

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

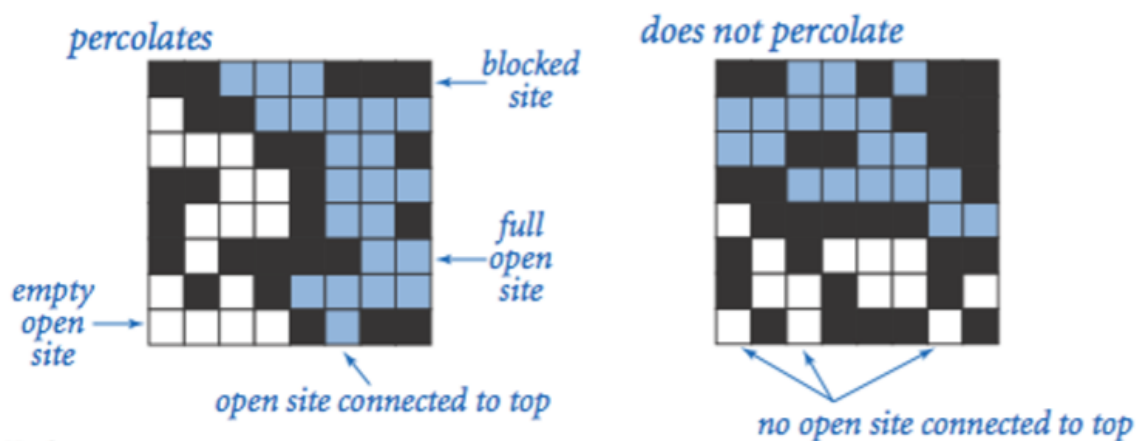
JVM: OpenJDK 64-Bit Server VM by JetBrains s.r.o

Windows 10 10.0

三、实验步骤

1. 构建 Percolation 类

$n \times n$ 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 $N \times N$ 个网格中的点进行操作，来模拟判断渗透情况

```

1 public class Percolation {
2     public Percolation(int n)           // create n-by-n grid, with all
      sites blocked
3     public void open(int row, int col)   // open site (row, col) if it is
      not open already
4     public boolean isOpen(int row, int col) // is site (row, col) open?
5     public boolean isFull(int row, int col) // is site (row, col) full?
6     public int numberOfOpenSites()        // number of open sites
7     public boolean percolates()           // does the system percolate?
8
9     public static void main(String[] args) // test client (optional)
10 }

```

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 $n \times n$ 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 x
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

```
Run: PercolationStats x
threshold[20](WeightedQuickUnionUF) = 0.593109 SpendTime: 1.765625
threshold[33](WeightedQuickUnionUF) = 0.590085 SpendTime: 1.703125
threshold[24](WeightedQuickUnionUF) = 0.591542 SpendTime: 1.734375
threshold[45](WeightedQuickUnionUF) = 0.598336 SpendTime: 1.703125

ALL THREAD FINISHED!! DONE!DONE!DONE!

mean      = 0.5922833599999999
stddev    = 0.002857191147877615
95% confidence interval:
confidenceLo = 0.5914913870195623
confidenceHi = 0.5930753329804376
```

对大小为 200 的网格进行 500 次模拟，结果如下

```
threshold[527](WeightedQuickUnionUF) = 0.591900 SpendTime: 0.015025
threshold[490](WeightedQuickUnionUF) = 0.584375 SpendTime: 0.046875
threshold[453](WeightedQuickUnionUF) = 0.594325 SpendTime: 0.031250
threshold[498](WeightedQuickUnionUF) = 0.607050 SpendTime: 0.046875
threshold[479](WeightedQuickUnionUF) = 0.580650 SpendTime: 0.031250
threshold[478](WeightedQuickUnionUF) = 0.598050 SpendTime: 0.031250
threshold[482](WeightedQuickUnionUF) = 0.584750 SpendTime: 0.015625
threshold[486](WeightedQuickUnionUF) = 0.591150 SpendTime: 0.031250
threshold[483](WeightedQuickUnionUF) = 0.579750 SpendTime: 0.031250

ALL THREAD FINISHED!! DONE!DONE!DONE!

mean      = 0.5930656000000005
stddev    = 0.009565491042165874
95% confidence interval:
confidenceLo = 0.5922271477422294
confidenceHi = 0.5939040522577717

Process finished with exit code 0
```

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

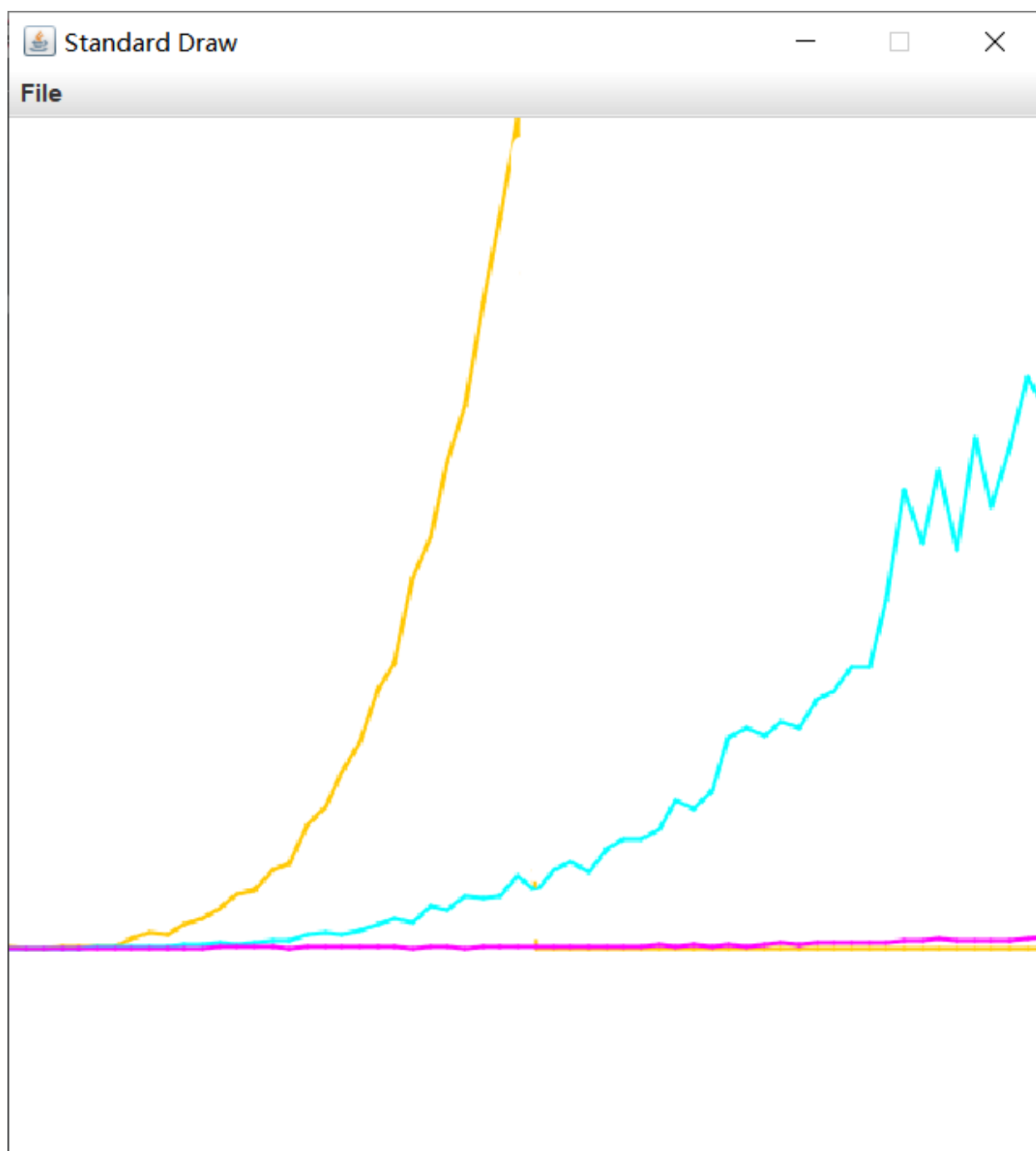
----- RUN: 20/30 -----
----- size: 333 -----
Thread:20 Use:QuickFindUF added into ThreadList
threshold[57](QuickFindUF) = 0.602855 SpendTime: 1.953125
Thread:20 Use:QuickUnionUF added into ThreadList
threshold[58](QuickUnionUF) = 0.599969 SpendTime: 0.187500
Thread:20 Use:WeightedQuickUnionUF added into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

