CS25100: Data Structures and Algorithms, Spring 2017

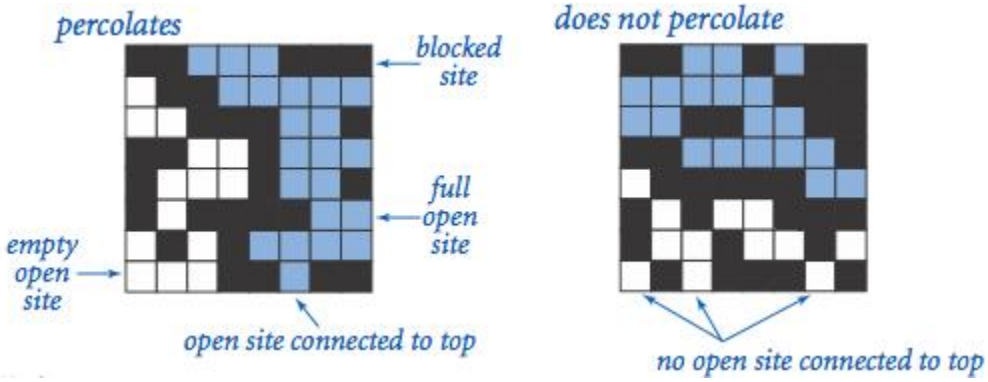
Project 1, Percolation

Handed out: Monday, January 23, 2017
Due: Monday, February 6, 2017 at 11:59pm.

Description

Percolation is a scientific model that is used to analyze the connectivity of systems. For example, it can be used to analyze if a porous landscape with water on the surface will eventually allow the water to drain through to the bottom. It can be used to analyze if oil would be able to reach the surface in a similar manner. The idea of the model is to analyze what conditions are necessary for the system to percolate (i.e. let the water or oil through).

The current assignment will allow students to apply the union-find data structure to solve this problem. The system will be represented as a *N*-by-*N* grid where each cell can be in one of 3 states: blocked (black), open (white) or full (blue). A grid where percolation has been achieved will have a path of full cells from the surface to the bottom.



The flow of the material goes from top to bottom and each cell routes that flow through the **top,left,right and bottom neighbors (4-neighbors)**. A system where at least one bottom-row cell is full (thanks to the flow route) is said to percolate.

Tasks

The current project will be divided into 3 programming tasks and an analysis phase that will require code simulation, experimental evaluation, and discussion of the results. You are free to use the implementation of QuickUnionUF and WeightedQuickUnionUF provided as starter code or develop your own.

If developing your own, replace the current files for QuickUnionUF and WeightedQuickUnionUF which have been provided, do not create new files for the same. You are also provided with 2 test cases along with their sample outputs for the tasks. Other than the files required, do not provide any other files as they may not be graded or your implementation may fail to execute or compile. Union find implementations have been provided in java and c++ and while you are free to choose either language, all code must be in the same language. Students are allowed use the standard library provided by the authors of the course textbook(if using java), along with the suggested implementations for union-find data structures in Algorithm 1.5 (Chapter 1). The standard library JAR file is provided on Vocareum in your workspace. Do not submit any output files, make files or the like. Only code files along with provided starter code are to be submitted.

Percolation API

Develop a Java class called Percolation that complies with the following interface:

- **public Percolation(int n):** Create a new n by n grid where all cells are initially blocked
- **public void open(int x, int y):** Open the site at coordinate (x,y), where x represents the row number and y the column number. For consistency purposes, (0,0) will be the bottom-left cell of the grid and (n-1,n-1) will be on the top-right. The graphical capabilities discussed later assume a similar convention.
- **public boolean isOpen(int x, int y):** Returns true if cell (x,y) is open due to a previous call to open(int x, int y)
- **public boolean isFull(int x, int y):** Returns true if there is a path from cell (x,y) to the surface (i.e. there is percolation up to this cell)
- **public boolean percolates():** Analyzes the entire grid and returns true if the whole system percolates
- **public static void main(String[] args):** Create a main method that reads a description of a grid from standard input and validates if the system described percolates or not, printing to standard output a simple "Yes" or "No" answer.

OR
Develop a C++ class called Percolation that complies with all the constructor/functions as public as mentioned below:

- **Percolation(int n):** Create a new n by n grid where all cells are initially blocked
- **void open(int x, int y):** Open the site at coordinate (x,y), where x represents the row number and y the column number. For consistency purposes, (0,0) will be the bottom-left cell of the grid and (n-1,n-1) will be on the top-right. The graphical capabilities discussed later assume a similar convention.
- **bool isOpen(int x, int y):** Returns true if cell (x,y) is open due to a previous call to open(int x, int y)
- **bool isFull(int x, int y):** Returns true if there is a path from cell (x,y) to the surface (i.e. there is percolation up to this cell)
- **bool percolates():** Analyzes the entire grid and returns true if the whole system percolates
- **int main(int argc, char *argv[]):** Create a main method that reads a description of a grid from

standard input and validates if the system described percolates or not, printing to standard output a simple "Yes" or "No" answer.

