实验一——基于蒙特卡罗模拟的渗透问题阈值估计

姓名：王越洋 学号：22009200894

1. 实验背景与目的

渗透问题（Percolation）是一种物理现象，它可以用来描述液体通过多孔介质流动的过程。通过计算机模拟，可以估算随机分布系统中渗透的临界值（渗透阈值）。在本实验中，我们通过使用并查集数据结构和蒙特卡罗模拟来估计二维网格系统的渗透阈值。

实验任务是实现一个 Percolation 类和 PercolationStats 类，利用这两个类来模拟渗透过程，并通过多次实验来估算渗透阈值及其置信区间。

2. 实验内容

实现 Percolation 数据类型，用于模拟一个 N×N 的网格，并使用并查集判断该系统是否渗透。

实现 PercolationStats 类，通过进行多次独立实验估算渗透阈值，并计算其均值、标准差和 95% 置信区间。

使用 QuickFind 和 WeightedQuickUnion 算法分别实现并查集，分析这两种算法在渗透问题中的效率差异。

3. 实验原理

在一个 N×N 的网格中，随机选择格点并将其设置为开放状态，直到系统形成从顶部到底部的连通路径。渗透阈值是指网格系统从不渗透状态转变为渗透状态时，开放格点所占的比例。

为了估计渗透阈值，通过蒙特卡罗模拟进行多次独立实验，记录每次实验中系统渗透时的开放格点比例，并统计多次实验的平均值、标准差和置信区间。

4. 程序设计

4.1 Percolation 类

（1）功能：模拟 N×N 网格系统中的渗透过程，使用并查集（QuickFindUF 或 WeightedQuickUnionUF）来判断系统是否渗透。

（2）主要方法：

Percolation(int N)：创建一个 N×N 的网格，所有格点初始状态为封闭。

void open(int i, int j)：打开指定位置的格点。

boolean isOpen(int i, int j)：检查某个格点是否为开放状态。

boolean isFull(int i, int j)：检查某个格点是否与顶部连通。

boolean percolates()：判断系统是否渗透，即从顶部到底部是否存在连通路径。

4.2 PercolationStats 类

（1）功能：进行多次独立实验，通过调用 Percolation 类的相关方法估计渗

透阈值，并计算统计结果。
（2）主要方法：
PercolationStats(int N, int T)：执行 T 次独立实验，在 N×N 的网格上估算渗透阈值。
double mean()：计算渗透阈值的平均值。
double stddev()：计算渗透阈值的标准差。
double confidenceLo()：计算 95% 置信区间的下界。
double confidenceHi()：计算 95% 置信区间的上界。

