算法上机实验报告

实验一 渗透问题

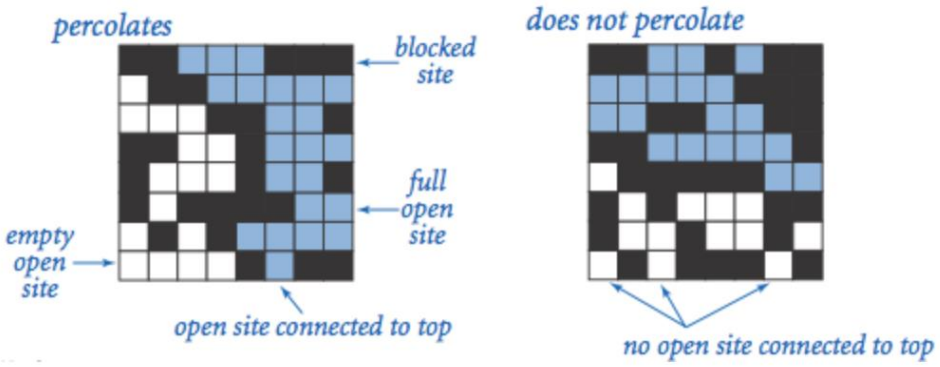
一、 实验内容

给定由随机分布的绝缘材料和金属材料构成的组合系统: 金属材料占多大比例才能使组合系 统成为电导体? 给定一个表面有水的多孔景观（或下面有油），水将在什么条件下能够通 过底部排出（或油渗透到表面）? 科学家已经定义了一个称为渗透（percolation）的抽象过程 来模拟这种现象。

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、 过程分析

建立模型: 我们使用 N×N 网格点来模型化一个渗透系统。 每个格点或是 open 格点或是 blocked 格点。 一个 full site 是一个 open 格点，它可以通过一系列的邻近（左、右、上、下）open 格 点连通到顶行的一个 open 格点。如果在底行中存在一个 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。）



当 N 足够大时，存在阈值 p*，使得当 p p* 时，随机 N× N 网格几乎总是渗透。

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

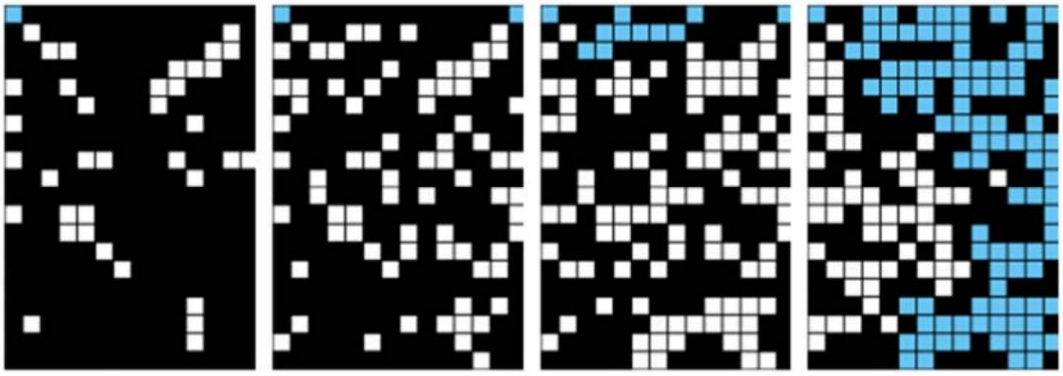
```
public class Percolation {
   public Percolation(int N)                // create N-by-N grid, with all sites blocked
   public void open(int i, int j)           // open site (row i, column j) if it is not already
   public boolean isOpen(int i, int j)      // is site (row i, column j) open?
   public boolean isFull(int i, int j)      // is site (row i, column j) full?
   public boolean percolates()              // does the system percolate?
   public static void main(String[] args)   // test client, optional
}
```

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(), and 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。



