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

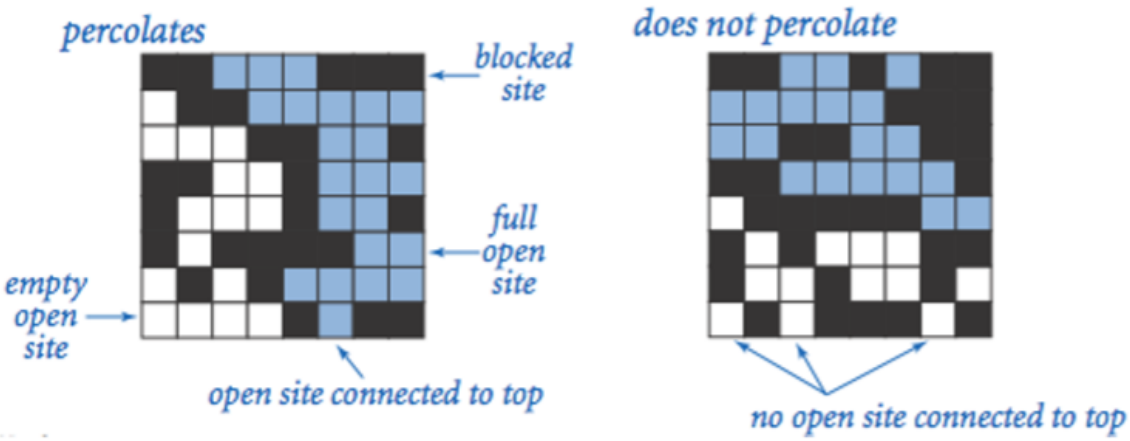
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

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

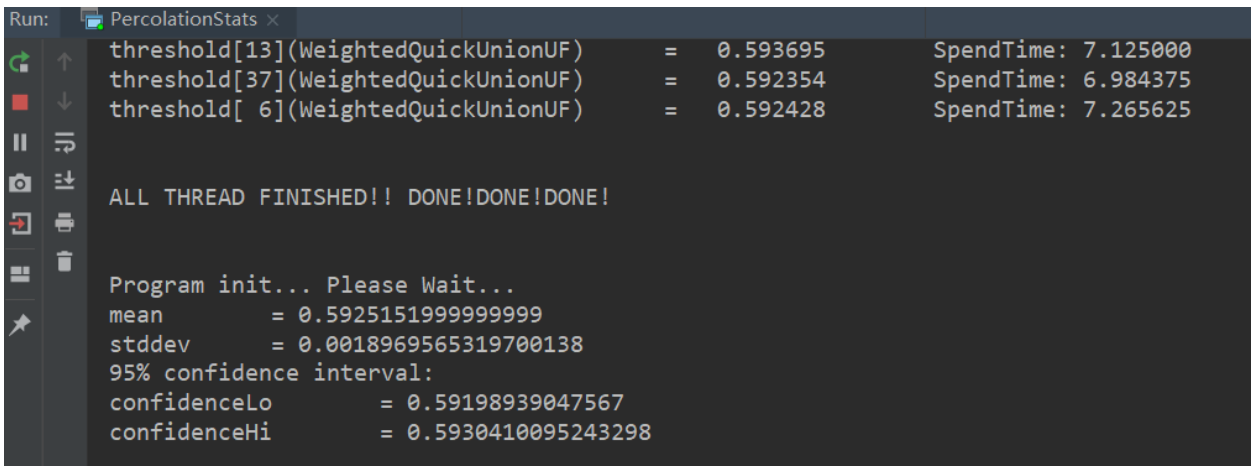
2. **蒙特卡洛模拟**

   本实验通过蒙特卡洛算法，估算渗透阈值，具体做法为：

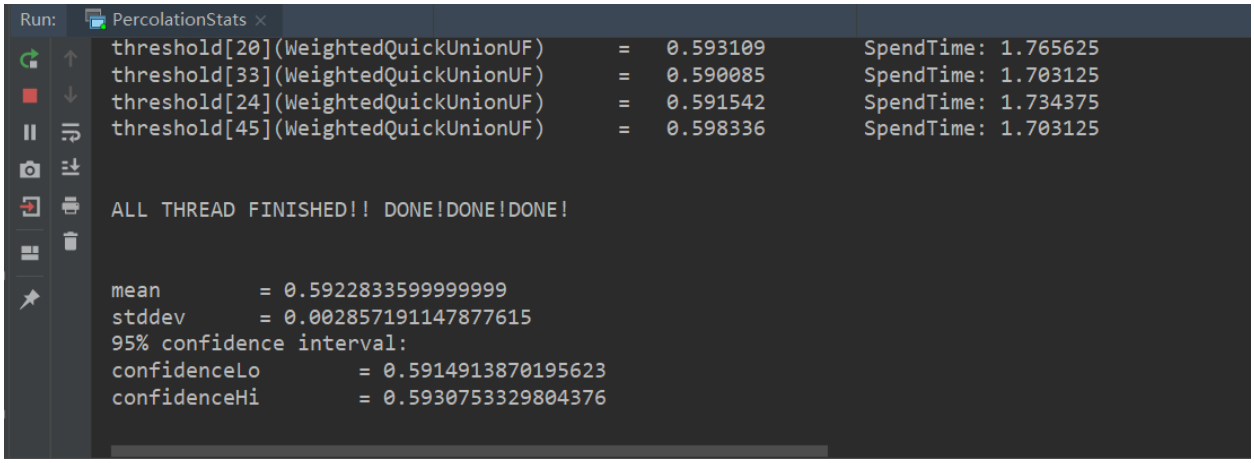
   - 初始化 n*n 全为 Blocked 的网格系统
   - 随机 Open 一个点，重复执行，直到整个系统变成渗透的为止
   - 上述过程重复 T 次，计算平均值、标准差、96% 置信区间

