SSBSE-2015, Yvan Labiche and Marcio Barros *Eds.*, LNCS 9275, p12–28, Bergamo, Italy, 5-7 Sept. Springer. Preprint DOI:10.1007/978-3-319-22183-0_2

Genetic Improvement of Software for Multiple Objectives

W. B. Langdon

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

Abstract. Genetic programming (GP) can increase computer program's functional and non-functional performance. It can automatically port or refactor legacy code written by domain experts. Working with programmers it can grow and graft (GGGP) new functionality into legacy systems and parallel Bioinformatics GPGPU code. We review Genetic Improvement (GI) and SBSE research on evolving software.

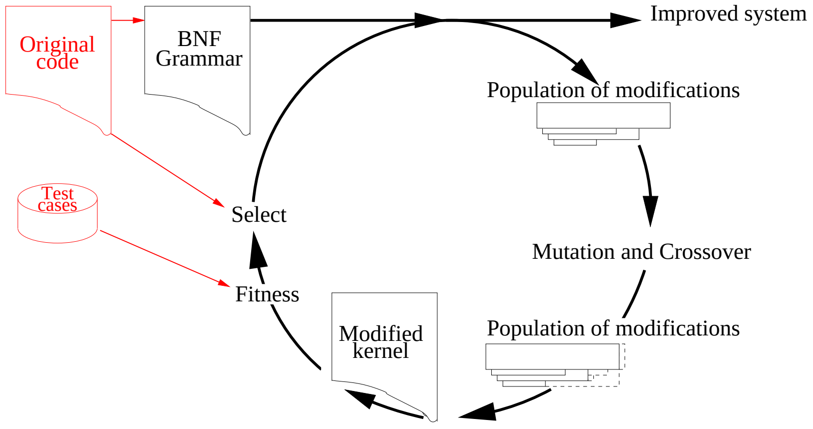

Fig. 1. Genetic Improvement of Program Source Code

1 Introduction

Although the idea of using evolutionary computation to improve existing software has been in the air for a little while [1], the use of genetic programming (GP) [2,3] to improve manually written code starts to take off in 2009 (see Figure 2). First with Wes Weimer et al.'s prize winning automatic bug fixing work [4,5,6,7,8,9] Section 3.1) and also Orlov and Sipper's [10] use of GP to improve manually written code by using it to seed the GP's population [11] (Section 2.3). The GISMO research project started four years ago with the lofty aim of transforming the way we think about and produce software. Now nearing its end, we can point to some successful applications (Sections 3.2 to 5.3) but perhaps the major impact has been the growth of "Genetic Improvement" [12] and the increasing acceptance that search based optimisation [13] can not only aid software engineers but also act upon their software directly.

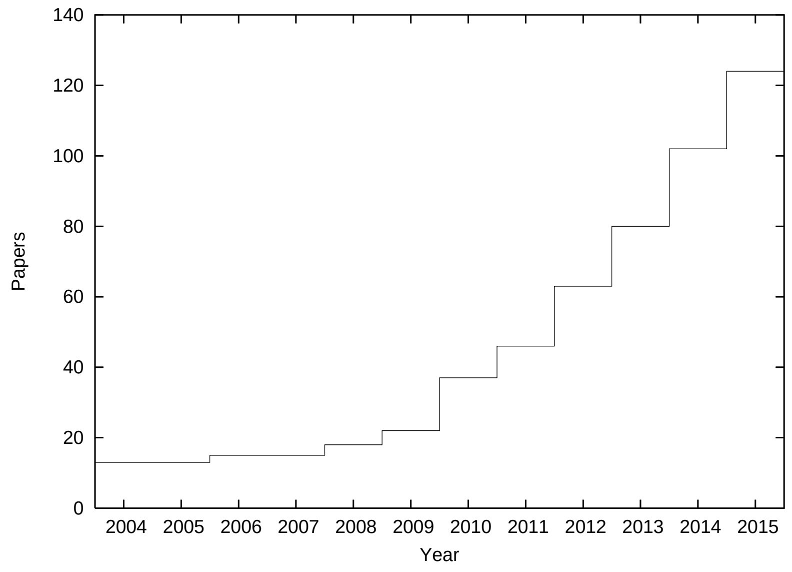

Fig. 2. Recent growth in number of entries in the genetic programming bibliography applying GP to generate or improve software (May 2015)

We shall give an overview of Genetic Improvement (GI). This is based in part on "Genetically Improved Software" [12] and work presented at the first international event on GI (held in Madrid 12^{th} July 2015 [14]). GI is the use of optimisation techniques such as Genetic Algorithms and Genetic Programming [2,3] to software itself. Although any optimisation technique might be used, so far published work has concentrated upon using GP to improve human written source code.

In the next section we start by briefly summarising research which evolved complete software [15] and then move on to GI. Section 3 starts with automatically fixing real bugs in real C/C++ programs (Section 3.1). This is followed by reviews of the GISMO project's work on gzip (Section 3.2), Bioinformatics (Section 3.3) and parallel computing (Sections 3.4 to 3.6). Section 4 describes evolving a human competitive version of MiniSAT from multiple existing programs. Whilst Section 5 describes three examples of GP and programmers working together including obtaining a 10 000 fold speedup. The last sections (Sections 6 and 7) conclude with the project's main lessons.

