《算法分析与设计实验报告》

18030400014-魏红旭

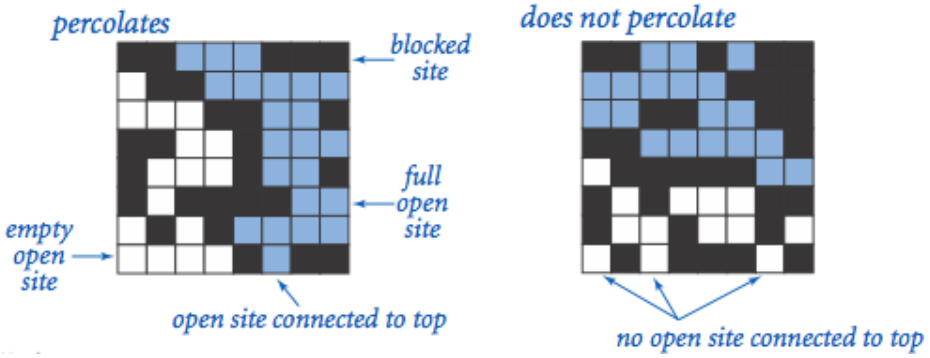
实验一：渗透问题（Percolation）

一、实验目的

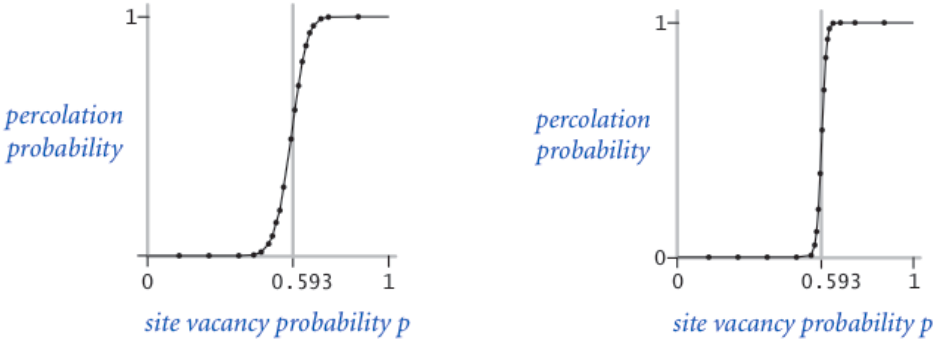
使用合并-查找（union-find）数据结构，编写程序通过蒙特卡罗模拟（Monte Carlo simulation）来估计渗透阈值的值。

二、题目描述

我们使用 N*N 网格点来模型一个渗透系统。每个格点或是 open 格点或是 blocked 格点。一个 full site 是一个 open 格点，它可以通过一连串的邻近（左，右，上，下）open 格点连通到顶行的一个 open 格点。如果在底行中有一个 full site 格点，则称系统是渗透的。（对于绝缘/金属材料的例子，open 格点对应于金属材料，渗透系统有一条从顶行到底行的金属路径，且 full sites 格点导电。对于多孔物质示例，open 格点对应于空格，水可能流过，从而渗透系统使水充满 open 格点，自顶向下流动。）



问题：在一个著名的科学问题中，研究人员对以下问题感兴趣：如果将格点以空置概率 p 独立地设置为 open 格点（因此以概率 1-p 被设置为 blocked 格点），系统渗透的概率是多少？当 p=0 时，系统不会渗出；当 p=1 时，系统渗透。下图显示了 20*20 随机网格（左）和 100*100 随机网格（右）的格点空置概率 p 与渗滤概率。



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

三、解决方法

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

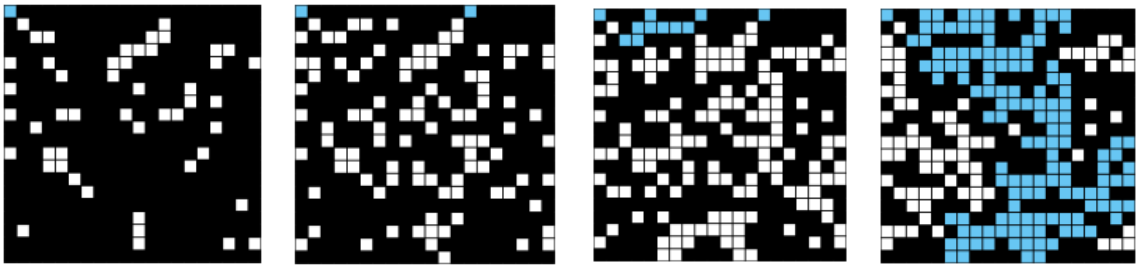
```
1.  public class Percolation {
2.      public Percolation(int N)           // create N-by-N grid
3.      public void open(int i, int j)      // open site if it is not already
4.      public boolean isOpen(int i, int j) // is site (row i, column j) open?
5.      public boolean isFull(int i, int j) // is site (row i, column j) full?
6.      public boolean percolates()         // does the system percolate?
7.      public static void main(String[] args)// test client, optional
8.  }
```

约定行 i 列 j 下标在 1 和 N 之间，其中(1,1)为左上格点位置：如果 open(),isOpen(),or isFull()不在这个规定的范围，则抛出 IndexOutOfBoundsException 例外。如果 N≤0，构造函数应该抛出 IllegalArgumentException 例外。构造函数应该与 N2 成正比。所有方法应该为常量时间加上常量次调用合并-查找方法 union(), find(), connected(), count()。

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

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

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



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

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

假设 T 足够大（例如至少 30），以下为渗滤阈值提供 95%置信区间：

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

我们创建数据类型 PercolationStats 来执行一系列计算实验，包含以下 API。

```
1.  public class PercolationStats {
2.     public PercolationStats(int N, int T)
3.     public double mean() // sample mean of percolation threshold
4.     public double stddev()// sample standard deviation
5.     public double confidenceLo() //lower bound of the 95% confidence interval
6.     public double confidenceHi() //upper bound of the 95% confidence interva
7.     public static void main(String[] args) // test client, described below
8.  }
```

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

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

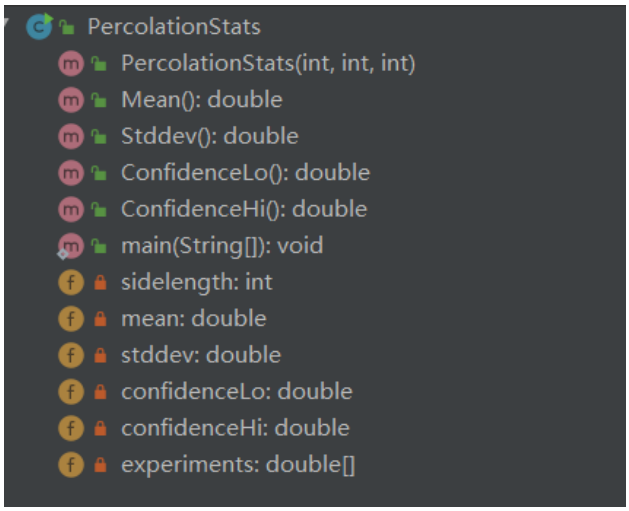
四、实验要求

- **运行时间和内存占用分析。**
  - ✧ 使用 quick-find 算法（QuickFindUF.java from algs4.jar，最好自己独立实现此算法）实现 Percolation 数据类型。进行实验表明当 N 加倍时对运行时间的影响；使用近似表示法，给出在计算机上的总时间，它是输入 N 和 T 的函数表达式。
  - ✧ 使用 weighted quick-union 算法（WeightedQuickUnionUF.java from algs4.jar，最好自己独立实现此算法）实现 Percolation 数据类型。进行实验表明当 N 加倍时对运行时间的影响；使用近似表示法，给出在计算机上的总时间，它是输入 N 和 T 的函数表达式。