   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

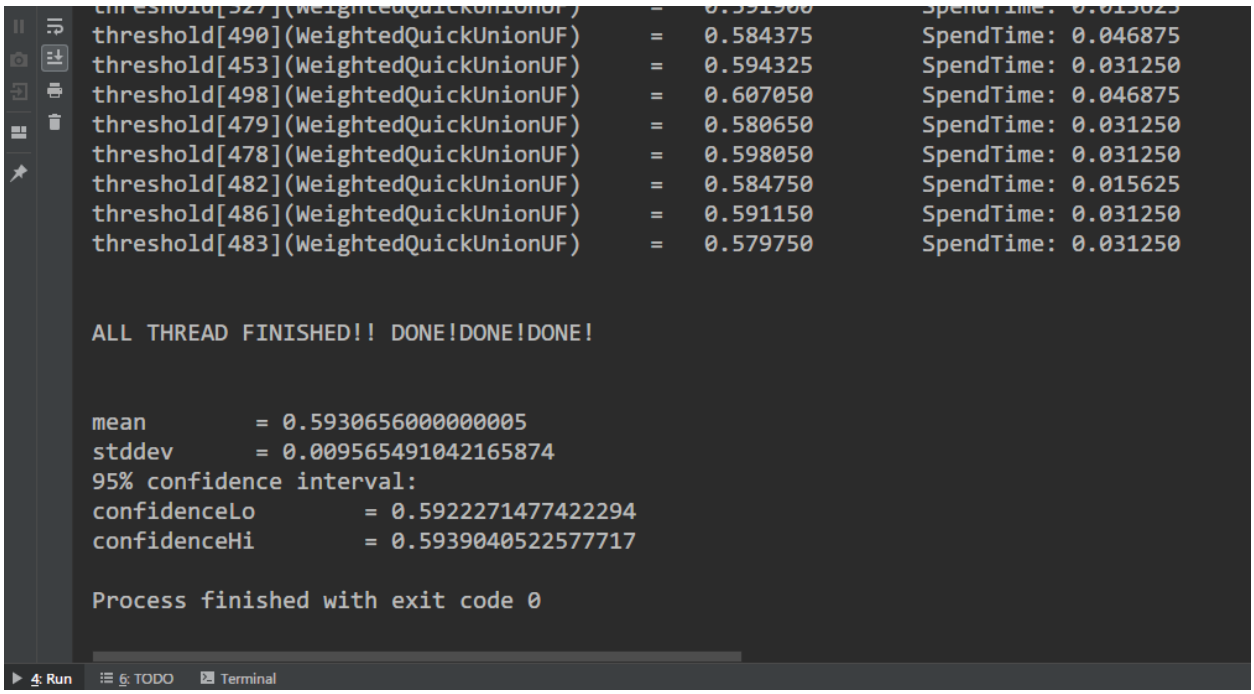
   对大小为 2000 的网格进行 50 次模拟，结果如下



   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

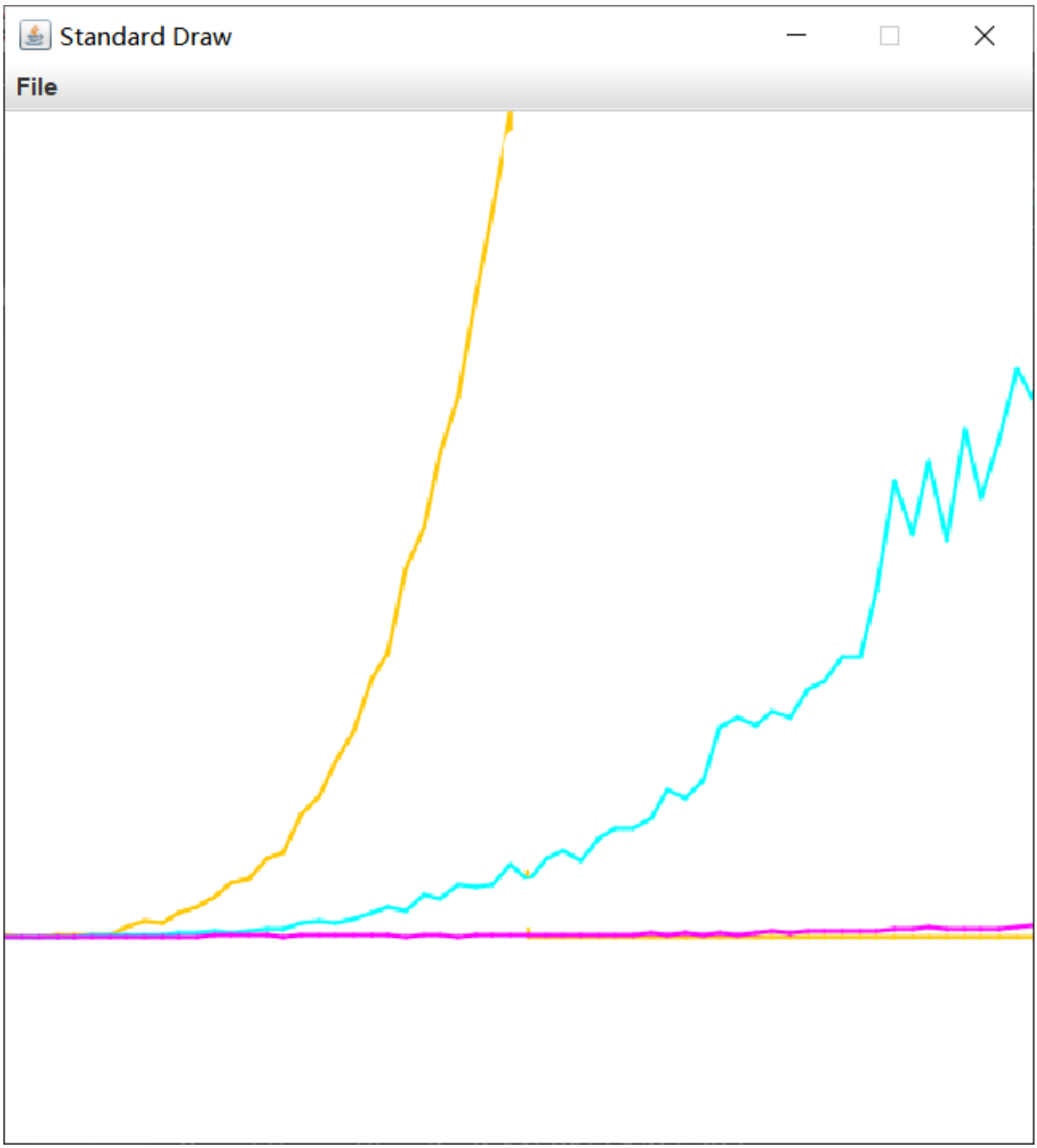
3. **不同的并查集算法性能比较**

   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：

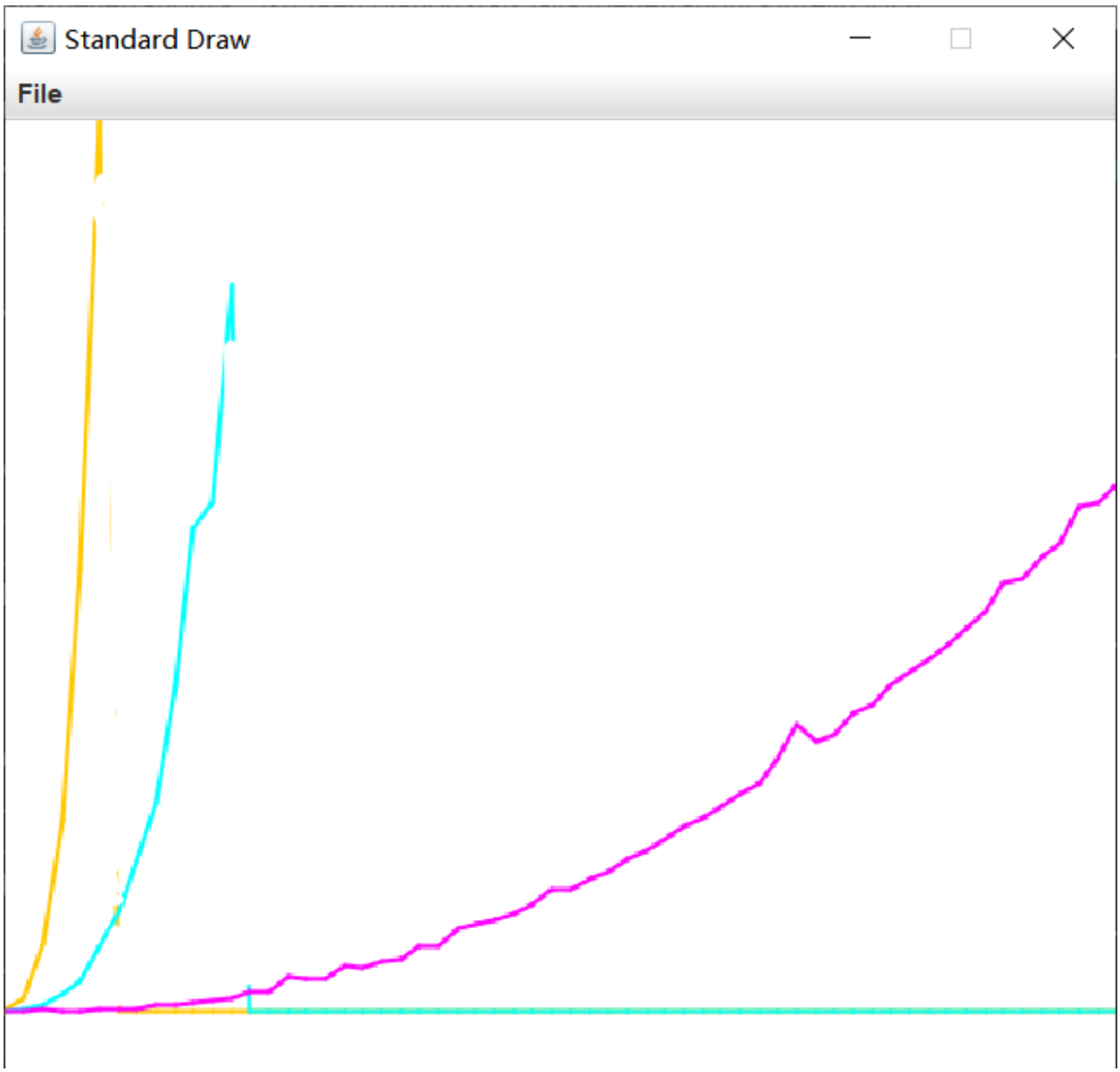
   Java 程序执行时，通过传入参数，控制最大网格规模，以最大网格规模为基础，由小到大，等间距取不同大小的网格，使用三种算法模拟渗透问题，进行算法性能分析

```
------ 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

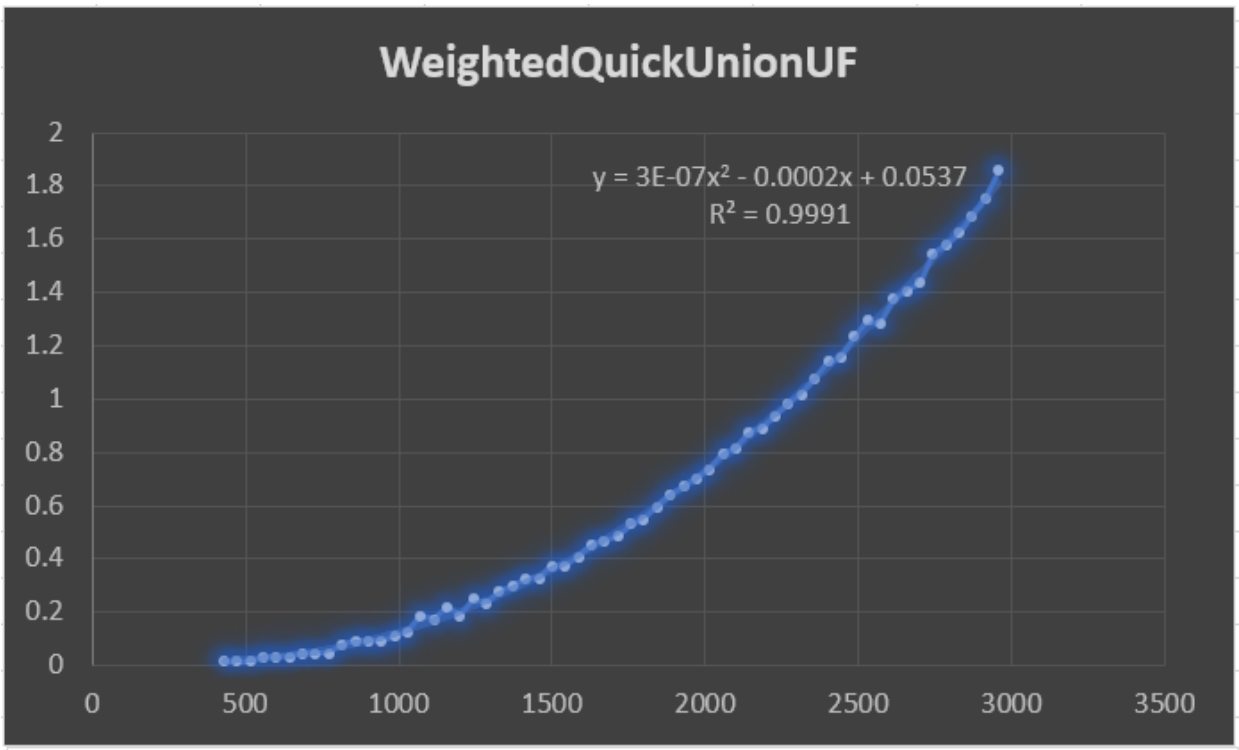


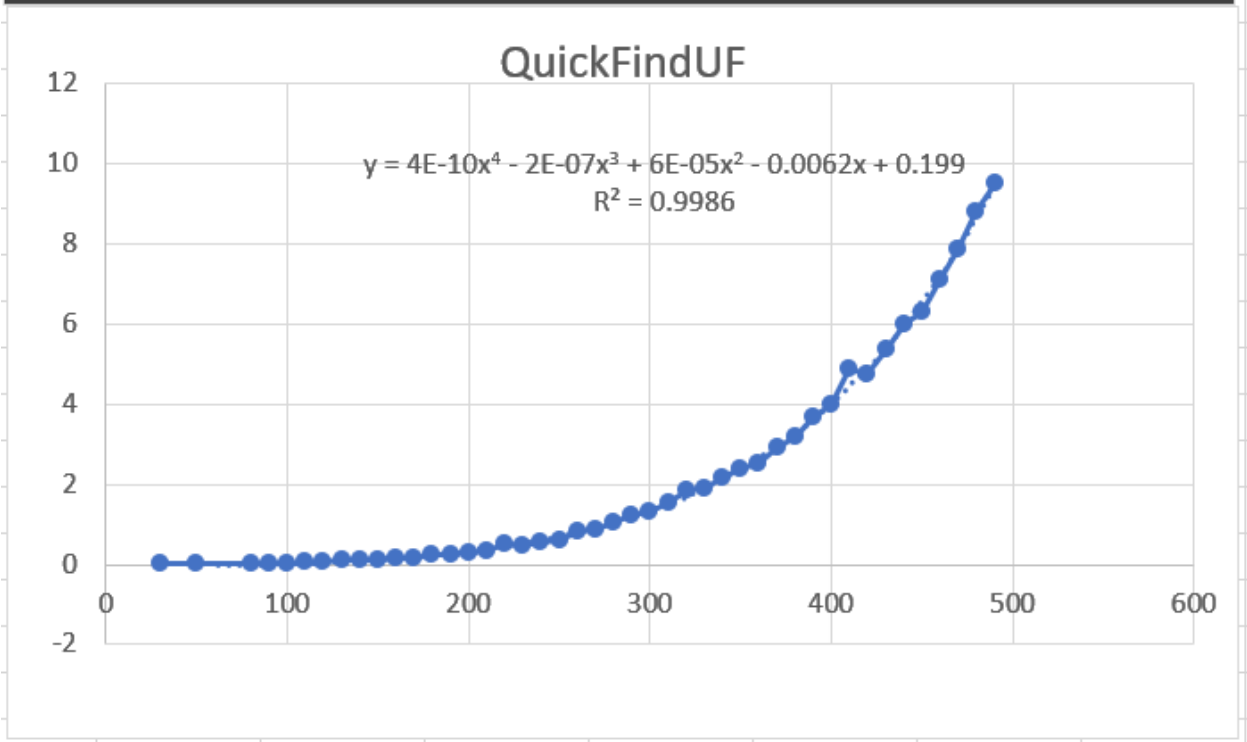
其中，橙色为QuickFindUF的运行时间，蓝色为QuickUnionUF 云香精时间，玫红色为WeightedQuickUnionUF 运行时间

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