2 Evolving useful Programs from Primordial Ooze

2.1 Hashes, Caches and Garbage Collection

Three early examples of real software being evolved using genetic programming are: hashing, caching and garbage collection. Each has the advantages of being small, potentially of high value and difficult to do either by hand or by theoretically universal principles. These include examples where GP generate code exceeded the state-of-the art human written code. Whilst this is not to say a human could not do better. Indeed they may take inspiration, or even code, from the evolved solution. It is that to do so, requires a programmer skilled in the art, for each new circumstance. Whereas, at least in principle, the GP can be re-run for each new use case and so automatically generate an implementation specific to that user.

Starting with Hussain and Malliaris [16] several teams have evolved good hashing algorithms ([17], [18] and [19]).

Paterson showed GP can create problem specific caching code [20]. O'Neill and Ryan [21] used their Grammatical Evolution approach also to create cache code. Whilst Branke et al. [22] looked at a slightly different problem: deciding which documents to retain to avoid fetching them again across the Internet.

Many computer languages provide a dynamic memory manager, which frees the programmer of the tedium of deciding exactly which memory is in use and provides some form of garbage collection whereby memory that is no longer in use can be freed for re-use. Even with modern huge memories, memory management can impose a significant overhead. Risco-Martin et al. [23] showed the GP can generate an optimised garbage collector for the C language [24].

2.2 Mashups, Hyper-heuristics and Multiplicity Computing

The idea behind web services is that useful services should be easily constructed from services across the Internet. Such hacked together systems are known as web mashups. A classic example is a travel service which invokes web servers from a number of airlines and hotel booking and car hire services, and is thus able to provide a composite package without enormous coding effort in itself. Since web services must operate within a defined framework ideally with rigid interfaces, they would seem to be ideal building blocks with which genetic programming might construct high level programs. Starting with Rodriguez-Mier, several authors have reported progress with genetic programming evolving composite web services [25,26,27].

There are many difficult optimisation problems which in practise are efficiently solved using heuristic search techniques, such as genetic algorithms. However typically the GA needs to be tweaked to get the best for each problem. This has lead to the generation of hyper-heuristics [28], in which the GA or other basic solver is tweaked automatically. Typically genetic programming is used. Indeed some solvers have been evolved by GP combining a number of basic techniques as well as tuning parameters or even re-coding GA components, such as mutation operators [29].

A nice software engineering example of heuristics is compiler code generation. Typically compilers are expected not only to create correct machine code but also that it should be in some sense be "good". Typically this means the code should be fast or small. Mahajan and Ali [30] used GP to give better code generation heuristics in Harvard's MachineSUIF compiler.

Multiplicity computing [31] seeks to over turn the current software monoculture where one particular operating system, web browser, software company, etc., achieves total dominance of the software market. Not only are such monopolies dangerous from a commercial point of view but they have allowed widespread problems of malicious software (especially computer viruses) to prosper. Excluding specialist areas, such as mutation testing [32,33], so far there has been only a little work in the evolution of massive numbers of software variants [34]. Only software automation (perhaps by using genetic programming) appears a credible approach to N-version programming (with N much more than 3). N-version programming has also been proposed as a way of improving predictive performance by voting between three or more classifiers [35,36] or using other non-linear combinations to yield a higher performing multi-classifier [37,38].

Other applications of GP include: creating optimisation benchmarks which demonstrate the relative strengths and weaknesses of optimisers [39] and first steps towards the use of GP on mobile telephones [40], connections to software product lines [41], security [42,43] and adaptability [44].

2.3 Genetic Programming and Non-Function Requirements

Andrea Arcuri was in at the start of inspirational work on GP showing it can create real code from scratch. Although the programs remain small, David White, he and John Clark [45] also evolved programs to accomplish real tasks such as creating pseudo random numbers for ultra tiny computers where they showed a trade off between "randomness" and energy consumption.

The Virginia University group (see next section) also showed GP evolving Pareto optimal trade offs between speed and fidelity for a graphics hardware display program [46]. Evolution seems to be particularly suitable for exploring such trade-offs [47,48] but (except for the work described later in this chapter) there has been little research in this area.

Orlov and Sipper [10] describe a very nice system, Finch, for evolving Java byte code. Effectively the GP population instead of starting randomly [49] is seeded [11] with byte code created by compiling the initial program. The Finch crossover operator acts on Java byte code to ensure the offspring program are also valid java byte code.

Archanjo and Von Zuben [50] present a GP system for evolving small business systems. They present an example of a database system for supporting a library of books.

Ryan [51] and Katz and Peled [52] provide interesting alternative visions. In genetic improvement the performance, particularly the quality of the mutated program's output, is assessed by running the program. Instead they suggest each mutation be provably correct and thus the new program is functionally the same as the original but in some way it is improved, e.g. by running in parallel. Katz and Peled [52] suggests combining GP with model checking to ensure correctness.

Cody-Kenny et al. [53] showed on a dozen Java examples (mostly different implementations of various types of sort from rosettacode.org) that GP was able to reduce the number of Java byte code instructions executed.

