CS5003: Design and Analysis of Algorithms

Programming Assgnments

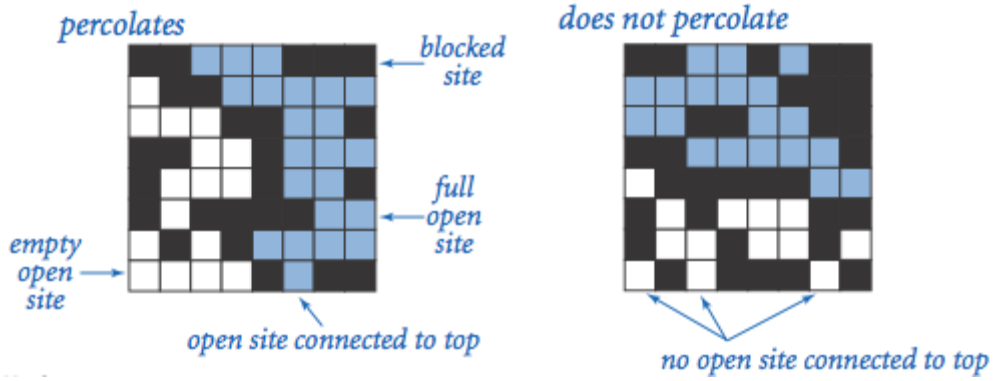
1. 使用合并-查找数据结构，实现估计渗漏(Percolation)问题阈值的程序。

Write a program to estimate the value of the *percolation threshold* via Monte Carlo simulation.

Install a Java programming environment. Install a Java programming environment on your computer by following these step-by-step instructions for your operating system [Mac OS X · Windows · Linux]. After following these instructions, the commands javac-algs4 and java-algs4 will classpath in both stdlib.jar and algs4.jar: the former contains libraries for reading data from *standard input*, writing data to *standard output*, drawing results to *standard draw*, generating random numbers, computing statistics, and timing programs; the latter contains all of the algorithms in the textbook.

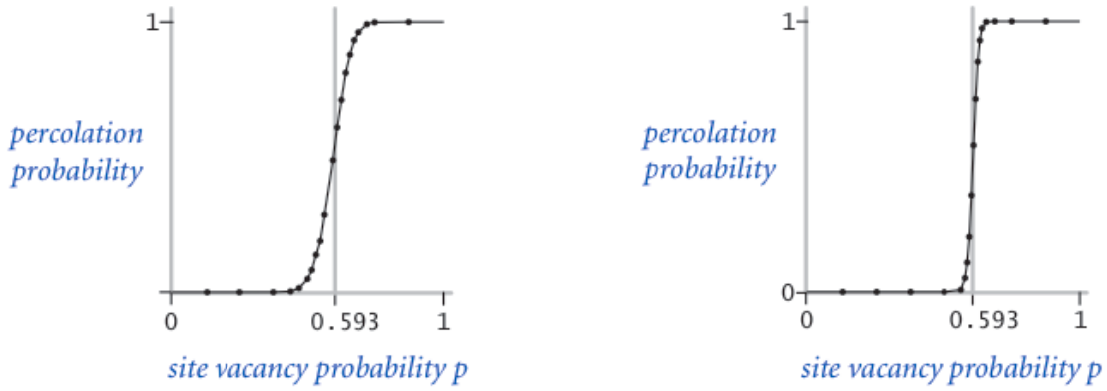
Percolation. Given a composite systems comprised of randomly distributed insulating and metallic materials: what fraction of the materials need to be metallic so that the composite system is an electrical conductor? Given a porous landscape with water on the surface (or oil below), under what conditions 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.)



The problem. 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 computer program to estimate p^*.

