SIMULATION SIMPLIFIED

Alan Kaminsky
Associate Professor
Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, NY, USA
`http://www.cs.rit.edu/~ark/`
`ark@cs.rit.edu`

February 4, 2014

Please consider the environment before printing this book!

Contents

Preface

While teaching various courses, including Data Communications and Networks, Ad Hoc Networks, and Distributed Systems, I have found myself wanting to explore questions such as these with the class:

- How long does it take for a packet to travel from source to destination through a network of routers?
- What is the probability that a packet will be dropped due to congestion in the network as a function of the network traffic level?
- A bunch of sensor nodes are placed at random positions within a certain area. The nodes' transmitters are not powerful enough to reach the base station; instead, messages are relayed from node to node until they reach the base station. Will the base station be able to communicate with every sensor node?
- A group of peer devices is organized into a distributed hash table (DHT). A query originates at a certain device and hops from device to device according to a certain algorithm until the query reaches the device containing the answer. The DHT algorithm's authors claim that the number of hops is proportional to the logarithm of the number of devices. Is it?

There are three ways to answer such questions:

- Build the **actual system,** collect data by observing the system in operation, and answer the questions by analyzing the data. But this might be prohibitively expensive, or it might take too long, or it might not be possible to measure the desired quantities, or one might want to design the system *before* actually building it.
- Make a **mathematical model** of the system and answer the questions by analyzing the model. But the system might be too complicated to make a mathematical model (at least one that can be easily analyzed), or one might lack the mathematical skill to build and analyze a model.
- Write a computer program to **simulate** the system, run the program to collect data, and answer the questions by analyzing the data. This is often much more practical than the alternatives.

There exist general-purpose simulation languages, as well as specialized simulation packages in various domains, such as network simulation. However, I've usually found these languages and packages to be large, complex, and feature-laden, such that writing even a simple little simulation requires climbing a steep learning curve. At the same time, I've usually found these languages and packages to lack features I want, and it's either difficult (because the language's or package's source code is unreadable) or impossible (because the language or package is proprietary) to add features.

All is not lost! I have found that with the ability to write high level language code plus a modicum of background in probability and statistics, one can develop a broad range of useful simulation programs. However, the techniques for writing simulation programs are not usually taught in computer science curricula. Through a series of examples in Chapters 1–20, this book teaches the art of simulation programming. This book is intended to be used as a supplement in a course that uses simulation as an investigatory tool.

The Java programming language is used for the examples in this book. However, those with experience in other high level languages should have no difficulty following the discussion and adapting the programs to their preferred language. The example program source files are free, GNU GPL licensed software and may be downloaded from my web site (`http://www.cs.rit.edu/~ark/ss/`). Source file listings appear in Appendix A.

This book also makes use of Java classes from my Parallel Java 2 Library. The Parallel Java 2 Library is free, GNU GPL licensed software and may be downloaded from my web site (`http://www.cs.rit.edu/~ark/pj2.shtml`). The download includes complete source code and Javadoc documentation. While the Parallel Java 2 Library is mainly intended for writing parallel programs in Java, it contains many classes useful for non-parallel programs as well. (I may discuss *parallel* simulation programming in a later edition of this book.)

For those desiring further information about various topics mentioned in this book, I recommend the sources listed in Appendix B, "Further Reading."

Alan Kaminsky
February 2014

Chapter 1

A Dice Game

Imagine you're in a casino playing this simple game of chance: You are rolling a six-sided die repeatedly. Each time you roll, the casino pays you one dollar. When you roll a 6, you stop. If you played this game many times, how much do you expect to win each time on the average? (Actually, that's how much the casino should charge you for the privilege of playing this game; then, on the average, you and the casino would be even.)

Let's write a program to simulate this dice game. First we need an object that can simulate the random throws of the die. That object is a **pseudo-random number generator (PRNG).** We will use class edu.rit.util.Random from the Parallel Java 2 Library. Here is the code to create a PRNG object:

```
   Random prng = new Random (seed);
```

(We will disregard the `seed` argument for now and return to it later.)

Now we need to use the PRNG to simulate throws of the die. Here is the code to generate a random number:

```
   int die = prng.nextInt (6) + 1;
```

Calling `nextInt (6)` returns a random integer in the range 0 through 5 inclusive, with each integer having an equal probability of occurring. Adding 1 to that yields a random integer in the range 1 through 6 inclusive. Each time we execute the above statement, we get a new random number.

To simulate the game, put the above statement in a loop, stopping when a 6 is thrown:

```
   Random prng = new Random (seed);
   int die;
   do
      {
      die = prng.nextInt (6) + 1;
      }
   while (die != 6);
```

We also need to keep track of the winnings:

```
   Random prng = new Random (seed);
   int win = 0;
   int die;
   do
      {
      ++ win;
      die = prng.nextInt (6) + 1;
      }
   while (die != 6);
```

Here's a nearly-complete program to simulate the game, printing out the numbers thrown on the die plus the final winnings:

```
import edu.rit.util.Random;
public class Dice01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      Random prng = new Random (seed);
      int win = 0;
      int die;
      do
         {
         ++ win;
         die = prng.nextInt (6) + 1;
         System.out.printf ("%d%n", die);
         }
      while (die != 6);
      System.out.printf ("Winnings = $%d%n", win);
      }
   }
```

The only missing piece is the `seed` argument used to initialize the PRNG. If you initialize two PRNGs with different seed values, the two PRNGs will generate different sequences of random numbers. If you initialize two PRNGs with the same seed value, the two PRNGs will generate the same sequence of random numbers. When working with simulation programs, it is *very, very important* to be able to repeat an experiment exactly, with a random number sequence identical to a previous experiment. Therefore, you must *always, always* let the experimenter specify the seed when creating a PRNG. We will generally specify the seed as a command line argument. Here is the complete game simulation program:

```
import edu.rit.util.Random;
public class Dice01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 1) usage();
      long seed = Long.parseLong (args[0]);
      Random prng = new Random (seed);
      int win = 0;
      int die;
      do
         {
         ++ win;
         die = prng.nextInt (6) + 1;
         System.out.printf ("%d%n", die);
         }
      while (die != 6);
      System.out.printf ("Winnings = $%d%n", win);
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice01 <seed>");
      System.exit (1);
      }
   }
```

Here's what the program prints with a seed of 142857:

```
$ java Dice01 142857
3
3
6
Winnings = $3
```

If we run the program again with the same seed, we get the same result:

```
$ java Dice01 142857
3
3
```

```
6
Winnings = $3
```

If we run the program with a different seed, we get different results:

```
$ java Dice01 285714
4
2
4
5
5
4
2
6
Winnings = $8
```

The Dice01 program simulates one play of the game, or in simulation parlance, one **trial.** However, we are interested in the average winnings for a large number of trials. To answer that question, we could run the Dice01 program many times and compute the average winnings by hand. But why not let the computer take over the drudgery? Here's version 2 of the game simulation program:

```
import edu.rit.util.Random;
public class Dice02
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
```

We'll let the experimenter specify two command line arguments, the number of trials N and the seed.

```
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
      Random prng = new Random (seed);
      int win = 0;
      int die;
```

We'll add an outer loop to do N trials. The `win` variable will accumulate the total winnings for all the trials. We'll omit printing the numbers thrown on the die and the winnings for each trial because we're only interested in the final average.

```
      for (int trial = 0; trial < N; ++ trial)
         {
         do
            {
            ++ win;
            die = prng.nextInt (6) + 1;
            }
         while (die != 6);
         }
```

The average winnings per trial is the total winnings divided by the number of trials. We have to convert `win` and `N` to floating point numbers (type `double`) before dividing them. If we don't, the computer will do an integer division, which discards the fractional part of the answer.

```
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice02 <N> <seed>");
      System.exit (1);
      }
   }
```

Now we can run the experiment with as many trials as we like. Let's see what happens as we increase the number of trials from 10 to 10 million:

```
$ java Dice02 10 142857
Average winnings = $3.50
$ java Dice02 100 142857
Average winnings = $6.73
$ java Dice02 1000 142857
Average winnings = $5.78
$ java Dice02 10000 142857
Average winnings = $5.91
$ java Dice02 100000 142857
Average winnings = $6.01
$ java Dice02 1000000 142857
Average winnings = $6.00
$ java Dice02 10000000 142857
Average winnings = $6.00
```

As the number of trials increases, the simulation program's result—the average winnings—jumps around, but eventually converges to a stable value of $6.00. This is in fact the result predicted by probability theory.

The Dice02 program illustrates that, like a real experiment, a simulation program does not necessarily produce the precise answer predicted by theory. There is always some amount of **experimental error.** For simulation programs using random numbers like this one, the relative error between the observed result and the theoretical true result is proportional to the square root of the number of trials. To get an answer accurate to three significant figures, or one part in 1,000, on the order of 1,000,000 trials are needed.

Chapter 2

Bernoulli Distribution

Now that we're no longer printing the actual numbers thrown on the die, it's a little bit of overkill to generate a random integer between 1 and 6. All we really need is a PRNG with two outcomes: "keep rolling" with a probability of 5/6, or "stop" with a probability of 1/6.

A random variable with only two possible values is said to obey a **Bernoulli distribution** (named after Swiss mathematician Jacob Bernoulli, 1654–1705). We'll call the two values "heads" and "tails." The Bernoulli distribution has a parameter p which is the probability of heads, $0 \le p \le 1$. The probability of tails is $q = 1 - p$.

All we really need for the dice game simulation program, then, is a PRNG that generates random values from a Bernoulli distribution with a given heads probability p. Later in the book we'll need to generate random numbers obeying all different kinds of probability distributions.

Class edu.rit.util.Random does not support arbitrary probability distributions. However, class edu.rit.util.Random can generate random numbers obeying a **uniform distribution.** The statement

```
    double x = prng.nextDouble();
```

sets x to a random floating point number uniformly distributed between 0 and 1. (To be precise, class edu.rit.util.Random draws the random numbers from a set of 2^{64} possible values: $\{0, 1/2^{64}, 2/2^{64}, \ldots, (2^{64} - 1)/2^{64}\}$, with each value equally likely; that is, from 0 to just a smidgen less than 1.)

To get a sequence of Bernoulli random values, we can *transform* a sequence of uniform random values. If `nextDouble()` is less than p, the outcome is heads, otherwise the outcome is tails (Figure 2.1).

Here's version 3 of the game simulation program. We get rid of the `die` variable and change the inner loop condition as suggested above.

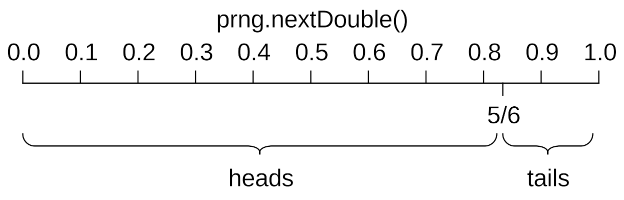


Figure 2.1: Generating a Bernoulli random value with $p = 5/6$

```
import edu.rit.util.Random;
public class Dice03
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
      Random prng = new Random (seed);
      int win = 0;
      for (int trial = 0; trial < N; ++ trial)
         {
         do
            {
            ++ win;
            }
         while (prng.nextDouble() < 5.0/6.0);
         }
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice03 <N> <seed>");
      System.exit (1);
      }
   }
```

Pop Quiz: Why is the inner loop condition not "`prng.nextDouble() < 5/6`"?
Answer: `5/6` does an integer division, yielding 0.

The Dice03 program makes the same decision about each random number generated—whether to keep rolling or to stop—as the Dice02 program. Therefore, if we run Dice03 with the same number of trials and the same PRNG seed as Dice02, Dice03 should encounter the same sequence of random numbers as Dice02, make the same decisions as Dice02, and produce the same result as Dice02. In fact, it does:

```
$ java Dice03 10 142857
Average winnings = $3.50
$ java Dice03 100 142857
Average winnings = $6.73
$ java Dice03 1000 142857
Average winnings = $5.78
$ java Dice03 10000 142857
Average winnings = $5.91
$ java Dice03 100000 142857
Average winnings = $6.01
$ java Dice03 1000000 142857
Average winnings = $6.00
$ java Dice03 10000000 142857
Average winnings = $6.00
```

Rather than writing expressions like “`prng.nextDouble() < 5.0/6.0`” in an *ad hoc* fashion every time we need a random variable with a Bernoulli distribution, it would be better to encapsulate the logic for the Bernoulli distribution in its own separate class. Class edu.rit.numeric.BernoulliPrng in the Parallel Java 2 Library does just that:

```
package edu.rit.numeric;
import edu.rit.util.Random;
public class BernoulliPrng
   {
```

The class needs two fields: a reference to the underlying PRNG object from which to obtain uniform random numbers, and the heads probability.

```
   private final Random myUniformPrng;
   private final double myP;
```

The constructor initializes those fields, after checking that the arguments are legal.

```
   public BernoulliPrng
      (Random theUniformPrng,
       double p)
      {
      if (theUniformPrng == null)
```

```
         {
         throw new NullPointerException
            ("BernoulliPrng(): theUniformPrng is null");
         }
      if (0.0 > p || p > 1.0)
         {
         throw new IllegalArgumentException
            ("BernoulliPrng(): p = "+p+" illegal");
         }
      myUniformPrng = theUniformPrng;
      myP = p;
      }
```

The `next()` method returns heads (true) with probability p and tails (false) with probability $1 - p$.

```
   public boolean next()
      {
      return myUniformPrng.nextDouble() < myP;
      }
   }
```

Armed with class BernoulliPrng, we can write version 4 of the game simulation program.

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.util.Random;
public class Dice04
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
```

We still construct a uniform PRNG object, but now we use it to construct a Bernoulli PRNG object with a heads probability of 5/6.

```
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      int win = 0;
      for (int trial = 0; trial < N; ++ trial)
         {
         do
```

```
            {
            ++ win;
            }
```

And we change the inner loop condition to test the random heads or tails value produced by the Bernoulli PRNG.

```
         while (prng.next());
         }
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice04 <N> <seed>");
      System.exit (1);
      }
   }
```

Thus, the Dice04 program uses class edu.rit.numeric.BernoulliPrng, which is layered on top of class edu.rit.util.Random (Figure 2.2).

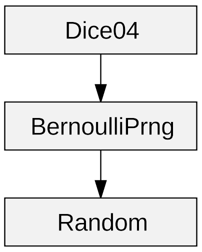


Figure 2.2: Dice04 program design

It's no surprise that the Dice04 program produces the same results as the Dice03 program, given the same PRNG seed.

```
$ java Dice04 10 142857
Average winnings = $3.50
$ java Dice04 100 142857
Average winnings = $6.73
$ java Dice04 1000 142857
Average winnings = $5.78
$ java Dice04 10000 142857
Average winnings = $5.91
```

```
$ java Dice04 100000 142857
Average winnings = $6.01
$ java Dice04 1000000 142857
Average winnings = $6.00
$ java Dice04 10000000 142857
Average winnings = $6.00
```

Chapter 3

Gathering and Displaying Data

Let's gather a little bit more data from the dice game simulation. Rather than just the average winnings for many trials, we want to know many times we won \$1, how many times we won \$2, and so on. That is, we want to know the **distribution** of the winnings for many trials. Viewing the winnings as a random variable, we are trying to determine the **probability distribution** of the winnings.

Because the winnings for one trial could be infinite—it is theoretically possible, however unlikely, to roll a die forever and never see a 6—we have to impose a cutoff. Let's say we'll gather data for winnings of \$1 through \$`maxwin` plus "everything else," lumping all winnings more than `maxwin` into the "everything else" category. We'll let the experimenter specify `maxwin` on the command line. Here's version 5 of the game simulation program:

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.util.Random;
public class Dice05
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
```

This time, instead of accumulating the total winnings, we'll accumulate occurrences of each winnings amount in an array of counters, indexed from 0 to `maxwin`. Thus, the length of the array is `maxwin` + 1. Index 0 is for winnings of $1, index 1 is for winnings of $2, and so on. Index `maxwin` is for winnings of more than $`maxwin`.

```
      int[] wincount = new int [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
```

At each trial, we have to reinitialize the winnings to zero, then proceed to simulate the game.

```
         int win = 0;
         do
            {
            ++ win;
            }
         while (prng.next());
```

At the end each trial, we increment the counter associated with the amount of winnings. Notice how we get winnings of more than $`maxwin` into the counter at index `maxwin` and the others into the counters at indexes 0 through `maxwin` − 1.

```
         ++ wincount[Math.min (win - 1, maxwin)];
         }
```

After all the trials are finished, we print out the data we gathered.

```
      System.out.printf ("Win\tCount%n");
      for (int i = 0; i <= maxwin; ++ i)
         {
         System.out.printf ("%d\t%d\n", i + 1, wincount[i]);
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice05 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

Our simulation programs are starting to get more **knobs.** A “knob” is a parameter that is not hard-coded into the program’s source code, but is specified by the experimenter when running the program. It is as though the simulator is a magical machine (Figure 3.1) with knobs you can adjust (by typing them on the command line); when you push the button (run the program), it spits out the results. You can run the simulator as many times as you want with whatever knob settings you want.

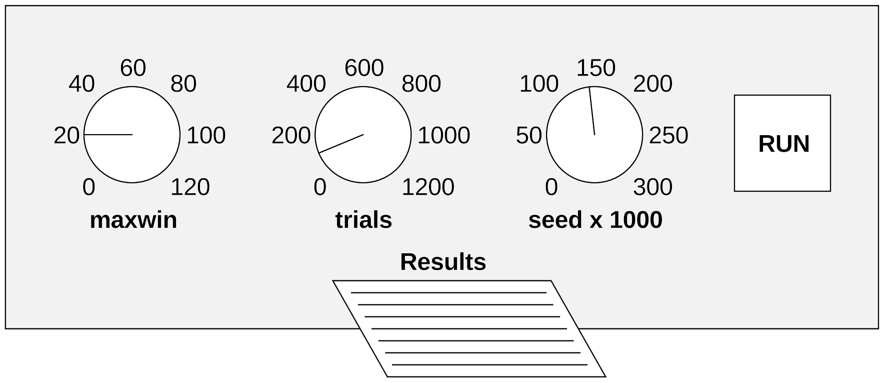


Figure 3.1: Dice game simulator machine

Pop Quiz: Why is the heads probability (5/6) hard-coded and not a knob?
Answer: It could be!

Here's the data we get when we do 100 trials with a `maxwin` of 20:

```
$ java Dice05 20 100 142857
Win     Count
1       19
2       6
3       9
4       8
5       6
6       8
7       11
8       6
9       6
10      3
11      3
12      3
13      2
14      1
15      3
16      0
```

```
17      0
18      1
19      0
20      0
21      5
```

Often, a **plot** displays the data better than a textual table. While you could take the above printout and plot the data by hand or feed the data into a separate plotting package, the Parallel Java 2 Library has classes that let you create plots right in your simulation program. Here's version 6 of the game simulation program. The first part is the same as version 5, except we use a `double` array for the counters (this is needed to plot the data).

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.numeric.plot.Plot;
import edu.rit.util.Random;
public class Dice06
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      double[] wincount = new double [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
         int win = 0;
         do
            {
            ++ win;
            }
         while (prng.next());
         wincount[Math.min (win - 1, maxwin)] += 1;
         }
```

Instead of doing a printout, we'll do a plot. We've already got the y coordinates of the points to be plotted (the `wincount` array). Now we need an array containing the x coordinates of the points to be plotted (the `winamount` array).

```
      double[] winamount = new double [maxwin + 1];
      for (int i = 0; i <= maxwin; ++ i) winamount[i] = i + 1;
```

Finally, we create a **Plot** object and call a bunch of methods on it. This is an instance of class edu.rit.numeric.plot.Plot:

```
      new Plot()
```

These methods set the plot title and the axis titles.

```
         .plotTitle ("Dice Game")
         .xAxisTitle ("Winnings")
         .yAxisTitle ("Occurrences")
```

This method adds an **X-Y series** to the plot. The first argument is an array of the x coordinates of the points to be plotted, the second argument is an array of the y coordinates.

```
         .xySeries (winamount, wincount)
```

These methods create a window (a "frame") for the plot and display the window (make it visible) on the screen.

```
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice06 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

Running the Dice06 program with the command

```
$ java Dice06 20 100 142857
```

causes a plot to appear in a window on the screen (Figure 3.2). The window has three menus:

- The **File** menu lets you save the plot as a PNG image file, as a PostScript file, or as a "drawing" file. The drawing file contains the plotted data and other settings. If you save the plot as a drawing in a file named, say, "`plot.dwg`", you can look at the plot later (without having to re-run the simulation program) by typing this command:

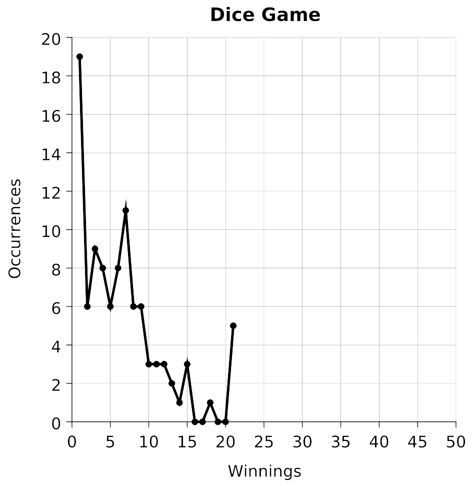


Figure 3.2: Dice06 program plot

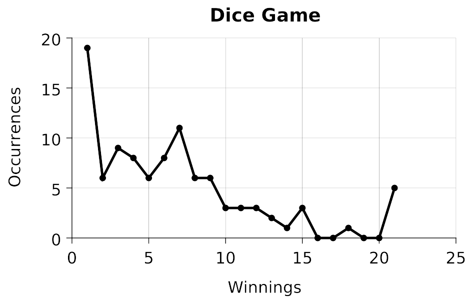


Figure 3.3: Dice06 program plot, adjusted

```
$ java View plot.dwg
```

- The **View** menu lets you zoom the plot in and out.
- The **Format** menu lets you adjust various attributes of the plot—the lengths of the axes, the axis titles, the grid lines, and so on. (You can also specify these by calling methods on the plot object.) Figure 3.3 shows the plot after making a few adjustments.

Chapter 4

Chi-Square Test

When using simulation to study a system, in some cases we have no idea beforehand how the system is supposed to behave, and we are using simulation to gain insight into the system's behavior. But in other cases, we do know how the system is supposed to behave; there is a **theoretical model** of the system. We then are interested in whether the simulation's results agree with the theoretical model.

Let's develop a theoretical model of the dice game.

- The probability of winning \$1 is the probability of getting a 6 on the first roll, 1/6.
- The probability of winning \$2 is the probability of getting 1 through 5 on the first roll and 6 on the second roll. Assuming rolls of the die are independent, this is $5/6 \cdot 1/6$.
- The probability of winning \$3 is the probability of getting 1 through 5 on the first roll, 1 through 5 on the second roll, and 6 on the third roll: $5/6 \cdot 5/6 \cdot 1/6$.

Generalizing, the probability of winning x dollars during one trial, $f(x)$, is

$$f(x) = p^{x-1}q \ , \ x \geq 1 \tag{4.1}$$

where p is the heads probability of the Bernoulli distribution and $q = 1 - p$ is the tails probability. For the dice game, $p = 5/6$ and $q = 1/6$.

We say that x is a **random variable** representing the winnings. Since x takes on integer values, it is a **discrete** random variable. (The other kind of random variable, a **continuous** random variable, takes on any real-number value in a continuous range.) $f(x)$ is the random variable's **probability function.** Table 4.1 lists $f(x)$ for x from 1 through 20 for the dice game.

What about the "everything else" category? If we're collecting data for winnings from 1 to m, the probability of "everything else" is the probability

Table 4.1: Dice game probability function

x	$f(x)$	x	$f(x)$
1	0.166667	11	0.026918
2	0.138889	12	0.022431
3	0.115741	13	0.018693
4	0.096451	14	0.015577
5	0.080376	15	0.012981
6	0.066980	16	0.010818
7	0.055816	17	0.009015
8	0.046514	18	0.007512
9	0.038761	19	0.006260
10	0.032301	20	0.005217

that the winnings will be $m+1$ or more, namely

$$\Pr[\text{everything else}] = \sum_{i=m+1}^{\infty} f(i) \ . \tag{4.2}$$

But because the probabilities of all possible values for the winnings must add up to 1,

$$\sum_{i=1}^{m} f(i) + \sum_{i=m+1}^{\infty} f(i) = 1 \ , \tag{4.3}$$

we see that

$$\Pr[\text{everything else}] = 1 - \sum_{i=1}^{m} f(i) \ . \tag{4.4}$$

In this example, the probability of “everything else” is 0.026084.

If we run the dice game simulation and do N trials, we expect to win \$1 in $N \cdot f(1)$ of the trials, to win \$2 in $N \cdot f(2)$ of the trials, and so on. In other words, the **expected count** in each category, or **bin,** is the number of trials times the probability of a trial falling in each bin. Of course, the bin probabilities must all add up to 1. As we run the simulation, we record the **actual count** in each bin. Table 4.2 lists the actual counts and the expected counts for the following simulation run with $N = 1000$ trials.

```
$ java Dice05 20 1000 142857
```

We see that none of the actual counts is precisely equal to the corresponding expected count—some are lower, some higher. Depending on the PRNG seed we put into the simulation, we would expect to see random variations in the actual counts from run to run, and we wouldn’t necessarily expect these to match the expected counts exactly. However, the actual counts shouldn’t be too far off the expected counts if the simulation really is behaving the way the theoretical model says it should. Are the results in Table 4.2 close enough, or not?

Table 4.2: Dice game simulation actual and expected counts

x	*Actual*	*Expected*	x	*Actual*	*Expected*
1	167	166.667	12	20	22.431
2	150	138.889	13	15	18.693
3	119	115.741	14	16	15.577
4	102	96.451	15	19	12.981
5	73	80.376	16	7	10.818
6	67	66.980	17	7	9.015
7	54	55.816	18	7	7.512
8	64	46.514	19	4	6.260
9	28	38.761	20	8	5.217
10	25	32.301	≥ 21	23	26.084
11	25	26.918			

We can answer this question by applying a **statistical test** to the data. The statistical test originates from a **hypothesis.** In this case, the hypothesis is that the random variable x, the dice game winnings, has the distribution whose probability function is given by Equation 4.1.

The appropriate statistical test to determine whether a *discrete* random variable obeys a certain probability distribution is the **chi-square test.** We compute the chi-square **statistic,** χ^2:

$$\chi^2 = \sum_{i=1}^{b} \frac{(N_i - Np_i)^2}{Np_i} \tag{4.5}$$

where b is the number of bins, N is the number of trials, N_i is the actual count in bin i, and p_i is the probability of a trial falling in bin i. Np_i is the expected count in bin i. For the data in Table 4.2, χ^2 is 21.678171.

Because χ^2 is computed from samples of the random variable x, χ^2 is itself a random variable with a certain probability distribution. If the hypothesis is true, that distribution is the **chi-square distribution** with $b - 1$ degrees of freedom (d.o.f.). χ^2 obeys the chi-square distribution exactly in the limit as N goes to infinity. If N is finite, χ^2 only approximately obeys the chi-square distribution. But if the expected count in each bin (Np_i) is 5 or greater, the approximation is acceptable.

If the actual counts all equaled the expected counts, χ^2 would be 0. As the actual counts start to deviate from the expected counts, χ^2 becomes greater than 0, and the more the deviation, the larger the χ^2. At some point, χ^2 becomes too large. To quantify how large is "too large," we compute the ***p*-value** of χ^2. The p-value is the probability that the statistic would have a value greater than or equal to the observed value by random chance, even if the hypothesis is true. For the chi-square distribution, the p-value is given by

$$\text{chi-square } p\text{-value} = 1 - \texttt{gammp}\left(\frac{\text{d.o.f.}}{2}, \frac{\chi^2}{2}\right) \ , \tag{4.6}$$

where `gammp` is the **incomplete gamma function,**

$$\texttt{gammp}(a,x) = \frac{\int_0^x t^{a-1}\mathrm{e}^{-t}\mathrm{d}t}{\int_0^\infty t^{a-1}\mathrm{e}^{-t}\mathrm{d}t}\ . \tag{4.7}$$

(You don't have to worry about evaluating the integrals. Most spreadsheets and mathematics packages, as well as the Parallel Java 2 Library, can calculate the chi-square p-value for you.)

If the hypothesis is true, then the p-value should not be too small. Therefore, if the p-value is too small, we conclude that the hypothesis is not true. Specifically, if the p-value falls below a certain **significance** threshold α chosen by the experimenter, we say that the chi-square test **fails** at a significance of α, and the hypothesis is disproven. Typical thresholds are $\alpha = 0.05$ or 0.01. If the p-value is greater than α, then the data is consistent with the hypothesis. However, we cannot say that the hypothesis is *proven;* the data might also be consistent with some other hypothesis that yields a p-value greater than α.

For a chi-square statistic of 21.678171 with 20 d.o.f. (b = 21 bins), the p-value is 0.358226. This shows that the data observed from the simulation is consistent with the theoretical model for the dice game.

Let's add a chi-square test to the dice game simulation program. We'll also print out the actual and expected counts and display them on the plot. Here is version 7:

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.numeric.Statistics;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
public class Dice07
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      double[] actual = new double [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
         int win = 0;
         do
            {
```

```
            ++ win;
            }
         while (prng.next());
         actual[Math.min (win - 1, maxwin)] += 1;
         }
      double[] winamount = new double [maxwin + 1];
      for (int i = 0; i <= maxwin; ++ i) winamount[i] = i + 1;
```

The above code, the same as in version 6, runs the simulation and accumulates the actual count in each bin (the `actual` array). For the chi-square test, we also need the expected count in each bin (the `expected` array).

```
      double[] expected = new double [maxwin + 1];
      double sum_p_i = 0.0;
      System.out.printf ("Win\tActual\tExpected%n");
      for (int i = 0; i < maxwin; ++ i)
         {
         double p_i = Math.pow (5.0/6.0, i)/6.0;
         sum_p_i += p_i;
         expected[i] = N*p_i;
         System.out.printf ("%.0f\t%.0f\t%.3f%n",
            winamount[i], actual[i], expected[i]);
         }
      expected[maxwin] = N*(1.0 - sum_p_i);
      System.out.printf ("%.0f\t%.0f\t%.3f%n",
         winamount[maxwin], actual[maxwin], expected[maxwin]);
```

Here is the code for the chi-square test. It calls two methods in class edu.rit.numeric.Statistics, one to calculate χ^2 from the actual and expected counts, the other to calculate the p-value from the d.o.f. and χ^2.

```
      double chisqr = Statistics.chiSquareTest (actual, expected);
      double pvalue = Statistics.chiSquarePvalue (maxwin, chisqr);
      System.out.printf ("Chi^2   = %.6f%n", chisqr);
      System.out.printf ("P-value = %.6f%n", pvalue);
```

This time we're going to put two data series on the plot: the actual counts (`actual`) and the expected counts (`expected`). Both will use the same x coordinate values (`winamount`).

```
      new Plot()
         .plotTitle ("Dice Game")
         .xAxisTitle ("Winnings")
         .yAxisTitle ("Occurrences")
```

When plotting the actual counts, we'll turn off the stroke (the line between the dots), leaving only the dots.

```
         .seriesStroke (null)
         .xySeries (winamount, actual)
```

When plotting the expected counts, we'll turn off the dots and turn the stroke back on.

```
         .seriesDots (null)
         .seriesStroke (Strokes.solid (1))
         .xySeries (winamount, expected)
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice07 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

Figure 4.1 shows the plot generated by the following simulation program run (after some manual tweaking).

