GENETIC IMPROVEMENT OF PROGRAMS

W. B. Langdon

University College, London
Department of Computer Science
Gower Street, London WC1E 6BT
UK
W.Langdon@cs.ucl.ac.uk [1]

Abstract: *Evolutionary computing, particularly genetic programming, can optimise software and software engineering, including evolving test benchmarks, search meta-heuristics, protocols, composing web services, improving hashing and garbage collection, redundant programming and even automatically fixing bugs. Often there are many potential ways to balance functionality with resource consumption. But a human programmer cannot try them all. Also the optimal trade off may be different on each hardware platform and it could vary over time or as usage changes. It may be genetic programming can automatically suggest different trade offs for each new market.*

Keywords: *Genetic programming, GISMOE, evolutionary computing, software engineering, refactoring, software maintenance, automatic software re-engineering, SBSE, GPGPU, CUDA, nVidia GPU, automatic programming, BNF grammar GP,*

1 Genetic Improvement of Programs

I shall start by describing when we might use genetic programming or other search techniques to automatically improve programs. Section 3 very briefly describes GP before Sections 4 and 5 talk about existing work on evolving software and improving human written software. Section 6 describes the GISMOE [12] approach whereby existing code is used as the *de facto* specification of the desired functionality neatly side stepping two major problems in current software engineering: 1) The difficulty of converting user desires into formal specifications. 2) The need for intensive manual input to check results produced by testing, particularly automatically generated tests. Thus the GSIMOE approach avoids the Oracle problem since for *any* test the existing code is the Oracle. I.e. the existing code defines the correct functionality. However the existing program may not run on new hardware, it might not be suitable for real-time use, be unable to meet new user load constraints, or fail other non-functional requirements. Section 7 summarises research [21] in which GP automatically generated replacement code for a vital part of the unix gzip file compression utility which was automatically written in the CUDA programming language and ran on an nVidia GeForce 295 GTX graphics card.

2 When to Automatically Improve Software

Although genetic programming has been highly successful (see following sections) most the programs evolved so far have been very small. Often human programmers would consider them too small to be called programs. Nonetheless even small amounts of code can be useful in software engineering. For example:

- in developing small amounts of "glue logic" to stick together large systems to give novel functionality, unavailable via any of the individual components. There is already some work on web "mashups" and using GP to create new web services by combining web services defined by the OWL protocol [30].
- Similarly we might consider using genetic programming to evolve solutions to problems that people find hard but which are as difficult to an insensitive machines as any other. For example small amounts of code to join diverse systems which are traditionally hard because it is impossible to find a programmer who is skilled in *all* the programming languages and tools used in the development and maintenance of all the components.
- Evolutionary algorithms, such as GP, have proved themselves many times to be well able to cope with evolving solutions not only to satisfy an objective but their population based search can accommodate multiple diverse solutions which make different tradeoffs between competing objectives [17, Chap. 3],[7]. Writing code and simultaneously managing the trade off between multiple objectives like, speed, minimising memory footprint and power consumption, is usually hard for people. But multiple objectives are not necessarily more difficult for a machine than dealing with one.

[1]Text to accompany keynote at MENDEL 2012

In future we might see automatic programming generating many many possible solutions but presenting only the Pareto optimal ones to a human programmer who chooses the one most suited for the current circumstances. Also any programme in use will constantly be faced with having to run in different circumstances, perhaps on different hardware platform and/or with very different user load. Currently because of the huge cost of software maintenance the usual response is to hope that the existing software will not perform too badly in its new circumstances. Automatic software generation or software tuning offers the possibility of tailoring the code to each new use case.

3 Genetic Programming

Following John Koza's first book [16] there have been at least 60 other books on GP and its applications. According to the GP Bibliography [18] more than 8000 scientific papers on GP have been published. Recently "A Field Guide to Genetic Programming" [28] was published on the web for free and hard copy can be purchased at a modest price. Some of the evolutionary computing conferences (e.g. GECCO) include both introductory and advanced tutorials on genetic programming. Also there are many web pages devoted to GP, including several tutorials such as John Koza's http://www.genetic-programming.com/gpanimatedtutorial.html.

There are far too many applications of genetic programming to include them all here. Look in the GP bibliography `http://www.cs.bham.ac.uk/~wbl/biblio/` for a more-or-less complete list. Also [28, Chapter 12] gives a taxonomy of GP applications.