Percolation data type. To model a percolation system, create a data type Percolation with the following API:

```
public class Percolation {
   public Percolation(int N)              // create N-by-N grid, with all sites blocked
   public void open(int i, int j)         // open site (row i, column j) if it is not already
   public boolean isOpen(int i, int j)    // is site (row i, column j) open?
   public boolean isFull(int i, int j)    // is site (row i, column j) full?
   public boolean percolates()            // does the system percolate?
   public static void main(String[] args) // test client, optional
}
```

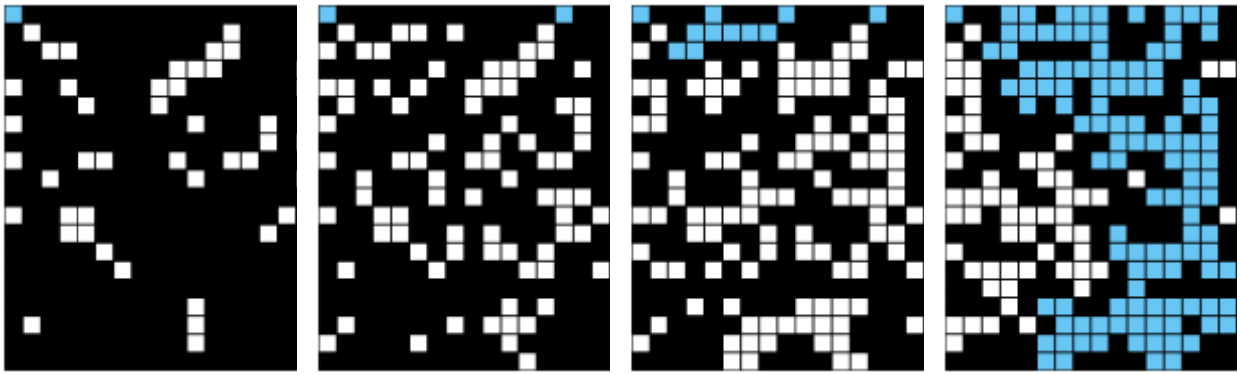
By convention, the row and column indices i and j are integers between 1 and N, where (1, 1) is the upper-left site: Throw an IndexOutOfBoundsException if any argument to open(), isOpen(), or isFull() is outside its prescribed range. The constructor should throw an IllegalArgumentException if $N \leq 0$. The constructor should take time proportional to N^2; all methods should take constant time plus a constant number of calls to the union-find methods union(), find(), connected(), and count().

Monte Carlo simulation. To estimate the percolation threshold, consider the following computational experiment:

- Initialize all sites to be blocked.
- Repeat the following until the system percolates:
  - Choose a site (row i, column j) uniformly at random among all blocked sites.
  - Open the site (row i, column j).

- The fraction of sites that are opened when the system percolates provides an estimate of the percolation threshold.

For example, if sites are opened in a 20-by-20 lattice according to the snapshots below, then our estimate of the percolation threshold is 204/400 = 0.51 because the system percolates when the 204th site is opened.



50 open sites *100 open sites* *150 open sites* *204 open sites*

By repeating this computation experiment T times and averaging the results, we obtain a more accurate estimate of the percolation threshold. Let x_t be the fraction of open sites in computational experiment t. The sample mean μ provides an estimate of the percolation threshold; the sample standard deviation σ 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:

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

To perform a series of computational experiments, create a data type PercolationStats with the following API.

```
public class PercolationStats {
   public PercolationStats(int N, int T)      // perform T independent computational
experiments on an N-by-N grid
   public double mean()                       // sample mean of percolation threshold
   public double stddev()                     // sample standard deviation of percolation
threshold
   public double confidenceLo()               // returns lower bound of the 95% confidence
interval
   public double confidenceHi()               // returns upper bound of the 95% confidence
interval
   public static void main(String[] args)     // test client, described below
}
```

The constructor should throw a java.lang.IllegalArgumentException if either $N \le 0$ or $T \le 0$.