```
$ java Dice07 20 1000 142857
Win     Actual  Expected
1       167     166.667
2       150     138.889
3       119     115.741
4       102     96.451
5       73      80.376
6       67      66.980
7       54      55.816
8       64      46.514
9       28      38.761
10      25      32.301
11      25      26.918
12      20      22.431
13      15      18.693
14      16      15.577
15      19      12.981
16      7       10.818
17      7       9.015
18      7       7.512
19      4       6.260
```

```
20      8       5.217
21      23      26.084
Chi^2   = 21.678171
P-value = 0.358226
```

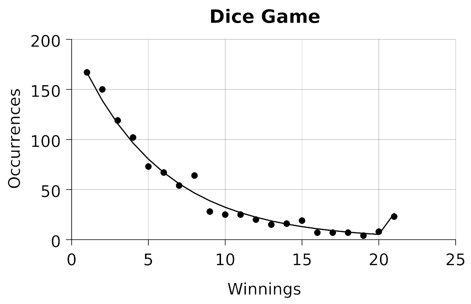


Figure 4.1: Dice07 program plot

Chapter 5

A Distributed Encyclopedia

Now let's build a simulation of a more substantial system: a **distributed encyclopedia.** The encyclopedia consists of a large number of **articles** that are stored in files. The articles (files) are dispersed among a large number of **nodes** (computers), each node holding a small number of articles. Each node also knows the Internet addresses of some of the other nodes. If node A knows the address of node B, we say B is **connected** to A.

To find a certain article, a user sends a **query** to one of the nodes. That originating node relays the query to its connected nodes, those nodes relay the query to their connected nodes, and eventually the query reaches the node that **matches** the query (the node that has the requested article). The matching node then sends the article directly back to the user's computer.

This is an example of a **peer-to-peer (P2P) system.** We will call it "P2Pedia."[1]

Whenever a query is transmitted from one node to another, that is one **hop.** (The transmission from the user's computer to the original node does not count.) We would like to know the average number of hops taken by a query. Specifically, we want to know the average length of (number of hops in) the **path** from the originating node to the matching node, over a large number of queries.

To simulate P2Pedia, we need to know several things: how the nodes are connected, how a query travels among the nodes, how likely it is for each node to originate a query, and how likely it is for each node to match a query.

P2Pedia's **topology**, the pattern of connections among its nodes, is as follows. There are N nodes. Each node has an ID in the range 0 to $N-1$. There is a connection from node 0 to node 1, from node 1 to node 2, . . ., and from node $N-1$ to node 0; these connections ensure that a query originating at any node will always eventually reach the matching node. In addition, each node has K additional connections to randomly chosen nodes (other than the immediate successor); these connections attempt to reduce the average query path length.

[1]There is a web site called P2Pedia, but it is not a distributed encyclopedia, nor is the name trademarked as far as I can tell, so I'll feel free to use the name for our system.

Figure 5.1 shows an example topology with $N = 12$ nodes and $K = 1$ additional connection per node. (Other P2P systems use other topologies.)

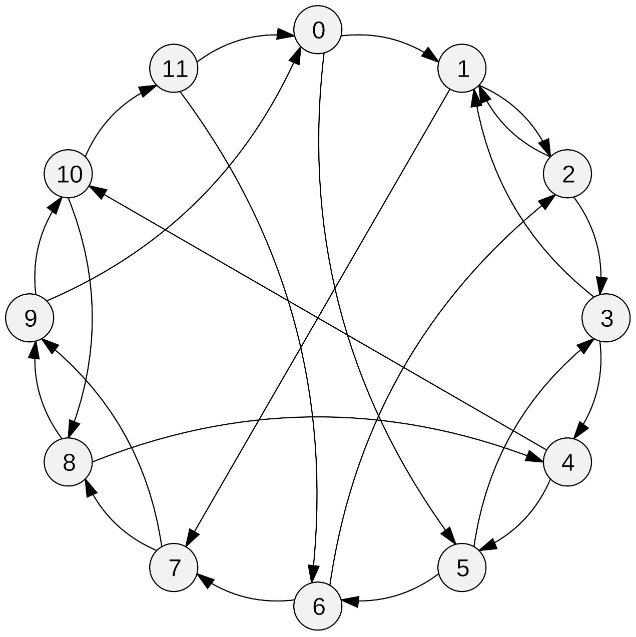


Figure 5.1: Example P2Pedia topology

P2Pedia's **query forwarding algorithm** is as follows. Upon receiving a query, a node X either (a) sends the article to the user, if X is the matching node; or (b) sends the query to each node connected to X, if X has not seen the query before; or (c) ignores the query, if X has seen the query before. This is known as a **flood routing** algorithm. (Other P2P systems use other query forwarding algorithms.)

For example, a query originating at node 5 and matching at node 10 is routed through the topology of Figure 5.1 as follows.

- First hop: 5 sends query to 3 and 6.
- Second hop: 3 sends query to 1 and 4. 6 sends query to 2 and 7.
- Third hop: 1 sends query to 2 and 7. 4 sends query to 5 and 10. 2 sends query to 1 and 3. 7 sends query to 8 and 9.

After three hops, the query has reached the matching node. This is shorter than the five-hop path the query would have to take if the "extra" connections were not present.

For this simulation, we will assume that for each query, each of the N nodes is equally likely to be the originating node, and each of the N nodes is equally likely to be the matching node. (If the matching node is the same as the originating node, the query path has 0 hops.)

Here is a pseudocode sketch of the P2Pedia simulation program for measuring the average query path length. N, K, and T are knobs we can use to explore P2Pedia's behavior.

```
$N \leftarrow$ number of nodes
$K \leftarrow$ number of extra connections per node
$T \leftarrow$ number of trials
Repeat $T$ times:
    Set up an $N$-node graph with a random P2Pedia topology
    Originating node $O \leftarrow \text{random}(0, N-1)$
    Matching node $M \leftarrow \text{random}(0, N-1)$
    Find shortest path from $O$ to $M$ in graph
    Record length of shortest path
Print average of path lengths
```

Rather than using the same topology for all the trials, at each trial we create a whole new random topology and pick different random originating and matching nodes. This ensures that the path lengths are averaged over many different topologies, originating nodes, and matching nodes, and so should be pertinent to any configuration and usage of the P2Pedia system.

We represent the P2Pedia topology as a **graph.** A graph consists of a set of **vertices,** or nodes, and a set of **edges**. Each edge is an ordered pair of vertices, representing a connection from one vertex to another vertex. Figure 5.1 depicts a graph, with the vertices drawn as circles and the edges drawn as arrows. This is a **directed graph,** or **digraph,** where each edge represents a connection in only one direction. (In another kind of graph, an **undirected graph,** each edge represents a connection in both directions.)

Note that the above program does *not* include all the gory detail in the actual P2Pedia system. The socket connections the nodes set up to communicate with each other; the format of the messages sent from node to node; the details of the flood routing algorithm; the criteria for determining whether a query matches an article stored on a node; the contents of the articles themselves—none of these things are present. The simulation program strips the P2Pedia system down to the bare minimum needed to answer the question we are studying, namely the average number of hops taken by a query.

At the same time, the simulation program *does* include knobs for parameters we suspect will affect the results. It seems reasonable to suspect that increasing the number of nodes N will increase the average query path length, and that increasing the number of extra connections K will decrease the average query path length. What's not clear is *exactly how* the results will be affected by the knob settings. Running the simulation and tweaking the knobs will give us insight into these relationships.

Much of the art of designing simulation programs lies in knowing which parts of the actual system to include in the program and, more importantly, to omit from the program, and in knowing which parameters might affect the results and so should be made knobs.

Before building the P2Pedia simulation program, we need to build a few tools. We need a data structure to store a graph. We need a way to set up edges from a given vertex to K other randomly chosen vertices. We need a way to find the shortest path from one vertex to another in a graph. Those will be the topics of the next three chapters. As we develop these tools, we'll incorporate them into the evolving simulation program.

Chapter 6

Graphs

Some data structures have simple and straightforward application programming interfaces (APIs). For a list, you can add elements, remove elements, and traverse (iterate) over the elements. For a set, you can add, remove, and traverse elements, plus query whether an element is in the set. For a map, you can add, remove, and traverse key-value pairs, plus query whether a key is in the map and find the value associated with a key.

A graph is a more complicated data structure than a list, set, or map. Graphs are put to a broader variety of uses than lists, sets, or maps. Graph algorithms are more numerous and more complex than list, set, or map algorithms. A general-purpose graph API, one that could support every possible usage of a graph, would be quite lengthy, and any one program would only utilize a fraction of the API.

In addition, there are many different ways to implement a graph, each with pros and cons. Two widely used implementations are the adjacency matrix and the adjacency list.

Adjacency matrix. If there is an edge from vertex A to vertex B, we say that B is **adjacent** to A. We can store the adjacency relationships in a two-dimensional matrix of Boolean values. The first index of the matrix is the source vertex; the second index is the destination vertex. Each matrix element is true if there is an edge from the source vertex to the destination vertex, false if there isn't. Determining whether a certain edge exists is fast, as we just index into the matrix. Traversing the vertices adjacent to a given vertex is slow, as we have to scan all the elements in a matrix row. The adjacency matrix requires $O(N^2)$ storage, where N is the number of vertices. This is fine for a **dense** graph, one where a significant fraction of the possible edges are present. However, this wastes storage for a **sparse** graph, where only a few edges are present.

Adjacency list. The adjacency relationships are stored in an array indexed by the source vertex. Each array element is a list of the destination vertices adjacent to the source vertex. Traversing the vertices adjacent to a given vertex is fast, as we scan only the elements in the source vertex's adjacency list. Determining whether a certain edge exists is slow, as we have to scan the source vertex's entire adjacency list. The adjacency list requires $O(N + E)$ storage, where N is the number of vertices and E is the number of edges. This is fine for a sparse graph, but for a dense graph this may require more storage than an adjacency matrix.

Other graph implementations can be envisioned, with different speed-storage tradeoffs. For example, a variation on an adjacency list would be a map from source vertex to a set of destination vertices.

Some graph algorithms associate additional information with the vertices and/or the edges. For example, a graph might be used to represent cities (vertices) and roads between the cities (edges). Associated with each edge is the distance between the two cities along the road. A **shortest path** algorithm uses the graph topology and the edge distances to find the shortest route from one city to another. Other graph algorithms do not need to associate additional information with the vertices or the edges.

For all these reasons, it is difficult to justify the effort to write a reusable software component for a graph—like the list, set, and map components in the Java Collections Framework. It's better to tailor the graph implementation for the problem at hand. That's what we'll do for the P2Pedia simulation program.

We'll assume that the P2Pedia topology is *sparse.* There may be many nodes, but there will be only a few connections from each node. This suggests an adjacency list representation would be appropriate. However, variable-length lists are overkill, because we know ahead of time exactly how many nodes are connected to each node: one immediate successor node plus K other nodes. So each node's adjacency list can be a fixed-size $(K + 1)$-element array. The whole graph ends up being an array of arrays, the first index being the source node from 0 to $N - 1$, the second index going from 0 to K. Figure 6.1 shows the data structure for the P2Pedia topology in Figure 5.1.

	0	1
0	1	5
1	2	7
2	3	1
3	4	1
4	5	10
5	6	3
6	7	2
7	8	9
8	9	4
9	10	0
10	11	8
11	0	6

Figure 6.1: Example P2Pedia graph data structure

Now we can begin writing the P2Pedia simulation program.

```
import edu.rit.util.Random;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
```

Here we allocate the matrix for storing the node connections (`conn`). We'll allocate it once and reuse the same matrix for each trial. This works because the matrix's dimensions are fixed. Reusing the matrix reduces the program's running time—we don't have to re-allocate the matrix, which also causes its elements to be initialized to the default value of 0, on every trial. The time savings might be negligible if there were ten nodes. But if there were a million nodes, and a thousand trials, the time savings would be substantial.

```
      int[][] conn = new int[N][K+1];
```

Furthermore, each node is always connected to its successor. Only the extra connections are different from trial to trial. So we can fill in those fixed connections just once as well.

```
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
```

We'll add up the trials' path lengths in `sumPathLength`. At the end of the program, we'll divide that by the number of trials to get the average path length.

```
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
```

At each trial, we fill in each node's extra connections at random. We're not ready to write the code for that, though.

```
         for (int i = 0; i < N; ++ i)
            {
            for (int j = 1; j <= K; ++ j)
               {
               // conn[i][j] = random node
               }
            }
```

We do know how to pick random originating and matching nodes.

```
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         sumPathLength +=
            shortestPathLength (conn, orig, match);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }
```

We'll put the code to find the shortest path in a subroutine so as not to clutter up the main program. For now, we'll stub out the subroutine.

```
   private static int shortestPathLength
      (int[][] conn,
       int orig,
       int match)
      {
      return 0; // STUB
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

In its present form, the program compiles and runs, but it doesn't produce any actual results.

```
$ java P2Pedia01 1000 10 1000 142857
Average query path length = 0.00
```

Chapter 7

Random Subsets and Permutations

To generate a P2Pedia topology, the simulation program has to come up with a **random subset** of the set of nodes. Specifically, for node i, the program needs to choose at random a subset of K nodes from the set $\{0, \ldots, i-1, i+2, \ldots, N-1\}$—the set of all nodes, minus node i itself and node i's immediate successor. Each node must be equally likely to be chosen for the random subset. The same node cannot appear in the subset more than once. The nodes in the subset are then connected to node i, along with node $i+1$ which is always connected to node i.

Choosing a random subset of a set happens frequently in simulations, so it is worth some effort to develop a reusable software component to do it. We will restrict the component to generating a random subset of the N integers from 0 to $N-1$.

Here is one, not particularly good, way to generate a random subset S.

```
$S \leftarrow \{\,\}$
While size($S$) $< K$:
    Repeat:
        $x \leftarrow$ random($0, N-1$)
    Until $x$ is not in $S$
    Add $x$ to $S$
```

This is not a good method because of the open-ended inner loop, which might have to repeat many times if it keeps choosing random numbers that are already in the subset. Indeed, as the subset grows larger, the number of inner loop iterations tends to grow larger as well.

For a better way to generate a random subset, let's first look at an algorithm to generate a **random permutation** of the integers 0 to $N-1$. The permutation is stored in an N-element array P.

```
$P \leftarrow (0, 1, \ldots, N-1)$
For $i$ from 0 to $N-2$:
    Swap $P[i]$ with $P[i + \text{random}(N-i)]$
```

"random(x)" generates a random number from 0 to $x-1$. At iteration i, when the first i elements of P have already been chosen, the algorithm chooses one of the remaining $N-i$ elements of P at random, then swaps the chosen element into position i. This algorithm consumes precisely one random number for each element of the permutation. Figure 7.1 shows this algorithm in action generating a random permutation of the integers 0 to 5.

	P					
Initially:	0	1	2	3	4	5
i=0, random(6)=2, swap $P[0]$ with $P[2]$:	2	1	0	3	4	5
i=1, random(5)=2, swap $P[1]$ with $P[3]$:	2	3	0	1	4	5
i=2, random(4)=3, swap $P[2]$ with $P[5]$:	2	3	5	1	4	0
i=3, random(3)=0, swap $P[3]$ with $P[3]$:	2	3	5	1	4	0
i=4, random(2)=1, swap $P[4]$ with $P[5]$:	2	3	5	1	0	4

Figure 7.1: Generating a random permutation

Now that we can generate a random *permutation* of the integers 0 to $N-1$, we can easily see how to generate a K-element random *subset* of the integers 0 to $N-1$: just take the first K elements of the permutation. The algorithm is the same, except i goes from 0 to $K-1$. The subset's elements are not guaranteed to appear in any particular order, but the P2Pedia simulation program doesn't mind. (If it matters, you can sort the subset.)

There's a catch, however. This random subset algorithm requires $O(N)$ storage to hold the array P. That's okay if N is small. That's also okay if N is large and K is large. But if N is large and K is small—say, if we want a 10-element random subset of the integers from 0 to 999999—then this algorithm wastes storage.

To fix that problem, we can use a **sparse array** implementation for P. We will represent P as a map of (key, value) pairs, where the key is the array index and the value is the array element at that index. Only those pairs where the array element does not equal the array index appear in the map (and consume storage). If a certain array index does not appear as a key in the map, it means the corresponding array element is the same as the array index. Initially, the map is empty, meaning every array element is equal to its index—the initial state of P.

For the random subset component's API, we will use Java's **iterator** interface. Each time the iterator's `next()` method is called, it returns the next element in the random subset. Here is class edu.rit.util.RandomSubset in the

Parallel Java 2 Library.

```
package edu.rit.util;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
public class RandomSubset
   implements Iterator<Integer>
   {
   // The underlying PRNG.
   private Random prng;

   // The size of the original set.
   private int N;

   // The number of random subset elements returned so far.
   private int M;

   // A sparse array containing a permutation of the integers
   // from 0 to N-1. Implemented as a mapping from array index to
   // array element. If an array index is not in the map, the
   // corresponding array element is the same as the array index.
   private HashMap<Integer,Integer> permutation =
      new HashMap<Integer,Integer>();
```

Here are four hidden methods for manipulating the sparse array. These will be used by the public methods.

```
   // Returns the element in the permutation array at index i.
   private int getElement
      (int i)
      {
      Integer element = permutation.get (i);
      return element == null ? i : element;
      }

   // Sets the element in the permutation array at index i to the
   // given value.
   private void setElement
      (int i,
       int value)
      {
      if (value == i)
         {
         permutation.remove (i);
         }
      else
         {
         permutation.put (i, value);
         }
      }

   // Swaps the elements in the permutation array at indexes i
   // and j.
   private void swapElements
      (int i,
       int j)
      {
      int tmp = getElement (i);
      setElement (i, getElement (j));
      setElement (j, tmp);
      }

   // Returns the index in the permutation array at which the
   // given value resides.
   private int indexOf
      (int value)
      {
      for (Map.Entry<Integer,Integer> e : permutation.entrySet())
         {
         if (e.getValue() == value) return e.getKey();
         }
      return value;
      }
```

To construct a random subset object, we must specify a PRNG object from which to draw random numbers, as well as the size of the original set N.

```
   public RandomSubset
      (Random prng,
       int N)
      {
      if (prng == null)
         {
         throw new NullPointerException
            ("RandomSubset(): prng is null");
         }
      if (N < 0)
         {
         throw new IllegalArgumentException
            ("RandomSubset(): N = "+N+" illegal");
         }
      this.prng = prng;
      this.N = N;
      }
```

The `hasNext()` method, part of the java.util.Iterator interface, returns true if there are more integers in the random subset, false if all the integers in the original set have been used up.

```
  public boolean hasNext()
    {
    return M < N;
    }
```

The `next()` method, part of the java.util.Iterator interface, performs the next iteration of the random subset algorithm, then returns the element that was chosen.

```
  public Integer next()
    {
    if (M >= N)
      {
      throw new NoSuchElementException
        ("RandomSubset.next(): No further elements");
      }
    swapElements (M, M + prng.nextInt (N - M));
    ++ M;
    return getElement (M - 1);
    }
```

The `remove()` method is part of the java.util.Iterator interface and has to be implemented. It is supposed to remove, from the collection being iterated over, the element that the previous `next()` method call returned. This makes no sense for the random subset object, so we make `remove()` an unsupported operation.

```
  public void remove()
    {
    throw new UnsupportedOperationException();
    }
```

The P2Pedia simulation program needs, and it is useful in general to have, a method to remove a specified element from the original set. That element will then never be chosen as part of the random subset. We do this by swapping out the specified element, wherever it might be in the array, rather than a randomly chosen element. (The `remove(i)` method is not part of the java.util.Iterator interface.)

```
   public RandomSubset remove
      (int i)
      {
      if (0 > i || i >= N)
         {
         throw new IllegalArgumentException
            ("RandomSubset.remove(): i = "+i+" illegal");
         }
      int j = indexOf (i);
      if (j >= M)
         {
         swapElements (M, j);
         ++ M;
         }
      return this;
      }
   }
```

Armed with class edu.rit.util.RandomSubset, we can fill in one of the missing pieces in the P2Pedia simulation program.

```
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
```

Now we can pick K extra connections for each node, chosen at random from the set of nodes 0 through $N-1$, minus nodes i and $i+1$.

```
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
```

Pop Quiz: Why did we declare class RandomSubset's `remove(i)` method to return the random subset object itself, instead of `void`?
Answer: To make the above coding idiom possible: Call a constructor to create an object, call multiple methods on the new object, and store a reference to the created and modified object, all in a single statement.

```
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         sumPathLength +=
            shortestPathLength (conn, orig, match);
```

The program still doesn't find any query paths, but we can at least print out the graph to see what kind of random topologies the program is generating.

```
         System.out.printf ("Trial %d%n", trial);
         System.out.printf ("  Node  Connected to nodes%n");
         for (int i = 0; i < N; ++ i)
            {
            System.out.printf ("  %-4d", i);
            for (int j = 0; j <= K; ++ j)
               {
               System.out.printf ("  %d", conn[i][j]);
               }
            System.out.println();
            }
         System.out.printf ("  Originating node = %d%n", orig);
         System.out.printf ("  Matching node = %d%n", match);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }

   private static int shortestPathLength
      (int[][] conn,
       int orig,
       int match)
      {
      return 0; // STUB
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

Here are a couple of random P2Pedia topologies with $N = 20$ nodes and $K = 3$ extra connections per node.

```
$ java P2Pedia01 20 3 2 142857
Trial 0
  Node  Connected to nodes
  0     1  8  10  18
  1     2  16  11  0
  2     3  13  1  15
  3     4  5  17  13
  4     5  6  2  1
  5     6  9  16  18
  6     7  19  4  17
  7     8  4  16  3
  8     9  19  2  17
  9     10  0  16  3
  10    11  16  12  19
  11    12  17  18  9
  12    13  16  5  9
  13    14  2  10  12
  14    15  16  12  7
  15    16  13  10  9
  16    17  7  18  11
  17    18  2  15  13
  18    19  7  2  3
  19    0  1  18  11
  Originating node = 1
  Matching node = 3
Trial 1
  Node  Connected to nodes
  0     1  8  19  16
  1     2  7  12  19
  2     3  1  9  7
  3     4  1  9  11
  4     5  19  1  17
  5     6  13  15  16
  6     7  11  10  12
  7     8  16  12  0
  8     9  0  14  13
  9     10  11  12  0
  10    11  3  13  16
  11    12  10  9  14
  12    13  4  5  15
  13    14  1  12  11
  14    15  19  8  7
  15    16  10  8  0
  16    17  12  3  0
  17    18  19  8  11
  18    19  17  5  2
  19    0  14  5  4
  Originating node = 6
  Matching node = 12
Average query path length = 0.00
```

Chapter 8

Breadth-First Search

The last piece needed for the P2Pedia simulation program is an algorithm to find the shortest path through the graph from the originating node to the matching node. To develop the algorithm, we'll mimic the flood routing of the query messages through the P2Pedia topology, recording the distance (number of hops) from the originating node of each node we encounter. When we reach the matching node, we stop and return the matching node's distance, which is the query path length.

To process the nodes in the order in which the query messages reach them, we must maintain a **queue** of nodes that have received the query but have not yet sent the query. The algorithm is:

```
If originating node = matching node:
    Return 0
For $i = 0$ to $N - 1$:
    distance($i$) $\leftarrow \infty$
distance(originating node) $\leftarrow 0$
Append originating node to queue
Repeat:
    Node $i \leftarrow$ remove first node from queue
    For each node $j$ connected to node $i$:
        If node $j$ = matching node:
            Return distance($i$) $+ 1$
        Else if distance($j$) $= \infty$:
            distance($j$) $\leftarrow$ distance($i$) $+ 1$
            Append node $j$ to queue
```

Note how if a node has already seen a query (the node's distance has been set to other than ∞), the node ignores the query (the node is not appended to the queue).

The above algorithm visits the nodes in **breadth-first** order: first the originating node, then the nodes connected to the originating node, then the nodes connected to the nodes connected to the originating node, and so on. Thus, it is doing a **breadth-first search** for the matching node. As a side effect, it is counting the number of hops taken. The breadth-first order ensures that the matching node will be reached with the fewest possible number of hops.

Finally, we can finish the P2Pedia simulation program.

```
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
```

As we did with the connection matrix (`conn`), we allocate an array to hold each node's distance from the originating node (`dist`), and we'll reuse that array in each trial.

```
      int[] dist = new int[N];
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         int len = shortestPathLength (conn, dist, orig, match);
         sumPathLength += len;
         System.out.printf ("Trial %d%n", trial);
         System.out.printf ("  Node  Connected to nodes%n");
         for (int i = 0; i < N; ++ i)
            {
```

```
            System.out.printf ("  %-4d", i);
            for (int j = 0; j <= K; ++ j)
               {
               System.out.printf ("  %d", conn[i][j]);
               }
            System.out.println();
            }
         System.out.printf ("  Originating node = %d%n", orig);
         System.out.printf ("  Matching node = %d%n", match);
         System.out.printf ("  Query path length = %d%n", len);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }
```

The breadth-first search algorithm goes in the `shortestPathLength()` method. It is a straightforward translation of the previous pseudocode into Java. A distance of -1 stands for ∞. We'll use class java.util.LinkedList for the queue of nodes.