50 open sites *100 open sites* *150 open sites* *204 open sites*

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 xt 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。

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

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:

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

通过创建数据类型 PercolationStats 来执行一系列计算实验,包含以下 API:

```
public class PercolationStats {
   public PercolationStats(int N, int T)     // perform T independent computational experiments
on an N-by-N grid
   public double mean()                      // sample mean of percolation threshold
   public double stddev()                    // sample standard deviation of percolation threshold
   public double confidenceLo()              // returns lower bound of the 95% confidence interval
   public double confidenceHi()              // returns upper bound of the 95% confidence interval
   public static void main(String[] args)    // test client, described below
}
```

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 μ、标准差 σ 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。

Example values after creating PercolationStats(200, 100)
mean() = 0.5929934999999997
stddev() = 0.00876990421552567
confidenceLow() = 0.5912745987737567
confidenceHigh() = 0.5947124012262428

Example values after creating PercolationStats(200, 100)
mean() = 0.592877
stddev() = 0.009990523717073799
confidenceLow() = 0.5909188573514536
confidenceHigh() = 0.5948351426485464

Example values after creating PercolationStats(2, 100000)
mean() = 0.6669475
stddev() = 0.11775205263262094
confidenceLow() = 0.666217665216461
confidenceHigh() = 0.6676773347835391

三、 实验代码

Percolation.java 与 PercolationStats.java

其中第一个文件是处理连通性的，包含对一个图的所有方法，其中中用一个 flattenGrid()方法使得二维数组变成一维数组，此时便可以使用并查集处理连通性。

第二个文件说进行数学统计的，设置矩阵为 200×200 的正方形，循环次数为 100，可以得出较为准确的阈值。

Percolation.java

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
```

```java
//

import edu.princeton.cs.algs4.StdOut;
import edu.princeton.cs.algs4.WeightedQuickUnionUF;

public class Percolation {
    private boolean[][] grid;
    private WeightedQuickUnionUF wqfGrid;
    private WeightedQuickUnionUF wqfFull;
    private int gridSize;
    private int gridSquared;
    private int virtualTop;
    private int virtualBottom;
    private int openSites;

    public Percolation(int n) {
        if (n <= 0) {
            throw new IllegalArgumentException("N must be > 0");
        } else {
            this.gridSize = n;
            this.gridSquared = n * n;
            this.grid = new boolean[this.gridSize][this.gridSize];
            this.wqfGrid = new WeightedQuickUnionUF(this.gridSquared + 2);
            this.wqfFull = new WeightedQuickUnionUF(this.gridSquared + 1);
            this.virtualBottom = this.gridSquared + 1;
            this.virtualTop = this.gridSquared;
            this.openSites = 0;
        }
    }

    public void open(int row, int col) {
        this.validateSite(row, col);
        int shiftRow = row - 1;
        int shiftCol = col - 1;
        int flatIndex = this.flattenGrid(row, col) - 1;
        if (!this.isOpen(row, col)) {
            this.grid[shiftRow][shiftCol] = true;
            ++this.openSites;
            if (row == 1) {
                this.wqfGrid.union(this.virtualTop, flatIndex);
                this.wqfFull.union(this.virtualTop, flatIndex);
            }

            if (row == this.gridSize) {
```

```
            this.wqfGrid.union(this.virtualBottom, flatIndex);
        }

        if (this.isOnGrid(row, col - 1) && this.isOpen(row, col - 1)) {
            this.wqfGrid.union(flatIndex, this.flattenGrid(row, col - 1) - 1);
            this.wqfFull.union(flatIndex, this.flattenGrid(row, col - 1) - 1);
        }

        if (this.isOnGrid(row, col + 1) && this.isOpen(row, col + 1)) {
            this.wqfGrid.union(flatIndex, this.flattenGrid(row, col + 1) - 1);
            this.wqfFull.union(flatIndex, this.flattenGrid(row, col + 1) - 1);
        }

