CMPSC 370
Artificial Intelligence
Spring 2012
Bob Roos

Final Project (= Labs 10, 11)
(Second Handout)
Part 1 due Thurs., 19 April
Part 2 due Thurs., 26 April
Final writeup due Mon., 30 April

If you wish to do an alternate project, you must see me as soon as possible to discuss it. Otherwise I am assuming that everyone in the class will be working on the genetic algorithm problem.

Part 2: Due Thurs., 26 April in `proj2` dropbox folder

You have two choices here:

- **Option 1.** (More programming intensive). Modify the code for the methods `crossover`, `mutate`, and (optionally) `select` in program `Proj.java` according to the guidelines below and test them against the original versions;
- **Option 2:** (Some programming, but more experiment intensive.) Experiment with a richer representation form than a simple binary string (in particular, a string with four or more legal symbols, not just '0' or '1') and try to solve an optimization problem selected from the list below

OPTION 1: Crossover, Mutation, and Selection

For each of the following, I suggest that you make a *copy* of program `Proj.java`, rename it to something else (e.g., "`ProjC.java`," "`ProjM.java`," "`ProjS.java`"), being sure to rename the class as well as the file name. This will enable you to compare behaviors between the original and the modified methods.

Crossover

1. In program `Proj.java`, modify method `crossover` by implementing two-point crossover. Here we randomly select *two* locations to split the parent strings and we alternate between strings. For instance, if parent1 is "`0000000`" and parent2 is "`1111111`", then we might choose to split after the first and fourth characters, giving children "`0111000`" and "`1000111`".
2. [**OPTIONAL.**] Implement K-point crossover, where K is an integer between 1 and `LENGTH-1`. The value of K can be set as a `static` parameter by the user either prior to compilation (by directly setting its value in the code) or else at execution time via an additional call to `getInt` in the `setup` method.

The following example illustrates 4-point crossover:

```
Before:   p1 = 0 0 1 0 1 1 1 0 1 1 1 0 0 0
          p2 = 1 1 1 0 1 1 0 1 0 1 0 1 1 1

Crossover points randomly selected at positions 1, 4, 5, 8:
          p1 = 0 | 0 1 0 | 1 | 1 1 0 | 1 1 1 0 0 0
                 |       |   |       |
          p2 = 1 | 1 1 0 | 1 | 1 0 1 | 0 1 0 1 1 1

After:    s1 = 0 1 1 0 1 1 0 1 1 1 1 0 0 0
          s2 = 1 0 1 0 1 1 1 0 0 1 0 1 1 1
```

This is not trivial code to write; that's why it's optional!

3. Demonstrate that your modified crossover method works by running the genetic algorithm on the "8 queens" problem (use the appropriate `calcFitness` method in file `BitString.java`). Show the final fitness for three different runs. For each of the three runs, use 30 elements in the population, crossover probability .5, mutation probability .01, tournament size 2, elitism size 1, 100 generations, with string length 64.

   You are not expected or required to get better results using your modified crossover!

Mutation

4. The mutation method we have been using (randomly flipping bits) is actually the most sensible one for a bit string representation. However, we will explore a slightly different one here.

   The following algorithm does not make any use of the `MUTATION_PROB` constant!

   Modify method "`mutate`" so that it performs an *inversion*. To do an inversion, choose two locations at random in the bitstring, then *reverse* the bitstring lying between those two locations. For instance:

```
Before:   0010100101010010 01

 Randomly select the positions after the 3rd and 8th characters:

          001 01001 0101001001

 Reverse portion between these positions:

After:    0011001000101001001
```

   (The inversion operator works better for problems like the Traveling Salesman problem, where each element in the population is a permutation.)

5. Demonstrate that your modified mutate method works by running the genetic algorithm on the 8-by-8 "ant" problem (use the appropriate `calcFitness` method in file `BitString.java`). Show the final fitness for three different runs. For each of the three runs, use 30 elements in the population, crossover probability .5, mutation probability .01, tournament size 2, elitism size 1, 100 generations, with string length 64.

   You are not expected or required to get better results using your modified mutate method!

Selection [OPTIONAL]

6. **[OPTIONAL.]** Modify the `select` method in program `Proj.java` so that it implements *roulette wheel selection*. Roulette wheel selection should be used only in cases where the fitness values are all non-negative. In order to implement it, you must do the following:

   - Add up all the fitnesses of the elements in the previous population, `pop`; call this value `fitnessSum` (it should be of type `double`)
   - Generate a random `double` value (it will be between 0 and 1); multiply it by `fitnessSum` to get a random number `x` between 0 and `fitnessSum`.
   - Loop through the elements in the population, adding up the fitnesses of the elements as you come to them. As soon as adding an element's fitness to the sum causes the sum to become greater than `x`, return the element that caused this.

