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. 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], to the generation of Assembly-language code for processor’s stress tests [1], to the generation of C++ classes for AIs in StarCraft [3].

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 gotos 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 cases. 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 gotos, 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

---

1http://ugp3.sourceforge.net/

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

3https://github.com/tchabin/treecko/blob/master/BestAgent.java
boolean function chosen by $\mu$GP among predefined functions. $\mu$GP 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 $\mu$GP are reported in Table 1.

In $\mu$GP, 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 $\sigma$: 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 $\alpha$, following the formula:

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

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>Population size</td>
<td>80</td>
</tr>
<tr>
<td>$\lambda$</td>
<td>Number of genetic operators applied at each generation</td>
<td>64</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>Strength of genetic operators</td>
<td>0.9</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Inertia of the self-adapting engine</td>
<td>0.9</td>
</tr>
<tr>
<td>SSC</td>
<td>Maximum number of generations without improvements of the best fitness value, after which the algorithm will stop</td>
<td>50</td>
</tr>
</tbody>
</table>

References