```
   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

Here are the same two random P2Pedia topologies as in Chapter 7, with $N = 20$ nodes and $K = 3$ extra connections per node, including the query path lengths.

```
$ java P2Pedia01 20 3 2 142857
Trial 0
  Node  Connected to nodes
  0     1  8  10  18
  1     2  16  11  0
  2     3  13  1  15
  3     4  5  17  13
  4     5  6  2  1
  5     6  9  16  18
  6     7  19  4  17
  7     8  4  16  3
  8     9  19  2  17
  9     10  0  16  3
  10    11  16  12  19
  11    12  17  18  9
  12    13  16  5  9
  13    14  2  10  12
  14    15  16  12  7
  15    16  13  10  9
  16    17  7  18  11
  17    18  2  15  13
  18    19  7  2  3
  19    0  1  18  11
  Originating node = 1
  Matching node = 3
  Query path length = 2
Trial 1
  Node  Connected to nodes
  0     1  8  19  16
  1     2  7  12  19
  2     3  1  9  7
  3     4  1  9  11
  4     5  19  1  17
  5     6  13  15  16
  6     7  11  10  12
  7     8  16  12  0
  8     9  0  14  13
  9     10  11  12  0
  10    11  3  13  16
  11    12  10  9  14
  12    13  4  5  15
  13    14  1  12  11
  14    15  19  8  7
  15    16  10  8  0
  16    17  12  3  0
  17    18  19  8  11
  18    19  17  5  2
  19    0  14  5  4
  Originating node = 6
  Matching node = 12
  Query path length = 1
Average query path length = 1.50
```

Here's what we get with $N = 20$ nodes, $K = 3$ extra connections per node, and increasing numbers of trials (after removing the debugging printouts). Once we get up to 1000 trials or so, the answer doesn't vary much.

```
$ java P2Pedia01 20 3 10 142857
Average query path length = 1.90
$ java P2Pedia01 20 3 100 142857
Average query path length = 1.97
$ java P2Pedia01 20 3 1000 142857
Average query path length = 1.99
$ java P2Pedia01 20 3 10000 142857
Average query path length = 2.00
$ java P2Pedia01 20 3 100000 142857
Average query path length = 2.01
$ java P2Pedia01 20 3 1000000 142857
Average query path length = 2.00
```

Chapter 9

Mean, Median, and the Like

In the simulation programs we've developed so far, we've written the code multiple times to calculate the average of a series of values—sum up the values, divide by the number of values. This strongly suggests that we should be employing a reusable software component to calculate the average. Furthermore, there are other **statistics** of a series of values that are often of interest, which a software component could calculate. Let n be the number of values and let x_i, $1 \le i \le n$, be the i-th value; then these statistics are:

- **Mean** (or average) $\mu = \dfrac{1}{n}\displaystyle\sum_{i=1}^{n} x_i$
- **Variance** $\sigma^2 = \dfrac{1}{n-1}\displaystyle\sum_{i=1}^{n}(x_i - \mu)^2$
- **Standard deviation** $\sigma = \sqrt{\sigma^2}$

If the values in the series are well-approximated by a normal (or Gaussian) distribution, then the peak of the distribution falls at μ, and 68.3 percent of the values fall within $\pm\sigma$ of μ. However, the values might not be well-approximated by a normal distribution. The distribution might be narrower or broader than a normal distribution, or it might be skewed to one side or the other, or it might have multiple peaks. In that case we might be interested in the **quantiles** of the series. The q-th quantile, where $0.0 \le q \le 1.0$, is that value x such that a fraction q of the values in the series are less than or equal to x. To find a quantile q of a series of length n, simply sort the values into ascending order and take the value at index $q \cdot n$.

Rather than using the mean to express the central value of the series' distribution, we can use the **median,** which is the 0.5 quantile—half the values in the series are at or below the median, half are above the median. Rather than using the standard deviation to express the width of the series' distribution, we can use a **confidence interval.** For example, the 90-percent confidence interval is defined by values a and b such that 90 percent of the values in the series fall between a and b. The 90-percent confidence interval bounds are just the 0.05 quantile a and the 0.95 quantile b; 95 percent of the values fall below b and 5 percent of the values fall below a, leaving 90 percent of the values between a and b.

Here is version 2 of the P2Pedia simulation program. It uses class edu.rit-.numeric.ListSeries in the Parallel Java 2 Library to calculate and print the aforementioned statistics for the query path length.

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.Series;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia02
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int[] dist = new int[N];
```

This time we'll use an instance of class ListSeries to accumulate the series of query path lengths measured in all the trials.

```
      ListSeries len = new ListSeries();
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
```

Here we append the measured path length to the query path length series.

```
         len.add (shortestPathLength (conn, dist, orig, match));
         }
```

When the trials are finished, `len` contains all the query path length values, and we can print their statistics. The mean, variance, and standard deviation are stored in an instance of class edu.rit.numeric.Series.Stats that is obtained from the query path length series object.

```
      System.out.printf ("Query path length:%n");
      Series.Stats stats = len.stats();
      System.out.printf ("Mean   = %.2f%n", stats.mean);
      System.out.printf ("Stddev = %.2f%n", stats.stddev);
```

The quantiles are stored in an instance of class edu.rit.numeric.Series.Robust-Stats that is obtained from the query path length series object. Although the series object holds values of type `double`, all the values in the series are in fact integers; so the median and confidence interval bounds are integers, and we print them as such (without any decimal fraction).

```
      Series.RobustStats rstats = len.robustStats();
      System.out.printf ("Median = %.0f%n", rstats.median);
      System.out.printf ("90%% ci = %.0f .. %.0f%n",
         rstats.quantile (0.05), rstats.quantile (0.95));
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia02 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

Here's what we get with $N = 100$, 200, 500, and 1000 nodes, $K = 1$ extra connection per node, and $T = 1000$ trials.

```
$ java P2Pedia02 100 1 1000 142857
Query path length:
Mean   = 5.65
Stddev = 2.07
Median = 6
90% ci = 2 .. 9
$ java P2Pedia02 200 1 1000 142857
Query path length:
Mean   = 6.76
Stddev = 2.10
Median = 7
90% ci = 3 .. 10
$ java P2Pedia02 500 1 1000 142857
Query path length:
Mean   = 8.08
Stddev = 2.21
Median = 8
90% ci = 4 .. 12
$ java P2Pedia02 1000 1 1000 142857
Query path length:
Mean   = 9.08
Stddev = 2.18
Median = 9
90% ci = 5 .. 13
```

To get a better sense of the shape of the query path length's probability distribution, we can plot a **histogram** of the distribution. The histogram partitions the range of possible query path lengths into a number of **bins,** or counters. For each trial, the bin corresponding to the measured path length is incremented. Afterwards, we plot the bin counts versus the query path lengths. If we do enough trials, the bin counts should approximate the shape of the query path length's distribution.

In Chapter 3, we made a histogram plot of the winnings in the dice game. There, however, we determined the bin boundaries beforehand, and we allocated and incremented the bin counters ourselves. Alternatively, class edu.rit.numeric.Series.RobustStats can do the work for us. It can determine histogram bin counts for a data series, with arbitrary bin boundaries, and with the bin boundaries chosen after all the data has been collected.

Here is version 3 of the P2Pedia simulation program. It is the same as version 2, except we add the following code to generate and plot the query path length histogram immediately after the code to print the statistics.

```
      . . .
      Series.RobustStats rstats = len.robustStats();
      System.out.printf ("Median = %.0f%n", rstats.median);
      int a = (int) rstats.quantile (0.05);
      int b = (int) rstats.quantile (0.95);
      System.out.printf ("90%% ci = %d .. %d%n", a, b);
```

First we create an **X-Y series** object to hold the (x, y) points to be plotted for the histogram; this is an instance of class edu.rit.numeric.ListXYSeries.

```
      ListXYSeries hist = new ListXYSeries();
```

The histogram bins will be for query path lengths of 0, 1, 2, and so on. The method call "`rstats.histogram (i, i + 1);`" returns the bin count for the histogram bin consisting of all values greater than or equal to i and less than $i + 1$; because all the values in the query path length series are integers, this is the number of values equal to i.

```
      for (int i = 0; i <= b + a; ++ i)
         {
         hist.add (i, rstats.histogram (i, i + 1));
         }
```

Finally, we plot the histogram.

```
      new Plot()
         .plotTitle ("P2Pedia03, N="+N+", K="+K+", T="+T)
         .xAxisTitle ("Query path length")
         .yAxisTitle ("Occurrences")
         .xySeries (hist)
         .getFrame()
         .setVisible (true);
      }
      . . .
```

Figure 9.1 shows the query path length histograms for the four experiments from earlier in the chapter. The histograms resemble normal distributions (bell-shaped curves). For comparison, Figure 9.2 superimposes on each histogram a plot of the normal distribution with the same mean and standard deviation as the query path length data. (The program that generated these plots, class P2Pedia03a, is listed in Appendix A.) The formula for the normal distribution's probability density function is

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \tag{9.1}$$

where μ is the mean and σ is the standard deviation. Roughly speaking, $f(x)$ is the probability that one trial will yield the value x. With T trials, the count in bin x should be $T \cdot f(x)$; this is what's plotted in Figure 9.2.

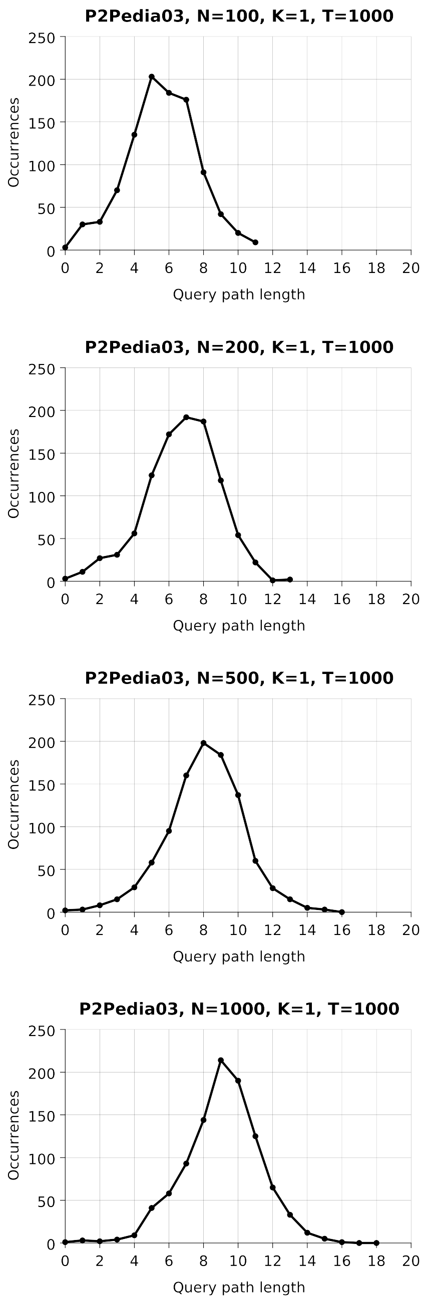


Figure 9.1: P2Pedia03 query path length histograms

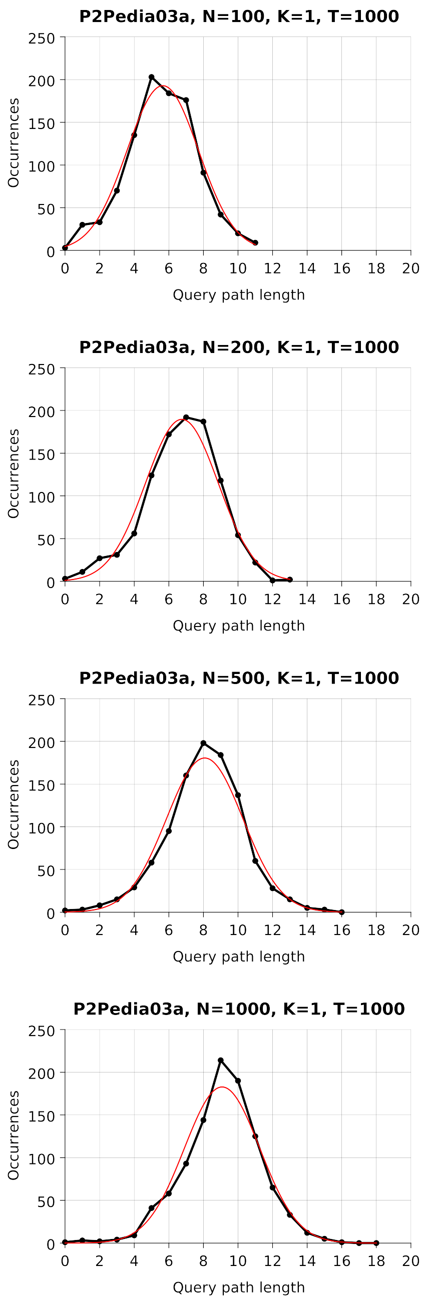


Figure 9.2: P2Pedia03 query path length histograms compared to normal distributions

Chapter 10

Knob Turning

Now that we can run P2Pedia simulation experiments, measure the query path lengths over many trials, and determine the query path length's mean, standard deviation, and other statistics, it's time for some deeper exploration. We'll begin by investigating the relationship between mean query path length and N, the number of nodes in the system. Figure 9.1 shows that, as N increases, the mean query path length also increases (the bell curve shifts to the right). But what is the relationship precisely? Can we express the mean query path length as some function of N?

To answer this question, we must "turn the knob." We must do multiple simulation runs, each with a different setting for the N knob. In each run, we must do multiple trials to determine the mean query path length for a certain value of N. We can then plot mean query path length versus N and see if we can gain any insight into their relationship.

Of course, the mean query path length also depends on K, the number of extra connections per node. It's best to investigate the effect of only one knob at a time. So in our experiments for now, we'll hold K constant at one extra connection per node; later we'll go back and investigate how K affects the mean query path length. Even though K will be constant, we'll still make it a knob (a command line argument) rather than hard-coding it in the simulation program.

As we have done before, we will let the simulation program itself do multiple runs with different settings for N. Instead of one knob, we'll provide three: the number of nodes lower bound, N_L; the number of nodes upper bound, N_U; and the number of nodes delta, N_Δ. The program will do simulation runs with N set to $N_L, N_L + N_\Delta, N_L + 2N_\Delta, N_L + 3N_\Delta, \ldots, N_U$. The other knobs are K, the number of extra connections per node; T, the number of trials per simulation run; and the pseudorandom number generator seed.

Here is version 4 of the P2Pedia simulation program.

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia04
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 6) usage();
      int N_L = Integer.parseInt (args[0]);
      int N_U = Integer.parseInt (args[1]);
      int N_D = Integer.parseInt (args[2]);
      int K = Integer.parseInt (args[3]);
      int T = Integer.parseInt (args[4]);
      long seed = Long.parseLong (args[5]);
      Random prng = new Random (seed);
```

We will store, in the X-Y series object `mqpl`, a series of (x, y) points to be plotted, where x is the number of nodes and y is the mean query path length. We'll also print the query path length means and standard deviations.

```
      ListXYSeries mqpl = new ListXYSeries();
      System.out.printf ("\tQuery path length%n");
      System.out.printf ("N\tMean\tStddev%n");
```

Here is the outer loop where we turn the knob for N, the number of nodes. The body of this outer loop is the same as one run of the whole P2Pedia03 simulation program.

```
      for (int N = N_L; N <= N_U; N += N_D)
         {
         int[][] conn = new int[N][K+1];
         for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
         int[] dist = new int[N];
         ListSeries len = new ListSeries();
         for (int trial = 0; trial < T; ++ trial)
            {
            for (int i = 0; i < N; ++ i)
               {
               RandomSubset rs = new RandomSubset (prng, N)
                  .remove (i) .remove ((i + 1)%N);
               for (int j = 1; j <= K; ++ j)
```

```
                  conn[i][j] = rs.next();
               }
            int orig = prng.nextInt (N);
            int match = prng.nextInt (N);
            len.add (shortestPathLength
               (conn, dist, orig, match));
            }
```

At the end of each simulation run, we print the query path length statistics and add the point (x, y) = (number of nodes, mean query path length) to the X-Y series for the plot.

```
         Series.Stats stats = len.stats();
         System.out.printf ("%d\t%.2f\t%.2f%n",
            N, stats.mean, stats.stddev);
         mqpl.add (N, stats.mean);
         }
```

After all the simulation runs, we display the plot.

```
      new Plot()
         .plotTitle ("P2Pedia04, K="+K+", T="+T)
         .xAxisTitle ("Number of nodes, N")
         .yAxisTitle ("Mean query path length")
         .xySeries (mqpl)
         .getFrame()
         .setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
```

```
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia04 <N_L> <N_U> <N_D> <K> <T> "+
          "<seed>");
      System.exit (1);
      }
   }
```

Here's what the program prints with the number of nodes N going from 50 to 2000 in steps of 50, with $K = 1$ extra connection per node, and with $T = 1000$ trials per simulation run. Figure 10.1 shows the plot of mean query path length versus N.

```
$ java P2Pedia04 50 2000 50 1 1000 142857
        Query path length
N       Mean    Stddev
50      4.73    2.00
100     5.66    2.12
150     6.38    2.06
200     6.70    2.13
250     6.97    2.23
300     7.37    2.16
350     7.46    2.22
400     7.71    2.22
450     8.05    2.12
500     8.21    2.19
550     8.24    2.28
600     8.43    2.20
650     8.39    2.26
700     8.39    2.25
750     8.55    2.25
800     8.61    2.25
```

```
850     8.76    2.22
900     8.95    2.16
950     8.93    2.26
1000    9.03    2.11
1050    9.13    2.20
1100    9.23    2.26
1150    9.24    2.42
1200    9.15    2.29
1250    9.36    2.17
1300    9.42    2.24
1350    9.54    2.16
1400    9.49    2.25
1450    9.56    2.26
1500    9.60    2.31
1550    9.71    2.30
1600    9.65    2.26
1650    9.72    2.29
1700    9.86    2.14
1750    9.86    2.28
1800    9.96    2.20
1850    9.91    2.07
1900    10.12   2.28
1950    9.96    2.35
2000    10.20   2.12
```

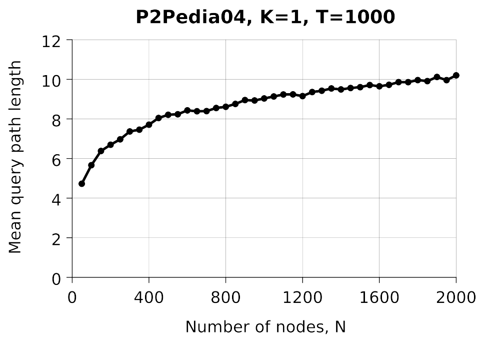


Figure 10.1: P2Pedia04 mean query path length versus number of nodes N

The plot shows that the relationship is not a linear one. As the number of nodes increases linearly, the mean query path length does *not* increase linearly, but increases at a seemingly decreasing rate. This suggests that the mean query path length might be proportional to the *logarithm* of the number of nodes.

As a quick check, we can change the plot to use a logarithmic scale for the X axis: In the plot window, select the “Format—X Axis” menu item, click

"Logarithmic", and click "OK". Figure 10.2 shows the result, a plot of mean query path length versus $\log N$. (The X axis ticks are still marked with the actual value of N.) The plot is close to a straight line. So the hypothesis that mean query path length is proportional to $\log N$ deserves further investigation.

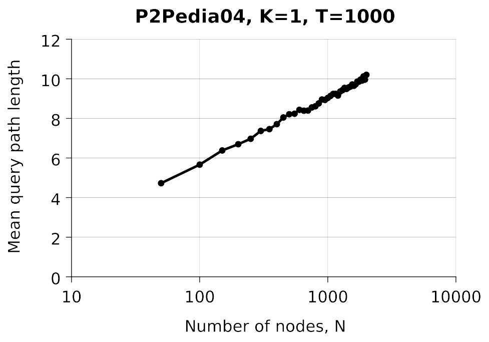


Figure 10.2: P2Pedia04 mean query path length versus $\log N$

Chapter 11

Regression

In Chapter 10, we gained an inkling that the P2Pedia system's mean query path length Q might be proportional to $\log N$, the logarithm of the number of nodes. To be precise, we are hypothesizing the following **model** for Q:

$$Q = a + b \log N \tag{11.1}$$

where a and b are the **model parameters.** We need to find the values of the model parameters that give the best agreement, or **best fit,** between the model and the data. We also need to decide whether that best-fit model is in fact a *good* fit to the data; that is, whether the hypothesis is true. To decide that, we'll do a **statistical goodness-of-fit test** on the model.

The model in Equation 11.1 is a **linear** model in the independent variable ($\log N$). We chose a linear model because there is a particularly convenient procedure, **linear regression,** that supplies both the best-fit model parameters and the p-value of the statistical test of the hypothesis that the model is a good fit to the data.

The linear regression procedure assumes that the dependent variable—Q, in this case—is not a single quantity, but is *a random variable with a normal distribution.* The randomness could arise either from measurement errors or, as in our case, from inherent variability in the quantity being measured. The linear regression procedure needs to know, for each data point, the value of the independent variable, the mean of the dependent variable's distribution, and the standard deviation of the dependent variable's distribution. We saw in Chapter 9 that the distribution of Q is close to a normal distribution. So we will feed the measured means and standard deviations of the query path lengths into the linear regression procedure.

The linear regression procedure fits a series of (x, y) data points to the linear model $y = a + bx$ by choosing the model parameters a and b to minimize the quantity

$$\chi^2 = \sum_{i=1}^{M} \left(\frac{y_i - (a + bx_i)}{\sigma_i} \right)^2 \tag{11.2}$$

where M is the length of the series, x_i is the independent variable value for the i-th data point, y_i is the dependent variable value for the i-th data point (the mean of the i-th data point's dependent variable's distribution), and σ_i is the standard deviation of the i-th data point's dependent variable's distribution. This is called a **least squares fit,** because we choose a and b to minimize the sum of the squares of the differences between the actual data y values and the model y values. If the hypothesis is true, that the linear model is a good fit to the data, then χ^2 obeys a chi-square distribution with $M-2$ degrees of freedom, and Equation 4.6 gives the p-value.

We'll have to tweak a few things in the simulation program if we're going to do a linear regression with $(\log N)$ as the independent variable. Notice how the data points in Figure 10.2 are all bunched together at the right-hand end of the plot. That's because N increases by a fixed amount as we go to the next data point. Instead, what we want is to increase *the logarithm of* N by a fixed amount as we go to the next data point. The data points will then be evenly spaced on a logarithmic scale. Furthermore, this will let us investigate a broader range of N values than the previous program.

We'll provide the following knobs: the logarithm of the number of nodes lower bound, $(\log N_L)$; the logarithm of the number of nodes upper bound, $(\log N_U)$; and the logarithm of the number of nodes delta, $(\log N_\Delta)$. When we say "logarithm," we mean "logarithm to the base 10." The program will do simulation runs with $(\log N)$ set to $(\log N_L)$, $(\log N_L + \log N_\Delta)$, $(\log N_L + 2\log N_\Delta)$, $(\log N_L + 3\log N_\Delta)$, . . . , $(\log N_U)$. In other words, the program will do simulation runs with N set to $10^{(\log N_L)}$, $10^{(\log N_L+\log N_\Delta)}$, $10^{(\log N_L+2\log N_\Delta)}$, $10^{(\log N_L+3\log N_\Delta)}$, . . . , $10^{(\log N_U)}$.

Here is the fifth version of the P2Pedia simulation program. The linear regression procedure is implemented in class edu.rit.numeric.ListXYZSeries in the Parallel Java 2 Library.

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.ListXYZSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.XYZSeries;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia05
   {
   public static void main
      (String[] args)
      throws Exception
      {
```

```
      if (args.length != 6) usage();
```

This time, the knobs $(\log N_L)$, $(\log N_U)$, and $(\log N_\Delta)$ are floating point numbers (type `double`), not integers (type `int`).

```
      double log_N_L = Double.parseDouble (args[0]);
      double log_N_U = Double.parseDouble (args[1]);
      double log_N_D = Double.parseDouble (args[2]);
      int K = Integer.parseInt (args[3]);
      int T = Integer.parseInt (args[4]);
      long seed = Long.parseLong (args[5]);
      Random prng = new Random (seed);
```

As in the previous program, the `mqpl` variable will accumulate data for the plot of Q versus N. The new variable `model` will accumulate data for the linear regression procedure on the model $Q = a + b \log N$.

```
      ListXYSeries mqpl = new ListXYSeries();
      ListXYZSeries model = new ListXYZSeries();
      System.out.printf ("\tQuery path length, Q%n");
      System.out.printf ("N\tMean\tStddev%n");
```

This time the outer loop increases $(\log N)$ from $(\log N_L)$ to $(\log N_U)$ in steps of $(\log N_\Delta)$. However, the loop control variable `r` is still an integer, and $(\log N)$ is computed afresh from `r` each time through the loop. Doing it this way, the values of $(\log N)$ are less susceptible to accumulated floating point roundoff error than if we repeatedly added $(\log N_\Delta)$ to $(\log N)$.

```
      double log_N;
      for (int r = 0; (log_N = log_N_L + r*log_N_D) <= log_N_U;
            ++ r)
         {
```

N itself is just $10^{(\log N)}$ rounded to the nearest integer.

```
         int N = (int) (Math.pow(10.0,log_N) + 0.5);
         int[][] conn = new int[N][K+1];
         for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
         int[] dist = new int[N];
         ListSeries len = new ListSeries();
         for (int trial = 0; trial < T; ++ trial)
            {
            for (int i = 0; i < N; ++ i)
               {
               RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j)
               conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         len.add (shortestPathLength
            (conn, dist, orig, match));
         }
      Series.Stats stats = len.stats();
      System.out.printf ("%d\t%.2f\t%.2f%n",
         N, stats.mean, stats.stddev);
      mqpl.add (N, stats.mean);
```

The above code for the simulation run with the current N is the same as in the previous program. At the end of the run we also accumulate, in the `model` variable, the information the linear regression procedure needs regarding the current data point: the independent variable value ($\log N$), the dependent variable mean, and the dependent variable standard deviation.

```
      model.add (Math.log10(N), stats.mean, stats.stddev);
      }
```

Pop Quiz: Why did we accumulate "`Math.log10(N)`" rather than just "`log_N`"?
Answer: Because `N`, the value actually used in the simulation, was rounded off to the nearest integer, so `log_N` might not be exactly equal to the logarithm of `N`.

When the simulation runs have finished, we perform the linear regression procedure by calling the `model.linearRegression()` method. The results are returned in an object of type edu.rit.numeric.XYZSeries.Regression. We print the model parameters a and b. Because the data going into the linear regression consisted of random variables, there is also variability in the model parameter values, and we print the standard deviations of a and b. Finally, we print the χ^2 statistic from Equation 11.2 and its p-value.

```
   XYZSeries.Regression regr = model.linearRegression();
   System.out.printf ("Q = a + b log N%n");
   System.out.printf ("a = %.2f%n", regr.a);
   System.out.printf ("b = %.2f%n", regr.b);
   System.out.printf ("stddev(a) = %.2f%n",
      Math.sqrt(regr.var_a));
   System.out.printf ("stddev(b) = %.2f%n",
      Math.sqrt(regr.var_b));
   System.out.printf ("chi^2 = %.6f%n", regr.chi2);
   System.out.printf ("p-value = %.6f%n", regr.significance);
   new Plot()
      .plotTitle ("P2Pedia05, K="+K+", T="+T)
      .xAxisTitle ("Number of nodes, N")
      .yAxisTitle ("Mean query path length, Q")
```

This time we specify a logarithmic scale for the plot's X axis, and we turn on some minor grid lines.

```
         .xAxisKind (Plot.LOGARITHMIC)
         .xAxisMinorDivisions (10)
         .minorGridLines (true)
```

We plot the (N, Q) data points with dots but no line.

```
         .seriesStroke (null)
         .xySeries (mqpl)
```

We also plot the **regression line** with no dots. The regression line shows the values predicted by the model with the fitted model parameters.

```
         .seriesDots (null)
         .seriesStroke (Strokes.solid (1))
         .xySeries
            (Math.pow(10.0,log_N_L), regr.a + regr.b*log_N_L,
             Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U)
         .getFrame()
         .setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia05 <log N_L> <log N_U> "+
          "<log N_D> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

Here's what the program prints with (log N) going from 1 to 4 in steps of 0.1 (so N goes from 10 to 10000), with $K = 1$ extra connection per node, and with $T = 1000$ trials per simulation run. Figure 11.1 shows the plot of mean query path length versus N, along with the regression line. The p-value of 1.000000 indicates that the model is an excellent fit to the data, as is also apparent on the plot.