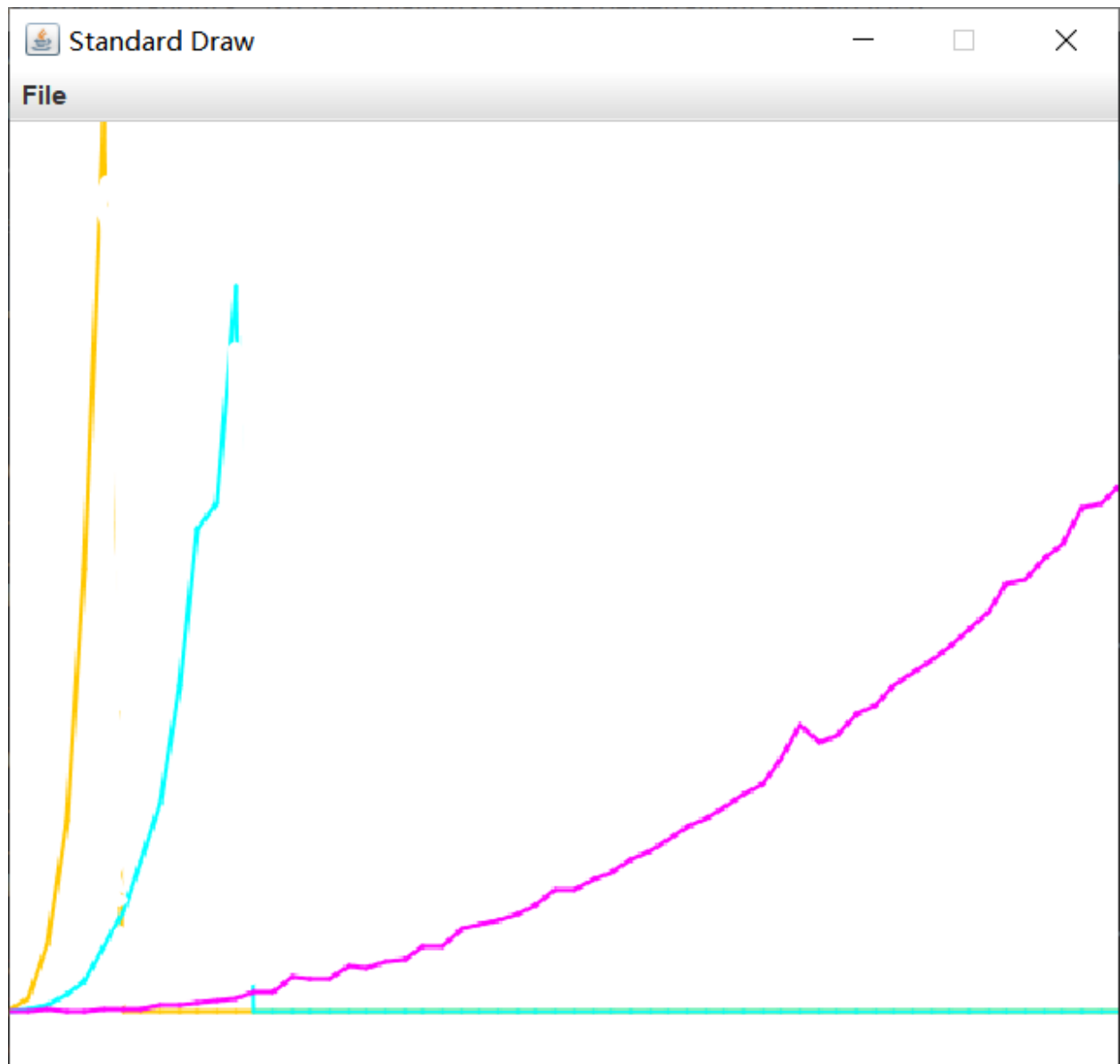
----- RUN: 22/30 -----
----- size: 366 -----
Thread:22 Use:QuickFindUF added into ThreadList
threshold[63](QuickFindUF) = 0.582512 SpendTime: 2.593750
Thread:22 Use:QuickUnionUF added into ThreadList

```

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

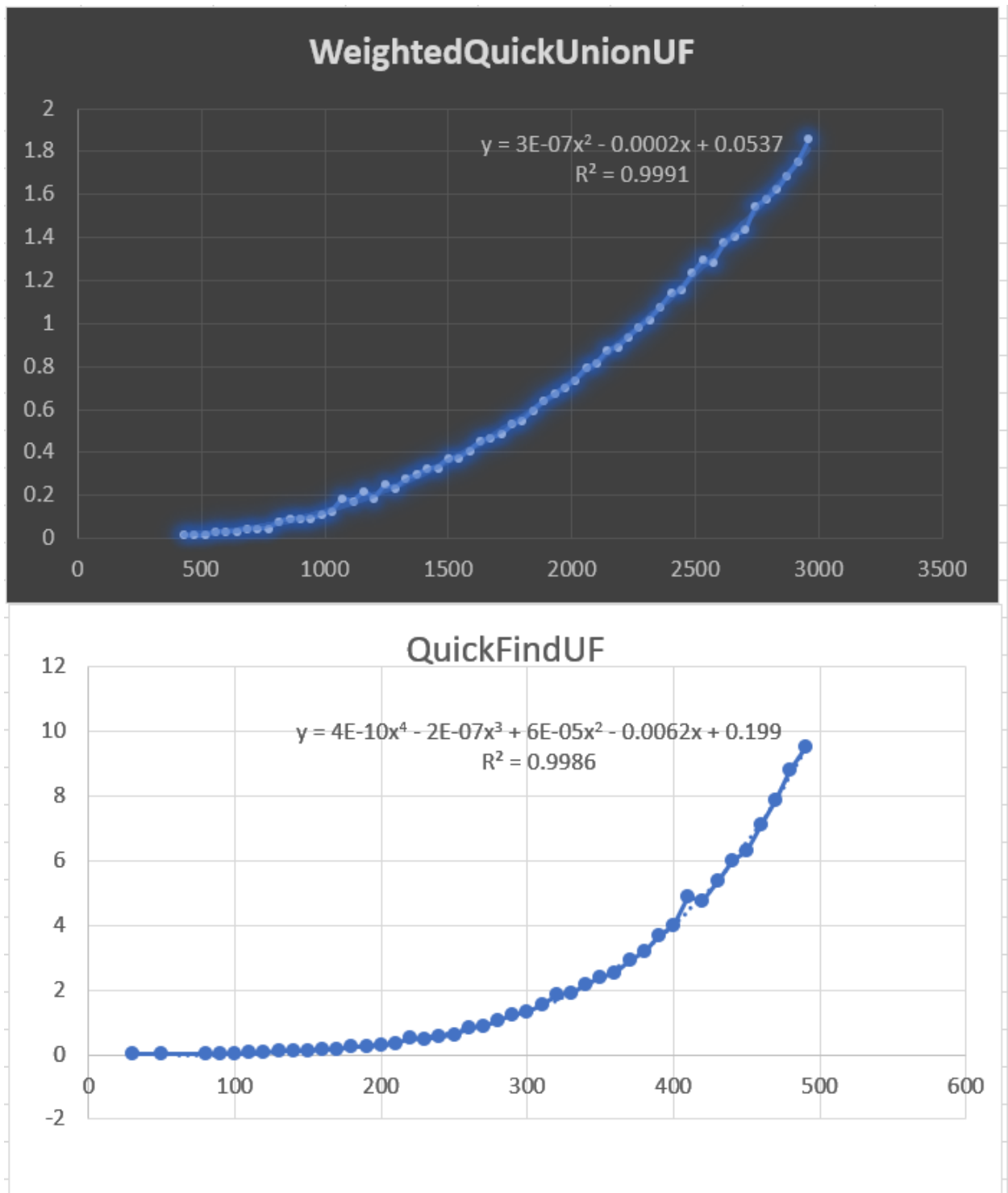


其中，橙色为QuickFindUF的运行时间，蓝色为QuickUnionUF 云香精时间，玫红色为WeightedQuickUnionUF 运行时间

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.*;
```

```

2
3
4 public class Percolation {
5     private UF uf;
6     private int N;
7     private boolean isopen[][];
8     private boolean isfulled[][];
9     boolean isvisited[][];
10    public Painter painter;
11    private boolean paint;
12
13    private int getUFId(int x, int y){
14        return (y*N + x + 1);
15    }
16
17    /**
18     * 初始化大小为 N 的可渗透区域
19     * @param N 渗透区域大小
20     */
21    public Percolation(int N, String ufType, boolean paint){
22        this.N = N;
23        this.paint = paint;
24        uf = new UF(N*N+2, ufType);
25
26        for (int i = 0; i < N; i++){
27            uf.union(0, getUFId(i, 0));
28        }
29        for (int i = 0; i < N; i++){
30            uf.union(N*N+1, getUFId(i, N-1));
31        }
32
33        isopen = new boolean[N][N];
34
35        for (int i = 0; i < N; i++){
36            for (int j = 0; j < N; j++){
37                isopen[i][j] = false;
38            }
39        }
40
41        if (paint){
42            isfulled = new boolean[N][N];
43            for (int i = 0; i < N; i++){
44                for (int j = 0; j < N; j++){
45                    isfulled[i][j] = false;
46                }
47            }
48            painter = new Painter(N);
49            isvisited = new boolean[N][N];
50        }
51    }
52
53    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