4 Genetic Programming to Write Programs

This section describes some recent work on evolving programs using GP, whilst the next section will talk about GP being used to improve human written software.

Hash functions are an attractive target for GP (see Section 2). They are usually small but difficult to code from theory and even the best human written approaches are validated by trying them on what is hoped will be typical user data and measuring their performance. Even a good hash function may be defeated by unexpected patterns in the data it is applied to. There have been several attempts to evolve hash functions using GP [14; 3; 8]. The desired code is small and trial data to be hashed lends itself to being a fitness function. An automated method like GP can readily generate orders of magnitude more candidate hash functions for trial-and-error than a human programmer.

Heap management systems are typically slightly bigger since they often consist of teams of agents whose actions must be co-ordinated. (A DMM team typically consists of routines to: allocate new memory, free memory and perform garbage collection.) However again a generic efficient heap manager implementation has to make guesses about an individual applications use of dynamic memory. If these prove incorrect, the efficiency of the whole application can be affected. Thus, provided each DMM can be trained on representative patterns of memory use, they make a good candidate for bespoke automated software generation [29].

Genetic programming has also been used to create distributed algorithms [35], mobile applications [5] and composing new webservice mashups from other web services [30].

There is also interest in using genetic programming in multiplicity computing [4], where there may be advantages in generating many different versions of the same program, even if they are functionally identical.

5 Genetic Programming to Improve Human Written Programs

Some of Sipper's Finch work, like [25; 26; 13], evolves Java byte code from scratch. However Orlov and Sipper [27] also show GP can improve Java byte code produced by the Java compiler from existing human written programs. Another approach might be to "seed" the initial population with a non-random starting point [22].

Sitthi-amorn *et al.* [33] have shown it is feasible for GP to generate many improvements to GPU shaders. (Shaders are small graphics kernel functions which render three dimensional models into two dimensional coloured pictures to display to the user.) In [33] they showed GP generating a Pareto front populated by programs which make different tradeoffs between speed and accuracy. Rinard's group at MIT have also considered speed vs. error tradeoffs [32] but only by omitting non-critical parts of the calculation, whereas the Virginia University group's use of GP [33] allows other changes to the human written code.

5.1 Multiple Objective Software Evolution

White and Arcuri [36] have used GP to improve various aspects of existing programs, including removing faults [2], and speeding up code (e.g. 87% speed up of factorial). They have also looked at multi-objective optimisation, showing GP can give the software designer various Pareto optimal tradeoffs between entropy and power consumption in software based pseudo random number generators (PRNGs).

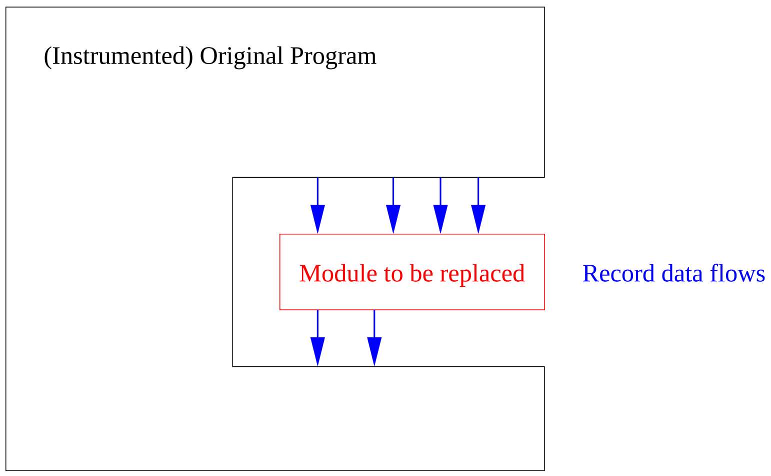

Figure 1: The original human written code is instrumented so that data (blue arrows) into and results returned by module to be replaced (red) are recorded as the original program is run on test cases representative of the new usage. GP will evolve new code which is tested to ensure it responds like the original program did. Effectively the data flow acts as a test Oracle.

