《算法设计与分析》实验教学大纲

课程编号：CS5102L

课程名称：算法设计与分析　　英文名称：Design and Analysis of Algorithms

学分/学时：2.5/40　　课程性质：专业选修

适用专业：计算机科学与技术　　建议开设学期：5

物联网工程，卓越，教改

先修课程：离散数学，数据结构　　开课单位：计算机学院

JAVA 程序设计

一、实验简介

本实验要求学生能够综合运用排序、搜索、图处理和字符串处理的基础算法和数据结构，将算法理论、算法工程和编程实践相结合开发相应的软件，解决科学、工程和应用环境下的实际问题。使学生能充分运用并掌握算法设计与分析的方法以及算法工程技术，为从事计算机工程和软件开发等相关工作打下坚实的基础。

二、实验课程目标与毕业要求

通过本课程的学习使学生系统掌握算法设计与分析的基本概念和基本原理，理解排序、搜索、图处理和字符串处理的算法设计理论及性能分析方法，掌握排序、搜索、图处理和字符串处理的数据结构与算法实现技术。课程强调算法的开发及 Java 实现，理解相应算法的性能特征，评估算法在应用程序中的潜在性能。

三、实验内容及基本要求

（一）渗透问题（Percolation）

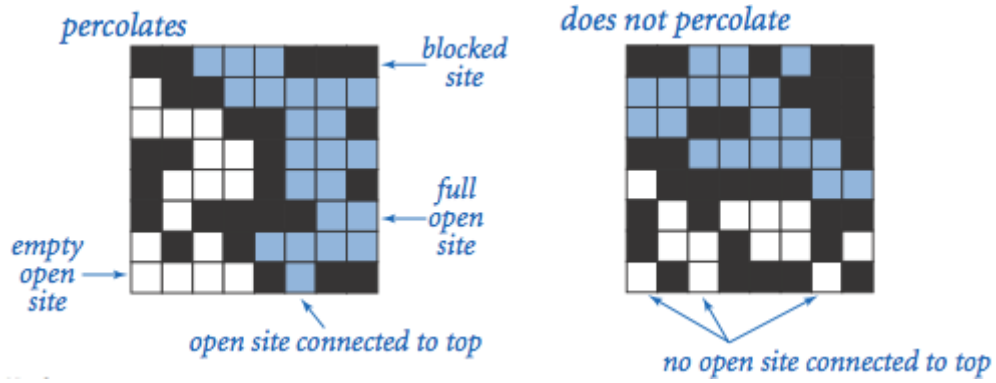
使用合并-查找（union-find）数据结构，编写程序通过蒙特卡罗模拟（Monte Carlo simulation）来估计渗透阈值。

安装 Java 编程环境。按照以下各步指令，在计算机上（操作系统 Mac OS X（http://algs4.cs.princeton.edu/mac）· Windows （http://algs4.cs.princeton.edu/windows）· Linux（http://algs4.cs.princeton.edu/linux）安装 Java 编程环境。执行这些指令后，在你的 Java classpath 下会有 stdlib.jar and algs4.jar。前者包含库：从标准输入读数据、向标准输出写数据

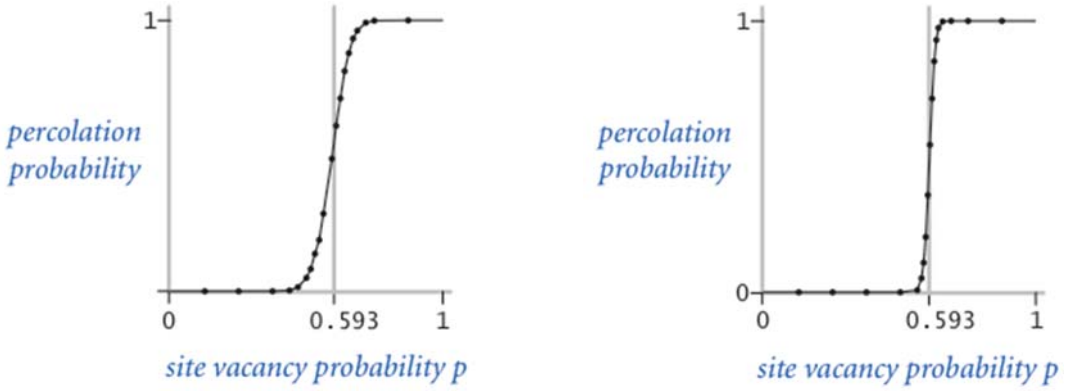
以及向标准绘制绘出结果，产生随机数、计算统计量以及计时程序；后者包含了教科书中的所有算法。