5. 代码实现

5.1 Percolation 类（基于 QuickFindUF/ WeightedQuickUnionUF）

（1）代码

```
1. package edu.princeton.cs.algs4;
2. /**
3.  * 该类模拟一个渗透系统，基于 N×N 网格，使
   用 QuickFindUF/WeightedQuickUnionUF 算法判断系统是否渗透。
4.  */
5.
6. public class Percolation {// WeightedQuickUnionUF
7.     private int N;
8.     private boolean[][] grid;
9.     private WeightedQuickUnionUF uf;
10.    private int virtualTop;
11.    private int virtualBottom;
12.
13.    public Percolation(int N) {
14.        if (N <= 0) throw new IllegalArgumentException("N 必须大
   于 0");
15.        this.N = N;
16.        grid = new boolean[N][N];
17.        uf = new WeightedQuickUnionUF(N * N + 2);  // 使用加权的并查集
18.        virtualTop = N * N;
19.        virtualBottom = N * N + 1;
20.    }
21.//public class Percolation { // QuickFindUF
22.//    private int N; // 网格的大小
23.//    private boolean[][] grid; // 网格，记录各格点是否为 open
24.//    private QuickFindUF uf;
25.//    private int virtualTop; // 虚拟顶点，用于快速检查顶行格点是否连接
26.//    private int virtualBottom; // 虚拟底点，用于快速检查底行格点是否连
   接
27.//
28.//    /**
29.//     * 构造函数，创建一个 N×N 网格，所有格点初始化为 blocked
```

```
30.//     * @param N 网格的大小
31.//     */
32.//    public Percolation(int N) {
33.//        if (N <= 0) throw new IllegalArgumentException("N 必须大
   于 0");
34.//        this.N = N;
35.//        grid = new boolean[N][N];
36.//        uf = new QuickFindUF(N * N + 2); // N*N + 2: 加上两个虚拟节
   点
37.//        virtualTop = N * N; // 顶部虚拟节点
38.//        virtualBottom = N * N + 1; // 底部虚拟节点
39.//    }
40.
41.    /**
42.     * 将格点 (i, j) 打开
43.     * @param i 行号
44.     * @param j 列号
45.     */
46.    public void open(int i, int j) {
47.        validate(i, j);
48.        if (!isOpen(i, j)) {
49.            grid[i - 1][j - 1] = true;
50.            int currentSite = xyTo1D(i, j);
51.
52.            // 如果是第一行，连接到虚拟顶点
53.            if (i == 1) uf.union(currentSite, virtualTop);
54.
55.            // 如果是最后一行，连接到虚拟底点
56.            if (i == N) uf.union(currentSite, virtualBottom);
57.
58.            // 连接周围的 open 格点
59.            connectAdjacent(i, j);
60.        }
61.    }
62.
63.    /**
64.     * 检查格点 (i, j) 是否是 open
65.     * @param i 行号
66.     * @param j 列号
67.     * @return true 如果该格点是 open
68.     */
69.    public boolean isOpen(int i, int j) {
70.        validate(i, j);
71.        return grid[i - 1][j - 1];
```

```
72.     }
73.
74.     /**
75.      * 检查格点 (i, j) 是否与顶行连接，即是否为 full
76.      * @param i 行号
77.      * @param j 列号
78.      * @return true 如果该格点是 full
79.      */
80.     public boolean isFull(int i, int j) {
81.         validate(i, j);
82.         int currentSite = xyTo1D(i, j);
83.         return uf.find(currentSite) == uf.find(virtualTop);
84.     }
85.
86.     /**
87.      * 判断系统是否渗透，即顶行与底行是否连接
88.      * @return true 如果系统渗透
89.      */
90.     public boolean percolates() {
91.         return uf.find(virtualTop) == uf.find(virtualBottom);
92.     }
93.
94.     // 将二维坐标 (i, j) 转换为一维
95.     private int xyTo1D(int i, int j) {
96.         return (i - 1) * N + (j - 1);
97.     }
98.
99.     // 连接与 (i, j) 相邻的 open 格点
100.        private void connectAdjacent(int i, int j) {
101.            int currentSite = xyTo1D(i, j);
102.
103.            if (i > 1 && isOpen(i - 1, j)) uf.union(currentSite, xyT
    o1D(i - 1, j)); // 上
104.            if (i < N && isOpen(i + 1, j)) uf.union(currentSite, xyT
    o1D(i + 1, j)); // 下
105.            if (j > 1 && isOpen(i, j - 1)) uf.union(currentSite, xyT
    o1D(i, j - 1)); // 左
106.            if (j < N && isOpen(i, j + 1)) uf.union(currentSite, xyT
    o1D(i, j + 1)); // 右
107.        }
108.
109.        // 验证 (i, j) 是否在有效范围内
110.        private void validate(int i, int j) {
111.            if (i < 1 || i > N || j < 1 || j > N) {
```

```
112.                 throw new IllegalArgumentException("坐
    标 (" + i + ", " + j + ") 超出范围");
113.             }
114.         }
115.     }
```

（2）分析

Percolation 类负责模拟一个 N × N 的网格，并通过使用并查集（Union-Find）数据结构来判断系统是否渗透。它模拟了渗透问题中的格点状态变化，并为动态连通性提供了支持。以下对 Percolation 类的关键部分进行分析：

5.1.1 open(int i, int j) 方法

将网格中的某个格点设置为开放状态。每当一个格点被打开时，它需要与其相邻的开放格点进行连接，以保证连通性。在调用该方法时，网格的状态会更新，并将该格点与相邻的开放格点合并到同一个连通集。

