CS 481 — AI — Project 2, GP Arithmetic Exprs

Project 2 — GP Arithmetic Exprs

Introduction

This is a GP (Genetic Programming) problem. As you know, GP is a variant of the GA (Genetic Algorithm) family, except that 1) it works on programs (each usually represented as a Tree) rather than on DNA gene strings (usually represented as bit strings), and 2) to evaluate an individual's fitness we will "run" the program tree to get a value, the fitness value. We will be using GP to find a good approximation to a goal: an unknown quadratic arithmetic expression in several variables. (If you don't remember what such a thing is, not to worry: the GP program will tell you.)

To simplify things, each individual of our population is an arithmetic expression (an "expr"). An expr can be composed of its "alphabet" of operators, variable names, and constant integers. See Luger, section 12.2.2 for more information.

Teams

This is a team of up-to-four project. For teams, establish tasks and each day or so discuss the Agile 3Qs: 1) What was completed yesterday?, 2) What is planned to complete today?, and 3) What obstacles are in the way? Break up a task so that some small part can be completed each day. If you expect to have down days (i.e., no-work days), tell your team up front.

Alphabet of Operators and Operands

Constant integer operands from **-9** to **+9**, including **zero**.
Variable name operands are **x**, **y**, or **z**. (These will be global variables in your program.)
Operators are **+**, **-**, and *****.

Expressions

An arithmetic expression is a tree of operators (internal nodes), and each leaf node is either a literal constant operand or a variable name operand.

As the target (or goal) expression is at most quadratic in 3 variables (hence, a 3-dimensional quadratic surface) you don't need a single term with more than two variable names in it. Thus, a term like (* 4 x x z) is not needed, but you can include these anyway as they will likely “die out”.

Here are some example exprs:

```
1. (* x)
2. (- (* 3 x 4) (+ y (* 8 z) (* x (+ 5 -8 z y))))
3. (+ (* 1 x x) 2 0 (* -3 x y z))
4. (* (* 4 (- x -7) (* 2)) 3 y)
```

Closure

The closure property means that the kids (GP-generated child exprs) are well-formed Lisp expressions. If they are not, then trying to run (evaluate) such an expression will cause Lisp to dump you into the debugger – not your favorite place.

So, the first element of any list or sub-list in an expr must be an operator, and an operator should only appear as the starting element of a sub-list. Don't create exprs like these:

```
1. (x * 3)  ;; x is in the operator position, wrong.
2. (+ (* 1 + x) 2 - (* 3 x y *)) ;; a +, *, and ,- are in operand positions.
```

Oddly, Lisp is rather flexible. You can have +, -, or * operators with only zero or one arguments, and they still work. (+) == (-) == 0, and (*) == 1.

CS 481 — AI — Project 2, GP Arithmetic Exprs

GP Framework

Randomly generate N (N=50) exprs, and away you go. Your GP program will look something like this – inside a function, of course – and this is too big, so should be broken into sub-functions, too:

```
(progn
  (do until current pool filled ;; init population
      create new expr
      add expr to current pool )
  (do until exceed terminal generation count
      bump generation count
      (for each expr in current pool
           calc fitness for expr )
      save a copy of most fit expr for this generation
      (do until no more exprs in current pool
          select 2 exprs as parents
          remove parents from current pool
          select crossover point/node in each parent
          make crossed kids
          expose each kid to mutation
          add kids to next pool )
      current pool = next pool ))
```

Fitness

Each expr is run against a set of test samples. The closer the expr's results match the Sample outputs, the more fit the expr 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.)

Note, you will need to set up the test sample's x,y,z values so that when you evaluate/run the expr, it "sees" those values. The easiest way is to set those values as globals. Once the globals are set, then run "eval" on a expr.

For example, if you have a list of exprs, you could do one of the following once you've set up the test sample's x,y,z values:

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

An alternative way is to use a function, like the following, to set up local variables in which to evaluate your expr:

```
(defun run_expr (rvars rexpr)
  "Evaluate expr using vars."
  (let ((x (car rvars))
        (y (nth 1 rvars))
        (z ( nth 2 rvars)))
    (eval rexpr)))
```

```
;; tests
(run_expr '(2 0 0) '(* x))
2
(run_expr '(2 0 0) '(* 3 x 4))
24
(run_expr '(2 0 0) ' (+ y (* 8 z) (* x (+ 5 -8 z y))))
-6
(run_expr '(2 0 0) '(- (* 3 x 4) (+ y (* 8 z) (* x (+ 5 -8 z y)))))
30
```

Crossover Point Selection

You can select a parent expr's crossover point at random. Each node should have a chance at being the point, both operator (internal) nodes and non-operator (leaf) nodes.

