实验一：渗透问题（Percolation）

张俊华 16030199025

一、实验内容

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。

二、实验环境

IntelliJ IDEA 2018.2.5 (Ultimate Edition)

JRE: 1.8.0_152-release-1248-b19 amd64

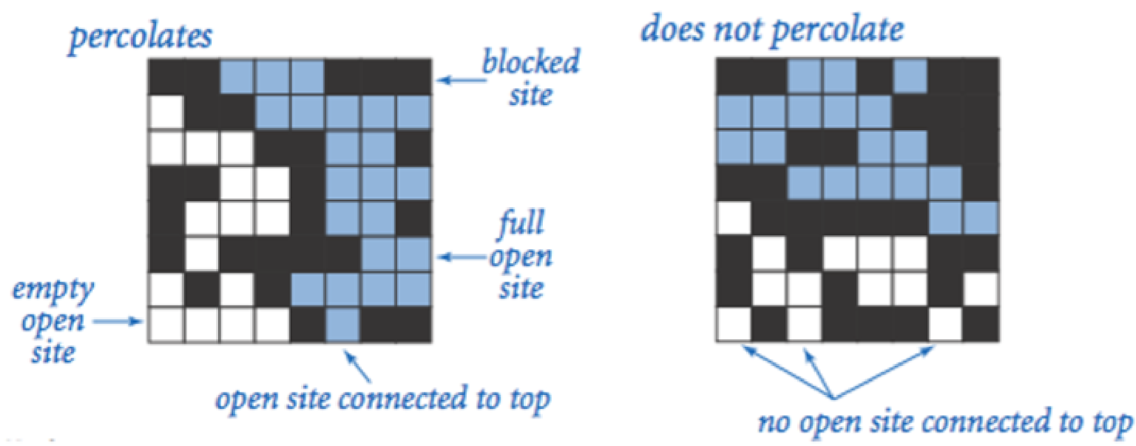
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Windows 10 10.0

三、实验步骤

1. **构建 Percolation 类**

   n*n 个点组成的网格，每个点是 Open 或 Closed 状态。假如最底部的点和最顶端的点连通，就说明这个网格系统是渗透的。 比如图中黑色表示 Closed 状态，白色表示 Open，蓝色表示与顶部连通。所以左图是渗透的，右图不是：



   创建一个 Percolation 类，通过对 N*N 个网格中的点进行操作，来模拟判断渗透情况

```java
public class Percolation {
   public Percolation(int n)                // create n-by-n grid, with all sites blocked
   public    void open(int row, int col)    // open site (row, col) if it is not open already
   public boolean isOpen(int row, int col)  // is site (row, col) open?
   public boolean isFull(int row, int col)  // is site (row, col) full?
   public     int numberOfOpenSites()       // number of open sites
   public boolean percolates()              // does the system percolate?

   public static void main(String[] args)   // test client (optional)
}
```

判断图是否渗透，关键是要判断顶部和底部是否连通。根据所学知识，使用并查集可以快速完成判断。每次打开网格中的点时，就讲该点与其上下左右四个相邻网格中开放的点并入同一集合。可以在顶部和底部创建两个虚拟节点，在初始化时将其分别与顶部和底部的节点并入同一集合，每次只需判断这两个虚拟节点是否在同一集合里，即可判断图是否渗透