        if (this.isOnGrid(row - 1, col) && this.isOpen(row - 1, col)) {
            this.wqfGrid.union(flatIndex, this.flattenGrid(row - 1, col) - 1);
            this.wqfFull.union(flatIndex, this.flattenGrid(row - 1, col) - 1);
        }

        if (this.isOnGrid(row + 1, col) && this.isOpen(row + 1, col)) {
            this.wqfGrid.union(flatIndex, this.flattenGrid(row + 1, col) - 1);
            this.wqfFull.union(flatIndex, this.flattenGrid(row + 1, col) - 1);
        }

    }
}

public boolean isOpen(int row, int col) {
    this.validateSite(row, col);
    return this.grid[row - 1][col - 1];
}

public boolean isFull(int row, int col) {
    this.validateSite(row, col);
    return this.wqfFull.connected(this.virtualTop, this.flattenGrid(row, col) - 1);
}

public int numberOfOpenSites() {
    return this.openSites;
}

public boolean percolates() {
    return this.wqfGrid.connected(this.virtualTop, this.virtualBottom);
}
```

```
    public static void main(String[] args) {
        int size = Integer.parseInt(args[0]);
        Percolation percolation = new Percolation(size);
        int argCount = args.length;

        for(int i = 1; argCount >= 2; i += 2) {
            int row = Integer.parseInt(args[i]);
            int col = Integer.parseInt(args[i + 1]);
            StdOut.printf("Adding row: %d  col: %d %n", new Object[]{row, col});
            percolation.open(row, col);
            if (percolation.percolates()) {
                StdOut.printf("%nThe System percolates %n", new Object[0]);
            }

            argCount -= 2;
        }

        if (!percolation.percolates()) {
            StdOut.print("Does not percolate %n");
        }

    }

    private int flattenGrid(int row, int col) {
        return this.gridSize * (row - 1) + col;
    }

    private void validateSite(int row, int col) {
        if (!this.isOnGrid(row, col)) {
            throw new IndexOutOfBoundsException("Index is out of bounds");
        }
    }

    private boolean isOnGrid(int row, int col) {
        int shiftRow = row - 1;
        int shiftCol = col - 1;
        return shiftRow >= 0 && shiftCol >= 0 && shiftRow < this.gridSize && shiftCol <
this.gridSize;
    }
}
```

PercolationStats.java

```
//
// Source code recreated from a .class file by IntelliJ IDEA
```

```java
// (powered by FernFlower decompiler)
//

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

public class PercolationStats {
    private int trialCount;
    private double[] trialResults;

    public PercolationStats(int n, int trials) {
        if (n > 0 && trials > 0) {
            int gridSize = n;
            this.trialCount = trials;
            this.trialResults = new double[this.trialCount];

            for(int trial = 0; trial < this.trialCount; ++trial) {
                Percolation percolation = new Percolation(gridSize);

                int openSites;
                while(!percolation.percolates()) {
                    openSites = StdRandom.uniform(1, gridSize + 1);
                    int col = StdRandom.uniform(1, gridSize + 1);
                    percolation.open(openSites, col);
                }

                openSites = percolation.numberOfOpenSites();
                double result = (double)openSites / (double)(gridSize * gridSize);
                this.trialResults[trial] = result;
            }

        } else {
            throw new IllegalArgumentException("N and T must be <= 0");
        }
    }

