CS4341
Assignment #2: genetic algorithms
Due date: 9/19/16 @ 11:59 p.m.

For this assignment, you will continue working on the same search for arithmetic expressions as you did in assignment #1. Except as noted below, the input file format are identical as for assignment #1. For this assignment, you will use genetic algorithms to find sequences of operators that reach the desired goal.

Note: this assignment has a moderate amount of analysis of the performance of your approach, and changing how it is written to improve performance. You will need to budget time after the code is completed to finish this assignment.

Reading in the search problem

The first line of the file now supports a new search type: genetic. For example:

```
genetic
4
11
2.5
+ 3
- 1
/ 2.5
* 5
^ 0.38
```

Would indicate that your program should perform a search using genetic algorithms, start with the value of 4, and its goal is to reach a value of 11 within 2.5 seconds. Operators it is permitted to apply are adding 3 to the current value, subtracting 1, dividing by 2.5, multiplying by 5, or taking the 0.38th root.

What your program should output

Your program should output the operators needed to reach the answer (or get as close as possible) in the following format:

```
4 ^ 2 = 16
16 / 2 = 8
8 + 3 = 11

Error: 0
Size of organism: 3
Search required: 0.3 seconds
Population size: 150
```

Number of generations: 216

Output should be displayed on the console, and follow the format shown in the example. It is in your interest to make life easy on the grader. (the example is meant to be illustrative; the exact numbers provided are probably not correct so don't worry if your program has different ones)

Implementing your genetic algorithm

You must follow the *sketch* presented in Figure 4.6 from the text. In other words, there must be an initial population, fitness function, and selection, crossover, and mutation steps. However, there are many ways to instantiate those aspects beyond what is given in the Figure. You do not have to use their method of calculating probability of selection, random cutpoints, etc. There are better approaches out there than those provided in this example. Twenty or thirty minutes spent reading may give you some good ideas how to attack this problem.

You must make use of culling and elitism.

Other approaches you learn about using genetic algorithms are fair to include as well.

Your genetic algorithm should minimize the same cost function as used in assignment #1: absolute error between your solution and the goal.

Relaxing the assumptions from assignment #1

Unlike in assignment #1, floating point results are acceptable and must be represented as such. I.e., 13/7 = 1.857143... In addition, operators can take on floating point values as well. For example, the operator "^ 0.38" is now acceptable.

This relaxation makes the search problem a bit more complex, and makes it less likely that an exact solution can be found. There may be multiple local optima that get very close to the goal, and it can be difficult to tell if a particular solution is good or bad. All these are characteristic of real problems.

Writeup

A large part of this assignment will be the writeup where you describe what you did and why you made your design decisions. Most design decisions should be supported by some experimentation. We are not looking for experimental results and statistical analysis that could appear in an international conference. Something along the lines of "for our suite of 5 test problems, here are graphs of how population size impacted performance. Overall, performance looked best at around 250…" is fine.

Working incrementally is fine. For example, you could first experiment to find the best approach for selection, then test 2 crossover approaches, and then focus on mutation and population size. It is possible that changing approaches for mutation will change which crossover would have worked best. Don't worry about it. Exploring all combinations will take too much time. The goal is to get you used to doing some experimentation.

Details about your approach

- What representation did you choose?
- How did you create your initial population? How large was your population?
- How did you select organisms?
- How did you implement crossover?
- How did you implement mutation?
- How did you make use of culling/elitism? Did they help?

Experiments

Create a moderately complex test problem, where moderately complex is defined as your program does not typically find an answer within 10 seconds. Perform 10 runs of your program, plotting the average quality of the answer obtained after 0.25, 0.5, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 30, and 60 seconds. Comment on the graph (e.g., does performance asymptote fairly quickly, or is it still making noticeable gains after 10 seconds?).

What to hand in

1. Your code
2. Instructions on how to execute your code
3. Your suite of 5 test cases
    a. 1 test case should be very simple to help demonstrate your code does something sensible
    b. At least 1 test case should be complex, with no exact solution possible.
4. Your writeup

Hints

1. Think about how you will represent an arithmetic expression. You should not have a fixed length of organisms such as every organism having 10 operators. How will you do crossover for sequences of operators of different sizes?
2. If it makes your life easier, you can place a cap on the number of operators in an organism at 30 maximum.
3. For generating your initial population, you want a diverse set of useful organisms. One possible approach is to first select a random length. Then start at the first position and for each operator have a 50/50 chance of selecting it either randomly or via greedy search.

4. There are several candidates for what mutation could do: modify an existing operator, delete an existing operator, add an operator to an organism.
5. Time management of your search is a bit simpler, as the length of time an iteration takes is fairly predictable. Therefore, it should be possible to come very close to the specified amount of time. The smallest time interval we will use for testing is 1 second.
6. This assignment has a lot of work to do **after** you get your initial code base working: you'll need to experiment to tune parameters, write up what you did, and run some experiments. If you're finishing your code Monday night, you'll be in for a bad time. Getting something work quickly is better than trying to finish something clever Monday morning.
7. It is fine if you have separate programs for the genetic algorithm and for assignment #1. If you have separate programs, you should probably recycle a fair amount of code from assignment #1.
8. Work incrementally. Figure out how you're representing an organism and get random initial population working. Then get your selection and crossover code working. Then think about mutation.