Selection Methods and the Iterated Prisoner's Dilemma Problem

The goals of this assignment are: one - for you to understand how a few standard selection methods work and how they are implemented efficiently, and two – to explore the IPD problem and how various environments and evaluation methods lead to different population dynamics.

Your Task Overview

1. Given code to simulate the IPD problem using an EA, add implementations of the selection methods
    a. *fitness proportional selection with sigma scaling and stochastic universal sampling, plus optional elitism*
    b. *tournament selection*
2. Design experimental tests to check that your selection methods work correctly, and describe your tests and results in a report.
3. Explore the effects of various evaluation methods for a population of IPD strategies and describe the results in your report.

Provided IPD code

You will start with an existing program that simulates the IPD problem using a genetic algorithm to evolve the player strategies. The program already includes an implementation of a working GA that does evolve "good" strategies in some environments. However, the current selection method is simplistic and you will implement better algorithms. The current method is described in comments in the *Breeder.java* file.

To run the existing program, navigate to the top level directory (inside PrisonersDilemmaMod), and use the command "java ie.errity.pd.epd". A GUI will open that you can explore. The main thing you will do is to choose the Rules tab which allows you to set the parameters of the simulation, and then use the Game Type tab to choose Tournament and then click Start to run the simulation. ***Note*** your rules settings are saved in a file so that they persist over closing and re-running the application, however they are not set until you open the rules dialogue and save them again, if you forget to do this the simulation will run with the default parameters. You can click *defaults* anytime to return to those default parameters and see what they are. The available parameters and descriptions are listed here:

- A few are obvious from their names: *Maximum Generations, Number of Players, Mutate Probability, Crossover Probability*
- *PD iterations* – Number of turns each player will take in a single "game".
- *Temptation* – amount added to score of player who defects when her opponent cooperates
- *Reward* – amount added to score of player who cooperates when his opponent cooperates
- *Punishment* – amount added to score of player who defects when eir opponent defects
- *TooTrusting Payoff* – amount added to score of player who cooperates when zir opponent defects.

- *Selection* – Integer that codes for which selection method to use. Currently implemented method uses the value 0. Your two methods will use 1 (fit prop), and 2 (tournament), and anything else should use the present "else" which fills the population with "always cooperate" strategies.
- *Selection parameter* – A generic value that can be used for any parameter associated with a selection method. You will use it to turn elitism on and off and to set the tournament size.
- *Evaluation method* – Again an integer that codes for how to evaluate the population of strategies. Several methods are implemented in the file *Tournament.java* using codes 0 to 5, they are described there in the comments.
- *Random number seed* – used to seed the random number generator so that you can repeat identical runs when testing. A value of -1 causes runs to use the current time as a seed, so every run will be different. Any value >=0 will cause any run using that same seed and identical parameters to go exactly the same every time.

Experiment with the existing code to understand how the various parameters work and what data is given in the graphs/diagrams for each run.

The main file you will make changes is *Breeder.java*, though you are welcome to add/modify code anywhere that it is useful, as long as your final submission conforms to the requested behavior for specific given parameter settings. To recompile the program after you make changes, again start in the top directory, and use the command "javac ie/errity/pd/*.java javac ie/errity/pd/*/*.java". If you have a better way to do this or want to use ant to build the project, great! The command I've provided works just fine though if not.

Selection Methods

The two selection methods you will implemented are described in the textbook reading, although briefly and without any examples, so it is reasonable for you to look up more thorough explanations of these methods. They are both standard and more information will be easy to find. Feel free to look at descriptions, examples, algorithms, even so far as pseudo-code, but do not look at any actual implementations. Really just skip by if you see any code, in any language, it's too easy to be influenced by exactly how someone else structured their implementation, even if you just read through their code without any intention of copying it. A slightly more detailed general description for each method is given here:

- *fitness proportional selection with sigma scaling and stochastic universal sampling, plus optional elitism* - This sounds like a lot of things, but each part is simply a step that you will put together in a sequence to get the final result.
    - *Optional elitism* – the very first thing to do is to check if elitism is enabled, and if so select the highest *M* fitness individuals and copy them to the next generation without any variation. The selection parameter will determine how many individuals to copy (i.e. set the value of *M*). A value of 0 means not to incorporate elitism at all.
    - *Sigma scaling* – the next step is to re-evaluate the fitnesses of all individuals using the sigma scaling method with the parameters given in the book (there are other forms of this method that allow for stronger or weaker selection, but we

will stick with Mitchell's version). This scaling formula is pretty straightforward, but you will need to calculate the standard deviation of the population. The standard deviation is the square root of the variance, and the formula for variance is

sum((orig_fitness – mean_fitness)^2) / pop_size

Take the square root of the above and you have the standard deviation. We're evaluating the entire population, summing over every individual's original fitness in the population, so we use the above formula, not the slightly modified formula for calculating the variance of a population sample. The scaled fitness of each individual is then

*(orig_fitness – mean_fitness) / 2*standard_deviation*

Once you've calculated all scaled fitnesses, you continue in the next step using them just as you would have the original fitnesses.

- *fitness proportional selection with stochastic universal sampling* – this is just an efficient method of performing fitness proportional selection but with far less randomness than the roulette wheel method. You can think of this method in the same way, with each individual receiving a pie-slice on the roulette wheel proportional to their fitness. However, instead of spinning the wheel N times, you make N tic marks around the circumference of the wheel spaced equally apart, and then for each mark you select the individual who's pie piece that mark falls in. The only randomness then is in how you line up that ring of tic marks with the wheel. There are many ways to implement this method, but you should as always look for as efficient of one as you can design.

Provided IPD code

An important component to designing Evolutionary Algorithms is to assess whether they are working correctly. This can be difficult due to the stochastic nature and complexity of the algorithms; you generally can't analyze the code itself for correctness. Instead you have to design test experiments where you know what the outcome or population dynamics should be. For each of the selection methods you have implemented, describe several tests you will use to verify their correctness. State the results of the tests and describe how they do or do not indicate your code is functioning correctly. You should always include tests of the extremes of your methods; how could you set parameters to turn evolution off? To select as strongly as possible? What other tests that are more realistic can you use with verifiable results?

To be continued….