresh()` does nothing. It is usually a good idea to reset `BestFitness` if the training data changes.

3.1.8 `run()`

Arguments: none
Return value: void
Defined in: TournamentGP.pm
Mainly called by: perlgp-run.pl
Usually calls: `tournament()`, `loadData()`, `refresh()`, `stopCondition()`, `Population::{backup(),immigrate(),emigrate()}`
Relevant attributes: `Tournaments`, `Tournament`, `RefreshInterval`, `ComplexityInterval`, `ComplexityLogFile`, `PopulationBackupInterval`, `EmigrateInterval`, `ImmigrateInterval`

This method will try to call the `tournament()` method `Tournaments` times, regardless of the starting tournament number (in the case of a restart). If `RefreshInterval` is non-zero, `refresh()` is called before any tournaments are

performed. Then `loadData` is called if `TrainingData` has not already been loaded.

Before each call to `tournament()`, `refresh()` is called if the tournament number (`Tournament`) modulus `RefreshInterval` equals zero. Afterward each tournament, if `stopCondition()` returns true, the main loop is terminated, and no more tournaments will be performed.

The `ComplexityLogFile` is updated every `ComplexityInterval` tournaments with size in characters of the Perl code of a random 10% of the population after compression with gzip. This gives an estimate of the complexity of the population of evolved programs. The 10% sample is made using perlgp-sample-pop.pl.

At intervals of `PopulationBackupInterval`, the Population object belonging to this Algorithm object is told to perform a backup. Immigration and emigration are handled similarly. See Section 5 for more details.

3.1.9 `tournament()`

Arguments: none
Return value: void
Defined in: TournamentGP.pm
Mainly called by: `run()`
Usually calls: `Population::selectCohort()`, `Individual::reInitialise()`, `calcFitnessFor()`, `makeFamilies()`, `crossoverFamily()`, `extraLogInfo()`, `saveOutput()`
Relevant attributes: `TournamentSize`, `TournamentKillAge`, `ForkForEval`, `BestFitness`, `TournamentsSinceBest`, `WorstPossibleFitness`, `LogInterval`

This is where it all happens. First a cohort[1] of size `TournamentSize` is generated by calling the Population's `selectCohort()` method. The Individuals in the cohort whose `Age` is greater than or equal to `TournamentKillAge` are removed and labelled "old", and those that remain are called "young".

The fitnesses of each young Individual are either retrieved from memory (see Section 4.1.44 and `AlwaysEvalFitness`) or calculated afresh with a call to `calcFitnessFor()`. Before `calcFitnessFor()` is invoked, the `reInitialise()` method is called on the Individual, to redefine any evolved methods, such as `evaluateOutput()`. This is important, because otherwise all Individuals would generate the same outputs.

There is an option to restrict run-time with an `alarm()` call (see `calcFitnessFor` and `AlarmTime`). Sometimes this causes instability in Perl when the alarm call arrives (particularly when it interrupts the evaluation of regular expressions). A work-around is to do the fitness calculation in a forked process by setting the `ForkForEval` attribute. Before any fitness evaluation is performed, the fitness of the Individual is set to `WorstPossibleFitness`, in case something goes wrong during forking or as a result of alarm calls.

The `Age` of each young Individual is incremented by one every tournament.

The young Individuals are sorted by fitness, so that the best Individuals are at the top of the list. The old Individuals are added back to the bottom of the list. The method `makeFamilies()` is called to convert the sorted cohort into an array of "families" each containing four individuals: two parents and

[1]Cohort: "A band of warriors" according to the Concise Oxford Dictionary

Table 1: Descriptions of columns in 'results/tournament.log'

column	description
1	unix time when log was updated
2	tournament number
3	`BestFitness` - best (training) fitness seen so far
4	(training) `Fitness` of best-of-tournament Individual
5	test set fitness for the same Individual as in column 4
6	result of best-of-tournament `Individual::Age()`
7	result of best-of-tournament `Individual::getSize()`

two unfit Individuals which will be replaced when the parents reproduce (see `crossoverFamily()`).

A number of log-files are created or appended every `LogInterval` tournaments. The most important is 'results/tournament.log', where a summary of the run's progress through time is stored. Table 1 explains the format of this file. Additional columns can be specified by redefining `extraLogInfo()` for both the Algorithm and Individual objects. The output of the best-of-tournament Individual for both training and testing data are saved using `saveOutput()` (to 'results/recent.training.output' and 'results/recent.testing.output') and the Perl code is also written to a file ('results/recent.pl').

If the best-of-tournament Individual has better fitness than `BestFitness` (this usually means this is the best Individual seen so far), the files discussed above are always saved (as 'results/best.*'), even if logging is not being done this tournament. `BestFitness` is updated and the counter `TournamentsSinceBest` is reset to zero.

3.1.10 `calcFitnessFor()`

Arguments: OBJECT *individual*, SCALAR *inputdata*
Return value (array context): NUMBER *fitness*, SCALAR *outputdata*
Return value (scalar context): NUMBER *fitness*
Defined in: TournamentGP.pm
Mainly called by: `tournament()`
Usually calls: `Individual::evaluateOutput()`, `fitnessFunction()`
Relevant attributes: `WorstPossibleFitness`, `AlarmTime`

This method just runs *individual*`->evaluateOutput()` on *inputdata* and then runs `fitnessFunction()` on the returned output. There are a few extra details; firstly, if `AlarmTime` is non-zero, a system alarm call is set for `AlarmTime`

seconds. The call to `evaluateOutput()` is protected within an `eval{ }` block, so that run-time errors or alarm calls do not terminate the entire GP run, but just the evaluation within the block. The call to `evaluateOutput()` is also timed using the `times()` Perl function/system call, which gives precision to around 0.01s. The elapsed time is passed to `fitnessFunction()`. The result of *individual*`->getSize()` is also passed to `fitnessFunction()`.

3.1.11 `makeFamilies()`

Arguments: ARRAYREF *cohort*, ARRAYREF *families*
Return value: void (but *families* is filled)
Defined in: TournamentGP.pm
Mainly called by: `tournament()`
Usually calls:
Relevant attributes: `TournamentParents`, `MateChoiceRandom`

The argument *cohort* is a reference to an array of Individuals (which are presumably sorted by fitness and age). This method generates "families" containing two potential parents and two Individuals which will be replaced by their offspring. First we remove the first `TournamentParents` elements from *cohort* and put them in an array called *parents*. Likewise we take the last `TournamentParents` elements from *cohort* and put them in an array called *rip* (from Rest In Peace). Then while these two arrays each contain at least two elements, we generate families as follows:

- take Parent 1 from top of *parents*
  - default: take Parent 2 from top of *parents*
  - if `MateChoiceRandom` is non-zero: take Parent 2 from any top-biased position in *parents* (using the rand(rand()) approach, see source code for details).
- take (potential) Child 1 from top of *rip*
- take Child 2 from top of *rip*

The array pointed to by *families* is filled with references to four-member arrays containing Parents 1&2 and Children 1&2. This routine does not check that *families* is empty.

3.1.12 `crossoverFamily()`

Arguments: ARRAYREF *family*
Return value: void
Defined in: TournamentGP.pm
Mainly called by: `tournament()`
Usually calls: `Individual::crossover()`, `Individual::mutate()`
Relevant attributes:

This method takes a single family (generated by `makeFamilies()`) and initiates the crossover mechanism (which is a method in the Individual class).

The two offspring from the crossover overwrite the two Individuals which were selected to “die” and `mutate()` is called on each offspring.

However, **crossover is not performed** if the fitnesses of the two parents are numerically identical. If they are identical then the second parent is subjected to mutation, and the two Individuals that would have been replaced by offspring survive.

3.1.13 `decideBetterFitness()`

Arguments: NUMBER *fitness1*, NUMBER *fitness2*
Return value: BOOLEAN
Defined in: TournamentGP.pm
Mainly called by: `tournament()`
Usually calls:
Relevant attributes: `FitnessDirection`

A simple wrapper which returns true if *fitness1* is greater than *fitness2* if the `FitnessDirection` is ‘up’. If the direction is ‘down’ then less-than is used.

3.1.14 `stopCondition()`

Arguments: none
Return value: BOOLEAN
Defined in: TournamentGP.pm
Mainly called by: `run()`
Usually calls:
Relevant attributes:

As supplied, this method always returns false, so a run will continue until `Tournaments` have been completed. If you override it to perfom some check, perhaps on `BestFitness`, then you can cleanly stop the run when some criteria have been met. Make sure you are not running perlgp-run.pl with the -loop option, or it will just start again!

3.1.15 `extraLogInfo()`

Arguments: none
Return value: STRING text
Defined in: TournamentGP.pm
Mainly called by: `tournament()`
Usually calls:
Relevant attributes:

If you need need more info about the progress of your Algorithm in ‘results/tournament.log’, then override this method to return a string which will be printed out in the log. See `parseExtraLogInfo()` before you implement this function.

3.1.16 `parseExtraLogInfo()`

Arguments: STRING lastline
Return value: void

Defined in: Tournament.pm
Mainly called by: `TournamentGP::_init()`
Usually calls:
Relevant attributes:

If you have restartable runs and you have special variables/attributes that need to persist between restarts, then you can re-initialise them from the values last written to 'results/tournament.log' by overriding this method. First make sure that the attributes are written to 'results/tournament.log' by overriding `extraLogInfo()`, but make sure each value is preceded by a unique string. Then implement `parseExtraLogInfo()` to recover those values from the last line of the log file as in the example below:

```
sub extraLogInfo {
  my $self = shift;
  return sprintf "SpecialAttrib %3d", $self->SpecialAttrib();
}

sub parseExtraLogInfo {
  my ($self, $lastline) = @_;
  if ($lastline =~ /\s+SpecialAttrib\s+(\d+)/) {
    $self->SpecialAttrib($1);
  }
}
```

See Section 6.1.2 on the dual use of attribute names as method names if the use of `SpecialAttrib()` is not clear to you.

3.2 Attributes & Variables

Attributes marked with * *must* be defined in `Algorithm::_init()` for each new application/experiment.

3.2.1 `TrainingSet*`

Data type: STRING
Default: none
Defined in: Algorithm.pm
Mainly used in: `loadData()`
See also: `TestingSet`

This is the name of the file which contains the training data and is eventually passed to `loadSet()`.

3.2.2 `TestingSet*`

Data type: STRING
Default: none
Defined in: Algorithm.pm
Mainly used in: `loadData()`
See also: `TrainingSet`

This is the name of the file which contains the testing data and is eventually passed to `loadSet()`.

3.2.3 `FitnessDirection*`

Data type: STRING
Default: none
Defined in: Algorithm.pm
Mainly used in: `tournament()`
See also: `WorstPossibleFitness`

May take the values 'up' or 'down' depending how your fitness measure works. Set it to 'up' if a big number means good fitness, and 'down' otherwise.

