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 true of the wider machine learning community [11]. 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 it set out to tackle originally i.e. evolving real software.

Keywords

Genetic Improvement (GI), Genetic Programming (GP)

There are a number of impressive examples of GI in the literature including GenProg, which fixed a number of bugs in real software for a $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. We need all input features (all n bits) 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 [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 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.

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 which invoke them many times.

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 \( f_1 \) of arithmetic operators \( \{+,-,\ast,\%\} \)). 2) 0 \(-\) 1 programs, where nodes are either executed once or not (e.g. programs constructed with a function set \( f_2 \) containing logical operators \{AND, OR, NOT\}, where short circuiting is used.) 3) \( n \) programs, containing for loops with a determined number of iterations (bounded time). 4) \( \infty \) programs, with while loops with an unknown termination condition (unbounded time).

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 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 last two types, programs make take an awfully long time to terminate (possibly not halting). Hyper-heuristics sit somewhere between these two 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 make a decision about instruction set, and 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 it is a valuable place to look 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. 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. It would make sense to investigate how amenable to GI search operators these program representations are. One hypothesis 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, 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 initial issue of GP being overly concerned with toy problems.

1. REFERENCES