```
1 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){

```

```
55          this(N,"",false);
56      }
57
58      /**
59       * 开放 x y 处的点
60       * @param x 点的横坐标
61       * @param y 点的纵坐标
62       */
63      public void open(int x, int y){
64          if (isopen(x, y)) return;
65          isopen[x][y] = true;
66          if (paint) painter.printOpen(x,y);
67          int dx[] = {0,0,-1,1};
68          int dy[] = {1,-1,0,0};
69          for (int i = 0; i<4;i++){
70              if (isopen(x+dx[i],y+dy[i])){
71                  uf.union(getUFId(x+dx[i],y+dy[i]), getUFId(x,y));
72              }
73          }
74          if (paint) if (isFull(x,y))    bfsIsFull(x,y);
75      }
76
77      /**
78       * 判断 x y 处的点是否开放
79       * @param x 点的横坐标
80       * @param y 点的纵坐标
81       */
82      public boolean isopen(int x, int y){
83          if (x<0 || x>=N || y<0 || y>=N){
84              return false;
85          }else {
86              return isopen[x][y];
87          }
88      }
89
90      /**
91       * 判断 x y 点处是否已经注入水（绘图用）
92       * @param x 点的横坐标
93       * @param y 点的纵坐标
94       */
95      public boolean isFull(int x, int y){
96          return isopen(x, y) && uf.connected(0, getUFId(x, y));
97      }
98
99      /**
100      * 判断全图是否渗透
101      *
102      * @return true 渗透
103      */
104     public boolean percolates(){
105         return uf.connected(0,N*N+1);
106     }
107
```

```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;
```

```
161             }
162         }
163     }
```

实验二：几种排序算法的实验性能比较

张俊华 16030199025

一、实验内容

实现插入排序（Insertion Soxt，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），Dikstra3-路划分快速排序（Quicksot with Dikstca 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。

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. 几种排序的实现

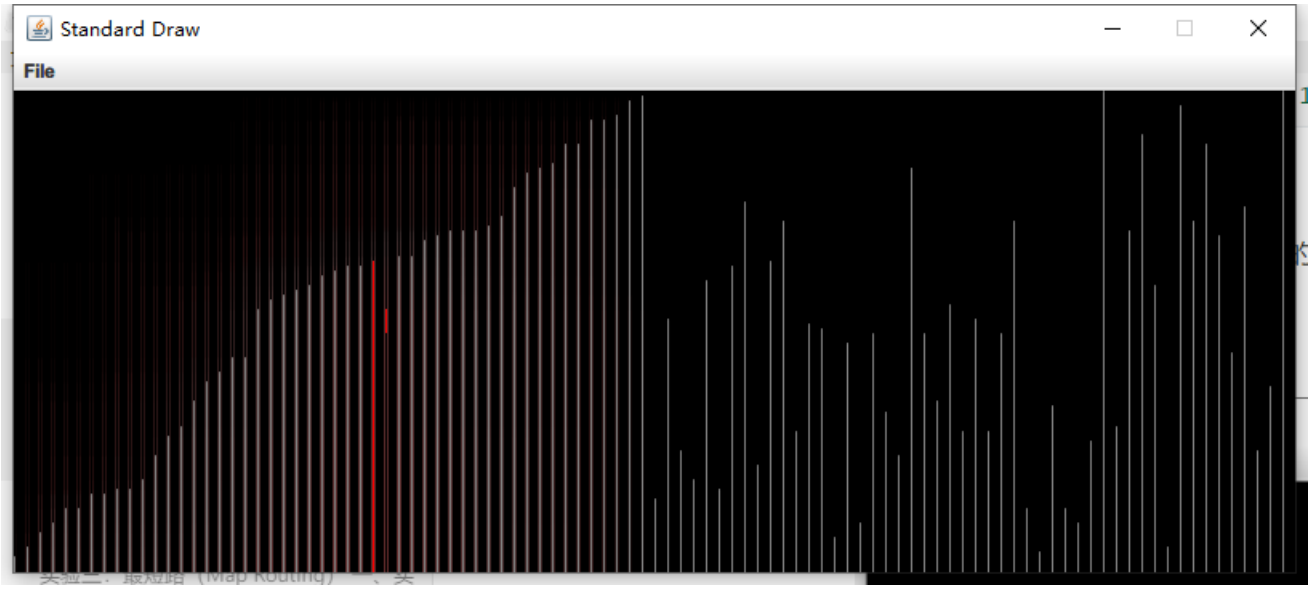
1. 按照题目要求，设计题目要求的插入排序（Insertion Sort，IS），自顶向下归并排序（Top-down Mergesort，TDM），自底向上归并排序（Bottom-up Mergesort，BUM），随机快速排序（Random Quicksort，RQ），Dijkstra 3-路划分快速排序（Quicksort with Dijkstra 3-way Partition，QD3P）五种排序算法
2. 编写 `generateRandom()` 函数，实现产生指定大小的随机数组功能，用于排序
3. 使用 `LinkedHashMap` 数据结构，对每次排序的时间和空间开销进行记录
4. 使用 JVM 虚拟机提供的 `Runtime` 类，对排序算法执行期间的空间开销进行计算。

```
        Comparable[] tlist = list.clone();
        LinkedHashMap<String,Double> timeResult = new LinkedHashMap<>();
        LinkedHashMap<String,Double> memResult = new LinkedHashMap<>();

        Stopwatch w1  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        long MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        Insertion.sort(tlist);
        long MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("IS",w1.elapsedTime());
        memResult.put("IS",1.0*(MemoryNow - MemoryBefore)/1000);
```

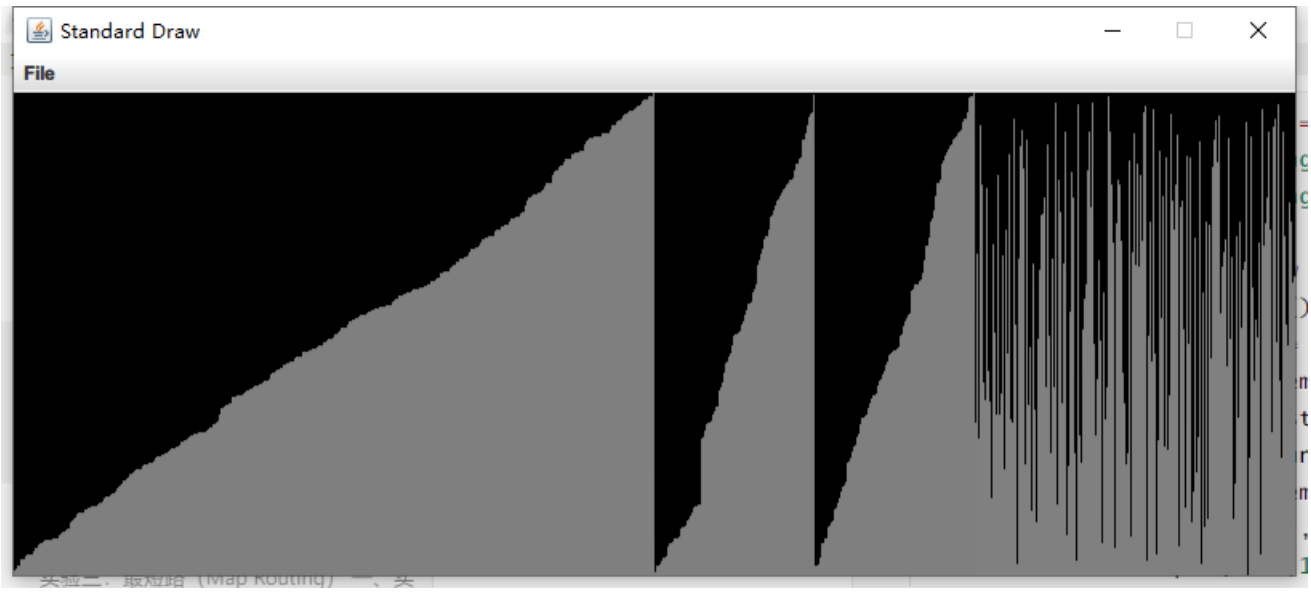
2. 排序算法的可视化

在实现了不同算法的时间空间开销记录的基础上，编写 `DrawGraph` 函数，可以将排序算法的执行过程可视化输出

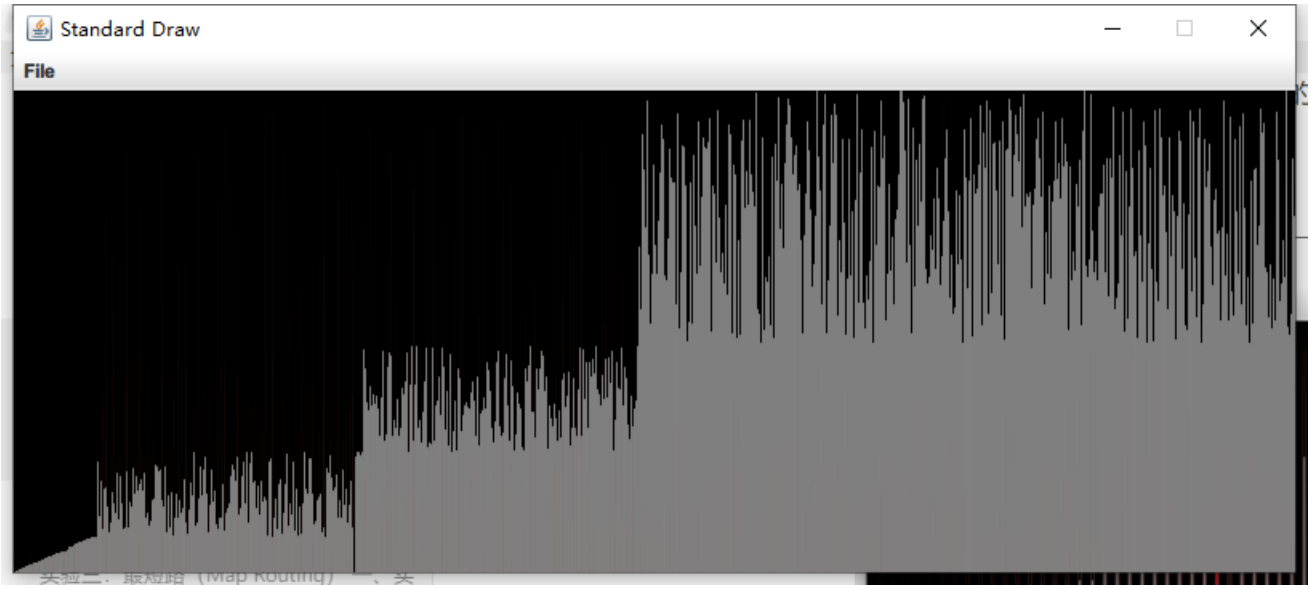
插入排序可视化中间过程：



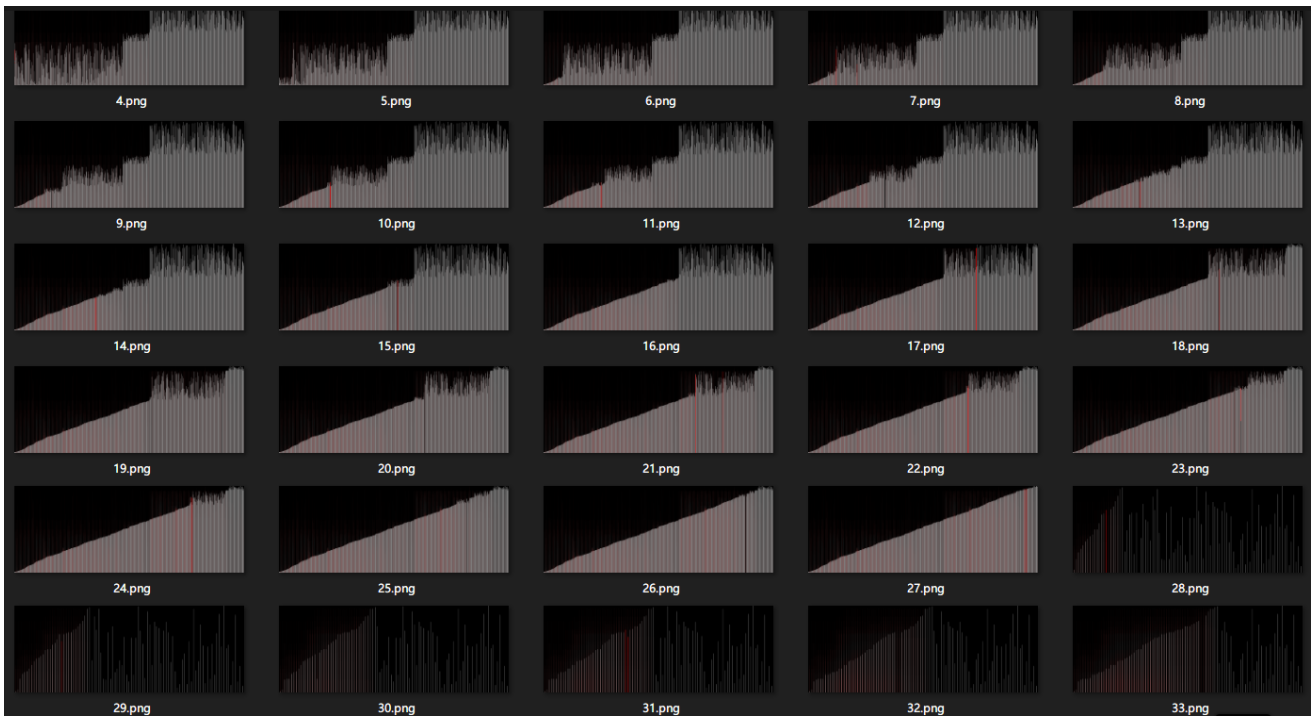
归并排序 1000 个数据元素 可视化中间过程：



快速排序 1000 个数据元素 可视化中间过程：



可视化的原理是重写了排序函数，在排序的关键部分，如元素交换，递归返回等位置，调用绘图函数，在改变数组元素的同时，重绘视窗中发生改变的区域，实现算法执行过程的动态展示。



四、实验结果

测试从 1000 到 20000 个元素的排序，从中选取部分测试结果如下所示

测试排序规模：3241 个元素