Schulte et al. [54] describes a system which can further optimise the low level Intel X86 code generated by optimising compilers. They show evolution can reduce energy consumption of non-trivial programs. (Their largest application contains 141 012 lines of code.) Mrazek et al. [55] showed it was possible to evolve an important function (the median) in a variety of machine codes.

3 Improvement of Substantial Human Written code

3.1 Automatic Bug Fixing

As described in the previous two sections, recently genetic programming has been applied to the production of programs itself, however so far relatively small programs have been evolved. Nonetheless GP has had some great successes when applied to existing programs. Perhaps the best known work is that on automatic bug fixing [56]. Particularly the Humie award winning[1] work of Westley Weimer (Virginia University) and Stephanie Forrest (New Mexico) [5]. This has received multiple awards and best paper prizes [4,6]. GP has been used repeatedly to automatically fix most (but not all) real bugs in real programs [57]. Weimer and Le Goues have now shown GP bug fixing to be effective on over a million lines of C++ code. Once GP had been used to *do the impossible* others tried [58,59,60] and it was improved [61] and also people felt brave enough to try other techniques, e.g. [62,63,64]. Indeed their colleague, Eric Schulte, has shown GP can operate below the source code level, e.g. [43]. In [8] he showed bugs can be fixed via mutating the assembler code generated by the compiler or even machine code [65]. After Weimer and co-workers showed that automatic bugfixing was not impossible, people studied the problem more openly. It turns out, for certain real bugs, with modern software engineering support tools, such as bug localisation (e.g. [66]), the problem may not even be hard [67].

Formal theoretical analysis [68] of evolving sizable software is still thin on the ground. Much of the work presented here is based on GP re-arranging lines of human written code. In a study of 420 million lines of open source software Gabel and Su [69] showed that excluding white space, comments and details of variable names, any human written line of code has probably been written before. In other words, given a sufficiently large feedstock of human written code, current programs could have been written by re-using and re-ordering existing source code. In many cases in this and the following sections, this is exactly what GP is doing. Schulte et al. [9] provides a solid empirical study which refutes the common assumption that software is fragile. (See also Figure 3). While a single random change may totally break a program, mutation and crossover operations can be devised which yield populations of offspring programs in which some may be very bad but the population can also contains many reasonable programs and even a few slightly improved ones. Over time the Darwinian processes of fitness selection and inheritance [70] can amplify the good parts of the population, yielding greatly improved programs.

[1] Human-competitive results presented at the annual GECCO conference http://www.genetic-programming.org/combined.php

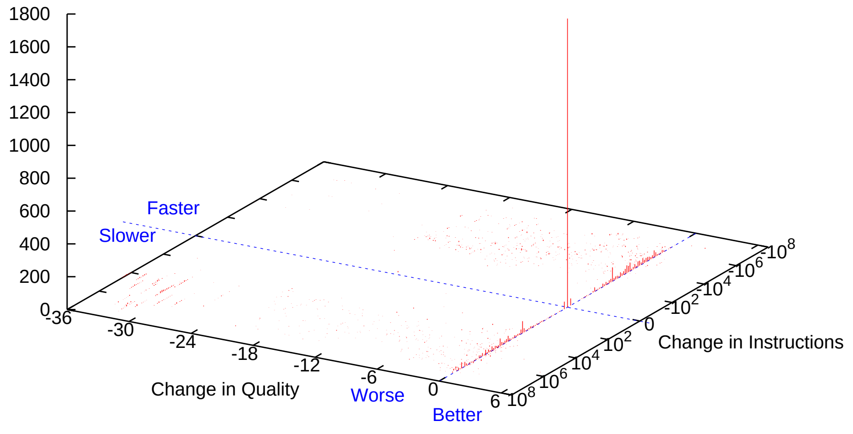

Fig. 3. Histogram of impact on speed and solution quality made by single mutations to Bowtie2 (Section 3.3). Many changes have no impact on quality, plotted along x=0. Indeed a large number do not change its speed either (note spike at the origin). There are a few mutations which give *better* quality solutions. It is from these GP evolves a seventy fold speed up.

3.2 Auto Porting Functionality

The Unix compression utility gzip was written in C in the days of Digital Equipment Corp.'s mini-computers. It is largely unchanged. However there is one procedure (of about two pages of code) in it, which is so computationally intensive that it has been re-written in assembler for the Intel 86X architecture (i.e. Linux). The original C version is retained and is distributed as part of Software-artifact Infrastructure Repository sir.unl.edu [72]. We showed genetic programming could evolve a parallel implementation for an architecture not even dreamt of when the original program was written [71].

Whereas Le Goues and others use the original program's AST (abstract syntax tree) to ensure that many of the mutated programs produced by GP compile, we have used a BNF grammar (see Figure 1). For CUDA gzip we created our grammar from generic code written by nVidia. The original function in gzip was instrumented to record its inputs and its outputs each time it was called (see Figure 4). Essentially GP was told to create parallel code from the BNF grammar which when given a small number of example inputs (based on the instrumented code, Figure 4) returned the same answers. The resulting parallel code is functionally the same as the old gzip code.