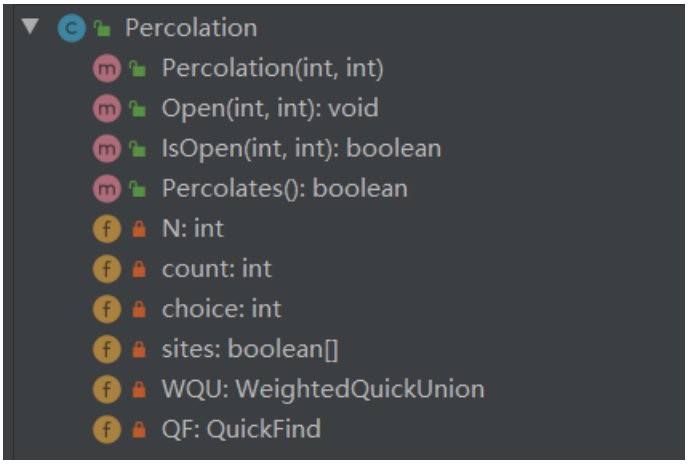
✧ 注：这个问题的实验由 Bob Sedgewick 和 Kevin Wayne 设计开发（Copyright © 2008）。更多信息可参考 http://algs4.cs.princeton.edu/home/。

五、程序设计及源代码

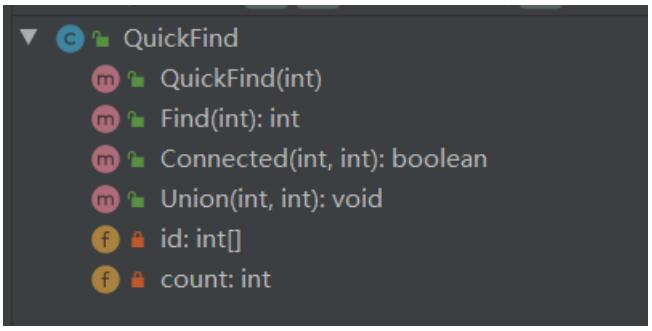
1. PercolationStats 类结构：
   a) PercolationStats(int,int,int)类构造函数，我们通过此函数来进行网格的构建以及所需实验结果的存储，例如渗透阈值的样本均值、渗透阈值的样本标准差、95%置信区间上下限等。
   b) Mean():返回渗透阈值的样本均值；
   c) Stddev():返回渗透阈值的样本标准差；
   d) ConfidenceLo():返回 95%置信区间下限；
   e) ConfidenceHi():返回 95%置信区间上限；



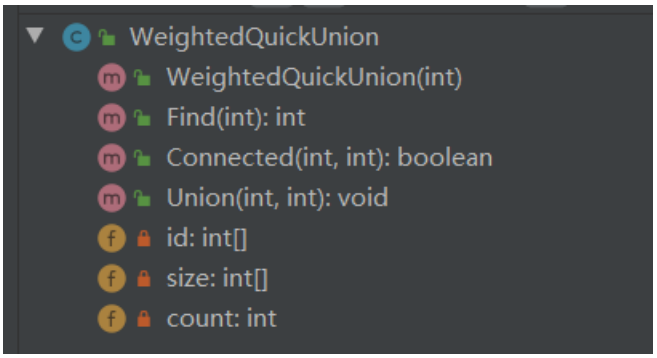
2. Percolation 类结构：
   a) Percolation(int,int)类构造函数，我们通过此函数来进行网格的初始化，初始化所有格点的状态为 blocked，除了 N*N 个格点之外，我们还需要创建两个虚拟格点，分别放在首尾部，我们通过这两个虚拟格点的连通性来判断网格是否已经发生渗透。这两个虚拟节点被初始化为 open 格点。
   b) Open(int,int):此函数来实现将随机某个格点的状态更改为 open，如果该节点在第一行，则将其与首部的虚拟格点进行连通；如果该节点在最后一行，则将其与尾部的虚拟格点进行连通；否则该格点在网格内部，连接其上下左右的 open 格点。
   c) IsOpen(int,int):此函数来返回该节点是否为 open 格点的结果；
   d) Percolates():此函数来判断首尾虚拟格点是否已经连通；



3. 独立实现 QuickFind()算法：



4. 独立实现 WeightedQuickUnion()算法：



5. 程序设计：

a) 首先我们在 PercolationStats 类中的 main 函数中，通过类对象的初始化设置网格数、迭代次数、选择 quickfind 或者 weightedquickunion 算法，并记录算法运行的开始时间；

b) PercolationStats 类构造函数：首先对特殊情况进行处理，即设置的网格数为 1 的情况，然后对一般情况的结构进行设计，我们通过

for 循环实现多次迭代，for 循环中通过 while 循环实现对随机格点的状态更改（即更改为 open 状态），当首尾虚拟节点连通时结束 while 循环，我们需要记录每次迭代 open 格点的个数，以实现对迭代空置概率 p 的计算。

- **PercolationStats 源代码：**

```
1.  package First_Experiment;
2.
3.  import edu.princeton.cs.algs4.StdRandom;
4.  import edu.princeton.cs.algs4.StdStats;
5.
6.  public class PercolationStats {
7.      private int sidelength;
8.      private double mean,stddev,confidenceLo,confidenceHi;
9.      private double[] experiments;
10.
11.     public PercolationStats(int n, int T_Iteration,int choice) {
12.         if(n <= 0 || T_Iteration <= 0)
13.             throw new IllegalArgumentException("Illegal Argument Exception")
    ;
14.         sidelength = n;
15.         if(sidelength == 1){
16.             mean = 1;            //渗透阈值的平均值
17.             stddev = Double.NaN; //渗透阈值的样本标准差
18.             confidenceLo = Double.NaN;  //95%置信区间下限
19.             confidenceHi = Double.NaN;  //95%置信区间上限
20.         }
21.         else{
22.             experiments = new double[T_Iteration]; //记录每次迭代的渗透阈值
23.             for(int i=0; i<T_Iteration; i++){      //进行 T_Iteration 次迭代
24.                 Percolation percolation = new Percolation(n,choice);
25.                 double count = 0;
26.                 while(!percolation.Percolates()) { //判断首尾是否已经连通
27.                     int row = StdRandom.uniform(n) + 1;//随机选择节点进行
    open
28.                     int col = StdRandom.uniform(n) + 1;
29.                     if(!percolation.IsOpen(row,col)){
30.                         percolation.Open(row,col);
31.                         count++;                    //记录 open 节点数量
32.                     }
33.                 }
```

```
34.                 experiments[i] = count*1.0/(n*n);   //记录此次迭代空置概率 p
35.             }
36.             mean = StdStats.mean(experiments);      //渗透阈值的样本均值
37.             stddev = StdStats.stddev(experiments);  //渗透阈值的样本标准差
38.             confidenceLo = mean - (1.96 * stddev) / Math.sqrt(T_Iteration);
    //95%置信区间下限
39.             confidenceHi = mean + (1.96 * stddev) / Math.sqrt(T_Iteration);
    //95%置信区间上限
40.         }
41.     }
42.     public double Mean() {
43.         return mean;
44.     }
45.     public double Stddev() {
46.         return stddev;
47.     }
48.     public double ConfidenceLo() {
49.         return confidenceLo;
50.     }
51.     public double ConfidenceHi() {
52.         return confidenceHi;
53.     }
54.     public static void main(String[] args) {
55.         long start_time=System.nanoTime(); //记录开始时间
56.         int n=50,T_Iteration=200;   //n 为网格数，T_Iteration 为迭代次数
57.         int choice=1;               //选择 quickfind or weighted quickunion
58.         System.out.println("----------------------------------------------
    --------");
59.         if(choice==1) {
60.             System.out.println("The algorithm currently used is:
    Quick_Find");
61.         }
62.         else if(choice==2) {
63.             System.out.println("The algorithm currently used is:
    Weighted Quick Union");
64.         }
65.         PercolationStats percolationStats = new PercolationStats(n,T_Iterati
    on,choice);
66.         long consumingtime=System.nanoTime()-start_time;  //计算总耗费时间
67.         System.out.println("Creating PercolationStats( "+n+" , "+T_Iteration
    +" )");
68.         System.out.println("mean:\t\t\t\t= " + percolationStats.Mean());
69.         System.out.println("stddev:\t\t\t\t= " + percolationStats.Stddev());
```

```
70.         System.out.println("confidence_Low:\t\t= " + percolationStats.Confid
    enceLo());
71.         System.out.println("confidence_High:\t= " + percolationStats.Confide
    nceHi());
72.         System.out.println("The cost of the time is "+consumingtime*1.0/1000
    000+"ms");
73.     }
74. }
```