Percolation 类实现的代码见附录

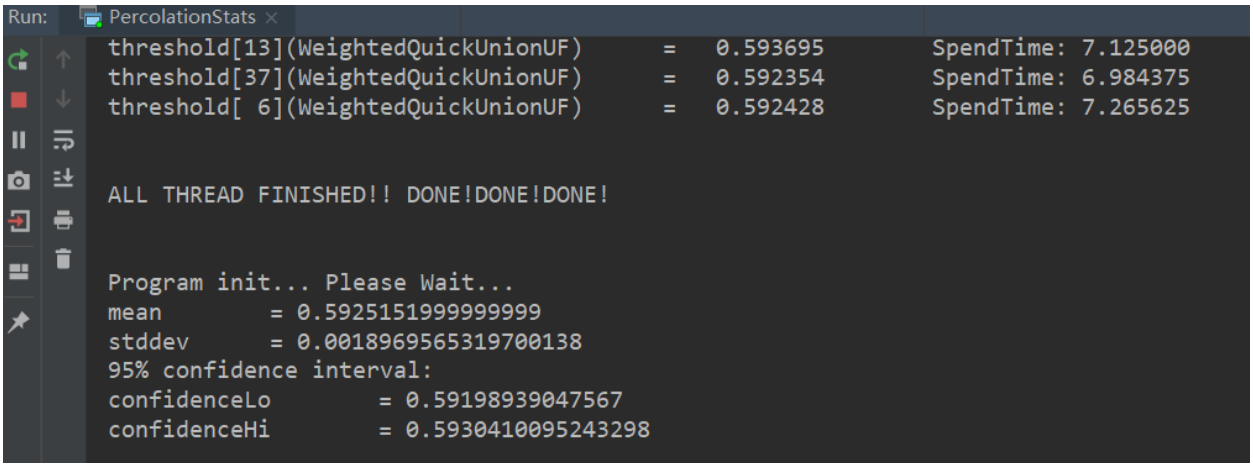
2. **蒙特卡洛模拟**

   本实验通过蒙特卡洛算法，估算渗透阈值，具体做法为：

   - 初始化 n*n 全为 Blocked 的网格系统
   - 随机 Open 一个点，重复执行，直到整个系统变成渗透的为止
   - 上述过程重复 T 次，计算平均值、标准差、96% 置信区间

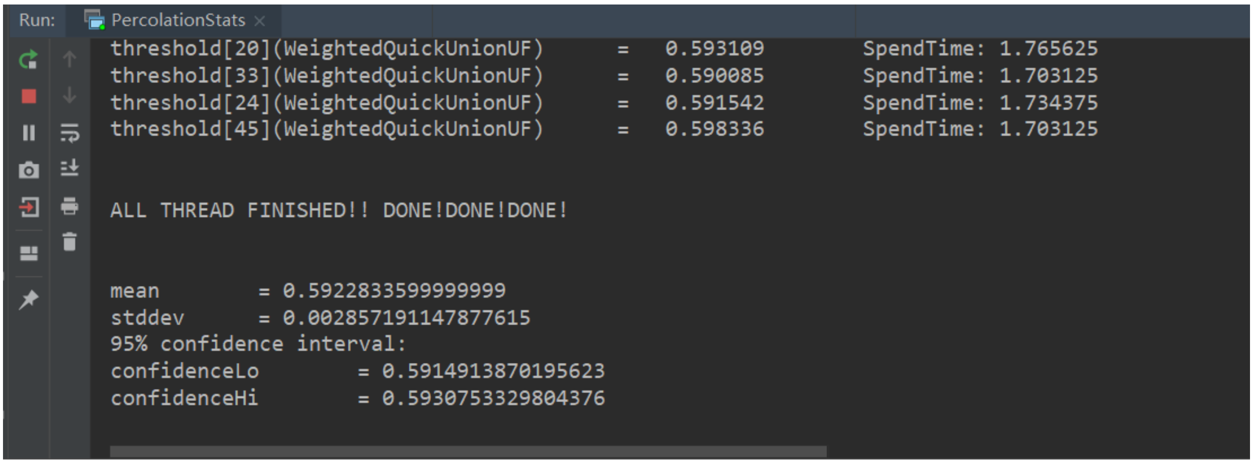
   为了提高计算效率，这里引入 Java 的多线程技术，采用 WeightedQuickUnion 并查集，对较大规模的网格，进行多次渗透测试，最终找到其 95% 置信区间

