Information Theory, Fitness, and Sampling Semantics

Colin G. Johnson[1] and John R. Woodward[2]
[1]School of Computing, University of Kent, Canterbury, UK.
C.G.Johnson@kent.ac.uk
[2]Division of Computer Science and Mathematics, University of Stirling, Stirling, UK.
john.woodward@cs.stir.ac.uk

One of the most significant developments in genetic programming (GP) research in the last few years has been a growing focus on the *semantics* of programs. Traditional GP is focused on program text and designed operators and representations around syntactic transformations. For example, a mutation operator will be designed to make a small-but-not-too-small change to the text of the program, with the hope that it would have a similar effect on program behaviour. By contrast, semantic methods are based on the idea that GP should be designed so that operators and representations work on the *meaning* and *behaviour* of the programs represented—i.e. on their input-output mappings. So, for example, a semantic mutation operator would (either by generate-and-test or by design) make a small-but-not-too-small change to that input-output behaviour.

Early work on semantic methods in GP focused on the creation of *semantic diversity*; for example, ensuring that the IO behaviours of programs in the initial population were all different [2], or that crossover and mutation operators did not give rise to child programs that had the same behaviour as the parent programs, or which made a transformation of the behaviour of a certain distance in semantic space [1, 3, 8, 9]. The core of this process is a generate-and-test approach that generates candidate programs and rejects them if some semantic measure is exceeded.

More recently, focus has shifted towards the creation of GP methods that operate on semantics by design. For example, the geometric semantic GP approach [7, 10] redesigns the operators so that, for example, a crossover operator generates a program that is "inbetween" the behaviours of the two parent programs. This is achieved by construction, by forming a child program consisting of the two parent programs plus a third, randomly generated function, in such a way that the outputs of the function is a form of interpolation between the two parent functions, mediated by the third randomly generated one. Naively, this leads to vast programs, but the combination of program simplification [7] and careful use of caching [10] has produced an efficient technique that has been applied successfully both to benchmark toy problems and to real-world applications in, for example, pharmacogenetics [10] and civil engineering [4].

A less well explored area is the connection between semantics and fitness. Here we will explore a promising link between one of the ideas in semantic GP, *sampling semantics*, and the idea of fitness measurement in GP-like systems. This will involve a new kind of fitness-like measurement grounded in ideas around entropy, Kolmogorov complexity [6], compression-based metrics [5] and information gain.

Sampling semantics [8] is one way of formalising the semantics of a program in a GP population. To define a sampling semantics, we begin by choosing a list of inputs $P = [p_0, p_1, \ldots, p_n]$ in the input space of the problem being tackled (typically these coincide with the fitness cases). The corresponding sampling semantics for each program is a set $S = [s_0, s_1, ..., s_n]$ in the output space of the program, where each s_i corresponds to the evaluation of p_i on that program. Thus, for each program, we have a set up input-output pairs on the same set of inputs. These have then been used to define measures that compare the semantics of two programs, for example by measuring the distances between the S vectors for two programs.

This vector can be used in an interesting new way to provide a new way of evaluating the fitness of a program—actually, of a program fragment. The traditional way of evaluating fitness in GP is to measure the number of fitness cases that the program solves, or the aggregate error across the fitness cases. However, this way of measuring fitness rewards solutions that solve particular training cases in an arbitrary and unstructured way, rather than building program fragments that contribute towards the overall solution of the problem. Furthermore, it is often difficult for more complicated problems to be solved using traditional GP, because *none* of the programs in the initial population come anywhere close to solving the problem, and so there is no fitness difference for evolution to work on.

Here we describe a new approach to fitness evaluation. Firstly, this takes populations that consist of *fragments* of programs rather than complete candidate solutions. Secondly, it takes the set of outputs that current population members generate (the sampling semantics of those incomplete programs), compares it with the target set of outputs, and *measures the algorithmic complexity of the program required to transform those inputs into the target outputs*. This measurement will be done using information-theoretic measures which approximate the complexity of that putative bridging program.

The key idea is as follows. As a successful program evaluates, each statement executed puts the program into a state where *less* computation needs to be done to solve the problem. Another way to think about this is that at each step, the length of the optimal program required to transform the current output of the program into the target will be be shorter than at the previous execution step. This measure—the length of the shortest program required to transform one vector of values into

another—is known as the *Kolmogorov complexity* of the transformation [6]. This is not computable, but good approximations can be made, most typically by measuring the *compression length* of the difference between the sequences—that is, how short it becomes when compressed with an algorithm such as *gzip*.