3.3 Bowtie2^{GP} Improving 50 000 lines of C++

Finding the best match between strings is the life blood of Bioinformatics. Wikipedia lists more than 140 programs which do some form of Bioinformatics string matching. Modern NextGen sequencing machines generate billions of (albeit very noisy) DNA base-pair sequences.

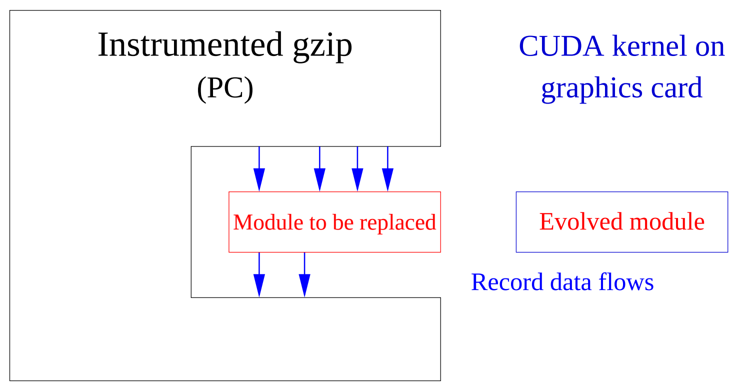

Fig. 4. Auto porting a program module to new hardware. The original code is instrumented to record the inputs (upper blue arrows) to the target function (red) and the result (lower blue arrows) it calculates. These become the test suite and fitness function when evolving the replacement code [71].

The authors of all this software are in a bind. For their code to be useful they have to chose a tradeoff between speed, machine resources, quality of solution and functionality, which will: 1) be important to Bioinformatics and 2) not be immediately dominated by other programs. They have to choose a target point when they start, as once basic design choices (e.g. target data sources and type and size of computer) have been made, few research teams have the resources to discard what they have written and start again. Potentially genetic programming offers them a way of exploring this space of tradeoffs [47,48]. (Figure 5 shows a two dimensional trade off between speed and quality.) GP can potentially produce many programs across a Pareto optimal front and so might say "here is a trade-off which you had not considered". This could be very useful even if the development team insist on coding a solution.

We have made a start by showing GP can transform human written DNA sequence matching code, moving it from one tradeoff point to another. In our example, the new program is specialised to a particular data source and sequence problem for which it is on average more than 70 times faster. Indeed on this particular problem, we were fortunate that not only is the variant faster but indeed it gives a slight quality improvement on average [75].

3.4 BarraCUDA

BarraCUDA [76] like Bowtie2$^{\mathrm{GP}}$ looks up DNA sequences. However BarraCUDA uses the computational power of nVidia graphics accelerators (GPUs) to process hundreds of thousands of short DNA sequences in parallel. Despite having been written by experts both on Bioinformatics and on GPUs, GP when targeted by a Human on a particular kernel was able to speed up that kernel by more than 100 times. Of course this kernel is only part of the whole program and overall speed up is more modest. Nevertheless on real examples the GI code [77] can be

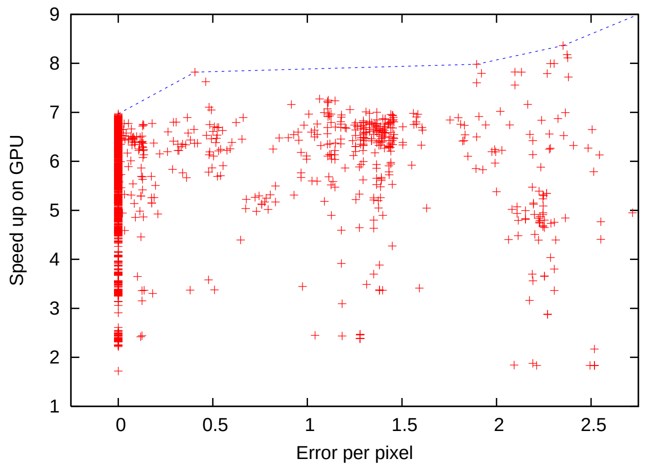

Fig. 5. Example of automatically generated Pareto tradeoff front [48]. Genetic programming used to improve 2D Stereo Camera code [73] for modern nVidia GPU [74]. Left (above 0) many programs are faster than the original code written by nVidia's image processing expert (human) and give exactly the same answers. Many other automatically generated programs are also faster but give different answers. Some (cf. dotted blue line) are faster than the best zero error program.

up to three times faster than the previous (100% human) version Indeed with a top end K80 Tesla BarraCUDA can now be more than ten times faster than BWA on a 12 core CPU [78].

The GI version of BarraCUDA has has been in use via SourceForge since 20 March 2015. In the first two months it was downloaded 230 times.

3.5 Genetically Improved GPU based Stereo Vision

Originally the StereoCamera [73] system was specifically written by nVidia's image processing expert to show off their hardware. However in [74] we show GP is able to improve the code for hardware which had not even been designed when it was originally written. Indeed GP gave up to a seven fold speed up in the graphics kernel.

