CSCI 210 Assn. 6: Optimizing with a profiler

Lab note: *You may work with up to one other person on this assignment and submit your solution together, or you may work alone. The write-up is due at 5:00pm on Thursday, May 6. It is worth 30 points.*

To maintain a total of 1000 points over the semester, the final exam wil be worth only 100 points.

Objectives

- to learn how to use a profiler to analyze the performance of a program.
- to experience the process of optimizing a program's performance.

Understanding the code

Run "`getcs 210 6b`" to fetch the code for this assignment. The program (whose `main` method is in the `Main.java` file) is one for identifying "twin cities," defined as cities that are close together (within 25 miles) and whose populations are close (within a factor of 2 of each other). The program first reads a file giving the populations of 601 cities, then it reads a file giving the latitude and longitude of 1,181 cities, storing this data into a CityGroup object. (The files have 324 cities in common.) Then it goes through all pairs of cities to determine which are pairs.

When you execute the Main class, the program will report how many pairs it found and how long the process took from start to finish.

```
unix% java Main
123 twins found among 1458 cities
  (2240 ms elapsed)
```

The program is rather slow. In this assignment, you'll work to try to make it work faster.

Understanding the profiler

A **profiler** is a programming tool useful for analyzing the behavior of a program as it executes. There are profilers for many purposes, including analyzing what code it executes most, the purpose of memory being used by the program, and understanding locks held by the process's threads. We'll use a very simple profiler called *simpprof*, which is designed for understanding how much time is spent in various methods of a Java program.[1]

The *simpprof* profiler works via **sampling**: It periodically interrupts the process to check what method the process is currently executing. After the process finishes, the profiler reports (in `simpprof.txt`) how often the process was in each method of the program. If it gathered several samples, then this should be a good approximation of how heavily the process used each method in the program.

You can run *simpprof* on the Solaris systems as follows.[2]

```
unix% simpprof-java Main
123 twins found among 1458 cities
  (4180 ms elapsed)
Dumping CPU usage by sampling running threads ... done.
```

Now, if we look at the `simpprof.txt` file, we will see the following.

[1] This is a modification of a profiling system that Sun includes with its Java system called *hprof*. You can read about *hprof* at the URL `http://java.sun.com/developer/TechTips/2000/tt0124.html#tip2`

[2] Running the profiler will slow the program significantly, since the profiler's sampling hinders the process's progress. In this case, the program took 2240 ms without the profiler, and 4180 ms with. This does not invalidate the profiler's sampling, but it does mean that you shouldn't try to make sense of the time that the the program takes while the profiler is running.

```
cburch@sun2.cs.csbsju.edu 115% cat simpprof.txt
THREAD START (obj=29a690, id = 1, name="main", group="main")
THREAD END (id = 1)
THREAD START (obj=2a5c0, id = 2, name="DestroyJavaVM", group="main")
THREAD END (id = 2)
samples taken: 77

rank  self  accum   method
   1  33.8%  33.8%  CityGroup.findTwinCities
   2  31.2%  64.9%  CityGroup.findCity
   3  20.8%  85.7%  Location.distanceTo
   4   7.8%  93.5%  Main.readLocationFile
   5   2.6%  96.1%  Main.readPopulationFile
   6   2.6%  98.7%  City.distanceTo
   7   1.3% 100.0%  Main.main
```

After several lines talking about threads used by the system, this output reports that *simpprof* was able to take 77 samples. Then it lists the methods in which it found the process for each sample. The first line reports that for 33.8% of the samples, the topmost method in the stack was the `findTwinCities` method of the CityGroup class. The next line reports that for 31.2% of the samples, the topmost method in the stack was the `findCity` method of the CityGroup class. (The *simpprof* profiler passes over Java API methods as it figures out the topmost method in the stack. Thus, when it reports that it is in the `findTwinCities` method, the process may in fact be in the `iterator` method of the API's LinkedList class, which was called by `findTwinCities`.) Unlisted methods (such as CityGroup's `isTwinCity` method) were never the topmost method on the stack during any of the samples.

A6–1. Compile this data on your computer. To get more accurate results, you should profile the same program running three different times and average these three runs together.

Optimizing the code

Now we can work on trying to improve the performance. First, we will determine where we should spend our effort.

A6–2. Suppose we were to rewrite Main's `readLocationFile` method, and our rewritten method worked twice as fast. Based on your measurements with *simpprof*, how much faster will the overall program be? For which three methods will doubling their speed make the biggest effect? For each of these methods, how much faster will the program be if you were to improve that method alone? (Express your answers as percentage decreases: If you compute that the program's time should be 90% of what it was prior to improving `readLocationFile`, then report this as a 10% improvement.)

Incidentally, this analysis is misleading in what methods it indicates you should work on. Often, the best way to improve a method is to avoid calling it altogether. A lot of time is spent in Location's `distanceTo` method, for example. You could try to rewrite it to make it faster, but there isn't much hope for this. But you could speed up the program by making it avoid needless calls to `distanceTo`.

A6–3. Now rewrite a few methods that you have determined are worth rewriting. You should be able to pare off at least 300 milliseconds in the program's running time. Your solution to this problem should be a description of your modifications.

Run "`handincs 210 6b`" to submit your code.

A6–4. Run your program five times and compute the average of the last three runs. Do the same for the original program (available on the command line as *twincities*). Report these numbers as evidence that your modified program is significantly faster than the original program.