Percolation - Compsci 201

The code for this assignment is available through Snarf (percolation) and on the CS webpage http://www.cs.duke.edu/courses/compsci201/fall14/assign/percolation/

ASSIGNMENT OUTLINE:

In this assignment you will write a program to estimate the value of the percolation threshold via Monte Carlo simulation.

What code do you need to write:

1. **PercolationDFS.java**: This class implements the brute force method for percolation. Please refer to Section 2.4 in the Introduction to Programming in Java. You will complete the following methods:
   - constructor: initialize a grid of size \( n \)
   - open: open a particular grid cell
   - isOpen & isFull: give current state of grid cell
   - percolates and dfs: determine whether current grid will percolate by implementing recursive scheme depth-first search as discussed in class and the textbook

2. **PercolationVisualizer.java**: complete main so that it repeatedly calls a percolator (i.e., something that implements IPercolate like PercolationDFS) to declare sites open, draw, and pause until the system percolates

3. **PercolationUF.java**: You will implement a more efficient solution that can use any union-find algorithm that implements IUnionFind (e.g., QuickFind.java)
   - constructor: initialize a grid of size \( n \) and union-find algorithm
   - open: open a particular grid cell and merge components as appropriate
   - isOpen & isFull: give current state of grid cell
   - getIndex: return an index that uniquely identifies (row, col), so that each grid square can be in its own set at the beginning
   - percolates: determine whether top and bottom are connected (i.e., in the same component)

4. **QuickUWPC.java**: class that implements weighted quick union with path compression data structure. See the Sedgewick & Wayne case study or the following reading for more information. You will need to create this file by adapting WeightedQuickUnionUF.java or WeightedQuickUnionHalvingUF.java.
   The class should implement the IUnionFind interface.

5. **PercolationStats.java**: prompts for \( N \) and \( T \), performs \( T \) experiments on \( N \times N \) grid, and prints mean, standard deviation, confidence interval, and timings.
BACKGROUND: PERCOLATION

Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Scientists have defined an abstract process known as percolation to model such situations.

THE MODEL

We model a percolation system using an $N$-by-$N$ grid of sites. Each site is either open or blocked. A full site is an open site that can be connected to an open site in the top row via a chain of neighboring (left, right, up, down) open sites. We say the system percolates if there is a full site in the bottom row. In other words, a system percolates if we fill all open sites connected to the top row and that process fills some open site on the bottom row. (For the insulating/metallic materials example, the open sites correspond to metallic materials, so that a system that percolates has a metallic path from top to bottom, with full sites conducting. For the porous substance example, the open sites correspond to empty space through which water might flow, so that a system that percolates lets water fill open sites, flowing from top to bottom.)

In a famous scientific problem, researchers are interested in the following question: if sites are independently set to be open with probability $p$ (and therefore blocked with probability $1 - p$), what is the probability that the system percolates? When $p$ equals 0, the system does not percolate; when $p$ equals 1, the system percolates. The plots below show
the site vacancy probability $p$ versus the percolation probability for 20-by-20 random grid (left) and 100-by-100 random grid (right).

When $N$ is sufficiently large, there is a threshold value $p^*$ such that when $p < p^*$ a random $N$-by-$N$ grid almost never percolates, and when $p > p^*$, a random $N$-by-$N$ grid almost always percolates. No mathematical solution for determining the percolation threshold $p^*$ has yet been derived. Your task is to write a program to:

1. visualize the percolation process
2. estimate $p^*$
3. compare brute force (depth-first search) to union-find for finding connected open sites

1. Visualizing the Percolation Process

Your completed PercolationVisualizer should prompt the user for $N$ and display the percolation process starting with a $N$-by-$N$ grid of sites (initially all blocked and black). After each site is opened, display full sites in cyan, open sites (that aren’t full) in white, and blocked sites in black using princeton.StdDraw. Here is an example of steps in a visualization on a 20x20 grid as in this movie and the following snapshots.

Going from 50 to 100 to 150 to 204 to 250 open sites in a 20x20 grid.
IPercolate data type

To model a percolation system, you will create different implementations of the IPercolate interface.

```java
public interface IPercolate {
    // Opens site at (row i, col j) if it is not already open
    public abstract void open(int i, int j);
    // Returns true if and only if site (row i, col j) is OPEN
    public abstract boolean isOpen(int i, int j);
    // Returns true if and only if site (row i, col j) is FULL
    public boolean isFull(int i, int j);
    // Returns true iff the simulated system percolates
    public abstract boolean percolates();
}
```

You will complete brute-force (PercolationDFS) and union-find (PercolationUF) versions of the IPercolate data type.