```
==================
RUN TEST (4/20) size: 3241
==================
TIME RESULTS        (ms)
  IS        39.00   44.00   45.00   32.00   32.00   26.00   28.00   27.00   24.00   24.00  32.1
 TDM        10.00   11.00   20.00   12.00   9.00    9.00    11.00   8.00    8.00    8.00   10.6
 BUM        10.00   15.00   14.00   8.00    9.00    9.00    9.00    7.00    8.00    8.00   9.7
  RQ        11.00   19.00   21.00   9.00    8.00    8.00    8.00    8.00    8.00    9.00   10.9
DQ3P        11.00   12.00   12.00   8.00    8.00    8.00    7.00    7.00    8.00    8.00   8.9
SPACE RESULTS       (KB)
  IS        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
 TDM        53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63  53.6328125
 BUM        53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63   53.63  53.6328125
  RQ        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
DQ3P        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
```

测试排序规模：5335 个元素

```
==================
RUN TEST (6/20) size: 5335
==================
TIME RESULTS        (ms)
  IS        56.00   55.00   61.00   59.00   55.00   54.00   53.00   54.00   55.00   54.00  55.6
 TDM        9.00    9.00    12.00   10.00   10.00   9.00    9.00    8.00    9.00    9.00   9.4
 BUM        8.00    9.00    10.00   8.00    8.00    9.00    9.00    9.00    8.00    8.00   8.6
  RQ        8.00    8.00    11.00   8.00    8.00    8.00    8.00    9.00    9.00    9.00   8.6
DQ3P        8.00    8.00    11.00   9.00    9.00    8.00    8.00    8.00    8.00    9.00   8.6
SPACE RESULTS       (KB)
  IS        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
 TDM        61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81  61.8125
 BUM        61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81   61.81  61.8125
  RQ        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
DQ3P        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
```

测试排序规模：13711 个元素

```
==================
RUN TEST (14/20) size: 13711
==================
TIME RESULTS        (ms)
  IS        374.00  392.00  381.00  395.00  382.00  396.00  339.00  356.00  377.00  378.00 377.0
 TDM        13.00   12.00   12.00   12.00   37.00   13.00   12.00   18.00   13.00   12.00  15.4
 BUM        13.00   13.00   11.00   12.00   43.00   12.00   12.00   20.00   13.00   12.00  16.1
  RQ        11.00   11.00   12.00   12.00   36.00   12.00   11.00   20.00   12.00   11.00  14.8
DQ3P        14.00   15.00   15.00   12.00   27.00   11.00   12.00   14.00   12.00   12.00  14.4
SPACE RESULTS       (KB)
  IS        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
 TDM        94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53  94.53125
 BUM        94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53   94.53  94.53125
  RQ        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
DQ3P        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
```

测试排序规模：19993 个元素

```
==================
RUN TEST (20/20) size: 19993
==================
TIME RESULTS        (ms)
  IS        788.00  775.00  778.00  777.00  773.00  786.00  764.00  777.00  778.00  801.00 779.7
 TDM        16.00   15.00   14.00   15.00   15.00   16.00   14.00   15.00   14.00   16.00  15.0
 BUM        16.00   15.00   15.00   14.00   16.00   17.00   15.00   14.00   16.00   14.00  15.2
  RQ        13.00   13.00   13.00   13.00   14.00   14.00   13.00   14.00   13.00   13.00  13.3
DQ3P        14.00   15.00   16.00   14.00   15.00   15.00   16.00   15.00   15.00   16.00  15.1
SPACE RESULTS       (KB)
  IS        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
 TDM        119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07 119.0703125
 BUM        119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07  119.07 119.0703125
  RQ        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
DQ3P        0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00    0.00   0.0
```

观察实验结果可以发现，由于插入排序的算法时间复杂度最高，因此在不同规模的测试中所用时间均最久。快速排序在几种排序中所用时间最少，归并排序次之，但两者差距不大。

从空间占用情况看，因为归并排序需要额外的数组空间实现归并操作，因此其空间占用一直是所有排序算法中最高，且其空间占用随着测试规模的增大而增大。

附：部分源代码

```
import edu.princeton.cs.algs4.Insertion;
import edu.princeton.cs.algs4.Merge;
import edu.princeton.cs.algs4.Quick;
import edu.princeton.cs.algs4.Quick3way;
import edu.princeton.cs.algs4.MergeBU;

import edu.princeton.cs.algs4.StdRandom;

import java.util.LinkedHashMap;


public class SortTest {
    public static Integer[] randomList;
    /**
     * 产生 size 大小的随机数组
     * @param size
     */
    public static Integer[] generateRandom(int size){
        randomList = new Integer[size];
        for (int j = 0; j<size;j++){
            randomList[j] = StdRandom.uniform(size);
        }
        StdRandom.shuffle(randomList);
        return randomList;
    }

    /**
     * 对 list 数组元素进行一次排序测试
     * @param list
     */
    public static LinkedHashMap[] runTest(Comparable[] list){
        Comparable[] tlist = list.clone();
        LinkedHashMap<String,Double> timeResult = new LinkedHashMap<>();
        LinkedHashMap<String,Double> memResult = new LinkedHashMap<>();

        Stopwatch w1  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        long MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        Insertion.sort(tlist);
        long MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("IS",w1.elapsedTime());
```

```
        memResult.put("IS",1.0*(MemoryNow - MemoryBefore)/1000);


        tlist = list.clone();
        Stopwatch w2  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        Merge.sort(tlist);
        MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("TDM",w2.elapsedTime());
        memResult.put("TDM",1.0*(MemoryNow - MemoryBefore)/1024);

        tlist = list.clone();
        Stopwatch w3  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        MergeBU.sort(tlist);
        MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("BUM",w3.elapsedTime());
        memResult.put("BUM",1.0*(MemoryNow - MemoryBefore)/1024);

        tlist = list.clone();
        Stopwatch w4  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        Quick.sort(tlist);
        MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("RQ",w4.elapsedTime());
        memResult.put("RQ",1.0*(MemoryNow - MemoryBefore)/1024);

        tlist = list.clone();
        Stopwatch w5  = new Stopwatch();
        Runtime.getRuntime().gc(); //空间回收
        MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        Quick3way.sort(tlist);
        MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
        timeResult.put("DQ3P",w5.elapsedTime());
        memResult.put("DQ3P",1.0*(MemoryNow - MemoryBefore)/1024);

        LinkedHashMap[] RESULT = new  LinkedHashMap[2];
        RESULT[0] = timeResult;
        RESULT[1] = memResult;
        return RESULT;
    }

```

```
    public  void runTests(int size){

        TestResult rec = new TestResult();
        for(int i = 0; i<10;i++){
            generateRandom(size);
            rec.record(runTest(randomList));
        }
        rec.printResult();
    }

}
```

```
import edu.princeton.cs.algs4.StdOut;
import edu.princeton.cs.algs4.StdStats;

import java.util.ArrayList;
import java.util.Map;
import java.util.LinkedHashMap;


public class TestResult {
    LinkedHashMap<String, ArrayList<Double>> timeResults = new LinkedHashMap<>();
    LinkedHashMap<String, ArrayList<Double>> spaceResults = new LinkedHashMap<>();
    int testTimes;


    /**
     * 记录一次测试结果
     */
    public void recordTimeResult(LinkedHashMap<String,Double> result){
        recordResult(result, timeResults);
    }
    /**
     * 记录一次测试结果
     */
    public void recordSpaceResult(LinkedHashMap<String,Double> result){
        recordResult(result, spaceResults);
    }

    private void recordResult(LinkedHashMap<String, Double> result,
LinkedHashMap<String, ArrayList<Double>> timeResults) {
        if (timeResults.isEmpty()){
            for (Map.Entry<String, Double> entry : result.entrySet()) {
                ArrayList<Double> time = new ArrayList<>();
                time.add(entry.getValue());
                timeResults.put(entry.getKey(),time);
            }
        }
        else {
            for (Map.Entry<String, Double> entry : result.entrySet()) {
                timeResults.get(entry.getKey()).add(entry.getValue());
            }
        }
```

```
    }

    public void record(LinkedHashMap<String, Double>[] result){
        recordTimeResult(result[0]);
        recordSpaceResult(result[1]);
    }

    /**
     * 打印测试结果
     */
    public void printResult(){
        StdOut.println("TIME RESULTS \t\t (ms)");
        prs(timeResults);
        StdOut.println("SPACE RESULTS \t\t (KB)");
        prs(spaceResults);