关键点：

二维坐标转换为一维索引：通过 xyTo1D(i, j) 方法将 (i, j) 转换为一维索引。这是因为并查集的数据结构是用一维数组表示的。

连接虚拟顶点和虚拟底点：为了简化渗透判断，open() 方法会将顶行的开放格点与虚拟顶点相连，底行的开放格点与虚拟底点相连。这确保了我们在判断系统是否渗透时，只需要检查虚拟顶点和虚拟底点是否连通即可，而不必遍历整个网格。

连接相邻的开放格点：每当一个格点被打开时，它会与上下左右相邻的开放格点进行合并。这通过 connectAdjacent(i, j) 方法来实现。相邻的开放格点被合并到同一个连通集后，可以确保系统的渗透连通性。

5.1.2 isOpen(int i, int j) 方法

该方法用于判断网格中某个格点是否处于开放状态。实现方式简单明了，通过检查 grid[i - 1][j - 1] 数组中的布尔值来判断格点是否已经被打开。

关键点：

输入的合法性检查：该方法首先通过 validate(i, j) 来检查输入的行列索引是否在有效范围内，以防止数组越界。

返回值：该方法直接返回 grid 数组中相应位置的值。true 表示格点开放，false 表示格点仍处于封闭状态。

5.1.3 xyTo1D(int i, int j) 方法

将二维坐标 (i,j) 转换为一维数组中的索引。这种转换是为了适应并查集的实现，因为并查集通常是通过一维数组来管理不同的集合。

关键点：

通过公式 index = (i - 1) * N + (j - 1)，二维网格中的每个格点都映射到并查集中的一维数组索引。这个公式的含义是：前面 i-1 行有 (i-1) * N 个格点，再加上第 i 行的第 j 个格点。

5.1.4 percolates() 方法

判断系统是否渗透。渗透的定义是：如果从网格的顶部（第一行）到底部（最后一行）存在一条连通路径，则系统渗透。通过引入虚拟顶点和虚拟底点，我们可以简化这一判断，只需检查虚拟顶点和虚拟底点是否连通即可。

关键点：

效率高：无需遍历整个网格，只需检查 find(virtualTop) 和 find(virtualBottom) 是否相等，即可判断系统是否渗透。这样能够大幅提高渗透判断的效率。

