CS3114 (Spring 2011)
PROGRAMMING ASSIGNMENT #1

Due Tuesday, February 15 @ 11:00 PM for 100 points
Early bonus date: Monday, February 14 @ 11:00 PM for a 10 point bonus
Initial Schedule due Thursday, January 27 @ 11:00 PM
Intermediate Schedule due Tuesday, February 8 @ 11:00 PM

Reaction networks are used to model problems such as chemical interactions in air and protein interactions within a cell. The mutual interactions of a collection of chemicals or proteins can be used to build a variety of behaviors surprisingly like computer circuits. The fundamental activity that is modeled by such a system is the changes in the amounts of the various reactants over time, as they combine and break apart. (Note that it is traditional to refer to the chemicals or proteins involved in the network as "chemical species" or just "species". Anytime you see the word "species" in this document, it refers to on of the chemicals or proteins in the system, *not* a biological species such as dog or a monkey!)

In this project, you will build a type of discrete event simulator for modeling the behavior of a network of reactions. Your program will read in the set of reactions and related rate constants, and then it will simulate the system over a specified period of time. The output will be information about how the amounts of the reactants changed over time and the number of times that each reaction fired. The heart of the simulation will be the data structures used to find the next reaction and update the populations of the associated reactants. Since the simulation is driven by random numbers, this approach is referred to as a stochastic simulation, and the algorithm to drive it is referred to as a stochastic simulation algorithm or SSA.

Implementation:

Gillespie's stochastic simulation algorithm, on which many variations have been proposed that improve its efficiency. You will be implementing one such variation of Gillespie's SSA. Suppose a set of reactions involves N species $\{S_1, S_2, \ldots, S_N\}$ and M reactions $\{R_1, R_2, \ldots, R_M\}$. The state vector defining the state of the system represents the populations of the N species at time t and is denoted by $X(t) = (x_1(t), x_2(t), \ldots, x_N(t))$, where $x_i(t)$ represents the population of species S_i at time t. For each reaction R_j, the probability for it to fire is calculated from the *propensity function* a_j, which is usually given by a reaction rate multiplied by the product of the populations for all its reactants. The logic behind this definition is that more of the reactants for a given reaction that are available, the more likely that they will "bump into each other" causing the reaction to take place. The reaction rate is a constant assigned to each reaction which, along with the population size of its reactions, also governs how often the reaction fires.

When one reaction fires, the populations of the reactants are decreased and the populations of the products are increased by the corresponding amounts. (Remember that population is denoted as x_i for species i.) For example, the propensity for the reaction

$$S_1 + S_2 \xrightarrow{k_1} 2S_3$$

is given by $k_1 x_1 x_2$, where k_1 is the rate constant assigned to this reaction. When it fires, x_1 and x_2 will each decrease by one and x_3 will increase by two. The propensity function for the reaction

$$2S_4 \xrightarrow{k_2} S_5$$

is given by $k_2 x_4(x_4 - 1)$. When it fires, x_4 will decrease by two, and x_5 will increase by one.

In this project you will implement a specific variation on the SSA, defined as follows. Denote the current system time by t. At the beginning, $t = 0$. The simulation simply executes a loop where it picks the next reaction to fire, updates the simulation clock to this reaction's firing time, updates the species populations appropriately, and then updates the propensities (since populations changed the propensities also changed) and next firing times for the reactions.

The next firing time for a given reaction j is defined as follows. The algorithm generates one uniform random number r_j in the range 0 to 1. The next firing time is given by $f_j = t + \tau_j$, where τ_j is given by

$$\tau_j = \frac{1}{a_j(x)} \log(\frac{1}{r_j}). \tag{1}$$

At each simulation cycle, the algorithm selects the reaction with the minimum next reaction time f_j. Then the system states are updated by firing reaction R_j. The simulation time proceeds to f_j. Then the propensities and next reaction times are updated. The simulation continues until it reaches the final time.

A simplest implementation for SSA (Gillespie's original algorithm) would re-calculate propensities for all reactions, re-generate those random numbers, and re-calculate the next reaction times in each step. However, for a large set of reactions, when a reaction fires it usually changes the population for only a couple of species. So the propensities and therefore the next reaction times of any reactions that do not involved those species remain the same. Thus it is not necessary to re-calculate propensities and next reaction times for all reactions. If smart data structures are adopted, then the simulation will be much more efficient. We can make use of two data structures to achieve this improvement.