   对大小为 2000 的网格进行 50 次模拟，结果如下

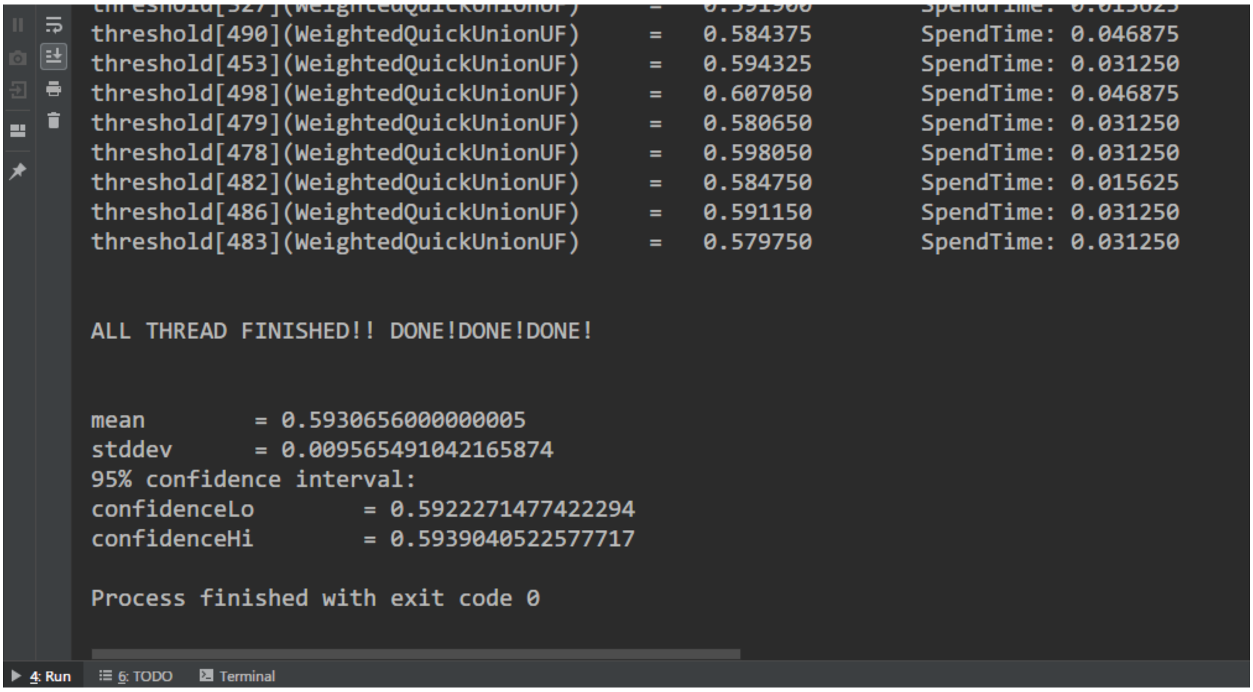


```
Run:  PercolationStats
threshold[13](WeightedQuickUnionUF)    =   0.593695       SpendTime: 7.125000
threshold[37](WeightedQuickUnionUF)    =   0.592354       SpendTime: 6.984375
threshold[ 6](WeightedQuickUnionUF)    =   0.592428       SpendTime: 7.265625


ALL THREAD FINISHED!! DONE!DONE!DONE!


Program init... Please Wait...
mean        = 0.5925151999999999
stddev      = 0.0018969565319700138
95% confidence interval:
confidenceLo        = 0.59198939047567
confidenceHi        = 0.5930410095243298
```

   对大小为 1000 的网格进行 50 次模拟，结果如下



对大小为 200 的网格进行 500 次模拟，结果如下



通过多次试验发现，随着模拟规模的增大，渗透阈值方差趋于稳定，95%置信区间稳定在 0.591~0.594，最终渗透阈值稳定在 0.5925 附近。并且，网格规模对渗透阈值无明显影响