Visualization

Create a program called PercolationVisualizer.java/PercolationVisualizer.cpp that receives an input file the same way Percolation.java/Percolation.cpp does but, instead of displaying a Yes/No answer, it will output the results as well as intermediate steps in a file and also display the same on the terminal. The format for the output file and terminal will be (to be strictly followed) -:

- The first line will be length of the side of grid i.e N followed by an empty line.
- Then for each of the steps of the percolation process write the matrix state in the file and on terminal. Use 0 for blocked state, 1 for open state and 2 for full state.
- Each intermediate state of matrix is to be separated by a line, and each cell of the matrix is separated from its neighbour in the same row by a space.

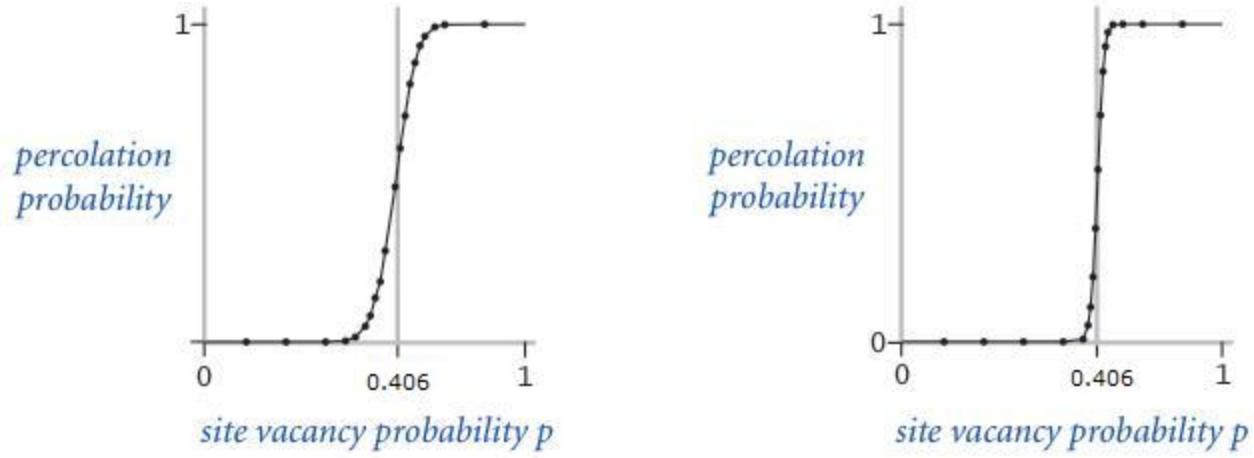
We have provided a file VisualizeFrames.java which you can use to draw the intermediate stages. For you to be able to visualize the PercolationVisualizer's output, your output file must be named visualMatrix.txt
Do not modify the code for VisualizeFrames.java. It is provided for students to give a visual flow of how percolation works.

NOTE: Even if you are using C++, you should use VisualizeFrames.java to visualize your output as this code only needs visualMatrix.txt

For grading purposes only small-sized grids will be used (up to n=25) thus, although Percolation code should handle any grid size theoretically, the visualization will only be checked using that size range.

Simulation

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 is 0, the system never percolates; when p is 1, the system always 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. In this project the threshold will be estimated using Monte Carlo simulations.

To do so, create a program called PercolationStats.java/PercolationStats.cpp that will generate systems randomly connected that can be used to estimate p^*. More specifically, do the following:

- Create a grid of size NxN where all cells are blocked initially
- Randomly select a blocked cell and open it
- Repeat the previous step until the system percolates
- Count the number of open cells and use that to obtain the p^* estimate.

The previous process can be repeated multiple times to obtain various estimates and the different values can be used to calculate a mean and standard deviation.
The main method in PercolationStats will receive three command line parameters, N, T and *type*, that will indicate the program to perform T repetitions of the previous process for a grid of size N-by-N. The type parameter will be a string, either slow or fast, which will determine the type of union method to be used. Algorithm 1.5 mentions 3 types of modifications that can be used to implement the union-find data structure. For this project, students will need to take into account quick find (slow) and the weighted quick union (fast). Depending on which parameter is given to the program a different implementation of the data structure should be used and it should reflect a distinct performance change from one to the other.

