Introducing a Perl Genetic Programming System – and Can Meta-evolution Solve the Bloat Problem?

Robert M. MacCallum

Stockholm Bioinformatics Center
Stockholm University
106 91 Stockholm
Sweden
maccallr@sbc.su.se

Abstract. An open source Perl package for genetic programming, called PerlGP, is presented. The supplied algorithm is strongly typed tree-based GP with homologous crossover. User-defined grammars allow any valid Perl to be evolved, including object oriented code and parameters of the PerlGP system itself. Time trials indicate that PerlGP is around 10 times slower than a C based system on a numerical problem, but this is compensated by the speed and ease of implementing new problems, particularly string-based ones. The effect of per-node, fixed and self-adapting crossover and mutation rates on code growth and fitness is studied. On a pi estimation problem, self-adapting rates give both optimal and compact solutions. The source code and manual can be found at `http://perlgp.org`

1 Introduction

Many packages for genetic programming (GP) are now freely available on the Internet. For example, the C package lilgp[10], the ECJ[5] Java system, and the Open BEAGLE framework[2] in C++. Many other languages are also represented but Perl is conspicuously absent, except for two proof-of-concept implementations[6, 9] which are not intended for general use. Perl is seen by many as just a quick-and-dirty tool for hacking together web interfaces or backup scripts. It has its origins in these areas but is now a mature language, with modularisation and object orientation. Speed of execution is not Perl's strong point since it is (usually) interpreted, however when intensive numerical computation is needed, C-coded extensions often exist to take care of it (such as the PDL extension [http://pdl.perl.org]). Perl allows fast project development and prototyping and it has a number of built-in features which make it easy to implement tree-based GP. These include hash tables, powerful string manipulation and run-time evaluation. Here I introduce perhaps the first major open source GP system written in Perl, called PerlGP. This paper intends to give a full introduction to the system, explain some of the design decisions and show examples of use as part of some brief analyses of execution speed, bloat and meta-evolution.

2 Implementation

The following sections describe how the PerlGP system is put together. Many features of the system are inspired from Nature or the literature. Unfortunately space limitations prevent proper attribution for all of them here.

2.1 Object-Oriented Design

There are three main object types: `Individual`, `Population` and `Algorithm`. The user provides the implementation for a few key methods (for example the fitness function, data input/output) in these classes and specifies the base classes from which they should inherit. The base classes, such as `TournamentGP` (`Algorithm`) and `GeneticProgram` (`Individual`) take care of the rest. Other search strategies and representations can be added to the package, and it should be trivial to swap them in and out as required using inheritance.

The `Individual` is the most important object, it *is* the genetic program. Each instance has a tree-represented genome and can convert it into Perl code for evaluation. The object also knows how to perform mutations, crossovers and saving to disk. `Population` is a container class for individuals, and does little except provide a method for picking random individuals, and methods for migration between populations, via disk. The `Algorithm` class is concerned with manipulating a population; selecting individuals, feeding them input data, collecting the output, and calculating fitnesses.

2.2 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 Figure 1. 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 2.3), and `xx` is a unique identifier (for there may be many nodes of the same type).

2.3 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 Figure 2.

2.4 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 in Figure 3.

```
$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.
```

Fig. 1. 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.

Tree termination and size control can be achieved in three ways. The author prefers 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.

2.5 Persistence of GP Individuals

The standard library for Perl contains many useful things, including the "DBM" modules. These provide a simple interface for storing key-value pairs on disk with fast indexed access. Their use is extremely simple: using `tie()`, a normal Perl hash-table is linked to a file, and every change made to the hash is also made to the file. Using `tie()`, the genome hash of every individual is transparently mirrored on disk. This is useful if the user wants to use populations that are too large to fit into RAM. It also provides continuous checkpointing, allows the population to be sampled/examined by another program during the run.

2.6 Code/Fitness Evaluation

Every object of type `Individual` has to implement the method `evaluateOutput()`. This method takes as input the training data and produces some kind of output

```
$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' ];
```

Fig. 2. 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.

```
 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';
```

Fig. 3. 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 Figure 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.

data structure that the fitness function (in `Algorithm`) can understand. The user can either provide an evolved `evaluateOutput()` method (the definition for this method is usually in the `ROOT` node of the tree), or some function or method which is called from a non-evolved `evaluateOutput()`.

When the fitness of an individual is required (and is not cached in memory), the genome is expanded into code which is evaluated with Perl's `eval()` function. This redefines `evaluateOutput()` and/or other methods - overwriting any previously defined methods (from the last individual's fitness evaluation, for example). Normally Perl would emit warnings about this, but these are suppressed. Then, `evaluateOutput()` is called and the fitness is calculated from the return value. The fitness is stored both in memory and on disk (in the genome hash), to avoid unnecessary recalculation and allow faster restarts.