Using this, we can build a new program construction algorithm. We call this TDF_{Comp}[1].

FUNCTION TDF_{Comp}.
Input a problem specification *Prob* consisting of fitness cases, terminal set (just the variable names), and a function set. Declare a set C, called the construction set.
Place all of the variable names in the construction set.
LOOP:
Construct m program fragments by combining two elements of C using the functions from *Prob*.
Evaluate each of the program fragments on the fitness cases to get a sampling semantics S.
Calculate the difference vector between S and the target value on the fitness cases.
Use *gzip* to compress each of these difference vectors.
Assign a fitness to each program fragment equal to the length of the compressed vector.
Add the best (most compressible) fragment to C (the remaining fragments can be thrown away).
UNTIL (problem is solved or timeout)
RETURN current best program.

Let us examine a run of this algorithm on the 4-bit even parity problem (variables notated as $v_0, \ldots, v_3$), the function set consisting of all two-input one-output boolean functions, and $m = 500$; the typical run of TDF_{Comp} is as follows (in a simple experiment, 18/20 runs found a solution of minimal size within 3 iterations):

Iteration 1:
$C = \{v_0, v_1, v_2, v_3\}$
Best fragment is $(\text{XNOR}\, v_3\ v_2)$ with compression value 26

Iteration 2:
$C = \{v_0, v_1, v_2, v_3, (\text{XNOR}\, v_3\ v_2)\}$
Best fragment is $(\text{XOR}\, v_1 (\text{XNOR}\, v_3\ v_2))$ with compression value 24

Iteration 3:
$C = \{v_0, v_1, v_2, v_3, (\text{XNOR}\, v_3\ \ v_2), (\text{XOR}\, v_1 (\text{XNOR}\, v_3\ v_2))\}$
Best fragment is $(\text{XNOR}(\text{XOR}\, v_1 (\text{XNOR}\, v_3\ v_2)) v_0)$ with compression value 23
Problem solved

We are currently experimenting with these methods (and related methods based on information gain ratio) for a wide variety of problems that are currently tackled using GP.

References

[1] Lawrence Beadle and Colin G. Johnson. Semantically driven crossover in genetic programming. In *Proceedings of the 2008 IEEE World Congress on Computational Intelligence*, pages 111–116. IEEE Press, 2008.

[2] Lawrence Beadle and Colin G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, 2009.

[3] Lawrence Beadle and Colin G. Johnson. Semantically driven mutation in genetic programming. In A. Tyrell, editor, *Proceedings of the 2009 IEEE Congress on Evolutionary Computation*, pages 1336–1342. IEEE Press, 2009.

[4] Mauro Castelli, Leonardo Vanneschi, and Sara Silva. Prediction of high performance concrete strength using genetic programming with geometric semantic operators. *Expert Systems and Applications*, 40(17):6856–6862, 2013.

[5] Rudi Cilibrasi and Paul M.B. Vitáynyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[6] Ming Li and Paul M.B. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications*. Springer, 2009. Third Edition.

[7] Alberto Moraglio, Krzystof Krawiec, and Colin G. Johnson. Geometric semantic genetic programming. In *Parallel Problem Solving from Nature, PPSN XII (Part 1)*, pages 21–31. Springer, 2012. Lecture Notes in Computer Science Volume 7831.

[8] Ngyuen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R.I. McKay, and Edgar Galván-Lopez. Semantically-based crossover in genetic programming: Application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[9] Ngyuen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, R.I. McKay, and Dao Ngoc Phong. On the roles of semantic locality of crossover in genetic programming. *Information Sciences*, 235:195–213, 2013.

[10] Leonardo Vanesschi, Mauro Castelli, Luca Manzoni, and Sara Silva. A new implementation of geometric semantic gp and its applications in pharmacogenetics. In *Genetic Programming: Proceedings of the 16th European Conference, EuroGP13*, pages 205–216. Springer, 2013. Lecture Notes in Computer Science Volume 7831.

[1] TDF is "theories destroy facts", a neat summary of scientific progress from biologist Peter Medawar; in this context it refers to the way in which program fragments ("theories") gradually replace and generalise variable values ("facts") as a program executes)