```
$ java P2Pedia05 1 4 0.1 1 1000 142857
        Query path length, Q
N       Mean    Stddev
10      2.08    1.28
13      2.57    1.43
16      2.85    1.59
20      3.29    1.62
25      3.55    1.75
32      3.87    1.88
40      4.42    1.86
50      4.64    1.91
63      4.98    1.93
79      5.27    2.06
100     5.72    2.07
```

```
126     6.04    2.20
158     6.29    2.14
200     6.80    2.14
251     7.07    2.22
316     7.48    2.17
398     7.65    2.19
501     7.99    2.20
631     8.22    2.32
794     8.78    2.20
1000    9.11    2.17
1259    9.41    2.15
1585    9.74    2.21
1995    10.11   2.26
2512    10.35   2.28
3162    10.73   2.20
3981    11.03   2.20
5012    11.32   2.21
6310    11.61   2.30
7943    11.93   2.12
10000   12.26   2.20
Q = a + b log N
a = -1.18
b = 3.40
stddev(a) = 0.92
stddev(b) = 0.38
chi^2 = 0.064024
p-value = 1.000000
```

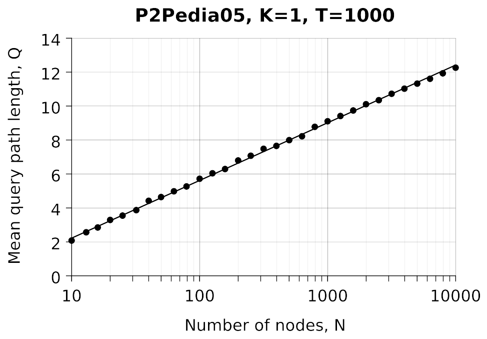


Figure 11.1: P2Pedia05 mean query path length Q versus number of nodes N

Let's look more closely at the fitted model parameters, particularly the slope $b = 3.40$. The model says that Q is 3.40 times the base-10 logarithm of N, minus 1.18. But because logarithms in different bases are proportional to each other, we can express the model using a different logarithm, particularly the base-2 logarithm. To go from base-10 to base-2 logarithms, we use these facts, where "ln" is the natural (base-e) logarithm:

$$\log_{10} x = \frac{\ln x}{\ln 10} \tag{11.3}$$

$$\log_2 x = \frac{\ln x}{\ln 2} \tag{11.4}$$

$$\log_{10} x = \frac{\ln 2}{\ln 10} \log_2 x \tag{11.5}$$

Therefore, we can express the model this way:

$$Q = a + b\,\frac{\ln 2}{\ln 10} \log_2 N \tag{11.6}$$

Plugging in the numbers, we get

$$\begin{aligned} Q &= -1.18 + 3.40\,\frac{\ln 2}{\ln 10} \log_2 N \\ &= -1.18 + 1.02\,\log_2 N \end{aligned} \tag{11.7}$$

This says that Q is very nearly equal to the base-2 logarithm of N, minus 1.18. It is curious that as the number of nodes increases, the mean query path length goes up almost exactly as the base-2 logarithm of the number of nodes, in a P2Pedia system with two connections per node (the connection to the successor node plus one extra connection). We will return to this observation later.

Chord, published in 2003,[1] is a well-researched and often-cited distributed hash table algorithm. In their paper, the authors showed (using a mathematical model) that Chord's mean query path length is proportional to the base-2 logarithm of the number of nodes, as we showed (using a simulation program) was the case for P2Pedia. However, Chord requires each node to be connected to a large number of other nodes (160, in the paper); and the IDs of these nodes are precisely specified, they cannot be chosen arbitrarily. In contrast, P2Pedia only requires two connections per node, one to the successor node and one to any other arbitrary node, to achieve a mean query path length proportional to $\log_2 N$. On the other hand, Chord forwards the query to just one node at a time; P2Pedia floods the query throughout the whole network.

We used the linear regression procedure to fit data to a logarithmic model. Linear regression can be used on four kinds of models:

Linear model $y = a + bx$. The independent variable is x and the dependent variable is y. The linear regression procedure yields the intercept a and the slope b.

[1] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking,* 11(1):17–32, February 2003.

Logarithmic model $y = a + b(\ln x)$. This is a linear model where the independent variable is $(\ln x)$ and the dependent variable is y. The linear regression procedure yields the intercept a and the slope b.

Power model $y = a \cdot x^b$. Taking the natural logarithm of both sides gives $(\ln y) = (\ln a) + b(\ln x)$. This is a linear model where the independent variable is $(\ln x)$ and the dependent variable is $(\ln y)$. The linear regression procedure yields the intercept $a' = (\ln a)$ and the slope b. Then in the original model, the multiplier $a = \mathrm{e}^{a'}$ and the exponent is b.

Exponential model $y = a \cdot b^x$. Taking the natural logarithm of both sides gives $(\ln y) = (\ln a) + (\ln b)x$. This is a linear model where the independent variable is x and the dependent variable is $(\ln y)$. The linear regression procedure yields the intercept $a' = (\ln a)$ and the slope $b' = (\ln b)$. Then in the original model, the multiplier $a = \mathrm{e}^{a'}$ and the base $b = \mathrm{e}^{b'}$.

Remember that the linear regression procedure needs to know the values of the independent variable, either x or $(\ln x)$, along with the corresponding means and standard deviations of the dependent variable, either y or $(\ln y)$. For further information about linear regression and other kinds of model fitting, see the *Numerical Recipes* book listed in Appendix B.

Chapter 12

Multiple Knob Turning

The P2Pedia05 program in Chapter 11 lets us explore the relationship between the P2Pedia system's mean query path length Q and one of the system attributes, the number of nodes N. To fully understand the system's behavior, however, we need to explore how Q depends on *both* the number of nodes N *and* the number of extra connections per node K.

It's straightforward to enclose the previous program's loop over N within another outer loop over K, putting multiple data series and regression lines on the plot. We'll provide knobs to let K go from K_L to K_U in steps of K_Δ. Here is version 6 of the P2Pedia simulation program.

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.ListXYZSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.XYZSeries;
import edu.rit.numeric.plot.Dots;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia06
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 8) usage();
      double log_N_L = Double.parseDouble (args[0]);
      double log_N_U = Double.parseDouble (args[1]);
      double log_N_D = Double.parseDouble (args[2]);
      int K_L = Integer.parseInt (args[3]);
      int K_U = Integer.parseInt (args[4]);
      int K_D = Integer.parseInt (args[5]);
      int T = Integer.parseInt (args[6]);
      long seed = Long.parseLong (args[7]);
      Random prng = new Random (seed);
```

We're going to be adding data series to the plot each time through the K loop, so we have to create the plot object outside the K loop.

```
      Plot plot = new Plot()
         .plotTitle ("P2Pedia06, T="+T)
         .xAxisTitle ("Number of nodes, N")
         .yAxisTitle ("Mean query path length, Q")
         .xAxisKind (Plot.LOGARITHMIC)
         .xAxisMinorDivisions (10)
         .minorGridLines (true)
```

We'll be putting a label to the right of each data series giving the value of K. The following methods set up the plot to display these labels properly.

```
         .rightMargin (36)
         .labelPosition (Plot.RIGHT)
         .labelOffset (6);
```

Here is the outer-outer loop where we turn the K knob. The body of this loop is the same as one run of the whole P2Pedia05 simulation program.

```
      for (int K = K_L; K <= K_U; ++ K)
         {
         ListXYSeries mqpl = new ListXYSeries();
         ListXYZSeries model = new ListXYZSeries();
         System.out.printf ("=======================%n");
         System.out.printf ("K = %d%n", K);
         System.out.printf ("\tQuery path length, Q%n");
         System.out.printf ("N\tMean\tStddev%n");
         double log_N;
         for (int r = 0; (log_N = log_N_L + r*log_N_D) <= log_N_U;
               ++ r)
            {
            int N = (int) (Math.pow(10.0,log_N) + 0.5);
            int[][] conn = new int[N][K+1];
            for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
            int[] dist = new int[N];
            ListSeries len = new ListSeries();
            for (int trial = 0; trial < T; ++ trial)
               {
               for (int i = 0; i < N; ++ i)
                  {
                  RandomSubset rs = new RandomSubset (prng, N)
                     .remove (i) .remove ((i + 1)%N);
                  for (int j = 1; j <= K; ++ j)
                     conn[i][j] = rs.next();
                  }
               int orig = prng.nextInt (N);
               int match = prng.nextInt (N);
               len.add (shortestPathLength
                  (conn, dist, orig, match));
               }
            Series.Stats stats = len.stats();
            System.out.printf ("%d\t%.2f\t%.2f%n",
               N, stats.mean, stats.stddev);
            mqpl.add (N, stats.mean);
            model.add (Math.log10(N), stats.mean, stats.stddev);
            }
         XYZSeries.Regression regr = model.linearRegression();
         System.out.printf ("Q = a + b log N%n");
         System.out.printf ("a = %.2f%n", regr.a);
         System.out.printf ("b = %.2f%n", regr.b);
         System.out.printf ("stddev(a) = %.2f%n",
            Math.sqrt(regr.var_a));
         System.out.printf ("stddev(b) = %.2f%n",
            Math.sqrt(regr.var_b));
         System.out.printf ("chi^2 = %.6f%n", regr.chi2);
         System.out.printf ("p-value = %.6f%n", regr.significance);
         plot.seriesDots (Dots.circle())
            .seriesStroke (null)
            .xySeries (mqpl)
            .seriesDots (null)
            .seriesStroke (Strokes.solid (1))
            .xySeries
               (Math.pow(10.0,log_N_L), regr.a + regr.b*log_N_L,
                Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U)
```

Here's where we label the current data series on the plot. The `label()` method's arguments are the label's text, x coordinate, and y coordinate. The `labelPosition()` and `labelOffset()` method calls at the top of the program specified that the label is to appear 6 pixels to the right of the given (x, y) point, which is the right endpoint of the current data series.

```
            .label ("K="+K,
                Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U);
         }
      plot.getFrame().setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia06 <log N_L> <log N_U> "+
          "<log N_D> <K_L> <K_U> <K_D> <T> <seed>");
      System.exit (1);
      }
   }
```

Figure 12.1 shows the plot the program generates with $(\log N)$ going from 1 to 4 in steps of 0.1 (so N goes from 10 to 10000), with K going from 1 to 4, and with $T = 1000$ trials per simulation run. Table 12.1 lists the regression models, χ^2 statistics, and p-values the program computed for each K. The p-values are all 1.000000, indicating that each model is an excellent fit to the data.

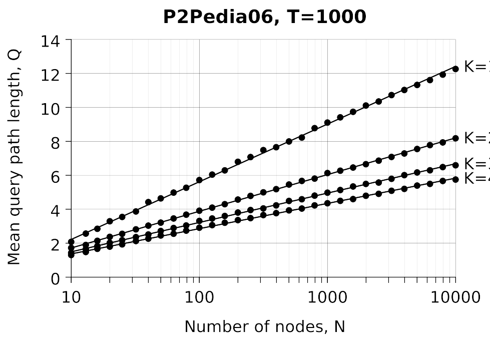


Figure 12.1: P2Pedia06 mean query path length Q versus number of nodes N and number of extra connections per node K

Table 12.1: P2Pedia06 mean query path length regression models

K	*Model*	χ^2	*p-value*
1	$Q = -1.18 + 3.40 \log N$	0.064024	1.000000
2	$Q = -0.44 + 2.16 \log N$	0.024922	1.000000
3	$Q = -0.24 + 1.73 \log N$	0.052495	1.000000
4	$Q = -0.12 + 1.49 \log N$	0.065369	1.000000

Table 12.1 tells us that N affects Q in that Q is proportional to $\log N$, and that K affects Q by altering the intercept a and the slope b in the regression model $Q = a + b \log N$. Let's rearrange the model a little:

$$\begin{aligned} Q &= a + b\ \log N \\ &= a + b\,\frac{\ln N}{\ln 10} \\ &= a + \frac{\ln N}{(\ln 10)/b} \\ &= a + \frac{\ln N}{\ln(\exp((\ln 10)/b))} \\ &= a + \frac{\ln N}{\ln t} \\ &= a + \log_t N \end{aligned} \tag{12.1}$$

In other words, Q is proportional to the base-t logarithm of N, where $t = \exp((\ln 10)/b)$. Table 12.2 lists the regression models in this form.

Table 12.2: P2Pedia06 mean query path length regression models, rewritten

K	*Model*
1	$Q = -1.18 + \log_{1.97} N$
2	$Q = -0.44 + \log_{2.90} N$
3	$Q = -0.24 + \log_{3.78} N$
4	$Q = -0.12 + \log_{4.69} N$

It's apparent from Table 12.2 that t, the base of the logarithm in the regression model, increases as K increases. Indeed, in each case t is just a little bit less than $K + 1$, the number of nodes connected to each node. This observation, finally, reveals some deep insight into the P2Pedia system's behavior.

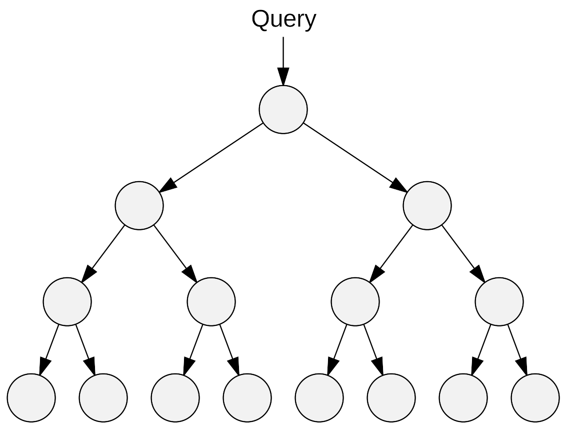


Figure 12.2: P2Pedia query tree with two connections per node

Consider a P2Pedia system with two connections per node ($K = 1$). When a query enters the system and floods from node to node, the query is essentially following a binary tree of nodes (Figure 12.2). The query path length is the number of levels in the tree, which is $\log_2 N$. However, because the extra connections go to randomly chosen nodes, some of the branches in the tree go to nodes that have already received the query, rather than nodes that have not. Thus, the query does not go to exactly two new nodes on every hop, but to slightly fewer new nodes on the average—1.97, according to our measurements. This explains why Q is proportional to $\log_{1.97} N$ rather than $\log_2 N$.

Furthermore, the query doesn't necessarily have to go all the way to the bottom of the binary tree. The query might encounter the matching node before reaching the bottom. Thus, Q is somewhat less than $\log_{1.97} N$ on the average—1.18 hops less, according to our measurements. This explains why Q is $-1.18 + \log_{1.97} N$ rather than just $\log_{1.97} N$.

Similarly, in a P2Pedia system with three connections per node ($K = 2$), a query traverses a ternary tree (Figure 12.3) whose height is $\log_3 N$. Measurements show that a query goes to 2.90 new nodes on the average rather than 3, and that a query reaches the matching node 0.44 hops before the bottom of the tree on the average. Similar behavior is observed for $K = 3$ and 4.

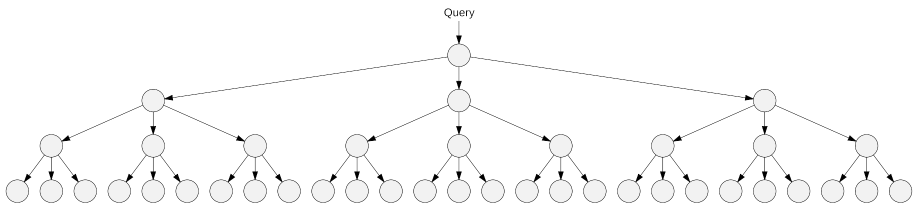


Figure 12.3: P2Pedia query tree with three connections per node

Chapter 13

Discrete Event Simulation

When we developed the P2Pedia simulation programs in the previous chapters, we omitted many aspects of the real system that were irrelevant to investigating the mean query path length. However, we may have *over*-simplified the system. In a real P2Pedia network, the nodes along the fewest-hops path might use slow dialup Internet connections, while nodes along other paths use fast broadband connections. Or perhaps some node along the fewest-hops path is experiencing a temporarily heavy processing load, delaying queries going through that node. For these and other reasons, a query might reach the matching node more quickly by flooding along some path other than the one with the fewest hops from the originating node. Furthermore, the previous simulation programs tell us nothing about *how long*—how many seconds—it takes to process a query; we can only find the average number of hops in the query paths.

For a more realistic simulation, we need to simulate *the passage of time.* That takes us into the realm of **discrete event simulation.**

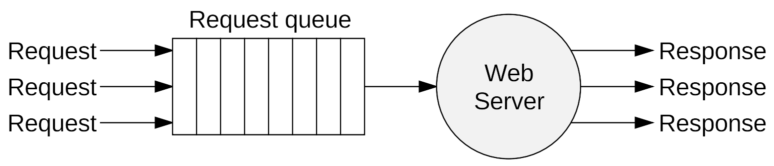


Figure 13.1: Web server

We'll ease into the topic by simulating a **web server** (Figure 13.1). HTTP request messages from users' web browsers arrive at the server and go into a queue. The server takes the first request in the queue, looks up the necessary information in its database, formulates an HTTP response message containing the requested web page, sends the response back to the browser, and goes on to the next request in the queue.

We want the web server simulation program to produce output like this:

```
0.000 Request 1 added to queue
0.000 Started serving Request 1
0.103 Request 1 removed from queue
0.106 Request 2 added to queue
0.106 Started serving Request 2
0.117 Request 2 removed from queue
0.134 Request 3 added to queue
0.134 Started serving Request 3
0.160 Request 4 added to queue
0.208 Request 3 removed from queue
0.208 Started serving Request 4
0.221 Request 5 added to queue
0.266 Request 6 added to queue
0.486 Request 7 added to queue
0.505 Request 8 added to queue
0.572 Request 9 added to queue
0.584 Request 4 removed from queue
0.584 Started serving Request 5
0.740 Request 5 removed from queue
0.740 Started serving Request 6
0.751 Request 10 added to queue
0.974 Request 6 removed from queue
0.974 Started serving Request 7
1.067 Request 7 removed from queue
1.067 Started serving Request 8
1.087 Request 8 removed from queue
1.087 Started serving Request 9
1.097 Request 9 removed from queue
1.097 Started serving Request 10
1.101 Request 10 removed from queue
```

This is a transcript of a series of **events.** The events consist of adding a request to the queue, starting to serve the first request in the queue, and removing a request from the queue (when finished serving it). Each event happens at a particular instant in time, listed at the beginning of each line. The unit of time doesn't matter, as long as we use the same unit throughout the simulation. We'll generally express time in seconds. The program simulates the events in ascending order of time. Because the program *simulates* a series of *events* that happen at *discrete* instants, it is called a *discrete event simulation* program.

We are still not simulating all the gory detail in the real web server. The socket connections, the contents of the request and response messages, the information on the web pages, the protocols used to send the messages over the network, and so on are not simulated. However—and this is the key difference from the simulation programs we've studied up to this point—we *are* simulating the passage of time.

A discrete event simulation program does not simulate in *real time.* The program only simulates what goes on at those discrete instants when an event occurs. Once the program has processed an event, the program does an *instantaneous time warp* to the time of the next event, and processes that. The program repeats this until it runs out of events. In this way, the passage of minutes, hours, or days of simulated time can take place in microseconds of program execution time.

Let's observe a few things in the transcript. Focus on the events where requests **arrive** at (and are added to) the queue:

```
0.000 Request 1 added to queue
0.106 Request 2 added to queue
0.134 Request 3 added to queue
0.160 Request 4 added to queue
0.221 Request 5 added to queue
0.266 Request 6 added to queue
0.486 Request 7 added to queue
0.505 Request 8 added to queue
0.572 Request 9 added to queue
0.751 Request 10 added to queue
```

The **interarrival time** is the difference between the times when successive requests arrive at the queue. In this example, the interarrival times are 0.106, 0.028, 0.026, 0.061, 0.045, 0.220, 0.019, 0.067, 0.179. We will model the interarrival time as a random variable with a certain probability distribution, based on what we know (or assume) are the characteristics of the users and web browsers sending the requests. So to write the web server simulation program, we will need to generate random numbers obeying this distribution, whatever it might be.

After arriving at the queue and possibly sitting there a while, the server begins serving the request, and sometime later finishes serving the request and removes it from the queue. Focus on the started-serving and removal events:

```
0.000 Started serving Request 1
0.103 Request 1 removed from queue
0.106 Started serving Request 2
0.117 Request 2 removed from queue
0.134 Started serving Request 3
0.208 Request 3 removed from queue
0.208 Started serving Request 4
0.584 Request 4 removed from queue
0.584 Started serving Request 5
0.740 Request 5 removed from queue
0.740 Started serving Request 6
0.974 Request 6 removed from queue
0.974 Started serving Request 7
```

```
1.067 Request 7 removed from queue
1.067 Started serving Request 8
1.087 Request 8 removed from queue
1.087 Started serving Request 9
1.097 Request 9 removed from queue
1.097 Started serving Request 10
1.101 Request 10 removed from queue
```

The **service time** is the difference between the times when the server starts and finishes serving a request. In this example, the service times are 0.103, 0.011, 0.074, 0.376, 0.156, 0.234, 0.093, 0.020, 0.010, 0.004. Like the interarrival time, we will model the service time as a random variable with a certain probability distribution, based on what we know (or assume) are the characteristics of the web server, and we will need to generate random numbers obeying this distribution.

Events can trigger the creation of other, future events. Adding a request to the queue also triggers the future event of adding the next request to the queue, after the proper random interarrival time has elapsed (provided there are more requests to simulate). When a request comes to the front of the queue and the server starts serving it, that also triggers the future event of removing the request from the queue, after the proper random service time has elapsed.

However, the times of the arrival, start-serving, and removal events for a particular request are completely independent of the event times for the other requests. Events related to one request could be interleaved arbitrarily with events related to other requests. The discrete event simulation program must somehow contrive to keep track of all these events and process them in ascending order of time.

In the next couple of chapters, we'll build the tools we need to write discrete event simulation programs. Then we'll write a simulation of the web server.

Chapter 14

Exponential Distribution

If events occur at random but at a fixed average rate, and if the number of events in any time interval is independent of the number of events in any other time interval, then it can be shown that the events' interarrival time is a random variable with an **exponential distribution.** For this reason, exponential distributions are often used in discrete event simulation programs.

In contrast to the discrete random variable with a Bernoulli distribution in Chapter 2, the exponential distribution pertains to a **continuous** random variable, one that can take on any real-number value in a continous range. The **probability density function (p.d.f.)** $f(x)$ for an exponentially distributed random variable x is

$$f(x) = \begin{cases} \lambda \mathrm{e}^{-\lambda x}\,, & x \ge 0 \\ 0\,, & \text{otherwise} \end{cases} \tag{14.1}$$

$f(x)$ is the probability that the random variable's value will fall in the infinitesimal interval $(x, x + \mathrm{d}x)$. When used for event interarrival times, λ is the mean rate at which events occur.

The **cumulative distribution function (c.d.f.)** $F(x)$ for an exponentially distributed random variable x is

$$\begin{aligned} F(x) &= \int_{-\infty}^{x} f(u)\,\mathrm{d}u \\ &= \int_{0}^{x} \lambda \mathrm{e}^{-\lambda u}\,\mathrm{d}u \\ &= 1 - \mathrm{e}^{-\lambda x}\,,\ x \ge 0 \\ &= 0\,,\ \text{otherwise} \end{aligned} \tag{14.2}$$

$F(x)$ is the probability that the random variable's value will be less than or equal to x. For example, Figure 14.1 shows the p.d.f. of an exponential distribution with $\lambda = 0.75$, and Figure 14.2 shows the c.d.f.

Now we need a way to generate values of a random variable x with an exponential distribution, for use in the web server simulation program. Here's

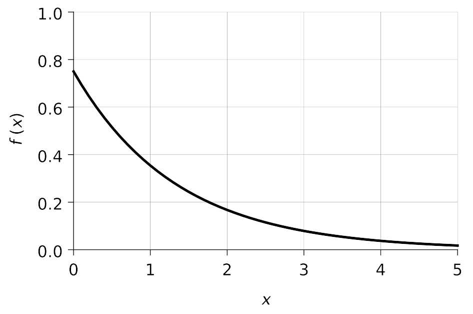


Figure 14.1: Exponential distribution p.d.f., $\lambda = 0.75$

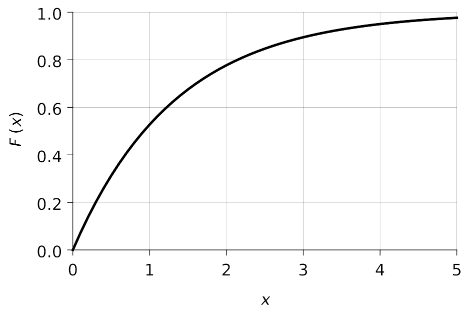


Figure 14.2: Exponential distribution c.d.f., $\lambda = 0.75$

how: Pick a random number y uniformly distributed between 0 and 1; return the number x such that $F(x) = y$. We know how to do the first step; just call the `nextDouble()` method on a PRNG. For the second step, we have to solve for x in the exponential c.d.f.:

$$\begin{aligned} F(x) &= y \\ 1 - \mathrm{e}^{-\lambda x} &= y \\ \mathrm{e}^{-\lambda x} &= 1 - y \\ -\lambda x &= \ln(1 - y) \\ x &= -\frac{\ln(1-y)}{\lambda} \end{aligned} \tag{14.3}$$

We can simplify the formula a bit by noting that if y is uniformly distributed between 0 and 1, then $1 - y$ is also uniformly distributed between 0 and 1. Therefore,

$$x = -\frac{\ln y}{\lambda} \tag{14.4}$$

will also yield an exponential distribution.

Here is class edu.rit.numeric.DoublePrng in the Parallel Java 2 Library. This is the abstract base class for a PRNG for a continuous random variable. It encapsulates the underlying PRNG from which to get random numbers uniformly distributed between 0 and 1. The abstract `next()` method, which returns random numbers from some continuous distribution, is defined in a subclass.

```
package edu.rit.numeric;
import edu.rit.util.Random;
public abstract class DoublePrng
   {
   /**
    * The underlying uniform PRNG.
    */
   protected final Random myUniformPrng;

   /**
    * Construct a new double PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if theUniformPrng is null.
    */
   public DoublePrng
      (Random theUniformPrng)
      {
      if (theUniformPrng == null)
         {
         throw new NullPointerException
            ("DoublePrng(): theUniformPrng is null");
         }
      myUniformPrng = theUniformPrng;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public abstract double next();
   }
```

Here is class edu.rit.numeric.ExponentialPrng in the Parallel Java 2 Library. This extends class DoublePrng and implements the `next()` method to generate exponentially distributed random numbers using Equation 14.4. The mean arrival rate λ is specified as a constructor argument. (Other subclasses of class DoublePrng could be defined to yield other distributions.)

```
package edu.rit.numeric;
import edu.rit.util.Random;
public class ExponentialPrng
   extends DoublePrng
   {
   private double lambda;

   /**
    * Construct a new exponential PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    * @param  lambda          Mean rate lambda > 0.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if theUniformPrng is null.
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if lambda <= 0.
    */
   public ExponentialPrng
      (Random theUniformPrng,
       double lambda)
      {
      super (theUniformPrng);
      if (lambda <= 0)
         {
         throw new IllegalArgumentException
            ("ExponentialPrng(): lambda = "+lambda+" illegal");
         }
      this.lambda = lambda;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public double next()
      {
      return -Math.log(myUniformPrng.nextDouble())/lambda;
      }
   }
```

Chapter 15

Priority Queues

We need two basic software components with which to build discrete event simulation programs: a class for an event, and a class for the simulation itself, the thing that performs the events.

Here is the fairly simple abstract base class for an event, class edu.rit.sim-.Event in the Parallel Java 2 Library. Class Event encapsulates a reference to the simulation in which the event occurs and the time at which the event is to take place, and it provides getter methods for these items.

```
package edu.rit.sim;
public abstract class Event
   {
   // Simulation in which this event occurs.
   Simulation sim;

   // Simulation time of this event.
   double time;

   /**
    * Construct a new event.
    */
   public Event()
      {
      }

   /**
    * Returns the simulation in which this event occurs.
    *
    * @return  Simulation.
    */
   public final Simulation simulation()
      {
```

```
      return sim;
      }

   /**
    * Returns this event's simulation time, the time when this
    * event is scheduled to take place.
    *
    * @return  Simulation time.
    */
   public final double time()
      {
      return time;
      }
```

Here are two methods for adding a new event to the simulation. The first method schedules the event to occur at the given absolute time. The second method schedules the event to occur the given amount of time in the future. (The event's simulation object is the one that actually assigns the new event's `sim` and `time` fields.)

```
   /**
    * Schedule the given event to be performed at the given time
    * in this event's simulation.
    *
    * @param  t      Simulation time for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if t is less than the
    *     current simulation time.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public final void doAt
      (double t,
       Event event)
      {
      sim.doAt (t, event);
      }

   /**
    * Schedule the given event to be performed at a time dt in the
    * future (at current simulation time + dt) in this event's
    * simulation.
    *
    * @param  dt     Simulation time delta for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if dt is less than zero.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public final void doAfter
      (double dt,
       Event event)
      {
      sim.doAfter (dt, event);
      }
```

The `perform()` method contains the code for the processing that is to take place when the event happens. It is an abstract method that will be defined in a subclass.

```
   /**
    * Perform this event. Called by the simulation when the
    * simulation time equals the time when this event is
    * scheduled to take place.
    */
   public abstract void perform();
   }
```

Now that we have a class for an event, the central data structure in the simulation is an **event queue.** As events are created, they are added to the queue. The simulation consists of repeatedly taking the first event off the queue and performing the event by calling its `perform()` method, until no more events are created and the event queue empties out.