    public double mean() {
        return StdStats.mean(this.trialResults);
    }

    public double stddev() {
        return StdStats.stddev(this.trialResults);
    }
```

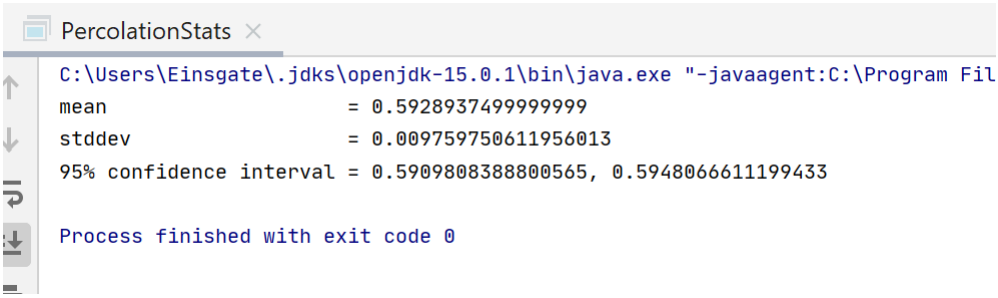
```
    public double confidenceLo() {
        return this.mean() - 1.96D * this.stddev() / Math.sqrt((double)this.trialCount);
    }

    public double confidenceHi() {
        return this.mean() + 1.96D * this.stddev() / Math.sqrt((double)this.trialCount);
    }

    public static void main(String[] args) {
        int gridSize = 200;
        int trialCount = 100;
        if (args.length >= 2) {
            gridSize = Integer.parseInt(args[0]);
            trialCount = Integer.parseInt(args[1]);
        }

        PercolationStats ps = new PercolationStats(gridSize, trialCount);
        String confidence = ps.confidenceLo() + ", " + ps.confidenceHi();
        StdOut.println("mean                    = " + ps.mean());
        StdOut.println("stddev                   = " + ps.stddev());
        StdOut.println("95% confidence interval = " + confidence);
    }
}
```

四、 执行结果



五、 实验总结

该实验通过并查集的算法来模拟对于一个渗透系统的阈值，该实验难点在于如何将一个二维矩阵转换成一个一维数组并且使用并查集来处理连通性。其中，随机生成函数也起到了很大作用，主要用于多次实验的估算以及置信区间的计算。

实验二 排序算法性能比较

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、 过程分析

对各种排序算法进行时间复杂度和空间复杂度分析，得出 O(N)表达式，空间根据占用内存和 JVM 分配内存实现。但由于 Java 具有自动垃圾回收功能，所以调用内部函数实现内存十分不便，于是根据编写程序给变量分配到内存决定使用多少空间。

三、 实验代码

四、

```
import sort.*;
import java.util.Random;
import java.util.Scanner;

public class Test {
    private long[] timeList;
    private long averageTime;

    public void time(String alg, Double[] a, int i) {
        long startTime = System.nanoTime(); //获取开始时间
        if (alg.equals("Insertion")) Insertion.sort(a);
        else if (alg.equals("Merge")) Merge.sort(a);
```

```
        else if (alg.equals("MergeBU")) MergeBU.sort(a);
        else if (alg.equals("Quick")) Quick.sort(a);
        else if (alg.equals("Quick3way")) Quick3way.sort(a);
        long endTime = System.nanoTime();  //获取结束时间
        timeList[i] = (endTime - startTime)/1000;
    }

    public void script(String alg, int N, int T) {
        long totalTime = 0;
        Double[] list = new Double[N];
        timeList = new long[T];
        Random rs = new Random();
        for (int i = 0; i < T; ++i) {
            for (int j = 0; j < N; ++j) {
                list[j] = rs.nextDouble();
            }
            time(alg, list, i);
            totalTime += timeList[i];
        }
        averageTime = totalTime/T;
    }

    public void show(String alg, int T) {
        System.out.println("<------使用算法：" + alg + "------>");
        for (int i = 0; i < T; ++i) {
            System.out.println("@@Run" + (i+1) + ": " + "time:  " +
                    timeList[i] );//+ "微秒"
        }
        System.out.println("<------平均运行时间：" + averageTime +
                "微秒------>");
    }

    public static void main(String[] args) {
        Test example = new Test();
        int N = 50000;
        int T = 10;
        Scanner sc = new Scanner(System.in );
        N = sc.nextInt();
        T = sc.nextInt();
        example.script("Merge", N,T);
        example.show("Merge", T);
        example.script("MergeBU", N,T);
        example.show("MergeBU", T);
        example.script("Quick", N,T);
```