```

```

108  /**
109   * 计算本次渗透阈值
110   *
111   * @return double 阈值
112   */
113  public double threshold(){
114      return (double)this.openSize()/(N*N);
115  }
116
117  public int openSize(){
118      int num = 0;
119      for(int i = 0;i<N;i++){
120          for (int j = 0; j<N;j++){
121              if(isopen(i,j)){
122                  num++;
123              }
124          }
125      }
126      return num;
127  }
128
129  private void bfsIsFull(int x, int y){
130      if (!isFull(x,y)) return;
131      for (int i = 0; i<N;i++){
132          for (int j = 0; j<N;j++){
133              isvisited[i][j] = false;
134          }
135      }
136
137      Queue<Node> queue = new Queue<>();
138      queue.enqueue(new Node(x,y));
139      while (!queue.isEmpty()){
140          Node n = queue.dequeue();
141          painter.printFull(n.x,n.y);
142          isfulled[n.x][n.y] = true;
143          int dx[] = {0,0,-1,1};
144          int dy[] = {1,-1,0,0};
145          for (int i = 0; i<4;i++){
146              int nx = n.x +dx[i];
147              int ny = n.y +dy[i];
148              if (isopen(nx,ny) && ! isvisited[nx][ny] && (!isfulled[nx][ny])){
149                  isvisited[nx][ny] = true;
150                  queue.enqueue(new Node(nx,ny));
151              }
152          }
153      }
154  }
155
156  private class Node{
157      int x, y;
158      public Node(int x, int y){
159          this.x = x;
160          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 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。

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

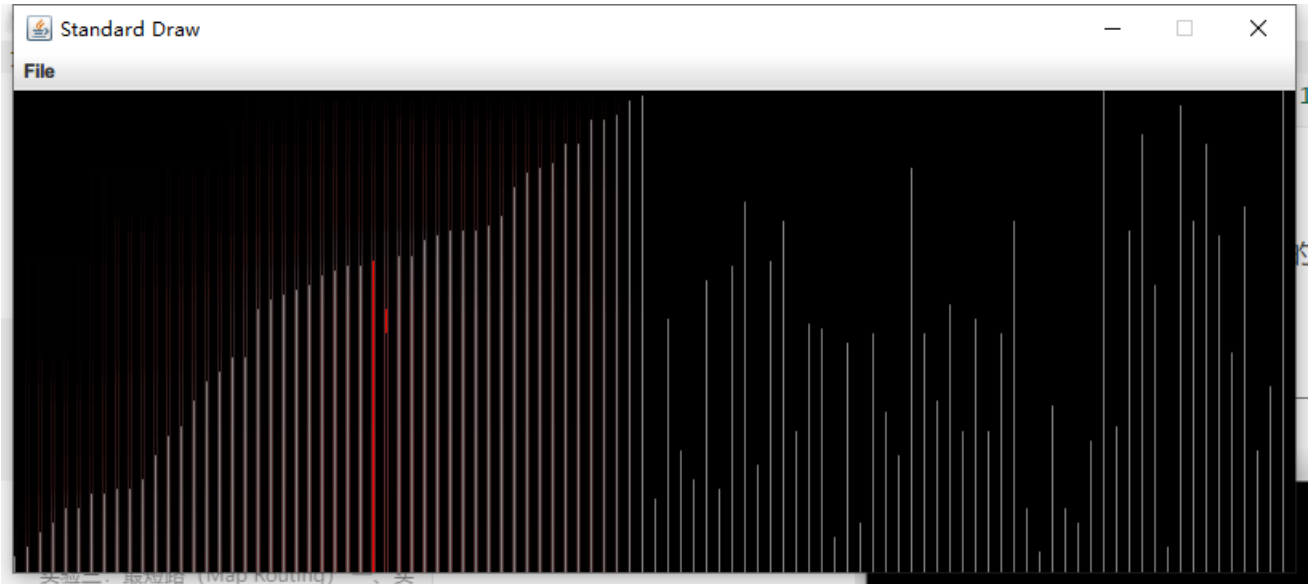
1. 几种排序的实现

- 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
- 编写 `generateRandom()` 函数，实现产生指定大小的随机数组功能，用于排序
- 使用 `LinkedHashMap` 数据结构，对每次排序的时间和空间开销进行记录
- 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。

