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

Alexander E.I. Brownlee
University of Stirling
Stirling
Scotland, United Kingdom
sbr@cs.stir.ac.uk

ABSTRACT

Genetic Programming (GP) has been criticized for targeting irrelevant problems [12], and 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 GI provides 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 evolution is the main approach to GI to manipulating programs, this connection with real software should persuade the GP community to confront the issues around what it originally set out to tackle i.e. evolving real software.

Keywords

Genetic Improvement (GI), Genetic Programming (GP)

1. POSITION

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

The GP community has tackled a number of *toy problems* including the even parity problem. With this problem, we need all n input bits to be able to classify an input sequence as 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. Another characteristic of the even parity problem which makes it interesting from a machine learning perspective is that we cannot use techniques such as feature selection methods [1]. Nor is there correlation between input variables [6]. Surely the GP community can be a little bit more ambitious than solving even parity.

There is one fact about program spaces that is different to search spaces typically targeted by metaheuristics. With many problems , the objective function is a direct mapping between the solution space and objective space. With GP, there is an intermediate space between these two spaces. 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 (i.e. the function defined by the sample of test cases). The mapping between functions and error scores is problem dependent, and in fact defines the problem.

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 define functions 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 containing for and $while$ loops, defined functions, and usually have side effects (e.g. writing to file).

GP has examined sorting, along with other short programs. Sorting has also been targeted with GI. While short programs may be interpreted in some senses as *toy problems*, they are of interest when included in larger programs that invoke them many times. Improving a small amount of code which is executed many times can have a large effect on the execution time and energy consumption.

We can classify programs into 4 types, depending on how they are executed. 1) *1-programs*, where all nodes in the syntax tree are executed once (e.g. programs constructed with a function set f_1 {+, -, *, %}). 2) *0-1-programs*, where nodes are either executed once or not (e.g. programs constructed with f_2 containing logical operators {AND, OR, NOT}, where short circuiting is used.) 3) *n-programs*, containing for loops with a determined number of iterations (bounded). 4) ∞-*programs*, with while loops with an undetermined termination condition (potentially unbounded).

Broadly speaking, GP concerns the former two types, while GI the latter two. Adopting a GI approach, which deals with software, forces us to confront programs that 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 f_1 or f_2. With the first two types, programs will execute in a comparatively short amount of time (bound by the size of the program). While with the

WOODSTOCK '97 El Paso, Texas USA

DOI: 10.475/123_4

last two types, programs may take a very long time to terminate (possibly not halting). Hyper-heuristics may include all 4 types, as they usually evolve 1 or $0-1$ programs as the body of a loop in a program [15, 16].

With GP, we have to choose an instruction set, and it is a trial and error process. However, with GI, we are essentially provided with the instruction set i.e. the instructions in the existing program. [2] show that source code is not that unique, and therefore the existing program(s) is a valuable source of code for repairs. Alternatively, we can transplant code from different versions of a program [9].

With GI, we are given the program representation i.e. the space of syntactically correct programs of the language the program is written in. At this early 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 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 concerning search properties of program representations is that imperative languages are less suitable for search than functional languages because of side effects [13].

Central to most programming languages, is the data-type system. GP has made use of types 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 is that GI will enrich GP research, as GI forces us to use "*the full capabilities of programming languages*" including loops, data types, 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 alleviate the issue of GP being overly concerned with toy problems.

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.

[15] J. R. Woodward and J. Swan. Automatically designing selection heuristics. In G. L. Pappa, A. A. Freitas, J. Swan, and J. Woodward, editors, *GECCO 2011 1st workshop on evolutionary computation for designing generic algorithms*, pages 583–590, Dublin, Ireland, 12-16 July 2011. ACM.

[16] J. R. Woodward and J. Swan. The automatic generation of mutation operators for genetic algorithms. In G. L. Pappa, J. Woodward, M. R. Hyde, and J. Swan, editors, *GECCO 2012 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms*, pages 67–74, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.