CS1537 Spring 2011

Project 4: Genetic Programming
Due: April 19th, midnight

As you know from Segaran, Chapter 11, genetic programming (GP) is a technique for automatically building algorithms using evolutionary principles similar to those used in genetic algorithms. For this last project, we will implement GP and experiment with several extensions. Specifically, we will use genetic programming to evolve programs for playing the Tron "lightcycle" game. The group who evolves the best algorithm for playing Tron will receive extra credit.

1 Tron Game

The Tron game involves two or more players driving "lightcycles" around a 2d game grid. Each player leaves a "light wall" behind himself. The object of the game is to avoid crashing into a "light wall", the boundaries of the grid, or other players, while simultaneously trying to make other players crash. If you haven't played it before, there are Flash versions of the game online at `http://www.albinoblacksheep.com/games/fltron` and `http://fltron.com/`.

We will use a slightly simplified version of this game. Each game will consist of two players. We will use a 15x16 game grid in which both players start on opposite sides of the grid from each other. We will omit the turbo boost in the online version and instead assume that each player always moves at a constant speed. Only left, right, up, and down moves will be allowed.

2 Baseline GP Implementation

To help with the evolution of our game playing algorithms, we will use a modified version of the TinyGP [1] genetic programming implementation. This implementation represents programs as arrays of characters, where each character represents a variable, constant, or function[2]. Each program is a flattened, linear tree using bracket-less prefix notation; location of brackets is automatically inferred by the arity (number of parameters) of each function. In addition to a modified version of TinyGP, I will also provide Java code that will act as a framework for the Tron game.

We will break up each game into discrete steps, where each step corresponds to each player moving one unit on the game grid. At each step the player algorithms will be executed. The algorithms will take as input the current board state, and will return the next move that the player should make. Since the algorithm could return any floating-point value, we will take the mod 4 of the rounded output in order to determine which of the four possible moves (up, down, left, right) to perform.

For the baseline implementation, you need only get the code running on your machine and read through the code, familiarizing yourself with its operation. Also, please modify the code so

[1] `http://cswww.essex.ac.uk/staff/rpoli/TinyGP/`

[2] since a char in Java is represented by a single byte, note that this representation limits us to a total of 256 variables, operators, and constants per program

that timing information is collected for the GP. Note that the program automatically saves the best program it generates to a file. Please make sure to hold on to the best program generated by the baseline implementation. Also, take a look at the "Field Guide to Genetic Programming"[1], available online at `http://cswww.essex.ac.uk/staff/rpoli/gp-field-guide/toc.html`. You will probably need this resource to implement some of the GP extensions in the following section.

3 GP Extensions

Since most of the baseline GP has been provided to you, the focus of this project will be on extending the GP with additional features. Please extend the GP implementation in at least four interesting ways. Several sample ideas for extension are given in the list below. Feel free to come up with your own ideas for extension, drawing on Segaran, [1], and other sources for inspiration.

- The initial population is generated in a very simple manner; all initial programs have the same initial tree depth. This kind of initialization is referred to as the *full* method. Implement the *ramped half-and-half* initialization method discussed in Section 2.2 of [1].
- The baseline GP implements very few functions. Please implement at least five additional functions, such as *if-then-else*, *while-loop*, *random* number generator, etc., that may be useful in this problem domain. Note that not all functions will necessarily have the same arity. You will need to take this into account, modifying the code accordingly (specifically, the "run()", "traverse()", and "grow()" methods, and possibly others).
- The baseline GP performs crossover by selecting crossover points uniformly at random. For large trees the number of leaf (terminal) nodes can outnumber the number of function nodes, leading to programs frequently only swapping leaf nodes. To counteract this, implement the 90% function / 10% leaf swapping discussed in Section 2.4 of [1]. Also, please implement subtree mutation, discussed in the same section.
- We currently have no empirical fitness function. The baseline GP selects programs for crossover based only on which programs win in the tournaments, but provides no numeric assessment of their quality. Come up with a fitness function that objectively assesses the fitness of each individual program. One method for implementing a fitness function is to define a list of variable values and the optimal output. In this case, we would need to define board positions and an ideal action, which may be difficult. (see `http://cswww.essex.ac.uk/staff/rpoli/TinyGP/index.html#163` for an example). Another method would be to define a metric which reflects the "goodness" of a particular board configuration.
- Implementing a fitness function (previous bullet point) opens up a number of additional improvements that can be made to the GP. If you choose to implement a fitness function, then an additional extension could involve the implementation of a fitness proportional (ie, "roulette-wheel") selection mechanism.
- Come up with a better way to represent the board state. Perhaps you can think of a better way to convey board state to the GP such that the GP evolves better solutions.

Please explain all of your extensions in the report. Also, collect timing information for your fully modified GP. Feel free to come up with your own extensions to the basic GP. Please ensure that any custom extensions are comparable in complexity to the extensions listed here. If you are uncertain as to whether your custom extensions are sufficiently complex, feel free to ask.

Finally, please run the best program evolved by your extended GP with the best program evolved by the baseline GP in a competition of 100 rounds. The two programs should alternate between controlling player 1 and player 2. What is the final score?

4 Submission and Grading

- Downloading, running, and understanding the fundamental GP: 10%
- Extensions (at least 4): 65%
- Report: 25%
- Extra Credit (more than 4 extensions, evolving the best Tron player): up to 10%

Please submit all of your source code, a readme which clearly explains how to execute your code, the best program your extended GP evolved, and your final report. These materials should be packaged inside a compressed archive and emailed to `alexander.p.conrad@gmail.com` before the deadline. Please include the names of the group members in the name of the archive. This project is due by the end of the day on April 19th (or in other words, before I wake up on the 20th).

References

[1] R. Poli, W. B. Langdon, and N. F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).