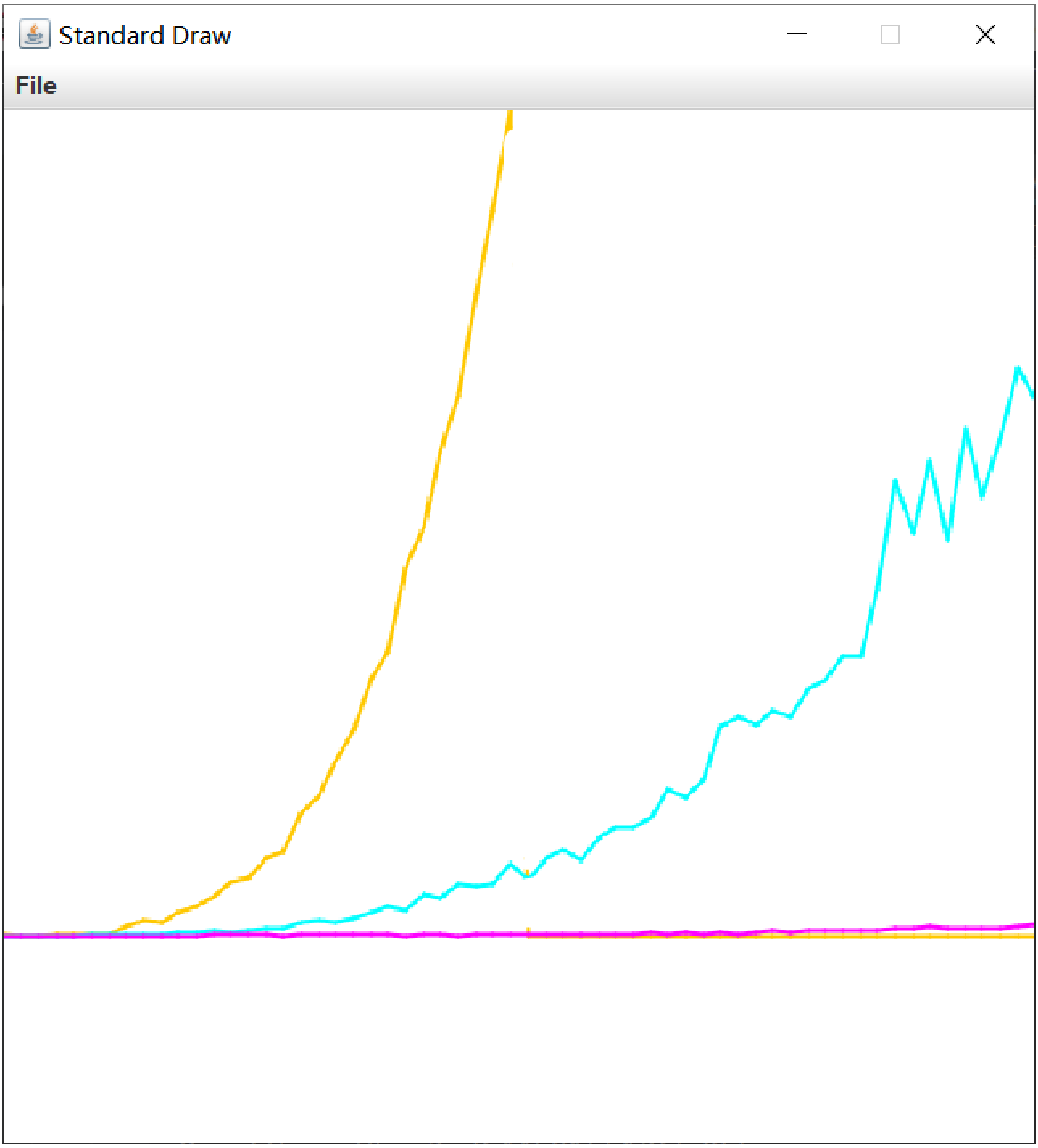
3. **不同的并查集算法性能比较**

为了研究不同的并查集算法性能，本实验重新构建了 UF 类，新的 UF 类，可以在实例化对象时，指定选用的并查集算法。在这里，对 QuickFindUF、QuickUnionUF 以及 WeightedQuickUnionUF 三种并查集算法进行比较分析，UF 类代码如下：

