TL;DR

You're encouraged to use the TL;DR document to understand what to do and how to do it at a very high level.

You can talk with one person, a partner, when completing the analysis. You cannot have a partner in developing code, that should be your own effort as is typical with 201 assignments. You can discuss and collaborate deeply on the analysis with your analysis partner.

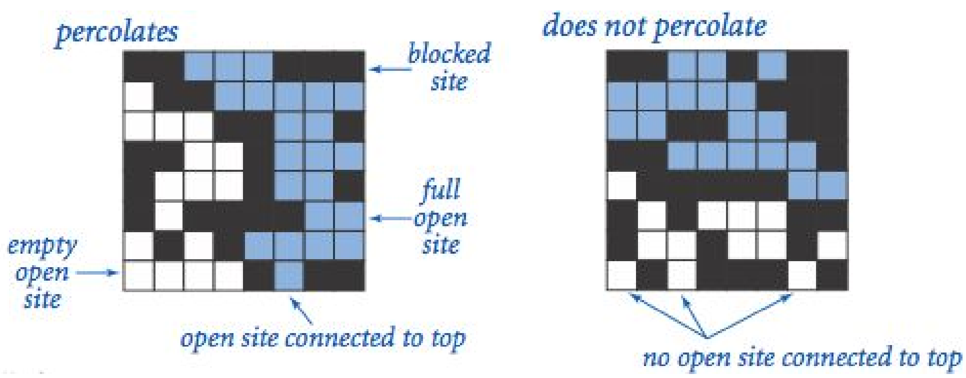
Overview with Motivation/Explanation

In this assignment, you will write a program to estimate the value of the percolation threshold via Monte Carlo simulation. In doing so, you will better understand depth-first-search, union-find structures, and the use of computer simulations for statistical inquiry. ***Your goal will be to explore trade-offs in several approaches to estimate the percolation threshold in an NxN system.***

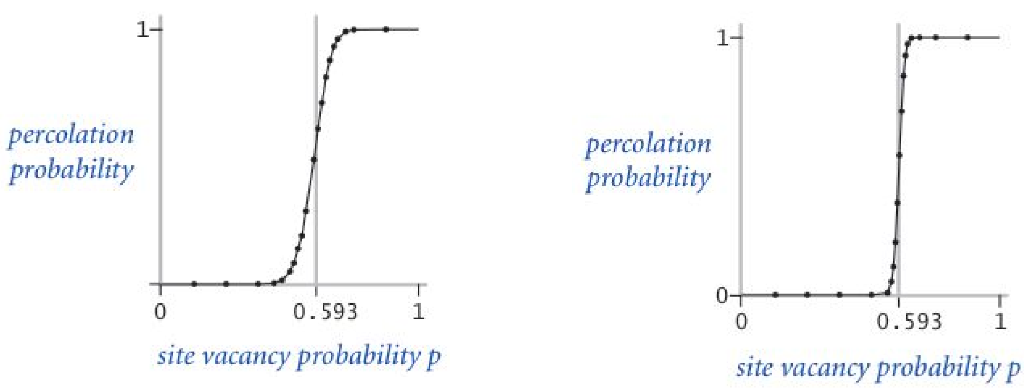
Given a porous landscape with water on the surface (or oil below), under what conditions will the water be able to drain through to the bottom (or the oil to gush through to the surface)? Given a composite systems comprised of randomly distributed insulating and metallic materials, what fraction of the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

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***. In diagrams we color full sites blue to model water flowing from the top through the system. We say the ***system percolates if there is at least one full site in the bottom row***. In other words, a system percolates if there is a path of open sites from the top row to the bottom row. 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.

For more on percolation see the Princeton Case Study.



The percolation threshold problem is: 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? In other words, in a N-by-N grid, would the system percolate if N^2p randomly chosen cells are opened? 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 suite of computer programs to visualize the percolation process and estimate p* using Monte Carlo techniques. As you can see above, the percolation threshold in an NxN grid is about 0.593. The size of the grid doesn't matter as your simulations will show.

The videos linked here show (1) an interactive simulation where you choose to open sites and (2) help explain the techniques you'll read about. These videos may be helpful after reading the assignment, or to get grounded before reading.

These videos help understand two parts of the assignment.

Open, full, DFS, and what percolation means

1. https://www.youtube.com/watch?v=ikVIiuCR4pk

From DFS to Union-Find: two approaches compared/contrasted

2. https://www.youtube.com/watch?v=IpYvgV5m1qM

Git

Fork, clone, and import the cloned project from the file system. Use this URL from the course GitLab site: https://coursework.cs.duke.edu/201spring19/percolation ***. Be sure to fork first*** (see screen shot). Then Clone using the SSH URL after using a terminal window to cd into your Eclipse workspace. Recall that ***you should import using*** the File>Import>General>Existing Projects into Workspace -- then navigate to where you cloned the diyad project.

DO NOT DO NOT import using the Git open -- use General>Existing Projects Into Workspace.