Also, include a main() method that takes two *command-line arguments* N and T, performs T independent computational experiments (discussed above) on an N-by-N grid, and prints out the mean, standard deviation, and the *95% confidence interval* for the percolation threshold. Use *standard random* from our standard libraries to generate random numbers; use *standard statistics* to compute the sample mean and standard deviation.

```
% java PercolationStats 200 100
mean                    = 0.5929934999999997
stddev                  = 0.00876990421552567
95% confidence interval = 0.5912745987737567, 0.5947124012262428

% java PercolationStats 200 100
mean                    = 0.592877
stddev                  = 0.009990523717073799
95% confidence interval = 0.5909188573514536, 0.5948351426485464

% java PercolationStats 2 10000
mean                    = 0.666925
stddev                  = 0.11776536521033558
95% confidence interval = 0.6646167988418774, 0.6692332011581226

% java PercolationStats 2 100000
mean                    = 0.6669475
stddev                  = 0.11775205263262094
95% confidence interval = 0.666217665216461, 0.6676773347835391
```

Analysis of running time and memory usage (optional and not graded). Implement the Percolation data type using the quick-find algorithm QuickFindUF.java from algs4.jar.

- Use the *stopwatch data type* from our standard library to measure the total running time of PercolationStats. How does doubling N affect the total running time? How does doubling T affect the total running time? Give a formula (using tilde notation) of the total running time on your computer (in seconds) as a single function of both N and T.
- Using the 64-bit memory-cost model from lecture, give the total memory usage in bytes (using tilde notation) that a Percolation object uses to model an N-by-N percolation system. Count all memory that is used, including memory for the union-find data structure.

Now, implement the Percolation data type using the weighted quick-union algorithm WeightedQuickUnionUF.java from algs4.jar. Answer the questions in the previous paragraph.

Deliverables. Submit only Percolation.java (using the weighted quick-union algorithm as implemented in the WeightedQuickUnionUF class) and PercolationStats.java. We will supply stdlib.jar and WeightedQuickUnionUF. Your submission may not call any library functions other than those in java.lang, stdlib.jar, and WeightedQuickUnionUF.

For fun. Create your own percolation input file and share it in the discussion forums.

This assignment was developed by Bob Sedgewick and Kevin Wayne.

2. You are asked to write and perform an empirical complexity analysis of the following sorting algorithms://实现并做性能分析比较

(1) Insertion sort;
(2) Merge sort (please write the recursive merge sort and bottom-up mergesort);
(3) Quick sort (please write a randomized quicksort method as covered in class);
(4) Quicksort with Dijkstra 2-way partitioning;
(5)* Quicksort with Bentley-McIlroy 3-way partitioning.
Note that the question with a star is optional.

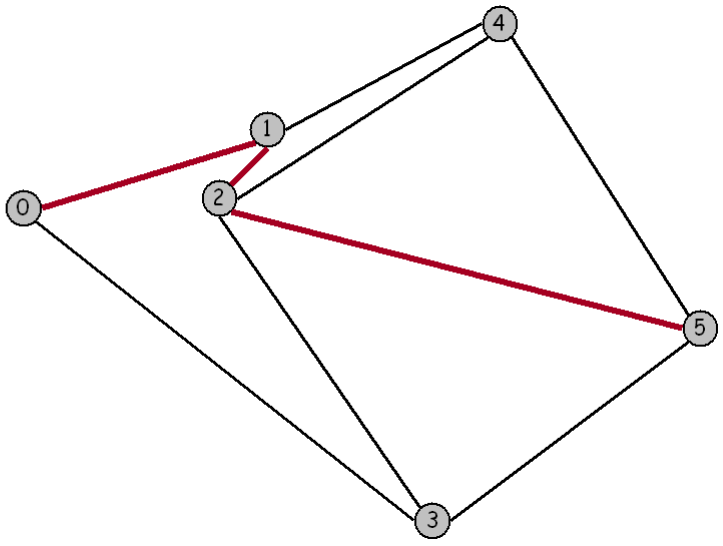
3. 实现 Dijkstra 单源点最短路径算法，并应用于 Map Routing/GIS 问题。

Implement the classic Dijkstra's shortest path algorithm and optimize it for maps. Such algorithms are widely used in geographic information systems (GIS) including MapQuest and GPS-based car navigation systems.