- Percolation 源代码：

```
1.  package First_Experiment;
2.
3.  public class Percolation {
4.      private int N,count,choice;
5.      private boolean[] sites;
6.      private WeightedQuickUnion WQU;
7.      private QuickFind QF;
8.      public Percolation(int n,int choice) {
9.          this.N = n;
10.         this.choice=choice;
11.         this.sites = new boolean[N*N+2];
12.         sites[0] = true;       //初始化所有格点为 blocked
13.         sites[N*N+1] = true;   //开头和末尾为 open
14.         count = 0;
15.         WQU = new WeightedQuickUnion(N*N+2);
16.         QF=new QuickFind(N*N+2);
17.     }
18.     //将节点变为 open 节点
19.     public void Open(int row, int col) {
20.         if(row < 1 || row > N || col < 1 || col > N)
21.             throw new IndexOutOfBoundsException("Index Out Of Bounds Excepti
    on");
22.         int a=0;
23.         if(sites[(row-1)*N+col]) return;
24.         sites[(row-1)*N+col] = true;
25.         count++;
26.
27.         if(choice==2){
28.             //如果在第一行，将其与开头连接
29.             if(row == 1)
30.                 WQU.Union(0,(row-1)*N+col);
31.             //如果在最后一行，将其与结尾连接
32.             if(row == N)
```

```
33.                 WQU.Union((row-1)*N+col,N*N+1);
34.
35.             //连接上下左右的 open 节点
36.             if(col != 1 && sites[(row-1)*N+col-1])
37.                 WQU.Union((row-1)*N+col,(row-1)*N+col-1);
38.             if(col != N && sites[(row-1)*N+col+1])
39.                 WQU.Union((row-1)*N+col,(row-1)*N+col+1);
40.             if(row != 1 && sites[(row-2)*N+col])
41.                 WQU.Union((row-1)*N+col,(row-2)*N+col);
42.             if(row != N && sites[row*N+col])
43.                 WQU.Union((row-1)*N+col,row*N+col);
44.         }else if(choice==1){
45.             if(row == 1)
46.                 QF.Union(0,(row-1)*N+col);
47.             //如果在最后一行，将其与结尾连接
48.             if(row == N)
49.                 QF.Union((row-1)*N+col,N*N+1);
50.
51.             //连接上下左右的 open 节点
52.             if(col != 1 && sites[(row-1)*N+col-1])
53.                 QF.Union((row-1)*N+col,(row-1)*N+col-1);
54.             if(col != N && sites[(row-1)*N+col+1])
55.                 QF.Union((row-1)*N+col,(row-1)*N+col+1);
56.             if(row != 1 && sites[(row-2)*N+col])
57.                 QF.Union((row-1)*N+col,(row-2)*N+col);
58.             if(row != N && sites[row*N+col])
59.                 QF.Union((row-1)*N+col,row*N+col);
60.         }
61.     }
62.     //判断节点是否为 open 节点
63.     public boolean IsOpen(int row, int col) {
64.         if(row < 1 || row > N || col < 1 || col > N)
65.             throw new IndexOutOfBoundsException("Index Out Of Bounds Excepti
    on");
66.         return sites[(row-1)*N+col];
67.     }
68.     //判断首尾是否已经连通
69.     public boolean Percolates(){
70.         if (choice==1) return QF.Find(0)==QF.Find(N*N+1);
71.         return WQU.Find(0)==WQU.Find(N*N+1);
72.     }
73. }
```

- QuickFind 源代码：

```java
package First_Experiment;

public class QuickFind {
    private int[] id;
    private int count;
    public QuickFind(int N){
        count = N;
        id = new int[N];
        for (int i = 0; i < N; i++)
            id[i] = i;
    }
    public int Find(int p){
        return id[p];
    }
    public boolean Connected(int p, int q) {
        return id[p] == id[q];
    }
    public void Union(int p, int q) {//将 p 和 q 归并到相同的分量中
        int pID = id[p],qID = id[q];
        if (pID == qID) return;
        for (int i = 0; i < id.length; i++)
            if (id[i] == pID) id[i] = qID;
        count--;
    }
}
```

- WeighedQuickUnion 源代码：

```java
package First_Experiment;

public class WeightedQuickUnion {
    private int[] id;       //父链接数组
    private int[] size;     //各个根节点所对应的分量大小
    private int count;      //连通分量数量
    public WeightedQuickUnion(int N){
        count=N;
        id=new int[N];
        size=new int[N];
        for (int i=0;i<N;i++){
            id[i]=i;
            size[i]=1;
        }
```

```
    }
    public int Find(int p){    //找到根结点
        while (p!=id[p]){
            p=id[p];
        }
        return p;
    }
    public boolean Connected(int p,int q){
        return Find(p)==Find(q);
    }
    public void Union(int p,int q){
        int i=Find(p),j=Find(q);
        if(i==j) return;
        if (size[i]<size[j]){   //将小树的根节点连接到大树的根节点
            id[i]=j;
            size[j]+=size[i];
        }else {
            id[j]=i;
            size[i]+=size[j];
        }
        count--;
    }
}
```

六、实验结果

- 使用 quick-find 算法：(迭代次数为 200)

1) N=10

```
------------------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 10 , 200 )
mean:                = 0.5935
stddev:              = 0.07887876327936699
confidence_Low:      = 0.582567961152336
confidence_High:     = 0.604432038847664
The cost of the time is 16.0221ms
```

2) N=20

```
------------------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 20 , 200 )
mean:                = 0.5901125000000002
stddev:              = 0.04460624597125654
confidence_Low:      = 0.5839303897141283
confidence_High:     = 0.596294610285872
The cost of the time is 34.025ms
```

3) N=40

```
-----------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 40 , 200 )
mean:                = 0.5917187500000004
stddev:              = 0.030107034640791334
confidence_Low:      = 0.5875461278822397
confidence_High:     = 0.5958913721177611
The cost of the time is 154.7997ms
```

4) N=80

```
-----------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 80 , 200 )
mean:                = 0.5912195312500002
stddev:              = 0.019132699957258605
confidence_Low:      = 0.5885678743210919
confidence_High:     = 0.5938711881789084
The cost of the time is 1874.4501ms
```

5) N=160

```
-----------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 160 , 200 )
mean:                = 0.5932671875000001
stddev:              = 0.011858252217757064
confidence_Low:      = 0.5916237175909855
confidence_High:     = 0.5949106574090146
The cost of the time is 26268.0185ms
```

6) N=320

```
-----------------------------------------------------
The algorithm currently used is: Quick_Find
Creating PercolationStats( 320 , 200 )
mean:                = 0.59328251953125
stddev:              = 0.0065809282643326415
confidence_Low:      = 0.5923704494068165
confidence_High:     = 0.5941945896556835
The cost of the time is 496730.8229ms
```

- 使用 weighted quick-union 算法：(迭代次数为 200)