    }

    private void prs(LinkedHashMap<String, ArrayList<Double>> spaceResults) {
        for (Map.Entry<String, ArrayList<Double>> entry : spaceResults.entrySet())
{
            StdOut.printf("%4s\t\t",entry.getKey());
            double[] entryTime = new double[entry.getValue().size()];
            int i = 0;
            for (Double time : entry.getValue()){
                StdOut.printf("% 4.2f\t",time);
                entryTime[i++] = time;
            }
            StdOut.println(StdStats.mean(entryTime));
        }
    }
}

```

实验三：最短路（Map Routing）

张俊华 16030199025

一、实验内容

实现经典的 Dijkstra 最短路径算法，并对其进行优化。

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。

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. Dijkstra 基本算法

在 `dijkstra.java` 文件中，对 Dijkstra 算法进行了基本实现。

其核心函数为

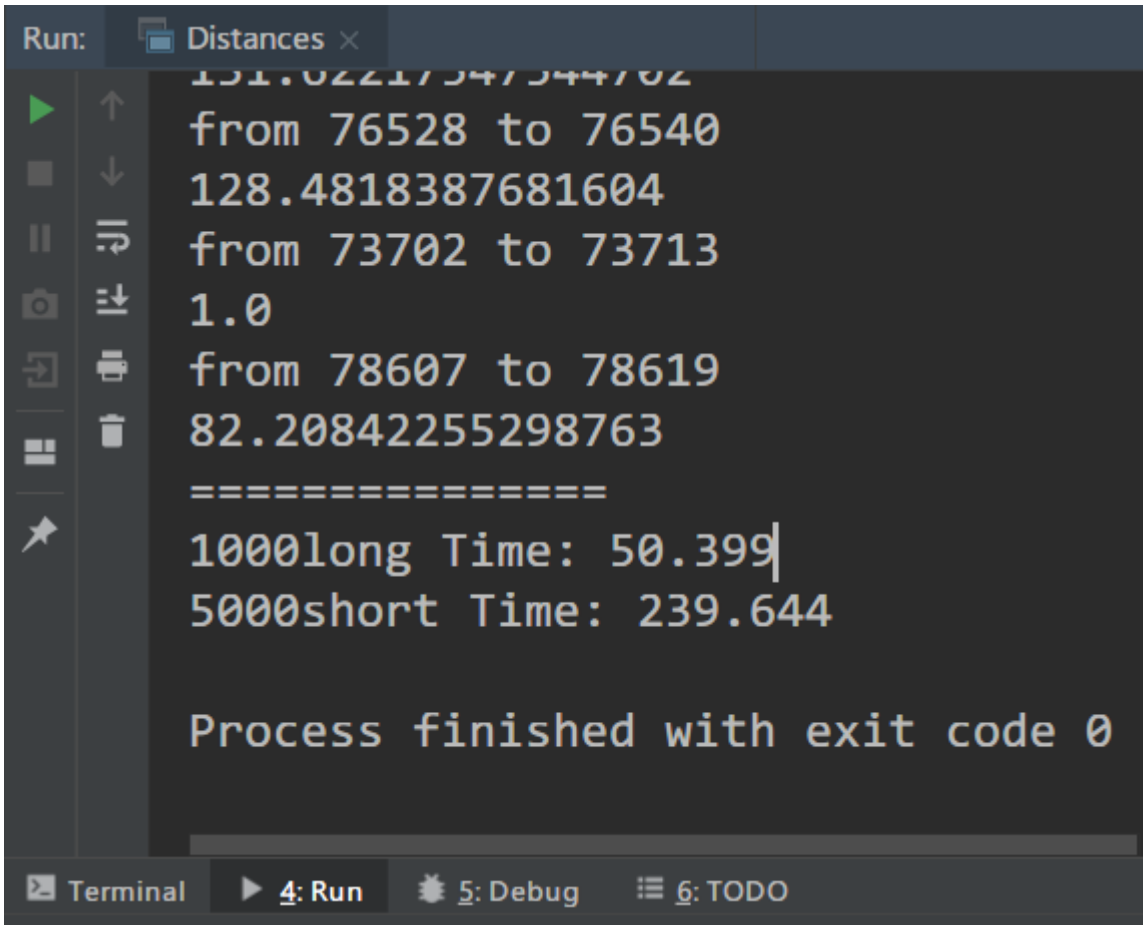
```
private void dijkstra(int s, int d)
```

实现了以 s 为起点，到其余各点的最短路径

```
private void dijkstra(int s, int d) {
        int V = G.V();

        // initialize
        dist = new double[V];
        pred = new int[V];
        for (int v = 0; v < V; v++) dist[v] = INFINITY;
        for (int v = 0; v < V; v++) pred[v] = -1;

        // priority queue
        IndexPQ pq = new IndexPQ(V);
        for (int v = 0; v < V; v++) pq.insert(v, dist[v]);

        // set distance of source
        dist[s] = 0.0;
```

```
        pred[s] = s;
        pq.change(s, dist[s]);

        // run Dijkstra's algorithm
        while (!pq.isEmpty()) {
            int v = pq.delMin();
            //// System.out.println("process " + v + " " + dist[v]);

            // v not reachable from s so stop
            if (pred[v] == -1) break;

            // scan through all nodes w adjacent to v
            IntIterator i = G.neighbors(v);
            while (i.hasNext()) {
                int w = i.next();
                if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
                    dist[w] = dist[v] + G.distance(v, w);
                    pq.change(w, dist[w]);
                    pred[w] = v;
                    //// System.out.println("    lower " + w + " to " + dist[w]);
                }
            }
        }
    }
```

该算法是未经优化过的最简实现，虽然能完成最短路径的计算，但是，求得每条最短路径的复杂度为 N^2



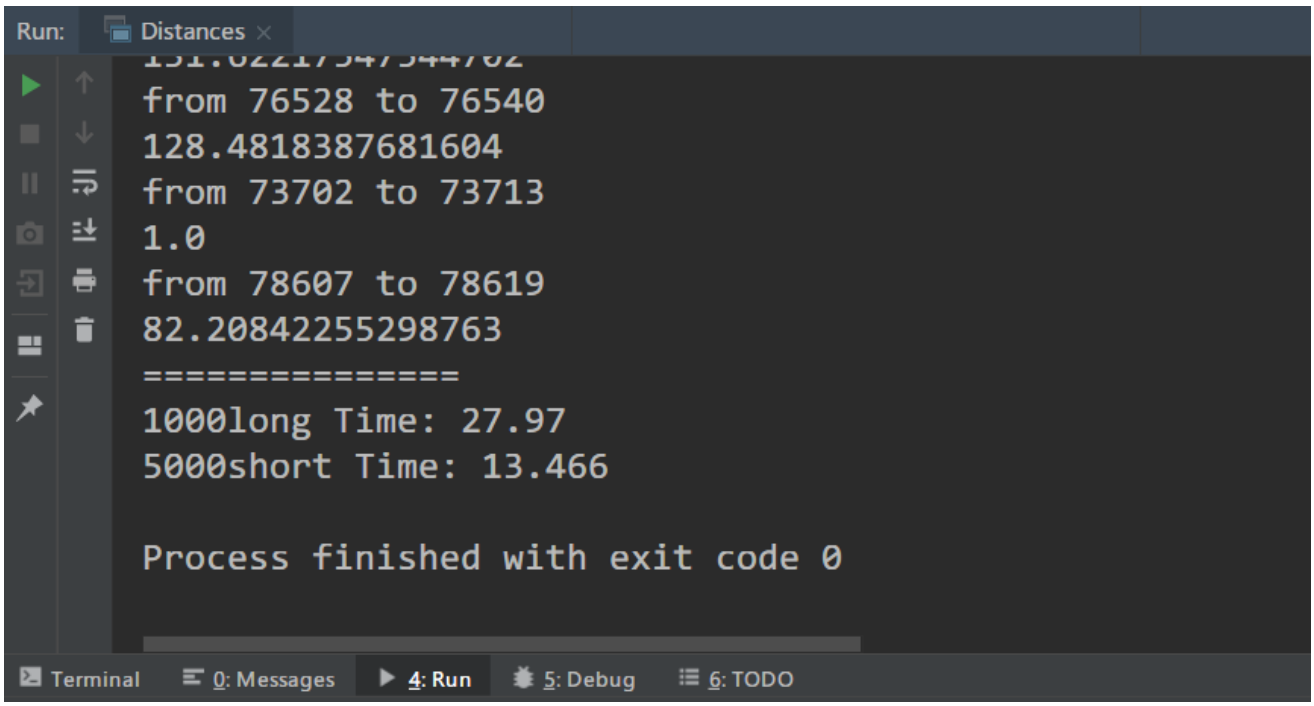
使用该算法完成 1000条长路径和 5000条短路径所用的总时间达到了 289s

2. 优化，发现最短路之后就停止搜索

由 Dijkstra 算法可知，从优先队列里面取出的最小顶点，其最短路一定已经确定，因此，可以直接停止搜索

```
while (!pq.isEmpty()) {
            int v = pq.delMin();
            if (v == d) break;
            ...
       }
```

提前停止搜索后，算法的执行时间有了显著的减少：



还是不够快?

因为每次查询最短路，整个 `dist` 数组，整个 `pred` 数组，全部都需要重新初始化，而这个初始化时间是与 V 成正比的，因此，如果能每次只初始化上次修改过的元素，还可以进一步提高速度。

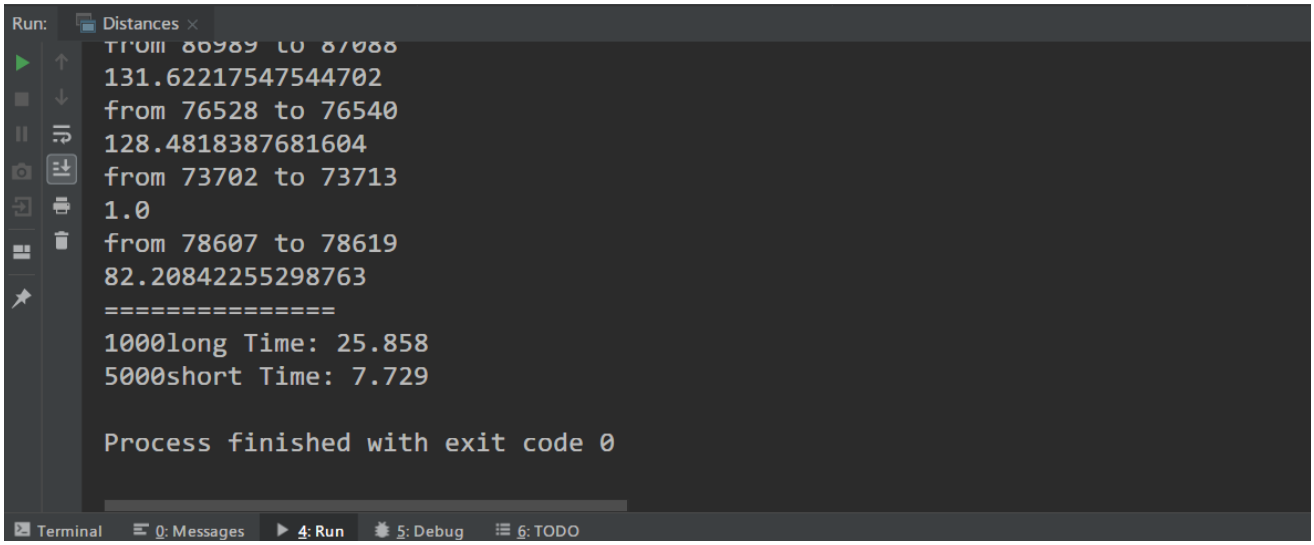
可以将 `dist` `pred` 两个数组的初始化，以及优先队列的初始化操作在 `dijstra` 对象的构造函数中执行：

```
public Dijkstra(EuclideanGraph G) {
        this.G = G;
        int V = G.V();
        // initialize
        dist = new double[V];
        pred = new int[V];
        for (int v = 0; v < V; v++) dist[v] = INFINITY;
        for (int v = 0; v < V; v++) pred[v] = -1;

        // priority queue
        pq = new IndexPQ(V);
        for (int v = 0; v < V; v++) pq.insert(v, dist[v]);
```

```
13
14      }
```

修改 dijstra 函数，使用 `LinkedList` 类型变量 `changed` 记录每次求最短路过程中发生了变化的顶点元素，之后，在每次求最短路之前，只对这些顶点初始化：