Pushing to Git

When you make a series of changes you want to 'save', you'll push those changes to your GitLab repository. You should do this after major changes, certainly every hour or so of coding. You'll need to use the standard Git sequence to commit and push to GitLab:

```
git add .
git commit -m 'a short description of your commit here'
git push
```

Overview of Programming

You'll create three new implementations of the **`IPercolate`** interface: one that uses a faster DFS, one that uses BFS, and one that uses the Union-Find algorithm rather than DFS/BFS.

1. Create a subclass of **`PercolationDFS`** named **`PercolationDFSFast`** that has a different implementation of the method **`updateOnOpen`** that will make the implementation faster.
2. Create a subclass of **`PercolationDFSFast`** named **`PercolationBFS`** that has a different implementation of the method **`dfs`**. This implementation will use BFS with a **`Queue`** rather than recursion.
3. Create class **`PercolationUF`** that implements the **`IPercolate`** interface. By using a Union-Find data structure, this class serves as a bridge between classes that implement the **`IUnionFind`** interface and those running a percolation simulation (that implement **`IPercolate`**).

You'll time each of the three and report on the timings in the **`analysis.txt`** file you create.

For each class you create you'll use one of the JUnit testing files to test your implementation. ***You'll need to change the class each uses to test your implementation.*** You should also use the **`InteractivePercolationVisualizer`** to help verify your classes work correctly.

PercolationDFSFast

Your new class **`PecolationDFSFast`** will extend **`PercolationDFS`** and thus inherit state and methods from that class.

1. You'll need to create a constructor with an int/size parameter that calls super to initialize the state in the parent class.
2. You'll need to **`@Override updateOnOpen`** to be more efficient.

In the **`updateOnOpen`** from **`PercolationDFS`** all cells marked as **`FULL`** are “cleared” or changed so that they're marked as **`OPEN`**, then the method **`dfs`** is called from every cell on the top row. Calling **`dfs`** on any open cell will explore all paths from that cell and mark all open cells as **`FULL`**. This is very inefficient since cells marked as **`FULL`** are cleared each time a new cell is opened, then marked as **`FULL`**, then cleared and the process repeats for each newly open cell, e.g., in a simulation.

You'll change this method so it does ***not*** clear all cells, but instead calls **`dfs`** once if the newly open cell should be marked as **`FULL`** because it's in the top row or because it's adjacent to an already **`FULL`** cell.

1. Determine if the newly open cell (parameters **`row`** and **`col`** of **`updateOnOpen`**) should be marked as full. To check:
    a. Is the cell in the top row? If so, it should be marked as full.
    b. Is the cell adjacent to a full cell? If so it should be marked as full.
2. If the cell should be marked as full, call **`dfs(row,col)`**

Note that you'll make one call of **`dfs`**. ***The `dfs` method is inherited and does not change. You must call dfs and NOT super.dfs in the code you write.*** The **`dfs`** implementation will mark an open cell as **`FULL`** and make recursive calls for each neighbor. The **`dfs`** implementation checks for cells already marked as **`FULL`** or that are not **`OPEN`** and does not visit these cells.

PercolationBFS

This class extends the **`PercolationDFSFast`** class you wrote previously. You'll override the **`dfs`** method inherited from **`PercolationDFS`** to use a **`Queue`** and a breadth-first-search (BFS) approach. Use the ideas from [IterativeBlobModel.java]() we've seen in class, but you'll create queue of **`int`** values and not **`Pair`** values. The **`dfs`** method will be called from the inherited **`updateOnOpen`** from **`PercolationDFSFast`**.

1. You'll need to create a constructor with an int/size parameter that calls super to initialize the state in the parent class.
2. You'll need to **`@Override dfs`** to use a **`Queue`** as explained below.

Create a **`Queue<Integer>`** from a **`LinkedList`** object since that class implements the **`Queue`** interface (see [IterativeBlobModel.java]() for details). Whereas the original dfs used recursion, this method will first mark the cell at (row,col) as **`FULL`** and put the cell on the queue, then repeat the following process:

- Dequeue a cell. (All cells in the queue should have been marked as **`FULL`**)
- Check the dequeued cell's four neighbors. ***If the neighboring cell is open and not full, it should be marked as full and enqueued.*** This is similar to the check in the recursive dfs method where cells that are marked as open but not full are visited by the recursive call.

To map (row,col) pairs to an integer value, use **`row*size + col`**, where **`size`** is the N in the NxN grid, e.g., use **`myGrid.length`** for **`size`**. When dequeuing an int **`value`**, you can determine the corresponding (row,col) using **`value/size`** and **`value%size`**, respectively. You can use a helper method to do this, or simply use **`row*myGrid.length + size`** each time an **`int`** value is required.

PercolationUF

This class implements the **IPercolate** interface and will use an **IUnionFind** object to keep track of open and full cells. See the video at the beginning of the assignment for general ideas. Each of the NxN cells is mapped to a number, and these numbers represent the set for the disjoint-set/union-find algorithm. You'll need two additional numbers, **VTOP** and **VBOTTOM**, for a total of N^2+2 values.