```
            example.show("Quick", T);
            example.script("Quick3way", N,T);
            example.show("Quick3way", T);
            example.script("Insertion", N,T);
            example.show("Insertion", T);
        }
    }
```

N=100,T=100

五、 执行结果

<------使用算法：Merge------><------平均运行时间：94 微秒------>

<------使用算法：MergeBU------><------平均运行时间：141 微秒------>

<------使用算法：Quick------><------平均运行时间：86 微秒------>

<------使用算法：Quick3way------><------平均运行时间：64 微秒------>

<------使用算法：Insertion------><------平均运行时间：398 微秒------>

六、 实验总结

不同的算法在取值范围不同时差异巨大：当数据较小时，不同的算法运行时间都很短，但当数据显著增大时，快排和归并排序明显快了很多。但当重复元素较多时，一般的快排却慢了很多，而三路快排却体现出了明显的优越性。但归并排序却因为额外开辟 O(N)的空间导致空间复杂度增大。

实验三 地图路由

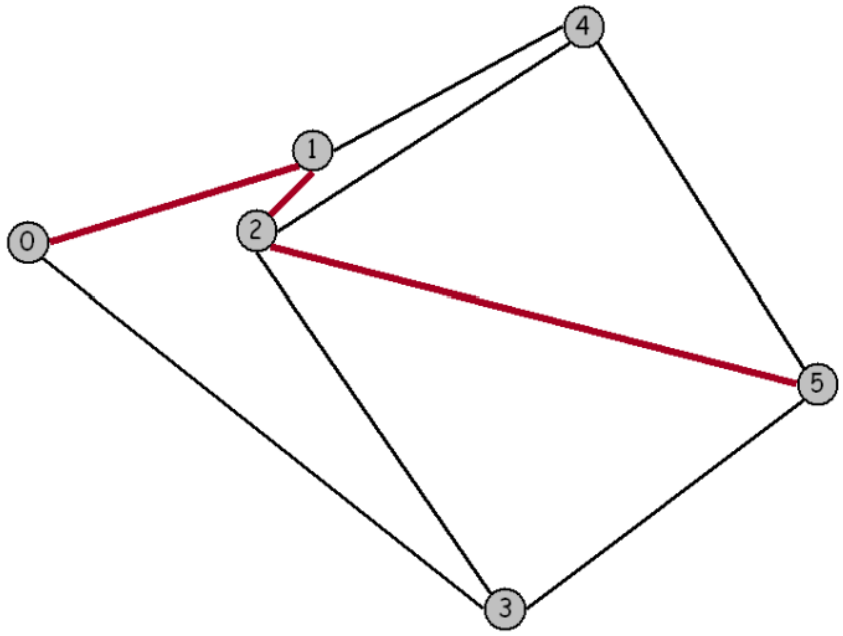
一、 实验内容

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。 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。 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
```



二、 过程分析

Dijkstra 算法。Dijkstra 算法是最短路径问题的经典解决方案。教科书 4.4 节描述了该算法。 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。

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。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 4.4 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 程序中的多路堆(Multiway heaps, Section 2.4)。 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。