3 Genetic Algorithm Design

3.1 The Genetic Algorithm

The first release of this software provides an `Algorithm` superclass implementing tournament-based GP, which is a good starting point for developing biologically realistic algorithms (at least with respect to higher organisms). Each tournament involves the random selection of a group of individuals which are sorted by decreasing fitness. Individuals which have participated in more than a certain number of tournaments are automatically given the worst possible fitness, to simulate the natural ageing process. Then, the first parent is taken from the top of the list, and a mate is selected either from the next in line, or a random selection biased towards the top of the list (the user decides which strategy). Crossover (see Section 3.2) is performed on these two parents to create two offspring which replace individuals at the bottom of the sorted list. Each offspring is subjected to mutation (see Section 3.3) before being placed back in the population. Mutation is only applied after reproduction because in biology, the only relevant mutations are those that occur in the germ-line. If desired, more pairs of parents are chosen from the same sorted list and are crossed over as before. As a somewhat crude anti-stagnation measure, two parents may only produce offspring if their fitnesses are not identical, otherwise the second parent is mutated.

3.2 "Homologous" Crossover

When trying to draw inspiration from biology, the crossover mechanism should perhaps deserve the most attention. In PerlGP, by default, the number of crossovers per reproduction event is variable and depends on the number of nodes in the tree (this is called a uniform, or per-node crossover rate). When no crossover is performed, parents are simply copied into the offspring. This design decision is discussed in more detail in Section 4.2.

Another biology-based decision was to attempt "homologous" crossover: where crossover points are biased to give subtrees of similar size and contents.

This is achieved, crudely, by randomly sampling two subtrees, A and B, from each parent respectively, until a pair is accepted as a crossover point with the probability:

$$\left(1 - \left(\frac{|N_A - N_B|}{\max(N_A, N_B)}\right)^s\right) \cdot \left(\frac{I_{A,B}}{\min(N_A, N_B)}\right)^h \quad ,$$

where N is the number of nodes in a subtree, and I is simply the number of identical nodes seen during the parallel descent of two subtrees (stopping at non-identical nodes and not allowing for insertions or deletions). This contrasts with Langdon's approach[4] which looks towards the root of the tree for similar contexts. The exponents s and h (which default to 1) can be changed to give more or less emphasis on size and "homology" respectively.

Crossover is only allowed between nodes of the same type and the subtree sampling may be biased to give larger subtrees than random sampling would normally give. When multiple crossover points are required, subsequent points are not allowed to lie within in the subtrees of previous crossover points.

3.3 Mutation Operators

As with crossover, the default behaviour of PerlGP is to apply a random number of mutations proportional to the number of nodes in the tree. The two main types of mutation are *point mutation* and *macromutation*, and the choice between them is random (with a user-defined bias). Point mutations involve picking a random node (internal or terminal) and picking a new function (of the same arity) or terminal from the grammar. In some cases, the replacement function or terminal will be identical to the original, so there is an option to repeat the process until some change is made (switched off by default). Numeric terminal nodes can be treated specially so that point mutations make a random adjustment to the number instead of replacing it - this is called *numeric mutation*.

Macromutation is a little more complex. Nodes are (optionally) biased towards internal nodes, and the following operations are chosen from at random:

Replace subtree Subtree replaced with a random subtree.

Copy subtree Two independent nodes (subtrees not containing each other) are selected and one subtree is copied, replacing the other.

Swap subtrees As above, but subtrees are swapped.

Insert internal A node is chosen, (e.g. a terminal node: `2`) and is replaced with a non-terminal node (e.g. `{NUM} + {NUM}`). One of branches is linked back to the original node, and any remaining branches are expanded with new subtrees (e.g. result: `1 + 2`).

Delete internal Two nodes are chosen, the second belongs to the subtree of the first and is of the same type. The nodes are reconnected, and any intervening nodes are removed from the tree.