1) N=10

```
-----------------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 10 , 200 )
mean:                = 0.5933500000000002
stddev:              = 0.07260603059245119
confidence_Low:      = 0.583287317548956
confidence_High:     = 0.6034126824510443
The cost of the time is 13.6582ms
```

2) N=20

```
-----------------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 20 , 200 )
mean:                = 0.5926
stddev:              = 0.04866034469389467
confidence_Low:      = 0.5858560202972461
confidence_High:     = 0.5993439797027539
The cost of the time is 20.8722ms
```

3) N=40

```
-----------------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 40 , 200 )
mean:                = 0.5925625
stddev:              = 0.0318000584683891
confidence_Low:      = 0.5881552367509152
confidence_High:     = 0.5969697632490848
The cost of the time is 50.0549ms
```

4) N=80

```
-----------------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 80 , 200 )
mean:                = 0.5928898437499999
stddev:              = 0.0187918097862603
confidence_Low:      = 0.5902854317883959
confidence_High:     = 0.5954942557116039
The cost of the time is 125.841ms
```

5） N=160

```
-----------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 160 , 200 )
mean:                = 0.5933451171875
stddev:              = 0.011323067052183224
confidence_Low:      = 0.5917758201582001
confidence_High:     = 0.5949144142168
The cost of the time is 372.4429ms
```

6） N=320

```
-----------------------------------------------------
The algorithm currently used is: Weighted Quick Union
Creating PercolationStats( 320 , 200 )
mean:                = 0.5925082519531248
stddev:              = 0.007266634834429779
confidence_Low:      = 0.5915011477466299
confidence_High:     = 0.5935153561596198
The cost of the time is 1612.7254ms
```

七、结果分析

算法/N/时间(ms)	N=10	N=20	N=40	N=80	N=160	N=320
① Quick-find	16.02	34.03	154.80	1874.45	26268.0	496730.8
② Weighted quick-union	13.66	20.87	50.05	125.84	372.44	1612.73
① 算法对应阈值 p*	0.5935	0.5901	0.5917	0.5912	0.5933	0.5933
② 算法对应阈值 p*	0.5934	0.5926	0.5926	0.5929	0.5933	0.5925

最后根据两种方法求得的阈值进行加权平均，解得题目对应的阈值 p*=0.5926。

《算法分析与设计实验报告》

18030400014-魏红旭

实验二：排序算法比较

一、 实验目的

通过对多种排序算法的实现，针对不同输入规模的数据进行实验，比较各种排序算法的时间性能。

二、 题目描述

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）

三、 实验要求

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性能。要求对于每次输入运行 10 次，记录每次时间，取平均值。

四、 程序设计及源代码

- 插入排序（Insertion Sort，IS）

```
1.  package Second_Experiment;
2.
3.  public class Insertion_Sort {
4.      public Insertion_Sort(int[] a) {
5.          int N = a.length;
6.          for(int i = 1;i < N;i++) {
7.              for(int j =i;j>0 &&less(a[j],a[j-1]) ; j--) {
8.                  exch(a, j, j-1);
9.              }
10.         }
11.     }
12.     private boolean less(int a,int b) {
13.         if(a < b) return true;
14.         else return false;
15.     }
16.     private void exch(int[] a,int i,int j) {
17.         int t = a[i];
18.         a[i] = a[j];
```

```
        a[j] = t;
    }
}
```

- 自顶向下归并排序（Top-down Mergesort，TDM）

```
package Second_Experiment;

public class Topdown_MergeSort {
    private static int[] aux;
    public Topdown_MergeSort(int[] a) {
        aux = new int[a.length];
        sort(a,0,a.length-1);
    }
    private void sort(int[] a, int lo,int hi) {
        if(hi <= lo) return;
        int mid = lo + (hi - lo)/2;
        sort(a,lo,mid);
        sort(a,mid+1,hi);
        merge(a,lo,mid,hi);
    }
    private void merge(int[] a, int lo,int mid,int hi) {
        int i = lo,j = mid+1;
        for(int k = lo; k<=hi;k++)
            aux[k] = a[k];
        for(int k = lo;k <=hi;k++) {
            if(i > mid)                  a[k] = aux[j++];
            else if(j > hi)              a[k] = aux[i++];
            else if(less(aux[j],aux[i])) a[k] = aux[j++];
            else                         a[k] = aux[i++];
        }
    }
    private boolean less(int a,int b) {
        if(a < b) return true;
        else return false;
    }
}
```

- 自底向上归并排序（Bottom-up Mergesort，BUM）

```
package Second_Experiment;

public class Bottomup_MergeSort {
    private static int[] aux;
```

```
    public Bottomup_MergeSort(int[] a) {
        sort(a);
    }
    private void sort(int[] a) {
        int N = a.length;
        aux = new int[N];
        for(int sz = 1;sz < N;sz = sz+sz) {
            for(int lo = 0;lo < N-sz;lo+=sz+sz) {
                merge(a,lo,lo+sz-1,Math.min(lo+sz+sz-1,N-1));
            }
        }
    }
    private void merge(int[] a, int lo,int mid,int hi) {
        int i = lo,j = mid+1;
        for(int k = lo; k<=hi;k++)
            aux[k] = a[k];
        for(int k = lo;k <=hi;k++) {
            if(i > mid)                  a[k] = aux[j++];
            else if(j > hi)              a[k] = aux[i++];
            else if(less(aux[j],aux[i])) a[k] = aux[j++];
            else                         a[k] = aux[i++];
        }
    }
    private boolean less(int a,int b) {
        if(a < b) return true;
        else return false;
    }
}
```

- 随机快速排序（Random Quicksort，RQ）

```
package Second_Experiment;

public class Random_QuickSort {
    public Random_QuickSort(int[] a) {
        sort(a,0,a.length-1);
    }
    private void sort(int[] a,int lo,int hi) {
        if(lo<hi){
            int j = partition(a,lo,hi);
            sort(a,lo,j -1);
            sort(a,j+1,hi);
        }
    }
```

```
    private int partition(int[] a, int lo, int hi) {
        int i = lo,j = hi+1;
        int v = a[lo];
        while(true) {
            while(less(a[++i],v)) if(i == hi) break;
            while(less(v,a[--j])) if(j == lo) break;
            if(i >=j) break;
            exch(a,i,j);
        }
        exch(a, lo, j);
        return j;
    }
    private boolean less(int a,int b) {
        if(a < b) return true;
        else return false;
    }
    private void exch(int[] a,int i,int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

- Dijkstra 3-路划分快速排序（Quicksort with Dijkstra 3-way Partition，QD3P）

```
package Second_Experiment;

public class QuickSort_With_D3P {
    public QuickSort_With_D3P(int[] a) {
        sort(a,0,a.length-1);
    }
    private void sort(int[] a,int lo,int hi) {
        if(hi<=lo) return;
        int lt = lo,i = lo +1,gt = hi;
        int v = a[lo];
        while(i <= gt) {
            int cmp = compare(a[i],v);
            if(cmp < 0) exch(a, lt++, i++);
            else if(cmp > 0) exch(a, i, gt--);
            else i++;
        }
        sort(a,lo,lt-1);
        sort(a,gt+1,hi);
```

```
    }
    private int compare(int i,int j) {
        if(i < j) return -1;
        else return 1;
    }
    private void exch(int[] a,int i,int j) {
        int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
}
```

- 程序主函数：

```
package Second_Experiment;

import edu.princeton.cs.algs4.StdStats;
import java.util.Random;
import java.util.Scanner;

public class CompareMain {
    private static double[][] experiments=new double[5][10];
    private static int[] RandomIndex;

    public CompareMain(int[] Randomindex){
        int index[]=new int[Randomindex.length];
        for (int i=0;i<12;i++){ //运行十次，i<10