给定由随机分布的绝缘材料和金属材料构成的组合系统：金属材料占多大比例才能使组合系统成为电导体？ 给定一个表面有水的多孔景观（或下面有油），水将在什么条件下能够通过底部排出（或油渗透到表面）？ 科学家已经定义了一个称为渗透（*percolation*）的抽象过程来模拟这种现象。

模型。 我们使用 $N \times N$ 网格点来模型化一个渗透系统。 每个格点或是 *open* 格点或是 *blocked* 格点。 一个 *full* site 是一个 *open* 格点，它可以通过一系列的邻近（左、右、上、下）*open* 格点连通到顶行的一个 *open* 格点。如果在底行中存在一个 *full* site 格点，则称系统是渗透的。（对于绝缘/金属材料的例子，*open* 格点对应于金属材料，渗透系统有一条从顶行到底行的金属路径，且 *full* sites 格点导电。对于多孔物质示例，*open* 格点对应于空格，水可能流过，从而渗透系统使水充满 *open* 格点，自顶向下流动。）



科学问题：如果将格点以概率 p 独立地设置为 *open* 格点（因此以概率 $1-p$ 被设置为 *blocked* 格点），系统渗透的概率是多少？ 当 $p = 0$ 时，系统不会渗出；当 $p=1$ 时，系统渗透。下图显示了 20×20 随机网格（左）和 100×100 随机网格（右）的格点空置概率 p 与渗滤概率。



当 N 足够大时，存在阈值 p^*，使得当 $p < p^*$，随机 $N \times N$ 网格几乎不会渗透，并且当 $p > p^*$ 时，随机 $N \times N$ 网格几乎总是渗透。 尚未得出用于确定渗滤阈值 p^*的数学解。你的任务是编写一个计算机程序来估计 p^*。

Percolation 数据类型。模型化一个 Percolation 系统，创建含有以下 API 的数据类型 Percolation。

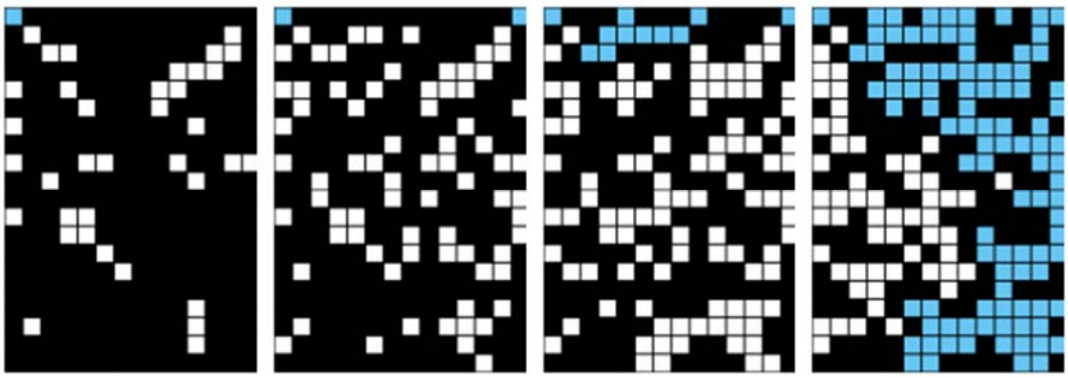
```
public class Percolation {
   public Percolation(int N)                // create N-by-N grid, with all sites blocked
   public void open(int i, int j)           // open site (row i, column j) if it is not already
   public boolean isOpen(int i, int j)      // is site (row i, column j) open?
   public 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
}
```

约定行 *i* 列 *j* 下标在 1 和 *N* 之间，其中(1, 1)为左上格点位置：如果 open(), isOpen(), or isFull() 不在这个规定的范围，则抛出 IndexOutOfBoundsException 例外。如果 $N \le 0$，构造函数应该抛出 IllegalArgumentException 例外。构造函数应该与 N^2 成正比。所有方法应该为常量时间加上常量次调用合并-查找方法 union(), find(), connected(), and count()。

蒙特卡洛模拟（Monte Carlo simulation）. 要估计渗透阈值，考虑以下计算实验：

- 初始化所有格点为 *blocked*。
- 重复以下操作直到系统渗出：
  - 在所有 *blocked* 的格点之间随机均匀选择一个格点 (row *i*, column *j*)。
  - 设置这个格点(row *i*, column *j*)为 *open* 格点。
- *open* 格点的比例提供了系统渗透时渗透阈值的一个估计。