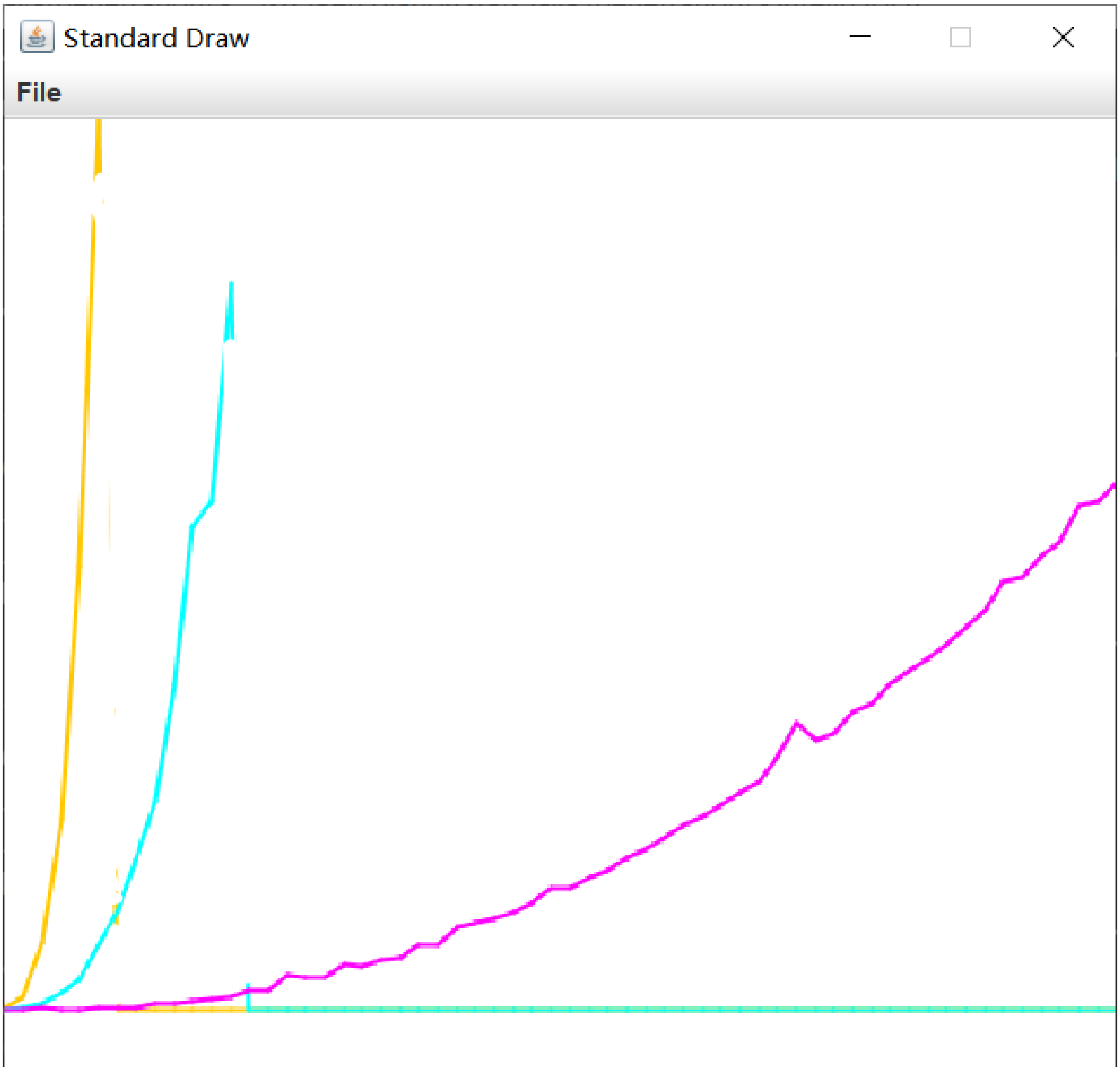
Java 程序执行时，通过传入参数，控制最大网格规模，以最大网格规模为基础，由小到大，等间距取不同大小的网格，使用三种算法模拟渗透问题，进行算法性能分析

```
------ RUN: 19/30 ------
------ size: 316   ------
Thread:19 Use:QuickFindUF added into ThreadList
threshold[54](          QuickFindUF)      =   0.603970        SpendTime: 1.593750
Thread:19 Use:QuickUnionUF added into ThreadList
threshold[55](         QuickUnionUF)      =   0.611851        SpendTime: 0.203125
Thread:19 Use:WeightedQuickUnionUF added into ThreadList
threshold[56](WeightedQuickUnionUF)      =   0.598352        SpendTime: 0.015625

------ RUN: 20/30 ------
------ size: 333   ------
Thread:20 Use:QuickFindUF added into ThreadList
threshold[57](          QuickFindUF)      =   0.602855        SpendTime: 1.953125
Thread:20 Use:QuickUnionUF added into ThreadList
threshold[58](         QuickUnionUF)      =   0.599969        SpendTime: 0.187500
Thread:20 Use:WeightedQuickUnionUF added into ThreadList
threshold[59](WeightedQuickUnionUF)      =   0.594540        SpendTime: 0.000000

------ RUN: 21/30 ------
------ size: 350   ------
Thread:21 Use:QuickFindUF added into ThreadList
threshold[60](          QuickFindUF)      =   0.580318        SpendTime: 2.046875
Thread:21 Use:QuickUnionUF added into ThreadList
threshold[61](         QuickUnionUF)      =   0.596588        SpendTime: 0.203125
Thread:21 Use:WeightedQuickUnionUF added into ThreadList
threshold[62](WeightedQuickUnionUF)      =   0.587143        SpendTime: 0.015625

------ RUN: 22/30 ------
------ size: 366   ------
Thread:22 Use:QuickFindUF added into ThreadList
threshold[63](          QuickFindUF)      =   0.582512        SpendTime: 2.593750
Thread:22 Use:QuickUnionUF added into ThreadList
```