```
1  private void dijkstra(int s, int d) {
2          int V = G.V();
3          pq.N = V;
4          while (!changed.isEmpty()){
5              int i = changed.removeFirst();
6              dist[i] = INFINITY;
7              pred[i] = -1;
8              pq.change(i,INFINITY);
9          }
10
11         //pq = new IndexPQ(V);
12         //for (int v = 0; v < V; v++) pq.insert(v, INFINITY);
13
14         // set distance of source
15         dist[s] = 0.0;
16         pred[s] = s;
17         changed.add(s);
18         pq.change(s, dist[s]);
19
20         // run Dijkstra's algorithm
21         while (!pq.isEmpty()) {
22             int v = pq.delMin();
23             if (v == d) break;
24             //// System.out.println("process " + v + " " + dist[v]);
25
26             // v not reachable from s so stop
27             if (pred[v] == -1) break;
28
29             // scan through all nodes w adjacent to v
30             IntIterator i = G.neighbors(v);
31             while (i.hasNext()) {
32                 int w = i.next();
33                 if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
34                     dist[w] = dist[v] + G.distance(v, w);
35                     //G.point(v).drawTo(G.point(w));
36                     pq.change(w, dist[w]);
37                     pred[w] = v;
38                     changed.add(w);
39                     //// System.out.println("    lower " + w + " to " + dist[w]);
40                 }
41             }
42         }
43     }
```



可以看到，在完成上述修改后，对短路径的执行效率提升效果最大，所用时间缩短为一半

3. 使用 A* 算法，减少搜索范围

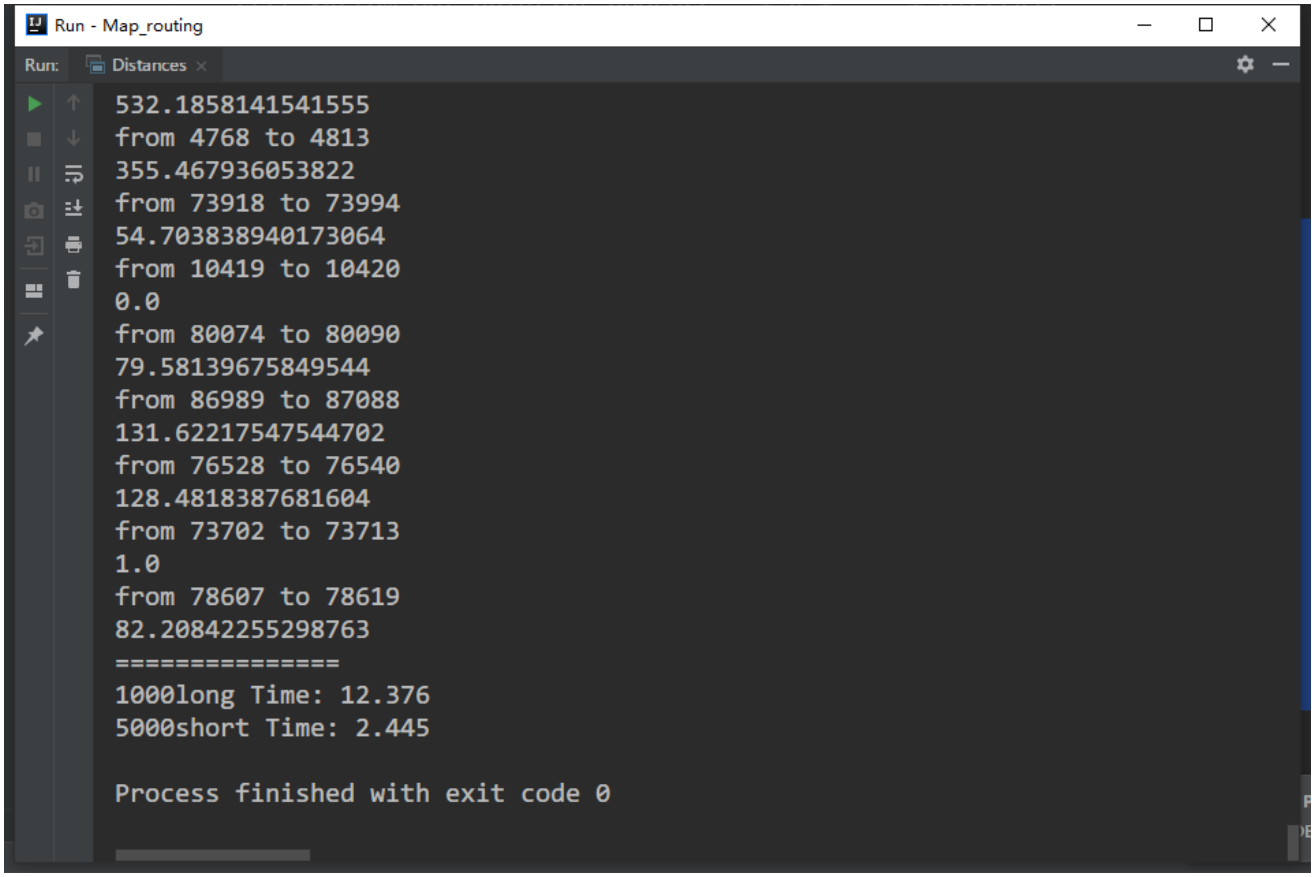
由于我们在地图中寻找最短路径，两点间路径的权重即为这两点之间的欧氏距离。因此，可以给 dijstra 加入一个启发式函数，构成 A* 算法，能更快的到达目的地。

按照题目给出的方案，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dw]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。

因此，将 Dijkstra 算法部分修改为如下：

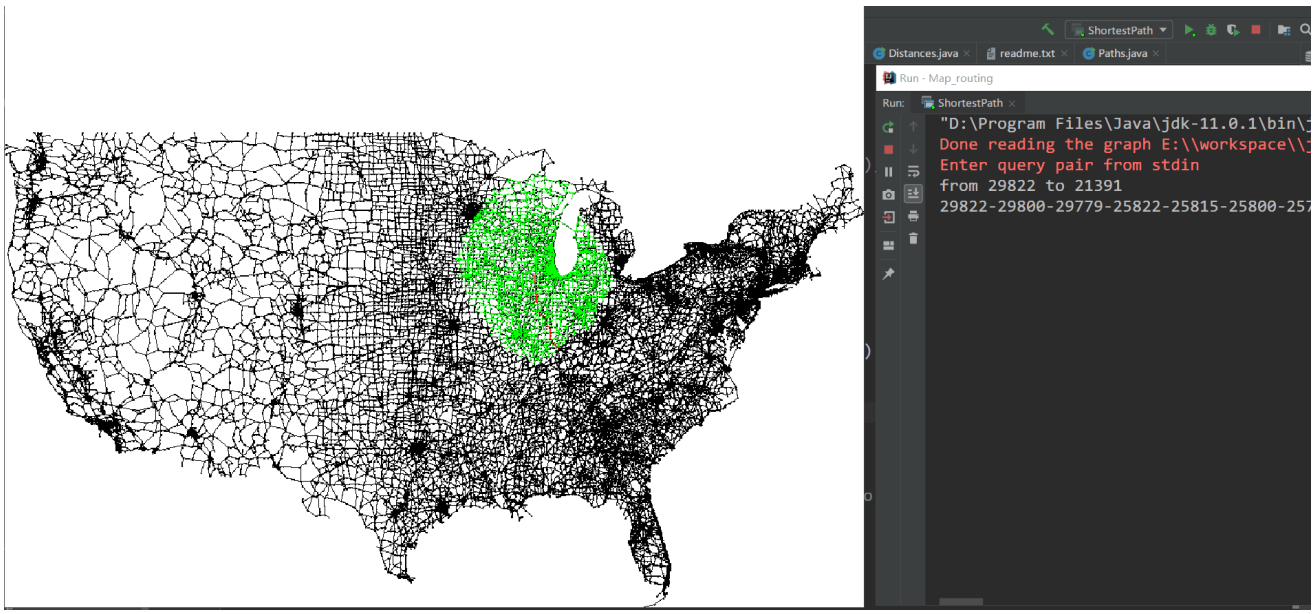
```
while (i.hasNext()) {
                int w = i.next();
                double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v, d);
                if (wt[v] + dt< wt[w] - EPSILON) {
                //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
                    dist[w] = dist[v] + G.distance(v, w);
                    wt[w]=wt[v] + dt;
                    //G.point(v).drawTo(G.point(w));
                    //pq.change(w, wt[w]);
                    mpq.changeKey(w,wt[w]);
                    pred[w] = v;
                    changed.add(w);
                    //Thread.sleep(2);
                    //// System.out.println("    lower " + w + " to " + dist[w]);
                }
            }
```

修改之后，重新计算完成 1000条长路径和 5000条短路径所用的总时间

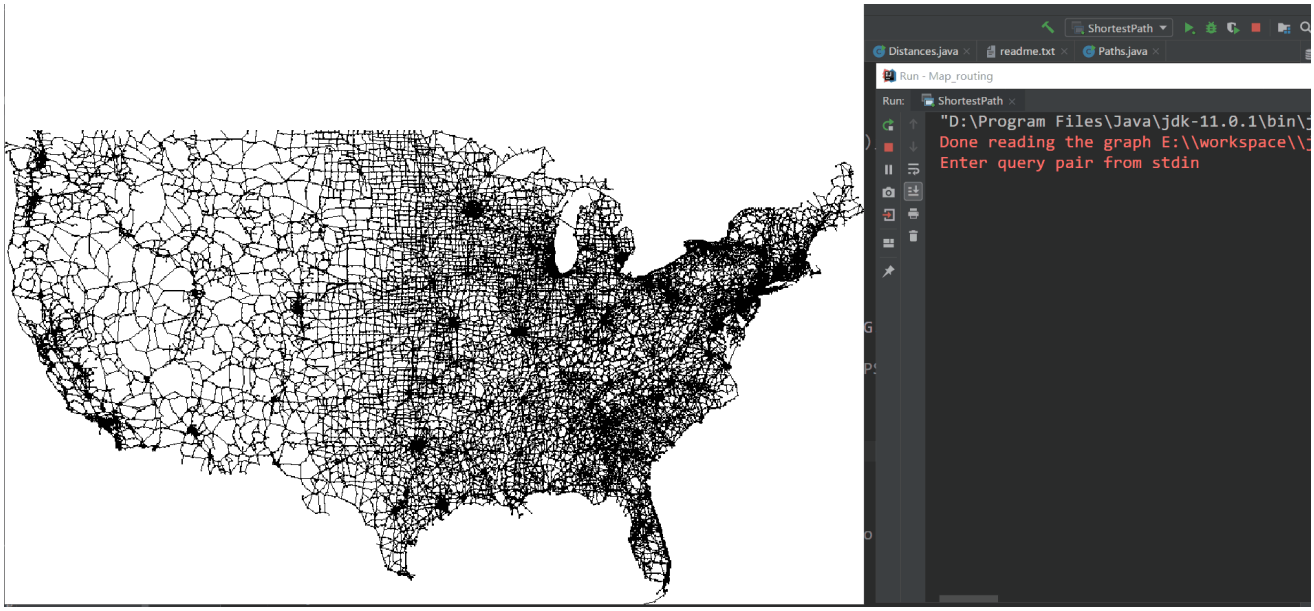


可以看到，完成上述修改后，算法的运行时间得到大幅减少，特别是执行短搜索的时间，从7s缩短到2s。

为算法执行可视化，可以明显看到，使用 A* 算法之前，整个搜索区域从起点开始以近似圆形展开，使用 A* 算法后，搜索区域有明显的指向性。大幅缩小了搜索范围

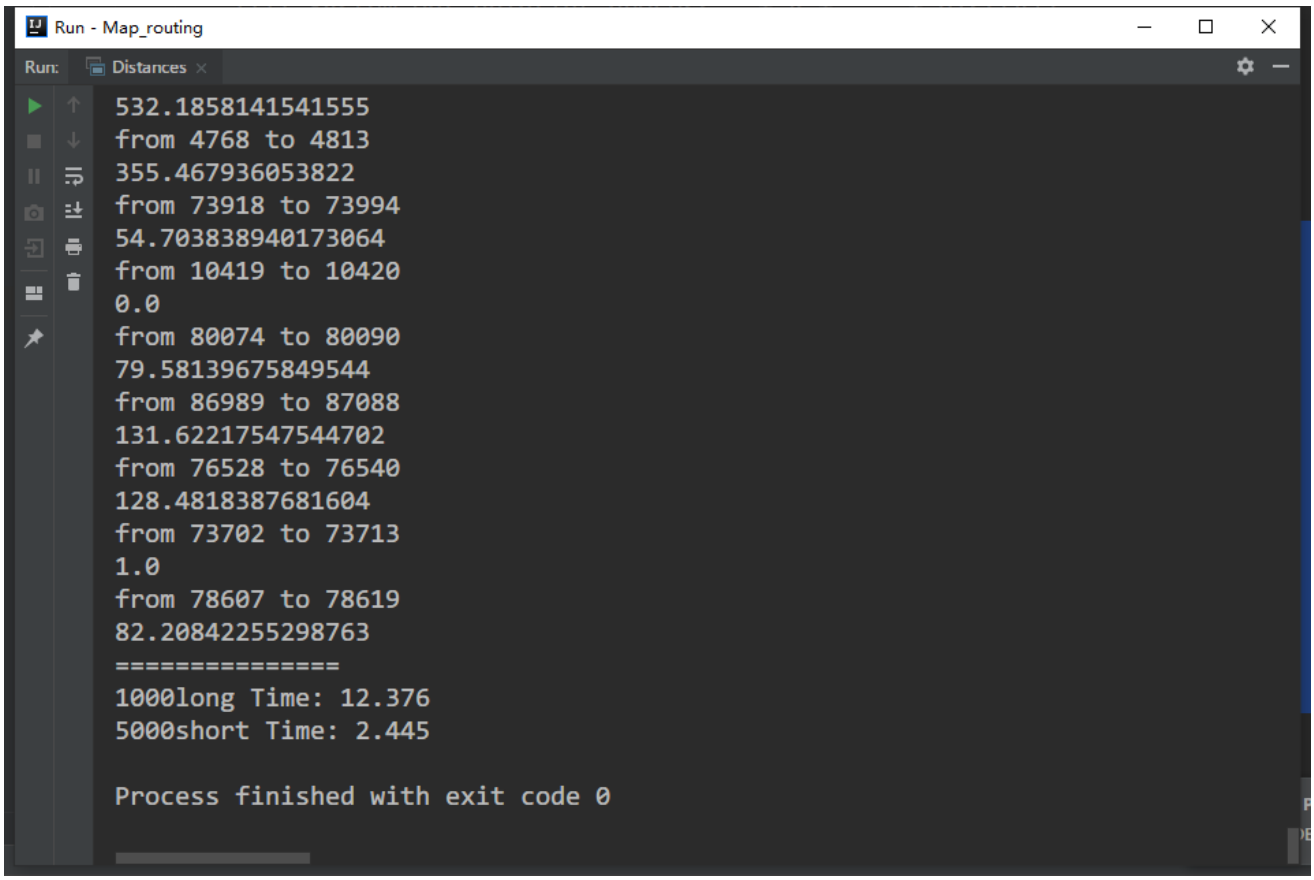


http://media.sumblog.cn/%E6%9C%AA%E4%BC%98%E5%8C%961.gif



四、实验结果

完成上述修改之后，计算完成 1000条长路径和 5000条短路径所用的总时间



1000 条长路径用时 12.376s

5000 条短路径用时 2.445 s

附：改进后的 Dijkstra 算法