```
1      Comparable[] tlist = list.clone();
2      LinkedHashMap<String, Double> timeResult = new LinkedHashMap<>();
3      LinkedHashMap<String, Double> memResult = new LinkedHashMap<>();
4
5      Stopwatch w1 = new Stopwatch();
6      Runtime.getRuntime().gc(); //空间回收
7      long MemoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
8      Insertion.sort(tlist);
9      long MemoryNow = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
10     timeResult.put("IS", w1.elapsedTime());
11     memResult.put("IS", 1.0 * (MemoryNow - MemoryBefore) / 1000);
```

2. 排序算法的可视化

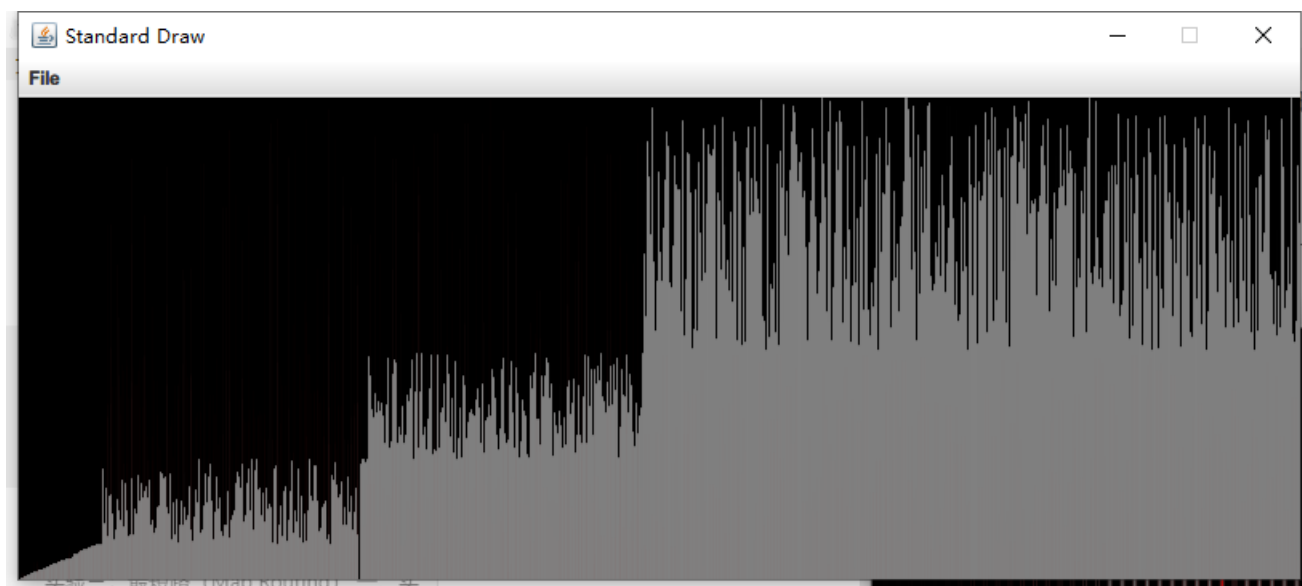
在实现了不同算法的时间空间开销记录的基础上，编写 `DrawGraph` 函数，可以将排序算法的执行过程可视化输出
插入排序可视化中间过程：



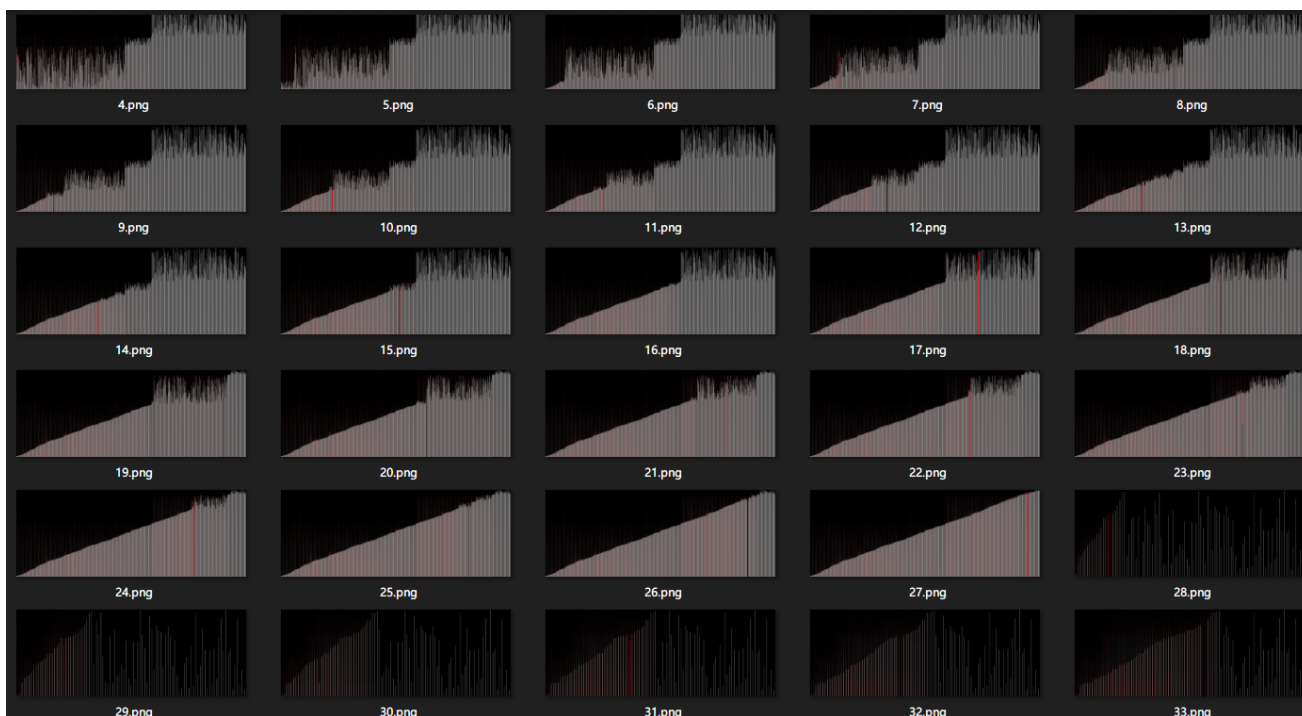
归并排序 1000 个数据元素 可视化中间过程：



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

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

```
1  import edu.princeton.cs.algs4.Insertion;
2  import edu.princeton.cs.algs4.Merge;
3  import edu.princeton.cs.algs4.Quick;
4  import edu.princeton.cs.algs4.Quick3way;
5  import edu.princeton.cs.algs4.MergeBU;
6
7  import edu.princeton.cs.algs4.StdRandom;
8
9  import java.util.LinkedHashMap;
10
11
12  public class SortTest {
13      public static Integer[] randomList;
14      /**
15       * 产生 size 大小的随机数组
16       * @param size
17       */
18      public static Integer[] generateRandom(int size){
19          randomList = new Integer[size];
20          for (int j = 0; j<size;j++){
21              randomList[j] = StdRandom.uniform(size);
22          }
23          StdRandom.shuffle(randomList);
24          return randomList;
25      }
26
27      /**
28       * 对 list 数组元素进行一次排序测试
29       * @param list
30       */
31      public static LinkedHashMap[] runTest(Comparable[] list){
32          Comparable[] tlist = list.clone();
33          LinkedHashMap<String,Double> timeResult = new LinkedHashMap<>();
34          LinkedHashMap<String,Double> memResult = new LinkedHashMap<>();
35
36          Stopwatch w1 = new Stopwatch();
37          Runtime.getRuntime().gc(); //空间回收
38          long MemoryBefore = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
39          Insertion.sort(tlist);
40          long MemoryNow = Runtime.getRuntime().totalMemory()-
Runtime.getRuntime().freeMemory();
41          timeResult.put("IS",w1.elapsedTime());
```

```

42         memResult.put("IS", 1.0 * (MemoryNow - MemoryBefore) / 1000);
43
44
45         tlist = list.clone();
46         Stopwatch w2 = new Stopwatch();
47         Runtime.getRuntime().gc(); //空间回收
48         MemoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
49         Merge.sort(tlist);
50         MemoryNow = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
51         timeResult.put("TDM", w2.elapsedTime());
52         memResult.put("TDM", 1.0 * (MemoryNow - MemoryBefore) / 1024);
53
54         tlist = list.clone();
55         Stopwatch w3 = new Stopwatch();
56         Runtime.getRuntime().gc(); //空间回收
57         MemoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
58         MergeBU.sort(tlist);
59         MemoryNow = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
60         timeResult.put("BUM", w3.elapsedTime());
61         memResult.put("BUM", 1.0 * (MemoryNow - MemoryBefore) / 1024);
62
63         tlist = list.clone();
64         Stopwatch w4 = new Stopwatch();
65         Runtime.getRuntime().gc(); //空间回收
66         MemoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
67         Quick.sort(tlist);
68         MemoryNow = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
69         timeResult.put("RQ", w4.elapsedTime());
70         memResult.put("RQ", 1.0 * (MemoryNow - MemoryBefore) / 1024);
71
72         tlist = list.clone();
73         Stopwatch w5 = new Stopwatch();
74         Runtime.getRuntime().gc(); //空间回收
75         MemoryBefore = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
76         Quick3way.sort(tlist);
77         MemoryNow = Runtime.getRuntime().totalMemory() -
Runtime.getRuntime().freeMemory();
78         timeResult.put("DQ3P", w5.elapsedTime());
79         memResult.put("DQ3P", 1.0 * (MemoryNow - MemoryBefore) / 1024);
80
81         LinkedHashMap[] RESULT = new LinkedHashMap[2];
82         RESULT[0] = timeResult;
83         RESULT[1] = memResult;
84         return RESULT;
85     }
86

```