However, to do the simulation properly, the events must not the taken off the queue in the order they were added. Rather, the events must be taken off the queue *in ascending order of the events' times.* This calls for a **priority queue** rather than the usual first-in-first-out (FIFO) queue. Events can be added to the queue in any order, but they are removed from the queue in priority order; that is, higher-priority events first. In a discrete event simulation, an event with an earlier time has a higher priority.

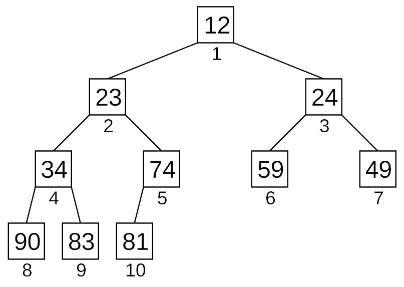
One way to build a priority queue is to scan the queue elements and insert a newly-added element in its proper place, so that the elements are always in ascending time order. We can then just remove the first element and know it is the highest-priority element. However, this means that adding an element takes $O(n)$ time, where n is the size of the queue. Removing the highest-priority element can take $O(1)$ or $O(n)$ time depending on how the queue is implemented, so adding and removing an element takes $O(n)$ time overall.

Another way to build a priority queue is to use a **heap** data structure. Both adding an element to a heap and removing the highest-priority element from a heap take $O(\log n)$ time, so adding and removing an element takes $O(\log n)$ time overall. This is better than the previous alternative.

A heap of n elements is stored in an array indexed from 1 to n:

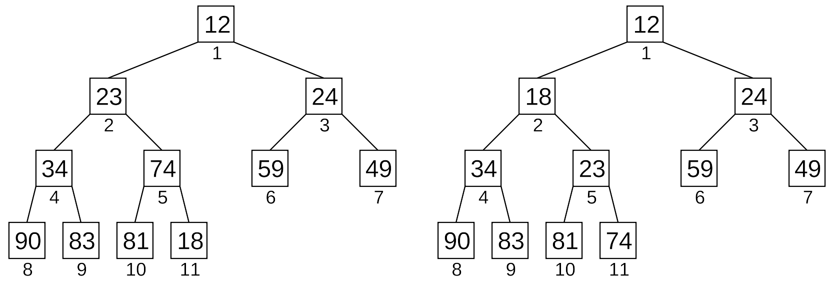
12	23	24	34	74	59	49	90	83	81
1	2	3	4	5	6	7	8	9	10

Although stored in a linear array, the elements are actually arranged in a binary tree. Element 1 is the root. The parent of element i is element $\lfloor i/2 \rfloor$. The children of element i are elements $2i$ and $2i + 1$. The elements fill in the tree levels from top to bottom, and within each level from left to right:



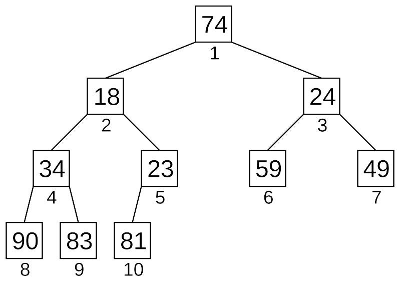
The tree is arranged so that every element is less than (has a higher priority than) both its children. This is the key property that makes $O(\log n)$ additions and removals possible.

To add an element to a heap of size n, put the new element in position $n+1$, then repeatedly swap the new element with its parent as long as the new element is less than its parent. This is called "sifting up" the element. Here's what happens when a new element (18) is added to the previous heap:

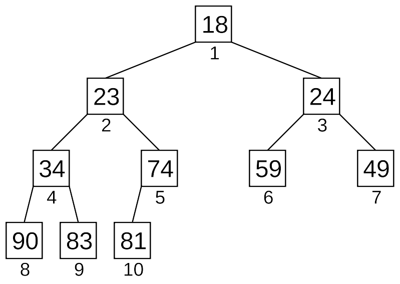


Before sifting up After sifting up

The highest-priority element is at the root of the tree, position 1. To remove the highest-priority element, move the element at position n into position 1, then repeatedly swap this element with its smaller child as long as its smaller child is less than this element. This is called "sifting down" the element. Here's what happens when the highest-priority element (12) is removed from the previous heap:



Before sifting down



After sifting down

The number of iterations in the sift-up and sift-down procedures is at most $\log_2 n$, the number of levels in the binary tree. Thus, adding an element to or removing the highest-priority element from a heap takes $O(\log n)$ time.

Now we can build a simulation object encapsulating a priority queue of events using the heap data structure. The simulation also keeps track of the current simulation time (the time of the event currently being processed). The simulation has methods to get the current simulation time, add an event to the simulation, and run the simulation by performing all events in ascending time order. Here is class edu.rit.sim.Simulation in the Parallel Java 2 Library.

```
package edu.rit.sim;
public class Simulation
   {
   // Minimum-priority queue of events. Uses a heap data
   // structure. The entry at index 0 is a sentinel with time =
   // 0.0.
   private Event[] heap = new Event [1024];

   // Number of entries in the heap (including the sentinel).
   private int N = 1;

   // Simulation time.
   private double T = 0.0;

   /**
    * Construct a new simulation.
    */
   public Simulation()
      {
      heap[0] = new Event() { public void perform() { } };
      heap[0].sim = this;
      heap[0].time = 0.0;
      }

   /**
```

```
     * Returns the current simulation time.
     *
     * @return  Simulation time.
     */
    public double time()
       {
       return T;
       }

    /**
     * Schedule the given event to be performed at the given time
     * in this simulation.
     *
     * @param  t      Simulation time for event.
     * @param  event  Event to be performed.
     *
     * @exception  IllegalArgumentException
     *     (unchecked exception) Thrown if t is less than the
     *     current simulation time.
     * @exception  NullPointerException
     *     (unchecked exception) Thrown if event is null.
     */
    public void doAt
       (double t,
        Event event)
       {
       // Verify preconditions.
       if (t < T)
          {
          throw new IllegalArgumentException
             ("Simulation.doAt(): t = "+t+
              " less than simulation time ="+T+", illegal");
          }
       if (event == null)
          {
          throw new NullPointerException
             ("Simulation.doAt(): event = null");
          }

       // Set event fields.
       event.sim = this;
       event.time = t;

       // Grow heap if necessary.
       if (N == heap.length)
```

```
      {
      Event[] newheap = new Event [N + 1024];
      System.arraycopy (heap, 0, newheap, 0, N);
      heap = newheap;
      }

   // Insert event into heap in min-priority order.
   heap[N] = event;
   siftUp (N);
   ++ N;
   }

/**
 * Schedule the given event to be performed at a time dt in the
 * future (at current simulation time + dt) in this simulation.
 *
 * @param  dt     Simulation time delta for event.
 * @param  event  Event to be performed.
 *
 * @exception  IllegalArgumentException
 *     (unchecked exception) Thrown if dt is less than zero.
 * @exception  NullPointerException
 *     (unchecked exception) Thrown if event is null.
 */
public void doAfter
   (double dt,
    Event event)
   {
   doAt (T + dt, event);
   }

/**
 * Run the simulation. At the start of the simulation, the
 * simulation time is 0. The run() method returns when there
 * are no more events.
 */
public void run()
   {
   while (N > 1)
      {
      // Extract minimum event from heap.
      Event event = heap[1];
      -- N;
      heap[1] = heap[N];
      heap[N] = null;
```

```
         if (N > 1) siftDown (1);

         // Advance simulation time and perform event.
         T = event.time;
         event.perform();
         }
      }

   /**
    * Sift up the heap entry at the given index.
    *
    * @param  c  Index.
    */
   private void siftUp
      (int c)
      {
      double c_time = heap[c].time;
      int p = c >> 1;
      double p_time = heap[p].time;
      while (c_time < p_time)
         {
         Event temp = heap[c];
         heap[c] = heap[p];
         heap[p] = temp;
         c = p;
         p = c >> 1;
         p_time = heap[p].time;
         }
      }

   /**
    * Sift down the heap entry at the given index.
    *
    * @param  p  Index.
    */
   private void siftDown
      (int p)
      {
      double p_time = heap[p].time;
      int lc = (p << 1);
      double lc_time =
         lc < N ? heap[lc].time : Double.POSITIVE_INFINITY;
      int rc = (p << 1) + 1;
      double rc_time =
         rc < N ? heap[rc].time : Double.POSITIVE_INFINITY;
      int c;
      double c_time;
      if (lc_time < rc_time)
         {
         c = lc;
         c_time = lc_time;
         }
      else
         {
         c = rc;
         c_time = rc_time;
         }
      while (c_time < p_time)
         {
         Event temp = heap[c];
         heap[c] = heap[p];
         heap[p] = temp;
         p = c;
         lc = (p << 1);
         lc_time =
            lc < N ? heap[lc].time : Double.POSITIVE_INFINITY;
         rc = (p << 1) + 1;
         rc_time =
            rc < N ? heap[rc].time : Double.POSITIVE_INFINITY;
         if (lc_time < rc_time)
            {
            c = lc;
            c_time = lc_time;
            }
         else
            {
            c = rc;
            c_time = rc_time;
            }
         }
      }
   }
```

Chapter 16

A Web Server

Now that we have the tools to generate exponentially distributed random numbers and to keep track of events, we can write the code for a discrete event simulation of the web server in Chapter 13 (Figure 16.1).

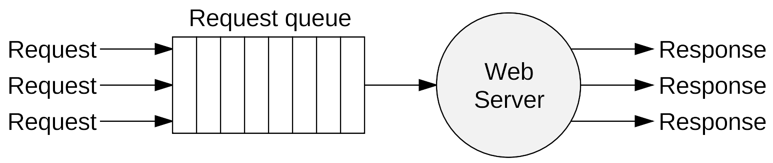


Figure 16.1: Web server

The simulation will have these knobs:

- Mean request rate λ. We'll assume that HTTP request messages arrive at at the web server at a mean rate of λ requests per second, and that the interarrival times are exponentially distributed.
- Mean service rate μ. We'll assume that the web server can respond to requests at a mean rate of μ requests per second, and that the service times are exponentially distributed.
- Number of requests to simulate, R.
- Pseudorandom number generator seed.

Here is the web server discrete event simulation program.

```
import edu.rit.numeric.ExponentialPrng;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import java.util.LinkedList;
public class WebServer01
   {
```

In the programs we've written so far, the variables have been local variables of the `main()` method. Now, however, we are going to break out pieces of the program into subroutines. It will be more convenient to declare the variables as global variables (static fields) so all the subroutines can access them. The first four variables are the knobs.

```
   private static double lambda;
   private static double mu;
   private static int R;
   private static long seed;
```

Next come a PRNG for generating random numbers uniformly distributed between 0 and 1, plus two PRNGs for generating exponentially distributed interarrival times and service times.

```
   private static Random prng;
   private static ExponentialPrng requestPrng;
   private static ExponentialPrng serverPrng;
```

Here we have the request queue, the number of requests generated so far, and the simulation object. The request queue is a linked list of Request objects; class Request will be defined later. Don't confuse the request queue with the event queue: the request queue is part of the system being simulated (as shown in Figure 16.1) and holds only the simulated incoming HTTP request messages; the event queue is hidden inside the simulation object and holds all the events in the simulation.

```
   private static LinkedList<Request> requestQueue;
   private static int requestCount;
   private static Simulation sim;

   public static void main
      (String[] args)
      {
```

This is the simulation main program. We get the knob settings off the command line.

```
      if (args.length != 4) usage();
      lambda = Double.parseDouble (args[0]);
      mu = Double.parseDouble (args[1]);
      R = Integer.parseInt (args[2]);
      seed = Long.parseLong (args[3]);
```

We set up the PRNGs. The `requestPrng` and `serverPrng` are both layered on top of the same underlying `prng`.

```
      prng = new Random (seed);
      requestPrng = new ExponentialPrng (prng, lambda);
      serverPrng = new ExponentialPrng (prng, mu);
```

We initialize the request queue, request count, and simulation object.

```
      requestQueue = new LinkedList<Request>();
      requestCount = 0;
      sim = new Simulation();
```

We generate the first request. This is done by calling a subroutine. Then we run the simulation. Further requests will be generated during the simulation run, as we will see.

```
      generateRequest();
      sim.run();
      }
```

Here is the class for a request. It is merely a holder for the request number. When a request object appears in a print statement, whatever the `toString()` method returns is printed, namely `"Request"` plus the request number.

```
   private static class Request
      {
      private int requestNumber;
      public Request
         (int requestNumber)
         {
         this.requestNumber = requestNumber;
         }
      public String toString()
         {
         return "Request " + requestNumber;
         }
      }
```

The `generateRequest()` subroutine creates a new request object with the next higher request number and adds it to the request queue. If there are still more requests to simulate, the subroutine also adds a future event to the simulation. This event will take place after the request interarrival time has elapsed. The request interarrival time is obtained from the request PRNG, which generates exponentially distributed random numbers with a mean arrival rate of λ. Later, when the event takes place, the `generateRequest()` method will be called to generate the next request.

```
   private static void generateRequest()
      {
      addToQueue (new Request (++ requestCount));
      if (requestCount < R)
         {
         sim.doAfter (requestPrng.next(), new Event()
            {
            public void perform() { generateRequest(); }
            });
         }
      }
```

The `addToQueue()` subroutine prints a line in the transcript reporting that the new request was added; adds the request to the end of the request queue; and, if the request queue had been empty—meaning the web server had been idle—starts serving the request. (If the request queue had not been empty, the web server is in the middle of serving another request, and it should not start serving this request.)

```
   private static void addToQueue
      (Request request)
      {
      System.out.printf ("%.3f %s added to queue%n",
         sim.time(), request);
      requestQueue.add (request);
      if (requestQueue.size() == 1) startServing();
      }
```

The `startServing()` subroutine prints a line in the transcript reporting that the web server started serving the first request in the queue. The subroutine then adds a future event to the simulation. This event will take place after the request service time has elapsed. The request service time is obtained from the service PRNG, which generates exponentially distributed random numbers with a mean service rate of μ. Later, when the event takes place, the `removeFromQueue()` method will be called to remove the request from the request queue.

```
   private static void startServing()
      {
      System.out.printf ("%.3f Started serving %s%n",
         sim.time(), requestQueue.getFirst());
      sim.doAfter (serverPrng.next(), new Event()
         {
         public void perform() { removeFromQueue(); }
         });
      }
```

The `removeFromQueue()` subroutine prints a line in the transcript reporting that the web server finished serving the first request in the queue; removes the first request from the queue; and, if the queue is not empty, starts serving the next request in the queue.

```
   private static void removeFromQueue()
      {
      System.out.printf ("%.3f %s removed from queue%n",
         sim.time(), requestQueue.removeFirst());
      if (requestQueue.size() > 0) startServing();
      }
```

The simulation stops once R requests have been generated, so that `generate-Request()` no longer adds any events to the simulation; and once all these events have been performed, so that the simulation's internal event queue is empty. At that point the simulation's `run()` method returns and the main program terminates.

```
   private static void usage()
      {
      System.err.println
         ("Usage: java WebServer01 <lambda> <mu> <R> <seed>");
      System.exit (1);
      }
   }
```

The transcript in Chapter 13 came from this simulation program run, with mean arrival rate $\lambda = 8$ requests per second, mean service rate $\mu = 10$ requests per second, and $R = 10$ requests.

```
$ java WebServer01 8 10 10 142857
0.000 Request 1 added to queue
0.000 Started serving Request 1
0.103 Request 1 removed from queue
0.106 Request 2 added to queue
0.106 Started serving Request 2
0.117 Request 2 removed from queue
0.134 Request 3 added to queue
0.134 Started serving Request 3
0.160 Request 4 added to queue
0.208 Request 3 removed from queue
0.208 Started serving Request 4
0.221 Request 5 added to queue
0.266 Request 6 added to queue
0.486 Request 7 added to queue
0.505 Request 8 added to queue
```

```
0.572 Request 9 added to queue
0.584 Request 4 removed from queue
0.584 Started serving Request 5
0.740 Request 5 removed from queue
0.740 Started serving Request 6
0.751 Request 10 added to queue
0.974 Request 6 removed from queue
0.974 Started serving Request 7
1.067 Request 7 removed from queue
1.067 Started serving Request 8
1.087 Request 8 removed from queue
1.087 Started serving Request 9
1.097 Request 9 removed from queue
1.097 Started serving Request 10
1.101 Request 10 removed from queue
```

Chapter 17

A DOS Attack

We're going to change several things to beef up the web server discrete event simulation program. First, we'll get rid of the transcript printouts. While these are useful for debugging the program, they'd be unwieldy if we were simulating thousands or millions of requests.

A request's **wait time** is the time the request is in the system, from the instant when the request arrived (and it went into the request queue) until the instant when the web server responded to the request (and it came out of the request queue). We are interested in the mean and standard deviation of the requests' wait times.

The previous program always accepted every incoming HTTP request message, regardless of how many were already queued up. This means the request queue could grow without bound, if requests arrived faster than the web server could respond. However, a real web server can have only a certain maximum number of pending requests (open socket connections). So we'll put a limit on the size of the request queue. If a request comes in when the queue is full, the web server **drops** (ignores) the request rather than putting it in the queue. We are interested in the **drop fraction,** the fraction of requests that were dropped.

We would like the program to print and plot the mean wait time, wait time standard deviation, and drop fraction as a function of the mean arrival rate λ, for a fixed mean service rate μ. So λ will be a knob the program turns automatically, from λ_L to λ_U in steps of λ_Δ.

With these capabilities, we can explore what happens when the web server experiences a **denial of service (DOS) attack.** In a DOS attack, the attackers try to send requests to the web server so quickly that legitimate users can't get through. We can simulate a DOS attack by turning the mean arrival rate knob way past the mean service rate knob.

Here is version 2 of the web server discrete event simulation program.

```
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import java.util.LinkedList;
public class WebServer02
   {
```

The knobs are the mean arrival rate lower bound λ_L, upper bound λ_U, and increment λ_Δ; the mean service rate μ; the maximum request queue length Q; the number of requests to simulate R; and the PRNG seed.

```
   private static double lambda_L;
   private static double lambda_U;
   private static double lambda_D;
   private static double mu;
   private static int Q;
   private static int R;
   private static long seed;
```

Here are two X-Y series objects to accumulate data points to be plotted.

```
   private static ListXYSeries meanWaitTimeVsLambda;
   private static ListXYSeries dropFractionVsLambda;
   private static Random prng;
   private static ExponentialPrng requestPrng;
   private static ExponentialPrng serverPrng;
   private static LinkedList<Request> requestQueue;
   private static int requestCount;
```

We keep track of the number of dropped requests to calculate the drop fraction. We keep track of the wait times of the requests to calculate their mean and standard deviation.

```
   private static int dropCount;
   private static ListSeries waitTime;
   private static Simulation sim;

   public static void main
      (String[] args)
      {
      if (args.length != 7) usage();
      lambda_L = Double.parseDouble (args[0]);
      lambda_U = Double.parseDouble (args[1]);
      lambda_D = Double.parseDouble (args[2]);
      mu = Double.parseDouble (args[3]);
      Q = Integer.parseInt (args[4]);
      R = Integer.parseInt (args[5]);
      seed = Long.parseLong (args[6]);
```

We initialize the X-Y series objects and the PRNG. Notice that we initialize the PRNG *outside* the loop over λ, not inside the loop. When we go to the next iteration of the loop, we want the PRNG to continue generating new random numbers, not recycle the previously-generated random numbers (as would happen if we reinitialized the PRNG inside the loop).

```
      meanWaitTimeVsLambda = new ListXYSeries();
      dropFractionVsLambda = new ListXYSeries();
      prng = new Random (seed);
      System.out.printf
         ("Mu = %.3f, Q = %d, R = %d, seed = %d%n",
          mu, Q, R, seed);
      System.out.printf ("        --Wait time---  Drop%n");
      System.out.printf ("Lambda  Mean    Stddev  Fraction%n");
```

Here is the loop over λ. Each iteration of the loop does an entire simulation run, the same as the whole WebServer01 program, with a different value of λ.

```
      double lambda;
      for (int r = 0;
            (lambda = lambda_L + r*lambda_D) <= lambda_U;
            ++ r)
         {
         requestPrng = new ExponentialPrng (prng, lambda);
         serverPrng = new ExponentialPrng (prng, mu);
         requestQueue = new LinkedList<Request>();
         requestCount = 0;
         dropCount = 0;
         waitTime = new ListSeries();
         sim = new Simulation();
         generateRequest();
         sim.run();
```

When the current simulation run finishes, we calculate the statistics, add them to the X-Y series objects to be plotted later, and print them out.

```
         Series.Stats wt = waitTime.stats();
         double df = ((double) dropCount)/((double) R);
         meanWaitTimeVsLambda.add (lambda, wt.mean);
         dropFractionVsLambda.add (lambda, df);
         System.out.printf ("%-8.3f%-8.3f%-8.3f%-8.3f%n",
            lambda, wt.mean, wt.stddev, df);
         }
```

After we've done all the simulation runs, we plot the gathered data.

```
      new Plot()
         .plotTitle
            ("WebServer02, \u03BC="+mu+", Q="+Q+", R="+R)
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Mean wait time")
         .xySeries (meanWaitTimeVsLambda)
         .getFrame()
         .setVisible (true);
      new Plot()
         .plotTitle
            ("WebServer02, \u03BC="+mu+", Q="+Q+", R="+R)
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Drop fraction")
         .xySeries (dropFractionVsLambda)
         .getFrame()
         .setVisible (true);
      }
```

Class Request holds the time when the request started. Calling the `waitTime()` method when the request finishes yields the request's wait time.

```
   private static class Request
      {
      private double startTime;
      public Request()
         {
         startTime = sim.time();
         }
      public double waitTime()
         {
         return sim.time() - startTime;
         }
      }

   private static void generateRequest()
      {
      addToQueue (new Request());
      if (++ requestCount < R)
         {
         sim.doAfter (requestPrng.next(), new Event()
            {
            public void perform() { generateRequest(); }
            });
         }
      }
```

Here's where we check whether the request queue is full and, if so, drop the request.

```
   private static void addToQueue
      (Request request)
      {
      if (requestQueue.size() < Q)
         {
         requestQueue.add (request);
         if (requestQueue.size() == 1) startServing();
         }
      else
         {
         ++ dropCount;
         }
      }

   private static void startServing()
      {
      sim.doAfter (serverPrng.next(), new Event()
         {
         public void perform() { removeFromQueue(); }
         });
      }
```

When a request finishes, we remove it from the request queue, determine its wait time, and add that to the wait time series for later use in calculating the mean and standard deviation.

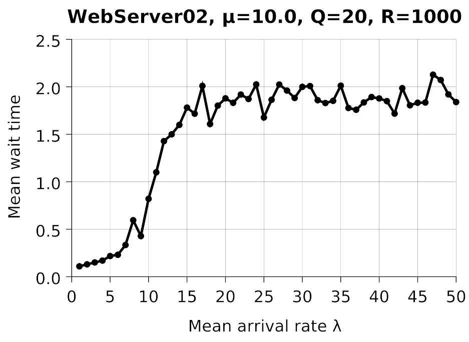
```
   private static void removeFromQueue()
      {
      waitTime.add (requestQueue.removeFirst().waitTime());
      if (requestQueue.size() > 0) startServing();
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java WebServer02 <lambda_L> <lambda_U> "+
          "<lambda_D> <mu> <Q> <R> <seed>");
      System.exit (1);
      }
   }
```

Here's what the program prints with the mean arrival rate λ going from 1 to 50, mean service rate $\mu = 10$, maximum request queue length $Q = 20$, and $R = 1000$ requests per simulation run. Figure 17.1 shows the plots.

```
$ java WebServer02 1 50 1 10 20 1000 142857
Mu = 10.000, Q = 20, R = 1000, seed = 142857
        --Wait time---  Drop
Lambda  Mean    Stddev  Fraction
1.000   0.109   0.117   0.000
2.000   0.130   0.135   0.000
3.000   0.150   0.159   0.000
4.000   0.170   0.160   0.000
5.000   0.218   0.224   0.000
6.000   0.231   0.257   0.000
7.000   0.334   0.324   0.000
8.000   0.595   0.567   0.005
9.000   0.428   0.316   0.000
10.000  0.821   0.516   0.018
11.000  1.100   0.641   0.053
12.000  1.428   0.630   0.123
13.000  1.500   0.591   0.213
14.000  1.599   0.516   0.248
15.000  1.781   0.505   0.340
16.000  1.717   0.492   0.327
17.000  2.010   0.586   0.468
18.000  1.608   0.631   0.348
19.000  1.802   0.587   0.450
20.000  1.879   0.515   0.500
21.000  1.831   0.470   0.473
22.000  1.919   0.506   0.544
23.000  1.872   0.492   0.539
24.000  2.025   0.546   0.620
25.000  1.677   0.438   0.522
26.000  1.864   0.533   0.581
27.000  2.024   0.682   0.657
28.000  1.961   0.493   0.631
29.000  1.883   0.564   0.637
30.000  2.000   0.540   0.670
31.000  2.008   0.567   0.682
32.000  1.859   0.577   0.674
33.000  1.829   0.492   0.660
```

```
34.000  1.852   0.496   0.686
35.000  2.012   0.521   0.715
36.000  1.777   0.455   0.687
37.000  1.759   0.566   0.688
38.000  1.835   0.488   0.712
39.000  1.893   0.469   0.710
40.000  1.878   0.504   0.747
41.000  1.849   0.619   0.735
42.000  1.718   0.547   0.723
43.000  1.986   0.501   0.767
44.000  1.805   0.540   0.756
45.000  1.832   0.561   0.749
46.000  1.835   0.500   0.749
47.000  2.128   0.624   0.793
48.000  2.072   0.626   0.798
49.000  1.920   0.709   0.784
50.000  1.840   0.400   0.775
```



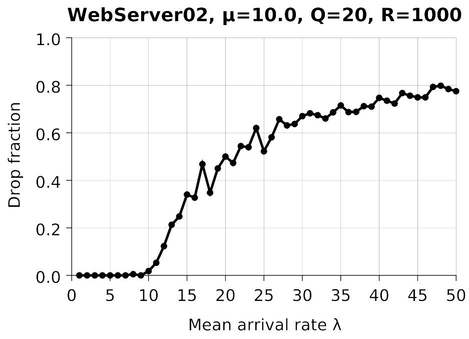


Figure 17.1: Web server mean wait time and drop fraction versus mean arrival rate

As the mean arrival rate approaches the mean service rate, the mean wait time slowly increases, but few or no requests are dropped. The DOS attack commences when the mean arrival rate exceeds the mean service rate. As the mean arrival rate continues to increase, more and more requests are dropped; eventually, nearly 80 percent of them are dropped and denied service. Those requests that do get through experience much longer wait times. Eventually, the queue is almost always full. With 20 requests in the queue and with the server responding to 10 requests per second, it takes about 2 seconds for a request to finish.

Chapter 18

Uniform Distribution

In the next chapter, we'll need to generate random numbers with a **uniform distribution**—that is, random numbers uniformly distributed between arbitrary bounds a and b (not just 0 and 1). The p.d.f. $f(x)$ for a uniformly distributed, continuous random variable x is

$$f(x) = \begin{cases} \dfrac{1}{b-a}\,, & a \le x \le b \\ 0\,, & \text{otherwise} \end{cases} \tag{18.1}$$

which shows that every value x in the interval (a, b) is equally likely. The c.d.f. $F(x)$ for a uniformly distributed random variable x is

$$\begin{aligned} F(x) &= \int_{-\infty}^{x} f(u)\,\mathrm{d}u \\ &= \int_{a}^{x} \frac{1}{b-a}\,\mathrm{d}u \\ &= \frac{x-a}{b-a}\,,\ a \le x \le b \\ &= 0\,,\ x < a \\ &= 1\,,\ x > b \end{aligned} \tag{18.2}$$

For example, Figure 18.1 shows the p.d.f. of a uniform$(3, 7)$ distribution, and Figure 18.2 shows the c.d.f.

To generate values of a random variable x with a uniform(a, b) distribution, we use the technique introduced in Chapter 14: Pick a random number y uniformly distributed between 0 and 1; return the number x such that $F(x) = y$. The formula for the second step is derived as follows:

$$\begin{aligned} F(x) &= y \\ \frac{x-a}{b-a} &= y \\ x - a &= y(b-a) \\ x &= y(b-a) + a \end{aligned} \tag{18.3}$$

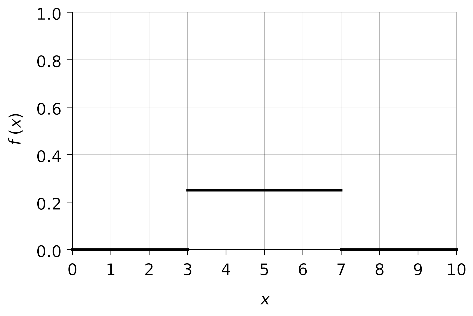


Figure 18.1: Uniform$(3, 7)$ distribution p.d.f.