3.2.4 `WorstPossibleFitness`

Data type: NUMBER
Default: 0 or 1e99
Defined in: TournamentGP.pm
Mainly used in: `tournament()`, `calcFitnessFor()`
See also: `FitnessDirection`

This is set automatically in `TournamentGP::_init()` if you don't provide a value for it. See Section 3.1.2.

3.2.5 `Population`

Data type: OBJECT
Default: none
Defined in: perlgp-run.pl
Mainly used in: `tournament()`
See also:

Each Algorithm object *has a* Population object, and this is where it is stored.

3.2.6 `Tournament`

Data type: NUMBER
Default: 1
Defined in: TournamentGP.pm
Mainly used in: `run()`, `tournament()`
See also: `Tournaments`

This variable contains the current tournament number. In the case of restarted runs, it is initialised to the value in the last line of 'results/tournament.log' (see Table 1), otherwise it defaults to 1. It is incremented in `run()`.

3.2.7 `Tournaments`

Data type: NUMBER
Default: 1000
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also: `Tournament`

This is the number of tournaments that `run()` tries to run, regardless of the initial value of `Tournament` (in the case of a restart).

3.2.8 `TournamentsSinceBest`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `BestFitness`

This is a counter, like `Tournament` which is incremented every tournament, however this is reset to zero when a new best-of-run (fitness better than `BestFitness`) Individual is found.

3.2.9 `BestFitness`

Data type: NUMBER
Default: value of `WorstPossibleFitness`
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `WorstPossibleFitness`

This variable holds the fitness of the best-of-run Individual and is used for checking to see when a new best-of-run Individual is found. On restarts it is reinitialised from 'results/tournament.log'. The term "best-of-run" may not always mean exactly that; if for example the `refresh()` routine is used to dynamically change the training data.

3.2.10 `TournamentSize`

Data type: NUMBER
Default: 50
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `TournamentParents`

The number of Individuals chosen for each tournament.

3.2.11 `TournamentParents`

Data type: NUMBER
Default: 20
Defined in: TournamentGP.pm

Mainly used in: `makeFamilies()`
See also: `TournamentSize`

The number of parents given the chance to reproduce during a tournament. It's a good idea if this is an even number, and is less than or equal to half of `TournamentSize`.

3.2.12 `TournamentKillAge`

Data type: NUMBER
Default: 2
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also:

Individuals in a tournament which have taken part in `TournamentKillAge` or more tournaments are not sorted on fitness, but are automatically placed at the bottom of the list. This ensures constant turnover and is the opposite of elitism. If you find that your population is never "getting off the ground", consider raising this to 4 or maybe more. You can think of `TournamentKillAge` as the number of attempts allowed for each Individual to produce viable offspring.

3.2.13 `AlwaysEvalFitness`

Data type: BOOLEAN
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also:

When true, forces re-evaluation of fitnesses every tournament (and does not take the value from memory to save time).

3.2.14 `MateChoiceRandom`

Data type: BOOLEAN
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `makeFamilies()`
See also:

When true, the second parent (the first is always taken from the top of the fitness-sorted list of Individuals) is taken from a random position in the list, but this position is biased towards the top.

3.2.15 `TournamentLogFile`

Data type: STRING
Default: 'results/tournament.log'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `LogInterval`

The name of the main log file.

3.2.16 `LogInterval`

Data type: NUMBER
Default: 10
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `TournamentLogFile`, `FitnessesFile`, `RecentTrainingOutputFile`, `RecentTestingOutputFile`, `RecentCodeFile`

How often `TournamentLogFile`, `FitnessesFile` and some other log files are written.

3.2.17 `FitnessesFile`

Data type: STRING
Default: 'results/fitnesses'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `LogInterval`

The log file where the sorted fitnesses of the Individuals in the last tournament are saved. The file does not contain the fitnesses of the Individuals older than `TournamentKillAge`.

3.2.18 `ComplexityLogFile`

Data type: STRING
Default: 'results/complexity.log'
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also: `ComplexityInterval`

The log file where population complexity information is saved.

3.2.19 `ComplexityInterval`

Data type: NUMBER
Default: 50
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also: `ComplexityLogFile`

How often `ComplexityLogFile` is written.

3.2.20 `RefreshInterval`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also: `refresh()`

If non-zero, how often the `refresh()` method is called.

3.2.21 `RecentTrainingOutputFile`

Data type: STRING
Default: 'results/recent.training.output'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `LogInterval`

The file where the training data output from the recent best-of-tournament Individual is written (using the `saveOutput()` method).

3.2.22 `RecentTestingOutputFile`

Data type: STRING
Default: 'results/recent.testing.output'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `LogInterval`

The file where the testing data output from the recent best-of-tournament Individual is written (using the `saveOutput()` method).

3.2.23 `RecentCodeFile`

Data type: STRING
Default: 'results/recent.pl'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `LogInterval`

The file where the Perl code of the recent best-of-tournament Individual is written.

3.2.24 `BestTrainingOutputFile`

Data type: STRING
Default: 'results/best.training.output'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also:

The file where the training data output from the best-of-run Individual is written (using the `saveOutput()` method).

3.2.25 `BestTestingOutputFile`

Data type: STRING
Default: 'results/best.testing.output'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also:

The file where the testing data output from the best-of-run Individual is written (using the `saveOutput()` method).

3.2.26 `BestCodeFile`

Data type: STRING
Default: 'results/best.pl'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also:

The file where the Perl code of the best-of-run Individual is written.

3.2.27 `KeepBest`

Data type: BOOLEAN
Default: 1
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `KeepBestDir`, `KeepMax`

If non-zero, all the best-of-run Individuals are saved in `KeepBestDir`.

3.2.28 `KeepBestDir`

Data type: STRING
Default: 'results/keptbest'
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `KeepBest`, `KeepMax`

The directory in which the complete history of best-of-run Individuals may be saved.

3.2.29 `KeepMax`

Data type: NUMBER
Default: 100
Defined in: TournamentGP.pm
Mainly used in: `toOurnament()`
See also: `KeepBest`, `KeepBestDir`

If non-zero, the maximum number of best-of-run Individuals saved in `KeepBestDir`. Older Individuals are removed when the limit is reached (to save disk space).

3.2.30 `PopulationBackupInterval`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also: environment variable PERLGP_POPS

If non-zero, how often is a tar.gz file of the complete population saved to the directory PERLGP_POPS.

3.2.31 `EmigrateInterval`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also:

If non-zero, how often `Population::emigrate()` is called on `Population`.

3.2.32 `ImmigrateInterval`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `run()`
See also:

If non-zero, how often `Population::immigrate()` is called on `Population`.

3.2.33 `AlarmTime`

Data type: NUMBER
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `calcFitnessFor()`
See also: `ForkForEval`

If non-zero, the number of seconds that the system `alarm()` call is set for, before calling `evaluateOutput()`. This is a one-size-fits-all alarm time. If you want the alarm time to be proportional to the amount of training data you have, you can put alarm calls directly inside the loop (over data-points) in the evolved `evaluateOutput()` method. For example, in the sine demo in Section 9.2 you could put an `alarm(1)` just before "begin evolved bit" comment and an `alarm(0)` just after the "end evolved bit" comment.

3.2.34 `ForkForEval`

Data type: BOOLEAN
Default: 0
Defined in: TournamentGP.pm
Mainly used in: `tournament()`
See also: `AlarmTime`

When non-zero, forks the main process before calling `evaluateOutput()`. This can prevent nasty crashes with "panic: leave scope inconsistency" messages when you are using non-zero `AlarmTime` or other `alarm()` calls. Don't use it unless you get these errors.

4 Class: Individual

4.1 Methods

There is only one method in this class which must be provided/implemented by the user, and it is `evaluateOutput()` (marked with a *).

4.1.1 `new()`

Arguments: `Population =>` OBJECT, `ExperimentId =>` STRING, `DBFileStem =>` STRING, ATTRIBUTE-HASH (optional)
Return value: OBJECT
Defined in: PerlGPObject.pm
Mainly called by: perlgp-run.pl, perlgp-rand-prog.pl & other utility scripts
Usually calls: `_init()`
Relevant attributes: all

As with the Algorithm constructor, you probably don't need to call this unless you are developing specialist applications. The obligatory arguments are: *Population*, `ExperimentId` and `DBFileStem`. For some applications it is OK to give a dummy value for `Population`.

4.1.2 `_init()`

Arguments: ATTRIBUTE-HASH
Return value: void
Defined in: Individual.pm, GeneticProgram.pm, BaseIndividual.pm
Mainly called by: constructor
Usually calls: `_init()` cascade (see below)
Relevant attributes: all

As with Algorithm, this is a cascaded method where the attributes are set. The defaults are defined in `GeneticProgram::_init()` and you should customise the object by editing attribute/value pairs in `Individual::_init()` because these override all defaults.

4.1.3 `reInitialise()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `Algorithm::tournament()`, `mutate()`, `crossover()`
Usually calls: `evalEvolvedSubs()`, `evolvedInit()`
Relevant attributes:

This doesn't do much in itself, but is a very important wrapper method - to be called whenever evolved Individual-specific methods need to be redefined.

4.1.4 `evalEvolvedSubs()`

Arguments: none
Return value: void

Defined in: GeneticProgram.pm
Mainly called by: `reInitialise()`
Usually calls: `getCode()`
Relevant attributes:

This is where the Perl code is actually evaluated, in order to redefine certain Individual-specific subroutines, commonly `evaluateOutput()` and `evolvedInit()`. The code is obtained using `getCode()` and simply passed to Perl's `eval()` function. Before that, a signal handler is set to filter out "Subroutine redefined" warnings.

If an error occurs during the `eval()`, this usually means that there is a syntax error in the evolved code (usually just subroutine definitions, nothing is actually executed), caused by a mistake in the Grammar. Warnings are printed to STDERR and the Individual is randomly reinitialised with a new genome before attempting `evalEvolvedSubs()` again (with a 15 second sleep in between). The user should fix these syntax errors to make sure valid Perl is generated in all circumstances.

4.1.5 `evolvedInit()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `reInitialise()`
Usually calls:
Relevant attributes: all

The supplied method does nothing, but you can provide an evolved version to set some of the Individual's attributes. The sine curve demo in Section 9.2 shows how to have evolvable mutation and crossover rates. Note that `reInitialise()` and hence `evolvedInit()` is called inside `mutate()` and `crossover()` in anticipation that you might use evolvable mutation and crossover parameters.

4.1.6 `evaluateOutput()*`

Arguments: SCALAR *inputdata*
Return value: SCALAR *outputdata*
Defined in: Grammar.pm or Individual.pm
Mainly called by: `Algorithm::calcFitnessFor()`
Usually calls:
Relevant attributes:

This is where you should provide the code which works on *inputdata* and produces *outputdata*. Normally you would setup up the Grammar so that `evaluateOutput()` is defined in the code-tree and is therefore evolved, but you could define `evaluateOutput()` as a non-evolvable function and have it call other evolved functions.