            //传入 5 个算法中的参数为 index 数组，原始数组存放在 Randomindex 中，所以每次计算之间需要重新将原始数组赋给 index
            System.arraycopy(Randomindex,0,index,0,Randomindex.length);
            long start_time=System.nanoTime();
            Insertion_Sort insertion_sort=new Insertion_Sort(index);
            long consumingtime=System.nanoTime()-start_time;  //计算总耗费时间，nanoTime 单位为纳秒
            if(i>=2) experiments[0][i-2]=consumingtime*1.0/1000000; //experiments 数组记录 5 个算法各 10 次运行的时间

            System.arraycopy(Randomindex,0,index,0,Randomindex.length);
            start_time=System.nanoTime();
            Topdown_MergeSort topdown_mergeSort=new Topdown_MergeSort(index);
            consumingtime=System.nanoTime()-start_time;  //计算总耗费时间
            if(i>=2) experiments[1][i-2]=consumingtime*1.0/1000000;
```

```
27.
28.             System.arraycopy(Randomindex,0,index,0,Randomindex.length);
29.             start_time=System.nanoTime();
30.             Bottomup_MergeSort bottomup_mergeSort=new Bottomup_MergeSort(ind
    ex);
31.             consumingtime=System.nanoTime()-start_time;  //计算总耗费时间
32.             if(i>=2) experiments[2][i-2]=consumingtime*1.0/1000000;
33.
34.             System.arraycopy(Randomindex,0,index,0,Randomindex.length);
35.             start_time=System.nanoTime();
36.             Random_QuickSort random_quickSort=new Random_QuickSort(index);
37.             consumingtime=System.nanoTime()-start_time;  //计算总耗费时间
38.             if(i>=2) experiments[3][i-2]=consumingtime*1.0/1000000;
39.
40.             System.arraycopy(Randomindex,0,index,0,Randomindex.length);
41.             start_time=System.nanoTime();
42.             QuickSort_With_D3P quickSort_with_d3P=new QuickSort_With_D3P(ind
    ex);
43.             consumingtime=System.nanoTime()-start_time;  //计算总耗费时间
44.             if(i>=2) experiments[4][i-2]=consumingtime*1.0/1000000;
45.         }
46.     }
47.     public static void main(String[] args){
48.         Random rd=new Random();       //随机
49.         Scanner in=new Scanner(System.in);
50.         System.out.print("Please enter the amount of data you want to enter:
     ");
51.         int count=in.nextInt();       //输入数据规模
52.         RandomIndex=new int[count];   //生成 count 数量的随机数，存放在数组
    RandomIndex 中
53.         for (int i=0;i<count;i++){
54.             RandomIndex[i]=rd.nextInt(10000);//生成 10000 以内的随机数
55.         }
56.         /*for (int i=0;i<count;i++){                  //升序测试
57.             RandomIndex[i]=i;
58.         }*/
59.         /*for (int i=0;i<count;i++){                  //降序测试
60.             RandomIndex[i]=count-i;
61.         }*/
62.         CompareMain compareMain=new CompareMain(RandomIndex);
63.         for (int i=0;i<5;i++){
64.             System.out.println("----------------------------------------
    ----------------------------------------------------------------------
    ");
```

```
65.             if(i==0) System.out.println("Insertion Sort Time: ");
66.             else if(i==1) System.out.println("Top-down Mergesort Time: ");
67.             else if(i==2) System.out.println("Bottom-
    up Mergesort Time: ");
68.             else if(i==3) System.out.println("Random Quicksort Time: ");
69.             else if(i==4) System.out.println("Quicksort with Dijkstra 3-
    way Partition Time: ");
70.             for (int j=0;j<10;j++){
71.                 System.out.print(experiments[i][j]+"ms\t");
72.             }
73.             System.out.println("\nThe average running time is: "+StdStats.me
    an(experiments[i])+"ms");
74.         }
75.     }
76. }
```

五、 实验结果

- **测试数据：随机数据**

1） N=2000

```
Please enter the amount of data you want to enter: 2000
-----------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
5.2802ms    0.7758ms    0.7867ms    0.8287ms    0.7796ms    0.6416ms    0.6597ms    0.6429ms    0.6878ms    0.6419ms
The average running time is: 1.1724900000000003ms
-----------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
0.2034ms    0.369ms 0.1871ms    0.3163ms    0.1874ms    0.1996ms    0.1862ms    0.2158ms    0.2572ms    0.1853ms
The average running time is: 0.23072999999999996ms
-----------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
0.2576ms    0.294ms 0.2858ms    0.6266ms    0.2554ms    0.2551ms    0.2545ms    0.2855ms    0.255ms 0.2698ms
The average running time is: 0.30393ms
-----------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
0.2934ms    0.1564ms    0.1308ms    0.1428ms    0.1481ms    0.1305ms    0.1289ms    0.1471ms    0.1299ms    0.1295ms
The average running time is: 0.15374ms
-----------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
0.3554ms    0.1704ms    0.1604ms    0.2418ms    0.1606ms    0.1598ms    0.1681ms    0.1594ms    0.172ms 0.16ms
The average running time is: 0.19078999999999996ms
```

2） N=10000

```
Please enter the amount of data you want to enter: 10000
-----------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
18.8811ms   16.8873ms   15.0177ms   14.2196ms   13.6693ms   15.2368ms   14.7681ms   15.1081ms   14.0802ms   14.5367ms
The average running time is: 15.240489999999998ms
-----------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
1.0785ms    0.9575ms    0.8914ms    0.8985ms    0.8581ms    0.916ms 0.8991ms    0.8655ms    0.893ms 1.019ms
The average running time is: 0.92766ms
-----------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
1.3811ms    1.3647ms    1.3054ms    1.3428ms    1.3892ms    0.9627ms    0.8552ms    0.9167ms    0.9383ms    1.0189ms
The average running time is: 1.1475ms
-----------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
0.7955ms    0.735ms 0.7382ms    0.7412ms    0.7349ms    0.7788ms    0.7617ms    0.6811ms    0.8418ms    0.9192ms
The average running time is: 0.77274ms
-----------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
0.9187ms    0.8458ms    0.8452ms    0.8562ms    0.8407ms    0.8049ms    0.8051ms    0.8078ms    1.1269ms    0.8953ms
The average running time is: 0.8746600000000001ms
```

3） N=50000

```
Please enter the amount of data you want to enter: 50000
-----------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
364.261ms   372.2784ms  362.8681ms  373.3696ms  370.1741ms  362.0584ms  368.6748ms  361.9045ms  367.2962ms  369.9714ms
The average running time is: 367.28565000000003ms
-----------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
5.0922ms    5.0172ms    5.5517ms    5.3238ms    4.9741ms    4.8797ms    6.138ms 5.1436ms    4.8942ms    5.8413ms
The average running time is: 5.28558ms
-----------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
5.6847ms    4.9409ms    5.6367ms    5.0489ms    4.7264ms    4.6105ms    4.9735ms    5.5205ms    4.9819ms    5.162ms
The average running time is: 5.1286000000000005ms
-----------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
4.63ms  4.038ms 4.0286ms    4.692ms 3.9575ms    3.8606ms    4.2374ms    4.1537ms    4.6745ms    4.1156ms
The average running time is: 4.23879ms
-----------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
5.1442ms    4.9025ms    5.2592ms    5.1028ms    4.7629ms    5.4949ms    4.9064ms    5.1049ms    4.6888ms    4.899ms
The average running time is: 5.02656ms
```

- 测试数据：升序数据

1） N=2000