4 Benchmarking

4.1 Speed Comparison with lilgp

A popular open source GP package is lilgp[10] which, being quite minimal and written in C, is presumably one of of the fastest GP systems available (excluding machine-code systems of course). To get a feeling for how much slower a Perl-based system executes compared to a system in a "proper" language, both PerlGP and lilgp were challenged with symbolic regression of a sine curve. The aim was to use identical training data, function sets, and fitness functions, but allow each each system to use its "default" algorithm to reach a certain fitness in a fixed time. The main reason behind this decision was to avoid a major of overhaul of PerlGP to make carbon-copy lilgp emulation possible, when we know the Perl system is going to be slower anyway. The experimental setup is outlined in Table 1, and a summary of the results is given in Table 2.

	lilgp	**PerlGP**
target function	$y = \sin(3x)$ for $-1 < x < 1$	
training data	100 points	
fitness function	$1/(1 + \sum \lvert y_{correct} - y_{gp} \rvert)$	
success criteria	fitness >= 0.4 (fit looks good) within 4 hours	
functions	$+ \quad - \quad *$ pdiv(a, b) (returns a if $b = 0$)	
terminals	ephemeral random constants $0 \rightarrow 1$	1000 random numbers $0 \rightarrow 1$
tree limits	init.method = half-and-half init.depth = 2-6 max_depth = 8 max_nodes = unlimited	naturally terminating trees max_depth = 20 (safety) max_nodes = 1000 (safety) rebuild trees < 20 nodes
population size	4000	2000
max. generations	5000	no limit
genetic algorithm	generational, fitness based selection, 90% crossover, 5% reproduction, 5% "keep trying" mutation	tournaments of 50 individuals, reproduce top 20, age limit: 4 tournaments

Table 1. Summary of settings for lilgp and PerlGP systems in the time trials.

The main conclusion is that lilgp manages to complete more runs within a fixed time period than PerlGP. When considering only the runs which reach the desired fitness, PerlGP takes about 3.5 times longer. It would be desirable to let all runs terminate naturally, but this was not feasible for a number of reasons; one being that an optimal solution cannot be guaranteed, another being that 17 if the lilgp runs behaved strangely and converged to a population of small unfit individuals. However, one can guestimate that PerlGP would be around 10 times slower than lilgp if all runs did terminate (this often quoted as a Perl-to-C speed differential).

GP system	total runs	successful runs in 4h	mean time (min)	mean tree size (nodes)
lilgp	100	70	24	120
PerlGP	100	55	82	83

Table 2. Speed comparison of PerlGP and lilgp. The mean time and tree sizes are calculated for successful runs only (achieving fitness of 0.4 in less than 4h).

PerlGP gives more compact solutions (more on this in Section 4.2). For the interested reader, one such solution is given here (slightly simplified from the evolved code):

$$sin(3x) \approx \frac{1.42291(0.57915 - x^2)}{0.38103/x + 0.32766x} + x \quad \text{for} \quad -1 < x < 1 \quad .$$

4.2 Mutation and Crossover Strategies

In biology, mutations result from uncorrected replication errors or by the action of mutagens such as radiation, chemicals, free radicals, viruses and transposable elements[1]. The number of mutation events occurring depends on the amount of DNA for most types of mutation. And while crossover points appear to be non-uniformly distributed in eukaryotic chromosomes, their numbers too are correlated with chromosome size[3]. Therefore PerlGP, which tries to be biologically inspired, applies mutations and crossovers with probabilities depending on the genome size (as default behaviour). Others have studied this approach and found that it (in contrast to the standard approach with fixed numbers of mutation and crossover) is an effective measure to protect against bloat[7, 8]. Bloat is the rapid increase in size of GP individuals without a corresponding increase in fitness. It is thought that larger GP individuals can "soak up" mutations and destructive crossovers and therefore are more likely to produce viable offspring. The counter argument is that uniform mutation and crossover are simply another form of parsimony pressure, penalising larger individuals. Here I ask the question, can a regime of uniform mutation and crossover allow code growth when the solution to the problem demands it?

The problem is a simple one: to find an integer arithmetic approximation of pi. The functions and terminals are `+ - * pdiv 1 2 3 5 7`, and the fitness function is the absolute error. Clearly, a more accurate solution will contain more terms, so the code tree will have to be bigger. Initially, two types of run were performed: one with per-node mutation/crossover rates of 1/102 and 1/34 respectively, and one with a single fixed mutation and crossover occurring with 3/10 and 9/10 probability (recall that mutation is applied to each offspring after crossover). The ratio of crossover to mutation in both cases is 3:1 and the rates were chosen to give similar amounts of actual mutation and crossover in the first few hundred tournaments of a run. Runs were terminated after 90 minutes and the final fitness *vs.* solution size from 50 runs of each type is shown in

Figure 4. It was not surprising to see that the fixed-mutation runs produced larger solutions, however the increased size was also accompanied by increased fitness. In this case, the size increase did not seem to be detrimental to learning (by slowing down evaluations or tree manipulation, for example). Indeed, many of the runs were well on their way to the best possible fitness (1e-15). In the other experiment, the high per-node mutation rate is slowing down learning by limiting code growth, but these runs would eventually find optimal solutions (data not shown).