As discussed in Section 3, the scalar variables *inputdata* and *outputdata* are usually references to arrays or hashes containing multiple data points. It is entirely up to you how you represent the data, since you are in control of reading it in (with `Algorithm::loadSet()`), generating the output (in

evaluateOutput()), assessing the fitness (in Algorithm::fitnessFunction()) and writing it out (in Algorithm::saveOutput()).

4.1.7 extraLogInfo()

Arguments: none
Return value: STRING
Defined in: GeneticProgram.pm
Mainly called by: Algorithm::tournament()
Usually calls:
Relevant attributes:

Here you can specify which extra information about the best-of-tournament Individual will be saved in 'results/tournament.log'. The default function just returns DBFileStem, but you could have it return evolved parameters, for example. The sine demo described in Section 9.2 uses this technique.

4.1.8 crossover()

Arguments: OBJECT *mate*, OBJECT *recipient1*, OBJECT *recipient2*,
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: TournamentGP::crossoverFamily()
Usually calls: reInitialise(), _tree_type_size(), _start_crossover()
Relevant attributes: NodeXoverProb, FixedXovers, FixedXoversProb, AsexualOnly, XoverDepthBias, XoverSizeBias, XoverHomologyBias, XoverLogProb

This method finds crossover points between the *self* object and the *mate* object. If FixedXovers is set and FixedXoverProb is satisfied, then FixedXovers pairs of crossover points will be selected. Otherwise the default and recommended behaviour is to use NodeXoverProb to calculated the number of crossovers to attempt, as described for in the following pseudocode:

```
crossovers_to_do = 0
for i = 1 to number_of_nodes_in_tree {
  if (rand(1) < NodeXoverProb) {
    crossovers_to_do++
  }
}
```

Crossover point selection is a little bit complex, so only an outline is given here. The general approach is to sample pairs of points until they satisfy certain criteria. To prevent infinite sampling maximum number of samples are attempted (equivalent to sampling each node 100 times). One node each from the *self* and *mate* genome trees are selected randomly (with depth bias XoverDepthBias); we will call them *mynode* and *matenode* in the following discussion. They are accepted if all the following conditions are met (in the order given):

- the nodes have the same nodetypes (e.g. NUM and NUM)

- *mynode* does not exist within the subtrees of any previously chosen crossover point in *self*, and likewise *matenode* not in any crossover point subtree of *mate*
- similar-size probability: $\left(1 - \left(\frac{|N_A - N_B|}{\max(N_A, N_B)}\right)^s\right)$ where N_A and N_B are the number of nodes in the two subtrees and s is `XoverSizeBias`
- similar-contents (homology) probability: $\left(\frac{I_{A,B}+0.1}{\min(N_A, N_B)}\right)^h$ where $I_{A,B}$ is the number of identical nodes found when decending the two subtrees in parallel (not allowing any insertions or deletions, see `_tree_id()`) and h is `XoverHomologyBias`
- none of the subtrees of the two crossover points contain previously selected crossover points

After the crossover points have been selected, they are passed to `_start_crossover()`. If no crossover points were found (or if none were requested) then the two parents are crossed over at the root node, which is equivalent to asexual reproduction. You can also set `AsexualOnly` to force asexual reproduction.

With probability `XoverLogProb`, some information about crossover points is written to `XoverLogFile`.

4.1.9 `_start_crossover()`

Arguments: OBJECT *mate*, OBJECT *recipient*, HASH-REF *crossover_pairs*
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `crossover()`
Usually calls: `_crossover()`, `retieGenome()`, `_fix_nodes()`
Relevant attributes:

A wrapper around the recursive `_crossover()` method, which first does a `tieGenome()` on the recipient and wipes its genome.

4.1.10 `_crossover()`

Arguments: OBJECT *mate*, OBJECT *recipient*, STRING *mynode*, STRING *matenode*, HASH-REF *crossover_pairs*,
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `_start_crossover`
Usually calls: recursive
Relevant attributes:

Recursive subroutine which builds up a new genome tree for *recipient* by tracing through the *self* genomes and switching over to the *mate* genome when a *crossover_pair* is met. To avoid name clashes mate nodes are copied with names appended with an "x" (see also `_xcopy_subtree()`) and these are tidied up by `_fix_nodes()` on returning to `_start_crossover()`.

4.1.11 `mutate()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `Algorithm::crossoverFamily()`
Usually calls: `reInitialise()`, `point_mutate_shallow()`, `macro_mutate()`, `initFitness()`
Relevant attributes: `NodeMutationProb`, `FixedMutations`, `FixedMutationProb`, `PointMutationFrac`, `NoNeutralMutations`, `MutationLogProb`, `MutationLogFile`

This method does one round of mutation on an Individual. First the number of mutations to be attempted is calculated according to the following pseudocode:

```
mutations_to_do = 0
for i = 1 to number_of_nodes_in_tree {
  if (rand(1) < NodeMutationProb) {
    mutations_to_do++
  }
}
```

However if `FixedMutations` is non-zero and `FixedMutationProb` is satisfied, then `FixedMutations` are attempted.

Each mutation is randomly chosen to be either point mutation (via method `point_mutate_shallow()`) or a macro mutation (via method `macro_mutate()`), according to the attribute `PointMutationFrac`.

If `NoNeutralMutations` is non-zero, then a check is made that the mutation made a visible (not necessarily functional) change to the Perl code. This is done by expanding the genome-tree before each mutation, and after each mutation and comparing the two strings. If the strings are identical then another mutation is attempted (up to a safety limit of 5000 tries). This feature is turned off by default.

Note that we say that mutations are *attempted* rather than performed. That is because some of the mutation operators fail to find suitable nodes to act on, and as a result do nothing.

After the mutations are are done, some logging is done with probability `MutationLogProb` into a file `MutationLogFile`, with some simple information about the nodes that were mutated (if they still exist - when more than one mutation is attempted, the subsequent mutations can delete the nodes affected by previous mutations).

If more than one mutation was attempted, then the cached fitness of the Individual is reset with a call to `initFitness()`.

4.1.12 `point_mutate_shallow()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `mutate()`

Usually calls: `point_mutate()`
Relevant attributes: `PointMutationDepthBias`

Simple wrapper to call `point_mutate()` with depth bias (`PointMutationDepthBias`) more in favour of leaves.

4.1.13 `point_mutate_deep()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `mutate()`
Usually calls: `point_mutate()`
Relevant attributes: `MacroMutationDepthBias`

Simple wrapper to call `point_mutate()` with depth bias (`MacroMutationDepthBias`) more in favour of internal nodes.

4.1.14 `point_mutate()`

Arguments: NUMBER *depthbias*
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `point_mutate_shallow()`, `point_mutate_deep()`
Usually calls: `_random_node()`
Relevant attributes: `NumericMutationFrac`, `NumericIgnoreNTypes`, `NumericAllowNTypes`, `NumericMutationRegex`

Attempts to make a point mutation, that is a change in the genome-tree which does not affect the branching pattern and involves just one node. The first thing it does is select a random node with depth bias *depthbias*.

4.1.15 `_random_node()`

Arguments: `depth_bias =>` NUMBER, `start_node =>` STRING (optional), `not_this_node =>` STRING (optional), `not_this_subtree =>` STRING (optional), `node_type =>` STRING
Return value: STRING *nodekey*
Defined in: GeneticProgram.pm
Mainly called by: mutation and crossover operators
Usually calls:
Relevant attributes:

Picks one node from the genome-tree according to the following pseudocode:

```
N = number_of_nodes_below(root_of_tree)
do {
  node = pick_a_random_node_from_tree
  S = number_of_nodes_below(node)
} until rand(depth_bias * N) <= S
```

In this way, it is possible to vary the amount of bias towards internal nodes. With a `depth_bias` of zero, all nodes are selected with equal probability, but with a `depth_bias` of 1 only the root node will be selected with absolute certainty. Of course, it is not very time-efficient to sample nodes in this way.

Other optional parameters allow a specific node type to be selected, or nodes within or outside a certain subtree. If a node cannot be found after 5000 tries, the routine gives up and returns the empty string.

4.1.16 `macro_mutate()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `mutate()`
Usually calls: macro mutation operators
Relevant attributes: `MacroMutationTypes`

Picks at random one of the available macro mutation operator names from the array `MacroMutationTypes` and invokes the method of that name (in Perl you can call a method using a string variable like this: `$method = 'swap_subtrees'; $individual->$method()`).

4.1.17 `replace_subtree()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`, `_del_subtree()`, `_grow_tree()`
Relevant attributes: `NewSubtreeDepthMean`, `NewSubtreeDepthMax`

Picks a random node with `MacroMutationDepthBias` and deletes the subtree and "grows" a new subtree in its place.

4.1.18 `insert_internal()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`, `_random_function()`, `_grow_tree()`
Relevant attributes: `MacroMutationDepthBias`, `NewSubtreeDepthMean`, `NewSubtreeDepthMax`

Inserts a single new function node internally in a tree.

First we pick a random node with `MacroMutationDepthBias` and copy its contents to a brand new node. The original node is then replaced with a random function with at least one branch point of the same type as the original node (if none exists, then we give up). We link up one of these branch points back to the new node, and any remaining branch points have new subtrees grown onto them. Figure 1 illustrates this complicated procedure.

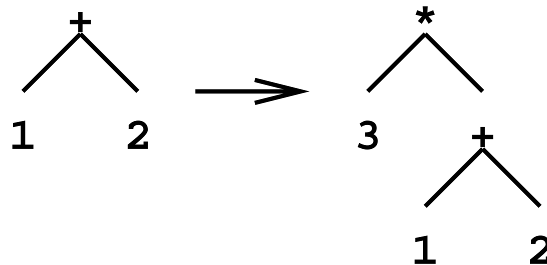


Figure 1: Example of the macro mutation operator `insert_internal`. The '+' node is chosen for mutation, and is moved to a new location. A new node '*' is inserted in the old position and one of its two branch points is connected back to the '+'. The remaining branch point for the '*' function is filled out with a new tree, which in this case is just a single node '3'.

4.1.19 `delete_internal()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`
Relevant attributes: `MacroMutationDepthBias`

Selects a node with `MacroMutationDepthBias`, and then, if possible, a second node of the same type *within* the subtree of the first. The two nodes are reconnected and the intervening nodes are deleted (actually it is implemented in another way, but the result is the same).

4.1.20 `copy_subtree()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`, `_del_subtree()`
Relevant attributes: `MacroMutationDepthBias`

Selects a node with `MacroMutationDepthBias`, and then, if possible, a second node of the same type *outside* the subtree of the first. Deletes any sub-nodes belonging to the second node and copy the first node's subtree to the second node.

4.1.21 `swap_subtrees()`

Arguments: none
Return value: void

Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`
Relevant attributes: `MacroMutationDepthBias`

Like `copy_subtree`, except that no nodes are deleted, the subtrees exchange positions.