```
Please enter the amount of data you want to enter: 2000
-------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
0.0844ms    0.1032ms    0.2499ms    0.0644ms    0.0733ms    0.1091ms    0.1291ms    0.1102ms    0.0643ms    0.0641ms
The average running time is: 0.1052ms
-------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
0.0707ms    0.0912ms    0.1293ms    0.0698ms    0.0808ms    0.1272ms    0.1275ms    0.1289ms    0.0725ms    0.0748ms
The average running time is: 0.09727ms
-------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
0.1257ms    0.1404ms    0.4621ms    0.1261ms    0.1411ms    0.4638ms    0.2258ms    0.2267ms    0.1257ms    0.1261ms
The average running time is: 0.21635000000000004ms
-------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
0.6616ms    0.9889ms    0.8255ms    0.7481ms    1.4901ms    1.8159ms    1.7068ms    1.3584ms    0.7724ms    0.7471ms
The average running time is: 1.1114799999999998ms
-------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
0.9653ms    1.9916ms    1.4114ms    2.2879ms    2.5607ms    2.4991ms    2.4401ms    1.6769ms    1.3196ms    1.2801ms
The average running time is: 1.8432700000000004ms
```

2） N=10000

```
Please enter the amount of data you want to enter: 10000
-------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
0.8356ms    0.7462ms    0.3033ms    0.3729ms    0.3256ms    0.1764ms    0.0505ms    0.0481ms    0.0634ms    0.1245ms
The average running time is: 0.30465ms
-------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
0.3047ms    0.3071ms    0.3043ms    0.3006ms    0.3035ms    0.604ms 0.3938ms    0.3337ms    0.3293ms    0.3036ms
The average running time is: 0.34846ms
-------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
0.6003ms    0.7112ms    0.6917ms    0.9454ms    0.6088ms    0.7825ms    0.293ms 0.3146ms    0.3679ms    0.4277ms
The average running time is: 0.5743099999999999ms
-------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
24.5745ms   22.98ms 22.2208ms   23.1768ms   22.8119ms   25.2444ms   24.0067ms   23.8931ms   23.762ms    24.1823ms
The average running time is: 23.68525ms
-------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
2.1105ms    2.1232ms    2.0201ms    2.38ms  2.2655ms    2.1377ms    2.1033ms    2.1359ms    2.097ms 2.0741ms
The average running time is: 2.14473ms
```

- 测试数据：降序数据

1） N=2000

```
Please enter the amount of data you want to enter: 2000
-----------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
1.3788ms    1.5851ms    1.8393ms    1.8443ms    1.6459ms    2.1041ms    1.3429ms    1.3421ms    1.3421ms    1.378ms
The average running time is: 1.5802599999999998ms
-----------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
0.0949ms    0.1533ms    0.1273ms    0.1222ms    0.1361ms    0.1312ms    0.0718ms    0.0653ms    0.0665ms    0.0664ms
The average running time is: 0.1035ms
-----------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
0.1569ms    0.4674ms    0.5809ms    0.6166ms    0.2915ms    0.4496ms    0.1558ms    0.1547ms    0.1566ms    0.1543ms
The average running time is: 0.31843000000000005ms
-----------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
0.9159ms    1.5001ms    1.4747ms    1.4734ms    1.561ms 0.8337ms    0.802ms 0.8016ms    0.803ms 0.7997ms
The average running time is: 1.09651ms
-----------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
1.5384ms    2.3744ms    2.3717ms    1.6993ms    2.2578ms    1.5292ms    1.5167ms    1.5488ms    1.5058ms    1.4984ms
The average running time is: 1.78405ms
```

2） N=10000

```
Please enter the amount of data you want to enter: 10000
-----------------------------------------------------------------------------------------------------------------------
Insertion Sort Time:
29.326ms    28.9461ms    30.6881ms    27.7012ms    29.28ms 29.251ms    30.2596ms    29.3821ms    27.5294ms    28.3494ms
The average running time is: 29.071289999999998ms
-----------------------------------------------------------------------------------------------------------------------
Top-down Mergesort Time:
0.2791ms    0.3079ms    0.3778ms    0.2808ms    0.3829ms    0.2808ms    0.2822ms    0.3117ms    0.2817ms    0.2823ms
The average running time is: 0.30672ms
-----------------------------------------------------------------------------------------------------------------------
Bottom-up Mergesort Time:
0.6816ms    0.6823ms    0.7984ms    0.6823ms    0.733ms 0.3933ms    0.3105ms    0.3304ms    0.3521ms    0.4667ms
The average running time is: 0.5430600000000001ms
-----------------------------------------------------------------------------------------------------------------------
Random Quicksort Time:
19.6617ms    19.4243ms    19.2272ms    19.8881ms    19.7758ms    19.4851ms    20.1596ms    19.3196ms    19.9851ms    19.617ms
The average running time is: 19.65435ms
-----------------------------------------------------------------------------------------------------------------------
Quicksort with Dijkstra 3-way Partition Time:
38.5826ms    39.1718ms    37.6548ms    38.2681ms    39.2958ms    38.9447ms    38.237ms    38.7167ms    38.8187ms    40.784ms
The average running time is: 38.84742ms
```

六、 结果分析

- **测试数据：随机数据**

时间 ms/ 算法	算法运行平均时间				
	IS	TDM	BUM	RQ	QD3P
N=2000	1.172	0.231	0.304	0.154	0.191
N=10000	15.24	0.928	1.148	0.773	0.875
N=50000	367.3	5.286	5.123	4.239	5.027

- **测试数据：升序数据**

时间 ms/ 算法	算法运行平均时间				
	IS	TDM	BUM	RQ	QD3P
N=2000	0.105	0.097	0.216	1.111	1.843
N=10000	0.304	0.348	0.574	23.68	2.145

- **测试数据：降序数据**

时间 ms/ 算法	算法运行平均时间				
	IS	TDM	BUM	RQ	QD3P
N=2000	1.580	0.103	0.318	1.097	1.784
N=10000	29.07	0.307	0.543	19.65	38.85

七、 回答问题

1. Which sort worked best on data in constant or increasing order (i.e., already sorted data)? Why do you think this sort worked best?
   哪种排序对不变或递增顺序的数据(即已经排序的数据)最有效?为什么你认为这种方法效果最好?

   答：插入排序；因为插入排序对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入，所以插入排序每次只要比较一次就能确定一个元素的位置。

2. Did the same sort do well on the case of mostly sorted data? Why or why not?
   同样的排序在排序最多的数据中表现良好吗?为什么或为什么不?

   答：不一定，因为初始数据是否有序对不同的排序方法的影响是不同的，比如快排的时候，初始有序的时候时间复杂度就变成了 N^2;

3. In general, did the ordering of the incoming data affect the performance of the sorting algorithms? Please answer this question by referencing specific data from your table to support your answer.
一般来说，传入数据的排序会影响排序算法的性能吗?请在回答这个问题时引用表格中的具体数据来支持你的答案。

   答：会影响，初始状况对于每一种排序的影响是很大的，以快速排序为例，当需要排序的数据为随机数据无序时，N=10000 时，处理时间为 0.773ms；但数据为升序排列时，处理时间变为了 23.68ms；

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

   答：在这个问题中，用 2000 的数据代替 1000 的数据，2000 的时候快速排序 RQ 在我的算法中表现的最好，在 10000 的时候，数据量无序的时候是 RQ，有序的时候时插入排序。大量有序数据的时候对于快速排序来说是最坏的情况，复杂度为 N2。数据量为 10000 的时候看表中，随机数据 RQ 为 0.773ms，用时最少；有序数据 RQ 用时为 23.67ms;

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

   答：综合来看，Dijkstra 3-路划分快速排序（QD3P）算法最好，他的空间复杂度是 lgN，而且也能处理大量重复数据的情况， 时间复杂度介于 NlgN 与 N2 之间，综合来看可以比 RQ 适用于更多的情况。

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

   答：存在这样的情况，例如（QD3P）在随机数据量为 10000 的时候，出现了一次运行时间大于 1ms 的情况。对于数据突然出现的偶然性情况，我认为应该是初始化随机数的时候造成了一些偶然情况的发生，使得初始的排序出现了某种比较差的情况。

《算法分析与设计实验报告》

18030400014-魏红旭

实验三：地理路由

一、 实验目的

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

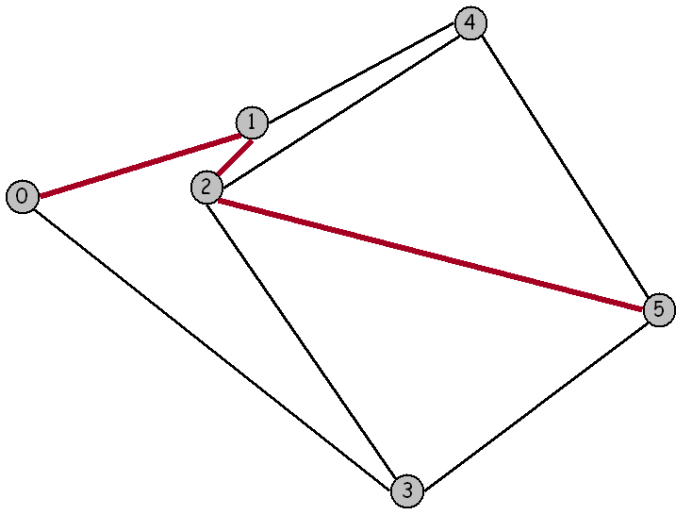
优化 Dijkstra 算法，使其可以处理给定图的数千条最短路径查询。一旦你读取图（并可选地预处理），你的程序应该在亚线性时间内解决最短路径问题。 一种方法是预先计算出所有顶点对的最短路径；然而，你无法承受存储所有这些信息所需的二次空间。你的目标是减少每次最短路径计算所涉及的工作量，而不会占用过多的空间。建议你选择下面的一些潜在想法来实现， 或者你可以开发和实现自己的想法。

二、 题目描述

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