例如，如果在 20×20 的网格中，根据以下快照的 *open* 格点数，那么对渗滤阈值的估计是 204/400 = 0.51，因为当第 204 个格点被 *open* 时系统渗透。



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

通过重复该计算实验 *T* 次并对结果求平均值，我们获得了更准确的渗滤阈值估计。 令 x_t 是第 *t* 次计算实验中 *open* 格点所占比例。样本均值μ提供渗滤阈值的一个估计值；样本标准差σ测量阈值的灵敏性。

$$\mu = \frac{x_1 + x_2 + \cdots + x_T}{T}, \quad \sigma^2 = \frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_T - \mu)^2}{T - 1}$$

假设 *T* 足够大（例如至少 30），以下为渗滤阈值提供 95％置信区间：

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

通过创建数据类型 PercolationStats 来执行一系列计算实验，包含以下 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
}
```

在 $N \le 0$ 或 $T \le 0$ 时，构造函数应该抛出 java.lang.IllegalArgumentException 例外。

此外，还包括一个 main()方法，它取两个命令行参数 N 和 T，在 $N \times N$ 网格上进行 T 次独立的计算实验（上面讨论），并打印出均值μ、标准差σ和 95% 渗透阈值的置信区间。 使用标准库中的标准随机数生成随机数； 使用标准统计库来计算样本均值和标准差。

```
Example values after creating PercolationStats(200, 100)
mean()                    = 0.5929934999999997
stddev()                  = 0.00876990421552567
confidenceLow()           = 0.5912745987737567
confidenceHigh()          = 0.5947124012262428

Example values after creating PercolationStats(200, 100)
mean()                    = 0.592877
stddev()                  = 0.009990523717073799
confidenceLow()           = 0.5909188573514536
confidenceHigh()          = 0.5948351426485464

Example values after creating PercolationStats(2, 100000)
mean()                    = 0.6669475
stddev()                  = 0.11775205263262094
confidenceLow()           = 0.666217665216461
confidenceHigh()          = 0.6676773347835391
```

运行时间和内存占用分析。

使用 ***quick-find*** 算法（QuickFindUF.java from algs4.jar）实现 Percolation 数据类型。进行实验表明当 N 加倍时对运行时间的影响；使用近似表示法，给出在计算机上的总时间，它是输入 N 和 T 的函数表达式。

使用 ***weighted quick-union*** 算法（WeightedQuickUnionUF.java from algs4.jar）实现 Percolation 数据类型。进行实验表明当 N 加倍时对运行时间的影响；使用近似表示法，给出在计算机上的总时间，它是输入 N 和 T 的函数表达式。

注：这个问题的实验由 Bob Sedgewick 和 Kevin Wayne 设计开发（Copyright © 2008）。更多信息可参考 http://algs4.cs.princeton.edu/home/。

（二）几种排序算法的实验性能比较

实现插入排序（Insertion Sort，IS），自顶向下归并排序（Top-down Mergesort，TDM），自底向上归并排序（Bottom-up Mergesort，BUM），随机快速排序（Random Quicksort，RQ），Dijkstra 3-路划分快速排序（Quicksort with Dijkstra 3-way Partition，QD3P）。在你的计算机上针对**不同输入规模数据**进行实验，对比上述排序算法的时间及空间占用性能。要求对于每次输入运行 10 次，记录每次时间/空间占用，取平均值。

Comparison of running time of sorting algorithms (in Micro Seconds)

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS											
TDM											
BUM											
RQ											
QD3P											

Comparison of space usage of sorting algorithms (in Kilo Bytes)

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
IS											
TDM											
BUM											
RQ											
QD3P											

回答以下问题：

1. Which sort worked best on data in constant or increasing order (i.e., already sorted data)? Why do you think this sort worked best?

2. Did the same sort do well on the case of mostly sorted data? Why or why not?

3. In general, did the ordering of the incoming data affect the performance of the sorting algorithms? Please answer this question by referencing specific data from your table to support your answer.

4. Which sort did best on the shorter (i.e., n = 1,000) data sets? Did the same one do better on the longer (i.e., n = 10,000) data sets? Why or why not? Please use specific data from your table to support your answer.

5. In general, which sort did better? Give a hypothesis as to why the difference in performance exists.

6. Are there results in your table that seem to be inconsistent? (e.g., If I get run times for a sort that look like this {1.3, 1.5, 1.6, 7.0, 1.2, 1.6, 1.4, 1.8, 2.0, 1.5] the 7.0 entry is not consistent with the rest). Why do you think this happened?

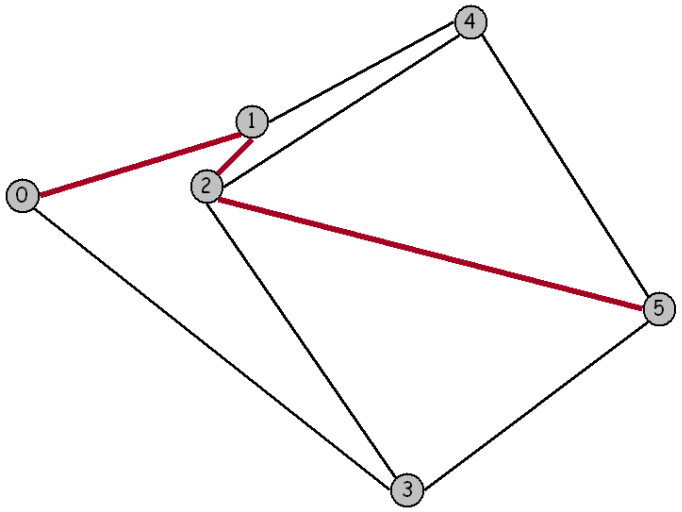
（三）地图路由（Map Routing）

实现经典的 Dijkstra 最短路径算法，并对其进行优化。 这种算法广泛应用于地理信息系统（GIS），包括 MapQuest 和基于 GPS 的汽车导航系统。

地图。 本次实验对象是图 *maps* 或 *graphs*，其中顶点为平面上的点，这些点由权值为欧氏距离的边相连成图。 可将顶点视为城市，将边视为相连的道路。 为了在文件中表示地图，我们列出了顶点数和边数，然后列出顶点（索引后跟其 *x* 和 *y* 坐标），然后列出边（顶点对），最后列出源点和汇点。 例如，如下左图信息表示右图：