4.1.22 `encapsulate_subtree()`

Arguments: none
Return value: void
Defined in: GeneticProgram.pm
Mainly called by: `macro_mutate()`
Usually calls: `_random_node()`, `_expand_tree()`, `simplify()`
Relevant attributes: `MacroMutationDepthBias`, `EncapsulateIgnoreNTypes`, `EncapsulateFracMax`
Relevant methods: `getSize()`

An experimental, *not fully tested* method which takes a subtree (which represents not more than fraction `EncapsulateFracMax` of the whole tree) and replaces it with a single terminal node containing the expanded code of the former subtree. The code is passed through `simplify()` to remove any redundancy (if you implement the method). The new node is tagged with a special prefix string ';;NUM;;' where 'NUM' is the size of the subtree before encapsulation. This number is used in `_tree_type_size()` (which is sometimes called by `getSize()`) during tree size calculation to make it fair when parsimony is used. As I said, this needs a thorough investigation because things have changed since I last used encapsulation, and it is a pretty complicated operator with many side effects.

4.1.23 `simplify()`

Arguments: STRING *longcode*
Return value: STRING `shortercode`
Defined in: GeneticProgram.pm
Mainly called by: `encapsulate_subtree()`
Usually calls:
Relevant attributes:

The default method does nothing, but you can define a method to do search-and-replace-with-eval simplification, for example:

```
sub simplify {
  my ($self, $code) = @_;
  my $z = 0;
  while ($code =~ s/(\d+ \+ \d+)/eval $1/e) {
    last if ($z++ > 1000); # safety measure
  }
  return $code;
}
```

will simplify '1 + 2 + 3 + 4' to '10'. This can be very powerful for creating human-readable output.

4.1.24 `_xcopy_subtree()`

Arguments: STRING *nodekey*
Return value:
Defined in: GeneticProgram.pm
Mainly called by:
Usually calls: recursive
Relevant methods: `_fix_nodes()`

A helper method which takes a node and generates a copy of its subtree with new nodekeys which are the same as the original tree plus a trailing 'x' (so 'nodeNUM23' is copied to 'nodeNUM23x'). This is a simple way to avoid node key clashes, as long as you call `_fix_nodes()` afterwards...

4.1.25 `_fix_nodes()`

Arguments: STRING *nodekey*
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `delete_internal()`, `copy_subtree()`, `_start_crossover()`
Usually calls: recursive
Relevant attributes:

Recursive method to fix the nodekeys which end in 'x' and replace them with valid nodekeys which don't end in 'x'. It searches incrementally for new unused nodekeys from zero (in other words, 'nodeNUM14x' is replaced by 'nodeNUM0' and if that already exists, then 'nodeNUM1', 'nodeNUM2' ... until a free key is found).

4.1.26 `_get_subnodes()`

Arguments: STRING *nodekey*
Return value: ARRAY *subnodes*
Defined in: GeneticProgram.pm
Mainly called by: `crossover()`, `_random_node()`, `_copy_subtree()`
Usually calls: recursive
Relevant attributes:

Returns the nodekeys of all subnodes of the given *nodekey*.

4.1.27 `_del_subtree()`

Arguments: STRING *nodekey*
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `_replace_subtree()`, `_delete_internal()`, `_copy_subtree()`, `encapsulate_subtree()`
Usually calls: recursive
Relevant attributes:

Recursive method which "on the way out" deletes all the nodes in the given subtree.

4.1.28 `_tree_type_size()`

Arguments: STRING *nodekey*, HASHREF *sizes* (optional), HASHREF *types* (optional), HASHREF *ignore* (optional)
Return value: NUMBER *nodes*
Defined in: GeneticProgram.pm
Mainly called by: `getSize()`, `crossover()`, `mutate()`
Usually calls: recursive
Relevant attributes:

This method descends a subtree (usually 'root') and fills a number of hashes (actually the hash references given as arguments: *sizes*, *types*) with the size of the subtree below each node, and the nodetype of each node (the hash keys are nodekeys).

If the *ignore* hash reference is defined, then this routine will not descend subtrees of any nodetype which exists as a key in the *ignore* hash.

The methods `crossover()` and `mutate()` efficiently make use of the cached size information when sampling nodes and repeated tree recursion is not needed.

4.1.29 `_display_tree()`

Arguments: STRING *nodekey*
Return value:
Defined in: GeneticProgram.pm
Mainly called by:
Usually calls:
Relevant attributes:

Prints a text dump of the nodes below *nodekey* to STDERR for debugging purposes.

4.1.30 `saveCode()`

Arguments: `Filename =>` STRING, `Tournament =>` NUMBER
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `TournamentGP::tournament()`
Usually calls: `Fitness()`, `getCode()`, `getSize()`
Relevant attributes:

Prints the Perl code and some other relevant information to `Filename` for the user.

4.1.31 `saveTree()`

Arguments: `Filename =>` STRING, `StartNode =>` STRING, `HighLight =>` HASH-REF
Return value:
Defined in: GeneticProgram.pm

Mainly called by: `TournamentGP::tournament()`
Usually calls:
Relevant attributes:

Prints the genome tree to `Filename` in a format readable by the third party program daVinci. The optional hash reference `HighLight` can be used to give certain background colours to certain nodes (key = node-id, value = hex colour).

4.1.32 `save()`

Arguments: `FileStem =>` STRING
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `TournamentGP::tournament()`, `GPPopulation::emigrate()`
Usually calls: `tieGenome()`, `untieGenome()`
Relevant attributes:

Copies the genome hash to a new DB file with `FileStem`.

4.1.33 `load()`

Arguments: `FileStem =>` STRING
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `GPPopulation::immigrate()`
Usually calls: `tieGenome()`, `untieGenome()`
Relevant attributes:

Loads up a genome hash from a DB file with `FileStem`. Erases the current genome, but consider making a call to `reInitialise()` to update evolved parameters afterwards.

4.1.34 `tieGenome()`

Arguments:
Return value: HASH-REF
Defined in: GeneticProgram.pm
Mainly called by: tree operating methods
Usually calls:
Relevant attributes: `DBFileStem`

By default, an Individual is not "connected" to the DBM file which stores the genome. If it was, then a typical run would have thousands of open filehandles and would probably crash. This method uses Perl's `tie()` to tie the hash stored in `$self->{genome}` to the file(s) identified by `DBFileStem`. This is done in read/write/create mode. See the Perl documentation for `tie()` for more details.

The return value is a reference to the genome hash, or you can use `$self->{genome}` if you prefer.

Make sure you *always* have a corresponding call to `untieGenome()` to avoid filehandle overload. Symmetrical nesting of ties and unties is allowed - if the

genome is already tied a counter is incremented in `tieGenome()` which is decremented by `untieGenome()` - the actual untie is done only at "depth zero".

Note that you don't want to have to many tie/untie tie/untie tie/untie sequences (big performance hit), it's better to wrap a tie...untie around them all. This is why there are `tieGenome()` calls in `TournamentGP::crossoverFamily()`. If you can't figure out where/when/why various tie/untie calls are not made, you can uncomment the debugging line and watch STDERR.

If you want to completely wipe the genome, see the source code for `_start_crossover()`.

4.1.35 `untieGenome()`

Arguments: STRING *debug*
Return value:
Defined in: GeneticProgram.pm
Mainly called by: tree operating methods
Usually calls:
Relevant attributes:

See `tieGenome()`.

4.1.36 `retieGenome()`

Arguments:
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `_start_crossover()`
Usually calls:
Relevant attributes: `DBFileStem`

The DBM files seem to grow with time, so this method will copy the genome hash into memory, and then delete the DBM files, and rewrite a new one. Obviously this is a bit risky since there is no backup made on disk.

Currently only called in `_start_crossover()` on the recipient/child after wiping the genome.

Does not affect the tie level counter (see `tieGenome()`).

4.1.37 `initTree()`

Arguments:
Return value:
Defined in: GeneticProgram.pm
Mainly called by:
Usually calls: `_init_tree()`
Relevant attributes:

This is a simple wrapper around `_init_tree()` which does `tieGenome()` first and `untieGenome()` afterwards.

4.1.38 `_init_tree()`

Arguments:
Return value:
Defined in: GeneticProgram.pm
Mainly called by:
Usually calls: `_grow_tree()`
Relevant attributes: `TreeDepthMax`, `MinTreeNodes`

Initialises the genome hash and starts building a new tree of max depth `TreeDepthMax`. Repeats this until the number of nodes is greater than `MinTreeNodes`.

4.1.39 `_grow_tree()`

Arguments: `depth =>` INTEGER, `type =>` STRING, `TreeDepthMax =>` INTEGER
Return value: STRING *new_nodekey*
Defined in: GeneticProgram.pm
Mainly called by: macro mutation operators, `_init_tree()`
Usually calls: recursive, `_random_terminal()`, `_random_function`
Relevant attributes: `TreeDepthMin`, `TerminateTreeProb`

Builds a tree or subtree by randomly selecting functions and terminals. `TreeDepthMax` is passed as an argument (the object attribute of the same name is not used) because you might want to use different limits for the whole tree *vs.* subtrees. The `depth` argument is incremented on each recursive call. Usually a function is chosen during tree building but if the depth is greater than `TreeDepthMax` then a terminal is always added. Alternatively you may use `TreeDepthMin` *and* `TerminateTreeProb` to force termination with some probability below a vertain depth.

4.1.40 `_expand_tree()`

Arguments: STRING *nodekey*
Return value: STRING *perl_code*
Defined in: GeneticProgram.pm
Mainly called by: `getCode()`, `mutate()`, `encapsulate_subtree()`
Usually calls:
Relevant attributes: `MaxTreeNodes`

This method expands the tree below *nodekey* into a single string which I presumably Perl code (or at least something that will `eval()`). It is done with an iterated search-and-replace. A counter keeps track of how many nodes have been expanded and if this is greater than `MaxTreeNodes` then no more tree expansion is done and random terminals are used to "finish off" the expansion to generate valid, but presumably unfit, Perl code.

You should probably use the public method `getCode()`, which wraps this.

4.1.41 `_tree_id()`

Arguments: OBJECT *mate*, STRING *mynode*, STRING *matenode*
Return value: NUMBER identities

Defined in: GeneticProgram.pm
Mainly called by: `crossover()`
Usually calls: recursive
Relevant attributes:

Recursive tree-descending method which goes down two subtrees and counts the number of nodes identical between them at equivalent topological positions. It stops descending whenever nodes are unequal, therefore it is quite crude and cannot handle insertions or deletions.

4.1.42 `initFitness()`

Arguments:
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `mutate()`
Usually calls:
Relevant attributes:

Deletes the two cached copies of the fitness (one in the genome, one in memory).

4.1.43 `eraseMemory()`

Arguments:
Return value:
Defined in: GeneticProgram.pm
Mainly called by: `_start_crossover()`
Usually calls:
Relevant attributes:

Deletes all contents of the `$self->{memory}` hash, which contains the cached fitness value, but you may also store other things here, using the `memory()` method.

4.1.44 `memory()`

Arguments: HASH-KEY or HASH
Return value: value
Defined in: GeneticProgram.pm
Mainly called by:
Usually calls:
Relevant attributes:

A get and set method for the `$self->{memory}` hash. With a single argument, it returns the value stored with that key. With an even number of arguments (in other words, a hash) it sets the key/value pairs.

