# GP vs GI: if you can't beat them, join them.

John R. Woodward
University of Stirling
Stirling
Scotland, United Kingdom
jrw@cs.stir.ac.uk

Colin G.Johnson
University of Kent
Kent
England, United Kingdom
C.G.Johnson@kent.ac.uk

Sandy Brownlee
University of Stirling
Stirling
Scotland, United Kingdom
sbr@cs.stir.ac.uk

## Keywords

Genetic Improvement, Genetic Programming

Genetic Programming (GP) has been criticized for targeting irrelevant problems [12]. This is also true of the wider machine learning community [11] which has become detached from the source of the data it is using to drive the field forward. However, recently Genetic Improvement (GI) has entered the field, providing a fresh perspective on automated programming. In contrast to GP, GI begins with existing software, and therefore *immediately* has the aim of tackling real software. As GP is the main approach in GI to manipulating programs, this connection with real software should persuade the GP community to confront some of the issues it set out to tackle originally i.e. evolving real software.

There are a number of impressive examples of GI in the literature [7]. These include papers on GenProg, which fixed a number of bugs in real software for a reported 8 dollars each. Work by Langdon has showcased the potential of GI on different applications including gene sequencing software and vision software [5].

The GP community has tackled a number of toy problems including the even parity problem. [1]. For example, we need all input features (all $n$ bit) to be able to classify an input at even or not. If a single bit is missing from the input, then we cannot solve the problem, as each bit is essential in determining the class. Therefore, we cannot use machine learning techniques such as feature selection methods. There is also no correlation between input variables which can be used to solve the problem. [6]. Surely the GP community can be a little bit more ambitious than this.

One of the implications of the No Free Lunch theorems is that, performance on one class of problems is off set against the performance on the disjoint set of problems. Therefore, it is difficult to draw conclusions about the performance of one metaheuristic on a new problem instance. However, there is one interesting fact about program spaces that is different to search spaces typically targeted by metaheuristics. In GP, programs are assigned fitness values: a program computes a function, which is then assigned a fitness value. The mapping between program and function depends on the programming language being used (i.e. the function set), and is independent of the function we are trying to compute. While the mapping between functions and error scores is problem dependent (and in fact defines the problem we are tacking).

Real software habitually contains loops, defined functions (procedures, methods, macros, routines), and so GI has to deal with the reality of existing software systems. However, most of the GP literature is not concerned with Turing Complete instruction sets. GP has also made less use of Automatically Defined Functions [3] over the past few years despite the ability to build modules being so central to constructing large programs. A review of GP with Turing Complete instruction sets reveals that programs typically consist of a small number of loops [14]. In contrast, the vast majority of GI papers are applied to programs with loops ($for$ and $while$ loops), programs that contain defined functions, and usually side effects (e.g. writing to file) as most programs of interest will have some sort of side effect.

GP has examined sorting along with other short programs. It has also been targeted with GI. While short programs may be classified in some senses as toy problems, they are of interest when included as part of a larger program which invokes them many times, and support the core of the overall functionality.

We can classify programs into 4 types, depending on how they are executed. 1) 1 where all nodes in the syntax tree are executed once (e.g. programs constructed with a function set $f1$ of arithmetic operators {+, -, *, %}). 2) $0-1$ where nodes are either executed once or not (e.g. programs constructed with a function set f2 containing logical operators {AND, OR, NOT}, where short circuiting is used.) 3) $n-programs$ containing for loops with a determined number of iterations (bounded execution time). 4) $infinityprograms$ with while loops with an unknown termination condition (unbounded execution time)

Broadly speaking, GP is with the former two types of program, while GI is with the latter two types. Adopting a GI approach, which deals with software, forces us to confront the fact that different programs can take vastly different amounts to time to execute. (of course GP work exists using Turing Complete instruction sets, and there is no reason why GI could not be applied to programs consisting of instruction sets such as f1 or f2). With the first two types, programs will execute in a comparatively short amount of



*WOODSTOCK '97 El Paso, Texas USA*



DOI: 10.475/123_4

time (bound by the size of the program). While with the last two types, programs make take an awfully long time to terminate (possibly not halting).