```
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 算法。Dijkstra 算法是最短路径问题的经典解决方案。教科书 4.4 节描述了该算法。基本思路不难理解。 对于图中的每个顶点，我们维护从源点到该顶点的最短已知的路径长度，并且将这些长度保持在优先队列（*priority queue*, *PQ*）中。 初始时，我们把所有的顶点放在这个队列中，并设置高优先级，然后将源点的优先级设为 0.0。 算法通过从 *PQ* 中取出最低优先级的顶点，然后检查可从该顶点经由一条边可达的所有顶点，以查看这条边是否提供了从源点到那个顶点较之之前已知的最短路径的更短路径。 如果是这样，它会降低优先级来反映这种新的信息。

这里给出了 Dijkstra 算法计算从 0 到 5 的最短路径 0-1-2-5 的详细过程。

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)

该方法计算最短路径的长度。 为了记录路径，我们还保持每个顶点的源点到该顶点最短路径上的前驱。文件 Euclidean Graph.java，Point.java，IndexPQ.java，IntIterator.java 和 Dijkstra.java 提供了针对 map 的 Dijkstra 算法的基本框架实现，你应该以此作为起点。客户端程序 ShortestPath.java 求解一个单源点最短路径问题，并使用图形绘制了结果。客户端程序 Paths.java 求解了许多最短路径问题，并将最短路径打印到标准输出。客户端程序 Distances.java 求解了许多最短路径问题，仅将距离打印到标准输出。

目标。 优化 Dijkstra 算法，使其可以处理给定图的数千条最短路径查询。 一旦你读取图（并可选地预处理），你的程序应该在**亚线性时间内解决最短路径问题**。一种方法是预先计算出所有顶点对的最短路径；然而，你无法承受存储所有这些信息所需的二次空间。你的目标是减少每次最短路径计算所涉及的工作量，而不会占用过多的空间。 建议你选择下面的一些潜在想法来实现，或者你可以开发和实现自己的想法。

想法 1. Dijkstra 算法的朴素实现检查图中的所有 V 个顶点。 减少检查的顶点数量的一种策略是一旦发现目的地的最短路径就停止搜索。 通过这种方法，可以使每个最短路径查询的运行时间与 E' log V'成比例，其中 E'和 V'是 Dijkstra 算法检查的边和顶点数。 然而，这需要一些小心，因为只是重新初始化所有距离为∞就需要与 V 成正比的时间。由于你在不断执行查询，因而只需重新初始化在先前查询中改变的那些值来大大加速查询。