```
6 9
0 1000 2400
1 2800 3000
2 2400 2500
3 4000    0
4 4500 3800
5 6000 1500

0 1
0 3
1 2
1 4
2 4
2 3
2 5
3 5
4 5
0 5
```



Dijkstra 算法。 Dijkstra 算法是最短路径问题的经典解决方案。 它在教科书第 21 章中有描述。 基本思路不难理解。 对于图中的每个顶点，我们维护从源点到该顶点的最短已知的路径长度，并且将这些长度保持在优先队列（priority queue, PQ）中。 初始时，我们把所有的顶点放在这个队列中，并设置高优先级，然后将源点的优先级设为 0.0。 算法通过从 PQ 中取出最低优先级的顶点，然后检查可从该顶点经由一条边可达的所有顶点，以查看这条边是否提供了从源点到那个顶点较之之前已知的最短路径的更短路径。 如果是这样，它会降低优先级来反映这种新的信息。这里给出了 Dijkstra 算法计算从 0 到 5 的最短路径 0-1-2-5 的详细过程。

```
process 0 (0.0)
    lower 3 to 3841.9
    lower 1 to 1897.4
process 1 (1897.4)
    lower 4 to 3776.2
    lower 2 to 2537.7
process 2 (2537.7)
    lower 5 to 6274.0
process 4 (3776.2)
process 3 (3841.9)
process 5 (6274.0)
```

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

三、 实验要求

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

下面的一些潜在想法来实现， 或者你可以开发和实现自己的想法。

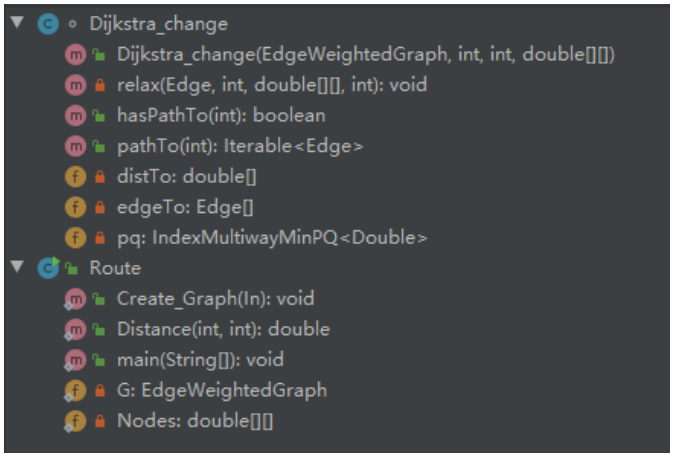
- **想法 1.** Dijkstra 算法的朴素实现检查图中的所有 V 个顶点。 减少检查的顶点数量的一种策略是一旦发现目的地的最短路径就停止搜索。 通过这种方法，可以使每个最短路径查询的运行时间与 E' log V'成比例，其中 E'和 V'是 Dijkstra 算法检查的边和顶点数。 然而，这需要一些小心，因为只是重新初始化所有距离为∞就需要与 V 成正比的时间。由于你在不断执行查询，因而只需重新初始化在先前查询中改变的那些值来大大加速查询。
- **想法 2.** 你可以利用问题的欧式几何来进一步减少搜索时间，这在算法书的第 21.5 节描述过。对于一般图，Dijkstra 通过将 d[w]更新为 d[v] + 从 v 到 w 的距离来松弛边 v-w。 对于地图，则将 d[w]更新为 d[v] + 从 v 到 w 的距离 + 从 w 到 d 的欧式距离 从 v 到 d 的欧式距离。 这种方法称之为 A*算法。这种启发式方法会有性能上的影响，但不会影响正确性。
- **想法 3.** 使用更快的优先队列。 在提供的优先队列中有一些优化空间。你也可以考虑使用 Sedgewick 程序 20.10 中的多路堆。

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

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

四、 程序设计及源代码

- 程序结构：



- **源代码：**

```
1.  package Three;
2.
3.  import edu.princeton.cs.algs4.*;
4.  import java.text.DecimalFormat;
5.  import java.util.Scanner;
6.
7.  class Dijkstra_change {
8.      private double[] distTo;
9.      private Edge[] edgeTo;
10.     private IndexMultiwayMinPQ<Double> pq;   //优化 3
11.     public Dijkstra_change(EdgeWeightedGraph G, int s, int d, double[][] Nod
    es) {
12.         for (Edge e : G.edges()) {
13.             if (e.weight() < 0)
14.                 throw new IllegalArgumentException("edge " + e + " has negat
    ive weight");
15.         }
16.         distTo = new double[G.V()];
17.         edgeTo = new Edge[G.V()];
18.         for (int v = 0; v < G.V(); v++)
19.             distTo[v] = Double.POSITIVE_INFINITY;
20.         distTo[s] = 0.0;
21.         pq = new IndexMultiwayMinPQ<Double>(G.V(), 2);
22.         pq.insert(s, distTo[s]);
23.         while (!pq.isEmpty()) {
24.             int v = pq.delMin();
25.             if (v == d)                 // 优化 1，当目标节点已发现，停止运行
26.                 return;
27.             for (Edge e : G.adj(v))     //从距离最近节点的邻居节点开始遍历
28.                 relax(e, v, Nodes, d); //更新 dist 数组的值
29.         }
30.     }
31.
32.     private void relax(Edge e, int v, double[][] Nodes, int d) {
33.         int w = e.other(v);           // distTo[W] = distTo[V] - V2D + U->V
    + W2D；A*算法距离公式：                  //优化 2
34.         double newDist = distTo[v] + e.weight() + Route.Distance(w,d) - Rout
    e.Distance(v,d);
35.         if (distTo[w]- 0.000001 > newDist) {
```