```

87     public void runTests(int size){
88
89         TestResult rec = new TestResult();
90         for(int i = 0; i<10;i++){
91             generateRandom(size);
92             rec.record(runTest(randomList));
93         }
94         rec.printResult();
95     }
96
97 }

```

```

1  import edu.princeton.cs.algs4.StdOut;
2  import edu.princeton.cs.algs4.StdStats;
3
4  import java.util.ArrayList;
5  import java.util.Map;
6  import java.util.LinkedHashMap;
7
8
9  public class TestResult {
10     LinkedHashMap<String, ArrayList<Double>> timeResults = new LinkedHashMap<>();
11     LinkedHashMap<String, ArrayList<Double>> spaceResults = new LinkedHashMap<>();
12     int testTimes;
13
14
15     /**
16      * 记录一次测试结果
17      */
18     public void recordTimeResult(LinkedHashMap<String,Double> result){
19         recordResult(result, timeResults);
20     }
21     /**
22      * 记录一次测试结果
23      */
24     public void recordSpaceResult(LinkedHashMap<String,Double> result){
25         recordResult(result, spaceResults);
26     }
27
28     private void recordResult(LinkedHashMap<String, Double> result,
29 LinkedHashMap<String, ArrayList<Double>> timeResults) {
30         if (timeResults.isEmpty()){
31             for (Map.Entry<String, Double> entry : result.entrySet()) {
32                 ArrayList<Double> time = new ArrayList<>();
33                 time.add(entry.getValue());
34                 timeResults.put(entry.getKey(),time);
35             }
36         }
37         else {
38             for (Map.Entry<String, Double> entry : result.entrySet()) {
39                 timeResults.get(entry.getKey()).add(entry.getValue());
40             }
41         }
42     }
43 }

```

```

41     }
42
43     public void record(LinkedHashMap<String, Double>[] result){
44         recordTimeResult(result[0]);
45         recordSpaceResult(result[1]);
46     }
47
48     /**
49      * 打印测试结果
50      */
51     public void printResult(){
52         StdOut.println("TIME RESULTS \t\t (ms)");
53         prs(timeResults);
54         StdOut.println("SPACE RESULTS \t\t (KB)");
55         prs(spaceResults);
56     }
57
58
59     private void prs(LinkedHashMap<String, ArrayList<Double>> spaceResults) {
60         for (Map.Entry<String, ArrayList<Double>> entry : spaceResults.entrySet())
61         {
62             StdOut.printf("%4s\t\t",entry.getKey());
63             double[] entryTime = new double[entry.getValue().size()];
64             int i = 0;
65             for (Double time : entry.getValue()){
66                 StdOut.printf("% 4.2f\t",time);
67                 entryTime[i++] = time;
68             }
69             StdOut.println(StdStats.mean(entryTime));
70         }
71     }
72

```

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

```
1 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

```
1 private void dijkstra(int s, int d) {
2     int V = G.V();
3
4     // initialize
5     dist = new double[V];
6     pred = new int[V];
7     for (int v = 0; v < V; v++) dist[v] = INFINITY;
8     for (int v = 0; v < V; v++) pred[v] = -1;
9
10    // priority queue
11    IndexPQ pq = new IndexPQ(V);
12    for (int v = 0; v < V; v++) pq.insert(v, dist[v]);
13
14    // set distance of source
15    dist[s] = 0.0;
```

```

16     pred[s] = s;
17     pq.change(s, dist[s]);
18
19     // run Dijkstra's algorithm
20     while (!pq.isEmpty()) {
21         int v = pq.delMin();
22         //// System.out.println("process " + v + " " + dist[v]);
23
24         // v not reachable from s so stop
25         if (pred[v] == -1) break;
26
27         // scan through all nodes w adjacent to v
28         IntIterator i = G.neighbors(v);
29         while (i.hasNext()) {
30             int w = i.next();
31             if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
32                 dist[w] = dist[v] + G.distance(v, w);
33                 pq.change(w, dist[w]);
34                 pred[w] = v;
35                 //// System.out.println("    lower " + w + " to " + dist[w]);
36             }
37         }
38     }
39 }

```

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

```

Run: Distances x
131.02217547544702
from 76528 to 76540
128.4818387681604
from 73702 to 73713
1.0
from 78607 to 78619
82.20842255298763
=====
1000long Time: 50.399
5000short Time: 239.644

Process finished with exit code 0
Terminal 4: Run 5: Debug 6: TODO

```

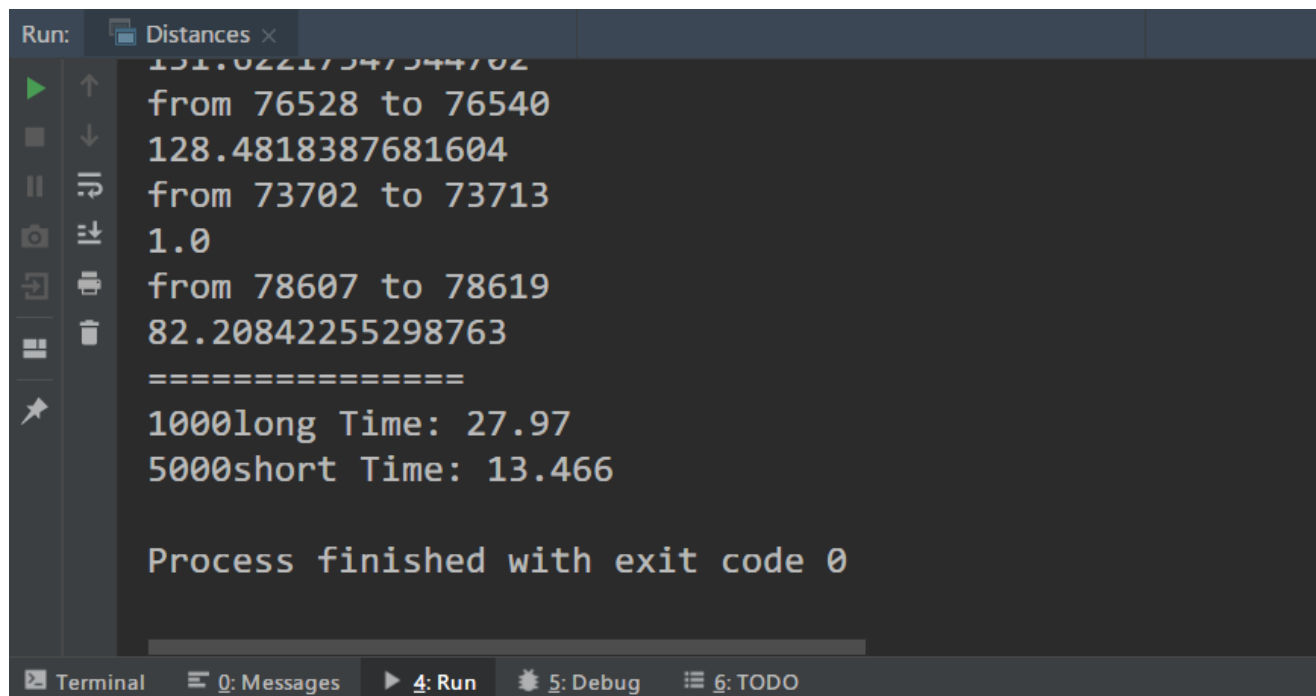
使用该算法完成 1000 条长路径和 5000 条短路径所用的总时间达到了 289s