3.6 Genetically Improved GPU based 3D Brain Imaging

GI can automatically tune an important CUDA kernel in the NiftyReg [79] medical imaging package for six very different graphics cards (see [80, Fig. 1]).

4 Plastic Surgery: Better MiniSAT from multiple Authors

Genetic Improvement has also been used to create an improved version of C++ code from multiple versions of a program written by different domain experts. The Boolean satisfiability community has advanced rapidly since the turn of the century. This is partly due to the “MiniSAT hack track”, which encourages people to make small changes to the MiniSAT code. Some of these variants were evolved together to give a new MiniSAT tailored to solve interaction testing problems [81]. It received a human competitive award (HUMIE) in 2014.

5 Creating and Incorporating New Functionality

5.1 Babel Pidgin: Adding Double Language Translation Feature

Jia et al. [82] describes another prize winning GI system. GP *including human hints* was able to evolved new functionality externally and then search based techniques [83] were used to graft the new code into an existing program (pidgin) of more than 200 000 lines of C++.

5.2 Grow and Serve GP Citations

The GP grew code and grafted it into a Django web server to provide a citation service based in Google Scholar. As an experiment, the GP bibliography used this GP produced service. In the first 24 hours it was used 369 times from 29 countries [84].

5.3 $10^4\times$ Speedup on Folding RNA Molecules

GP was told approximately where to evolve new code within an existing parallel program pknots [85]. It converted the original CUDA kernel, which processes one Dynamic Programming matrix at a time, into one which processes multiple matrices. Although only trained on five matrices, the evolved kernel can work on up to 200 000 matrices, delivering speed ups of up ten thousand fold [86].

6 GISMO Key Findings

The idea of using existing code as its own specification is very valuable (see also Figure 4). Many existing specifications are informal and often incomplete. Whilst the existing code may contain errors, the fact that it is in use shows it to be near what is wanted and so can be used as a basis for new work. Also by using existing test suites or automatic test case generation tools, the output of the existing code under test can be used as its own test oracle and indeed the test oracle for the new code. The number of tests available for validating the new code is now only limited by machine (rather than human) resource limits. However many of the GISMO examples given above show a very small number of tests, perhaps just a handful (provided they are frequently changed), may be sufficient to guide the GI. With a much larger pool of tests or other validation techniques being available post evolution. Indeed when working at the source code level, GI generated software can potentially be validated by any of today's techniques, including manual inspection as well as intensive regression testing.

While human written code may be optimised for a particular objective, GI can optimise it for multiple objectives (Figure 5). This may be particularly important if, whilst maintaining functionality of the existing code, GI can suggest unsuspected tradeoffs between speed, memory usage, energy consumption, network loading, etc. and quality.

Although code evolved from scratch tends to be small, grow and graft (GGGP) (Section 5) is a potential way around the problem. GGGP still evolves small new components but also uses GP to graft them into much bigger human written codes, thus creating large hybrid software.

The work on miniSAT (Section 4) shows GP can potentially scavenge not just code from the program it is improving but code from multiple programs by multiple authors. This GP plastic surgery [87] created in a few hours an award winning version of miniSAT tailored to solving an important software engineering problem. The automatically customised code was better at problems of this type than generic versions of miniSAT which has been optimised by leading SAT experts for years.

In software engineering there has always been a strong pressure to keep software as uniform as possible. To try and keep all the users running just a few versions. With the popularity of software product lines and possibility of multiplicity computing, we see an opposing trend. A desire to reduce the impact of malicious programmers by avoiding the current software monoculture and for more bespoke and adaptable systems. Already there is a little GI work in both avenues.

7 GISMO Impact

While the GI version of BarraCUDA has been in use since March 2015, perhaps the biggest impact of the project has been to show automatic or even human assisted evolution of software can be feasible. Before 2009 automatic bug fixing was regarded as fantasy but following [4] this changed. The biggest impact of the project will be encouraging people to do what was previously considered impossible.

Sources and Datasets

The grammar based genetic programming systems and associated benchmarks are available via the GISMO project web pages. Other authors have also made their systems available (e.g. Le Goues' genprog) or may be asked directly.

Acknowledgements

I am grateful for the assistance of Andrea Arcuri, Robert Feldt, Marc Schoenauer, Wes Weimer and Darrell Whitley.

Tesla donated by nVidia.