```
36.                 distTo[w] = newDist;      //更新 dist 数组
37.                 edgeTo[w] = e;            //记录最短距离选择的边
38.                 if (pq.contains(w))       //重新插入到队列之中
39.                     pq.decreaseKey(w, distTo[w]);
40.                 else
41.                     pq.insert(w, distTo[w]);
42.             }
43.         }
44.         public boolean hasPathTo(int v) {   //判断目标节点是否可达
45.             return distTo[v] < Double.POSITIVE_INFINITY;
46.         }
47.         public Iterable<Edge> pathTo(int v) {
48.             if (!hasPathTo(v)) return null;
49.             Stack<Edge> path = new Stack<Edge>();
50.             int x = v,j=0;
51.             String result[] = new String[10000];
52.             for(Edge e = edgeTo[v]; e != null; e = edgeTo[x]) {
53.                 path.push(e);
54.                 DecimalFormat df = new DecimalFormat("#.00");
55.                 String str = df.format(e.weight());
56.                 result[j++]="<"+e.other(x) + "-" + x +"> --- " + str;
57.                 x = e.other(x);
58.             }
59.             for (int i=0;i<j;i++){
60.                 StdOut.println(result[j-i-1]);
61.             }
62.             return path;
63.         }
64. }
65.
66. public class Route {
67.     private static EdgeWeightedGraph G;  //图数据
68.     private static double[][] Nodes;     //记录节点坐标
69.     public static void Create_Graph(In in) {
70.         int V = in.readInt();
71.         int E = in.readInt();
72.         G = new EdgeWeightedGraph(V);
73.         Nodes = new double[V][2];
74.         int v;
75.         for (int i = 0; i < V; i++) {
76.             v = in.readInt();
77.             Nodes[v][0] = in.readDouble();
78.             Nodes[v][1] = in.readDouble();
79.         }
```

```
80.         int v1, v2;
81.         for (int i = 0; i < E; i++) {     //读取边数据，并插入到图中
82.             v1 = in.readInt();
83.             v2 = in.readInt();
84.             G.addEdge(new Edge(v1, v2, Distance(v1,v2)));
85.         }
86.     }
87.     public static double Distance(int v1, int v2) {//计算两个节点之间的距离
88.         return Math.sqrt(Math.pow((Nodes[v1][0]-
    Nodes[v2][0]), 2) + Math.pow((Nodes[v1][1]-Nodes[v2][1]), 2));
89.     }
90.     public static void main(String[] args) {
91.         In in = new In("G:\\web_file\\SF_Travis\\src\\Three\\usa.txt");
92.         Scanner input =new Scanner(System.in);
93.         StdOut.print("Please enter a start node: ");
94.         int start =input.nextInt();
95.         StdOut.print("Please enter an end node: ");
96.         int end =input.nextInt();
97.         Create_Graph(in);
98.
99.         long startTime = System.currentTimeMillis();
100.         Dijkstra_change DC= new Dijkstra_change(G, start, end, Nodes);
101.         long endTime = System.currentTimeMillis();
102.
103.         if (DC.hasPathTo(end)) {
104.             double count = 0;
105.             for (Edge e : DC.pathTo(end)) {
106.                 count += e.weight();
107.             }
108.             StdOut.println("Length of path: " + count);
109.         } else {
110.             StdOut.println("Path doesn't exist!");
111.         }
112.         StdOut.println("-------
    The program running time is " + (double)(endTime - startTime) + "ms -------
    ");
113.     }
114. }
```

- **分析：**

1. 改进 1，我们的目标是求 S 和 D 之间的最短路径，所以并不需要把 S 到所有节点的最短路径求出来,所以,我们可以加一个判断条件，当 PQ.delmin（）输出的节点是 D 的时候直接退出,其实就是把 Dijsktra 算法的加了两行.

```
while (!pq.isEmpty()) {
    int v = pq.delMin();
    /*if (v == d)                  // 优化1，当目标节点已发现，停止运行
        return;*/
    for (Edge e : G.adj(v))    //从距离最近节点的邻居节点开始遍历
        relax(e, v, Nodes, d); //更新dist数组的值
}
```

2. 改进 2，此时我们更改 dist 数组的定义，我们的目标是求 s 与 d 之间的最短路径，那么在这个几何平面上，我们不妨定义 dist[V]现在不是 S 到 V 之间的最短距离，而是 S 到 V 的最短距离+V 到 D 的直接距离的和.

```
private void relax(Edge e, int v, double[][] Nodes, int d) {
    int w = e.other(v);           // distTo[W] = distTo[V] - V2D + U->V + W2D; A*算法距离公式;
    double newDist = distTo[v] + e.weight() + Route.Distance(w,d) - Route.Distance(v,d);
    /*if (distTo[w]- 0.000001 > newDist) {
        distTo[w] = newDist;      //更新dist数组
        edgeTo[w] = e;            //记录最短距离选择的边
        if (pq.contains(w))       //重新插入到队列之中
            pq.decreaseKey(w, distTo[w]);
        else
            pq.insert(w, distTo[w]);
    }*/
    if (distTo[w] > distTo[v] + e.weight()) {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else                pq.insert(w, distTo[w]);
    }
```

3. 改进 3，可以指将我们此时使用多路堆来改变单路堆来加速，我们此时使用 IndexMultiwayMinPQ 代替 IndexPQ 就行了，我们的 IndexMultiwayMinPQ 使用的是 algs4.jar 包中的实现，也就是教科书的源码。

五、 实验结果

无优化：

```
<410-422> --- 3.16
<422-441> --- 8.54
<441-459> --- 8.94
<459-479> --- 9.00
<479-514> --- 17.03
<514-542> --- 13.60
<542-522> --- 13.60
<522-498> --- 10.63
<498-497> --- 1.00
<497-496> --- 3.16
<496-493> --- 7.00
<493-495> --- 7.00
<495-521> --- 9.22
<521-517> --- 2.24
<517-510> --- 4.24
<510-500> --- .00
Length of path: 399.7712512641509
-------The program running time is 103.0ms -------

Process finished with exit code 0
```

优化一：

```
<459-479> --- 9.00
<479-514> --- 17.03
<514-542> --- 13.60
<542-522> --- 13.60
<522-498> --- 10.63
<498-497> --- 1.00
<497-496> --- 3.16
<496-493> --- 7.00
<493-495> --- 7.00
<495-521> --- 9.22
<521-517> --- 2.24
<517-510> --- 4.24
<510-500> --- .00
Length of path: 399.7712512641509
-------The program running time is 37.0ms -------
```

优化二：

```
<522-498> --- 10.63
<498-497> --- 1.00
<497-496> --- 3.16
<496-493> --- 7.00
<493-495> --- 7.00
<495-521> --- 9.22
<521-517> --- 2.24
<517-510> --- 4.24
<510-500> --- .00
Length of path: 399.7712512641509
-------The program running time is 107.0ms -------

Process finished with exit code 0
```

优化三：

```
<422-441> --- 8.54
<441-459> --- 8.94
<459-479> --- 9.00
<479-514> --- 17.03
<514-542> --- 13.60
<542-522> --- 13.60
<522-498> --- 10.63
<498-497> --- 1.00
<497-496> --- 3.16
<496-493> --- 7.00
<493-495> --- 7.00
<495-521> --- 9.22
<521-517> --- 2.24
<517-510> --- 4.24
<510-500> --- .00
Length of path: 399.7712512641509
-------The program running time is 146.0ms -------
```

优化种类	原始	想法 1	想法 2	想法 3
时间	103ms	37ms	107ms	146ms

六、 结果分析

综上，我们完成了此次 map routing 的算法实验，我们比较了三种改进方案，并且亲自实现迪杰斯特拉算法。在进行测试的时候我们选择的起始节点为 0 节点，终节点为 500 节点，根据实验结果我们进行分析，对于 0—500 这一搜索的最小路径，改进一的效果是最好的。