5.1.5 两个虚拟节点的作用

虚拟顶点：连接网格第一行所有的格点，便于快速判断是否有路径从顶部延伸到该格点。

虚拟底点：连接网格最后一行所有的格点，用于判断从底部是否存在到某个格点的连通路径。

简化渗透判断：通过判断虚拟顶点与虚拟底点的连通性，简化了整个系统的渗透状态的判断过程。如果没有虚拟节点，判断系统是否渗透需要遍历最后一行所有的格点。

5.2 PercolationStats 类

（1）代码

```
1. package edu.princeton.cs.algs4;
2.
3. import edu.princeton.cs.algs4.StdRandom;
4. import edu.princeton.cs.algs4.StdStats;
5.
6. public class PercolationStats {
7.     private double[] thresholds; // 记录每次实验的渗透阈值
8.     private int T; // 实验次数
9.
10.    /**
11.     * 构造函数，进行 T 次独立实验
12.     * @param N 网格大小
13.     * @param T 实验次数
14.     */
15.    public PercolationStats(int N, int T) {
16.        if (N <= 0 || T <= 0) throw new IllegalArgumentException("N
   和 T 必须大于 0");
17.        this.T = T;
18.        thresholds = new double[T];
19.
20.        // 记录开始时间
```

```
21.         long startTime = System.nanoTime();
22.
23.         // 进行 T 次实验
24.         for (int t = 0; t < T; t++) {
25.             Percolation percolation = new Percolation(N);
26.             int openSites = 0;
27.
28.             // 打开格点直到系统渗透
29.             while (!percolation.percolates()) {
30.                 int i = StdRandom.uniform(N) + 1; // 生成随机行
31.                 int j = StdRandom.uniform(N) + 1; // 生成随机列
32.
33.                 if (!percolation.isOpen(i, j)) {
34.                     percolation.open(i, j);
35.                     openSites++;
36.                 }
37.             }
38.             thresholds[t] = (double) openSites / (N * N); // 计算渗透
    阈值
39.         }
40.
41.         // 记录结束时间
42.         long endTime = System.nanoTime();
43.
44.         // 计算总运行时间并输出
45.         long elapsedTime = endTime - startTime;
46.         double elapsedTimeInSeconds = elapsedTime / 1e9; // 转换为秒
47.         System.out.println("总运行时间
    （秒）：  " + elapsedTimeInSeconds);
48.     }
49.
50.     /**
51.      * 返回渗透阈值的均值
52.      * @return 均值
53.      */
54.     public double mean() {
55.         return StdStats.mean(thresholds);
56.     }
57.
58.     /**
59.      * 返回渗透阈值的标准差
60.      * @return 标准差
61.      */
62.     public double stddev() {
```

```
63.         return StdStats.stddev(thresholds);
64.     }
65.
66.     /**
67.      * 返回 95% 置信区间的下界
68.      * @return 下界
69.      */
70.     public double confidenceLo() {
71.         return mean() - (1.96 * stddev() / Math.sqrt(T));
72.     }
73.
74.     /**
75.      * 返回 95% 置信区间的上界
76.      * @return 上界
77.      */
78.     public double confidenceHi() {
79.         return mean() + (1.96 * stddev() / Math.sqrt(T));
80.     }
81.
82.     /**
83.      * 主函数，用于从命令行接收参数并执行实验
84.      * @param args 输入参数
85.      */
86.     public static void main(String[] args) {
87.         if (args.length < 2) {
88.             System.out.println("用法: java PercolationStats <网格大
    小 N> <实验次数 T>");
89.             return;
90.         }
91.
92.         int N = Integer.parseInt(args[0]); // 网格大小
93.         int T = Integer.parseInt(args[1]); // 实验次数
94.
95.         PercolationStats stats = new PercolationStats(N, T);
96.
97.         System.out.println("渗透阈值均值 = " + stats.mean());
98.         System.out.println("渗透阈值标准差 = " + stats.stddev());
99.         System.out.println("95% 置信区
    间 = [" + stats.confidenceLo() + ", " + stats.confidenceHi() + "]");
100.        }
101.    }
```

（2）分析

PercolationStats 类负责使用 Percolation 模拟渗透问题，并通过蒙特卡罗实验来估计渗透阈值。

5.2.1 构造函数 PercolationStats(int N, int T)

进行 T 次独立实验。在每次实验中，它使用 Percolation 模拟 N×N 网格，并记录每次实验中系统首次渗透时的开放格点比例。

关键点：

初始化参数检查：如果输入的网格大小 N 或实验次数 T 小于等于 0，程序会抛出异常。

实验过程：在每次实验中，不断随机开放格点，直到系统渗透。然后将开放格点数与总格点数的比例记录为一次实验的渗透阈值。

5.2.2 mean() 方法

计算渗透阈值的均值。实验多次后，系统的渗透阈值不会每次相同，因此需要计算所有实验中渗透阈值的平均值，作为对真实渗透阈值的估计。

公式：mean() 调用了 StdStats.mean(thresholds)，其中 thresholds 是所有实验的渗透阈值数组。均值的计算公式为：

其中，T 是实验次数，threshold_i 是第 i 次实验的渗透阈值。

5.2.3 stddev() 方法

计算渗透阈值的标准差，表示实验结果的波动性。标准差越大，说明实验结果的波动越大。

公式：stddev() 调用了 StdStats.stddev(thresholds)。标准差的计算公式为：

$$\text{mean} = \frac{1}{T}\sum_{i=1}^{T} \text{threshold}_i$$

5.2.4 confidenceLo() 和 confidenceHi() 方法

计算 95% 置信区间的下界和上界，即估计渗透阈值的波动范围。在 95% 的置信水平下，真实的渗透阈值有 95% 的概率落在这个区间内。

公式：

$$\text{stddev} = \sqrt{\frac{1}{T-1}\sum_{i=1}^{T} (\text{threshold}_i - \text{mean})^2}$$

