CPSC 481 — AI —Project 2b, GP Arithmetic Critters

Project 2 — GP Arithmetic Critters
Introduction

To recap, this alternative Genetic Program project involves a population of critters, each of which is a Lisp arithmetic expression. It should work the same was as the prior GP Blocks project, but with a much simpler closure effort.

Closure

The closure property means that the kids are well-formed Lisp expressions, because if they are not then trying to run (evaluate) such an expression will cause Lisp to crash on you.

For closure, the allowable operators are +, -, *, and div1. The new div1 function is exactly like '/' division except it checks for divide by zero and returns 1 in that case. (The normal division by 0 will crash your program, dropping you into the Lisp debugger.)

The allowable arguments are the variables x, y, z, and the integers from -9 to +9, including zero. Note that each variable can be included more than once as an argument to an operator (e.g., (* -8 x x)); however, the "golden goal" critter (the "Midas bug") you are trying to get close to is represented by a ratio of two quadratic polynomials, so no multiplication (or division) operator will have more than two variable symbols as arguments (e.g., nothing like this quintic term (* 4 x x y z z)).

GP Framework

Other than replacing the stacker expressions with arithmetic expressions, the Genetic Programming framework remains the same. Randomly generate N (N=100) critters (arithmetic expressions).

Fitness

Each critter is run against the test samples. The closer the critter's results match the Sample outputs, the more fit the critter is. (We recommend summing the absolute deltas; without absolutes, some large negative deltas may cancel large positive deltas, and we want all deltas to approach zero.) Two things. Firstly, the critter's result on a test sample may (likely) be a float, not an integer. Don't be surprised. Secondly, you will need to set up the test sample's x,y,z values so that when you evaluate/run the critter, it "sees" those values. The easiest way is to set those values as globals. Once the globals are set, then run "eval" on a critter. For example, if you have a list of critters (`crits`), you could do one of the following once you've set up the test sample's x,y,z values:

```
1. (eval (car crits)) // Get result for critter #0.
2. (eval (nth 0 crits)) // Get result for critter #0.
3. (mapcar #'eval crits) // Get list of results, one for each critter.
```

If your chosen Lisp dialect has problems with the symbols x,y,z being global, you could rename them (e.g., gx, gy, gz). The alternative fancy way around such a problem is to wrap each critter in lambda form which takes 3 arguments: x,y,z.

Survivor Pool

Use the fittnesses to select half the population as survivors for mating and purge the other half. (If you like, you can run a "lottery" to randomly replace a few bottom survivors with purgees.) This represents a mild attempt to avoid losing potentially useful "genetic" material.

Kids

Then randomly draw 2 survivors from the "pool" and mate them. For cross-over, randomly pick a sub-tree in each mom and build the kids with those sub-trees swapped. (If you want, you can flip a coin and swap only the sub-tree's main (first) operators instead.) This builds a set of N/2 kids to add to the N/2 survivors to give you N critters, again, for the next generation.

Brazil Nuts on a Banana Split in Denver

At this point, expose them all to (mild) high-altitude and/or food-based radiation mutation: randomly select about 1% of the operators throughout the entire population and swap each of them with a random operator. (It doesn't have to be different.) It doesn't have to be different, but for closure you must swap operator with only operator, and literal arguments with only literal arguments.

Also, you can choose to remove a literal argument, or append a new extra literal argument (to some sub-list). If you remove a literal argument, make sure you leave at least two arguments in its list, because all the arithmetic operators should have at least two arguments.

Generations

Run for 100 generations. Save the two best and worst starting (random) critters and ditto for the last two generations, plus their fitness scores for your report summary. Also, a nice chart of fitness over generational time would be nice, showing the best, worst and average fitnesses of each generation. (For the graphics chart, check out MS Excell, or the freeware Open Office's spreadsheet, etc.)

Test Samples

Here are the Test Samples in the format (x y z output). Note that the output can get up to three digits. And a critter's output could get much higher.

```
(1 0 2 2)
(0 -2 1 -16)
(9 8 -6 72)
(9 -7 5 113)
(-8 7 3 150)
(-4 -5 -3 58)
(5 4 -5 20)
(6 -4 6 41)
(-5 3 -7 -24)
(-6 -5 9 -18)
```

Readme File (unchanged)

You should provide a README.txt text file that includes the class and section, your (team) name, the project/program name, instructions for building, instructions for use, any extra features, and any known bugs to avoid. Be clear in your instruction on how to build and use the project by providing instructions a novice programmer would understand. If there are any external dependencies for building, the README must also list them and how to find and incorporate them. Usage should include an example invocation.

A README would cover the following:

- Program name
  - Your Name (authors, team)
  - Contact info (email)
  - Class number, Section (eg, 01),
- “Project Part” and its number
- Intro (see the Introduction section, above)
- External Requirements

- Build, Installation, and Setup
- Usage
- Extra Features
- Bugs

Academic Rules

Correctly and properly attribute all third party material and references, if any, lest points be taken off.

Submission (unchanged)

Your submission must, at a minimum, include a plain ASCII text file called **`README.txt`** (e.g., title, contact info (of all team members), files list, installation/run info, bugs remaining, features added) all necessary source files to allow the submission to be built and run independently by the instructor. [For this project, no unusual files are expected.] Note, the instructor doesn't necessarily use your IDE or O.S.

All source code files must include a comment header identifying the author, author's contact info (please, no phone numbers), and a brief description of the file.

Do not include any IDE-specific files, object files, binary **executables**, or other superfluous files.

Place your submission files in a **folder named** `X-pY_lastname-firstinitial`. (If working in a team, use the team's 3-letter acronym: `X-pY_RAK`, for team RAK.) Where X is the class course number (e.g., 123 for course CS-123) and Y is the project part number (eg, 9 for Project part #9) For example in CS-123 for Project #9, if your name were Tom Cruise, then you should use this:

```
123-p9_Cruise-T
```

Then zip up this folder. Name the .zip file the **same as the folder name**.
Turn in by 11pm on the due date (in the bulletin-board post) by **sending me email** (see the Syllabus for the correct email address) with the zip file attached. The email subject title should also include **the folder name**. [NB, If your emailer will not email a .zip file, then change the file extension from .zip to .zap, attach that, and tell me so in the email.] Please include your name and campus ID (for each team member) at the end of the email (because some email addresses don't make this clear). If there is a problem with your project, don't put it in the email body – put it in the README.txt file. Do not provide a link to Dropbox, Gdrive, or other cloud storage.

Grading (unchanged)

- 65% for compiling and executing with no errors or warnings
- 20% for clean and well-documented code
- 10% for a clean and reasonable **`README`** file
- 5% for successfully following Submission rules