One simple method is to get the length, L, of a list and then then pick a random number from 0 to L-1, using the mod operator '%' like this:

```
      (% (random) (length expr))
```

Given a picked list element, if it is itself a sub-list, then you could give the nodes in the sub-list a chance to get picked, by flipping a coin, (% (random) 2), and if heads (1) comes up you could then pick an element of the sub-list as the cross-over point in place of the parent list element.

An alternative way is to count the nodes in the expr (via treewalk), pick a random number for that count, and then visit that node (again via a treewalk). Here is a sample treewalk.

```
(defun ptree (rtree)                                    ;; tests
  "Print tree, pre-order."                              (ptree 'a)
  (cond                                                 a1
   ((listp rtree) ;; Is internal node?                  (ptree '(a b c))
    (let ((sum 0))                                      (abc)6
      (princ "(")                                       (ptree '(a (b) c))
      (setq sum (apply #'+                              (a(b)c)7
                       ;; Get list, all elts' results.
                       (mapcar #'ptree rtree)))
      (princ ")")
      (+ sum (length rtree))))
   (t ;; It is a leaf node.
    (prin1 rtree)
    1)))
```

Once you find the crossover point, you need to remember that point. If it is an internal node, then it is itself a sub-list (i.e., a "cons" cell), and you can save it in a local variable and later compare for it via the "eq" function (which tests if two arguments have the same memory address).

However, if it is a leaf node, then it will be an op, an int or a var. Here, you will want to save the location of the leaf in its parent's list, and also save the parent node (cons cell). This is because one 'x' looks like any other 'x', and ditto for a '5' or a '*'. Thus, you will check if you at the crossover point's parent list via an "eq", and then count elements till you get to the point.

Remember, for closure, you must ensure that the start of a list is an op.

Crossed Kid

To make a crossed kid, you need do a treewalk to copy every element from the first parent, but when you get to its crossover point, you need to switch and copy from the other parent's crossover point. Once the second parent's crossover point sub-tree is copied, you need to switch back to the first parent again.

You can use a function like copy-tree to get a duplicate of the parent's expr tree, but that means the cross-over has still to be done. If you want to get a bit fancy (and dangerous) you can try surgery to do the replacement on the copy: using the rplaca and rplacd functions. (If you find a function on the web to do this “copy plus replacement”, that is also okay, but cite it.)

Finally, note that you may want to practice surgery anyway as that is an easy way to do mutation.

Brazil Nuts on a Banana Split in Denver – Mutation

At this point, expose all the nodes to (mild) high-altitude and/or food-based radiation mutation. Look at each leaf node and randomly decide, with about 1-5% probability, to replace it with a random replacement leaf. You might replace an op with an op. You might add a non-op at some point in a list thus extending the list. You might remove a non-op. You might replace a non-op with a different non-op. You might replace a non-op with a sub-list headed by an op.

Generations

Run for 50 generations. Print out the best expr for each generation along with its score. Also, provide a nice chart of fitness over generational time, showing the best and the worst and the average fitness values for each generation. (For the graphics chart, check out MS Excel, or the freeware Open Office's spreadsheet, etc. -- you'll need to capture the raw data from your GP program.)

Test Samples

Here are the Test Samples in the format (x y z correct_output). These are 3D points along with the goal function's value at that point (which is an integer for this function). Note that the output can get up to three digits. And a expr's output could get much higher.

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

Readme File

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. Usage should include an example invocation.

A README would cover the following:

- •Program name
- •Your Name (authors, team)
- •Contact info (email)
- •Class number (481)
- •Intro (see the Introduction section, above)
- •External Requirements (eg, “none”)
- •Build, Installation, and Setup (eg, Lisp type)
- •Usage
- •Extra Features (if any, none needed)
- •Bugs

Academic Rules

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

Submission

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** `481-p2_XYZ` (for team named XYZ).

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

- •75% for "compiling" and executing with no errors or warnings
- •10% for clean and well-documented code
- •10% for a clean and reasonable README file
- •5% for successfully following Submission rules