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

```
1 while (!pq.isEmpty()) {
2     int v = pq.delMin();
3     if (v == d) break;
4     ...
5 }
```

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 `dijkstra` 对象的构造函数中执行：

```
1 public Dijkstra(EuclideanGraph G) {
2     this.G = G;
3     int v = G.V();
4     // initialize
5     dist = new double[v];
6     pred = new int[v];
7     for (int v = 0; v < V; v++) dist[v] = INFINITY;
8     for (int v = 0; v < V; v++) pred[v] = -1;
9
10    // priority queue
11    pq = new IndexPQ(V);
12    for (int v = 0; v < V; v++) pq.insert(v, dist[v]);
13 }
```

```
13  
14 }
```

修改 dijkstra 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 }
```



```
Run: Distances x
FROM 86989 TO 87088
131.62217547544702
from 76528 to 76540
128.4818387681604
from 73702 to 73713
1.0
from 78607 to 78619
82.20842255298763
=====
1000long Time: 25.858
5000short Time: 7.729

Process finished with exit code 0
```

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 dijkstra 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：

```
1 while (i.hasNext()) {
2     int w = i.next();
3     double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v, d);
4     if (wt[v] + dt < wt[w] - EPSILON) {
5         //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
6             dist[w] = dist[v] + G.distance(v, w);
7             wt[w] = wt[v] + dt;
8             //G.point(v).drawTo(G.point(w));
9             //pq.change(w, wt[w]);
10            mpq.changeKey(w, wt[w]);
11            pred[w] = v;
12            changed.add(w);
13            //Thread.sleep(2);
14            //// System.out.println("    lower " + w + " to " + dist[w]);
15        }
16    }
```

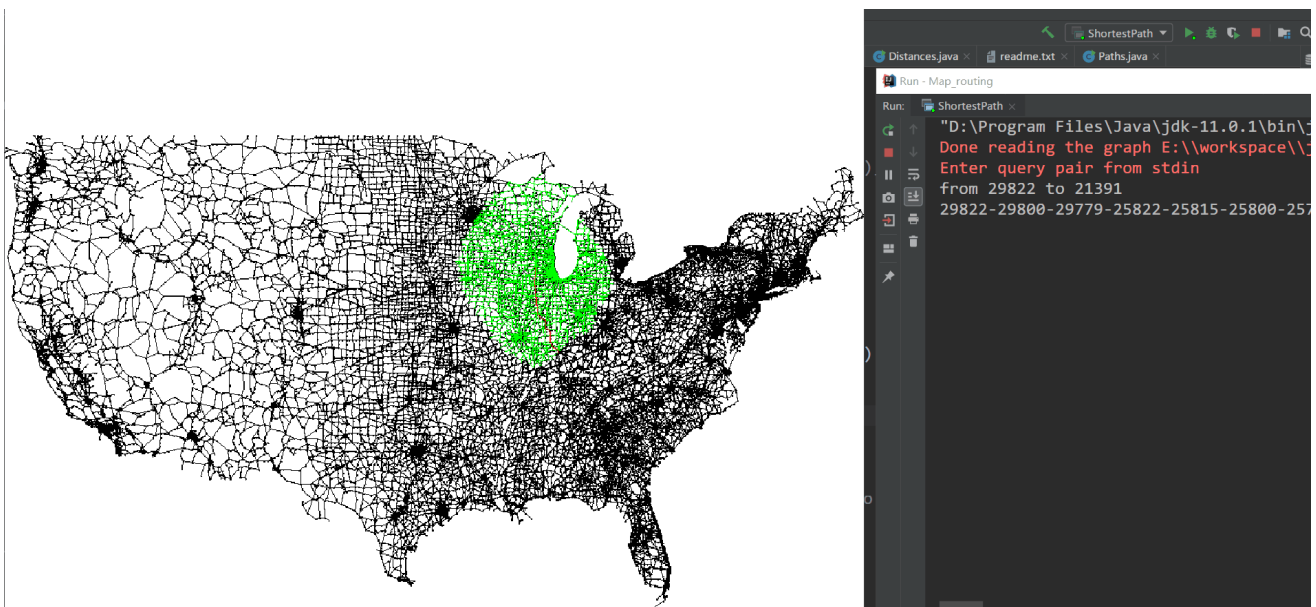
修改之后，重新计算完成 1000 条长路径和 5000 条短路径所用的总时间

```
Run - Map_routing
Distances x
532.1858141541555
from 4768 to 4813
355.467936053822
from 73918 to 73994
54.703838940173064
from 10419 to 10420
0.0
from 80074 to 80090
79.58139675849544
from 86989 to 87088
131.62217547544702
from 76528 to 76540
128.4818387681604
from 73702 to 73713
1.0
from 78607 to 78619
82.20842255298763
=====
1000long Time: 12.376
5000short Time: 2.445

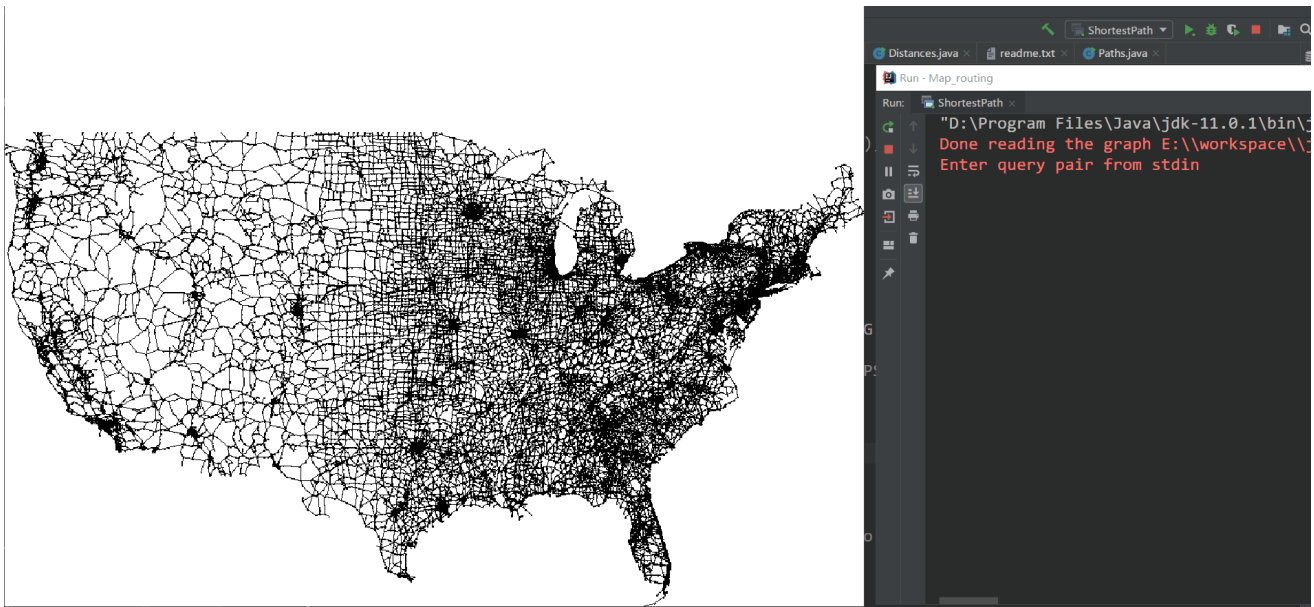
Process finished with exit code 0
```

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

```
Run - Map_routing
Run: Distances x
532.1858141541555
from 4768 to 4813
355.467936053822
from 73918 to 73994
54.703838940173064
from 10419 to 10420
0.0
from 80074 to 80090
79.58139675849544
from 86989 to 87088
131.62217547544702
from 76528 to 76540
128.4818387681604
from 73702 to 73713
1.0
from 78607 to 78619
82.20842255298763
=====
1000long Time: 12.376
5000short Time: 2.445