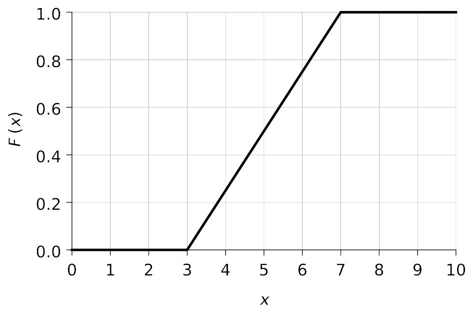


Figure 18.2: Uniform$(3, 7)$ distribution c.d.f.

Here is class edu.rit.numeric.UniformPrng in the Parallel Java 2 Library. This extends class edu.rit.numeric.DoublePrng and implements the `next()` method to generate uniformly distributed random numbers using Equation 18.3. The bounds a and b are specified as constructor arguments.

```
package edu.rit.numeric;
import edu.rit.util.Random;
public class UniformPrng
   extends DoublePrng
   {
   private double a;
   private double b_minus_a;

   /**
    * Construct a new uniform PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    * @param  a               Interval lower bound.
    * @param  b                Interval upper bound.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if <TT>theUniformPrng</TT>
    *     is null.
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if <I>a</I> &ge; <I>b</I>.
    */
   public UniformPrng
      (Random theUniformPrng,
       double a,
       double b)
      {
      super (theUniformPrng);
      if (a >= b)
         {
         throw new IllegalArgumentException
            ("UniformPrng(): a ("+a+") >= b ("+b+") illegal");
         }
      this.a = a;
      this.b_minus_a = b - a;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public double next()
      {
      return myUniformPrng.nextDouble()*b_minus_a + a;
      }
   }
```

Chapter 19

A Distributed Encyclopedia, Part 2

Now we can put all the pieces together from the previous eighteen chapters and build a discrete event simulation program for the P2Pedia distributed encyclopedia system. As in the previous P2Pedia simulations, the network topology consists of N nodes, each connected to its successor and to K other randomly

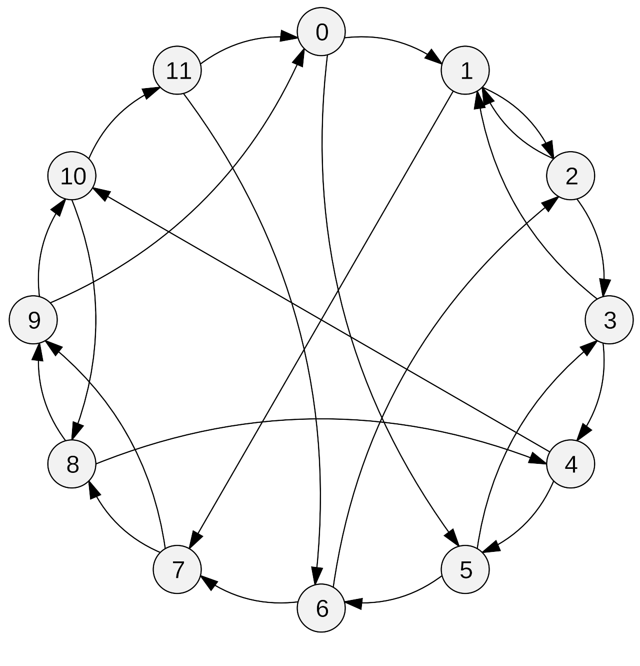


Figure 19.1: Example P2Pedia topology

chosen nodes (Figure 19.1).

We are interested in the mean and standard deviation of the queries' elapsed times. The elapsed time starts when the query arrives at the originating node and finishes when the query is first processed by the matching node. (We are not simulating the messages going between the user and the P2Pedia nodes.)

Each node in the P2Pedia system acts like the web server we simulated. The node has a queue of incoming queries. The queue has a finite maximum size Q; queries arriving when the queue is full are dropped. The node repeatedly processes the first query in the queue. Because queries can be dropped, a query might fail to reach the matching node. We are interested in the failure fraction, the fraction of all queries that did not reach the matching node.

The nodes' query processing differs somewhat from the web server's request processing. There are three cases:

1. If this node has already seen the query, this node does nothing with the query. This case arises because there may be more than one path to this node from the originating node, so multiple copies of the query might arrive at this node as the query floods through the network.

2. Otherwise, if this node is the matching node, the query has been successful. The query's elapsed time finishes at this instant.

3. Otherwise, this node sends the query to each of this node's connected nodes.

We will assume that the processing for cases 1 and 2 is essentially instantaneous, and so does not consume any time in the simulation. However, for case 3, the query messages are sent over the Internet and so are not instantaneous. We will assume that the time to send a query from one node to another is a random variable with a uniform distribution between certain lower and upper bounds. Also, the node sends the queries one at a time, in sequence. The node does not start processing the next query until it has finished sending all the messages for the current query.

We will assume that the queries' interarrival times are exponentially distributed with a certain mean arrival rate λ. We don't need to simulate the users as separate entities. Whether we have one user generating ten queries per second or ten users generating one query per second, the interarrival time distribution is the same, as long as the query times are independent. Each query's originating and matching nodes are chosen uniformly at random from all the nodes.

The P2Pedia discrete event simulation program's software design is more elaborate than the web server's. The web server program had only one request queue, which we declared as a global variable. The P2Pedia program has many nodes, each with its own separate query queue. This argues that we should make a class for a node and create multiple instances of the class. The classes we need are:

- Class **Query** provides a query. It holds the query's start time, so we can calculate the elapsed time; and the query's matching node, so we can tell when the query has reached its destination.
- Class **Node** provides a node. It holds the node's query queue. The web server program's subroutines for processing a request become methods of class Node for processing a query. Class Node also holds a list of the connected nodes, to which non-matching queries are forwarded. Finally, class Node maintains a set of queries that the node has already seen, to detect when to ignore a duplicate query.[1]
- Class **Network** provides a network of multiple nodes. It is responsible for creating all the Node objects, setting up the connections between them, and choosing the queries' originating and matching nodes at random. Thus, the network graph is implemented as one Network object plus many Node objects each with a list of adjacent nodes, which is essentially an adjacency list representation.

Here is version 7 of the P2Pedia simulation program.

```
import edu.rit.numeric.DoublePrng;
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.UniformPrng;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.HashSet;
import java.util.LinkedList;
public class P2Pedia07
   {
   // Knobs
   private static int N; // Number of nodes
   private static int K; // Number of extra connections per node
   private static double lambda; // Mean query arrival rate
   private static double tx_L; // Query transmit time lower bound
   private static double tx_U; // Query transmit time upper bound
   private static int Q; // Maximum queries in query queue
   private static int R; // Number of queries to simulate
   private static long seed; // PRNG seed

   // Global variables
```

[1]This wouldn't work in a real system, because the set of queries grows without bound. This is fine for the simulation, though, as we will only be simulating a fixed number of queries.

```
   private static Random prng;
   private static DoublePrng queryPrng;
   private static DoublePrng txPrng;
   private static int queryCount;
   private static ListSeries elapsedTimes;
   private static int successCount;
   private static Simulation sim;
   private static Network network;
```

Here is the **Query** class.

```
   private static class Query
      {
      private int id;
      private Node matchingNode;
      private double startTime;

      public Query
         (int id,
          Node matchingNode)
         {
         this.id = id;
         this.matchingNode = matchingNode;
         this.startTime = sim.time();
         }

      public boolean matches
         (Node node)
         {
         return node == matchingNode;
         }

      public double elapsedTime()
         {
         return sim.time() - startTime;
         }
```

We are going to be adding queries to a set, which will be an instance of class edu.rit.util.HashSet. This means the query object will be the key in a hashed data structure. This in turn means the query class must override Java's `equals()` and `hashCode()` methods. The `equals()` method must return true if this query object represents the same key value as the given object. When a query is generated, it is given a unique ID, so two query objects represent the same key value if they have the same ID.

```
      public boolean equals
         (Object obj)
         {
         return
            obj instanceof Query &&
            ((Query) obj).id == this.id;
         }
```

If two query objects represent the same key value, then their `hashCode()` methods must return the same integer value; otherwise, their `hashCode()` methods should (but don't strictly have to) return different integer values. Returning the query ID guarantees this.

```
      public int hashCode()
         {
         return id;
         }

      public String toString()
         {
         return "Query " + id;
         }
      }
```

Here is the **Node** class.

```
   private static class Node
      {
      private int id;
      private LinkedList<Node> connectedNodes =
         new LinkedList<Node>();
      private LinkedList<Query> queryQueue =
         new LinkedList<Query>();
      private HashSet<Query> priorQueries =
         new HashSet<Query>();

      public Node
         (int id)
         {
         this.id = id;
         }
```

Class Network uses the `connectTo()` method to hook nodes together.

```
      public void connectTo
         (Node node)
         {
         connectedNodes.add (node);
         }
```

The `addToQueue()` method is called when this node receives an incoming query. The query is dropped if the query queue is full; the query is added to the query queue if it isn't full; if the query queue had been empty, the node starts serving the query.

```
      public void addToQueue
         (Query query)
         {
         if (queryQueue.size() < Q)
            {
            System.out.printf ("%.3f %s received by %s%n",
               sim.time(), query, this);
            queryQueue.add (query);
            if (queryQueue.size() == 1) startServing();
            }
         else
            {
            System.out.printf ("%.3f %s dropped by %s%n",
               sim.time(), query, this);
            }
         }
```

The `startServing()` method is called to process the first query in the queue.

```
      private void startServing()
         {
         final Query query = queryQueue.getFirst();
```

Case 1: This node has already seen the query, so we do nothing.

```
         if (priorQueries.contains (query))
            {
            System.out.printf ("%.3f %s previously seen by %s%n",
               sim.time(), query, this);
            removeFromQueue();
            }
```

Case 2: This node is the matching node. We record the query's elapsed time and increment the number of successful queries.

```
         else if (query.matches (this))
            {
            System.out.printf ("%.3f %s matches at %s%n",
               sim.time(), query, this);
            elapsedTimes.add (query.elapsedTime());
            ++ successCount;
            removeFromQueue();
            }
```

Case 3: This node is not the matching node, so we send the query to each connected node. This requires adding several future events to the simulation. When each event occurs, one connected node's `addToQueue()` method is called to cause the node to receive the query. The first event occurs at the present time plus the message send time; each subsequent event occurs at the previous event's time plus the message send time; this simulates the node sending the query messages sequentially. The message send times are obtained from the PRNG with a uniform distribution. One final event occurs at the same time as the last node receives the query; the event causes this node to remove the query from the queue.

```
         else
            {
            System.out.printf ("%.3f %s sent from %s to",
               sim.time(), query, this);
            double txTime = 0.0;
            for (Node nextNode : connectedNodes)
               {
               System.out.printf (" %s", nextNode);
               final Node nn = nextNode;
               txTime += txPrng.next();
               sim.doAfter (txTime, new Event()
                  {
                  public void perform()
                     {
                     nn.addToQueue (query);
                     }
                  });
               }
            System.out.println();
            sim.doAfter (txTime, new Event()
               {
               public void perform()
                  {
                  removeFromQueue();
                  }
               });
         }
      }
```

The `removeFromQueue()` method is called to remove the first query from the queue when finished serving it. We also add the query to the set of previously-seen queries, and we start serving the next query in the queue, if any.

```
      private void removeFromQueue()
         {
         Query query = queryQueue.removeFirst();
         System.out.printf ("%.3f %s removed from %s%n",
            sim.time(), query, this);
         priorQueries.add (query);
         if (queryQueue.size() > 0) startServing();
         }

      public String toString()
         {
         return "Node " + id;
         }
      }
```

Here is the **Network** class. The constructor sets up an array of N nodes and establishes the connections.

```
   private static class Network
      {
      private Node[] node;

      public Network()
         {
         node = new Node[N];
         for (int i = 0; i < N; ++ i) node[i] = new Node (i);
         for (int i = 0; i < N; ++ i)
            {
            node[i].connectTo (node[(i + 1)%N]);
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 0; j < K; ++ j)
               {
               node[i].connectTo (node[rs.next()]);
               }
            }
         }
```

The `generateQuery()` method is called to inject a new query into the system. We pick originating and matching nodes at random; set up a new query object with a unique ID; make the originating node receive the query; and, if more queries need to be simulated, add a future event to the simulation to generate the next query. The event will take place after the query interarrival time, which is obtained from the PRNG with an exponential distribution.

```
      public void generateQuery()
         {
         Node origNode = randomNode();
         Node matchNode = randomNode();
         Query query = new Query (++ queryCount, matchNode);
         System.out.printf
            ("%.3f %s generated, originating %s, matching %s%n",
             sim.time(), query, origNode, matchNode);
         origNode.addToQueue (query);
         if (queryCount < R)
            {
            sim.doAfter (queryPrng.next(), new Event()
               {
               public void perform()
                  {
                  generateQuery();
                  }
               });
            }
         }

      private Node randomNode()
         {
         return node[prng.nextInt (node.length)];
         }
      }
```

The simulation main program sets up the knobs and global variables, generates the first query, runs the simulation, and calculates and prints statistics.

```
   public static void main
      (String[] args)
      {
      // Knobs
      if (args.length != 8) usage();
      N = Integer.parseInt (args[0]);
      K = Integer.parseInt (args[1]);
      lambda = Double.parseDouble (args[2]);
      tx_L = Double.parseDouble (args[3]);
      tx_U = Double.parseDouble (args[4]);
      Q = Integer.parseInt (args[5]);
      R = Integer.parseInt (args[6]);
      seed = Long.parseLong (args[7]);

      // Global variables
      prng = new Random (seed);
      queryPrng = new ExponentialPrng (prng, lambda);
      txPrng = new UniformPrng (prng, tx_L, tx_U);
      queryCount = 0;
      elapsedTimes = new ListSeries();
      successCount = 0;
      sim = new Simulation();
      network = new Network();

      // Run simulation
      network.generateQuery();
      sim.run();
      Series.Stats etStats = elapsedTimes.stats();
      double ff = 1.0 - (double)successCount/(double)R;
      System.out.printf ("Elapsed time mean = %.3f%n",
         etStats.mean);
      System.out.printf ("Elapsed time stddev = %.3f%n",
         etStats.stddev);
      System.out.printf ("Failure fraction = %.3f%n", ff);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia07 <N> <K> <lambda> <tx_L> "+
          "<tx_U> <Q> <R> <seed>");
      System.exit (1);
      }
   }
```

Here is a P2Pedia07 simulation program run with $N = 10$ nodes, $K = 1$ extra connection per node, query arrivals exponentially distributed at a mean rate of $\lambda = 0.1$ queries per second, message send times uniformly distributed between 0.25 and 1.25 seconds, maximum queue length $Q = 20$ queries, simulating $R =$ just one query.

```
$ java P2Pedia07 10 1 0.1 0.25 1.25 20 1 142857
0.000 Query 1 generated, originating Node 8, matching Node 5
0.000 Query 1 received by Node 8
0.000 Query 1 sent from Node 8 to Node 9 Node 7
0.490 Query 1 received by Node 9
```

```
0.490 Query 1 sent from Node 9 to Node 0 Node 3
0.836 Query 1 received by Node 0
0.836 Query 1 sent from Node 0 to Node 1 Node 4
0.952 Query 1 removed from Node 8
0.952 Query 1 received by Node 7
0.952 Query 1 sent from Node 7 to Node 8 Node 2
1.480 Query 1 removed from Node 9
1.480 Query 1 received by Node 3
1.480 Query 1 sent from Node 3 to Node 4 Node 8
1.903 Query 1 received by Node 1
1.903 Query 1 sent from Node 1 to Node 2 Node 5
2.161 Query 1 received by Node 8
2.161 Query 1 previously seen by Node 8
2.161 Query 1 removed from Node 8
2.490 Query 1 received by Node 2
2.490 Query 1 sent from Node 2 to Node 3 Node 9
2.490 Query 1 removed from Node 7
2.554 Query 1 received by Node 4
2.554 Query 1 sent from Node 4 to Node 5 Node 1
2.919 Query 1 removed from Node 3
2.919 Query 1 received by Node 8
2.919 Query 1 previously seen by Node 8
2.919 Query 1 removed from Node 8
2.923 Query 1 received by Node 2
3.062 Query 1 received by Node 4
3.062 Query 1 removed from Node 0
3.653 Query 1 received by Node 5
3.653 Query 1 matches at Node 5
3.653 Query 1 removed from Node 5
3.722 Query 1 received by Node 3
3.722 Query 1 previously seen by Node 3
3.722 Query 1 removed from Node 3
3.932 Query 1 removed from Node 1
3.932 Query 1 received by Node 5
3.932 Query 1 previously seen by Node 5
3.932 Query 1 removed from Node 5
4.300 Query 1 received by Node 1
4.300 Query 1 previously seen by Node 1
4.300 Query 1 removed from Node 1
4.300 Query 1 removed from Node 4
4.300 Query 1 previously seen by Node 4
4.300 Query 1 removed from Node 4
4.951 Query 1 received by Node 9
4.951 Query 1 previously seen by Node 9
4.951 Query 1 removed from Node 9
```

```
4.951 Query 1 removed from Node 2
4.951 Query 1 previously seen by Node 2
4.951 Query 1 removed from Node 2
Elapsed time mean = 3.653
Elapsed time stddev = 0.000
Failure fraction = 0.000
```

In this simulation's network topology there happen to be two minimal-length 4-hop paths from originating node 8 to matching node 5, namely 8–9–0–1–5 and 8–9–3–4–5. However, the query happens to reach the matching node more quickly along the first path (3.653 seconds) than the second (3.932 seconds). Notice also how the query is received and processed by almost all the nodes in the network, and how the query arrives more than once at each node.

Here is a P2Pedia07 simulation program run with the same knob settings, except 1000 queries are simulated. (The event transcript is omitted.)

```
$ java P2Pedia07 10 1 0.1 0.25 1.25 20 1000 142857 | tail -3
Elapsed time mean = 2.283
Elapsed time stddev = 1.343
Failure fraction = 0.000
```

We know from our previous simulations that when $K = 1$, the query path length is about $\log_2 N$, or 3 hops. The average time to send a query across one hop is halfway between the uniform distribution limits, or 0.75 seconds. Therefore, the mean elapsed time should be about $3 \cdot 0.75 = 2.25$ seconds. So the measured mean elapsed time of 2.283 seconds looks reasonable. Because the mean query interarrival time—one query every 10 seconds—is so much longer than the message transmission times, the query queues are never full, none of the queries are dropped, and the failure fraction is 0.

Chapter 20

Encyclopedia: Scalable?

Now let's build a discrete event simulation program to investigate the P2Pedia distributed encyclopedia system's performance as the number of users, and therefore the query arrival rate, scales up. We will simulate a series of mean query arrival rates λ, from λ_L to λ_U in steps of λ_Δ, and plot the mean query elapsed time versus λ and the query failure fraction versus λ. For each value of λ, we will simulate a number R_1 of random network topologies, so we can average over many different topologies. For each topology, we will simulate a number R_2 of queries. Thus, we will simulate $R_1 \cdot R_2$ queries for each value of λ. The simulation's other characteristics are the same as in the previous version.

Here is version 8 of the P2Pedia simulation program. The Query, Node, and Network classes are the same, except the event transcript printouts are omitted. The simulation itself is repeated inside a pair of nested loops, the outer loop over the λ values, the inner loop over the R_1 network topologies.

```
import edu.rit.numeric.DoublePrng;
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.UniformPrng;
import edu.rit.numeric.plot.Plot;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.HashSet;
import java.util.LinkedList;
public class P2Pedia08
   {
   // Knobs
   private static int N; // Number of nodes
   private static int K; // Number of extra connections per node
   private static double lambda_L; // First query arrival rate
   private static double lambda_U; // Last query arrival rate
   private static double lambda_D; // Query arrival rate increment
   private static double tx_L; // Query transmit time lower bound
   private static double tx_U; // Query transmit time upper bound
   private static int Q; // Maximum queries in query queue
   private static int R1; // Number of topologies per arrival rate
   private static int R2; // Number of queries per topology
   private static long seed; // PRNG seed

   // Global variables
   private static ListXYSeries meanElapsedTimeVsLambda;
   private static ListXYSeries failureFractionVsLambda;
   private static Random prng;
   private static DoublePrng txPrng;
   private static DoublePrng queryPrng;
   private static ListSeries elapsedTimes;
   private static int successCount;
   private static int queryCount;
   private static Simulation sim;
   private static Network network;

   private static class Query
      {
      private int id;
      private Node matchingNode;
      private double startTime;

      public Query
         (int id,
          Node matchingNode)
         {
         this.id = id;
         this.matchingNode = matchingNode;
         this.startTime = sim.time();
         }

      public boolean matches
         (Node node)
         {
         return node == matchingNode;
         }

      public double elapsedTime()
         {
         return sim.time() - startTime;
         }

      public boolean equals
         (Object obj)
         {
         return
            obj instanceof Query &&
            ((Query) obj).id == this.id;
         }

      public int hashCode()
         {
         return id;
         }
      }

   private static class Node
      {
      private int id;
      private LinkedList<Node> connectedNodes =
         new LinkedList<Node>();
      private LinkedList<Query> queryQueue =
         new LinkedList<Query>();
      private HashSet<Query> priorQueries =
         new HashSet<Query>();

      public Node
         (int id)
         {
         this.id = id;
         }

      public void connectTo
         (Node node)
         {
         connectedNodes.add (node);
         }

      public void addToQueue
         (Query query)
         {
         if (queryQueue.size() < Q)
            {
```

```
         queryQueue.add (query);
         if (queryQueue.size() == 1) startServing();
         }
      }

   private void startServing()
      {
      final Query query = queryQueue.getFirst();
      if (priorQueries.contains (query))
         {
         removeFromQueue();
         }
      else if (query.matches (this))
         {
         elapsedTimes.add (query.elapsedTime());
         ++ successCount;
         removeFromQueue();
         }
      else
         {
         double txTime = 0.0;
         for (Node nextNode : connectedNodes)
            {
            final Node nn = nextNode;
            txTime += txPrng.next();
            sim.doAfter (txTime, new Event()
               {
               public void perform()
                  {
                  nn.addToQueue (query);
                  }
               });
            }
         sim.doAfter (txTime, new Event()
            {
            public void perform()
               {
               removeFromQueue();
               }
            });
         }
      }

   private void removeFromQueue()
      {
      Query query = queryQueue.removeFirst();
      priorQueries.add (query);
      if (queryQueue.size() > 0) startServing();
      }
   }

private static class Network
   {
   private Node[] node;

   public Network()
      {
      node = new Node[N];
      for (int i = 0; i < N; ++ i) node[i] = new Node (i);
      for (int i = 0; i < N; ++ i)
         {
         node[i].connectTo (node[(i + 1)%N]);
         RandomSubset rs = new RandomSubset (prng, N)
            .remove (i) .remove ((i + 1)%N);
         for (int j = 0; j < K; ++ j)
            {
            node[i].connectTo (node[rs.next()]);
            }
         }
      }

   public void generateQuery()
      {
      Node origNode = randomNode();
      Node matchNode = randomNode();
      Query query = new Query (++ queryCount, matchNode);
      origNode.addToQueue (query);
      if (queryCount < R2)
         {
         sim.doAfter (queryPrng.next(), new Event()
            {
            public void perform()
               {
               generateQuery();
               }
            });
         }
      }

   private Node randomNode()
```

```
         {
         return node[prng.nextInt (node.length)];
         }
      }

   public static void main
      (String[] args)
      {
      // Knobs
      if (args.length != 11) usage();
      N = Integer.parseInt (args[0]);
      K = Integer.parseInt (args[1]);
      lambda_L = Double.parseDouble (args[2]);
      lambda_U = Double.parseDouble (args[3]);
      lambda_D = Double.parseDouble (args[4]);
      tx_L = Double.parseDouble (args[5]);
      tx_U = Double.parseDouble (args[6]);
      Q = Integer.parseInt (args[7]);
      R1 = Integer.parseInt (args[8]);
      R2 = Integer.parseInt (args[9]);
      seed = Long.parseLong (args[10]);

      // Global variables
      meanElapsedTimeVsLambda = new ListXYSeries();
      failureFractionVsLambda = new ListXYSeries();
      prng = new Random (seed);
      txPrng = new UniformPrng (prng, tx_L, tx_U);

      // Loop over all mean arrival rates.
      System.out.printf ("        -Elapsed Time-  Failure%n");
      System.out.printf ("Lambda  Mean    Stddev  Fraction%n");
      double lambda;
      for (int i = 0;
            (lambda = lambda_L + i*lambda_D) <= lambda_U; ++ i)
         {
         // Global variables
         queryPrng = new ExponentialPrng (prng, lambda);
         elapsedTimes = new ListSeries();
         successCount = 0;

         // Loop over R1 random topologies.
         for (int j = 0; j < R1; ++ j)
            {
            queryCount = 0;
            sim = new Simulation();
            network = new Network();

            // Run simulation
            network.generateQuery();
            sim.run();
            }

         // Print statistics.
         Series.Stats etStats = elapsedTimes.stats();
         double ff = 1.0 - (double)successCount/(double)(R1*R2);
         System.out.printf ("%-8.3f%-8.3f%-8.3f%-8.3f%n",
            lambda, etStats.mean, etStats.stddev, ff);
         meanElapsedTimeVsLambda.add (lambda, etStats.mean);
         failureFractionVsLambda.add (lambda, ff);
         }

      // Plot results.
      new Plot()
         .plotTitle ("P2Pedia08")
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Mean elapsed time")
         .xySeries (meanElapsedTimeVsLambda)
         .getFrame()
         .setVisible (true);
      new Plot()
         .plotTitle ("P2Pedia08")
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Failure fraction")
         .xySeries (failureFractionVsLambda)
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia08 <N> <K> <lambda_L> <lambda_U> "+
          "<lambda_D> <tx_L> <tx_U> <Q> <R1> <R2> <seed>");
      System.exit (1);
      }
   }
```

Here's what the program prints with a 100-node network, one extra connection per node, mean arrival rate λ going from 0.1 to 10 in steps of 0.1, message transmission time uniformly distributed between 0.25 and 1.25, maximum query queue length 20, 100 topologies per λ value, and 1000 queries per topology. Figure 20.1 shows the plots.

```
$ java P2Pedia08 100 1 0.1 10 0.1 0.25 1.25 20 100 1000 142857
        -Elapsed Time-  Failure
Lambda  Mean    Stddev  Fraction
0.100   6.169   2.282   0.000
0.200   6.802   2.602   0.000
0.300   7.711   3.082   0.000
0.400   9.055   3.867   0.000
0.500   11.455  5.304   0.001
0.600   17.677  9.545   0.004
0.700   38.198  20.108  0.039
0.800   60.930  30.085  0.112
0.900   78.020  39.156  0.179
1.000   91.478  46.705  0.241
1.100   100.468 53.330  0.288
1.200   108.277 58.915  0.329
1.300   114.669 63.296  0.363
1.400   120.458 67.900  0.390
1.500   124.488 71.118  0.415
1.600   128.616 74.414  0.436
1.700   131.894 77.955  0.459
1.800   136.631 81.274  0.475
1.900   138.724 82.976  0.486
2.000   140.374 84.547  0.500
2.100   142.972 87.054  0.512
2.200   145.667 89.108  0.522
2.300   146.978 89.740  0.532
2.400   149.846 92.766  0.540
2.500   151.120 93.301  0.551
2.600   151.932 94.128  0.556
2.700   154.140 96.292  0.563
2.800   154.942 96.777  0.568
2.900   157.133 98.755  0.573
3.000   158.505 99.798  0.581
3.100   159.603 100.874 0.581
3.200   160.621 101.770 0.589
3.300   161.976 103.099 0.589
3.400   162.988 103.132 0.596
3.500   163.514 103.547 0.601
3.600   165.394 105.089 0.601
3.700   165.794 105.254 0.606
3.800   167.303 106.188 0.610
3.900   167.462 106.638 0.612
4.000   168.922 107.683 0.614
4.100   169.492 108.242 0.616
```

```
4.200   169.697 108.202 0.621
4.300   172.269 109.413 0.622
4.400   171.650 109.713 0.622
4.500   172.537 109.979 0.627
4.600   173.437 110.220 0.630
4.700   174.530 110.981 0.630
4.800   176.219 111.722 0.633
4.900   175.778 111.848 0.633
5.000   177.079 112.635 0.634
5.100   177.096 112.678 0.638
5.200   178.414 113.103 0.638
5.300   178.802 113.288 0.640
5.400   178.766 113.260 0.640
5.500   179.891 113.906 0.642
5.600   180.456 114.341 0.642
5.700   179.909 114.151 0.644
5.800   180.744 113.979 0.649
5.900   180.332 114.286 0.647
6.000   183.309 115.130 0.647
6.100   181.847 114.787 0.650
6.200   183.138 115.708 0.649
6.300   184.054 116.079 0.648
6.400   183.557 115.915 0.650
6.500   184.132 115.955 0.652
6.600   182.860 115.416 0.654
6.700   185.304 116.483 0.655
6.800   186.612 117.073 0.653
6.900   186.285 116.583 0.655
7.000   186.938 117.896 0.654
7.100   186.288 116.872 0.655
7.200   187.170 116.823 0.655
7.300   186.857 117.635 0.657
7.400   187.294 117.440 0.658
7.500   188.251 117.565 0.658
7.600   188.550 117.915 0.659
7.700   189.212 118.792 0.659
7.800   188.405 117.911 0.660
7.900   189.689 118.504 0.660
8.000   190.228 118.818 0.661
8.100   189.913 118.266 0.663
8.200   190.973 118.735 0.661
8.300   189.402 118.561 0.662
8.400   190.939 118.677 0.662
8.500   190.113 118.398 0.662
8.600   191.325 118.935 0.665
```

```
8.700   191.027 118.834 0.664
8.800   192.229 119.380 0.663
8.900   193.357 119.959 0.665
9.000   192.633 119.321 0.666
9.100   192.441 119.025 0.667
9.200   191.479 118.933 0.666
9.300   192.448 119.108 0.665
9.400   193.899 119.768 0.666
9.500   194.537 119.879 0.667
9.600   194.114 119.786 0.667
9.700   194.314 119.823 0.669
9.800   194.729 119.652 0.669
9.900   194.895 119.502 0.667
10.000  194.916 120.284 0.667
```

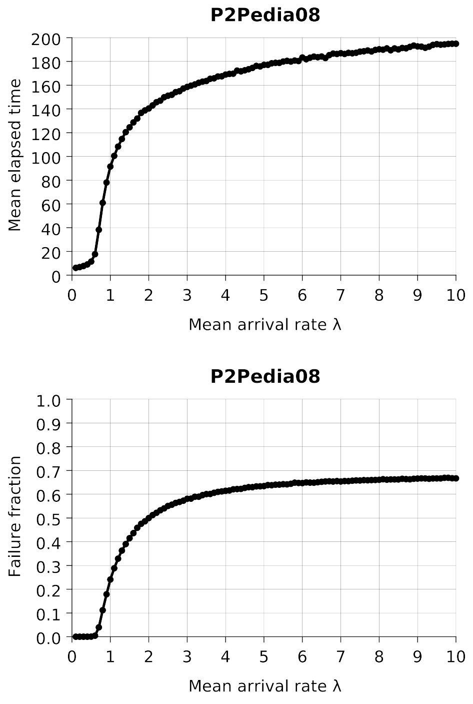


Figure 20.1: P2Pedia mean query elapsed time and query failure fraction versus mean query arrival rate

The data shows that the 100-node P2Pedia system scales well up to about 0.6 queries per second. Above that, the query elapsed times and failure fractions shoot up quickly and continue increasing. Eventually, nearly 70 percent of the queries fail. This behavior is due primarily to P2Pedia's flood routing algorithm, which sends multiple copies of every query to almost every node in the system, clogging up all the query queues and causing many queries to be lost. It's a good thing we unearthed this behavior during simulation, before actually building the P2Pedia system. To be practical, a peer-to-peer distributed encyclopedia is going to have to use some other query routing algorithm, one that doesn't generate such a multiplicity of messages. We can continue to simulate different P2Pedia design alternatives until we find one that works.