下界：confidenceLo() = mean() - (1.96 * stddev() / Math.sqrt(T))

上界：confidenceHi() = mean() + (1.96 * stddev() / Math.sqrt(T))

其中 1.96 是标准正态分布中 95% 置信区间的对应系数，T 是实验次数，stddev() 是标准差。

关键点：

置信区间的计算反映了实验结果的波动情况。如果标准差较小，则置信区间会较窄，表明实验结果较为稳定。

通过多次实验和计算均值、标准差，可以对渗透阈值的准确度有更好的把

握。

6.结果分析

设置 150*150 大小的方格，进行 100 次试验。对于不同算法结果如下：

6.1 实验 1：WeightedQuickUnionUF（高效率）

```
总运行时间（秒）：  0.1556943
渗透阈值均值 = 0.5933391111111109
渗透阈值标准差 = 0.011506865659310063
95% 置信区间 = [0.5910837654418861, 0.59559445678033357]
```

运行时间非常短，仅为 0.1556943 秒，主要由于 WeightedQuickUnionUF 通过加权合并策略，使得树的高度不会过度增长，这保证了查找（find）和合并（union）操作的效率。随着网格规模增大，算法的时间复杂度仍保持在 O(log N)。

因此表现了加权合并策略在大规模网格中的高效性。

6.1 实验 2：QuickFindUF（低效率）

```
总运行时间（秒）：  4.6963348
渗透阈值均值 = 0.5924306666666667
渗透阈值标准差 = 0.013076989127623012
95% 置信区间 = [0.5898675767976526, 0.5949937565356808]
```

QuickFindUF 的每次 union 操作都需要遍历整个数组，导致在处理大规模网格时效率非常低，尤其是当系统中有大量的连通操作时，时间复杂度为 O(N)。

在 150 × 150 的网格上，QuickFindUF 的算法随着网格规模的增加而变得更加低效，从而大幅增加了运行时间。

6.2 渗透阈值均值的对比

实验 1 的渗透阈值均值为 0.593339，而实验 2 为 0.592431。两者的差异很小，基本都接近理论值 0.592746。这表明，两种算法在估算渗透阈值时，尽管效率不同，但得到的结果是一致且合理的。

无论使用哪种并查集算法，最终的渗透阈值均值应接近理论值。这表明即使 QuickFindUF 算法效率较低，它仍能正确估计出渗透阈值。

6.3 渗透阈值标准差的对比

实验 1 的标准差为 0.011506，而实验 2 为 0.013077。

标准差反映了实验结果的波动性。较小的标准差意味着实验结果更加集中，波动较小，而较大的标准差表示实验结果有更多的波动。

虽然实验 1 的标准差略小于实验 2，但差异不大。这说明使用 Weighted QuickUnionUF 算法的结果稍微更稳定一些，但总体上两者的波动性都在合理范围内。

6.4 95% 置信区间的对比

实验 1 的置信区间为 [0.5910837654418861, 0.5955944567803357]。

实验 2 的置信区间为 [0.5898675767976526, 0.5949937565356808]。

两组置信区间的范围有轻微差异，但总体上它们都包含理论值 0.592746。实验 1 的置信区间略微靠上，而实验 2 的置信区间范围略大。置信区间的差异与标准差一致：标准差越大，置信区间的范围越广。

说明两组实验都能够给出合理的渗透阈值估计，且都提供了对实际渗透阈值的较好估计。

6.5 总结

运行时间的差异表明 WeightedQuickUnionUF 在大规模网格上的性能远优于 QuickFindUF，尤其是在需要频繁进行连通判断的大规模渗透问题中。QuickFindUF 由于其 union 操作效率低，在处理较大网格时需要更长的时间。