- **Reaction Dependency Table**: This table stores, for each reaction, the indices of all reactions that are affected in turn when this reaction fires. When one reaction fires, update the species populations and then check this table to see what other reactions have to update their propensities and next reaction times.

- **A Heap for Next Reaction Times**: This data structure stores next reaction times for all reactions. In each step, the minimum next reaction time is selected and removed from the heap. Then for each reaction whose propensity function (and therefore its next reaction time) is affected, its next reaction time should be re-calculated and updated in the heap.

Input and Output:

Your program will be named P1 and will be invoked with three command line parameters. The first will be an integer that indicates the number of simulation runs to be made. The second will be the name of a file containing the description of the model to be simulated. The third will be the name of the output file.

You are required to design and implement the two data structures and implement the simulation algorithm described above. To simplify the task, we assume that reactions are limited to at most two reactants and at most two products. Models to be simulated will be defined by a text file, formatted as follows.

- The first line consists of four values, N, M, D and $SimulationTime$, where integer N specifies the number of species, integer M specifies the number of reactions, integer D specifies the number of chemical species that will be displayed in the output and integer $SimulationTime$ is the final time at which the simulation should end.
- The second line contains N integers representing the initial populations of the N species.
- The third line contains D integers representing the indices of the D species whose populations will be tracked.
- Starting from the fourth line, each line contains the information for one reaction. For each reaction, the line has the following format. It can have zero, one, or two reactants (two reactants are separated by a plus sign). It can have zero, one, or two products (two products are separated by a plus sign). Reactants are separated by products by the symbol "`->`" which is immediately followed by a real-valued rate constant. A reactant or products is represented as Si where i is the index for the species. A reactant or product can be preceded by an integer indicating the number of molecules of that species that are involved.

For example, consider the following model description:

```
3 4 1 10
1000 200 0
2
->100 S1
2S1 ->0.001 S2
S2 ->0.5 2S1
S2 ->0.04 S3
```

It represents a system with 3 species and 4 reactions, where one species (S2) is tracked in the output, and the initial values given for populations are $x_1 = 1,000$, $x_2 = 200$, and $x_3 = 0$. The simulation time is from $t = 0$ to $t = 10$. The four reactions are

$$\begin{array}{rcl} \emptyset & \longrightarrow & S_1, \\ S_1 + S_1 & \longrightarrow & S_2, \\ S_2 & \longrightarrow & S_1 + S_1, \\ S_2 & \longrightarrow & S_3, \end{array} \tag{2}$$

and the initial propensities are

$$\begin{array}{rcl} a_1(x) & = & 100, \\ a_2(x) & = & 0.001(1000)(999), \\ a_3(x) & = & 0.5(200), \\ a_4(x) & = & 0.04(200). \end{array} \tag{3}$$

You will not need to do error testing on the format of the input file. All input test files that we give you or use for grading will be syntactically correct.

Output for the project will depend on the number of simulation runs, as specified in the first command line parameter. If the number of simulation runs is one, then you will be outputting

a list of the events (reactions fired) and the populations of the tracked species. For each of the D species being tracked, a line of output is written to the output file every time the population of any tracked species is changed. That line will give the current simulation time followed by the populations of the D species in the same order as given in the third line if the input file. Also, you should keep track of the number of times each reaction fires. At the end of the simulation, you will output M lines where the ith line indicates the number of times that reaction R_i fired.

If the number of simulation runs is greater than one, then you will be outputting summary information about each run, and final summary statistics about the entire series of runs. For each simulation run, you will send to the output file a single line that lists the final populations of the D species being tracked, in the order specified in the model input file. After all simulations have been run you will print out a final two lines: The mean final population for each species being tracked, and the final variance of the population for each species being tracked. These final two lines (means and variances) should also be written to Standard Output in addition to the output file.

Implementation Advice:

You will be given several sample input files to help you test your project, along with sample output.

You are given roughly four weeks to complete the project. We recommend that you gain a thorough understanding of the project specification during the first week work up your initial schedule (see below). During the second week, you should implement the parser for reading the input files, and implement a simple version of the Gillespie algorithm where you just recompute the propensities and generate new firing times for all reactions. In the third week, once you are sure that the basic simulation works correctly, add in the data structures to speed up your simulation. Then you can finalize the project in the fourth week.