Process finished with exit code 0
```

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

```
1  /*****
2  *   Dijkstra's algorithm.
3  *
4  *****/
5
6  import edu.princeton.cs.algs4.StdOut;
7
8  import java.awt.*;
9  import java.util.LinkedList;
10 //import edu.princeton.cs.algs4.IndexMultiwayMinPQ;
11
12
13
14 public class Dijkstra {
15     private static double INFINITY = Double.MAX_VALUE;
16     private static double EPSILON = 0.000001;
17
18     private EuclideanGraph G;
19     private double[] dist;
20     private int[] pred;
21     private double[] wt;
22     IndexPQ pq;
23     IndexMultiwayMinPQ mpq;
24
25     private LinkedList<Integer> changed = new LinkedList<Integer>();
26
27     public Dijkstra(EuclideanGraph G) {
28         this.G = G;
29         int V = G.V();
30         // initialize
31         dist = new double[V];
32         pred = new int[V];
33         wt = new double[V];
34         for (int v = 0; v < V; v++) dist[v] = INFINITY;
35         for (int v = 0; v < V; v++) wt[v] = INFINITY;
36         for (int v = 0; v < V; v++) pred[v] = -1;
37
38         // priority queue
39         pq = new IndexPQ(V);
40         for (int v = 0; v < V; v++) pq.insert(v, dist[v]);
41         mpq = new IndexMultiwayMinPQ<Double>(V,6);
42         for (int v = 0; v < V; v++) mpq.insert(v, INFINITY);
43
44     }
45
46     // return shortest path distance from s to d
47     public double distance(int s, int d) throws InterruptedException {
48         dijkstra(s, d);
49         return dist[d];
50     }
51 }
```

```

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.setStroke(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.setStroke(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();

```

```

103         dist[i] = INFINITY;
104         wt[i] = INFINITY;
105         pred[i] = -1;
106         //StdOut.println(i);
107         mpq.changeKey(i, INFINITY);
108     }
109
110     //mpq = new IndexMultiwayMinPQ<Double>(V,4);
111     //for (int v = 0; v < V; v++) mpq.insert(v, INFINITY);
112
113     // set distance of source
114     dist[s] = 0.0;
115     wt[s] = 0.0;
116     pred[s] = s;
117     changed.add(s);
118     //pq.change(s, wt[s]);
119     mpq.changeKey(s, wt[s]);
120
121     // run Dijkstra's algorithm
122     while (!mpq.isEmpty()) {
123         //int v = pq.delMin();
124         int v = mpq.delMin();
125         if (v == d) break;
126         //// System.out.println("process " + v + " " + dist[v]);
127
128         // v not reachable from s so stop
129         if (pred[v] == -1) break;
130
131         // scan through all nodes w adjacent to v
132         IntIterator i = G.neighbors(v);
133         while (i.hasNext()) {
134             int w = i.next();
135             double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v,
d);
136
137             if (wt[v] + dt < wt[w] - EPSILON) {
138                 //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
139                 dist[w] = dist[v] + G.distance(v, w);
140                 wt[w] = wt[v] + dt;
141                 //G.point(v).drawTo(G.point(w));
142                 //pq.change(w, wt[w]);
143                 mpq.changeKey(w, wt[w]);
144                 pred[w] = v;
145                 changed.add(w);
146                 //Thread.sleep(2);
147                 //// System.out.println("    lower " + w + " to " + dist[w]);
148             }
149         }
150     }
151
152     private void dijkstra(int s, int d, boolean draw) throws InterruptedException {
153
154         int v = G.V();

```

```

155     mpq.n = v;
156     mpq.nmax = v;
157     while (!changed.isEmpty()){
158         int i = changed.removeFirst();
159         dist[i] = INFINITY;
160         wt[i] = INFINITY;
161         pred[i] = -1;
162         mpq.changeKey(i, INFINITY);
163     }
164
165     //pq = new IndexPQ(V);
166     //for (int v = 0; v < V; v++) pq.insert(v, INFINITY);
167
168     // set distance of source
169     dist[s] = 0.0;
170     wt[s] = 0.0;
171     pred[s] = s;
172     changed.add(s);
173     mpq.changeKey(s, wt[s]);
174
175     // run Dijkstra's algorithm
176     while (!mpq.isEmpty()) {
177         int v = mpq.delMin();
178         if (v == d) break;
179         //// System.out.println("process " + v + " " + dist[v]);
180
181         // v not reachable from s so stop
182         if (pred[v] == -1) break;
183
184         // scan through all nodes w adjacent to v
185         IntIterator i = G.neighbors(v);
186         while (i.hasNext()) {
187             int w = i.next();
188             double dt = G.distance(v, w) + G.distance(w, d) - G.distance(v,
d);
189
190             if (wt[v] + dt < wt[w] - EPSILON) {
191                 //if (dist[v] + G.distance(v, w) < dist[w] - EPSILON) {
192                 dist[w] = dist[v] + G.distance(v, w);
193                 wt[w] = wt[v] + dt;
194                 Turtle.setColor(Color.blue);
195                 //drawPath(s,v);
196                 //drawPath(s,d);
197                 Turtle.setColor(Color.green);
198                 G.point(v).drawTo(G.point(w));
199                 Turtle.render();
200                 mpq.changeKey(w, wt[w]);
201                 pred[w] = v;
202                 changed.add(w);
203
204                 //// System.out.println("    lower " + w + " to " + dist[w]);
205             }
206         }
207     }
208     //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。

```
1  std::ifstream ifstream("D:\\alice29.txt");
2  std::stringstream stream;
3  ifstream.seekg(0, std::ios::end);    // go to the end
4  int length = ifstream.tellg();       // report location (this is the length)
5  std::cout << "Input File length: " << length << std::endl;
6  ifstream.seekg(0, std::ios::beg);    // go back to the beginning
7  ifstream.read(str, length);          // read the whole file into the buffer
8  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

```
1  void suffixSort(int n){
2      //sort suffixes according to their first characters
3      for (int i=0; i<n; ++i){
4          pos[i] = i;
5      }
6      std::sort(pos, pos + n, smaller_first_char);
7      //{pos contains the list of suffixes sorted by their first character}
8
9      for (int i=0; i<n; ++i){
10         bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
11         b2h[i] = false;
```

```

12     }
13
14     for (int h = 1; h < n; h <= 1){
15         //{bh[i] == false if the first h characters of pos[i-1] == the first h
characters of pos[i]}
16         int buckets = 0;
17         for (int i=0, j; i < n; i = j){
18             j = i + 1;
19             while (j < n && !bh[j]) j++;
20             next[i] = j;
21             buckets++;
22         }
23         if (buckets == n) break; // We are done! Lucky bastards!
24         //{suffixes are separted in buckets containing strings starting with the same h
characters}
25
26         for (int i = 0; i < n; i = next[i]){
27             cnt[i] = 0;
28             for (int j = i; j < next[i]; ++j){
29                 rank[pos[j]] = i;
30             }
31         }
32
33         cnt[rank[n - h]]++;
34         b2h[rank[n - h]] = true;
35         for (int i = 0; i < n; i = next[i]){
36             for (int j = i; j < next[i]; ++j){
37                 int s = pos[j] - h;
38                 if (s >= 0){
39                     int head = rank[s];
40                     rank[s] = head + cnt[head]++;
41                     b2h[rank[s]] = true;
42                 }
43             }
44             for (int j = i; j < next[i]; ++j){
45                 int s = pos[j] - h;
46                 if (s >= 0 && b2h[rank[s]]){
47                     for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
48                 }
49             }
50         }
51         for (int i=0; i<n; ++i){
52             pos[rank[i]] = i;
53             bh[i] |= b2h[i];
54         }
55     }
56     for (int i=0; i<n; ++i){
57         rank[pos[i]] = i;
58     }
59 }
60 void getHeight(int n){
61     for (int i=0; i<n; ++i) rank[pos[i]] = i;
62     height[0] = 0;

```