渗透阈值的均值和标准差在两种算法中非常接近，说明即使是低效的算法 QuickFindUF，它在正确性上仍然能够较好地估算渗透阈值。

95% 置信区间的结果说明两种算法都能够提供合理的渗透阈值估计，并且估计值在多次实验中趋于一致。

7. 实验反思与体会

本次实验通过实现渗透问题并结合蒙特卡罗模拟，我们深入了解了并查集数据结构的作用，并分析了不同算法在渗透问题中的效率表现。实验的主要目的在于估算二维网格系统的渗透阈值，并分析其置信区间。

1. 理论与实践的结合

在理论学习中，渗透问题的核心概念是“渗透阈值”，这在统计物理学中有重要应用。然而，通过实际编程实现 Percolation 类和 PercolationStats 类，我们对这些理论概念有了更加直观的理解。在代码实现过程中，如何通过不断打开网格中的格点、判断系统是否渗透，结合并查集的优化策略，让理论不再抽象，而是通过每次实验的结果进行验证。

2. 并查集算法的深刻认识

本实验中的一个关键任务是选择并查集算法来解决动态连通性问题。通过实现和对比 QuickFindUF 和 WeightedQuickUnionUF，我们能够切实感受到不同算法在效率上的巨大差异。具体体会如下：

QuickFindUF 的局限性：在理论上我们知道 QuickFindUF 的时间复杂度较高，特别是在处理较大规模网格时表现不佳。然而在实际运行中，看到运行时间长达数秒甚至更多时，这种算法的劣势被直观地放大。这让我进一步认识到算法效率的重要性，特别是在处理大规模数据时，低效算法可能会导致不可接受的性能问题。

WeightedQuickUnionUF 的效率优势：加权合并策略有效避免了树的高度无限增长，从而显著提高了查找和合并操作的效率。在实验中，这一点通过极短的运行时间得到了验证。特别是在 N = 150 的网格上，WeightedQuickUnionUF 能够在不到一秒钟内完成实验，而 QuickFindUF 则需要数秒，充分体现了加权优化的优势。

3. 实验中的挑战

在实验中，主要的挑战来自：

（1）坐标转换与数组处理：二维网格的坐标与一维数组的映射在实现中是一个需要特别注意的细节。通过 xyTo1D 函数，准确地将二维坐标转换为一维索引，并使用并查集管理这些索引，使我们能够更高效地进行动态连通性判断。

（2）置信区间的计算：置信区间是统计实验中的重要概念，它反映了实验结果的稳定性。在实验中，计算渗透阈值的均值、标准差和 95% 置信区间时，我们需要使用准确的数学公式，并通过 PercolationStats 类进行多次实验来确保结果的可靠性。对于置信区间的实际意义，通过代码的实现，我进一步理解了如何在统计学中估计参数，并解释结果的波动性。

4. 实验的收获

算法效率的直观体会：通过对比两种不同的并查集算法，深刻体会到算法优化的重要性。即使在相对简单的问题场景中，优化后的算法依然能够大幅提高性能。在渗透问题这样的动态连通性场景下，正确选择并查集算法至关重要。

理论验证：通过蒙特卡罗模拟多次实验后，得出的渗透阈值结果接近理论值 0.592746，这让我们认识到，即使在现实世界中，随机现象的统计特性可以通过大量实验进行逼近和验证。这也是统计物理学和计算模拟中常见的工作方法。

5. 未来的改进与思考

更高效的数据结构：在这次实验中，虽然 WeightedQuickUnionUF 相比 QuickFindUF 有显著的性能提升，但仍然存在进一步优化的空间。通过路径压缩进一步减少树的高度，能够在大型网格上进一步提高效率。

更复杂的应用场景：渗透问题是物理学中的一个经典问题，但在实际应用中，渗透问题的模型可以更加复杂。未来实验可以考虑三维网格的渗透模拟，或者引入更加复杂的动态规则，这些都将是进一步研究的方向。

6. 实验体会

通过这次实验，我不仅巩固了对并查集算法的理解，还对如何通过模拟实验解决现实问题有了新的认识。算法不仅仅是理论知识，它在实际应用中决定了程序的效率和可行性。优化数据结构、设计高效的算法、处理大量的实验数据是本次实验带来的重要收获。