If you prefer, you can use `getMemory()` and `setMemory()` which are simply wrappers to this method.

4.1.45 `getCode()`

Arguments:
Return value: STRING *perl_code*
Defined in: GeneticProgram.pm
Mainly called by: `evalEvolvedSubs()`, `saveCode()`
Usually calls:
Relevant attributes:

This is the main method for getting the Perl code from the genome.

4.1.46 `getSize()`

Arguments:
Return value: NUMBER *code_size*
Defined in: GeneticProgram.pm
Mainly called by: `TournamentGP::calcFitnessFor()`
Usually calls:
Relevant attributes: `GetSizeIgnoreNTypes`

This is the method for calculating the size of the genome. If the attribute `GetSizeIgnoreNTypes` is used then `_tree_type_size()` is used to calculate the number of nodes by descent (ignoring nodes which lie below the nodetypes defined by `GetSizeIgnoreNTypes`). Otherwise, the number of keys in the genome hash is returned (which is much quicker of course).

4.1.47 `Fitness()`

Arguments: NUMBER *set_value* (optional)
Return value: NUMBER *fitness*
Defined in: GeneticProgram.pm
Mainly called by: `TournamentGP::tournament()`,
`TournamentGP::crossoverFamily()`
Usually calls:
Relevant attributes:

Without an argument routine returns the Individual's `Fitness`, first from the memory hash (see `memory()`) or if that is undefined, then a `tieGenome()` is called and the fitness value is retrieved from disk, assigned into the memory cache, and returned. Note that "undefined" value for `Fitness` usually signifies that that the fitness has not yet been evaluated, but sometimes just means that the fitness has not been retrieved from disk.

With an argument, a new value for `Fitness` is stored in memory and on disk.

4.1.48 `Age()`

Arguments: NUMBER *increment* (optional)
Return value: NUMBER
Defined in: GeneticProgram.pm
Mainly called by: `TournamentGP::tournament()`
Usually calls:
Relevant attributes:

Without an argument, it returns the "age" of an Individual, which usually means the number of tournaments participated. It is stored only in the memory cache (see `memory()`, and *not on disk*.

4.1.49 `_random_terminal()`

Arguments: STRING *nodetype*
Return value: STRING
Defined in: GeneticProgram.pm
Mainly called by: `_grow_tree()`, `_random_function()`
Usually calls: `_random_existing_terminal()`
Relevant attributes: `Terminals`

This routine first calls `_random_existing_terminal()` and returns its result if true, otherwise it returns one random element from the array stored in the hash pointed to by `Terminals` using the hash-key *nodetype*.

4.1.50 `_random_existing_terminal()`

Arguments: STRING *nodetype*, BOOLEAN *encapsulated* (optional)
Return value: STRING
Defined in: GeneticProgram.pm
Mainly called by: `_random_terminal()`, `_random_function()`
Usually calls:
Relevant attributes: `UseExistingTerminalsFrac`, `UseEncapsTerminalsFrac`

If you have numeric mutation switched on, perhaps you want to use existing refined numeric terminals during the construction of new subtrees. If so, you can set `UseExistingTerminalsFrac` to a fraction/probability, then this method will sometimes return a randomly chosen existing terminal of type *nodetype* (if one exists) from the genome tree of this organism. For the random selection, each unique terminal is counted only once (to prevent saturation of that terminal).

If `UseEncapsTerminalsFrac` is non-zero and if the optional argument *encapsulated* is true, then with this probability, the method will return only terminals of the requested type which were generated by encapsulation (these are tagged with ";;NUM;;", see `encapsulate_subtree`). If none are found, an empty string is returned.

4.1.51 `_random_function()`

Arguments: STRING *nodetype*
Return value: STRING
Defined in: GeneticProgram.pm
Mainly called by: `_grow_tree()`
Usually calls: `_random_terminal()`, `_random_existing_terminal()`
Relevant attributes: `Functions`, `UseEncapsTerminalsFrac`

If no set of functions of type *nodetype* is defined in the `Functions`, then this function passes through to `_random_terminal()`. Otherwise, with probability `UseEncapsTerminalsFrac` a random encapsulated terminal is attempted to be chosen through a call to `_random_existing_terminal()`.

But the default behaviour is, of course, to return a random function of type *nodetype* from the `Functions` hash.

4.2 Attributes & Variables

4.2.1 `NodeMutationProb`

Data type: NUMBER
Default: 1/100
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also:

The probability that each node in the genome-tree will be subjected to mutation (see `mutate()`).

4.2.2 `FixedMutations`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also: `FixedMutationProb`

If non-zero, then a fixed number of mutations (equal to `FixedMutations`) will be attempted, regardless of genome size. This attribute overrides any setting of `NodeMutationProb`.

4.2.3 `FixedMutationProb`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also: `FixedMutations`

The probability with which `FixedMutations` will be performed. A setting of 0.1 would mean that 10% of the offspring from crossover are mutated.

4.2.4 `PointMutationFrac`

Data type: NUMBER
Default: 0.7
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also:

A mutation is either a point mutation or a macro mutation, this attribute controls how much of either. A setting of 1 means all point mutation and 0 means all macro mutation.

4.2.5 `NumericMutationFrac`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `NumericIgnoreNTypes`, `NumericAllowNTypes`,
`NumericMutationRegex`

With this probability, point mutations on numbers (satisfying `NumericMutationRegex`) will be altered instead of replaced.

4.2.6 `NumericMutationRegex`

Data type: REGEX
Default: `qr/^[+-]?\d+(?:\.\d+)?([eE][+-]?\d+)?$/`
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `NumericMutationFrac`, `NumericIgnoreNTypes`,
`NumericAllowNTypes`

Numeric mutations will only be done if this pattern matches the terminal node contents.

4.2.7 `NumericIgnoreNTypes`

Data type: HASH-REF
Default: empty
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `NumericAllowNTypes()`

Define the keys in this hash as the nodetypes you don't want to be numerically mutated, for example:

```
NumericAllowNTypes => { ANGLE => 1, INDEX => 1 },
```

By default, all other terminals which match `NumericMutationRegex` will be mutated, with the default change factor of 0.1 (see `NumericAllowNTypes`).

4.2.8 `NumericAllowNTypes`

Data type: HASH-REF
Default: empty
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `NumericIgnoreNTypes()`

Define the keys in this hash as the nodetypes you want to be numerically mutated, for example:

```
NumericAllowNTypes => { NUM => 1, CONST => 2 },
```

Where the values indicate how much change occurs on mutation. The change is calculated as:

```
if (rand(1)<0.5) {
  $num *= (1 + $changefactor);
} else {
  $num /= (1 + $changefactor);
}
```

where `$changefactor` is the value given in your hash.

4.2.9 `NoNeutralMutations`

Data type: BOOLEAN
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also:

If true, mutations will be repeated until a visible change in the program code is observed. This involves tree expansion and is therefore not very efficient. Effects not properly benchmarked.

4.2.10 `PointMutationDepthBias`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `MacroMutationDepthBias`, `XoverDepthBias`

Depth bias for point mutation node sampling. See `_random_node()` for details. The default of 0 means no bias at all (and so terminal nodes are relatively likely).

4.2.11 `MacroMutationDepthBias`

Data type: NUMBER
Default: 0.7
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `PointMutationDepthBias`, `XoverDepthBias`

Depth bias for macro mutation node sampling. See `_random_node()` for details.

4.2.12 `MacroMutationTypes`

Data type: ARRAY-REF
Default: see below
Defined in: GeneticProgram.pm
Mainly used in: `macro_mutate()`
See also:

This array contains the names of the macro mutation operators which may be used. An example of a name is `swap_subtrees` (do not include the `()` which is used in this manual for purposes of clarity only). Choice of operator is done in `macro_mutate()` by random sampling, so if you want to bias a certain operator, include its name in this array more than once.

The default array contains a single copy each of: `swap_subtrees` `copy_subtree` `replace_subtree` `insert_internal` `delete_internal`

Note that `encapsulate_subtree` and `point_mutate_deep` are not included by default.

4.2.13 `EncapsulateIgnoreNTypes`

Data type: HASH-REF
Default: empty
Defined in: GeneticProgram.pm
Mainly used in: `encapsulate_subtree()`
See also: `EncapsulateFracMax`

Don't allow these nodetypes (the keys in the hash) to be encapsulated. The values are not used. Example definition:

```
EncapsulateIgnoreNTypes => { ADF => 1 },
```

4.2.14 `EncapsulateFracMax`

Data type: NUMBER
Default: 0.25
Defined in: GeneticProgram.pm
Mainly used in: `encapsulate_subtree()`
See also:

During node sampling in `encapsulate_subtree()`, subtrees are skipped if they contain more than `EncapsulateFracMax` of the total nodes in the tree. This is intended to prevent too much of a program being encapsulated. Not benchmarked!

4.2.15 `UseEncapsTerminalsFrac`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `_random_function()`
See also:

The probability that an already encapsulated subtree will be used instead of a *function*, during new subtree growth. Not benchmarked!

4.2.16 `MutationLogFile`

Data type: STRING
Default: 'results/mutation.log'
Defined in: GeneticProgram.pm

Mainly used in: `mutate()`
See also: `MutationLogProb`

The filename for the mutation log. Relative to the experiment directory.

4.2.17 `MutationLogProb`

Data type: NUMBER
Default: 0.02
Defined in: GeneticProgram.pm
Mainly used in: `mutate()`
See also: `MutationLogFile`

The probability with which a log entry is written to `MutationLogFile`.

4.2.18 `NodeXoverProb`

Data type: NUMBER
Default: 1/50
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `FixedXovers`, `FixedXoverProb`

The per node crossover point selection probability.

4.2.19 `FixedXovers`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `FixedXoverProb`

If non-zero, then a fixed number of crossover points (equal to `FixedXovers`) will be attempted, regardless of genome size. This attribute overrides any setting of `NodeXoverProb`.

4.2.20 `FixedXoverProb`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `FixedXovers`

The probability with which `FixedXovers` crossover points will be attempted during reproduction (otherwise no crossovers will be attempted, which is the same as asexual reproduction).

4.2.21 `XoverDepthBias`

Data type: NUMBER
Default: 0.1
Defined in: GeneticProgram.pm
Mainly used in: `point_mutate()`
See also: `PointMutationDepthBias`, `MacroMutationDepthBias`

Depth bias for crossover point sampling. See `_random_node()` for details. If the depth wasn't biased, then too often crossovers would take place between "uninteresting" leaf nodes and small subtrees.

4.2.22 `XoverSizeBias`

Data type: NUMBER (must not be zero)
Default: 1
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `XoverHomologyBias`

This controls the strictness of size-equal crossover point selection. By size-equal, I mean that crossover points are selected with a bias towards similar sized subtrees. Increase this (say to 2 or 4) and the algorithm described in `crossover()` spends longer looking for similar sized subtrees.