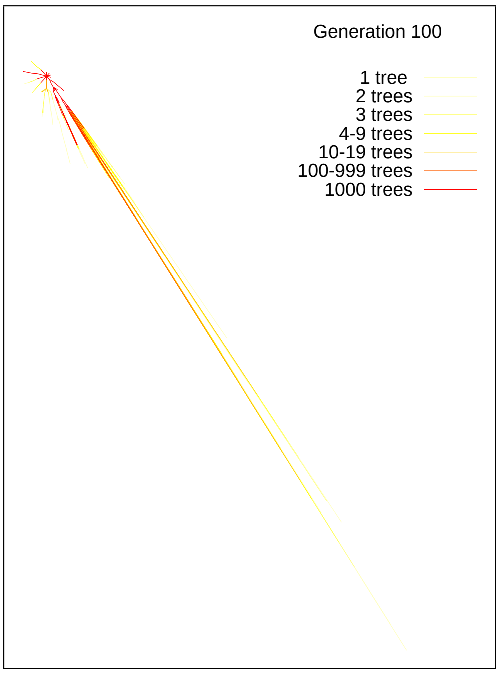

Figure 2: Aligning all 1000 trees in the GP population at their roots (red) stresses much of trees near their roots are identical. Unique parts (light) are only far from root. http://www.cs.ucl.ac.uk/staff/W.Langdon/lisp2dot.html gives some code to display trees as circular lattices [6].

5.2 Genetic Programming and Automatic Bug Fixing

The work of Wes Weimer's group on using GP to automate bug fixing [24] is increasingly well known, through a series of best paper awards [34; 9] and winning the gold prize at the 2009 GECCO Awards for Human-Competitive Results (Humies). Although mostly looking at repairing C programs at the sources level, the Virgina University group have also considered repairing programs at lower levels [31]. GP has also been used to repair Python programs [1].

6 GISMOE: Genetic Programming and Automatic Coding

Figure 1 shows the use of existing code in the gismoe framework. The idea is that the human written code acts as a scaffold to support the evolution of replacement components. The existing code defines via a fitness function the goals that GP should meet when generating new code. Once evolved code has been created by fitness testing it can be 1) tested on all test cases 2) manually inspected 3) run in parallel with existing code using a test harness which reports differences (if any) between the human written code and the GP written code. Once sufficient confidence has been established, the evolved code can be permanently enabled. Alternatively GP can be used as a tool to cheaply demonstrate the existence of interesting new Pareto optimal trade off points, which can then be implemented in the traditional (expensive) way.

7 gzip

The remainder of the paper summarises experiments performed on the unix gzip file compression utility [21]. The idea is to take a well known, well coded, non-trivial program and show the gismoe approach automatically generating new code for a part of it. We chose gzip, which is approximately 5000 lines of C code. It is open source and the software-artifact Infrastructure Repository (SIR) [15] includes a test suite for it. Almost all the time taken by gzip is used by one function (`longest_match`). The `longest_match` source code consists of several nested loops covering about two pages. We showed GP can re-create it in the CUDA parallel programming language running on nVidia graphics cards. I.e. using a test suit GP can re-create a vital non-trivial component of a legacy program which runs in a very different environment.

7.1 CUDA template

Together with the CUDA compiler, nVidia supply lots of small examples of how to use their hardware and CUDA. We chose `scan_naive_kernel.cu` as the starting point for GP since it is a simple CUDA function (kernel) which scans its inputs and we know that the existing gzip C code (`longest_match`) scans its inputs. However its too naive to hope for a speed up. Instead we merely used it as a "proof of concept" example.

`scan_naive_kernel.cu` is 15 lines long. It was converted into a straight forward BNF grammar with 169 rules. (The BNF grammar and training examples are available, e.g. via FTP.) The BNF grammar encapsulates what the GPU manufacturer has told us about using his graphics hardware but contains nothing about gzip. By using it, we ensure that the code generated by GP will be syntactically correct, it will be compiled by the CUDA compiler (nvcc), it will run and the grammar enforces termination. Thus, although the GP individuals may be very poor, we will be able to run them on the graphics hardware and assign each a fitness value.

7.2 Fitness

An instrumented version of gzip was run on the whole SIR gzip test suite. This yielded 1 599 028 records containing the inputs to `longest_match` and the answer it returned. This distribution is very asymmetric and contains many repeated tests and tests comparing very short strings. For evolving CUDA versions of `longest_match` the duplicate test cases where removed and the fraction of harder tests was increased by randomly selecting from tests with inputs of the same length. This gives 29 315 tests for training GP.

The computational load of testing many generations of complete populations of CUDA kernels on all 29 315 tests is still too much. Therefore each generation, 100 different tests are chosen from the 29 315 available. This not only reduces cost of the fitness function but improves generalisation [19]. (This is a bit like Gathercole's DSS [11], as used commercially [10], but spreads training data evenly rather than learning which training examples are particularly difficult.)