Appendix A

Source Code Listings

The source code for each Java class discussed in the preceding chapters is listed here in its entirety, without any explanatory text interspersed.

When compiling and running these programs, you must set the Java class path to include the current directory plus the Parallel Java 2 Library. For further information, see the Parallel Java 2 Library documentation (Javadoc).

A.1 A Dice Game

A.1.1 Class Dice01

```
import edu.rit.util.Random;
public class Dice01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 1) usage();
      long seed = Long.parseLong (args[0]);
      Random prng = new Random (seed);
      int win = 0;
      int die;
      do
         {
         ++ win;
         die = prng.nextInt (6) + 1;
         System.out.printf ("%d%n", die);
         }
      while (die != 6);
      System.out.printf ("Winnings = $%d%n", win);
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice01 <seed>");
      System.exit (1);
      }
   }
```

A.1.2 Class Dice02

```
import edu.rit.util.Random;
public class Dice02
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
      Random prng = new Random (seed);
      int win = 0;
      int die;
      for (int trial = 0; trial < N; ++ trial)
         {
         do
            {
            ++ win;
            die = prng.nextInt (6) + 1;
            }
         while (die != 6);
         }
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice02 <N> <seed>");
      System.exit (1);
      }
   }
```

A.2 Bernoulli Distribution

A.2.1 Class Dice03

```
import edu.rit.util.Random;
public class Dice03
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
      Random prng = new Random (seed);
      int win = 0;
      for (int trial = 0; trial < N; ++ trial)
         {
         do
            {
            ++ win;
            }
         while (prng.nextDouble() < 5.0/6.0);
         }
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice03 <N> <seed>");
      System.exit (1);
      }
   }
```

A.2.2 Class edu.rit.numeric.BernoulliPrng

```
package edu.rit.numeric;
import edu.rit.util.Random;
public class BernoulliPrng
   {
   private final Random myUniformPrng;
   private final double myP;

   public BernoulliPrng
      (Random theUniformPrng,
       double p)
      {
      if (theUniformPrng == null)
         {
         throw new NullPointerException
            ("BernoulliPrng(): theUniformPrng is null");
         }
      if (0.0 > p || p > 1.0)
         {
         throw new IllegalArgumentException
            ("BernoulliPrng(): p = "+p+" illegal");
         }
      myUniformPrng = theUniformPrng;
      myP = p;
      }

   public boolean next()
      {
      return myUniformPrng.nextDouble() < myP;
      }
   }
```

A.2.3 Class Dice04

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.util.Random;
public class Dice04
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 2) usage();
      int N = Integer.parseInt (args[0]);
      long seed = Long.parseLong (args[1]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      int win = 0;
      for (int trial = 0; trial < N; ++ trial)
         {
         do
            {
            ++ win;
            }
         while (prng.next());
         }
      System.out.printf ("Average winnings = $%.2f%n",
         ((double) win)/((double) N));
      }

   private static void usage()
      {
      System.err.println ("Usage: java Dice04 <N> <seed>");
      System.exit (1);
      }
   }
```

A.3 Gathering and Displaying Data

A.3.1 Class Dice05

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.util.Random;
public class Dice05
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      int[] wincount = new int [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
         int win = 0;
         do
            {
            ++ win;
            }
         while (prng.next());
         ++ wincount[Math.min (win - 1, maxwin)];
         }
      System.out.printf ("Win\tCount%n");
      for (int i = 0; i <= maxwin; ++ i)
         {
         System.out.printf ("%d\t%d\n", i + 1, wincount[i]);
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice05 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

A.3.2 Class Dice06

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.numeric.plot.Plot;
import edu.rit.util.Random;
public class Dice06
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      double[] wincount = new double [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
         int win = 0;
         do
            {
            ++ win;
            }
         while (prng.next());
         wincount[Math.min (win - 1, maxwin)] += 1;
         }
      double[] winamount = new double [maxwin + 1];
      for (int i = 0; i <= maxwin; ++ i) winamount[i] = i + 1;
      new Plot()
         .plotTitle ("Dice Game")
         .xAxisTitle ("Winnings")
         .yAxisTitle ("Occurrences")
         .xySeries (winamount, wincount)
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice06 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

A.4 Chi-Square Test

A.4.1 Class Dice07

```
import edu.rit.numeric.BernoulliPrng;
import edu.rit.numeric.Statistics;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
public class Dice07
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 3) usage();
      int maxwin = Integer.parseInt (args[0]);
      int N = Integer.parseInt (args[1]);
      long seed = Long.parseLong (args[2]);
      BernoulliPrng prng =
         new BernoulliPrng (new Random (seed), 5.0/6.0);
      double[] actual = new double [maxwin + 1];
      for (int trial = 0; trial < N; ++ trial)
         {
         int win = 0;
         do
            {
            ++ win;
            }
         while (prng.next());
         actual[Math.min (win - 1, maxwin)] += 1;
         }
      double[] winamount = new double [maxwin + 1];
      for (int i = 0; i <= maxwin; ++ i) winamount[i] = i + 1;
      double[] expected = new double [maxwin + 1];
      double sum_p_i = 0.0;
      System.out.printf ("Win\tActual\tExpected%n");
      for (int i = 0; i < maxwin; ++ i)
         {
         double p_i = Math.pow (5.0/6.0, i)/6.0;
         sum_p_i += p_i;
         expected[i] = N*p_i;
         System.out.printf ("%.0f\t%.0f\t%.3f%n",
            winamount[i], actual[i], expected[i]);
         }
      expected[maxwin] = N*(1.0 - sum_p_i);
      System.out.printf ("%.0f\t%.0f\t%.3f%n",
         winamount[maxwin], actual[maxwin], expected[maxwin]);
      double chisqr = Statistics.chiSquareTest (actual, expected);
      double pvalue = Statistics.chiSquarePvalue (maxwin, chisqr);
      System.out.printf ("Chi^2   = %.6f%n", chisqr);
      System.out.printf ("P-value = %.6f%n", pvalue);
      new Plot()
         .plotTitle ("Dice Game")
         .xAxisTitle ("Winnings")
         .yAxisTitle ("Occurrences")
         .seriesStroke (null)
         .xySeries (winamount, actual)
         .seriesDots (null)
         .seriesStroke (Strokes.solid (1))
         .xySeries (winamount, expected)
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java Dice07 <maxwin> <N> <seed>");
      System.exit (1);
      }
   }
```

A.5 A Distributed Encyclopedia

No classes

A.6 Graphs

A.6.1 Class P2Pedia01

```
import edu.rit.util.Random;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            for (int j = 1; j <= K; ++ j)
               {
               // conn[i][j] = random node
               }
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         sumPathLength +=
            shortestPathLength (conn, orig, match);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }
   private static int shortestPathLength
      (int[][] conn,
       int orig,
       int match)
      {
      return 0; // STUB
      }

   private static void usage()
```

```
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.7 Random Subsets and Permutations

A.7.1 Class edu.rit.util.RandomSubset

```
package edu.rit.util;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
public class RandomSubset
   implements Iterator<Integer>
   {
   // The underlying PRNG.
   private Random prng;

   // The size of the original set.
   private int N;

   // The number of random subset elements returned so far.
   private int M;

   // A sparse array containing a permutation of the integers
   // from 0 to N-1. Implemented as a mapping from array index to
   // array element. If an array index is not in the map, the
   // corresponding array element is the same as the array index.
   private HashMap<Integer,Integer> permutation =
      new HashMap<Integer,Integer>();

   // Returns the element in the permutation array at index i.
   private int getElement
      (int i)
      {
      Integer element = permutation.get (i);
      return element == null ? i : element;
      }

   // Sets the element in the permutation array at index i to the
   // given value.
   private void setElement
      (int i,
       int value)
      {
      if (value == i)
         {
         permutation.remove (i);
         }
      else
         {
         permutation.put (i, value);
         }
      }

   // Swaps the elements in the permutation array at indexes i
   // and j.
   private void swapElements
      (int i,
       int j)
      {
      int tmp = getElement (i);
      setElement (i, getElement (j));
      setElement (j, tmp);
      }

   // Returns the index in the permutation array at which the
   // given value resides.
   private int indexOf
      (int value)
      {
      for (Map.Entry<Integer,Integer> e : permutation.entrySet())
         {
         if (e.getValue() == value) return e.getKey();
         }
      return value;
      }

   public RandomSubset
      (Random prng,
       int N)
      {
      if (prng == null)
         {
         throw new NullPointerException
            ("RandomSubset(): prng is null");
         }
      if (N < 0)
         {
         throw new IllegalArgumentException
            ("RandomSubset(): N = "+N+" illegal");
         }
      this.prng = prng;
      this.N = N;
      }

   public boolean hasNext()
      {
      return M < N;
      }

   public Integer next()
      {
      if (M >= N)
         {
         throw new NoSuchElementException
            ("RandomSubset.next(): No further elements");
         }
      swapElements (M, M + prng.nextInt (N - M));
      ++ M;
      return getElement (M - 1);
      }

   public void remove()
      {
      throw new UnsupportedOperationException();
      }

   public RandomSubset remove
      (int i)
      {
      if (0 > i || i >= N)
         {
         throw new IllegalArgumentException
            ("RandomSubset.remove(): i = "+i+" illegal");
         }
      int j = indexOf (i);
      if (j >= M)
         {
         swapElements (M, j);
         ++ M;
         }
      return this;
      }
   }
```

A.7.2 Class P2Pedia01

```
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         sumPathLength +=
            shortestPathLength (conn, orig, match);
         System.out.printf ("Trial %d%n", trial);
         System.out.printf ("  Node  Connected to nodes%n");
         for (int i = 0; i < N; ++ i)
            {
            System.out.printf ("  %-4d", i);
            for (int j = 0; j <= K; ++ j)
               {
               System.out.printf ("  %d", conn[i][j]);
               }
            System.out.println();
            }
         System.out.printf ("  Originating node = %d%n", orig);
         System.out.printf ("  Matching node = %d%n", match);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }

   private static int shortestPathLength
      (int[][] conn,
       int orig,
       int match)
      {
      return 0; // STUB
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.8 Breadth-First Search

A.8.1 Class P2Pedia01

```
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia01
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int[] dist = new int[N];
      int sumPathLength = 0;
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         int len = shortestPathLength (conn, dist, orig, match);
         sumPathLength += len;
         System.out.printf ("Trial %d%n", trial);
         System.out.printf ("  Node  Connected to nodes%n");
         for (int i = 0; i < N; ++ i)
            {
            System.out.printf ("  %-4d", i);
            for (int j = 0; j <= K; ++ j)
               {
               System.out.printf ("  %d", conn[i][j]);
               }
            System.out.println();
            }
         System.out.printf ("  Originating node = %d%n", orig);
         System.out.printf ("  Matching node = %d%n", match);
         System.out.printf ("  Query path length = %d%n", len);
         }
      System.out.printf ("Average query path length = %.2f%n",
         ((double) sumPathLength)/((double) T));
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia01 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.9 Mean, Median, and the Like

A.9.1 Class P2Pedia02

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.Series;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia02
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int[] dist = new int[N];
      ListSeries len = new ListSeries();
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         len.add (shortestPathLength (conn, dist, orig, match));
         }
      System.out.printf ("Query path length:%n");
      Series.Stats stats = len.stats();
      System.out.printf ("Mean   = %.2f%n", stats.mean);
      System.out.printf ("Stddev = %.2f%n", stats.stddev);
      Series.RobustStats rstats = len.robustStats();
      System.out.printf ("Median = %.0f%n", rstats.median);
      System.out.printf ("90%% ci = %.0f .. %.0f%n",
         rstats.quantile (0.05), rstats.quantile (0.95));
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia02 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.9.2 Class P2Pedia03

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia03
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int[] dist = new int[N];
      ListSeries len = new ListSeries();
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         len.add (shortestPathLength (conn, dist, orig, match));
         }
      System.out.printf ("Query path length:%n");
      Series.Stats stats = len.stats();
      System.out.printf ("Mean   = %.2f%n", stats.mean);
      System.out.printf ("Stddev = %.2f%n", stats.stddev);
      Series.RobustStats rstats = len.robustStats();
      System.out.printf ("Median = %.0f%n", rstats.median);
      int a = (int) rstats.quantile (0.05);
      int b = (int) rstats.quantile (0.95);
      System.out.printf ("90%% ci = %d .. %d%n", a, b);
      ListXYSeries hist = new ListXYSeries();
      for (int i = 0; i <= b + a; ++ i)
         {
         hist.add (i, rstats.histogram (i, i + 1));
         }
      new Plot()
         .plotTitle ("P2Pedia03, N="+N+", K="+K+", T="+T)
         .xAxisTitle ("Query path length")
         .yAxisTitle ("Occurrences")
         .xySeries (hist)
         .getFrame()
         .setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
```

```
      {
      System.err.println
         ("Usage: java P2Pedia03 <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.9.3 Class P2Pedia03a

```
import edu.rit.numeric.Function;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.SampledXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.awt.Color;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia03a
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 4) usage();
      int N = Integer.parseInt (args[0]);
      int K = Integer.parseInt (args[1]);
      final int T = Integer.parseInt (args[2]);
      long seed = Long.parseLong (args[3]);
      Random prng = new Random (seed);
      int[][] conn = new int[N][K+1];
      for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
      int[] dist = new int[N];
      ListSeries len = new ListSeries();
      for (int trial = 0; trial < T; ++ trial)
         {
         for (int i = 0; i < N; ++ i)
            {
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j) conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         len.add (shortestPathLength (conn, dist, orig, match));
         }
      System.out.printf ("Query path length:%n");
      final Series.Stats stats = len.stats();
      System.out.printf ("Mean   = %.2f%n", stats.mean);
      System.out.printf ("Stddev = %.2f%n", stats.stddev);
      Series.RobustStats rstats = len.robustStats();
      System.out.printf ("Median = %.0f%n", rstats.median);
      int a = (int) rstats.quantile (0.05);
      int b = (int) rstats.quantile (0.95);
      System.out.printf ("90%% ci = %d .. %d%n", a, b);
      ListXYSeries hist = new ListXYSeries();
      for (int i = 0; i <= b + a; ++ i)
         {
         hist.add (i, rstats.histogram (i, i + 1));
         }
      new Plot()
         .plotTitle ("P2Pedia03a, N="+N+", K="+K+", T="+T)
         .xAxisTitle ("Query path length")
         .yAxisTitle ("Occurrences")
         .xySeries (hist)
         .seriesDots (null)
         .seriesStroke (Strokes.solid (1))
         .seriesColor (Color.RED)
         .xySeries
            (new SampledXYSeries
               (new Function()
                  {
                  private double mean = stats.mean;
                  private double var = stats.var;
                  private double stddev = stats.stddev;
                  private double A = T/Math.sqrt(2*Math.PI)/stddev;
                  public double f (double x)
                     {
                     double d = x - mean;
                     return A*Math.exp(-0.5*d*d/var);
                     }
                  },
                0.0, 0.1, 10*(b + a) + 1))
         .getFrame()
         .setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia03a <N> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.10 Knob Turning

A.10.1 Class P2Pedia04

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia04
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 6) usage();
      int N_L = Integer.parseInt (args[0]);
      int N_U = Integer.parseInt (args[1]);
      int N_D = Integer.parseInt (args[2]);
      int K = Integer.parseInt (args[3]);
      int T = Integer.parseInt (args[4]);
      long seed = Long.parseLong (args[5]);
      Random prng = new Random (seed);
      ListXYSeries mqpl = new ListXYSeries();
      System.out.printf ("\tQuery path length%n");
      System.out.printf ("N\tMean\tStddev%n");
      for (int N = N_L; N <= N_U; N += N_D)
         {
         int[][] conn = new int[N][K+1];
         for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
         int[] dist = new int[N];
         ListSeries len = new ListSeries();
         for (int trial = 0; trial < T; ++ trial)
            {
            for (int i = 0; i < N; ++ i)
               {
               RandomSubset rs = new RandomSubset (prng, N)
                  .remove (i) .remove ((i + 1)%N);
               for (int j = 1; j <= K; ++ j)
                  conn[i][j] = rs.next();
               }
            int orig = prng.nextInt (N);
            int match = prng.nextInt (N);
            len.add (shortestPathLength
               (conn, dist, orig, match));
            }
         Series.Stats stats = len.stats();
         System.out.printf ("%d\t%.2f\t%.2f%n",
            N, stats.mean, stats.stddev);
         mqpl.add (N, stats.mean);
         }
      new Plot()
         .plotTitle ("P2Pedia04, K="+K+", T="+T)
         .xAxisTitle ("Number of nodes, N")
         .yAxisTitle ("Mean query path length")
         .xySeries (mqpl)
         .getFrame()
         .setVisible (true);
      }

   private static int shortestPathLength
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }
```

```
private static void usage()
   {
   System.err.println
      ("Usage: java P2Pedia04 <N_L> <N_U> <N_D> <K> <T> "+
       "<seed>");
   System.exit (1);
   }
}
```

A.11 Regression

A.11.1 Class P2Pedia05

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.ListXYZSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.XYZSeries;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia05
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 6) usage();
      double log_N_L = Double.parseDouble (args[0]);
      double log_N_U = Double.parseDouble (args[1]);
      double log_N_D = Double.parseDouble (args[2]);
      int K = Integer.parseInt (args[3]);
      int T = Integer.parseInt (args[4]);
      long seed = Long.parseLong (args[5]);
      Random prng = new Random (seed);
      ListXYSeries mqpl = new ListXYSeries();
      ListXYZSeries model = new ListXYZSeries();
      System.out.printf ("\tQuery path length, Q%n");
      System.out.printf ("N\tMean\tStddev%n");
      double log_N;
      for (int r = 0; (log_N = log_N_L + r*log_N_D) <= log_N_U;
           ++ r)
         {
         int N = (int) (Math.pow(10.0,log_N) + 0.5);
         int[][] conn = new int[N][K+1];
         for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
         int[] dist = new int[N];
         ListSeries len = new ListSeries();
         for (int trial = 0; trial < T; ++ trial)
            {
            for (int i = 0; i < N; ++ i)
               {
```

```
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 1; j <= K; ++ j)
               conn[i][j] = rs.next();
            }
         int orig = prng.nextInt (N);
         int match = prng.nextInt (N);
         len.add (shortestPathLength
            (conn, dist, orig, match));
         }
      Series.Stats stats = len.stats();
      System.out.printf ("%d\t%.2f\t%.2f%n",
         N, stats.mean, stats.stddev);
      mqpl.add (N, stats.mean);
      model.add (Math.log10(N), stats.mean, stats.stddev);
      }
   XYZSeries.Regression regr = model.linearRegression();
   System.out.printf ("Q = a + b log N%n");
   System.out.printf ("a = %.2f%n", regr.a);
   System.out.printf ("b = %.2f%n", regr.b);
   System.out.printf ("stddev(a) = %.2f%n",
      Math.sqrt(regr.var_a));
   System.out.printf ("stddev(b) = %.2f%n",
      Math.sqrt(regr.var_b));
   System.out.printf ("chi^2 = %.6f%n", regr.chi2);
   System.out.printf ("p-value = %.6f%n", regr.significance);
   new Plot()
      .plotTitle ("P2Pedia05, K="+K+", T="+T)
      .xAxisTitle ("Number of nodes, N")
      .yAxisTitle ("Mean query path length, Q")
      .xAxisKind (Plot.LOGARITHMIC)
      .xAxisMinorDivisions (10)
      .minorGridLines (true)
      .seriesStroke (null)
      .xySeries (mqpl)
      .seriesDots (null)
      .seriesStroke (Strokes.solid (1))
      .xySeries
         (Math.pow(10.0,log_N_L), regr.a + regr.b*log_N_L,
          Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U)
      .getFrame()
      .setVisible (true);
   }

private static int shortestPathLength
```

```
      (int[][] conn,
       int[] dist,
       int orig,
       int match)
      {
      if (orig == match) return 0;
      Arrays.fill (dist, -1);
      dist[orig] = 0;
      LinkedList<Integer> queue = new LinkedList<Integer>();
      queue.addLast (orig);
      for (;;)
         {
         int i = queue.removeFirst();
         for (int j : conn[i])
            {
            if (j == match)
               {
               return dist[i] + 1;
               }
            else if (dist[j] == -1)
               {
               dist[j] = dist[i] + 1;
               queue.addLast (j);
               }
            }
         }
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia05 <log N_L> <log N_U> "+
          "<log N_D> <K> <T> <seed>");
      System.exit (1);
      }
   }
```

A.12 Multiple Knob Turning

A.12.1 Class P2Pedia06

```
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.ListXYZSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.XYZSeries;
import edu.rit.numeric.plot.Dots;
import edu.rit.numeric.plot.Plot;
import edu.rit.numeric.plot.Strokes;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.Arrays;
import java.util.LinkedList;
public class P2Pedia06
   {
   public static void main
      (String[] args)
      throws Exception
      {
      if (args.length != 8) usage();
      double log_N_L = Double.parseDouble (args[0]);
      double log_N_U = Double.parseDouble (args[1]);
      double log_N_D = Double.parseDouble (args[2]);
      int K_L = Integer.parseInt (args[3]);
      int K_U = Integer.parseInt (args[4]);
      int K_D = Integer.parseInt (args[5]);
      int T = Integer.parseInt (args[6]);
      long seed = Long.parseLong (args[7]);
      Random prng = new Random (seed);
      Plot plot = new Plot()
         .plotTitle ("P2Pedia06, T="+T)
         .xAxisTitle ("Number of nodes, N")
         .yAxisTitle ("Mean query path length, Q")
         .xAxisKind (Plot.LOGARITHMIC)
         .xAxisMinorDivisions (10)
         .minorGridLines (true)
         .rightMargin (36)
         .labelPosition (Plot.RIGHT)
         .labelOffset (6);
      for (int K = K_L; K <= K_U; ++ K)
         {
         ListXYSeries mqpl = new ListXYSeries();
         ListXYZSeries model = new ListXYZSeries();
```

```
System.out.printf ("=======================%n");
System.out.printf ("K = %d%n", K);
System.out.printf ("\tQuery path length, Q%n");
System.out.printf ("N\tMean\tStddev%n");
double log_N;
for (int r = 0; (log_N = log_N_L + r*log_N_D) <= log_N_U;
      ++ r)
   {
   int N = (int) (Math.pow(10.0,log_N) + 0.5);
   int[][] conn = new int[N][K+1];
   for (int i = 0; i < N; ++ i) conn[i][0] = (i + 1)%N;
   int[] dist = new int[N];
   ListSeries len = new ListSeries();
   for (int trial = 0; trial < T; ++ trial)
      {
      for (int i = 0; i < N; ++ i)
         {
         RandomSubset rs = new RandomSubset (prng, N)
            .remove (i) .remove ((i + 1)%N);
         for (int j = 1; j <= K; ++ j)
            conn[i][j] = rs.next();
         }
      int orig = prng.nextInt (N);
      int match = prng.nextInt (N);
      len.add (shortestPathLength
         (conn, dist, orig, match));
      }
   Series.Stats stats = len.stats();
   System.out.printf ("%d\t%.2f\t%.2f%n",
      N, stats.mean, stats.stddev);
   mqpl.add (N, stats.mean);
   model.add (Math.log10(N), stats.mean, stats.stddev);
   }
XYZSeries.Regression regr = model.linearRegression();
System.out.printf ("Q = a + b log N%n");
System.out.printf ("a = %.2f%n", regr.a);
System.out.printf ("b = %.2f%n", regr.b);
System.out.printf ("stddev(a) = %.2f%n",
   Math.sqrt(regr.var_a));
System.out.printf ("stddev(b) = %.2f%n",
   Math.sqrt(regr.var_b));
System.out.printf ("chi^2 = %.6f%n", regr.chi2);
System.out.printf ("p-value = %.6f%n", regr.significance);
plot.seriesDots (Dots.circle())
   .seriesStroke (null)
```

```
         .xySeries (mqpl)
         .seriesDots (null)
         .seriesStroke (Strokes.solid (1))
         .xySeries
            (Math.pow(10.0,log_N_L), regr.a + regr.b*log_N_L,
             Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U)
         .label ("K="+K,
             Math.pow(10.0,log_N_U), regr.a + regr.b*log_N_U);
      }
   plot.getFrame().setVisible (true);
   }

private static int shortestPathLength
   (int[][] conn,
    int[] dist,
    int orig,
    int match)
   {
   if (orig == match) return 0;
   Arrays.fill (dist, -1);
   dist[orig] = 0;
   LinkedList<Integer> queue = new LinkedList<Integer>();
   queue.addLast (orig);
   for (;;)
      {
      int i = queue.removeFirst();
      for (int j : conn[i])
         {
         if (j == match)
            {
            return dist[i] + 1;
            }
         else if (dist[j] == -1)
            {
            dist[j] = dist[i] + 1;
            queue.addLast (j);
            }
         }
      }
   }

private static void usage()
   {
   System.err.println
      ("Usage: java P2Pedia06 <log N_L> <log N_U> "+
       "<log N_D> <K_L> <K_U> <K_D> <T> <seed>");
   System.exit (1);
   }
}
```

A.13 Discrete Event Simulation

No classes

A.14 Exponential Distribution

A.14.1 Class edu.rit.numeric.DoublePrng

```
package edu.rit.numeric;
import edu.rit.util.Random;
public abstract class DoublePrng
   {
   /**
    * The underlying uniform PRNG.
    */
   protected final Random myUniformPrng;

