Combining Programs to Counter Code Disruption

Cyril Fonlupt
Universite du Littoral
LISIC - BP 719
62228 CALAIS Cedex, France
fonlupt@lisic.univ-littoral.fr

Denis Robilliard
Universite du Littoral
LISIC - BP 719
62228 CALAIS Cedex, France
robilliard@lisic.univ-littoral.fr

ABSTRACT

In usual Genetic Programming (GP) schemes, only the best programs survive from one generation to the next. This implies that useful code, that might be hidden inside introns in low fitness individuals, is often lost. In this paper, we propose a new representation borrowing from Linear GP (LGP), called PhenoGP, where solutions are coded as ordered lists of instruction blocks. The main goal of evolution is then to find the best ordering of the instruction blocks, with possible repetitions. When the fitness remains stalled, ignored instruction blocks, which have a low probability to be useful, are replaced. Experiments show that PhenoGP achieve competitive results against standard LGP.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Genetic Programming

1. INTRODUCTION

Genetic Programming (GP) is a technique aiming at the automatic generation of programs. It was successfully used to solve a wide variety of problems, and it can now be viewed as a mature method as even patents for old and new discovery have been filled. GP is used in fields as different as quantum computing, classification or software engineering.

The most widely used representation scheme in GP is the Tree Based GP where each evolved program is coded as a Lisp-like tree. Several alternatives to this tree representation were devised these last years, one of the most successful being Linear Genetic Programming (LGP) [3], where individuals in the population are sequences of high-level languages instructions. The representation of individuals (i.e. programs) in LGP is usually a bounded-length list of integers. This list is mapped into a sequence of imperative instructions, typically from a 3-register instruction language.

One may deem that standard GP and LGP have an inherent weakness: a program fitness is only based on the whole individual results. This evaluation is done regardless of any combinations that the program code fragments may form with other programs. It is expected that the genetic operators will take care about preserving interesting features, while it is well-known that crossover and mutation have disruptive effects, negatively impacting fitness evolution.

Many researchers have tackled this problem since code disruption may stall the evolutionary process, being able to remove good code notably through genetic drift in limited size populations. Some schemes have been proposed to counter disruption either by using a smarter crossover (Size Fair Crossover [5]) or by changing the way to evaluate the fitness (Parallel Linear Genetic Programming [4]).

In this work, we make the assumption that disruption is inherently present in evolution, and should be dealt with more explicitly. Our main idea is to evolve integer vectors that are mapped to concatenations of short LGP programs (or code blocks). These blocks are not the focus of evolution, but are the primitives from which larger programs can be built by successively picking blocks with possible repetition. This scheme recreates a distance between integer vector genotype and complete program phenotype, which is less emphasized in usual GP and LGP, thus the name PhenoGP. This representation also has the benefit of limiting bloat.

2. PHENO GP PRINCIPLES

PhenoGP (ΦGP) borrows some of its components from LGP. The algorithm works with two pools of data: pool 1 is a conventional set of short LGP programs (or code blocks), while pool 2 is a set of fixed size integer vectors (or lists). A vector in pool 2 is a ΦGP program where each integer indexes one of the LGP programs in pool 1 (with 0 serving as a *do not care* symbol). The phenotype of the ΦGP solution is the concatenation of these LGP programs in sequence, with possible repetitions of some code blocks, as illustrated in Figure 1. A code block in pool 1 may be executed zero, once, twice or as many times according to the evolution process that affects pool 2. Evolution focuses on pool 2, exploring combinations of code blocks. When the fitness stagnates, we trigger an evolution round on pool 1 with strong elitism, i.e. the code blocks that are components of good individuals in pool 2 are left unchanged.

The ΦGP scheme may look similar to other works about teams of LGP programs such as [2, 6, 4]. It is different since we do not extract one answer from a member of the team, nor do we combine linearly their answers, keeping a low complexity component from the whole team. In ΦGP we construct a composition of functions which will most often

.

Code block 1	Code block 2
`r[0] = r[1] + 1.27` `r[1] = r[0] + r[1]`	`r[1] = r[1] / 0.5` `r[0] = r[1] * r[1]`

Pool 1: LGP code blocks

$$\begin{cases} \text{Program 1}: & <1,1,2,0,0> \\ \text{Program 2}: & <0,2,2,2,0> \\ \text{Program 3}: & <1,1,1,0,2> \end{cases}$$

Pool 2: ΦGP programs (block execution lists)

```
r[0]  =  r[1] + 1.27
r[1]  =  r[0] + r[1]
r[0]  =  r[1] + 1.27
r[1]  =  r[0] + r[1]
r[1]  =  r[1] / 0.5
r[0]  =  r[1] * r[1]
```

Phenotype of ΦGP program 1

Figure 1: Basics of PhenoGP representation

be non linear. The goal is to build up *higher* complexity from simple components. Thus one cannot expect previous works results to be straightly applicable to ΦGP.