三、 实验代码

四、

```
package MR;

import edu.princeton.cs.algs4.Edge;
import edu.princeton.cs.algs4.EdgeWeightedGraph;
import edu.princeton.cs.algs4.In;
import edu.princeton.cs.algs4.Stack;
import edu.princeton.cs.algs4.StdOut;
import java.util.Scanner;
```

```
class point2D {
   private double x;
   private double y;

   public point2D(double x, double y) {
      this.x = x;
      this.y = y;
   }

   public double distanceTo(point2D that) {
      double dx = this.x - that.x;
      double dy = this.y - that.y;
      return Math.sqrt(dx * dx + dy * dy);
   }
}

class DijkstraUndirectedSP {
   private double[] distTo;
   private Edge[] edgeTo;
   private indexPQ<Double> pq;
   private EdgeWeightedGraph mGraph;
   private point2D[] point2DS;
   private int from;
   private int to;

   public DijkstraUndirectedSP(EdgeWeightedGraph G, point2D[] point) {
      point2DS = point;
      mGraph = G;
      for (Edge e : G.edges()) {
         if (e.weight() < 0)
            throw new IllegalArgumentException("权重不能为负数！");
      }
      distTo = new double[G.V()];
      edgeTo = new Edge[G.V()];
      for (int v = 0; v < G.V(); v++)
         distTo[v] = Double.POSITIVE_INFINITY;
      pq = new indexPQ<>(G.V());
   }

   public void setFrom(int from) {
      initDijkstra();
      this.from = from;
      distTo[from] = 0.0;
```

```
      pq.insert(from, distTo[from]);
    }

    public void setTo(int to) {
      this.to = to;
    }

    private void relax(Edge e, int v) {
      int w = e.other(v);
      double weight = distTo[v] + e.weight();
      // 优化：A*算法
      // 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
//    double weight = distTo[v] + e.weight() + point2DS[v].distanceTo(point2DS[to]) - point2DS[w].distanceTo(point2DS[to]) ;
      if (distTo[w] > weight) {
        distTo[w] = weight;
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else pq.insert(w, distTo[w]);
      }
    }

    public double distTo(int v) {
      return distTo[v];
    }

    public boolean hasPathTo(int v) {
      while (!pq.isEmpty()) {
        int x = pq.delMin();
        // 优化：找出目标节点就退出循环
        if (x == v) {
          return true;
        }
        for (Edge e : mGraph.adj(x))
          relax(e, x);
      }
      return distTo[v] < Double.POSITIVE_INFINITY;
    }

    public void initDijkstra() {
      for (int i = 0; i < mGraph.V(); i++) {
        if (pq.contains(i)) {
          pq.delete(i);
        }
```

```
            if (edgeTo[i] != null) {
               edgeTo[i] = null;
            }
            if (!Double.isInfinite(distTo(i))) {
               distTo[i] = Double.POSITIVE_INFINITY;
            }
         }
      }

      public Iterable<Edge> pathTo(int v) {
         if (!hasPathTo(v)) return null;
         Stack<Edge> path = new Stack<Edge>();
         int x = v;
         for (Edge e = edgeTo[v]; e != null; e = edgeTo[x]) {
            path.push(e);
            x = e.other(x);
         }
         return path;
      }
   }

   public class MapRouting {
      public static void init_Point(In in, point2D[] p) {
         for (int i = 0; i < p.length; i++) {
            int q = in.readInt();
            int x = in.readInt();
            int y = in.readInt();
            p[i] = new point2D(x, y);
         }

      }

      public static void init_Graph(In in, int b, EdgeWeightedGraph e, point2D[] a) {
         for (int i = 0; i < b; i++) {
            int q = in.readInt();
            int l = in.readInt();
            double weight = a[q].distanceTo(a[l]);
            e.addEdge(new Edge(q, l, weight));
         }

      }

      public static void findRoute(EdgeWeightedGraph mDigraph, point2D[] mpoint) {
         Scanner in = new Scanner(System.in);
```

```
        DijkstraUndirectedSP msp = new DijkstraUndirectedSP(mDigraph, mpoint);

        while (true) {
            int from = in.nextInt();
            int to = in.nextInt();
            msp.setFrom(from);
            msp.setTo(to);
            boolean boo = msp.hasPathTo(to);
            if (boo) {
                StdOut.println("最短路径为：");
                for (Edge e : msp.pathTo(to)) {
                    StdOut.println(e + " ");
                }
                StdOut.println(from + " --> " + to + "的最短路径长度为：" + msp.distTo(to));
                StdOut.println();
            } else {
                StdOut.println("不存在这样一条路径");
            }
        }
    }

    public static void main(String[] args) {
        In in = new In("D:\\Java\\coding\\Arithemtic\\Exp\\exp3\\Data.txt");

        int a = in.readInt();
        int b = in.readInt();

        EdgeWeightedGraph mDigraph = new EdgeWeightedGraph(a);
        point2D[] mPoints = new point2D[a];
        init_Point(in, mPoints);
        init_Graph(in, b, mDigraph, mPoints);

        findRoute(mDigraph, mPoints);
    }
}
```