4.2.23 `XoverHomologyBias`

Data type: NUMBER (must not be zero)
Default: 1
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `XoverSizeBias`

This controls the strictness of homologous subtree crossover point selection. Increase this value to 2 or 4, for example, to tell the algorithm described in `crossover()` to spend longer looking for crossover points with similar subtree contents.

4.2.24 `AsexualOnly`

Data type: BOOLEAN
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also:

If true, force all crossovers to use the root node, and hence make identical copies of both parents. All other crossover parameters are ignored.

4.2.25 `XoverLogFile`

Data type: STRING
Default: 'results/crossover.log'
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `XoverLogProb`

The file (relative to the experiment directory) where some information about crossover point selection is logged.

4.2.26 `XoverLogProb`

Data type: NUMBER
Default: 1/50
Defined in: GeneticProgram.pm
Mainly used in: `crossover()`
See also: `XoverLogFile`

The fraction of recombination events for which `XoverLogFile` is written to.

4.2.27 `MaxTreeNodes`

Data type: NUMBER
Default: 1000
Defined in: GeneticProgram.pm
Mainly used in: `_expand_tree()`
See also: `MinTreeNodes`, `TreeDepthMax`, `TreeDepthMin`, `TerminateTreeProb`

Controls the maximum allowable tree size. The way this works is counter-intuitive so pay attention... Trees are allowed to be bigger than this size limit, however when the genome tree is converted into code, only the first `MaxTreeNodes` are expanded properly into what the "code for", all subsequent nodes are expanded into a randomly picked terminal (see `_expand_tree()`). This probably gives an disfunctional program and so indirectly prevents tree growth beyond this size. *Note that this is really just a safety limit, not a recommended way to control code growth.* Don't forget about `MaxTreeNodes` though, one day you will probably need to raise it.

4.2.28 `MinTreeNodes`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `_init_tree()`
See also: `MaxTreeNodes`, `TreeDepthMax`, `TreeDepthMin`, `TerminateTreeProb`

This attribute defines a minimum tree size (in nodes) *for new trees only.* If a brand new random tree (usually made during Population initialisation) has fewer than this number of nodes, then the tree is discarded and made again (and repeated until a big enough tree is generated). Currently this is implemented without a check for infinite looping, so watch out.

This attribute does not affect subtree generation.

4.2.29 `TreeDepthMax`

Data type: NUMBER
Default: 20
Defined in: GeneticProgram.pm
Mainly used in: `_init_tree()`, `_grow_tree()`
See also: `TreeDepthMin`, `TerminateTreeProb`, `MaxTreeNodes`, `MinTreeNodes`, `NewSubtreeDepthMax`

A fixed limit on tree depth for brand new trees only. When a growing branch of a tree reaches this limit, only terminals are added (like in Koza's and other GP implementations). Note that this is different to the implementation of `MaxTreeNodes`.

The default is just a safety limit because it is assumed that you are using a "naturally terminating grammar".

4.2.30 `TerminateTreeProb`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `_grow_tree()`
See also: `TreeDepthMin`

You may force the choice of terminals during new tree and subtree generation by setting this probability. It only has an effect at tree depths below `TreeDepthMin`. Note the default is zero.

4.2.31 `TreeDepthMin`

Data type: NUMBER
Default: 1
Defined in: GeneticProgram.pm
Mainly used in: `_grow_tree()`
See also: `TerminateTreeProb`

The minimum depth before tree growth is probabilistically terminated with `TerminateTreeProb`.

4.2.32 `NewSubtreeDepthMean`

Data type: NUMBER
Default: 20
Defined in: GeneticProgram.pm
Mainly used in: macro mutation operators
See also: `NewSubtreeDepthMax`

Many of the macro mutation operators need to make new subtrees. They call `_grow_tree()` with a specific tree depth maximum which is generated from a capped Poisson distribution with the mean parameter `NewSubtreeDepthMean`, and cap `NewSubtreeDepthMax`. The default setting is quite high and assumes that you have a "naturally terminating grammar".

4.2.33 `NewSubtreeDepthMax`

Data type: NUMBER
Default: 20
Defined in: GeneticProgram.pm
Mainly used in: macro mutation operators
See also: `NewSubtreeDepthMean`

A hard upper limit on the size of new subtrees (see `NewSubtreeDepthMean`).

4.2.34 `UseExistingTerminalsFrac`

Data type: NUMBER
Default: 0
Defined in: GeneticProgram.pm
Mainly used in: `_random_existing_terminal()`
See also: `NumericMutationFrac`, `UseEncapsTerminalsFrac`

The fraction or probability with which existing terminals (already in the genome tree) are used instead of the usual pool of terminals held in `Terminals`. Usually you will want to combine this with numeric mutation. See also `point_mutate()`.

4.2.35 `GetSizeIgnoreNTypes`

Data type: HASH-REF
Default: empty
Defined in: GeneticProgram.pm
Mainly used in: `getSize()`
See also:

If you are using parsimony pressure in your fitness function, you might want `getSize()` to ignore the trees below certain nodetypes. In this case you can specify these nodetypes in this hash.

4.2.36 `DBFileStem`

Data type: STRING
Default: none, required attribute
Defined in: GeneticProgram.pm
Mainly used in: `tieGenome()`
See also: `Population::findNewDBFileStem()`

This is the file stem for the DBM file where the genome tree is stored. It is a required attribute. See the program perlgp-rand-prog.pl to see how you can make a random program using a temporary `DBFileStem`.

4.2.37 `ExperimentId`

Data type: STRING
Default: none, required attribute
Defined in: GeneticProgram.pm

Mainly used in: perlgp-run.pl
See also:

This is the *name* of the current experiment directory (not the full path).

4.2.38 `Population`

Data type: OBJECT
Default: none, required attribute
Defined in: GeneticProgram.pm
Mainly used in: not really used!
See also: perlgp-rand-prog.pl

Each Individual knows to which Population it belongs. Currently this is not actually used anywhere. Since it is compulsory, you have to proivide a dummy value in applications such as perlgp-rand-prog.pl, sorry about that - I guess this will change.

4.2.39 `Functions`

Data type: HASH-REF
Default: `\%Grammar::F`
Defined in: GeneticProgram.pm
Mainly used in: `_random_function()`
See also: `Terminals`

This is where the non-terminal grammar definitions are stored. In the hash, keys are nodetypes and the values are anonymous arrays containing different function options. More details of the format used can be found in Section 7. Usually this hash is shared between all Individual objects (otherwise memory usage would be heavy), hence the default is a reference to the hash `%Grammar::F`.

You could customise the Individual class to dynamically alter `Functions` and `Terminals` if you wanted to.

4.2.40 `Terminals`

Data type: HASH-REF
Default: `\%Grammar::T`
Defined in: GeneticProgram.pm
Mainly used in: `_random_terminal()`
See also: `Functions`

The terminal node options are stored here, see the entry for `Functions`, and also Section 7.

5 Class: Population

5.1 Methods

5.1.1 `new()`

Arguments: `ExperimentId =>` STRING, ATTRIBUTE-HASH (optional)

Return value: OBJECT
Defined in: PerlGPObject.pm
Mainly called by: perlgp-run.pl
Usually calls: `_init()`
Relevant attributes: all

The constructor for the Population class. You may customise the object by setting attributes as arguments to the constructor or edit `_init()` in Population.pm.

5.1.2 `_init()`

Arguments: ATTRIBUTE-HASH
Return value:
Defined in: GPPopulation.pm
Mainly called by:
Usually calls:
Relevant attributes:

A standard cascading `_init()` routine, which sets attributes. It also creates `PopulationDir` if it doesn't exist. If you want to change the attributes, edit the hash `%defaults` in `_init()` in Population.pm.

5.1.3 `addIndividual()`

Arguments: OBJECT *individual*
Return value:
Defined in: GPPopulation.pm
Mainly called by:
Usually calls:
Relevant attributes:

Adds *individual* to `Individuals`.

5.1.4 `findNewDBFileStem()`

Arguments:
Return value:
Defined in: GPPopulation.pm
Mainly called by: perlgp-run.pl
Usually calls:
Relevant attributes:

Searches the scratch directory (PERLGP_SCRATCH) for an unused filename of the format "Individual-%06d" where the last bit is a serial number. For speed, instead of checking for the existence of files, a hash table in memory (`%usedfilestems`) is used.

5.1.5 `repopulate()`

Arguments: `RandomFraction =>` NUMBER (optional)
Return value:

Defined in: GPPopulation.pm
Mainly called by: perlgp-run.pl
Usually calls: `addIndividual()`
Relevant attributes: `PopulationDir`

This is called on a newly created Population object in perlgp-run.pl in order to load up any pre-existing Individuals. First, it looks in `PopulationDir` for DBM files. If there are no files and if a gzipped tar file backup exists (see `backup()` for this Population, then the tar file is unpacked into `PopulationDir`. In other words, the Population directory in PERLGP_SCRATCH is recovered from the backup.

Then all the unique filestems in `PopulationDir` are used to create new Individual objects which are passed to `addIndividual()`. In other words, new Individuals are made, each one tied to a DBM file in `PopulationDir`.

5.1.6 `backup()`

Arguments:
Return value:
Defined in: GPPopulation.pm
Mainly called by: `TournamentGP::run()`
Usually calls:
Relevant attributes: `ExperimentId`

This routine simply tars up the entire `PopulationDir` and saves it in the directory PERLGP_POPS with a name based on `ExperimentId`. To recover a population, see `repopulate()`.

5.1.7 `emigrate()`

Arguments:
Return value:
Defined in: GPPopulation.pm
Mainly called by: `TournamentGP::run()`
Usually calls: `selectCohort()`, `Individual::save()`
Relevant attributes: `MigrationSize`

This method randomly selects `MigrationSize` Individuals and saves them in a temporary directory, which is then tarred up into a file in PERLGP_POPS (the name comes from the method `export_tar_file()`).

5.1.8 `export_tar_file()`

Arguments:
Return value: STRING
Defined in: GPPopulation.pm
Mainly called by: `emigrate()`, `immigrate()`
Usually calls:
Relevant attributes: `ExperimentId`

All this does is return the following string:

```
sprintf "%s/%s.export.tar.gz", $ENV{PERLGP_POPS}, $self->ExperimentId();
```

(Sometimes Perl is easier to understand than English.)

5.1.9 `immigrate()`

Arguments:
Return value:
Defined in: GPPopulation.pm
Mainly called by: `TournamentGP::run()`
Usually calls: `randomIndividual()`, `Individual::load()`
Relevant attributes:

First, some explanation of a major assumption we make about migrating populations: that is that mutually migrating populations are assumed to have the same base `ExperimentId` followed by a minus sign and an integer, i.e. `pi-01 pi-02 pi-03 pi-04`. These directories are made automatically by the wrapper script perlgp-mrun.pl.

Potential immigrant population samples are gathered by a glob in PERLGP_POPS for `ExperimentId-*.export.tar.gz`. Then one of these is selected at random (but not a sample exported from this Population, of course), and is unpacked into a temporary directory. This temporary directory contains DBM files, and these are loaded up into randomly selected Individuals from this Population using `Individual::load()`. Each of the affected Individuals now has a new genome so its fitness is reset with a call to `initFitness()` (because the previous fitness may have been calculated on different training examples in the other Population).

