COEN 166/266 - Spring 2015
Homework
Genetic Algorithms

Description

For this assignment, you will be developing a genetic algorithm to perform a local search. Your algorithm will be searching for the global maximum in a 7-dimensional landscape.

Goal

Develop a genetic algorithm-based local search in Scheme that will attempt to find the global maximum across 6 variables, each of which will have an integer value between 1 and 20, inclusive. The interface to this function will be:

search-ga

Parameters:	A fitness function that will accept a single parameter (a list of arguments).
Returns:	Your best individual solution in the population at the end of the local search. This should be a list containing 6 integer values.
Error conditions:	None, assumes the input is valid.

Each time that search-ga is invoked, you will be allowed to call the fitness function 50,000 times before you must return a value. Calls more than this many times to the fitness function will result in undefined behavior.

Here's an example of a possible invocation:

```
scheme@(guile-user)> (define (get-nth index alist)
                       (cond ((= 1 index) (car alist))
                             (#t (get-nth (- index 1) (cdr alist)))))
scheme@(guile-user)> (define (my-eval-fn alist)
                       (let ((a (get-nth 1 alist))
                             (b (get-nth 2 alist))
                             (c (get-nth 3 alist))
                             (d (get-nth 4 alist))
                             (e (get-nth 5 alist))
                             (f (get-nth 6 alist)))
                         (+ (* 6 a b)
                            (* 4 c d d)
                            (* -3 a a)
                            (* 46 e f)
                            (* -5 f c c))))
scheme@(guile-user)> (search-ga my-eval-fn)
(3 18 18 20 13 2)
scheme@(guile-user)>
```

Note that when compiled (this should happen automatically, otherwise give the --auto-compile option on the command line) this process should take a few seconds to run. If it takes longer than a minute, you are probably doing something wrong and should debug this before submission. You may want to display your progress by printing occasional status updates, such as the current generation count. Don't overwhelm the screen with excessive diagnostics, however.

The basic algorithm used for search-ga should follow the structure we discussed in class, however you may customize your agent to improve its performance by fine-tuning the:
- size of population
- algorithm for selecting which children to mate
- algorithm for replacing previous candidates with new candidates
- crossover point selection and algorithm
- mutation rate and algorithm

As an added bonus to spend some time tweaking your algorithm, I will award a 25 point bonus to the 5 students whose algorithms perform the best across a series of undisclosed test functions.

Recommended Steps

1. Determine a string representation and write your accessors
The string representation isn't particularly challenging here - a list of 6 numbers is probably reasonable. Remember, however, that you will also want to save fitness values along with each candidate, so you will want to decide how to represent an unevaluated candidate and an evaluated candidate.

2. generate-population
Write a function that will generate a population of candidates. You will probably want to use Scheme's "random" function to generate random values in the desired range.

3. evaluate-population
Write a function that will take a population and return that same population with fitness values assigned to them. An alternate approach would be to maintain the fitness values as a separate list. However it is implemented, this function will call the fitness function provided to search-ga.

4. choose-individual
Write a function to select an individual from the population, giving preference to those individuals with superior fitness. One approach would be to implement this as a roulette selection, with a simple linear weighting of the fitness values.

5. mate-individuals
Write a function that will take two individuals, select a crossover point, and return those two individuals with their attributes crossed over. This is perhaps the trickiest logic in the assignment - one approach that may make it easier would be to write a function that returns the first *n* elements in a list and another that returns the last 6-*n* elements in the list. Then, you just have to split the two candidates and recombine them using a call to append.

6. mutate-population
This function should go through all of your candidates, and occasionally change values. It should be invoked on the result of mate-individuals.

7. (optional) selectively-replace
This function will choose which members of the population to replace with which members from the next generation. For simplicity's sake, you may just want to replace the entire generation, at least to start.

8. process-generation
Finally, you should have enough of the functionality in place to implement the top-level function, which will repeatedly call (mutate-population (evaluate-population (mate-population population))). Remember to keep track of the number of calls to the fitness function that you have made and return a value before calling it more than 50,000 times. Also, upon completion you will want to look for the best candidate in the final population.

9. Testing!
Once you think you have completed the assignment, you'll want to verify that your algorithm is functional. You may want to dump some stats, like the average fitness value of each generation. This will give you an idea of how well your results are converging and will help you to fine-tune your parameters.

Submission

The assignment is due Wednesday, May 6th at 4:00pm and must be submitted to Camino. Your top-level function must use the interface specified in the assignment. Put all of your code into a file named "main.scm" (or, if you feel the need to split it up, make sure that all other files are loaded from within your main.scm file).

Grading

This assignment will be worth 200 points.

Submissions that do not meet the API specified in the assignment and require instructor time to modify for automated testing will incur a penalty of 10% per 5 minutes required (rounded up).

This is an independent assignment. Submissions that show signs of code sharing will be harshly penalized and the offending students will be reported to the COEN department for a violation of academic integrity. Talking about code (in general terms) is typically OK. Looking at each other's code is typically not.