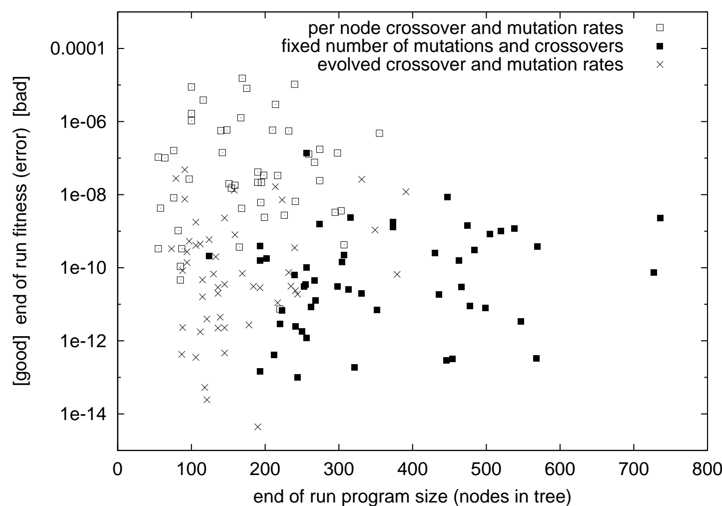

Fig. 4. End-of-run fitness on the pi approximation problem with different mutation and crossover regimes.

Out of curiosity, a third type of run was performed using self-adapting mutation and crossover probabilities. Meta-evolution is simple to implement in PerlGP - the user simply creates evolved code for the `evolvedInit()` method, which is called before fitness evaluation, mutation and crossover. In this function, the per-node mutation and crossover probabilities are redefined, and these values are allowed to change by numeric mutation only. The results on the pi problem (also shown in Figure 4) were surprising. The distribution of fitnesses in these runs was indistinguishable from the fixed-mutation/crossover runs (two-sample t-test on logged or unlogged errors gives $d < 1.96$), but the solution trees were significantly smaller ($d = 8.7$).

Meta-evolution of parameters has not yet been fully explored in the PerlGP system. The pi problem has a fitness landscape where improvements are always possible. In flatter or more complex fitness landscapes, self-adapting mutation and crossover rates may converge towards zero because increases in fitness are so rare that conservatism is the most rewarding strategy, in terms of survival.

5 Conclusion

PerlGP is a robust and flexible tool which has already been applied in my group to a variety of string and number based projects, including protein secondary structure prediction and time-series modelling of medical data. New projects can be started quickly without the need to provide code for the functions and terminals that Perl already has. Object oriented code can also be evolved, along with self-adapting parameters and genetic operators. Results suggest that self-adapting uniform mutation and crossover rates may be the answer to the bloat problem. Open source status will ensure that the project evolves in response to the demands of the community. The project's homepage is `http://perlgp.org`.

References

1. J. S. Bertram. The molecular biology of cancer. *Mol Aspects Med*, 21(6):167–223, Dec 2000.
2. Christian Gagné and Marc Parizeau. Open BEAGLE: A new C++ evolutionary computation framework. In W. B. Langdon and *et al*, editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, page 888, New York, 9-13 July 2002. Morgan Kaufmann Publishers.
3. J. L. Gerton, J. DeRisi, R. Shroff, M. Lichten, P. O. Brown, and T. D. Petes. Inaugural article: global mapping of meiotic recombination hotspots and coldspots in the yeast Saccharomyces cerevisiae. *Proc. Natl. Acad. Sci. USA*, 97(21):11383–11390, Oct 2000.
4. W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf, Jason Daida, Agoston E. Eiben, Max H. Garzon, Vasant Honavar, Mark Jakiela, and Robert E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1092–1097, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.
5. Sean Luke. A java-based evolutionary computation and genetic programming research system. Technical report, George Mason University, USA, Nov 2002.
6. Brad Murray and Ken Williams. Genetic algorithms with Perl. The Perl Journal (online), Issue 15 Vol. 5 No.3 http://www.samag.com/tpj, 1999.
7. J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.
8. Terry Van Belle and David H. Ackley. Uniform subtree mutation. In James A. Foster, Evelyne Lutton, Julian Miller, Conor Ryan, and Andrea G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 152–161, Kinsale, Ireland, 3-5 April 2002. Springer-Verlag.
9. Mark S. Withall, Chris J. Hinde, and Roger G. Stone. Evolving perl. In Erick Cantú-Paz, editor, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 474–481, New York, NY, July 2002. AAAI.
10. Douglas Zongker and Bill Punch. lilgp 1.01 user's manual. Technical report, Michigan State University, USA, 26 March 1996.