Instance variables for PercolationUF

1. A two-dimensional array of boolean values, **myGrid**, that represents whether a cell is open. Initially **myGrid[r][c]** should be **false** which is the default value when you create the grid. Each time a cell (r,c) is open, **myGrid[r][c]** will be set to true.
2. An integer **myOpenCount** which is the number of open cells, i.e., the number of true values in **myGrid**.
3. An **IUnionFind** object **myFinder** to store/reference the **IUnionFind** object passed to the constructor (which should be a **QuickUWPC** object in this assignment).
4. Two final values for **VTOP** and **VBOTTOM**, set to size*size and size*size+1, for example, in the constructor.
```java
private final int VTOP;
private final int VBOTTOM;
```

Constructor for PercolationUF

Create a constructor **PercolationUF(int size, IUnionFind finder)** that will construct and initialize the NxN grid stored in the instance variable **myGrid** (where N is given by the **size** parameter). The constructor should initialize the **IUnionFind** object ***of size NxN + 2*** by calling **finder.initialize** appropriately and then storing this object in the appropriate instance variable which is **myFinder**.

Methods for PercolationUF

You must **@Override** each method from the **IPercolate** interface. As with methods you can see in **PercolationDFS**, these methods you write with (row,col) parameters should throw exceptions when the (row,col) are not in bounds.

As with the Queue described in [PercolationBFS](#) above you'll need to map a (row,col) pair to an int value to use with the **IUnionFind** object referenced by **myFinder**. Be sure to deal with **VTOP** and **VBOTTOM** as well.

1. Method **isOpen** throws an Exception when needed and otherwise simply returns the appropriate **myGrid** value.
2. Method **isFull** throws an Exception when needed and otherwise checks if the (row,col) cell is connected to **VTOP**.

3. Method **`percolates`** checks to see if **`VTOP`** is connected to **`VBOTTOM`**.
4. Method **`numberOfOpenSites`** simply returns the value of the appropriate instance variable.
5. Method **`open`** throws an Exception when needed, checks to be sure the cell is not already open, and then marks the cell as open and connects with open neighbors as described below.

When a cell is marked as open, you'll write code to check each of the cell's four neighbors. If any of these cells is open, the newly marked cell and the open neighbor should be connected by a call to **`myFinder.union`**. If the newly marked cell is in the top row it should be connected to **`VTOP`** by a call to **`myFinder.union`**. If the newly marked cell is in the bottom row it should be connected to **`VBOTTOM`** by a call to **`myFinder.union`**.

Testing `PercolationUF`

You can test using **`TestUFPercolation`**. Create a **`QuickUWPC`** object to use with your **`PercolationUF`** object and run JUnit tests. You should also use the **`InteractivePercolationVisualizer`** to help verify your classes work correctly. You'll need to uncomment/comment a line in **`getPercolation`** of the **`TestUFPercolation`** class.

Analysis

Copy/Paste the runs described below and answer the questions indicated in the analysis.txt file. Here is a copy of the file.

You're given PercolationStats.java which performs benchmarks for an **`IPercolate`** object using grid sizes of 100, 200, 400, 800, 1600, and 3200. If you don't change the value of the public **`RANDOM_SEED`** variable you should see the same mean values of the Percolation threshold shown below. Your timing values may vary, but for all implementations using 20 trials you should see the same mean and standard deviation values.

```
simulation data for 20 trials
grid  mean       stddev     time
100   0.593      0.014      0.142
200   0.591      0.010      0.150
400   0.590      0.006      0.958
800   0.594      0.004      5.925
1600  0.592      0.002      39.266
3200  0.593      0.001      183.074
```

Copy/paste the results for each **`IPercolate`** object (**`PercolationDFSFast`**, **`PercolationBFS`**, **`PercolationUF`**) as indicated in the analysis.txt file. Then answer these questions using data from **`PercolationUF`** with **`QuickUWPC`**.

1. How does doubling the grid size affect running time (keeping # trials fixed)
2. How does doubling the number of trials affect running time.
3. Estimate the largest grid size you can run in 24 hours with 20 trials. Explain your reasoning.

After completing the analysis questions you should push to Git and run the autograder again to make sure you response is submitted to Gradescope. Please note that Analysis is part of the assignment, and Gradescope is the only portal to submit the assignment, including the analysis. We reserve the right to reject your analysis submission if you forget to upload it to Gradescope, even if it's in your Git repo.

Submitting

You'll submit the code to Gradescope after pushing program and **`analysis.txt`** file to GitLab.

Reflect

Complete the Reflect form linked here.

You must complete the Reflect form to get all points, but you should NOT complete the reflect form until you're done with all the coding portion of the assignment. Since you may uncover bugs from the autograder, you should wait until you've completed debugging and coding before completing the reflect form.

Grading

Points	Grading Criteria
6	PercolationDFSFast
6	PercolationBFS
6	PercolationUF
8	Analysis, UTAs will grade and comment

22-26: A
17-21: B
11-16: C
7-10: D

Due date: on time 4/16, 24 hours no penalty until 4/17, 10% late on 4/22, then 20% and so on.