Maps. For this assignment we will be working with *maps*, or graphs whose vertices are points in the plane and are connected by edges whose weights are Euclidean distances. Think of the vertices as cities and the edges as roads connected to them. To represent a map in a file, we list the number of vertices and edges, then list the vertices (index followed by its x and y coordinates), then list the edges (pairs of vertices), and finally the source and sink vertices. For example, Input6 represents the map below:

```
6 9
0 1000 2400
1 2800 3000
2 2400 2500
3 4000    0
4 4500 3800
5 6000 1500

0 1
0 3
1 2
1 4
2 4
2 3
2 5
3 5
4 5
0 5
```



Dijkstra's algorithm. Dijkstra's algorithm is a classic solution to the shortest path problem. It is described in section 4.4 in the text book. The basic idea is not difficult to understand. We maintain, for every vertex in the graph, the length of the shortest known path from the source to that vertex, and we maintain these lengths in a *priority queue*. Initially, we put all the vertices on the queue with an artificially high priority and then assign priority 0.0 to the source. The algorithm proceeds by taking the lowest-priority vertex off the PQ, then checking all the vertices

that can be reached from that vertex by one edge to see whether that edge gives a shorter path to the vertex from the source than the shortest previously-known path. If so, it lowers the priority to reflect this new information.

Here is a step-by-step description that shows how Dijkstra's algorithm finds the shortest path 0-1-2-5 from 0 to 5 in the example above.

```
process 0 (0.0)
    lower 3 to 3841.9
    lower 1 to 1897.4
process 1 (1897.4)
    lower 4 to 3776.2
    lower 2 to 2537.7
process 2 (2537.7)
    lower 5 to 6274.0
process 4 (3776.2)
process 3 (3841.9)
process 5 (6274.0)
```

This method computes the length of the shortest path. To keep track of the path, we also maintain for each vertex, its predecessor on the shortest path from the source to that vertex. The files EuclideanGraph.java, Point.java, IndexPQ.java, IntIterator.java, and Dijkstra.java provide a bare bones implementation of Dijkstra's algorithm for maps, and you should use this as a starting point. The client program ShortestPath.java solves a single shortest path problem and plots the results using turtle graphics. The client program Paths.java solves many shortest path problems and prints the shortest paths to standard output. The client program Distances.java solves many shortest path problems and prints only the distances to standard output.

Your goal. Optimize Dijkstra's algorithm so that it can process thousands of shortest path queries for a given map. Once you read in (and optionally preprocess) the map, your program should solve shortest path problems in *sublinear* time. One method would be to precompute the shortest path for all pairs of vertices; however you cannot afford the quadratic space required to store all of this information. Your goal is to reduce the amount of work involved per shortest path computation, without using excessive space. We suggest a number of potential ideas below which you may choose to implement. Or you can develop and implement your own ideas.

Idea 1. The naive implementation of Dijkstra's algorithm examines all V vertices in the graph. An obvious strategy to reduce the number of vertices examined is to stop the search as soon as you discover the shortest path to the destination. With this approach, you can make the running time per shortest path query proportional to E' log V' where E' and V' are the number of edges and vertices examined by Dijkstra's algorithm. However, this requires some care because just re-initializing all of the distances to ∞ would take time proportional to V. Since you are doing repeated queries, you can speed things up dramatically by only re-initializing those values that changed in the previous query.

Idea 2. You can cut down on the search time further by exploiting the Euclidean geometry of the problem, as described in section 21.5 in the book of Algorithm in C Part V. For general graphs, Dijkstra's relaxes edge v-w by updating d[w] to the sum of d[v] plus the distance from v to w. For maps, we instead update d[w] to be the sum of d[v] plus the distance from v to w *plus*

the Euclidean distance from w to d *minus* the Euclidean distance from v to d. This is known as the *A* algorithm*. This heuristics affects performance, but not correctness.

Idea 3. Use a faster priority queue. There is some room for optimization in the supplied priority queue. You could also consider using a multiway heap as in Sedgewick Program 20.10.

Testing. The file usa.txt contains 87,575 intersections and 121,961 roads in the continental United States. The graph is very sparse - the average degree is 2.8. Your main goal should be to answer shortest path queries quickly for pairs of vertices on this network. Your algorithm will likely perform differently depending on whether the two vertices are nearby or far apart. We provide input files that test both cases. You may assume that all of the x and y coordinates are integers between 0 and 10,000.

4*. Huffman Compression

前三个题目是必做题，后一个题目是选做题。