Project 1 (Percolation)

Clarifications and Hints

Prologue

Project goal: write a program to estimate the percolation threshold of a system

The zip file (`http://www.swamiiyer.net/cs210/percolation.zip`) for the project contains

- project specification (`percolation.pdf`)
- starter files
  - `Percolation.java`
  - `PercolationStats.java`
- test script (`run_tests.py`)
- test data and reference solutions (`data/`)
- visualization clients (`PercolationVisualizer` and `InteractivePercolationVisualizer`)
- report template (`report.txt`)

Problem 1 (*Model a Percolation System*) To model a percolation system, create a data type `Percolation` with the following API:

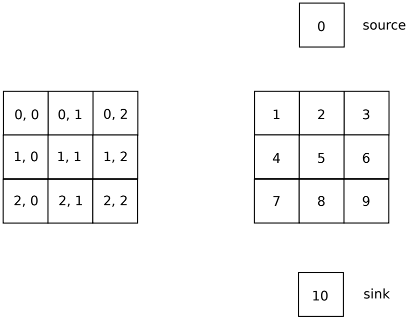
method	description
`Percolation(int N)`	create an N-by-N grid, with all sites blocked
`void open(int i, int j)`	open site (i, j)
`boolean isOpen(int i, int j)`	is site (i, j) open?
`boolean isFull(int i, int j)`	is site (i, j) full?
`int numberOfOpenSites()`	number of open sites
`boolean percolates()`	does the system percolate?

Hints

- Model percolation system as an $N \times N$ array of booleans (`true` $\implies$ open cell and `false` $\implies$ blocked cell)
- Can implement the API by scanning the array directly, but that does not meet all the performance requirements; use Union-find (`UF`) data structure instead
- Create an `UF` object with $N^2 + 2$ sites and use the private `encode()` method to map cells $(0, 0), (0, 1), \ldots, (N - 1, N - 1)$ of the array to sites $1, 2, \ldots, N^2$ of the `UF` object; sites 0 (source) and $N^2 + 1$ (sink) are virtual, ie, not part of the percolation system

Problems

- A 3×3 system and its UF representation



0, 0	0, 1	0, 2
1, 0	1, 1	1, 2
2, 0	2, 1	2, 2

0 source

1	2	3
4	5	6
7	8	9

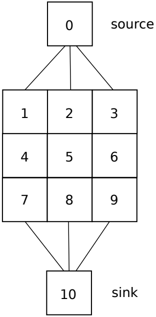
10 sink

- Instance variables
  - Percolation system size, `int N`
  - Percolation system, `boolean[][] open`
  - Number of open sites, `int openSites`
  - Union-find representation of the percolation system, `WeightedQuickUnionUF uf`

Problems

- `public Percolation(int N)`
  - Initialize instance variables
  - Connect the sites corresponding to first and last rows of the percolation system with the source and sink sites respectively
  - The 3×3 system with its top and bottom row sites connected to the source and sink sites respectively

0, 0	0, 1	0, 2
1, 0	1, 1	1, 2
2, 0	2, 1	2, 2

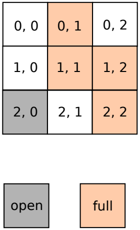


Problems

- `void open(int i, int j)`
  - Open the cell `(i, j)` if it is not already open
  - Increment `openSites` by one
  - Check if any of the neighbors to the north, east, west, and south of `(i, j)` is open, and if so, connect the site corresponding to `(i, j)` with the site corresponding to that neighbor
- `boolean isOpen(int i, int j)`
  - Return whether cell `(i, j)` is open or not
- `boolean isFull(int i, int j)`
  - Return whether cell `(i, j)` is full or not; a cell is full if it is open and its corresponding site is connected to the source site
- `int numberOfOpenSites()`
  - Return the number of open sites
- `boolean percolates()`
  - Return whether the system percolates or not; the system percolates if the sink site is connected to the source site

Problems

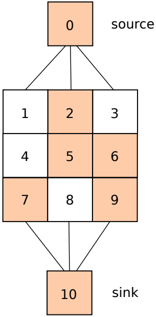
- Using virtual source and sink sites introduces what is called the *back wash* problem
- In the 3×3 system, consider opening the cells $(0, 1)$, $(1, 2)$,$(1, 1)$, $(2, 0)$, and $(2, 2)$, and in that order; the system percolates once $(2, 2)$ is opened



0, 0	0, 1	0, 2
1, 0	1, 1	1, 2
2, 0	2, 1	2, 2

open

full



0

source

1	2	3
4	5	6
7	8	9

10

sink

- The cell $(2, 0)$ is technically not full since it is not connected to an open cell in the top row via a path of neighboring (north, east, west, and south) open cells, but the corresponding site (7) is connected to the source, so is incorrectly reported as being full — this is the back wash problem
- To receive full credit for the problem, you need to fix the back wash issue

Problems

Problem 2 (*Estimate Percolation Threshold*) To estimate the percolation threshold, create a data type `PercolationStats` with the following API:

method	description
`PercolationStats(int N, int T)`	perform T independent experiments on an N-by-N grid
`double mean()`	sample mean of percolation threshold
`double stddev()`	sample standard deviation of percolation threshold
`double confidenceLow()`	low endpoint of 95% confidence interval
`double confidenceHigh()`	high endpoint of 95% confidence interval

Hints

- Instance variables
    - Number of independent experiments, `int T`
    - Percolation thresholds for the `T` experiments, `double[] p`
- `PercolationStats(int N, int T)`
    - Perform the following experiment `T` times
        - Create an $N \times N$ percolation system
        - Until the system percolates, choose a cell `(i, j)` at random and open it if it is not already open
        - Calculate percolation threshold as the fraction of sites opened, and store the value in `p[]`

Problems

- `double mean()`
  - Return the mean μ of the values in `p[]`
- `double stddev()`
  - Return the standard deviation σ of the values in `p[]`
- `double confidenceLow()`
  - Return $\mu - \frac{1.96\sigma}{\sqrt{T}}$
- `double confidenceHigh()`
  - Return $\mu + \frac{1.96\sigma}{\sqrt{T}}$

Epilogue

The `data` directory contains some sample files for use with the percolation clients, and associated with most input `.txt` files are output `.png` files that show the desired output

We provide two visualization clients that serve as large-scale visual traces and we highly recommend using them for testing and debugging your `Percolation` data type

1. `PercolationVisualizer` takes as command-line argument the name of a file specifying the size and open sites of a percolation system, and visually reports if the system percolates or not
2. `InteractivePercolationVisualizer` constructs an N-by-N percolation system, where N is specified as command-line argument, and allows you to interactively open sites in the system by clicking on them and visually inspect if the system percolates or not

Epilogue

Your project report (use the given template, `report.txt`) must include

- time (in hours) spent on the project
- short description of how you approached each problem, issues you encountered, and how you resolved those issues
- acknowledgement of any help you received
- other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Before you submit your files

- make sure your programs meet the input and output specifications by running the following command on the terminal

  ```
  $ python run_tests.py -v [<problems>]
  ```

- make sure your programs meet the style requirements by running the following command on the terminal

  ```
  $ check_style <program>
  ```

- make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes