Problem Set 3

Due before midnight on Wednesday, October 6, 2010.

1 Assignment Goals

1. To learn how to organize a simulation of a large system.
2. To learn about a simple model of asynchronous distributed computing.
3. To learn how to use derived classes and virtual functions.

---

2 Problem

We consider the *consensus problem* in a simple setting. The players are trying to reach agreement on a course of action. Each player has a current preference called her *choice*, which is the current value stored in her *choice register*. We assume a simple binary choice, so the choice value is either 0 or 1. The players communicate with each other, and from time to time player may change their choices. The goal is for the players to arrive at a stage where all players are making the same choice and nobody subsequently changes her choice. In this case, we say the players have *reached consensus*, and we call the common choice the *consensus value*.

We assume a very primitive model of communication. A *communication step* consists of a randomly chosen player (called the *sender*) sending a message containing her current choice value to another randomly chosen player (called the *receiver*). The receiver may then choose to change her choice, depending on the message received and her internal state. She may also update her internal state depending on the message received.

A population of players *solves* the consensus problem if the following is true:

1. For all possible initial choices of the players, if the players start in their designated initial states, the computation reaches consensus with probability 1.
2. Once consensus has been reached, no player ever changes her choice. In particular, if all players have the same initial choice, then that choice is the consensus value.

There are many possible algorithms for reaching consensus. A particularly simple algorithm is *fickle*. Here, whenever a player receives a message, she changes her choice if necessary to agree with the sender. That is, she sets her choice register to the value contained in the message she receives. It is easy to see that there is some sequence of message transmissions that causes the system to reach consensus, and once consensus has been reached, no player will change her choice. It is also easy to believe that it might take a very large number of random steps to reach consensus.

The algorithm, *follow the crowd*, seems to be a big improvement. Here, each player has a 1-bit state register in which she saves the last message received. She changes her choice

only when she gets two messages in a row that both disagree with her current choice. Thus, she waits until she gets a sense of the crowd before deciding to follow. We assume that all players start with 0 in their state registers.

The purpose of this assignment is to simulate the two algorithms *fickle* and *follow the crowd* to determine how long each takes on average to reach consensus. Assume a population of `numPlayers` many players. An experiment for a particular algorithm consists of `numTrials` runs of the algorithm. Each run begins with `numPlayers/2` players choosing 1 and the remaining players choosing 0. Random communication steps are generated and applied until consensus is reached, at which point the run ends. For each step, a sender is selected uniformly at random from the set of all players, and a receiver is selected uniformly at random from the remaining players (other than the sender). The receiver's choice and state are updated according to the rules of the selected algorithm. Note that in both of these algorithms, no further changes of choice are possible once consensus has been reached. The result of the experiment is the number of steps taken to reach consensus, averaged over the runs in the experiment.

The three parameters of interest for an experiment are the algorithm being used, the number of runs in the experiment (`numTrials`), and the population size (`numPlayers`. Once your program is working, you should use it to explore the parameter space. This means that you should run experiments with many different combinations of these parameters so as to give one a pretty good idea of the performance characteristics of the two algorithms.

Increasing `numTrials` will reduce the variance in the average run time. This is not a statistic course, and I am not going to ask you to do a confidence analysis. Rather, you may take `numTrials` to be 100 in your experiments (but it should still be a parameter to your program).

Increasing the population size `numPlayers` will cause a big increase in run time. You may terminate your experiments once the run time grows to more than a few seconds. This may come rather quickly with *fickle*, but that is for you to find out.

3 Program Notes

For ease of testing, your executable program should be called `agree` and should take up to four command line arguments: The name of the algorithm to run (either `fickle` or `crowd`), `numPlayers`, `numTrials`, and `seed`, where `seed` is the seed to be used for the pseudorandom number generator used to choose the sender and receiver at each step. Missing parameters should use default values of `fickle` for the algorithm, `10` for `numPlayers`, `100` for `numTrials`, and the result of `time(0)` for the seed. Parameters can only be missing from the right end of the command line, so if the command has two arguments, for example, then the missing arguments are `numTrials` and `seed`.

The code should be modular and structured into natural well-defined classes of small to modest size. In particular, you should define a base class `Player` to represent the player. This class should have derived classes `Fickle` and `Crowd` to represent the two different kinds of players – those that use the *fickle* algorithm and those that use the *follow the crowd* algorithm. The function that updates the receiver's choice and state registers (perhaps called `receive()`) should be virtual in class `Player` and be defined appropriately in classes `Fickle` and `Crowd`. You will need several other classes as well. You may use the various demo programs such as `BarGraph` and problem set solutions as guides for how to structure a program of modest complexity such as this.

4 Deliverables

You should submit this assignment using the `submit` script that you will find in `/c/cs427/bin/` on the Zoo. Remember to submit the following items:

1. Source code and header files.
2. A `makefile` and any other files necessary to build your project. (If you're using Eclipse with the default settings, the makefiles will be found in the directory `Debug`.)
3. Test programs, output, and methodology that you used to test the correctness of your program. For example, it is easy to see that, with a population size of 2, *fickle* will reach consensus in exactly one step, so if your program gives an answer different from 1.0 in that case, it has a bug that must be fixed.
4. The data from the experiments you ran in the form of a csv file, where each row contains four comma-separated fields: The algorithm name (enclosed in double quotes), numPlayers, numTrials, and the computed average number of steps as a fixed point decimal number.
5. A human-readable summary of the data in whatever form you choose, perhaps graphs comparing the two algorithms for different population sizes, or a textual description of what you can conclude from examining the data. Graphs can be produced easily from a csv file using OpenOffice (available on the Zoo) or other spreadsheet program.
6. The scripts or other programs that you used to run the experiments. It is perfectly acceptable (and probably easier) to write a second command that runs the experiments directly and generates the csv file automatically instead of calling the command `agree` repeatedly with different arguments.
7. A brief report named `report.txt` (or any other common format such as `.pdf`) describing the design choices you made in implementing the required code extensions. You can also put any other information here that the TA should know about your program.