网格大小从 0~1000 的运行时间图:



网格大小从 0~5000 的运行时间图

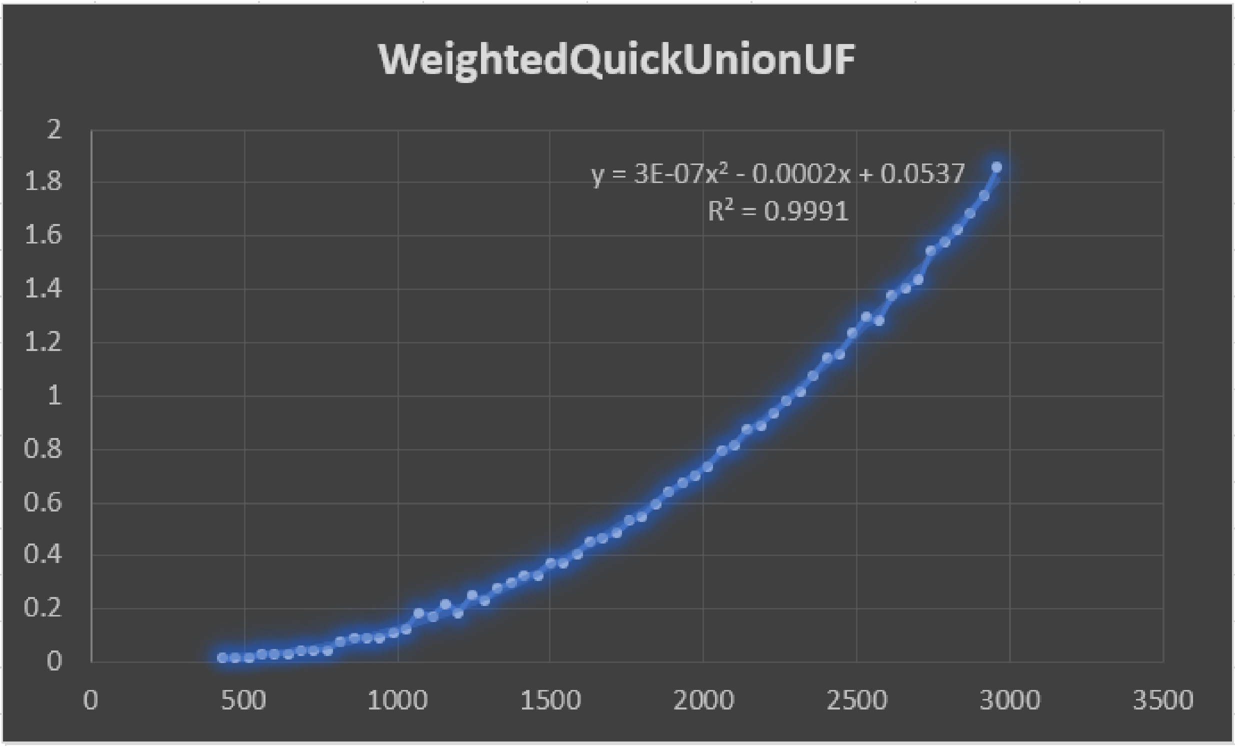


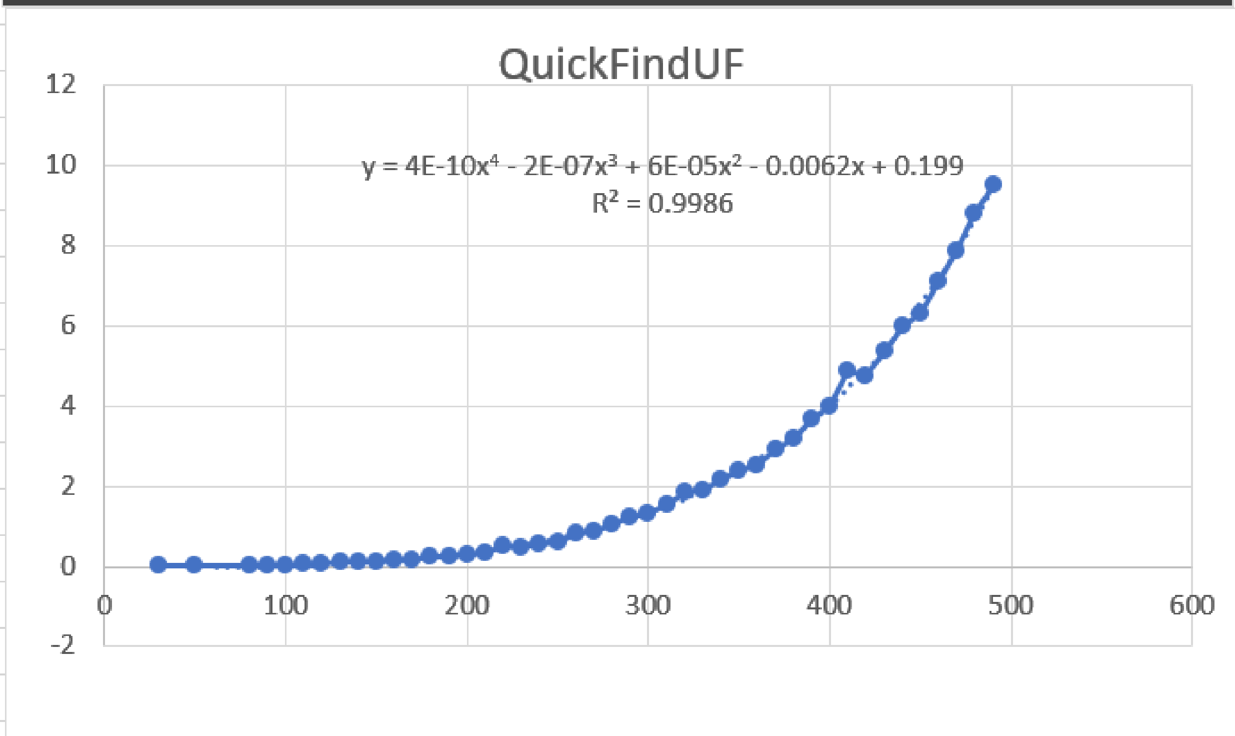
其中，橙色为QuickFindUF的运行时间，蓝色为QuickUnionUF 云香精时间，玫红色为 WeightedQuickUnionUF 运行时间

由图可见，QuickFindUF 算法的运算时间，随问题规模的增长速度最大，在 500*500 规模附近，QuickFindUF 的单次运行时间已经达到了 10S，QuickUnionUF 次之，10S可以模拟 1100* 1100 大小以内的渗透问题，

WeightedQuickUnionUF 表现最为优异，截止到 5000*5000， WeightedQuickUnionUF 算法的单次运行耗费时间仍不足 10S，仍在可接受的时间范围内。

