An Efficient Implementation of GSGP using Higher-Order Functions and Memoization

Alberto Moraglio

Abstract

Geometric Semantic Genetic Programming (GSGP) [1] is a novel form of Genetic Programming (GP) that can be interpreted as searching directly the semantic space of programs. This new form of GP is very promising as it induces *always* a simple unimodal fitness landscape for any problem it is applied to, hence it converges to the optimum very quickly. A drawback of GSGP with crossover is the exponential growth of individuals due to the fact that the offspring tree *contains* both parent trees, hence individuals double their size at each generation. Vanneschi et al. [2] have proposed an implementation of GSGP with crossover using a complex pointer-based data structure that prevents the exponential growth by keeping trace of the ancestry of individuals rather than storing them directly. We propose a new implementation of GSGP also based on tracing the ancestry of individuals, that however does not explicitly build and maintain a data structure, but uses higher-order functions and memoization to achieve the same effect, leaving the burden of book-keeping to the compiler. The resulting implementation is fast, elegant and concise. A Python implementation (under 100 lines without comments) is on GitHub at `https://github.com/amoraglio/GSGP`.
SOLUTION REPRESENTATION: We represent solutions using directly functions of the programming language used to program the GSGP system. E.g., in a GSGP to evolve Boolean expressions written in Python, the representation of a Boolean expression is directly a Python (anonymous) function computing that Boolean expression, and not a data structure (e.g., a tree) representing the Boolean expression.
SEARCH OPERATORS: Geometric sematic crossover and mutation can be interpreted as higher-order functions. We implement them directly as such: they do not manipulate data structures representing solutions, but take directly in input (anonymous) parent functions and return (anonymous) offspring functions. The returned offspring function *calls* the parent functions in its definition. In particular, the parent function definitions are *not substituted* in the offspring definition, hence there is *no growth* of the offspring function. The function calls to the parents in the offspring *implicitly build* the data structure that relates offspring to parents all the way down the ancestry without the need to use pointers, manage memory and maintain an archive of past solutions.

FITNESS EVALUATION: Even if individuals do not grow, evaluating them takes exponential time, as querying a function for some input requires calling both its parents on that input that in turn need to call their parents on it and so forth, doubling the number of calls at each generation. The complexity of queries can be reduced from *exponential to constant* time by *memoization* (i.e., caching the output values of a function of previously encountered inputs rather than recomputing them) of all individuals (functions) generated in the course of evolution. This works because each individual caches its outputs on the training examples the first time its fitness is computed, and the fitness of the offspring is computed by calling its parents on the training examples whose outputs are ready available in the cache as they were already encountered when their fitness was computed. This reduces the number of calls needed to compute the fitness of an individual from the size of its ancestry to two i.e., number of parents. Memoization is easily implemented as a higher-order wrapping function (it is a standard library function in Python 3.2).
FINAL SOLUTION: As solutions are represented directly as *compiled* Python functions, displaying them would require decompilation, which is not very practical. In particular, this would be required to extract the final solution. Note also that storing directly solutions during evolution is not feasible as their size grows exponentially (this is the original problem we wanted to solve to obtain an efficient implementation).

The technique we have used to display functions that avoids both decompilation and direct representations of functions during evolution consists of doubling the implicit call structures in individuals, where the second structure implicitly keeps track of how to reconstruct the final genotype of the individual (i.e., its source code) mirroring the first call structure (i.e., its semantics) interpreting subroutine calls as function bodies substitutions (i.e., asking the parents to return their source code to embed in the offspring source code). Then individuals can be asked to display themselves by calling their associated 'source code' function. This can be implemented with minor additions to the code.

Naturally displaying the final solution after evolution takes exponential time as its genotype is exponentially long. However, querying the final solution on unseen values (i.e., to make predictions) takes only linear time thanks to the memoization of individuals.

References

[1] Moraglio, A., Krawiec, K., Johnson, C.: Geometric semantic genetic programming. In: Proc. PPSN XII, Springer (2012) 21–31

[2] Vanneschi, L., Castelli, M., Manzoni, L., Silva, S.: A new implementation of geometric semantic gp and its application to problems in pharmacokinetics. In: EuroGP. (2013) 205–216