Each generation, GP uses the BNF template grammar to create 1000 CUDA kernels. To avoid the overhead of running nvcc 1000 times, the GP individuals are stored in ten files, each containing 100 kernels. The ten object files are linked with test harness into a single binary executable. The parallel operation of the nVidia graphics card allows all 1000 kernels to be tested in parallel and their answers are returned to the host PC and compared with the answer gzip's `longest_match` gave when given the same inputs.

Each GP individual's performance is the sum of the (absolute) error between its answer and that of gzip on all one hundred test cases being used in this generation. (Evolution has to minimise these errors). There is also a penalty for those programs which ignore their inputs and return the same answer for all test cases. The penalty is set so large that these programs effectively have no chance of having children in the next generation.

Seventy one percent of the initial random population ignore their inputs and return zero regardless. However this fraction quickly fell to about 20% but even after 100 generations about 7% of children produced by highly fit parents are useless constants.

Due to the continuous resampling of the 100 test cases used, fitness did not improve monotonically but over time the error reduced until in generation 50 the first kernel which makes no errors on its 100 test cases was found. Within a few generations approximately 74% of the population passed all the tests, even though they have not seen them before.

Figure 2 shows all the trees in the GP population after 100 generations and shows that large parts of the trees are identical (darker colour) even though each tree is different. Conversely plots of size v. depth emphasis trees are different but show they evolve under that action of crossover, etc. from whatever their initial shape is to have random shapes, e.g. [23, page 205].

7.3 Evolved Solution

Figure 3 shows one of the solutions evolved in generation 55. We would normally clean up code before presenting it (e.g. to remove bloat [20]) however we have coloured the essential code in Figure 3 to highlight it.

With small examples, like Figure 3, unwanted code can be removed by hand. E.g. when checking the code is correct. However there are a number of automated techniques to do this. For example Le Goues *et al.* [24] use delta debugging, whilst (after the solution has been found) we have used multi-objective evolution in which the original (functional) objectives are augmented by another which asks GP to minimise the solutions [17].

8 Conclusions

Genetic programming has been used in may diverse applications. These applications now include applying GP to software engineering itself. It has been applied to software engineering benchmarks, including both single and multi-objective Pareto optimisation and to small bespoke components, such as dynamic memory managers.

```
__device__ int kernel978(const uch *g_idata, const int strstart1, const int strstart2)
{
int thid = 0;
int pout = 0;
int pin = 0 ;
int offset = 0;
int num_elements = 258;
 for (offset = 1 ; G_idata( strstart1+ pin ) == G_idata( strstart2+ pin ) ;offset ++ )
{
if(!ok()) break;
thid = G_idata( strstart2+ thid ) ;
  pin = offset ;
}
return pin ;
}
```

Figure 3: Automatically generated C++ code for the gzip CUDA kernel [21]. Fixed text in blue is given by a BNF grammar, the rest is evolved by GP. Text highlighted in red is critical for its operation.

Also GP has been applied to repair sizable programs and automatically fixed real bugs and in total more than a million lines of real C source code has been processed. Section 7 showed GP can automatically port a non-trivial critical component of an existing well known unix utility (gzip) to a new environment (CUDA). In current work we expect genetic programming to improve still bigger programs.

Acknowledgement: Funded by EPSRC grant EP/I033688/1.

References

[1] T. Ackling et al. Evolving patches for software repair. In N. Krasnogor et al., editors, *GECCO 2011*, pages 1427–1434, Dublin, Ireland, 12-16 July 2011. ACM.

[2] A. Arcuri. On the automation of fixing software bugs. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1003–1006, Leipzig, Germany, 2008. ACM. Doctoral symposium session.

[3] P. Berarducci et al. GEVOSH: Using grammatical evolution to generate hashing functions. In E. G. Berkowitz, editor, *Proceedings of the Fifteenth Midwest Artificial Intelligence and Cognitive Sciences Conference, MAICS 2004*, pages 31–39, Chicago, USA, Apr. 16-18 2004. Omnipress.

[4] C. Cadar et al. Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In K. Sullivan, editor, *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 81–86, Santa Fe, New Mexico, USA, 7-11 Nov. 2010. ACM.

[5] A. Cotillon et al. Android genetic programming framework. In A. Moraglio et al., editors, *Proceedings of the 15th European Conference on Genetic Programming, EuroGP 2012*, volume 7244 of *LNCS*, pages 13–24, Malaga, Spain, 11-13 Apr. 2012. Springer Verlag.