   Here is an example:

```
Population array (only fitness values are shown):

   +-----+-----+-----+-----+-----+-----+-----+
   |  2  |  5  |  4  |  3  |  6  |  1  |  3  |
   +-----+-----+-----+-----+-----+-----+-----+
     0     1     2     3     4     5     6

Sum of fitnesses = 24

Random real number: .31254
Times sum: .31254 * 24 = 7.50096

Since pop[0].fitness() + pop[1].fitness() add up to only 7,
but adding pop[2].fitness() brings the total to 11, we return
pop[2].
```

7. **[OPTIONAL.]** Demonstrate that your modified select method works by running the genetic algorithm on the "palindrome" problem (use the appropriate `calcFitness` method in file `BitString.java`). Show the final fitness for three different runs. For each of the three runs, use 30 elements in the population, crossover probability .5, mutation probability anything (since it is not used), tournament size 2, elitism size 1, 100 generations, with string length 63.

You are not expected or required to get better results using your modified select method!

Final Experiment

Choose any *one* of the problems I provided you with, or choose one of your own devising (e.g., from lab 8), and choose *one* of the modified methods you created above (either `crossover`, `mutate`, or `select`).

Run a number of experiments to determine whether or not your modified method does better, worse, or about the same as the original method. See part 1 of the project handout for explanations of how to run experiments. Generate summary data and boxplot graphs (you'll have two of these—one for the original method, one for the modified one). Use identical parameters in your experiments (if you are testing inversion, use a .01 mutation probability for the original method; it doesn't matter what you use for the inversion method).

OPTION 2: Beyond Bit Strings

Genetic algorithms can be applied to things other than bit strings. For instance, suppose we have FOUR symbols 0, 1, 2, and 3, rather than just two. Crossover is not affected by this. Selection is not affected, either. In fact, only three things need to be changed in the `Proj` and `BitString` files:

- In file `Proj.java`, one line in the *mutate* method needs to be changed. Replace the line:

```
result = result + (1-s.charAt(i)+'0'); // OLD
```

  with the line:

```
result = result + "0123".charAt(rng.nextInt(4)); // NEW
```

- In file `BitString.java`, in the `randomFill` method, replace the line:

```
item += "01".charAt(rng.nextInt(2));
```

  with the line

```
item += "0123".charAt(rng.nextInt(4));
```

- In `BitString`, provide a `calcFitness` method that operates on strings whose symbols include '0', '1', '2', and '3'.

9. Use a genetic algorithm to solve the following problem:

   starting with zero, generate a value as close to $\pi = 3.14159265358979$ as possible by performing n arithmetic operations selected from the following set of operations: "add 1," "subtract 2," "multiply by 3," "divide by 4." Each operation is applied to the result of the previous one. For instance, if $n = 7$ we can use:

$$((((((0 + 1) \div 4) \times 3) \div 4) \div 4) + 1) \times 3 = 3.140625$$

(I have no idea if that is the closest we can come to π in just seven of these operations.)

Represent a sequence of n operations using a string of length n consisting only of the characters '0' (which stands for "add 1"), '1' (which stands for "subtract 2"), '2' (which stands for "multiply by 3"), and '3' (which stands for "divide by 4"). In the example just given, the sequence of operations would be represented by the string `0323302`.

Your fitness method will need to calculate the number represented by this sequence of operations. Find the absolute value of the difference between this number and the value π and return its *negative* (since our GA always maximizes).

10. What is the closest you can get to π using no more than 5000 generations and using string lengths no bigger than 20? Show the results of experiments for several values of the parameters; for each combination of parameters, run experiments (at least 100) and compute summary statistics and boxplots as you did for part 1 of the project; show the results of these boxplots side by side.

    Note: it is actually easy to get close to π using large values of n, so the challenge is to find good solutions for smaller values of n like 10 or 12.

11. Invent one other problem that requires use of a four-symbol alphabet (e.g., each symbol is a command given to an artificial ant, or each symbol represents one of four colors to be used for a pixel in an image, or each symbol is a compass direction: north, south, east, or west). See me if you need ideas.

    Find the best solution you can. Perform extensive experiments for several values of the parameters; show statistics and boxplots as you did above and for part 1 of the project.

By Thursday, 26 April, you should upload at least one completed portion of the project (for instance, a `ProjM.java` file with modified mutation method plus graphs, R files, experiment scripts, and experiment output files).

By Monday, 30 April, you should have completed the remainder of the project and created a final report summarizing what you did (which parts of the project are you submitting?), what your experiments revealed (for instance, did two-point crossover ever do noticeably better or noticeably worse than one-point crossover?), and your observations (for instance, did you expect that the inversion operator would be useful in one particular problem because of the nature of that problem?).

The goal of the report is to show me that you have thought about genetic algorithms, how they are implemented, and what they can do. To the extent that you are able, try to make it self-contained (for instance, paste your graphs into the document if you can). You don't need to include full listings of things like `Proj.java` or `BitString.java`, but you might want to include "snippets" of code if, for example, you want to explain how you calculated fitness for a particular problem or how you dealt with a tricky problem in computing two-point crossover. fitne