Note that since you are implementing a stochastic simulation, different seeds to the random number generator will give different outputs. Thus the population trajectories for different runs will not be exactly the same. But the system behavior will be similar from run to run. Also the mean and variance of the states over a large number (typically $10,000$) of runs should be accurate. This large number of runs required to get averages is why code efficiency is so important for practical application.

Programming Standards:

You must conform to good programming/documentation standards. Some specifics:

- You must include a header comment, preceding main(), specifying the compiler and operating system used and the date completed.
- Your header comment must describe what your program does; don't just plagiarize language from this spec.
- You must include a comment explaining the purpose of every variable or named constant you use in your program.
- You must use meaningful identifier names that suggest the meaning or purpose of the constant, variable, function, etc.
- Always use named constants or enumerated types instead of literal constants in the code.

- Precede every major block of your code with a comment explaining its purpose. You don't have to describe how it works unless you do something so sneaky it deserves special recognition.
- You must use indentation and blank lines to make control structures more readable.
- Precede each function and/or class method with a header comment describing what the function does, the logical significance of each parameter (if any), and pre- and post-conditions.
- Decompose your design logically, identifying which components should be objects and what operations should be encapsulated for each.

Neither the GTAs nor the instructors will help any student debug an implementation unless it is properly documented and exhibits good programming style. Be sure to begin your internal documentation right from the start.

You may only use code you have written, either specifically for this project or for earlier programs, or code taken from the textbook. Note that the textbook code is not designed for the specific purpose of this assignment, and is therefore likely to require modification. It may, however, provide a useful starting point.

Testing:

A sample data file will be posted to the website to help you test your program. This is not the data file that will be used in grading your program. The test data provided to you will attempt to exercise the various syntactic elements of the command specifications. It makes no effort to be comprehensive in terms of testing the data structures required by the program. Thus, while the test data provided should be useful, you should also do testing on your own test data to ensure that your program works correctly.

Deliverables:

When structuring the source files of your project (be it in Eclipse as a "Managed Java Project," or in another environment), use a flat directory structure; that is, your source files will all be contained in the project root. Any subdirectories in the project will be ignored. If you used a makefile to compile your code, or otherwise did something that won't automatically compile in Eclipse, be sure to include any necessary files or instructions so that the TAs can compile it.

If submitting through Eclipse, the format of the submitted archive will be managed for you. If you choose not to develop in Eclipse, you will submit either a ZIP-compressed archive (compatible with Windows ZIP tools or the Unix `zip` command) or else a tar'ed and gzip'ed archive. Either way, your archive should contain all of the source code for the project, along with any files or instructions necessary to compile the code. If you need to explain any pertinent information to aid the TA in the grading of your project, you may include an optional "readme" file in your submitted archive.

You will submit your project through the automated Web-CAT server. Links to the Web-CAT client and instructions for those students who are not developing in Eclipse are posted at the class website. If you make multiple submissions, only your last submission will be evaluated.

You are permitted (and ecouraged) to work with a partner on this project. When you work with a partner, then **only one member of the pair** will make a submission. Be sure both names are

included in the documentation. Whatever is the final submission from either of the pair members is what we will grade unless you arrange otherwise with the GTA.

Scheduling:

In addition to the project submission, you are also required to submit an initial project schedule, an intermediate schedule, and a final schedule with your project submission. The schedule sheet template for this project is posted at the course website. You won't receive direct credit for submitting the schedule as required, but each instance of failing to submit scheduling information as required will lose 10 points from the project grade.

Pledge:

Your project submission must include a statement, pledging your conformance to the Honor Code requirements for this course. Specifically, you must include the following pledge statement near the beginning of the file containing the function main() in your program. The text of the pledge will also be posted online.

```
// On my honor:
//
// - I have not used source code obtained from another student,
//   or any other unauthorized source, either modified or
//   unmodified.
//
// - All source code and documentation used in my program is
//   either my original work, or was derived by me from the
//   source code published in the textbook for this course.
//
// - I have not discussed coding details about this project with
//   anyone other than my partner (in the case of a joint
//   submission), instructor, ACM/UPE tutors or the TAs assigned
//   to this course. I understand that I may discuss the concepts
//   of this program with other students, and that another student
//   may help me debug my program so long as neither of us writes
//   anything during the discussion or modifies any computer file
//   during the discussion. I have violated neither the spirit nor
//   letter of this restriction.
```

Programs that do not contain this pledge will not be graded.