References

1. Ryan, C., Ivan, L.: Automatic parallelization of arbitrary programs. In Poli, R., et al., eds.: EuroGP'99. LNCS 1598, Goteborg, Sweden, Springer-Verlag 244–254
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Natural Selection. MIT press (1992)
3. Poli, R., Langdon, W.B., McPhee, N.F.: 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).
4. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S.: Automatically finding patches using genetic programming. In Fickas, S., ed. ICSE 2009, Vancouver 364–374
5. Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In Raidl, G., et al., eds.: GECCO '09, Montreal, ACM 947–954 Best paper.
6. Weimer, W., Forrest, S., Le Goues, C., Nguyen, T.: Automatic program repair with evolutionary computation. Communications of the ACM **53**(5) (2010) 109–116
7. Fast, E., Le Goues, C., Forrest, S., Weimer, W.: Designing better fitness functions for automated program repair. In Branke, J., et al., eds.: GECCO '10, ACM 965–972
8. Schulte, E., Forrest, S., Weimer, W.: Automated program repair through the evolution of assembly code. In: ASE 2010, Antwerp, ACM 313–316
9. Schulte, E., Fry, Z.P., Fast, E., Weimer, W., Forrest, S.: Software mutational robustness. Genetic Programming and Evolvable Machines **15**(3) (2014) 281–312
10. Orlov, M., Sipper, M.: Flight of the FINCH through the Java wilderness. IEEE Trans. on EC **15**(2) (2011) 166–182
11. Langdon, W.B., Nordin, J.P.: Seeding GP populations. In Poli, R., et al., eds.: EuroGP'2000. LNCS 1802., Edinburgh, Springer-Verlag 304–315
12. Langdon, W.B.: Genetically improved software. In Gandomi, A.H., et al., eds.: Handbook of Genetic Programming Applications. (Springer) Forthcoming.
13. Harman, M., Jones, B.F.: Search based software engineering. Information and Software Technology **43**(14) (2001) 833–839
14. Langdon, W.B., Petke, J., White, D.R.: Genetic improvement 2015 chairs' welcome. In GECCO'15 Companion, Madrid, ACM
15. Arcuri, A., Yao, X.: Co-evolutionary automatic programming for software development. Information Sciences **259** (2014) 412–432
16. Hussain, D., Malliaris, S.: Evolutionary techniques applied to hashing: An efficient data retrieval method. In Whitley, D., et al., eds.: GECCO-2000, Las Vegas, Nevada, USA, Morgan Kaufmann 760
17. Berarducci, P., Jordan, D., Martin, D., Seitzer, J.: GEVOSH: Using grammatical evolution to generate hashing functions. In Poli, R., et al., eds.: GECCO 2004 Workshop Proceedings, Seattle, Washington, USA
18. Estebanez, C., Saez, Y., Recio, G., Isasi, P.: Automatic design of noncryptographic hash functions using genetic programming. Computational Intelligence. Forthcoming.

19. Karasek, J., Burget, R., Morsky, O.: Towards an automatic design of non-cryptographic hash function. In: TSP 2011, Budapest 19–23
20. Paterson, N., Livesey, M.: Evolving caching algorithms in C by genetic programming. In Koza, J.R., et al., eds.: Genetic Programming 1997, Stanford University, CA, USA, Morgan Kaufmann 262–267
21. O'Neill, M., Ryan, C.: Automatic generation of caching algorithms. In Miettinen, K., et al., eds.: Evolutionary Algorithms in Engineering and Computer Science, Jyväskylä, Finland, John Wiley & Sons (1999) 127–134
22. Branke, J., Funes, P., Thiele, F.: Evolutionary design of en-route caching strategies. Applied Soft Computing **7**(3) (2006) 890–898
23. Risco-Martin, J.L., Atienza, D., Colmenar, J.M., Garnica, O.: A parallel evolutionary algorithm to optimize dynamic memory managers in embedded systems. Parallel Computing **36**(10-11) (2010) 572–590
24. Wu, F., Weimer, W., Harman, M., Jia, Y., Krinke, J.: Deep parameter optimisation. In: GECCO '15, Madrid, ACM
25. Rodriguez-Mier, P., Mucientes, M., Lama, M., Couto, M.I.: Composition of web services through genetic programming. Evolutionary Intelligence **3**(3-4) (2010) 171–186
26. Fredericks, E.M., Cheng, B.H.C.: Exploring automated software composition with genetic programming. In Blum, C., et al., eds.: GECCO '13 Companion, Amsterdam, The Netherlands, ACM 1733–1734
27. Xiao, Liyuan, Chang, Carl K., Yang, Hen-I, Lu, Kai-Shin, Jiang, Hsin-yi: Automated web service composition using genetic programming. In: COMPSACW 2012, Izmir 7–12
28. Burke, E.K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Ozcan, E., Qu, R.: Hyper-heuristics: a survey of the state of the art. JORS **64**(12) (2013) 1695–1724
29. Pappa, G.L., Ochoa, G., Hyde, M.R., Freitas, A.A., Woodward, J., Swan, J.: Contrasting meta-learning and hyper-heuristic research: the role of evolutionary algorithms. Genetic Programming and Evolvable Machines **15**(1) (2014) 3–35
30. Mahajan, A., Ali, M.S.: Superblock scheduling using genetic programming for embedded systems. In: ICCI 2008. IEEE 261–266
31. Cadar, C., Pietzuch, P., Wolf, A.L.: Multiplicity computing: a vision of software engineering for next-generation computing platform applications. In Sullivan, K., ed.: FoSER '10 FSE/SDP workshop, Santa Fe, New Mexico, USA, ACM 81–86
32. DeMillo, R.A., Offutt, A.J.: Constraint-based automatic test data generation. IEEE Trans. Software Engineering **17**(9) (1991) 900–910
33. Langdon, W.B., Harman, M., Jia, Y.: Efficient multi-objective higher order mutation testing with genetic programming. JSS **83**(12) (2010) 2416–2430
34. Feldt, R.: Generating diverse software versions with genetic programming: an experimental study. IEE Proceedings **145**(6) (1998) 228–236
35. Imamura, K., Foster, J.A.: Fault-tolerant computing with N-version genetic programming. In Spector, L., et al., eds.: GECCO-2001, San Francisco, California, USA, Morgan Kaufmann 178
36. Imamura, K., Soule, T., Heckendorn, R.B., Foster, J.A.: Behavioral diversity and a probabilistically optimal GP ensemble. Genetic Programming and Evolvable Machines **4**(3) (2003) 235–253
37. Langdon, W.B., Buxton, B.F.: Genetic programming for combining classifiers. In Spector, L., et al., eds.: GECCO-2001, San Francisco, California, USA, Morgan Kaufmann 66–73
38. Buxton, B.F., Langdon, W.B., Barrett, S.J.: Data fusion by intelligent classifier combination. Measurement and Control **34**(8) (2001) 229–234

39. Langdon, W.B., Poli, R.: Evolving problems to learn about particle swarm and other optimisers. In Corne, D., et al., eds.: CEC-2005, Edinburgh, UK, IEEE Press 81–88
40. Cotillon, A., Valencia, P., Jurdak, R.: Android genetic programming framework. In Moraglio, A., et al., eds.: EuroGP 2012. LNCS 7244., Malaga, Spain, Springer Verlag 13–24
41. Lopez-Herrejon, R.E., Linsbauer, L.: Genetic improvement for software product lines: An overview and a roadmap. In GECCO'15 Companion, Madrid, ACM
42. Landsborough, J., Harding, S., Fugate, S.: Removing the kitchen sink from software. In GECCO'15 Companion, Madrid, ACM
43. Schulte, E., Weimer, W., Forrest, S.: Repairing COTS router firmware without access to source code or test suites: A case study in evolutionary software repair. In GECCO'15 Companion, Madrid, ACM
44. Yeboah-Antwi, K., Baudry, B.: Embedding adaptivity in software systems using the ECSELR framework. In GECCO'15 Companion, Madrid, ACM
45. White, D.R., Arcuri, A., Clark, J.A.: Evolutionary improvement of programs. IEEE Trans. EC **15**(4) (2011) 515–538
46. Sitthi-amorn, P., Modly, N., Weimer, W., Lawrence, J.: Genetic programming for shader simplification. ACM Trans. Graphics **30**(6) (2011) article:152
47. Feldt, R.: Genetic programming as an explorative tool in early software development phases. In Ryan, C., Buckley, J., eds.: Proceedings of the 1st International Workshop on Soft Computing Applied to Software Engineering, University of Limerick, Ireland, Limerick University Press (1999) 11–20
48. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The GISMOE challenge: Constructing the Pareto program surface using genetic programming to find better programs. In: ASE 12, Essen, Germany, ACM 1–14
49. Lukschandl, E., Holmlund, M., Moden, E.: Automatic evolution of Java bytecode: First experience with the Java virtual machine. In Poli, R., et al., eds.: Late Breaking Papers at EuroGP'98, Paris, France, CSRP-98-10, The University of Birmingham, UK (1998) 14–16
50. Archanjo, G.A., Von Zuben, F.J.: Genetic programming for automating the development of data management algorithms in information technology systems. Advances in Software Engineering (2012)
51. Ryan, C.: Automatic re-engineering of software using genetic programming. Kluwer Academic Publishers (1999)
52. Katz, G., Peled, D.: Synthesizing, correcting and improving code, using model checking-based genetic programming. In Bertacco, V., Legay, A., eds.: HVC 2013. LNCS 8244, Haifa, Israel, Springer 246–261 Keynote Presentation.
53. Cody-Kenny, B., Lopez, E.G., Barrett, S.: locoGP: improving performance by genetic programming java source code. In GECCO'15 Companion, Madrid, ACM
54. Schulte, E., Dorn, J., Harding, S., Forrest, S., Weimer, W.: Post-compiler software optimization for reducing energy. In: ASPLOS'14, Salt Lake City, Utah, USA, ACM) 639–652
55. Mrazek, V., Vasicek, Z., Sekanina, L.: Evolutionary approximation of software for embedded systems: Median function. In GECCO'15 Companion, Madrid, ACM
56. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In Wang, J., ed.: in WCCI 2008, IEEE 162–168
57. Le Goues, C., Dewey-Vogt, M., Forrest, S., Weimer, W.: A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Glinz, M., ed.: ICSE 2012, Zurich 3–13