```

63     for (int i=0, h=0; i<n; ++i){
64         if (rank[i] > 0){
65             int j = pos[rank[i]-1];
66             while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
67             height[rank[i]] = h;
68             if (h > 0) h--;
69         }
70     }
71 }

```

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

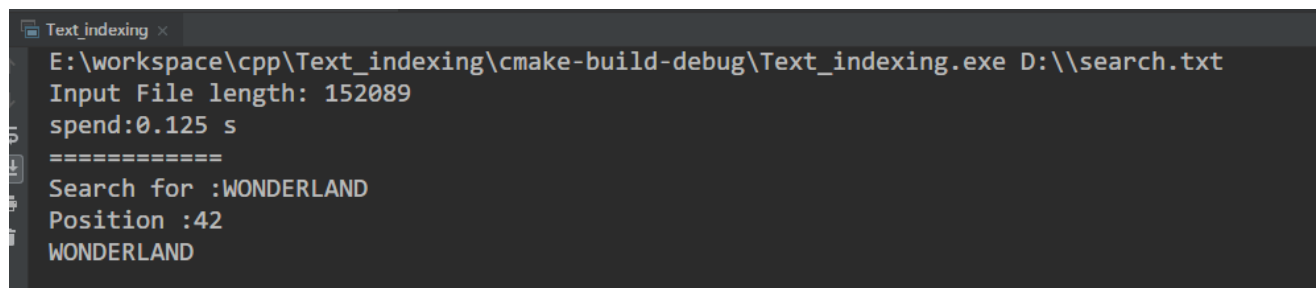
```

1  int binarychop(char* key, int key_length, int left, int right){
2      if (left > right){
3          return -1;
4      }
5      int mid = (right-left)/2+left;
6      int p = pos[mid];
7      for (int i = 0; i < key_length; i++){
8          if (key[i]<str[p+i]){
9              return binarychop(key,key_length,left,mid-1);
10         } else if (key[i] > str[p+i]){
11             return binarychop(key,key_length,mid+1,right);
12         }
13     }
14     return p;
15 }

```

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



```

Text_indexing x
E:\workspace\cpp\Text_indexing\cmake-build-debug\Text_indexing.exe D:\search.txt
Input File length: 152089
spend:0.125 s
=====
Search for :WONDERLAND
Position :42
WONDERLAND

```

```
Text_indexing x
=====
Search for :chain
Position :697
chain would be worth the trouble
of getting up and picking t
=====
Search for :would
Position :106466
would cost them their lives.

    All the time they were playi
=====
Search for :be
Position :57817
believe it,' said the Pigeon; `but if they do, why
then they
=====
Search for :of
Position :40577
of the window, I only wish they COULD! I'm sure I
don't wan
=====
Search for :the
```

附：实验完整代码

```
1  #include <iostream>
2  #include <algorithm>
3  #include <sstream>
4  #include <cstring>
5  #include <fstream>
6  #include <ctime>
7
8  const int N = 2000000;
9
10 char str[N]; //input
11 int rank[N], pos[N]; //output
12 int cnt[N], next[N]; //internal
13 bool bh[N], b2h[N];
14
```

```

15 // Compares two suffixes according to their first characters
16 bool smaller_first_char(int a, int b){
17     return str[a] < str[b];
18 }
19
20 void suffixSort(int n){
21     //sort suffixes according to their first characters
22     for (int i=0; i<n; ++i){
23         pos[i] = i;
24     }
25     std::sort(pos, pos + n, smaller_first_char);
26     //{pos contains the list of suffixes sorted by their first character}
27
28     for (int i=0; i<n; ++i){
29         bh[i] = i == 0 || str[pos[i]] != str[pos[i-1]];
30         b2h[i] = false;
31     }
32
33     for (int h = 1; h < n; h <= 1){
34         //{bh[i] == false if the first h characters of pos[i-1] == the first h
characters of pos[i]}
35         int buckets = 0;
36         for (int i=0, j; i < n; i = j){
37             j = i + 1;
38             while (j < n && !bh[j]) j++;
39             next[i] = j;
40             buckets++;
41         }
42         if (buckets == n) break;
43         //{suffixes are parted in buckets containing strings starting with the same
h characters}
44
45         for (int i = 0; i < n; i = next[i]){
46             cnt[i] = 0;
47             for (int j = i; j < next[i]; ++j){
48                 rank[pos[j]] = i;
49             }
50         }
51
52         cnt[rank[n - h]]++;
53         b2h[rank[n - h]] = true;
54         for (int i = 0; i < n; i = next[i]){
55             for (int j = i; j < next[i]; ++j){
56                 int s = pos[j] - h;
57                 if (s >= 0){
58                     int head = rank[s];
59                     rank[s] = head + cnt[head]++;
60                     b2h[rank[s]] = true;
61                 }
62             }
63             for (int j = i; j < next[i]; ++j){
64                 int s = pos[j] - h;
65                 if (s >= 0 && b2h[rank[s]]){

```

```

66         for (int k = rank[s]+1; !bh[k] && b2h[k]; k++) b2h[k] = false;
67     }
68 }
69 }
70 for (int i=0; i<n; ++i){
71     pos[rank[i]] = i;
72     bh[i] |= b2h[i];
73 }
74 }
75 for (int i=0; i<n; ++i){
76     rank[pos[i]] = i;
77 }
78 }
79 // End of suffix array algorithm
80
81
82
83 int height[N];
84
85 void getHeight(int n){
86     for (int i=0; i<n; ++i) rank[pos[i]] = i;
87     height[0] = 0;
88     for (int i=0, h=0; i<n; ++i){
89         if (rank[i] > 0){
90             int j = pos[rank[i]-1];
91             while (i + h < n && j + h < n && str[i+h] == str[j+h]) h++;
92             height[rank[i]] = h;
93             if (h > 0) h--;
94         }
95     }
96 }
97 // End of longest common prefixes algorithmd
98
99 int binarychop(char* key, int key_length, int left, int right){
100     if (left > right){
101         return -1;
102     }
103     int mid = (right-left)/2+left;
104     int p = pos[mid];
105     for (int i = 0; i < key_length; i++){
106         if (key[i]<str[p+i]){
107             return binarychop(key,key_length,left,mid-1);
108         } else if (key[i] > str[p+i]){
109             return binarychop(key,key_length,mid+1,right);
110         }
111     }
112     return p;
113 }
114
115 int main(int argc, char ** argv) {
116     std::ifstream ifstream("D:\\alice29.txt");
117     std::stringstream stream;
118     ifstream.seekg(0, std::ios::end);    // go to the end

```

```

119     int length = ifstream.tellg();           // report location (this is the
length)
120     std::cout << "Input File length: " << length << std::endl;
121     ifstream.seekg(0, std::ios::beg);       // go back to the beginning
122     ifstream.read(str, length);             // read the whole file into the buffer
123     ifstream.close();                       // close file handle
124     clock_t start,end;
125     start = clock();
126     suffixSort(strlen(str));
127     end = clock();
128     std::cout<<"spend:"<< (double)(end-start)/CLOCKS_PER_SEC << " s" << std::endl;
129
130     std::ifstream search(argv[1]);
131     while (search.peek() != EOF){
132         std::cout << "======" << std::endl;
133         char key[1000];
134         search >> key;
135         search.get();
136         std::cout << "Search for :" << key << std::endl;
137         int p = binarychop(key, strlen(key),0,strlen(str));
138         std::cout << "Position :" << p << "    " << std::endl;
139         for (int i = 0 ; i < 60 ; i++){
140             std::cout << str[p+i] ;
141         }
142         std::cout << std::endl;
143     }
144
145     return 0;
146 }

```