五、 执行结果

2 7

最短路径为:

2-19 15.00000

19-27 13.03840

27-26 1.41421

26-31 7.61577

31-25 8.54400

25-225.38516

22-21 9.00000

21-24 19.02630

24-35 17.72005

35-41 11.40175

41-47 37.12142

47-4810.00000

30-48 8.06226

29-30 1.00000

14-29 25.63201

14-76.70820

2 -->7 的最短路径长度为:196.6695523224872

六、 实验总结

该算法实验使用迪杰斯特拉算法求得最短路径，该算法本质是一种贪心算法，利用局部最优解得到最终的最优解。本次实验利用问题的欧式几何来进一步减少搜索时间，即 A*算法。 对于本地图， 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。

实验四 文本索引

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。你的程序应该使用两个文件名作为命令行参数：文本文件（我们称为语料库）和包含查询的文件。过程分析

二、 过程分析

假设这两个文件只包含小写字母、空格和换行符，查询文件中的查询由换行符分隔。 这不是一个限制，因为你可以使用一个过滤器将任何文件转换为此格式。 你的程序应该读取语料库，将其存储为（可能巨大）字符串，并可能为其创建索引，如下所述。然后它应该逐个读取查询(假设在命令行中的第二个命名文件中，每行有一个查询)，并打印出语料库中每个查询在文本文件中首次出现的位置。对于由如下内容构成的 corpus 文件 it was the best of times it was the worst of times it was the age of wisdom it was the age of foolishness it was the epoch of belief it was the epoch of incredulity it was the season of light it was the season of darkness it was the spring of hope it was the winter of despair 以及如下内容构成的 query 文件 wisdom season

age of foolishness age of fools 查询结果如下： 18 wisdom 40 season 22 age of foolishness -- age of fools 解决这个问题有很多种不同的方法，这些方法在实现方便性，空间要求和时间要求方面 都有不同的特点。 此任务的一部分是吸收此信息，以帮助你确定使用哪种方法以及如何将其 应用于此特定任务。 你可以从本书中基于程序 3.15 的蛮力搜索实现开始。 也就是说，不建立索引：只需搜索每个查询字符串的语料库即可。 如果语料库很小或者查询不多，这个解决方案是很好的。 但是，当语料库庞大且查询量大的时候，这种方法太慢了，因此，你需要实现一个更快的解决方案。一种快速搜索的方法是在语料库（每个字符位置一个指针）上进行指针排序，然后使用折半搜索。 如果你想采用这种方法，可以使用标准 C 库中的 qsort 和 bsearch 函数。这种方法的主要挑战是完全理解程序 3.17; 开发一个类似的程序来构建指针索引，按照排序顺序访 问关键字（如图 3.13 所示）；并找出调用 bsearch 的必要接口使用索引执行查询。特别地， 对于进行排序和搜索你需要适当的“比较”功能（不同的！）。另一种可能的方法是从程序 12.10 开始。这段代码基本上提供了一个完整的解决方案，但为了使其正常工作，你必须进行一些小的更改，因为它们在许多细节上与此处指定的问题 不同，并且因为缺少各种小的代码。 你可能需要编辑此代码，或从头开始编写自己的代码。再次，必须仔细考虑“比较”功能。