The output of the program should include the mean percolation threshold and the corresponding standard deviation. Additionally, it should output the total execution time of all experiments, the mean time for each experiment and its standard deviation. The following example shows a call that intends to run 50 simulations with a grid size of 20, using the weighted quick union method (the values after the = should be actual values in your program):

```
java PercolationStats 20 50 fast
```

```
**OUTPUT BELOW**
mean threshold=the_threshold_value
std dev=the_threshold_stddev
time=total_time_in_seconds
mean time=mean_time_in_seconds
stddev time=stddev_time_in_second
```

Assuming output is the name of the C++ executable

```
./output 20 50 fast

**OUTPUT BELOW**
mean threshold=the_threshold_value
std dev=the_threshold_stddev
time=total_time_in_seconds
mean time=mean_time_in_seconds
stddev time=stddev_time_in_second
```

Analysis

Using PercolationStats.java/PercolationStats.cpp, analyze the consequences of using different values of N and each of the two modifications to the data structure. Make a qualitative analysis of using T=30 and N={10,25,50,100,250,500}, for each of the two methods. What differences can be found in terms of execution time? What is the behavior of the estimates of p*?

Create a PDF file named yourName_yourSurname.pdf with your analysis and submit it along with your source code.
Your report should include the following two plots:

- **Running time:** Plot N on the x-axis and mean run time on the y-axis. You should show one line for the slow implementation and another for the fast one.
- ***p* estimates:*** Plot N on the x-axis and and mean p* on the y-axis. One line for each implementation.

Besides those plots you should include your discussion of the results found.

Sample inputs and outputs

The input file format is: the first line contains the value of n, as in the Percolation constructor parameter, and a variable number of extra lines each containing a cell coordinate, following the format "<x-value><space><y-value>". The program should open the sites in the same order they appear in the file and print the final result after processing all of them. For consistency purposes, (0,0) will be the bottom-left cell of the grid and (n-1,n-1) will be on the top-right, with the pair being . DemoYes.txt and DemoNo.txt are provided in your Vocareum workspace. The input to Percolation and PercolationVisualizer will be in this format.
Output of Percolation will be Yes or No and printed to terminal(please note the sentence case used)

Output of PercolationVisualizer will be the same in the output file visualMatrix.txt and on the terminal.
We have provided DemoVisYes.txt and DemoVisNo.txt in your workspace only for the purpose of format and so you know what the output should look like. You should name the output file as visualMatrix.txt .

Compilation and execution

Java

- javac -classpath .:stdlib.jar WeightedQuickUnionUF.java
- javac -classpath .:stdlib.jar QuickUnionUF.java
- javac -classpath .:stdlib.jar Percolation.java
- javac -classpath .:stdlib.jar PercolationVisualizer.java
- javac -classpath .:stdlib.jar PercolationStats.java
- javac -classpath .:stdlib.jar VisualizeFrames.java
- java -classpath .:stdlib.jar Percolation < testCase.txt
- java -classpath .:stdlib.jar PercolationVisualizer < testCase.txt
- java -classpath .:stdlib.jar PercolationStats 20 50 fast
- java -classpath .:stdlib.jar VisualizeFrames

C++

For C++, please initialize all variables to prevent errors on the platform due to garbage values and different gcc versions

- g++ -std=c++11 PercolationVisualizer.cpp WeightedQuickUnionUF.cpp -o output
- g++ -std=c++11 Percolation.cpp WeightedQuickUnionUF.cpp -o output
- ./output < testCase.txt

For Compiling and Executing Simulation program in C++
Please ensure that main method is present only in PercolationStats.cpp and there are no main methods in Percolation.cpp and PercolationQuick.cpp while testing Simulation. However, please uncomment all the main methods during submission

- g++ -std=c++11 PercolationStats.cpp PercolationQuick.cpp Percolation.cpp WeightedQuickUnion.cpp QuickUnion.cpp -o output
- ./output 4 3 fast

Submission

Submit your solution before **Monday, February 6, 2017 at 11:59pm**. Vocareum will be used to submit assignments. You can access Vocareum using the link provided for this assignment on

blackboard, which will auto enroll you once you do so.

Upload all source code used and libraries using shift select(or drag the mouse and select all the files) to your workspace. If you upload the code in a zip folder, the code will not be at the root and grading will fail. The report should be uploaded too. Finally, provide a README file with your name, instructions to compile and run your code (Please do not use absolute paths anywhere in the code) and anything you would like us to know about your program (like errors, special conditions, etc).
Since you will need to use 2 implementations of union-find, for the purposes of grading, submit your Percolation.java/Percolation.cpp file using the Weighted Quick Union Find implementation.

NOTE:File names must be exactly as provided in compilation instructions. Any difference will lead to code not being compiled due to the auto grading system being used. Also output formats are strictly enforced. No extra printing to be done other than what is asked. Please comment out all other print statements before submitting

Based on an assignment developed by Bob Sedgewick and Kevin Wayne.
Copyright © 2008.