When presented with a problem we wish to tackle with GP, we have to make the choice of instructions to include in the instruction set. This is an open question as to how best do this, and is usually arrived at after a trial and error process. However, when presented with a GI problem of how to improve a given piece of software, we are essentially provided with the instruction set i.e. the instructions in the existing code. [2] show that source code is not that unique, and therefore it is a valuable place to look for repairs. Alternatively, we can use different versions of a program, and transplant code [9].

When dealing with GI, we are given the representation to work with i.e. the space of syntactically correct programs. At this stage of GI research, we do not need to invent new forms of representation, but should investigate existing ones. In contrast, the GP community has invented an array of new program representations [10] though some representations have potentially useful properties e.g. modularity "come for free" [10]. As there are already a large number of existing programming paradigms (imperative, object oriented, functional). It would make sense to investigate how suitable these are as a representation which is amenable to search operators. One hypothesis is that functional languages are more suitable for evolving that imperative languages because of the side effects [13].

Central to most programming languages, is the type system. GP has made use of type in the past [8]. In the GI setting, almost all programs will consist of instructions which operate on different data types, so once again we are force to confront what is part of normal software engineering with our automated methods.

The position of this paper then is that GI will enrich GP research, as GI forces us to use "the full scale of programming languages" including loops, reading and writing to memory, defined functions (macros, procedures, methods). GI is also concerned with potentially large software systems [4], which have previously been out of reach of the traditional synthesis approach taken by GP.

In conclusion, this paper has contrasted GP and GI. Superficially the difference is that GI starts with existing software, where GP attempts to evolve from an empty program. However, the differences are deeper and more interesting than may first appear. GI may solve the initial problem of GP being overly concerned with toy problems.

## 1. ADDITIONAL AUTHORS

## 2. REFERENCES


[1] P. Flach. *Machine Learning: The Art and Science of Algorithms That Make Sense of Data*. Cambridge University Press, New York, NY, USA, 2012.

[2] M. Gabel and Z. Su. A study of the uniqueness of source code. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 147–156, New York, NY, USA, 2010. ACM.

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

[4] W. B. Langdon and M. Harman. Genetically improving 50000 lines of C++. Research Note RN/12/09, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, UK, 19 Sept. 2012.

[5] W. B. Langdon and M. Harman. Optimising existing software with genetic programming. *IEEE Transactions on Evolutionary Computation*, 19(1):118–135, Feb. 2015.

[6] W. B. Langdon and R. Poli. Why "building blocks" don't work on parity problems. Technical Report CSRP-98-17, University of Birmingham, School of Computer Science, 13 July 1998.

[7] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, Jan.-Feb. 2012.

[8] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[9] J. Petke, M. Harman, W. B. Langdon, and W. Weimer. Using genetic improvement and code transplants to specialise a C++ program to a problem class. In M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. Garcia-Sanchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, editors, *17th European Conference on Genetic Programming*, volume 8599 of *LNCS*, pages 137–149, Granada, Spain, 23-25 Apr. 2014. Springer.

[10] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[11] K. Wagstaff. Machine learning that matters. *CoRR*, abs/1206.4656, 2012.

[12] D. R. White, J. McDermott, M. Castelli, L. Manzoni, B. W. Goldman, G. Kronberger, W. Jaskowski, U.-M. O'Reilly, and S. Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, Mar. 2013.

[13] J. Woodward. Evolving Turing complete representations. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003*, pages 830–837, Canberra, 8-12 Dec. 2003. IEEE Press.

[14] J. R. Woodward and R. Bai. Why evolution is not a good paradigm for program induction: a critique of genetic programming. In L. Xu, E. D. Goodman, G. Chen, D. Whitley, and Y. Ding, editors, *GEC '09: Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 593–600, Shanghai, China, June 12-14 2009. ACM.