**NB:** By convention, the indices \(i\) and \(j\) are integers between 0 and \(N-1\), where \((0, 0)\) is the upper-left cell. Your code follows this convention in order to pass our tests.

2. Estimating \(p^*\)

To estimate the percolation threshold, perform the following computational experiment:

1. Initialize all sites to be blocked.
2. Repeat the following until the system percolates:
   a. Choose a blocked site \((\text{row } i, \text{column } j)\) uniformly at random among all blocked sites.
   b. Open the site \((\text{row } i, \text{column } j)\).
3. The fraction of sites that are opened until the system percolates provides an estimate of the percolation threshold.

In the 20-by-20 example above, our estimate of the percolation threshold is \(204/400 = 0.51\) because the system percolates when the 204th site is opened.

To obtain an accurate estimate of the percolation threshold, repeat the experiment \(T\) times and average the results. Let \(x_t\) be the fraction of open sites in experiment \(t\). The sample mean, \(\mu\), provides an estimate of the percolation threshold. The sample standard deviation, \(\sigma\), measures the sharpness of the threshold.

\[
\mu = \frac{x_1 + x_2 + \cdots + x_T}{T}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_T - \mu)^2}{T - 1}
\]
Assuming $T$ is sufficiently large (say, at least 30), the following provides a 95% confidence interval for the percolation threshold:

\[
\left[ \mu - \frac{1.96\sigma}{\sqrt{T}}, \mu + \frac{1.96\sigma}{\sqrt{T}} \right],
\]

Write a client program `PercolationStats` that prompts the user for $N$ and $T$, performs $T$ independent experiments on an $N$-by-$N$ grid, and prints out the 95% confidence interval for the percolation threshold. Use `java.util.Random` to generate random numbers and follow steps above to compute the sample mean and standard deviation. Below is an example run with $N=200$ and $T=100$.

<table>
<thead>
<tr>
<th>mean percolation threshold</th>
<th>0.5921</th>
</tr>
</thead>
<tbody>
<tr>
<td>stddev</td>
<td>0.0098</td>
</tr>
<tr>
<td>95% confidence interval</td>
<td>[0.5902, 0.5940]</td>
</tr>
<tr>
<td>total time</td>
<td>2.074s</td>
</tr>
<tr>
<td>mean time per experiment</td>
<td>0.02074</td>
</tr>
<tr>
<td>stddev</td>
<td>0.00376</td>
</tr>
</tbody>
</table>

### 3. Comparing DFS to Union-Find

Implement PercolationUF using the quick find data structure (`QuickFind.java`) and by creating a version using the weighted quick union with path compression data structure. You will adapt the code from class or `WeightedQuickUnionUF.java` to create `QuickUWPC.java` which implements the `IUnionFind` interface. You should will run `PercolationStats` to gather timings for

1. `PercolationDFS`,
2. `PercolationUF` with `QuickFind`, and
3. `PercolationUF` with `QuickUWPC`.

In your `README`, you will answer the following questions.

- How does doubling $N$ affect the running time?
- How does doubling $T$ affect the running time?
- Measure running time (using calls to `System.currentTimeMillis`) of the three versions of your program (DFS, Quick Find, and weighted quick union with path compression).
- Give a formula (using Big-Oh notation) of the running time on your computer (in seconds) as a function of both $N$ and $T$.
- Give a formula (using Big-Oh notation) that describes the amount of memory (in bytes) that your program consumes as a function of $N$. 
Submitting

Submit your code, and README.txt using either the Ambient plugin, or Ambient web submit (https://www.cs.duke.edu/csed/websubmit/app/) to assignment *percolation*

This assignment is worth 40 points.

- **55% algorithmic/correctness:** for the correctness of your implementation of PercolationVisualizer, PercolationDFS, and PercolationUF.
- **20% engineering:** efficiency of your solution, use of weighted path compression union-find, and the *documentation*, structure and style of your Percolation code
- **25% analysis:** your PercolationStats and your README.txt

Last updated 10/22/2014 12:22:00 PM