三、 实验代码

四、

```
package WBSY;
```

```java
import java.io.File;
import java.io.FileInputStream;
import edu.princeton.cs.algs4.In;
import edu.princeton.cs.algs4.StdOut;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;

// BM 字符串搜索算法
// 坏字符算法（bad-character shift）和好后缀算法（good-suffix shift）
class BoyerMoore {
   private final int R;
   private int[] right;
   private String pat;

   public BoyerMoore(String pat) {
      this.R = 256;
      this.pat = pat;
      right = new int[R];
      for (int c = 0; c < R; c++)
         right[c] = -1;
      for (int j = 0; j < pat.length(); j++)
         right[pat.charAt(j)] = j;
   }

   public int search(String txt) {
      int m = pat.length();
      int n = txt.length();
      int skip;
      for (int i = 0; i <= n - m; i += skip) {
         skip = 0;
         for (int j = m - 1; j >= 0; j--) {
            if (pat.charAt(j) != txt.charAt(i + j)) {
               skip = Math.max(1, j - right[txt.charAt(i + j)]);
               break;
            }
         }
         if (skip == 0) return i;// 找到了，则返回第一个字符的位置
      }
      return -1; // 没找到 返回-1
   }

   public int searchPos(String txt, int n) {
```

```
      int count = 0;
      char[] tx = txt.toCharArray();
      for (int i = 0; i < n; i++) {
        if (tx[i] == ' ') count++;
      }
      return count;
    }
}

public class TextIndex {
    public static void main(String[] args) {
      String txt1 = readTxt();
//       StdOut.println(txt1);
//       File file = new File("D:\\Java\\coding\\Arithemtic\\Exp\\exp4\\p1.txt");
      File file = new File("D:\\Java\\coding\\Arithemtic\\Exp\\exp4\\p2.txt");
      FileInputStream is = null;
      try {
        if (file.length() != 0) {
          is = new FileInputStream(file);
          InputStreamReader streamReader = new InputStreamReader(is);
          BufferedReader reader = new BufferedReader(streamReader);
          String line;
          while ((line = reader.readLine()) != null) {
            BoyerMoore boyermoore = new BoyerMoore(line);
            int offset = boyermoore.search(txt1);
            if (offset == -1)
              StdOut.println("-- "+line);// 没找到
            else {
              int pos = boyermoore.searchPos(txt1,offset);
              StdOut.print(pos+1);
              StdOut.println(' '+line);
            }
          }
          reader.close();
          is.close();
        } else {
          StdOut.println("打开文件为空");
        }
      } catch (Exception e) {
        e.printStackTrace();
      }
    }

    public static String readTxt() {
```

```
//       File file = new File("D:\\Java\\coding\\Arithemtic\\Exp\\exp4\\test1.txt");
      File file = new File("D:\\Java\\coding\\Arithemtic\\Exp\\exp4\\test2.txt");
      FileInputStream is = null;
      StringBuilder stringBuilder = null;
      try {
        if (file.length() != 0) {
          is = new FileInputStream(file);
          InputStreamReader streamReader = new InputStreamReader(is);
          BufferedReader reader = new BufferedReader(streamReader);
          String line;
          stringBuilder = new StringBuilder();
          while ((line = reader.readLine()) != null) {
            stringBuilder.append(line);
            stringBuilder.append(' ');
          }
          reader.close();
          is.close();
        } else {
          StdOut.println("文件为空");
        }

      } catch (Exception e) {
        e.printStackTrace();
      }
      return String.valueOf(stringBuilder);
    }
}
```

五、 执行结果

18 wisdom

40 season

22 age of foolishness

六、 实验总结

该实验使用坏字符算法（bad-character shift）和好后缀算法（good-

suffix shift) 来进行查找匹配。其中，在搜索时采用了折半查找的方法，使用双指针(两个相向而行的指针)进行遍历，使得算法复杂度有了显著改善。