Using Linear Genetic Programming to Evolve a Controller for the game *2048*

T. Chabin, M. Elouafi, P. Carvalho, A. Tonda

June 30, 2015

Abstract

This short paper describes the method used to evolve a controller for the game *2048* for the GECCO 2015 competition. A generic Linear Genetic Programming open-source tool is used to directly write a Java class, that is subsequently corrected, compiled inside the provided framework, and run to obtain a fitness value.

1 μGP

The evolutionary core used in this experience is μGP (or MicroGP) [4], a generic open-source evolutionary tool hosted on SourceForge[1]. As individuals in this framework are internally encoded as directed multi-graphs, μGP technically belongs to the family of Linear Genetic Programming. The software is chosen for several desirable features: self-adapting parameters; customizable structure of the individual genome, specified by an XML file; design that makes it possible to use parallel, external fitness evaluation.

Thanks to the user-customizable and richly expressive structure of the genome, μGP is successfully used in different applications concerning automatic code generation, ranging from the design of AIs in the game *Core Wars*[2] [2], to the generation of Assembly-language code for processor's stress tests [1], to the generation of C++ classes for AIs in StarCraft [3].

[1] `http://ugp3.sourceforge.net/`

[2] For more information, see `http://www.corewars.org/`

2 Proposed Approach

The approach used in the competition relies upon exploiting μGP to directly write valid Java classes for the controller, modifying the code with a script to take into account some possible issues, compiling the generated class in the provided framework, and finally executing the resulting `jar` to obtain the fitness values.

2.1 Genome structure

In previous applications of μGP, the presence of *jump*-like structures in the target language (such as `goto`s in C/C++ and `jmp` in Assembly) proved to be particularly beneficial to the evolutionary algorithm. In Java, by design, there is no such structure: in order to obtain a similar behavior, the main part of the genome describes a `for` loop that contains a `switch...case` statement, with a non-fixed number of `case`s. The variable that drives the loop is also used in the `switch`, and its value can be changed by the statements themselves. This rather complex structure factually behaves like a series of `goto`s, leaving the evolutionary algorithm free to branch over multiple conditions. An example of the generated code is reported on [3].

A generated individual consists of a fixed block code called "prologue", followed by a generated series of macros, where each macro is a `case` in the `switch`. Those macros are the bases of the program's genome. A macro is an `if-else` statement, where the condition and the different possible results are evolved. The condition is a

[3] `https://github.com/tchabin/treecko/blob/master/BestAgent.java`

boolean function chosen by μGP among predefined functions. μGP also evolves the parameters used in this function (value of a specific tile, arbitrary possible tile's value,...). The possible results of this `if-else` statement are: adding an evolved value to a score attributed to every possible action, make a jump to another `case`, or return the best action according to the score. Finally another fixed block of code, called "epilogue" is put at the end of the generated individual.

2.2 Fitness Function

Two ideas sprang to mind: using just the mean score itself, and using the percentages of games in which the controller achieved a given tile in a vector's form. After some deliberation, we decided to test many different configurations, and ended up concluding that the fitness function that seemed to generate the best individuals was one that took just the mean score.

It is our belief, based on bibliographical research, that a fitness function that also took the mean number of empty tiles per move into consideration would be more suitable, but since that information was not given us by the simulator, we thought it best not to modify it for that purpose.

3 Experimental results

The experiments are run until a stagnation condition is reached. The settings for μGP are reported in Table 1.

In μGP, every time a mutation is applied on an individual, it is executed again, unless a randomly generated number in $(0, 1)$ is higher than the current value of σ: this parameter is self-adapted during the evolution, trying to find a good balance between exploration (many mutations) and exploitation (few mutations). Self-adapting of parameters is regulated by α, following the formula:

Table 1: Parameter settings used in the experiments.

Parameter	Meaning	Value
μ	Population size	80
λ	Number of genetic operators applied at each generation	64
σ	Strength of genetic operators	0.9
α	Inertia of the self-adapting engine	0.9
SSC	Maximum number of generations without improvements of the best fitness value, after which the algorithm will stop	50

$$\texttt{new\_value} = \alpha \cdot \texttt{old\_value} + (1-\alpha) \cdot \texttt{target\_value} \quad (1)$$

The activation probability of the genetic operator is self-adapted.

After 19h of computation using 16 cores of Intel(R) Xeon(R) CPU E7-4830, We obtained a mean score of 5599.8

References

[1] Fulvio Corno, Gianluca Cumani, M Sonza Reorda, and Giovanni Squillero. Fully automatic test program generation for microprocessor cores. In *Design, Automation and Test in Europe Conference and Exhibition, 2003*, pages 1006–1011. IEEE, 2003.

[2] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. Evolving assembly programs: how games help microprocessor validation. *Evolutionary Computation, IEEE Transactions on*, 9(6):695–706, 2005.

[3] Pablo Garcia Sanchez, Alberto Tonda, Antonio Mora, Giovanni Squillero, and Juan Julian Merelo Guervos. Towards Automatic StarCraft Strategy Generation Using Genetic Programming. In *Proceedings of the 2015 Congress on Evolutionary Computation (CEC)*, 2015 (to appear).

[4] Giovanni Squillero. MicroGP - an evolutionary assembly program generator. *Genetic Programming and Evolvable Machines*, 6(3):247–263, 2005.