想法 2. 你可以利用问题的欧式几何来进一步减少搜索时间，这在算法书的第 4.4 节描述过。对于一般图，Dijkstra 通过将 d[w]更新为 d[v] + 从 v 到 w 的距离来松弛边 v-w。 对于地图，则将 d[w]更新为 d[v] + 从 v 到 w 的距离 + 从 w 到 d 的欧式距离 – 从 v 到 d 的欧式距离。这种方法称之为 A*算法。这种启发式方法会有性能上的影响，但不会影响正确性。

想法 3. 使用更快的优先队列。 在提供的优先队列中有一些优化空间。 你也可以考虑使用 Sedgewick 程序中的多路堆（Multiway heaps, Section 2.4）。

测试。 美国大陆文件 usa.txt 包含 87,575 个交叉口和 121,961 条道路。 图形非常稀疏 - 平均的度为 2.8。 你的主要目标应该是快速回答这个网络上的顶点对的最短路径查询。 你的算法可能会有不同执行时间，这取决于两个顶点是否在附近或相距较远。 我们提供测试这两种情况的输入文件。 你可以假设所有的 x 和 y 坐标都是 0 到 10,000 之间的整数。

注：这个问题的实验由 Bob Sedgewick 和 Kevin Wayne 设计开发（Copyright © 2004）。更多信息可参考 http://algs4.cs.princeton.edu/。

（四）文本索引

编写一个构建大块文本索引的程序，然后进行快速搜索，来查找某个字符串在该文本中的出现位置。

你的程序应该使用两个文件名作为命令行参数：文本文件（我们称为语料库）和包含查询的文件。 假设这两个文件只包含小写字母、空格和换行符，查询文件中的查询由换行符分隔。 这不是一个限制，因为你可以使用一个过滤器将任何文件转换为此格式。

你的程序应该读取语料库，将其存储为（可能巨大）字符串，并可能为其创建索引，如下所述。 然后它应该逐个读取查询（假设在命令行中的第二个命名文件中，每行有一个查询），并打印出语料库中每个查询在文本文件中首次出现的位置。对于由如下内容构成的

corpus 文件

it was the best of times it was the worst of times it was the age of wisdom it was the age of foolishness it was the epoch of belief it was the epoch of incredulity it was the season of light it was the season of darkness it was the spring of hope it was the winter of despair

以及如下内容构成的 query 文件

wisdom
season
age of foolishness
age of fools

查询结果如下：

18 wisdom
40 season
22 age of foolishness
-- age of fools

解决这个问题有很多种不同的方法，这些方法在实现方便性，空间要求和时间要求方面都有不同的特点。 此任务的一部分是吸收此信息，以帮助你确定使用哪种方法以及如何将其应用于此特定任务。

你可以从本书中基于程序 3.15 的蛮力搜索实现开始。 也就是说，不建立索引：只需搜索每个查询字符串的语料库即可。 如果语料库很小或者查询不多，这个解决方案是很好的。但是，当语料库庞大且查询量大的时候，这种方法太慢了，因此，你需要实现一个更快的解决方案。

一种快速搜索的方法是在语料库（每个字符位置一个指针）上进行指针排序，然后使用折半搜索。 如果你想采用这种方法，可以使用标准 C 库中的 qsort 和 bsearch 函数。 这种方法的主要挑战是完全理解程序 3.17; 开发一个类似的程序来构建指针索引，按照排序顺序访问关键字（如图 3.13 所示）； 并找出调用 bsearch 的必要接口使用索引执行查询。 特别地，对于进行排序和搜索你需要适当的“比较”功能（不同的!）。

另一种可能的方法是从程序 12.10 开始。这段代码基本上提供了一个完整的解决方案，但为了使其正常工作，你必须进行一些小的更改，因为它们在许多细节上与此处指定的问题不同，并且因为缺少各种小的代码。 你可能需要编辑此代码，或从头开始编写自己的代码。再次，必须仔细考虑“比较”功能。

你可以从这个网站（http://corpus.canterbury.ac.nz/descriptions/）上下载 corpus 数据测试你的程序。

鼓励修改你的程序使其能够计算出每次查询串在 corpus 中出现的次数。

注：本题章节是指该书的 C 语言版

执笔人：霍红卫

2019 年 8 月 27 日