   /**
    * Construct a new double PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if theUniformPrng is null.
    */
   public DoublePrng
      (Random theUniformPrng)
      {
      if (theUniformPrng == null)
         {
         throw new NullPointerException
            ("DoublePrng(): theUniformPrng is null");
         }
      myUniformPrng = theUniformPrng;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public abstract double next();
   }
```

A.14.2 Class edu.rit.numeric.ExponentialPrng

```java
package edu.rit.numeric;
import edu.rit.util.Random;
public class ExponentialPrng
   extends DoublePrng
   {
   private double lambda;

   /**
    * Construct a new exponential PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    * @param  lambda          Mean rate lambda > 0.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if theUniformPrng is null.
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if lambda <= 0.
    */
   public ExponentialPrng
      (Random theUniformPrng,
       double lambda)
      {
      super (theUniformPrng);
      if (lambda <= 0)
         {
         throw new IllegalArgumentException
            ("ExponentialPrng(): lambda = "+lambda+" illegal");
         }
      this.lambda = lambda;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public double next()
      {
      return -Math.log(myUniformPrng.nextDouble())/lambda;
      }
   }
```

A.15 Priority Queues

A.15.1 Class edu.rit.sim.Event

```
package edu.rit.sim;
public abstract class Event
   {
   // Simulation in which this event occurs.
   Simulation sim;

   // Simulation time of this event.
   double time;

   /**
    * Construct a new event.
    */
   public Event()
      {
      }

   /**
    * Returns the simulation in which this event occurs.
    *
    * @return  Simulation.
    */
   public final Simulation simulation()
      {
      return sim;
      }

   /**
    * Returns this event's simulation time, the time when this
    * event is scheduled to take place.
    *
    * @return  Simulation time.
    */
   public final double time()
      {
      return time;
      }

   /**
    * Schedule the given event to be performed at the given time
    * in this event's simulation.
    *
    * @param  t      Simulation time for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if t is less than the
    *     current simulation time.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public final void doAt
      (double t,
       Event event)
      {
      sim.doAt (t, event);
      }

   /**
    * Schedule the given event to be performed at a time dt in the
    * future (at current simulation time + dt) in this event's
    * simulation.
    *
    * @param  dt     Simulation time delta for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if dt is less than zero.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public final void doAfter
      (double dt,
       Event event)
      {
      sim.doAfter (dt, event);
      }

   /**
    * Perform this event. Called by the simulation when the
    * simulation time equals the time when this event is
    * scheduled to take place.
    */
   public abstract void perform();
   }
```

A.15.2 Class edu.rit.sim.Simulation

```
package edu.rit.sim;
public class Simulation
   {
   // Minimum-priority queue of events. Uses a heap data
   // structure. The entry at index 0 is a sentinel with time =
   // 0.0.
   private Event[] heap = new Event [1024];

   // Number of entries in the heap (including the sentinel).
   private int N = 1;

   // Simulation time.
   private double T = 0.0;

   /**
    * Construct a new simulation.
    */
   public Simulation()
      {
      heap[0] = new Event() { public void perform() { } };
      heap[0].sim = this;
      heap[0].time = 0.0;
      }

   /**
    * Returns the current simulation time.
    *
    * @return  Simulation time.
    */
   public double time()
      {
      return T;
      }

   /**
    * Schedule the given event to be performed at the given time
    * in this simulation.
    *
    * @param  t      Simulation time for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if t is less than the
    *     current simulation time.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public void doAt
      (double t,
       Event event)
      {
      // Verify preconditions.
      if (t < T)
         {
         throw new IllegalArgumentException
            ("Simulation.doAt(): t = "+t+
             " less than simulation time ="+T+", illegal");
         }
      if (event == null)
         {
         throw new NullPointerException
            ("Simulation.doAt(): event = null");
         }

      // Set event fields.
      event.sim = this;
      event.time = t;

      // Grow heap if necessary.
      if (N == heap.length)
         {
         Event[] newheap = new Event [N + 1024];
         System.arraycopy (heap, 0, newheap, 0, N);
         heap = newheap;
         }

      // Insert event into heap in min-priority order.
      heap[N] = event;
      siftUp (N);
      ++ N;
      }

   /**
    * Schedule the given event to be performed at a time dt in the
    * future (at current simulation time + dt) in this simulation.
    *
    * @param  dt     Simulation time delta for event.
    * @param  event  Event to be performed.
    *
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if dt is less than zero.
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if event is null.
    */
   public void doAfter
      (double dt,
       Event event)
      {
      doAt (T + dt, event);
      }

   /**
    * Run the simulation. At the start of the simulation, the
    * simulation time is 0. The run() method returns when there
    * are no more events.
    */
   public void run()
      {
      while (N > 1)
         {
         // Extract minimum event from heap.
         Event event = heap[1];
         -- N;
         heap[1] = heap[N];
         heap[N] = null;
         if (N > 1) siftDown (1);

         // Advance simulation time and perform event.
         T = event.time;
         event.perform();
         }
      }

   /**
    * Sift up the heap entry at the given index.
    *
    * @param  c  Index.
    */
   private void siftUp
      (int c)
      {
      double c_time = heap[c].time;
      int p = c >> 1;
      double p_time = heap[p].time;
      while (c_time < p_time)
         {
         Event temp = heap[c];
         heap[c] = heap[p];
         heap[p] = temp;
         c = p;
         p = c >> 1;
         p_time = heap[p].time;
         }
      }

   /**
    * Sift down the heap entry at the given index.
    *
    * @param  p  Index.
    */
   private void siftDown
      (int p)
      {
      double p_time = heap[p].time;
      int lc = (p << 1);
      double lc_time =
         lc < N ? heap[lc].time : Double.POSITIVE_INFINITY;
      int rc = (p << 1) + 1;
      double rc_time =
         rc < N ? heap[rc].time : Double.POSITIVE_INFINITY;
      int c;
      double c_time;
      if (lc_time < rc_time)
         {
         c = lc;
         c_time = lc_time;
         }
      else
         {
         c = rc;
         c_time = rc_time;
         }
      while (c_time < p_time)
         {
         Event temp = heap[c];
         heap[c] = heap[p];
         heap[p] = temp;
         p = c;
         lc = (p << 1);
         lc_time =
            lc < N ? heap[lc].time : Double.POSITIVE_INFINITY;
         rc = (p << 1) + 1;
         rc_time =
            rc < N ? heap[rc].time : Double.POSITIVE_INFINITY;
         if (lc_time < rc_time)
            {
            c = lc;
            c_time = lc_time;
            }
         else
            {
            c = rc;
            c_time = rc_time;
            }
         }
      }
   }
```

A.16 A Web Server

A.16.1 Class WebServer01

```
import edu.rit.numeric.ExponentialPrng;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import java.util.LinkedList;
public class WebServer01
   {
   private static double lambda;
   private static double mu;
   private static int R;
   private static long seed;
   private static Random prng;
   private static ExponentialPrng requestPrng;
   private static ExponentialPrng serverPrng;
   private static LinkedList<Request> requestQueue;
   private static int requestCount;
   private static Simulation sim;

   public static void main
      (String[] args)
      {
      if (args.length != 4) usage();
      lambda = Double.parseDouble (args[0]);
      mu = Double.parseDouble (args[1]);
      R = Integer.parseInt (args[2]);
      seed = Long.parseLong (args[3]);
      prng = new Random (seed);
      requestPrng = new ExponentialPrng (prng, lambda);
      serverPrng = new ExponentialPrng (prng, mu);
      requestQueue = new LinkedList<Request>();
      requestCount = 0;
      sim = new Simulation();
      generateRequest();
      sim.run();
      }

   private static class Request
      {
      private int requestNumber;
      public Request
         (int requestNumber)
         {
```

```
         this.requestNumber = requestNumber;
         }
      public String toString()
         {
         return "Request " + requestNumber;
         }
      }

   private static void generateRequest()
      {
      addToQueue (new Request (++ requestCount));
      if (requestCount < R)
         {
         sim.doAfter (requestPrng.next(), new Event()
            {
            public void perform() { generateRequest(); }
            });
         }
      }

   private static void addToQueue
      (Request request)
      {
      System.out.printf ("%.3f %s added to queue%n",
         sim.time(), request);
      requestQueue.add (request);
      if (requestQueue.size() == 1) startServing();
      }

   private static void startServing()
      {
      System.out.printf ("%.3f Started serving %s%n",
         sim.time(), requestQueue.getFirst());
      sim.doAfter (serverPrng.next(), new Event()
         {
         public void perform() { removeFromQueue(); }
         });
      }

   private static void removeFromQueue()
      {
      System.out.printf ("%.3f %s removed from queue%n",
         sim.time(), requestQueue.removeFirst());
      if (requestQueue.size() > 0) startServing();
      }
   private static void usage()
      {
      System.err.println
         ("Usage: java WebServer01 <lambda> <mu> <R> <seed>");
      System.exit (1);
      }
   }
```

A.17 A DOS Attack

A.17.1 Class WebServer02

```
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.plot.Plot;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import java.util.LinkedList;
public class WebServer02
   {
   private static double lambda_L;
   private static double lambda_U;
   private static double lambda_D;
   private static double mu;
   private static int Q;
   private static int R;
   private static long seed;
   private static ListXYSeries meanWaitTimeVsLambda;
   private static ListXYSeries dropFractionVsLambda;
   private static Random prng;
   private static ExponentialPrng requestPrng;
   private static ExponentialPrng serverPrng;
   private static LinkedList<Request> requestQueue;
   private static int requestCount;
   private static int dropCount;
   private static ListSeries waitTime;
   private static Simulation sim;

   public static void main
      (String[] args)
      {
      if (args.length != 7) usage();
      lambda_L = Double.parseDouble (args[0]);
      lambda_U = Double.parseDouble (args[1]);
      lambda_D = Double.parseDouble (args[2]);
      mu = Double.parseDouble (args[3]);
      Q = Integer.parseInt (args[4]);
      R = Integer.parseInt (args[5]);
      seed = Long.parseLong (args[6]);
      meanWaitTimeVsLambda = new ListXYSeries();
      dropFractionVsLambda = new ListXYSeries();
      prng = new Random (seed);
      System.out.printf
         ("Mu = %.3f, Q = %d, R = %d, seed = %d%n",
          mu, Q, R, seed);
      System.out.printf ("        --Wait time---  Drop%n");
      System.out.printf ("Lambda  Mean    Stddev  Fraction%n");
      double lambda;
      for (int r = 0;
            (lambda = lambda_L + r*lambda_D) <= lambda_U;
            ++ r)
         {
         requestPrng = new ExponentialPrng (prng, lambda);
         serverPrng = new ExponentialPrng (prng, mu);
         requestQueue = new LinkedList<Request>();
         requestCount = 0;
         dropCount = 0;
         waitTime = new ListSeries();
         sim = new Simulation();
         generateRequest();
         sim.run();
         Series.stats wt = waitTime.stats();
         double df = ((double) dropCount)/((double) R);
         meanWaitTimeVsLambda.add (lambda, wt.mean);
         dropFractionVsLambda.add (lambda, df);
         System.out.printf ("%-8.3f%-8.3f%-8.3f%-8.3f%n",
            lambda, wt.mean, wt.stddev, df);
         }
      new Plot()
         .plotTitle
            ("WebServer02, \u03BC="+mu+", Q="+Q+", R="+R)
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Mean wait time")
         .xySeries (meanWaitTimeVsLambda)
         .getFrame()
         .setVisible (true);
      new Plot()
         .plotTitle
            ("WebServer02, \u03BC="+mu+", Q="+Q+", R="+R)
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Drop fraction")
         .xySeries (dropFractionVsLambda)
         .getFrame()
         .setVisible (true);
      }
```

```
private static class Request
   {
   private double startTime;
   public Request()
      {
      startTime = sim.time();
      }
   public double waitTime()
      {
      return sim.time() - startTime;
      }
   }

private static void generateRequest()
   {
   addToQueue (new Request (sim.time()));
   if (++ requestCount < R)
      {
      sim.doAfter (requestPrng.next(), new Event()
         {
         public void perform() { generateRequest(); }
         });
      }
   }

private static void addToQueue
   (Request request)
   {
   if (requestQueue.size() < Q)
      {
      requestQueue.add (request);
      if (requestQueue.size() == 1) startServing();
      }
   else
      {
      ++ dropCount;
      }
   }

private static void startServing()
   {
   sim.doAfter (serverPrng.next(), new Event()
      {
      public void perform() { removeFromQueue(); }
      });
   }

private static void removeFromQueue()
   {
   waitTime.add (requestQueue.removeFirst().waitTime());
   if (requestQueue.size() > 0) startServing();
   }

private static void usage()
   {
   System.err.println
      ("Usage: java WebServer02 <lambda_L> <lambda_U> "+
       "<lambda_D> <mu> <Q> <R> <seed>");
   System.exit (1);
   }
}
```

A.18 Uniform Distribution

A.18.1 Class edu.rit.numeric.UniformPrng

```
package edu.rit.numeric;
import edu.rit.util.Random;
public class UniformPrng
   extends DoublePrng
   {
   private double a;
   private double b_minus_a;

   /**
    * Construct a new uniform PRNG.
    *
    * @param  theUniformPrng  The underlying uniform PRNG.
    * @param  a               Interval lower bound.
    * @param  b               Interval upper bound.
    *
    * @exception  NullPointerException
    *     (unchecked exception) Thrown if <TT>theUniformPrng</TT>
    *     is null.
    * @exception  IllegalArgumentException
    *     (unchecked exception) Thrown if <I>a</I> &ge; <I>b</I>.
    */
   public UniformPrng
      (Random theUniformPrng,
       double a,
       double b)
      {
      super (theUniformPrng);
      if (a >= b)
         {
         throw new IllegalArgumentException
            ("UniformPrng(): a ("+a+") >= b ("+b+") illegal");
         }
      this.a = a;
      this.b_minus_a = b - a;
      }

   /**
    * Returns the next random number.
    *
    * @return  Random number.
    */
   public double next()
```

```
      {
      return myUniformPrng.nextDouble()*b_minus_a + a;
      }
   }
```

A.19 A Distributed Encyclopedia, Part 2

A.19.1 Class P2Pedia07

```
import edu.rit.numeric.DoublePrng;
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.UniformPrng;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.HashSet;
import java.util.LinkedList;
public class P2Pedia07
   {
   // Knobs
   private static int N; // Number of nodes
   private static int K; // Number of extra connections per node
   private static double lambda; // Mean query arrival rate
   private static double tx_L; // Query transmit time lower bound
   private static double tx_U; // Query transmit time upper bound
   private static int Q; // Maximum queries in query queue
   private static int R; // Number of queries to simulate
   private static long seed; // PRNG seed

   // Global variables
   private static Random prng;
   private static DoublePrng queryPrng;
   private static DoublePrng txPrng;
   private static int queryCount;
   private static ListSeries elapsedTimes;
   private static int successCount;
   private static Simulation sim;
   private static Network network;

   private static class Query
      {
      private int id;
      private Node matchingNode;
      private double startTime;

      public Query
         (int id,
          Node matchingNode)
         {
         this.id = id;
         this.matchingNode = matchingNode;
         this.startTime = sim.time();
         }

      public boolean matches
         (Node node)
         {
         return node == matchingNode;
         }

      public double elapsedTime()
         {
         return sim.time() - startTime;
         }

      public boolean equals
         (Object obj)
         {
         return
            obj instanceof Query &&
            ((Query) obj).id == this.id;
         }

      public int hashCode()
         {
         return id;
         }

      public String toString()
         {
         return "Query " + id;
         }
      }

   private static class Node
      {
      private int id;
      private LinkedList<Node> connectedNodes =
         new LinkedList<Node>();
      private LinkedList<Query> queryQueue =
         new LinkedList<Query>();
      private HashSet<Query> priorQueries =
         new HashSet<Query>();
      public Node
         (int id)
         {
         this.id = id;
         }

      public void connectTo
         (Node node)
         {
         connectedNodes.add (node);
         }

      public void addToQueue
         (Query query)
         {
         if (queryQueue.size() < Q)
            {
            System.out.printf ("%.3f %s received by %s%n",
               sim.time(), query, this);
            queryQueue.add (query);
            if (queryQueue.size() == 1) startServing();
            }
         else
            {
            System.out.printf ("%.3f %s dropped by %s%n",
               sim.time(), query, this);
            }
         }

      private void startServing()
         {
         final Query query = queryQueue.getFirst();
         if (priorQueries.contains (query))
            {
            System.out.printf ("%.3f %s previously seen by %s%n",
               sim.time(), query, this);
            removeFromQueue();
            }
         else if (query.matches (this))
            {
            System.out.printf ("%.3f %s matches at %s%n",
               sim.time(), query, this);
            elapsedTimes.add (query.elapsedTime());
            ++ successCount;
            removeFromQueue();
            }
         else
            {
            System.out.printf ("%.3f %s sent from %s to",
               sim.time(), query, this);
            double txTime = 0.0;
            for (Node nextNode : connectedNodes)
               {
               System.out.printf (" %s", nextNode);
               final Node nn = nextNode;
               txTime += txPrng.next();
               sim.doAfter (txTime, new Event()
                  {
                  public void perform()
                     {
                     nn.addToQueue (query);
                     }
                  });
               }
            System.out.println();
            sim.doAfter (txTime, new Event()
               {
               public void perform()
                  {
                  removeFromQueue();
                  }
               });
            }
         }

      private void removeFromQueue()
         {
         Query query = queryQueue.removeFirst();
         System.out.printf ("%.3f %s removed from %s%n",
            sim.time(), query, this);
         priorQueries.add (query);
         if (queryQueue.size() > 0) startServing();
         }

      public String toString()
         {
         return "Node " + id;
         }
      }
   private static class Network
      {
      private Node[] node;

      public Network()
         {
         node = new Node[N];
         for (int i = 0; i < N; ++ i) node[i] = new Node (i);
         for (int i = 0; i < N; ++ i)
            {
            node[i].connectTo (node[(i + 1)%N]);
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 0; j < K; ++ j)
               {
               node[i].connectTo (node[rs.next()]);
               }
            }
         }

      public void generateQuery()
         {
         Node origNode = randomNode();
         Node matchNode = randomNode();
         Query query = new Query (++ queryCount, matchNode);
         System.out.printf
            ("%.3f %s generated, originating %s, matching %s%n",
             sim.time(), query, origNode, matchNode);
         origNode.addToQueue (query);
         if (queryCount < R)
            {
            sim.doAfter (queryPrng.next(), new Event()
               {
               public void perform()
                  {
                  generateQuery();
                  }
               });
            }
         }

      private Node randomNode()
         {
         return node[prng.nextInt (node.length)];
         }
      }

   public static void main
      (String[] args)
      {
      // Knobs
      if (args.length != 8) usage();
      N = Integer.parseInt (args[0]);
      K = Integer.parseInt (args[1]);
      lambda = Double.parseDouble (args[2]);
      tx_L = Double.parseDouble (args[3]);
      tx_U = Double.parseDouble (args[4]);
      Q = Integer.parseInt (args[5]);
      R = Integer.parseInt (args[6]);
      seed = Long.parseLong (args[7]);

      // Global variables
      prng = new Random (seed);
      queryPrng = new ExponentialPrng (prng, lambda);
      txPrng = new UniformPrng (prng, tx_L, tx_U);
      queryCount = 0;
      elapsedTimes = new ListSeries();
      successCount = 0;
      sim = new Simulation();
      network = new Network();

      // Run simulation
      network.generateQuery();
      sim.run();
      Series.Stats etStats = elapsedTimes.stats();
      double ff = 1.0 - (double)successCount/(double)R;
      System.out.printf ("Elapsed time mean = %.3f%n",
         etStats.mean);
      System.out.printf ("Elapsed time stddev = %.3f%n",
         etStats.stddev);
      System.out.printf ("Failure fraction = %.3f%n", ff);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia07 <N> <K> <lambda> <tx_L> "+
          "<tx_U> <Q> <R> <seed>");
      System.exit (1);
      }
```

```
    }
```

A.20 Encyclopedia: Scalable?

A.20.1 Class P2Pedia08

```
import edu.rit.numeric.DoublePrng;
import edu.rit.numeric.ExponentialPrng;
import edu.rit.numeric.ListSeries;
import edu.rit.numeric.ListXYSeries;
import edu.rit.numeric.Series;
import edu.rit.numeric.UniformPrng;
import edu.rit.numeric.plot.Plot;
import edu.rit.sim.Event;
import edu.rit.sim.Simulation;
import edu.rit.util.Random;
import edu.rit.util.RandomSubset;
import java.util.HashSet;
import java.util.LinkedList;
public class P2Pedia08
   {
   // Knobs
   private static int N; // Number of nodes
   private static int K; // Number of extra connections per node
   private static double lambda_L; // First query arrival rate
   private static double lambda_U; // Last query arrival rate
   private static double lambda_D; // Query arrival rate increment
   private static double tx_L; // Query transmit time lower bound
   private static double tx_U; // Query transmit time upper bound
   private static int Q; // Maximum queries in query queue
   private static int R1; // Number of topologies per arrival rate
   private static int R2; // Number of queries per topology
   private static long seed; // PRNG seed

   // Global variables
   private static ListXYSeries meanElapsedTimeVsLambda;
   private static ListXYSeries failureFractionVsLambda;
   private static Random prng;
   private static DoublePrng txPrng;
   private static DoublePrng queryPrng;
   private static ListSeries elapsedTimes;
   private static int successCount;
   private static int queryCount;
   private static Simulation sim;
   private static Network network;

   private static class Query
      {
```

```
      private int id;
      private Node matchingNode;
      private double startTime;

      public Query
         (int id,
          Node matchingNode)
         {
         this.id = id;
         this.matchingNode = matchingNode;
         this.startTime = sim.time();
         }

      public boolean matches
         (Node node)
         {
         return node == matchingNode;
         }

      public double elapsedTime()
         {
         return sim.time() - startTime;
         }

      public boolean equals
         (Object obj)
         {
         return
            obj instanceof Query &&
            ((Query) obj).id == this.id;
         }

      public int hashCode()
         {
         return id;
         }
      }

   private static class Node
      {
      private int id;
      private LinkedList<Node> connectedNodes =
         new LinkedList<Node>();
      private LinkedList<Query> queryQueue =
         new LinkedList<Query>();
      private HashSet<Query> priorQueries =
         new HashSet<Query>();

      public Node
         (int id)
         {
         this.id = id;
         }

      public void connectTo
         (Node node)
         {
         connectedNodes.add (node);
         }

      public void addToQueue
         (Query query)
         {
         if (queryQueue.size() < Q)
            {
            queryQueue.add (query);
            if (queryQueue.size() == 1) startServing();
            }
         }

      private void startServing()
         {
         final Query query = queryQueue.getFirst();
         if (priorQueries.contains (query))
            {
            removeFromQueue();
            }
         else if (query.matches (this))
            {
            elapsedTimes.add (query.elapsedTime());
            ++ successCount;
            removeFromQueue();
            }
         else
            {
            double txTime = 0.0;
            for (Node nextNode : connectedNodes)
               {
               final Node nn = nextNode;
               txTime += txPrng.next();
               sim.doAfter (txTime, new Event()
                  {
                  public void perform()
                     {
                     nn.addToQueue (query);
                     }
                  });
               }
            sim.doAfter (txTime, new Event()
               {
               public void perform()
                  {
                  removeFromQueue();
                  }
               });
            }
         }

      private void removeFromQueue()
         {
         Query query = queryQueue.removeFirst();
         priorQueries.add (query);
         if (queryQueue.size() > 0) startServing();
         }
      }

   private static class Network
      {
      private Node[] node;

      public Network()
         {
         node = new Node[N];
         for (int i = 0; i < N; ++ i) node[i] = new Node (i);
         for (int i = 0; i < N; ++ i)
            {
            node[i].connectTo (node[(i + 1)%N]);
            RandomSubset rs = new RandomSubset (prng, N)
               .remove (i) .remove ((i + 1)%N);
            for (int j = 0; j < K; ++ j)
               {
               node[i].connectTo (node[rs.next()]);
               }
            }
         }
      public void generateQuery()
         {
         Node origNode = randomNode();
         Node matchNode = randomNode();
         Query query = new Query (++ queryCount, matchNode);
         origNode.addToQueue (query);
         if (queryCount < R2)
            {
            sim.doAfter (queryPrng.next(), new Event()
               {
               public void perform()
                  {
                  generateQuery();
                  }
               });
            }
         }

      private Node randomNode()
         {
         return node[prng.nextInt (node.length)];
         }
      }

   public static void main
      (String[] args)
      {
      // Knobs
      if (args.length != 11) usage();
      N = Integer.parseInt (args[0]);
      K = Integer.parseInt (args[1]);
      lambda_L = Double.parseDouble (args[2]);
      lambda_U = Double.parseDouble (args[3]);
      lambda_D = Double.parseDouble (args[4]);
      tx_L = Double.parseDouble (args[5]);
      tx_U = Double.parseDouble (args[6]);
      Q = Integer.parseInt (args[7]);
      R1 = Integer.parseInt (args[8]);
      R2 = Integer.parseInt (args[9]);
      seed = Long.parseLong (args[10]);

      // Global variables
      meanElapsedTimeVsLambda = new ListXYSeries();
      failureFractionVsLambda = new ListXYSeries();
      prng = new Random (seed);
      txPrng = new UniformPrng (prng, tx_L, tx_U);

      // Loop over all mean arrival rates.
      System.out.printf ("        -Elapsed Time-  Failure%n");
      System.out.printf ("Lambda  Mean    Stddev  Fraction%n");
      double lambda;
      for (int i = 0;
            (lambda = lambda_L + i*lambda_D) <= lambda_U; ++ i)
         {
         // Global variables
         queryPrng = new ExponentialPrng (prng, lambda);
         elapsedTimes = new ListSeries();
         successCount = 0;

         // Loop over R1 random topologies.
         for (int j = 0; j < R1; ++ j)
            {
            queryCount = 0;
            sim = new Simulation();
            network = new Network();

            // Run simulation
            network.generateQuery();
            sim.run();
            }

         // Print statistics.
         Series.Stats etStats = elapsedTimes.stats();
         double ff = 1.0 - (double)successCount/(double)(R1*R2);
         System.out.printf ("%-8.3f%-8.3f%-8.3f%-8.3f%n",
            lambda, etStats.mean, etStats.stddev, ff);
         meanElapsedTimeVsLambda.add (lambda, etStats.mean);
         failureFractionVsLambda.add (lambda, ff);
         }

      // Plot results.
      new Plot()
         .plotTitle ("P2Pedia08")
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Mean elapsed time")
         .xySeries (meanElapsedTimeVsLambda)
         .getFrame()
         .setVisible (true);
      new Plot()
         .plotTitle ("P2Pedia08")
         .xAxisTitle ("Mean arrival rate \u03BB")
         .yAxisTitle ("Failure fraction")
         .xySeries (failureFractionVsLambda)
         .getFrame()
         .setVisible (true);
      }

   private static void usage()
      {
      System.err.println
         ("Usage: java P2Pedia08 <N> <K> <lambda_L> <lambda_U> "+
          "<lambda_D> <tx_L> <tx_U> <Q> <R1> <R2> <seed>");
      System.exit (1);
      }
   }
```

Appendix B

Further Reading

On probability and statistics:

- J. Devore. *Probability and Statistics for Engineering and the Sciences, Seventh Edition.* Brooks/Cole, 2009.

On scientific computation—random numbers, statistical tests, mean, variance, standard deviation, median, quantiles, confidence intervals, linear regression, and the like:

- W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes: The Art of Scientific Computing, Third Edition.* Cambridge University Press, 2007.

On data structures and algorithms:

- T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* MIT Press, 2009.

- R. Sedgewick and K. Wayne. *Algorithms, Fourth Edition.* Addison-Wesley, 2011.

On the heap data structure:

- J. Bentley. Programming pearls: thanks, heaps. *Communications of the ACM* 28(3):245–250, March 1985.

- J. Bentley. *Programming Pearls, Second Edition.* Addison-Wesley, 1999.

On the Parallel Java 2 Library:

- A. Kaminsky. *BIG CPU, BIG DATA: Solving the World's Toughest Computational Problems with Parallel Computing.* Creative Commons, 2014. `http://www.cs.rit.edu/~ark/bcbd/`

- A. Kaminsky. Parallel Java 2 Library. `http://www.cs.rit.edu/~ark/pj2.shtml`

- A. Kaminsky. Parallel Java 2 Library Documentation. `http://www.cs.rit.edu/~ark/pj2/doc/index.html`

On the Parallel Java Library (the Parallel Java 2 Library's predecessor):

- A. Kaminsky. *Building Parallel Programs: SMPs, Clusters, and Java.* Course Technology, 2010.
- A. Kaminsky. Parallel Java: A unified API for shared memory and cluster parallel programming in 100% Java. In *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007),* March 2007.