随后，将算法运行时间统计数据导出之后进行回归分析





根据线性拟合结果可知，使用 QuickFindUF 算法模拟渗透问题，在本计算机中 T(时间) 与 N(渗透网格的边长)的四次方正比，拟合得 T 与 N 的函数关系式为：

$$y = 4 * 10^{-10}x^4 - 2 * 10^{-07}x^3 + 6 * 10^{-5}x^2 - 0.0062x + 0.199$$

而使用 QuickFindUF 算法模拟渗透问题，拟合得 T 与 N 的函数关系式为：

$$y = 3 * 10^{-7}x^2 - 0.0002x + 0.0537$$

附：实验源码

```
import edu.princeton.cs.algs4.*;
```

```java


public class Percolation {
    private UF uf;
    private int N;
    private boolean isopen[][];
    private boolean isfulled[][];
    boolean isVisited[][];
    public Painter painter;
    private boolean paint;

    private int getUFId(int x, int y){
        return (y*N + x + 1);
    }

    /**
     * 初始化大小为 N 的可渗透区域
     * @param N 渗透区域大小
     */
    public Percolation(int N, String ufType,boolean paint){
        this.N = N;
        this.paint = paint;
        uf = new UF(N*N+2,ufType);

        for (int i = 0; i< N;i++){
            uf.union(0, getUFId(i,0));
        }
        for (int i = 0; i< N;i++){
            uf.union(N*N+1, getUFId(i,N-1));
        }

        isopen = new boolean[N][N];

        for (int i = 0;i <N;i++){
            for (int j = 0; j<N;j++){
                isopen[i][j] = false;
            }
        }

        if (paint){
            isfulled = new boolean[N][N];
            for (int i = 0;i <N;i++){
                for (int j = 0; j<N;j++){
                    isfulled[i][j] = false;
                }
            }
            painter = new Painter(N);
            isVisited = new boolean[N][N];
        }

    }

    public Percolation(int N){
```

```java
        this(N,"",false);
    }

    /**
     * 开放 x y 处的点
     * @param x 点的横坐标
     * @param y 点的纵坐标
     */
    public void open(int x, int y){
        if (isopen(x, y)) return;
        isopen[x][y] = true;
        if (paint) painter.printOpen(x,y);
        int dx[] = {0,0,-1,1};
        int dy[] = {1,-1,0,0};
        for (int i = 0; i<4;i++){
            if (isopen(x+dx[i],y+dy[i])){
                uf.union(getUFId(x+dx[i],y+dy[i]), getUFId(x,y));
            }
        }
        if (paint) if (isFull(x,y))    bfsIsFull(x,y);
    }

    /**
     * 判断 x y 处的点是否开放
     * @param x 点的横坐标
     * @param y 点的纵坐标
     */
    public boolean isopen(int x, int y){
        if (x<0 || x>=N || y<0 || y>=N){
            return false;
        }else {
            return isopen[x][y];
        }
    }

    /**
     * 判断 x y 点处是否已经注入水（绘图用）
     * @param x 点的横坐标
     * @param y 点的纵坐标
     */
    public boolean isFull(int x, int y){
        return isopen(x, y) && uf.connected(0, getUFId(x, y));
    }

    /**
     * 判断全图是否渗透
     *
     * @return true 渗透
     */
    public boolean percolates(){
        return uf.connected(0,N*N+1);
    }

```

```java
    /**
     * 计算本次渗透阈值
     *
     * @return double 阈值
     */
    public double threshold(){
        return (double)this.openSize()/(N*N);
    }

    public int openSize(){
        int num = 0;
        for(int i = 0;i<N;i++){
            for (int j = 0; j<N;j++){
                if(isopen(i,j)){
                    num++;
                }
            }
        }
        return num;
    }

    private void bfsIsFull(int x, int y){
        if (!isFull(x,y))   return;
        for (int i = 0; i<N;i++){
            for (int j = 0; j<N;j++){
                isVisited[i][j] = false;
            }
        }

        Queue<Node> queue = new Queue<>();
        queue.enqueue(new Node(x,y));
        while (!queue.isEmpty()){
            Node n = queue.dequeue();
            painter.printFull(n.x,n.y);
            isfulled[n.x][n.y] = true;
            int dx[] = {0,0,-1,1};
            int dy[] = {1,-1,0,0};
            for (int i = 0; i<4;i++){
                int nx = n.x +dx[i];
                int ny = n.y +dy[i];
                if (isopen(nx,ny) && ! isVisited[nx][ny] && (!isfulled[nx][ny])){
                    isVisited[nx][ny] = true;
                    queue.enqueue(new Node(nx,ny));
                }
            }
        }
    }

    private class Node{
        int x, y;
        public Node(int x, int y){
            this.x = x;
            this.y = y;
```

```
            }
        }
}
```