5.1.10 `initFitnesses()`

Arguments:
Return value:
Defined in: GPPopulation.pm
Mainly called by:
Usually calls:
Relevant attributes:

A routine to call `initFitness()` on each Individual in the Population.

5.1.11 `countIndividuals()`

Arguments:
Return value: NUMBER
Defined in: BasePopulation.pm
Mainly called by:
Usually calls:
Relevant attributes:

Returns the number of Individuals in the Population, not to be confused with `PopulationSize`.

5.1.12 `randomIndividual()`

Arguments:
Return value: OBJECT
Defined in: BasePopulation.pm
Mainly called by: `selectCohort()`
Usually calls:
Relevant attributes:

Returns one randomly selected Individual from `Individuals`.

5.1.13 `selectCohort()`

Arguments: NUMBER *size*
Return value: ARRAY
Defined in: BasePopulation.pm
Mainly called by: `TournamentGP::tournament()`, `emigrate()`
Usually calls: `randomIndividual()`
Relevant attributes:

Returns *size* different, randomly selected Individuals from the Population.

5.2 Attributes & Variables

5.2.1 `Individuals`

Data type: ARRAY-REF
Default: empty
Defined in: BasePopulation.pm
Mainly used in: `addIndividual()`, `countIndividuals()`, `randomIndividual()`
See also: `PopulationSize`

The array where the Individuals in the Population are stored.

5.2.2 `PopulationSize`

Data type: NUMBER
Default: 2000
Defined in: BasePopulation.pm
Mainly used in: perlgp-run.pl
See also:

This is the maxium allowed size of the Population, if you use perlgp-run.pl to start a run. The Population class does not control the number of Individuals itself.

5.2.3 `MigrationSize`

Data type: NUMBER
Default: 50
Defined in: GPPopulation.pm
Mainly used in: `emigrate()`
See also:

If migration is enabled (in the Algorithm class with `EmigrateInterval` and `ImmigrateInterval`), then this parameter determines how many Individuals move from one population to another at a time.

5.2.4 `PopulationDir`

Data type: STRING
Default: see below
Defined in: GPPopulation.pm
Mainly used in: `repopulate()`
See also: `ExperimentId`

The directory where all the per-Individual DBM files are stored. The default is PERLGP_SCRATCH/`ExperimentId`.

5.2.5 `ExperimentId`

Data type: STRING
Default:
Defined in: GPPopulation.pm
Mainly used in:
See also:

The same as the attribute in Algorithm with the same name.

6 Universal Base Class: PerlGPObject

6.1 Methods

6.1.1 `new()`

Arguments: none
Return value: void
Defined in: PerlGPObject.pm
Mainly called by:
Usually calls:
Relevant attributes:

This is actually where the constructor for all PerlGP classes is defined, but it has been explained in more detail above.

6.1.2 `AUTOLOAD()` (get and set attributes)

Arguments: Attribute name or value
Return value: Attribute value or `$self`
Defined in: PerlGPObject.pm
Mainly called by:
Usually calls:
Relevant attributes: all

Thanks to the magic of Perl's AUTOLOAD mechanism, we can use all valid attribute names as methods to get and set their values. So the usage is simply:

```
# getting
my $prob = $individual->NodeMutationProb();
my $size = $population->PopulationSize();

# setting
$algorithm->TournamentsSinceBest(0);
$individual->Terminals({ NUM=>[0 .. 9], CHAR=>['a' .. 'z'] });
```

However, sometimes it's not convenient to use this notation, particularly during string interpolation or with +=, so you will often see direct access to the attributes in the object hash.

```
$algorithm->{Tournament}++;
print "my file stem is $self->{DBFileStem}\n";
```

In order for this to mechanism to work, an object needs to know which attributes it has. It looks to see if the requested method name is a key in the object's hash table. The simplest way to define these is in the `%defaults` hash in the top level class definition file (e.g. Algorithm.pm). You can also "reserve" an attribute for later use with `optionalParams()`, which is equivalent to initialising the attribute with an undefined value.

Because the set routine returns `$self`, you can chain "set" calls like this:

```
$object->NodeMutationProb(0.01)->NodeXoverProb(0.02);
```

6.1.3 optionalParams()

Arguments: ARRAY *attribute_names*
Return value:
Defined in: PerlGPObject.pm
Mainly called by: `_init()` methods
Usually calls:
Relevant attributes:

See the discussion for `AUTOLOAD`. Add a call to this method in the `_init()` routine (after the `SUPER::_init()` call - see GPPopulation.pm for an example) to reserve an attribute name so that the AUTOLOAD get/set routines will recognise it.

6.1.4 compulsoryParams()

Arguments: ARRAY *attribute_names*
Return value:
Defined in: PerlGPObject.pm
Mainly called by: `_init()` methods
Usually calls:
Relevant attributes:

If you want to make sure that the user provides values for certain attributes in the constructor or top level `_init()` routine, then you can use this routine. See `GeneticProgram::_init()` for an example.

6.2 Attributes & Variables

6.2.1 `Class`

Data type: STRING
Default:
Defined in: PerlGPObject.pm
Mainly used in: nowhere, actually
See also:

All objects store their own type in the `Class` attribute, but this isn't used anywhere. Could be useful for debugging, but then you could just print out the object reference.

7 Grammar definition

You control the space explored by your evolving programs by specifying a grammar for their construction. This is done in Grammar.pm in a fairly standard way, except that the production rules are split into functions and terminals. The following sections explain some of the concepts and how the grammar is converted into a real Perl program. But first I explain some strange Perl syntax that you may not have seen before:

```
$F{ROOT} = [ <<'___',
package Individual;

sub evaluateOutput {
  my ($self, $data) = @_;
  my ($x, $y, $z, @output);
  foreach $input (@$data) {
    $x = $input->{x};
    # begin evolved bit

    $y = {NUM};

    # end evolved bit
    push @output, { 'y'=> $y };
  }
  return \@output;
}
___
];
```

The right hand side of this assignment is an anonymous array (or reference to an array) containing a single string element. The contents of that string are everything on the lines *between* the `<<'___'` and the following `___`. This is known as *here-text*, and is often used in shell scripts. In Perl there are different flavours of here-text, the one used here is single-quoted here-text, which means things that look like variables (like `$x`) are not interpolated.

Why not just put single quotes round this string? Well there are some single quotes in the string and we like those and don't want to mess around with

backslashes. OK, so why not use q() or q//? Then we have to make sure that the delimiter is not in the string anywhere, which is a pain. Here-text does just fine thanks, and tends not to mess up emacs colouring too!

Note that the following, with the square brackets closed before the here-text, is also correct:

```
$F{FOO} = [ <<'___' ];
foreach $foo (@foo) {
  $foo->{FUNC}();
}
___
```

Now read on for a short explanation of what's going on with hashes and grammar definition (modified from my EuroGP2003 paper/poster).

7.1 Tree-as-Hash-Table Genotype Representation

Hash tables, also known as associative arrays, can be hijacked to encode string-based tree structures as explained in the code snippet below. The keys in the genome hash-tree follow the syntax: `nodeTYPExx`, where `TYPE` is replaced by an all-capitals string describing the type of the node (see Section 7.2), and `xx` is a unique identifier (for there may be many nodes of the same type).

```
$tree{nodeS0} = 'One day in {nodeS1}.';
$tree{nodeS1} = '{nodeS2} {nodeS3}';
$tree{nodeS2} = 'late';
$tree{nodeS3} = 'August';
$string = $tree{nodeS0};
do { print "$string\n" } while ($string =~ s/{(\w+)}/$tree{$1}/);

# outputs the following:
  One day in {nodeS1}.
  One day in {nodeS2} {nodeS3}.
  One day in late {nodeS3}.
  One day in late August.
```

Tree-as-hash-table explanation. In Perl, the syntax `$one{two} = 'three'` means that in a hash table named 'one', the value 'three' is stored for the key 'two'. The iterated search-and-replace (`s/patt/repl/`) looks for hash keys contained within curly braces and replaces them with the contents of the hash.

7.2 Grammar Specification

PerlGP is a strongly typed system. In fact, all evolved code must be syntactically correct to be awarded fitness. When random individuals or subtrees are generated, PerlGP follows a grammar (defined by the user). The format of this grammar is analogous to the tree-as-hash encoding described above, and is explained in the code below:

```
$F{ROOT} = [ '{STATEMENT}' ];
$T{ROOT} = [ '# nothing' ];
```

```
$F{STATEMENT} = [ 'print "{STRING}!\n";',
                  '$s = "{WORD}";',
                  '{STATEMENT}
{STATEMENT}' ];

$T{STATEMENT} = [ '# just a comment',
                  'chomp($s);',];

$F{STRING} = [ '{STRING}, {STRING}',
               '{WORD}' ];

$T{STRING} =
  $T{WORD} = [ 'donuts', 'mmm', '$s' ];
```

Grammar specification as a pair of hashes, `%F` for functions and `%T` for terminals. The keys in the hashes are the user-defined node types (i.e. data types). Node types must be in capital letters only. The values are anonymous arrays containing the possible expansions for that type. When another function or terminal is needed, it is signalled by a node type in curly braces. The `ROOT` node type must always be defined. Function definitions are optional (in this example there is no function of type `WORD`) but terminals must be defined for every type.

7.3 Random Initialisation of Programs

A random tree is generated simply by starting with a new node of type `ROOT`, picking a random element from the array stored in `$F{ROOT}`, creating new nodes wherever `{TYPE}` is seen. This is illustrated below:

```
 1  $genome{nodeROOT0}       = '{nodeSTATEMENT0}';

 2  $genome{nodeSTATEMENT0} = '{nodeSTATEMENT1} {nodeSTATEMENT2}';

 3  $genome{nodeSTATEMENT1} = '$s = "{nodeWORD0}";';

 4  $genome{nodeSTATEMENT2} = 'print "{nodeSTRING0}!\n";';

 5  $genome{nodeSTRING0}    = '{nodeSTRING1}, {nodeSTRING2}';

 6  $genome{nodeSTRING1}    = '{nodeWORD1}';

 7  $genome{nodeSTRING2}    = '{nodeWORD2}';

 8  $genome{nodeWORD0}      = 'donuts';

 9  $genome{nodeWORD1}      = 'mmm';

10  $genome{nodeWORD2}      = '$s';
```

To make a new tree: start with a `ROOT` node, assign a new genome key `nodeROOT0` and pick one of the available `ROOT` type functions from the grammar (see

Section 7.2). In this case there is only one choice (line 1). The contents of the new node require a new STATEMENT type node to be created, and a random function of that type is chosen (line 2). Now there are two child nodes to be expanded (lines 3 and 4). The process continues recursively along all branches and when a function can not be found, a terminal node is used instead.