[6] J. M. Daida et al. Visualizing tree structures in genetic programming. *Genetic Programming and Evolvable Machines*, 6(1):79–110, Mar. 2005.

[7] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr 2002.

[8] C. Estebanez et al. Evolving hash functions by means of genetic programming. In M. Keijzer et al., editors, *GECCO 2006:*, volume 2, pages 1861–1862, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[9] S. Forrest et al. A genetic programming approach to automated software repair. In G. Raidl et al., editors, *GECCO 2009*, pages 947–954, Montreal, 8-12 July 2009. ACM. Best paper.

[10] J. A. Foster. Review: Discipulus: A commercial genetic programming system. *Genetic Programming and Evolvable Machines*, 2(2):201–203, June 2001.

[11] C. Gathercole and P. Ross. Some training subset selection methods for supervised learning in genetic programming. Presented at ECAI'94 Workshop on Applied Genetic and other Evolutionary Algorithms, 1994.

[12] M. Harman et al. The GISMOE challenge: Constructing the pareto program surface using genetic programming to find better programs. In *The 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 12)*, Essen, Germany, Sept. 3-7 2012. ACM.

[13] B. Harvey et al. Towards byte code genetic programming. In W. Banzhaf et al., editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 1999*, volume 2, page 1234, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[14] D. Hussain and S. Malliaris. Evolutionary techniques applied to hashing: An efficient data retrieval method. In D. Whitley et al., editors, *GECCO-2000*, page 760, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering, ICSE-16*, pages 191–200, May 1994.

[16] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[17] W. B. Langdon. *Genetic Programming and Data Structures.* Kluwer, 1998.

[18] W. B. Langdon. Web usage of the GP bibliography. *SIGEvolution newsletter of the ACM Special Interest Group on Genetic and Evolutionary Computation*, 1(4):16–21, Dec. 2006.

[19] W. B. Langdon. A many threaded CUDA interpreter for genetic programming. In A. I. Esparcia-Alcazar et al., editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 146–158, Istanbul, 7-9 Apr. 2010. Springer.

[20] W. B. Langdon et al. The evolution of size and shape. In L. Spector et al., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, 1999.

[21] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In P. Sobrevilla, editor, *2010 IEEE World Congress on Computational Intelligence*, pages 2376–2383, Barcelona, 18-23 July 2010. IEEE.

[22] W. B. Langdon and J. P. Nordin. Seeding GP populations. In R. Poli et al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 304–315, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[23] W. B. Langdon and R. Poli. *Foundations of Genetic Programming.* Springer-Verlag, 2002.

[24] C. Le Goues et al. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan.-Feb. 2012.

[25] E. Lukschandl et al. Evolving routing algorithms with the JBGP-System. In R. Poli et al., editors, *Evolutionary Image Analysis, Signal Processing and Telecommunications: First European Workshop, EvoIASP'99 and EuroEcTel'99*, volume 1596 of *LNCS*, pages 193–202, Goteborg, Sweden, 28-29 May 1999. Springer-Verlag.

[26] E. Lukschandl et al. Distributed java bytecode genetic programming. In R. Poli et al., editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 316–325, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[27] M. Orlov and M. Sipper. Flight of the FINCH through the Java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–182, Apr. 2011.

[28] R. Poli et al. *A field guide to genetic programming.* Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[29] J. L. Risco-Martin et al. A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. *Parallel Computing*, 36(10-11):572–590, 2010.

[30] P. Rodriguez-Mier et al. Composition of web services through genetic programming. *Evolutionary Intelligence*, 3(3-4):171–186, 2010.

[31] E. Schulte et al. Automated program repair through the evolution of assembly code. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 313–316, Antwerp, 20-24 Sept. 2010. ACM.

[32] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[33] P. Sitthi-amorn et al. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(6):article:152, Dec. 2011.

[34] W. Weimer et al. Automatically finding patches using genetic programming. In S. Fickas, editor, *International Conference on Software Engineering (ICSE) 2009*, pages 364–374, Vancouver, May 16-24 2009.

[35] T. Weise and K. Tang. Evolving distributed algorithms with genetic programming. *IEEE Transactions on Evolutionary Computation*, 16(2):242–265, Apr. 2012.

[36] D. R. White et al. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation*, 15(4):515–538, Aug. 2011.