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 in which 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.

You will add your selection methods to the file *Breeder.java,* an example selection method is given there that you can use as a template. You should make separate functions, however, for your methods, and in general use good coding standard, readable comments, etc. A more detailed description for each selection 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.

- *Tournament selection* – This is much more straightforward. For each tournament, randomly choose *k* unique individuals from the population (*k* is given by the selection parameter), and select the one that has the highest fitness in that group. Repeat to perform pop_size tournaments total, thus choosing pop_size individuals as parents for the next generation.

Testing

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.

The IPD application you are using gives you some basic data for each run that will help you determine if an experiment is behaving the way you think it should, given the parameters you use. First it graphs 3 values over time in generations, the max fitness, the min fitness, and the average fitness. These values can tell you how diverse the population is (how different is the average fitness from the max), what strategy is being used (depending on your payoff matrix), and more. Also shown is the percentage of the population that falls into each of the "niceness" bins, determined by the number of C's vs D's in an individual's genome. Note that this is NOT the same as the number of times an individual cooperates or defects in an actual game; it only looks at the genome, much of which may not be used in an actual game. Still this can also tell you a lot about diversity and about the dominant strategy (which bin would you expect tit-for-tat to fall in?)

The program also gives you the ability to look closely at the final fittest/weakest individual. If you click "view fittest (or weakest) individual", you will see the genome of that individual. If you choose "save" you can give that genome a name and save it. In interactive mode (found in "game type" menu) you can pit a saved individual against any of the standard computer strategies and see how it performs in detail, or you can even play against it yourself.

YOUR TASK: For each of the selection methods you have implemented, describe several tests you will use to verify their correctness. State the specific parameters and 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?

You should start with tests where you know exactly (or at least mostly) what the final population should look like. For this you may find helpful the alternative evaluation methods available besides "play all against all"; you can use the "evaluation method" parameter described above to select among these. The actual methods are described in comments in the code (*Tournament.java*). For example, you know that the best an individual can do if it's only going to play other "all defect" individuals is to always defect itself, so you could use that evaluation method to make sure that with selection on the whole population does move towards "always defect".

A Real Problem

Currently the initial population is filled with random genomes; each of the 71 decisions is randomly set to cooperate or defect. Given some reasonable settings, you should find that the population usually converges to more or less tit-for-tat as a strategy. It won't be exact by any means, but if you test in interactive mode, you should find you are getting strategies that do

well against themselves, and do reasonably well against tit-for-tat. But what happens if we start with a "bad" initial population? Change the code to initialize a population of all "always defect" strategies. The initial population is created in Prisoner.java (line 278ish). This getRand function is called by TournamentPanel.java (line 320ish). Experiment with various mutation/selection parameters. Do not increase population size above 50. Can you get similar tit-for-tat type strategies to evolve now? Does strong or weak selection work better for this?

State the best parameters you find (to evolve more cooperative strategies like tit-for-tat), and give the specific results of experiments using those parameters. Keeping all other parameters equal, try stronger and weaker selection. How much can you vary selection strength and still get similar results? Again state actual parameter values and resulting experimental data to support your answer.