3. EXPERIMENTS

In order to validate our scheme, a set of 6 symbolic regression problems were conducted, where the goal was to find a symbolic expression that associates input and output on a given set of training pairs. In our case, 20 evenly distributed data points x_k in the range $[-1.0, +1.0]$ are chosen as inputs, the expected outputs being given by the following test functions:

$$f_1 = x^3 + x^2 + x$$
$$f_2 = x^4 + x^3 + x^2 + x$$
$$f_3 = x^5 + x^4 + x^3 + x^2 + x$$
$$f_4 = x^4 - 2x^3 + x$$
$$f_5 = \pi \quad \text{(constant function)}$$
$$f_6 = \frac{x}{\pi} + \frac{x}{\pi} + 2x\pi$$

For all problems, the set of operators available for evolution is $\{+, -, \times, \div\}$ and the fitness is the sum of absolute deviations over all pairs. Pool 1 code blocks have a length of 12 instructions, while pool 2 contains size 10 vectors. We report the best fitness averaged over 40 independent runs, with $50,000$ evaluations per run, and also a ratio of *hit* solutions with fitness lower than $1e^{-3}$.

Results are shown in Table 1: on problem f_1 there is no significant difference between both heuristics, on f_2 and f_3 LGP is a clear leader, while ΦGP delivers the best solutions for the last 3 problems (f_4 to f_6).

It is interesting to notice that ΦGP seems abler at finding non trivial constant values, producing hit solutions for both f_5 and f_6. In DoZh11 it was shown that short LGP programs do not suffer from code disruption as much as larger programs do. This remark might explain the difference in ΦGP and LGP behavior: producing the π constant adds to the complexity of solutions, thus hampering standard LGP. On the opposite, the first four regression problems can be solved using short programs, thus it hints at the difficulty for ΦGP to find short combinations of code blocks.

Table 1: ΦGP versus LGP for regression

	ΦGP		standard LGP	
Problem	Fitness	% hits	Fitness	% hits
f_1	0.04	82.5%	0.002	98%
f_2	0.15	45%	0.0	100%
f_3	0.28	22.5%	0.02	93%
f_4	0.12	30%	0.33	23%
f_5	0.017	47.5%	0.07	0%
f_6	0.177	2.5%	0.21	0%

Table 2: ΦGP versus LGP for Hanoi

	PGP		standard GP	
#	Fitness	% hits	Fitness	% hits
	0.81	17.5%	1.18	15%

We also experimented with the famous towers of Hanoï benchmark with 3 pegs and 4 disks. The primitive instructions are those from [1], with a limit of 32 moves per program and 40 run of 50000 evaluations as before.

The results are shown in 2, which confirm that ΦGP is worth exploring further.

4. CONCLUSION AND FUTURE WORKS

The goal of this work was to investigate a GP scheme designed explicitly to counter code disruption. This ΦGP scheme is compared to the standard LGP algorithm on several regression problems and on the towers of Hanoï.

These first results are encouraging. ΦGP seems competitive against LGP for the more complex problems, requiring longer solutions: i.e. it seems able to counter disruption. On the opposite ΦGP is deceiving on the simpler problems, when disruption does not impede standard LGP. This problem could be addressed in future works by allowing easier construction of short code blocks combinations.

5. REFERENCES

[1] P. J. Angeline and J. B. Pollack. Coevolving high-level representations. Technical Report Technical report 92-PA-COEVOLVE, Laboratory for Artificial Intelligence. The Ohio State University, 1993.

[2] M. Brameier and W. Banzhaf. Evolving teams of predictors with linear genetic programming. *Genetic Programming and Evolvable Machines*, 2(4):381–407, 2001.

[3] M. Brameier and W. Banzhaf. *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, 2007.

[4] C. Downey and M. Zhang. Parallel linear genetic programming. In *Proceedings of the 14th European conference on Genetic programming*, LLNCS, pages 178–189. Springer, 2011.

[5] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1092–1097. Morgan-Kaufmann, 1999.

[6] P. Lichodzijewski and M. I. Heywood. Managing team-based problem solving with symbiotic bid-based genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 363–370. Morgan Kaufmann, 2008.