```java
/******************************************************************************
 *  Dijkstra's algorithm.
 *
 ******************************************************************************/

import edu.princeton.cs.algs4.StdOut;

import java.awt.*;
import java.util.LinkedList;
//import edu.princeton.cs.algs4.IndexMultiwayMinPQ;



public class Dijkstra {
    private static double INFINITY = Double.MAX_VALUE;
    private static double EPSILON  = 0.000001;

    private EuclideanGraph G;
    private double[] dist;
    private int[] pred;
    private double[] wt;
    IndexPQ pq;
    IndexMultiwayMinPQ mpq;

    private LinkedList<Integer> changed = new LinkedList<Integer>();

    public Dijkstra(EuclideanGraph G) {
        this.G = G;
        int V = G.V();
        // initialize
        dist = new double[V];
        pred = new int[V];
        wt = new double[V];
        for (int v = 0; v < V; v++) dist[v] = INFINITY;
        for (int v = 0; v < V; v++) wt[v] = INFINITY;
        for (int v = 0; v < V; v++) pred[v] = -1;

        // priority queue
        pq = new IndexPQ(V);
        for (int v = 0; v < V; v++) pq.insert(v, dist[v]);
        mpq = new IndexMultiwayMinPQ<Double>(V,6);
        for (int v = 0; v< V; v++) mpq.insert(v, INFINITY);

    }

    // return shortest path distance from s to d
    public double distance(int s, int d) throws InterruptedException {
        dijkstra(s, d);
        return dist[d];
    }
```

```
51
52      // print shortest path from s to d  (interchange s and d to print in right
    order)
53      public void showPath(int d, int s,boolean draw) throws InterruptedException {
54          if (!draw)
55              dijkstra(s, d);
56          else
57              dijkstra(s,d,draw);
58          if (pred[d] == -1) {
59              System.out.println(d + " is unreachable from " + s);
60              return;
61          }
62          for (int v = d; v != s; v = pred[v])
63              System.out.print(v + "-");
64          System.out.println(s);
65      }
66
67      public void showPath(int d, int s) throws InterruptedException {
68          dijkstra(s, d);
69          if (pred[d] == -1) {
70              System.out.println(d + " is unreachable from " + s);
71              return;
72          }
73          for (int v = d; v != s; v = pred[v])
74              System.out.print(v + "-");
75          System.out.println(s);
76      }
77
78      // plot shortest path from s to d
79      public void drawPath(int s, int d) throws InterruptedException {
80          dijkstra(s, d);
81          if (pred[d] == -1) return;
82          //Turtle.setColor(Color.red);
83          //Turtle.set_Stroke(2.0f);
84          for (int v = d; v != s; v = pred[v]){
85              G.point(v).drawTo(G.point(pred[v]));
86              Turtle.render();
87          }
88
89          //Turtle.set_Stroke(1.0f);
90
91
92      }
93
94      // Dijkstra's algorithm to find shortest path from s to d
95      private void dijkstra(int s, int d) throws InterruptedException {
96          int V = G.V();
97          //pq.N = V;
98          mpq.n = V;
99          mpq.nmax = V;
100
101         while (!changed.isEmpty()){
102             int i = changed.removeFirst();
```

```
            dist[i] = INFINITY;
            wt[i] = INFINITY;
            pred[i] = -1;
            //StdOut.println(i);
            mpq.changeKey(i,INFINITY);
        }

        //mpq = new IndexMultiwayMinPQ<Double>(V,4);
        //for (int v = 0; v< V; v++) mpq.insert(v, INFINITY);

        // set distance of source
        dist[s] = 0.0;
        wt[s] = 0.0;
        pred[s] = s;
        changed.add(s);
        //pq.change(s, wt[s]);
        mpq.changeKey(s,wt[s]);

        // run Dijkstra's algorithm
        while (!mpq.isEmpty()) {
            //int v = pq.delMin();
            int v = mpq.delMin();
            if (v == d) break;
            //// System.out.println("process " + v + " " + dist[v]);

            // v not reachable from s so stop
            if (pred[v] == -1) break;

            // scan through all nodes w adjacent to v
            IntIterator i = G.neighbors(v);
            while (i.hasNext()) {
                int w = i.next();
                double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v,
d);
                if (wt[v] + dt< wt[w] - EPSILON) {
                //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
                    dist[w] = dist[v] + G.distance(v, w);
                    wt[w]=wt[v] + dt;
                    //G.point(v).drawTo(G.point(w));
                    //pq.change(w, wt[w]);
                    mpq.changeKey(w,wt[w]);
                    pred[w] = v;
                    changed.add(w);
                    //Thread.sleep(2);
                    //// System.out.println("    lower " + w + " to " + dist[w]);
                }
            }
        }
    }

    private void dijkstra(int s, int d,boolean draw) throws InterruptedException {

        int V = G.V();
```

```
        mpq.n = V;
        mpq.nmax = V;
        while (!changed.isEmpty()){
            int i = changed.removeFirst();
            dist[i] = INFINITY;
            wt[i] = INFINITY;
            pred[i] = -1;
            mpq.changeKey(i,INFINITY);
        }

        //pq = new IndexPQ(V);
        //for (int v = 0; v < V; v++) pq.insert(v, INFINITY);

        // set distance of source
        dist[s] = 0.0;
        wt[s] = 0.0;
        pred[s] = s;
        changed.add(s);
        mpq.changeKey(s, wt[s]);

        // run Dijkstra's algorithm
        while (!mpq.isEmpty()) {
            int v = mpq.delMin();
            if (v == d) break;
            //// System.out.println("process " + v + " " + dist[v]);

            // v not reachable from s so stop
            if (pred[v] == -1) break;

            // scan through all nodes w adjacent to v
            IntIterator i = G.neighbors(v);
            while (i.hasNext()) {
                int w = i.next();
                double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v,
d);
                if (wt[v] + dt< wt[w] - EPSILON) {
                    //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
                    dist[w] = dist[v] + G.distance(v, w);
                    wt[w]=wt[v] + dt;
                    Turtle.setColor(Color.blue);
                    //drawPath(s,v);
                    //drawPath(s,d);
                    Turtle.setColor(Color.green);
                    G.point(v).drawTo(G.point(w));
                    Turtle.render();
                    mpq.changeKey(w, wt[w]);
                    pred[w] = v;
                    changed.add(w);

                    //// System.out.println("    lower " + w + " to " + dist[w]);
                }
            }
            //Thread.sleep(1);
```

```
207             }
208         }
209
210
211 }
```

实验四：文本索引

张俊华 16030199025

一、实验内容

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。

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. 构建后缀数组

使用 c++ 的流操作运算，从 txt 文件中读取待查找文本。将文件内容保存在 `str` 字符串中。

```
std::ifstream ifstream("D:\\alice29.txt");
std::stringstream stream;
ifstream.seekg(0, std::ios::end);    // go to the end
int length = ifstream.tellg();           // report location (this is the length)
std::cout << "Input File length: " << length << std::endl;
ifstream.seekg(0, std::ios::beg);    // go back to the beginning
ifstream.read(str, length);       // read the whole file into the buffer
ifstream.close();                    // close file handle
```

2. 编写 suffixSort() 函数，实现后缀数组的排序

后缀数组保存在 pos[] 数组中，后缀数组的逆保存在 rank 数组中。使用 MSD 算法进行实现

```
void suffixSort(int n){
  //sort suffixes according to their first characters
  for (int i=0; i<n; ++i){
    pos[i] = i;
  }
  std::sort(pos, pos + n, smaller_first_char);
  //{pos contains the list of suffixes sorted by their first character}

  for (int i=0; i<n; ++i){
    bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
    b2h[i] = false;
```