58. Wilkerson, J.L., Tauritz, D.: Coevolutionary automated software correction. In Branke, J., et al., eds.: GECCO '10, Portland, Oregon, USA, ACM 1391–1392
59. Bradbury, J.S., Jalbert, K.: Automatic repair of concurrency bugs. In Di Penta, M., et al., eds.: SSBSE '10, Benevento, Italy. Fast abstract.
60. Ackling, T., Alexander, B., Grunert, I.: Evolving patches for software repair. In Krasnogor, N., et al., eds.: GECCO '11, Dublin, Ireland, ACM 1427–1434
61. Kessentini, M., Kessentini, W., Sahraoui, H., Boukadoum, M., Ouni, A.: Design defects detection and correction by example. In: ICPC 2011, Kingston, Canada, IEEE 81–90
62. Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: SemFix: program repair via semantic analysis. In Cheng, B.H.C., Pohl, K., eds.: ICSE 2013, San Francisco, USA, IEEE 772–781
63. Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: ICSE 2013, San Francisco, USA 802–811
64. Tan, S.H., Roychoudhury, A.: relifix: Automated repair of software regressions. In Canfora, G., et al., eds.: ICSE 2015, Florence Italy, IEEE 471–482
65. Schulte, E., DiLorenzo, J., Weimer, W., Forrest, S.: Automated repair of binary and assembly programs for cooperating embedded devices. In: ASPLOS 2013, Houston, Texas, USA, ACM 317–328
66. Yoo, S.: Evolving human competitive spectra-based fault localisation techniques. In Fraser, G., et al., eds.: SSBSE 2012. LNCS 7515, Riva del Garda, Italy, Springer 244–258
67. Weimer, W.: Advances in automated program repair and a call to arms. In Ruhe, G., Zhang, Y., eds.: SSBSE 2013. LNCS 8084, Leningrad, Springer 1–3 Invited keynote.
68. Cody-Kenny, B., Barrett, S.: The emergence of useful bias in self-focusing genetic programming for software optimisation. In Ruhe, G., Zhang, Y., eds.: SSBSE 2013. LNCS 8084, Leningrad, Springer (2013) 306–311. Graduate Student Track.
69. Gabel, M., Su, Z.: A study of the uniqueness of source code. In: FSE '10, ACM 147–156
70. Darwin, C.: The Origin of Species. Penguin classics, 1985 edn. John Murray (1859)
71. Langdon, W.B., Harman, M.: Evolving a CUDA kernel from an nVidia template. In Sobrevilla, P., ed.: WCCI 2010, Barcelona, IEEE 2376–2383
72. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: ICSE 1994 191–200
73. Stam, J.: Stereo imaging with CUDA. Technical report, nVidia (2008)
74. Langdon, W.B., Harman, M.: Genetically improved CUDA C++ software. In Nicolau, M., et al., eds.: EuroGP 2014. LNCS 8599, Granada, Spain, Springer 87–99
75. Langdon, W.B., Harman, M.: Optimising existing software with genetic programming. IEEE Trans. EC **19**(1) (2015) 118–135
76. Klus, P., Lam, S., Lyberg, D., Cheung, M.S., Pullan, G., McFarlane, I., Yeo, G.S.H., Lam, B.Y.H.: BarraCUDA - a fast short read sequence aligner using graphics processing units. BMC Research Notes **5**(27) (2012)
77. Langdon, W.B., Lam, B.Y.H., Petke, J., Harman, M.: Improving CUDA DNA analysis software with genetic programming. In: GECCO '15, Madrid, ACM
78. Langdon, W.B., Lam, B.Y.H.: Genetically improved barraCUDA. Research Note RN/15/03, Department of Computer Science, University College London (2015)

79. Modat, M., Ridgway, G.R., Taylor, Z.A., Lehmann, M., Barnes, J., Hawkes, D.J., Fox, N.C., Ourselin, S.: Fast free-form deformation using graphics processing units. Computer Methods and Programs in Biomedicine **98**(3) (2010) 278–284
80. Langdon, W.B., Modat, M., Petke, J., Harman, M.: Improving 3D medical image registration CUDA software with genetic programming. In Igel, C., et al., eds.: GECCO '14, Vancouver, BC, Canada, ACM 951–958
81. Petke, J., Harman, M., Langdon, W.B., Weimer, W.: Using genetic improvement and code transplants to specialise a C++ program to a problem class. In Nicolau, M., et al., eds.: EuroGP 2014. LNCS 8599, Granada, Spain, Springer 137–149
82. Harman, M., Jia, Y., Langdon, W.B.: Babel pidgin: SBSE can grow and graft entirely new functionality into a real world system. In Le Goues, C., Yoo, S., eds.: SSBSE 2014. LNCS 8636, Fortaleza, Brazil, Springer 247–252. Winner Challange Track.
83. Harman, M.: Software engineering meets evolutionary computation. Computer **44**(10) (2011) 31–39 Cover feature.
84. Jia, Y., Harman, M., Langdon, W.B., Marginean, A.: Grow and serve: Growing Django citation services using SBSE. In Yoo, S., Minku, L., eds.: SSBSE 2015 Challenge Track, Bergamo, Italy
85. Reeder, J., Steffen, P., Giegerich, R.: pknotsRG: RNA pseudoknot folding including near-optimal structures and sliding windows. Nucleic Acids Research **35**(suppl 2) (2007) W320–W324
86. Langdon, W.B., Harman, M.: Grow and graft a better CUDA pknotsRG for RNA pseudoknot free energy calculation. In GECCO'15 Companion, Madrid, ACM.
87. Barr, E.T., Brun, Y., Devanbu, P., Harman, M., Sarro, F.: The plastic surgery hypothesis. In Orso, A., et al., eds.: FSE 2014, Hong Kong, ACM