Here are a few examples of random code generated from the actual grammar defined in Section 7.2, generated with the utility program perlgp-rand-prog.pl, each separated by a blank line.

```
print "$s, donuts, mmm, mmm, $s!\n";
print "$s, mmm, $s!\n";
print "$s!\n";

$s = "$s";

print "mmm, mmm!\n";
$s = "$s";
print "mmm, $s, $s, mmm, mmm, $s!\n";
print "donuts!\n";
$s = "mmm";
print "mmm, donuts, mmm, mmm, mmm, $s, mmm, donuts, $s, mmm, $s, $s, donuts,
 donuts, donuts, donuts, mmm, $s, donuts, donuts, donuts, donuts, mmm!\n";
print "$s, $s!\n";

$s = "donuts";
$s = "donuts";
$s = "mmm";
$s = "$s";
print "donuts!\n";

$s = "donuts";
```

Tree termination and size control can be achieved in three ways. I prefer to construct the Grammar with biased frequencies of branching and non-branching functions so that trees terminate naturally.

Whereas the following grammar definition tends to produce very deep trees:
`$F{STRING} = ['{STRING}, {STRING}','{WORD}'];`,
this modification produces more reasonably sized trees:
`$F{STRING} = ['{STRING}, {STRING}','{WORD}','{WORD}','{WORD}'];`
because the WORD type is non-branching and only terminals are defined for it.

Alternatively or additionally, maximum and minimum tree sizes (number of nodes) can be imposed, along with an early termination probability and a maximum tree depth limit.

8 Utility Scripts

8.1 perlgp-run.pl

This is the main program for running a PerlGP experiment. It expects to be run from the "experiment directory". The only option available is -loop which

causes the program to restart after crashing (with a 60 second sleep). See also the Algorithm attribute `ForkForEval` if you are having problems with crashing.

8.2 plot-tlog.pl

This is the script for plotting the tournament log file (explained in Table 1 in Section 3.1.9). It uses gnuplot, so you will need to install this if you don't have it and want to use this (simple to adapt) script.

```
usage: plot-tlog.pl
```

The main options are:

-refresh S - sleep for S seconds and then replot

-geometry GEOM - X11 geometry specification for window (e.g. -geometry 400x300-0+0)

-timebased - show the number of hours elapsed since the start of the run on the x-axis

-logs AXIS - use a logarithmic scale on AXIS (x, y, x2, y2)

-yrange 'RANGE' - fix the y-axis range, (-yrange '[0:100]')

-xrange, -y2range - see -yrange, the program size and complexity values are plotted against the y2 axis

8.3 perlgp-wipe-expt.pl

When run from inside an experiment directory, it removes the results directory and also any on-disk populations belonging to this run in PERLGP_SCRATCH and PERLGP_POPS.

When run from outside an experiment directory, you can give multiple experiment directory paths on the command line and it will go into each and clean them as above.

8.4 perlgp-rand-prog.pl

Makes and prints to STDOUT a random program from your grammar definition. (For the interested: it does this by generating a new Individual object with a temporary `DBFileStem`.)

Extremely useful when defining a new grammar. It is recommended to set `MinTreeNodes` to zero while experimenting with new grammars (otherwise you don't get a proper sense of how the trees are naturally terminating).

8.5 perlgp-sample-pop.pl

```
usage: perlgp-sample-pop.pl [ 0.1 ] | less
(runs inside experiment directory)
```

Samples a random fraction (default 0.1, but you can give a different value on the command line) of the Population and prints the Perl code to STDOUT. It can be useful to pipe the output through grep and sort to give you an idea of what are the common "genes" (lines of code) in your gene pool.

8.6 perlgp-show-prog.pl

```
usage: perlgp-show-prog.pl db_file_stem

example: perlgp-show-prog.pl results/keptbest/Tournament-0000139
```

Loads up an Individual from a DBM file, which is not human readable, and expands the Perl code and prints it to STDOUT.

8.7 perlgp-mrun.pl

A wrapper script for running multiple copies of experiments (in copied directories). You should make sure that there are no large files in the experiment directories (use links for large data files) if you cannot afford the disk space.

On a single processor, this script will run jobs in serial. If you have a cluster, or a multi-processor machine, you can use the -queue option but *you must modify the script to work with your local queueing system.*

You run the script from the parent directory of the experiment directory:

```
usage: perlgp-mrun.pl -num 20 -hours 4 expt_dir_name
```

The main options are:

-number X - makes X copies of the experiment and runs them

-hours H - runs the jobs for H hours

-mins M - or use this if you count in minutes

-queue - use a queueing system (see above)

-loop - pass the -loop option to perlgp-run.pl

Other options should be self-explanatory in the script itself.

8.8 perlgp-avg-logs.pl

When you've run 50 copies of the same job using perlgp-mrun.pl, this script will take the last line of each 'results/tournament.log' file and calculate the means and standard deviations from each numeric column. For non-numeric columns it prints the most common string seen in that column.

```
usage: perlgp-avg-logs.pl label1 'glob1-??' label2 'glob2-??' label3...
```

Where each label is an arbitrary identifier for the experiments which the following shell glob (protected in quotes) expands to. Perhaps this is best explained with an example. Imagine you are running two symbolic regression experiments, one with trigonometric functions, one without. Assuming you ran perlgp-mrun.pl on two experiments named 'fit-withtrig' and 'fit-notrig', then you would use this program as follows:

```
example: perlgp-avg-logs.pl notrig 'fit-notrig-??' trig 'fit-withtrig-??'
```

When exactly two sets of experiments are given, as above, this program also prints out the d value from a paired t-test (asks if the means from two assumed normal distributions are significantly different). Look this up in any statistics text book, but as a rough guide, if you have done more than 30 replicates of each experiment and the absolute value of d is greater than 1.96, then the two means are significantly different at the 5% level.

If you want to do the averaging at a particular tournament number, and not on the final tournament, add the option -tournament T, where T is the tournament number. If this tournament is not in the logfile, the last entry in the log is used.

You can specify logarithm-taking of certain columns in the log file with -logs COL1 -logs COL2 or -logs 1,5 and set the log base with -base N. Column numbers start at zero.

9 Demos

9.1 Approximation of pi

Please change directory to 'PERLGP_BASE/demos/pi' and look at the README file before running this demo.

9.1.1 Problem definition

Find a integer arithmetic approximation to π, for example 3 + 1/7

Arithetic operations allowed: plus, minus, multiply and protected division

Integers allowed: 1,2,3,5,7

Fitness function: absolute error from π defined as `2*atan2(2,0)` (small is good)

9.1.2 PerlGP approach

In this case the training and testing data structures (`TrainingData`, `TestingData`), and the output from `evaluateOutput()` are simple scalar numbers. Usually, as discussed in Section 2.4, they would be scalar references to larger data structures. There is no input variable in this situation (unlike the more familiar regression problem). There is also no meaningful concept of "testing"; because no data is sampled, there can also be no out-of-sample data. Hence `loadSet()` returns just the "true" value of pi for both training and testing instances. This value is only accessed by `fitnessFunction()` of course, not by the evolved code.

There's not much more to explain, except of course that when the fitness reaches around 1e-15 it can go no further because the limit of double precision floating point numbers has been reached.

In this example, self-adapting mutation and crossover probabilities (`NodeMutationProb` and `NodeXoverProb`) are used. You can follow the evolution of these attributes with the gnuplot script plot-evparams.gp.

9.2 Symbolic regression of sine function

Please change directory to 'PERLGP_BASE/demos/sin' and look at the README file before running this demo.

9.2.1 Problem definition

Find an approximation to $\sin 3x$ using arithmetic operations plus, minus, multiply and protected division, and the integers 1,2,3,5,7 and the input x.

The fitness measure is basically the root mean squared error or deviation of the predicted function from the target function, measured on a set of randomly sampled points, between -1.0 and +1.0 initially, but the range and number of points increases during the run (see below).

9.2.2 PerlGP approach

Here, `loadSet()` reads in data (randomly sampled x values from the files `TrainingSet` or `TestingSet`) which is generated by a simple script ./generate_data.pl (which gets run by the `refresh()` method, but more on this later). The general flow of data is discussed in detail in Section 2.4.

Two subtle "hacks" seemed to be needed to get well behaved GP runs on this problem. Domain specific knowledge was used in both cases. Firstly, the error function incorporates a cap on the per-sample error. If the squared difference between the predicted and known y values is greater than 1.0, it is counted as 1.0. This may not be essential, but with prior knowledge of the Taylor expansions of trigonometric functions we know that at $\pm$ infinity the approximation also deviates infinitely from the x axis. So this error capping should help not to punish functions containing high powers of x. The second hack is to start training on a small range of x values centred around zero and then increase the range (attribute `Range`) as the evolved solutions reach a certain target fitness. Again we are using domain specific knowledge: we know that the early terms in the Taylor expansion best fit the sine function for values of x around zero. You can play around with more complex target functions (for example, try $\sin(3(x - 1.23)))$ where there is no trivial solution early on.

Take a look at the `refresh()` method in Algorithm.pm, this is called on the first tournament and subsequently every `RefreshInterval` tournaments. When the best training fitness goes below a threshold, the `Range` is increased and the data is regenerated, and loaded up again (and the whole Population's fitnesses are reset). An "anti-stagnation" measure is also implemented in `refresh()`: if more than 1000 tournaments go by without an improvement in fitness, the data is resampled and reloaded (but `Range` does not increase).

This demo also uses self-adapting `NodeXoverProb` and `NodeMutationProb` attributes.

The various devices employed here may or may not be necessary to get a good result, but they illustrate some of the ways you can customise PerlGP.

Note: the fitness function and data structures are not really optimised for speed (wasteful use of hashes and so on). You can read about speed comparisons against lilgp on the same problem in my EuroGP 2003 paper (available on request and later on the web). It's always a good idea to have something else to do while jobs are running however...

9.3 Compound interest

Please change directory to 'PERLGP_BASE/demos/interest' and look at the README file before running this demo.

9.3.1 Problem definition

Given the initial amount deposited in a bank, a fixed yearly deposit amount, the interest rate and the number of years invested, what is the final amount assuming a simple model of compound interest.

The fitness function is the root mean square deviation of the predicted final amount from the real final amount.

9.3.2 PerlGP approach

This is treated more like a modelling problem than the previous regression problem. The training data is generated by ./generate_data.pl and is read in with `loadSet` in the usual way. The grammar is constructed to allow looping and allows six variables to be manipulated: the four input variables and an additional two which are initialised to zero. Note how excessive looping is avoided with the use of the `$z` variable.

This demo has a nice simple `refresh()` method that you can look at. It also uses self-adapting `NodeXoverProb` and `NodeMutationProb` attributes.

This appears to be a challenging problem, because it is difficult to get good results without unfairly biasing the grammar to give the kind of programs you want to see. For example it is tempting to add a `{VAR} += {NUM}` statement. This is a common problem in genetic programming - the search space is large, local minima are prevalent, and CPU time and memory is limited, so it is tempting to push the system in the direction you think it should take. Another temptation is to use huge populations, but that changes the problem to a brute force search and evolution becomes less important.