```
  }

  for (int h = 1; h < n; h <<= 1){
    //{bh[i] == false if the first h characters of pos[i-1] == the first h characters of pos[i]}
    int buckets = 0;
    for (int i=0, j; i < n; i = j){
      j = i + 1;
      while (j < n && !bh[j]) j++;
      next[i] = j;
      buckets++;
    }
    if (buckets == n) break; // We are done! Lucky bastards!
    //{suffixes are separted in buckets containing strings starting with the same h characters}

    for (int i = 0; i < n; i = next[i]){
      cnt[i] = 0;
      for (int j = i; j < next[i]; ++j){
        rank[pos[j]] = i;
      }
    }

    cnt[rank[n - h]]++;
    b2h[rank[n - h]] = true;
    for (int i = 0; i < n; i = next[i]){
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0){
          int head = rank[s];
          rank[s] = head + cnt[head]++;
          b2h[rank[s]] = true;
        }
      }
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0 && b2h[rank[s]]){
          for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
        }
      }
    }
    for (int i=0; i<n; ++i){
      pos[rank[i]] = i;
      bh[i] |= b2h[i];
    }
  }
  for (int i=0; i<n; ++i){
    rank[pos[i]] = i;
  }
}
void getHeight(int n){
  for (int i=0; i<n; ++i) rank[pos[i]] = i;
  height[0] = 0;
```

```
  for (int i=0, h=0; i<n; ++i){
    if (rank[i] > 0){
      int j = pos[rank[i]-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      height[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
```

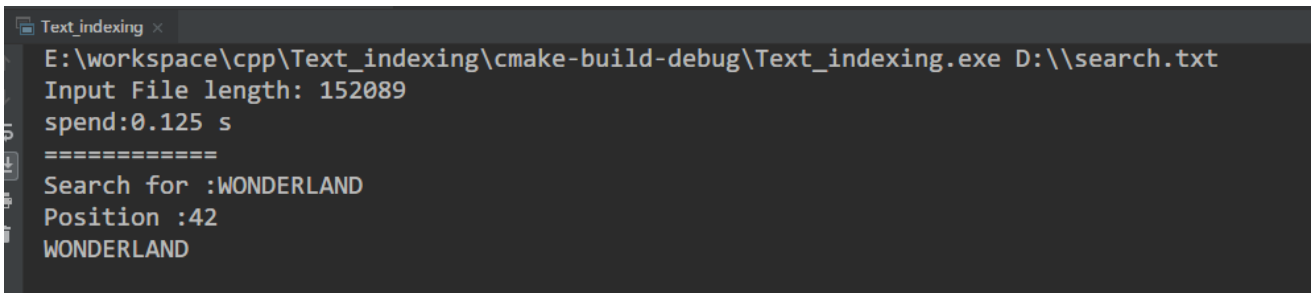
3. 编写二分查找函数

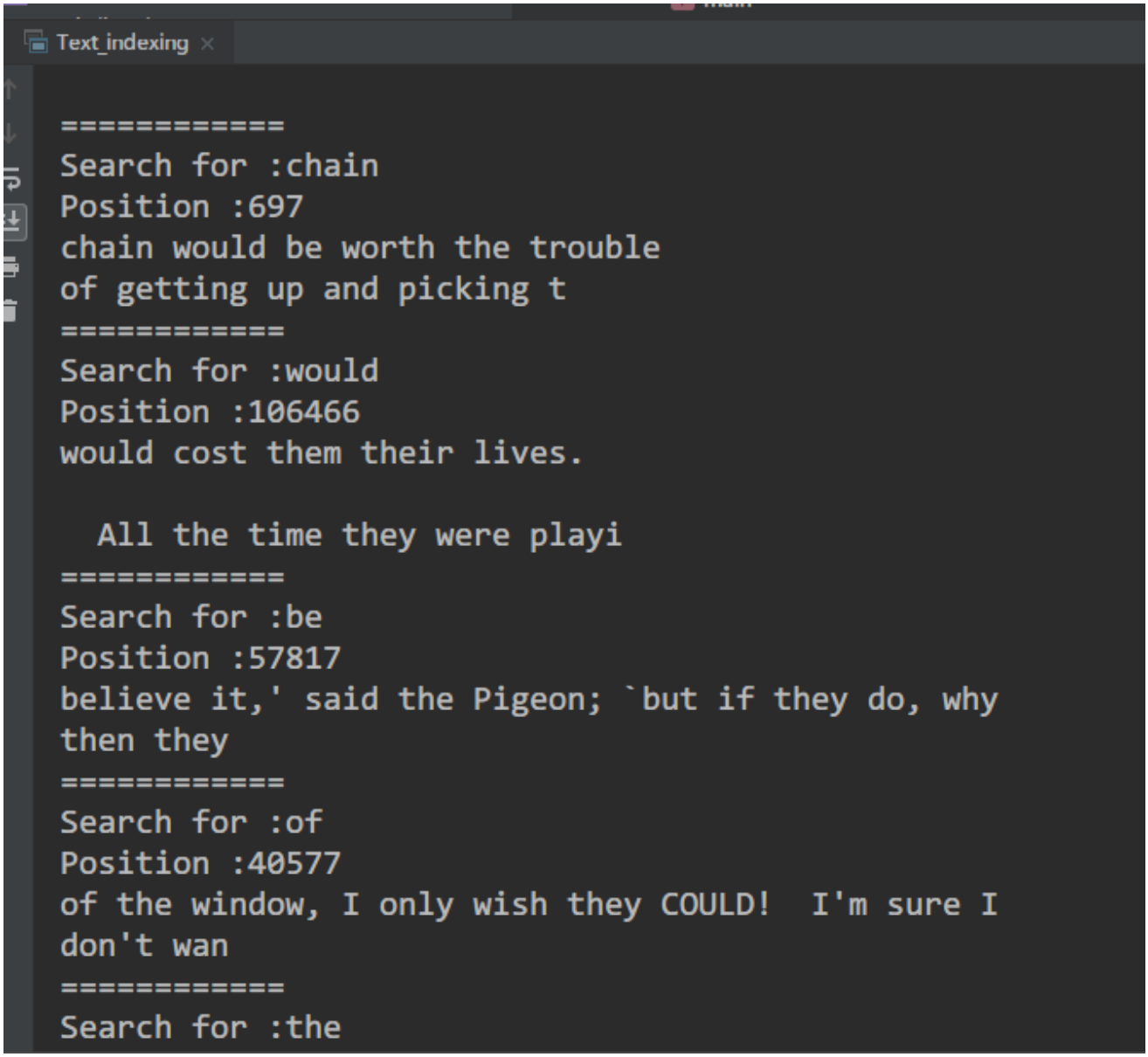
编写 `binarychop` 函数，利用二分查找，实现对输入的 key 关键字的查找匹配

```
int binarychop(char* key, int key_lengh, int left, int right){
    if (left > right){
        return -1;
    }
    int mid = (right-left)/2+left;
    int p = pos[mid];
    for (int i = 0; i < key_lengh; i++){
        if (key[i]<str[p+i]){
            return binarychop(key,key_lengh,left,mid-1);
        } else if (key[i] > str[p+i]){
            return binarychop(key,key_lengh,mid+1,right);
        }
    }
    return p;
}
```

四、实验结果

使用 `alice29.txt` 文本数据进行测试 （长度：152089）可以在 0.125s 实现对整个后缀数组排序





附：实验完整代码

```cpp
#include <iostream>
#include <algorithm>
#include <sstream>
#include <cstring>
#include <fstream>
#include <ctime>

const int N = 2000000;

char str[N]; //input
int rank[N], pos[N]; //output
int cnt[N], next[N]; //internal
bool bh[N], b2h[N];

```

```
// Compares two suffixes according to their first characters
bool smaller_first_char(int a, int b){
  return str[a] < str[b];
}

void suffixSort(int n){
  //sort suffixes according to their first characters
  for (int i=0; i<n; ++i){
    pos[i] = i;
  }
  std::sort(pos, pos + n, smaller_first_char);
  //{pos contains the list of suffixes sorted by their first character}

  for (int i=0; i<n; ++i){
    bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
    b2h[i] = false;
  }

  for (int h = 1; h < n; h <<= 1){
    //{bh[i] == false if the first h characters of pos[i-1] == the first h characters of pos[i]}
    int buckets = 0;
    for (int i=0, j; i < n; i = j){
      j = i + 1;
      while (j < n && !bh[j]) j++;
      next[i] = j;
      buckets++;
    }
    if (buckets == n) break;
    //{suffixes are separted in buckets containing strings starting with the same h characters}

    for (int i = 0; i < n; i = next[i]){
      cnt[i] = 0;
      for (int j = i; j < next[i]; ++j){
        rank[pos[j]] = i;
      }
    }

    cnt[rank[n - h]]++;
    b2h[rank[n - h]] = true;
    for (int i = 0; i < n; i = next[i]){
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0){
          int head = rank[s];
          rank[s] = head + cnt[head]++;
          b2h[rank[s]] = true;
        }
      }
      for (int j = i; j < next[i]; ++j){
        int s = pos[j] - h;
        if (s >= 0 && b2h[rank[s]]){
```

```
          for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
        }
      }
    }
    for (int i=0; i<n; ++i){
      pos[rank[i]] = i;
      bh[i] |= b2h[i];
    }
  }
  for (int i=0; i<n; ++i){
    rank[pos[i]] = i;
  }
}
// End of suffix array algorithm



int height[N];

void getHeight(int n){
  for (int i=0; i<n; ++i) rank[pos[i]] = i;
  height[0] = 0;
  for (int i=0, h=0; i<n; ++i){
    if (rank[i] > 0){
      int j = pos[rank[i]-1];
      while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
      height[rank[i]] = h;
      if (h > 0) h--;
    }
  }
}
// End of longest common prefixes algorithmd

int binarychop(char* key, int key_lengh, int left, int right){
    if (left > right){
        return -1;
    }
    int mid = (right-left)/2+left;
    int p = pos[mid];
    for (int i = 0; i < key_lengh; i++){
        if (key[i]<str[p+i]){
            return binarychop(key,key_lengh,left,mid-1);
        } else if (key[i] > str[p+i]){
            return binarychop(key,key_lengh,mid+1,right);
        }
    }
    return p;
}

int main(int argc, char ** argv) {
    std::ifstream ifstream("D:\\alice29.txt");
    std::stringstream stream;
    ifstream.seekg(0, std::ios::end);    // go to the end
```

```
    int length = ifstream.tellg();           // report location (this is the
length)
    std::cout << "Input File length: " << length << std::endl;
    ifstream.seekg(0, std::ios::beg);    // go back to the beginning
    ifstream.read(str, length);       // read the whole file into the buffer
    ifstream.close();                    // close file handle
    clock_t start,end;
    start = clock();
    suffixSort(strlen(str));
    end = clock();
    std::cout<<"spend:"<< (double)(end-start)/CLOCKS_PER_SEC << " s" << std::endl;

    std::ifstream search(argv[1]);
    while (search.peek()!=EOF){
        std::cout << "============" << std::endl;
        char key[1000];
        search >> key;
        search.get();
        std::cout << "Search for :" << key << std::endl;
        int p = binarychop(key, strlen(key),0,strlen(str));
        std::cout << "Position :" << p << "   " << std::endl;
        for (int i = 0 ;i < 60 ; i++){
            std::cout << str[p+i] ;
        }
        std::cout << std::endl;
    }

    return 0;
}
```