实验三 地图路由（Map Rputing）

一、 实验目的

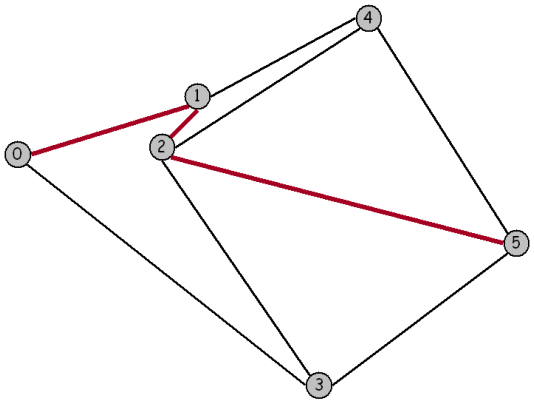
实现经典的 Dijkstra 最短路径算法，并对其进行优化。 这种算法广泛应用于地理信息系统（GIS），包括 MapQuest 和基于 GPS 的汽车导航系统。

二、 实验内容

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。

```
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 算法是最短路径问题的经典解决方案。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。

本次实验，我们需要优化 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。优化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。

三、 优化方法

1. 减少检查的顶点数量

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。程序实现如下：

```
1.  public boolean hasPathTo(int v) {
2.          while (!pq.isEmpty()) {
3.              int x = pq.delMin();
4.              if (x == v) {//优化 1 进行优化
5.                  return true;
6.              }
7.              for (Edge e : mGraph.adj(x))
```

```
8.                 relax(e, x);
9.         }
10.        return distTo[v] < Double.POSITIVE_INFINITY;
11. }
```

其中加粗的 if 语句用来判断是否已找到目标节点。如果已经找到，则停止搜索。

2. 欧氏距离松弛边

我们可以利用问题的欧式几何来进一步减少搜索时间。对于一般图，常规的 Dijkstra 算法通过将 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。程序实现如下：

```
1.  private void relax(Edge e, int v) {
2.         int w = e.other(v);
3.         double weight = distTo[v] + e.weight() + nodes[v].euDist(nodes[d]) -
    nodes[w].euDist(nodes[d]);//(语句 1)优化 2 进行优化
4.         //double weight = distTo[v] + e.weight();//(语句 2)优化 2 未优化的语句
5.         if (distTo[w] > weight) {
6.             distTo[w] = weight;
7.             edgeTo[w] = e;
8.             if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
9.             else pq.insert(w, distTo[w]);
10.        }
11. }
```

此方法改变了 weight 的更新值来优化。语句 1 为优化后的语句，语句 2 为未优化的语句。选择语句 1 或语句 2 即可进行优化或不优化。

3. 使用多路优先堆

我们也可以考虑使用更快的优先队列作为基础数据结构。本次实验中我选择使用 Sedgewick 在教材中提出的多路最小优先堆（IndexMultiwayMinPQ）。程序实现如下：

```
1.  public DijkstraUndirectedSP(EdgeWeightedGraph G, node2D[] node) {
2.         nodes = node;
3.         mGraph = G;
4.         for (Edge e : G.edges()) {
5.             if (e.weight() < 0)
6.                 throw new IllegalArgumentException("edge " + e + " has negative
    weight");
7.         }
8.         distTo = new double[G.V()];
9.         edgeTo = new Edge[G.V()];
10.        for (int v = 0; v < G.V(); v++)
11.            distTo[v] = Double.POSITIVE_INFINITY;
12.        distTo[s] = 0.0;
13.        pq = new IndexMultiwayMinPQ<Double>(G.V(), 2);//(语句 1)优化 3 进行优化
14.        //pq = new indexPQ<Double>(G.V());//(语句 2)
15. }
```

语句 1 为使用多路最小优先堆的数据结构，是优化后的算法；语句 2 为普通的优先队列。我们可以选择语句 1 或语句 2 来进行优化或不优化。使用前记得在类中进行参数声明。

四、 源代码

下附代码使用了优化三的方法进行优化。其他优化方法只需参照上述的优化方法对代码稍作改动即可。

```
1.  //Opt3.java
2.  import edu.princeton.cs.algs4.Edge;
3.  import edu.princeton.cs.algs4.EdgeWeightedGraph;
4.  import edu.princeton.cs.algs4.In;
5.  import edu.princeton.cs.algs4.IndexMultiwayMinPQ;
6.  import edu.princeton.cs.algs4.Stack;
7.  import edu.princeton.cs.algs4.StdOut;
8.
9.  import java.util.Scanner;
10.
11.
12. public class Opt3 {
13.     public static void main(String[] args) {
14.         In in = new In("D:\\Mdesktop\\usa.txt");
15.         int a = in.readInt();
16.         int b = in.readInt();
17.         EdgeWeightedGraph mEWgraph = new EdgeWeightedGraph(a);
18.         node2D[] mPoints = new node2D[a];
19.         createPoint(in, mPoints);//创建位置数据
20.         createGraph(in, b, mEWgraph, mPoints);//创建图
21.         long StartTime = System.nanoTime();
22.         Route(mEWgraph, mPoints);
23.         long endTime = System.nanoTime();
24.         StdOut.println("时间占用->:" + (endTime - StartTime) / 1000000.0 + " ms");
25.     }
26.
27.     public static void createPoint(In in, node2D[] p) {
28.         for (int i = 0; i < p.length; i++) {
29.             int q = in.readInt();
30.             int x = in.readInt();
31.             int y = in.readInt();
32.             p[i] = new node2D(x, y);
33.         }
34.     }
35.
36.     public static void createGraph(In in, int b, EdgeWeightedGraph g, node2D[] a)
    {
37.         for (int i = 0; i < b; i++) {
38.             int q = in.readInt();
39.             int l = in.readInt();
40.             double weight = a[q].euDist(a[l]);
41.             g.addEdge(new Edge(q, l, weight));
42.         }
43.
44.     }
45.
46.     public static void Route(EdgeWeightedGraph mEWgraph, node2D[] mPoint) {
47.         Scanner in = new Scanner(System.in);
48.         DijkstraUndirectedSP mSP = new DijkstraUndirectedSP(mEWgraph, mPoint);
49.
50.         int s = in.nextInt();
51.         int d = in.nextInt();
52.         mSP.setSource(s);
53.         if (mSP.hasPathTo(d)) {
54.             double sum = 0.0;
55.             StdOut.println("最短路径为: ");
56.             for (Edge e : mSP.pathTo(d)) {
57.                 StdOut.println(e);
58.                 sum += e.weight();
59.             }
60.             StdOut.println(s + "--->" + d + "的最短路径长度为: " + sum);
61.
```

```
62.             StdOut.println();
63.         }
64.         else {
65.             StdOut.println("不存在一条路径");
66.         }
67.     }
68.
69.
70. }
71.
72. class DijkstraUndirectedSP {
73.     private double[] distTo;
74.     private Edge[] edgeTo;
75.     private IndexMultiwayMinPQ<Double> pq;
76.     private EdgeWeightedGraph mGraph;
77.     private node2D[] nodes;
78.     private int s;
79.     private int d;
80.
81.     public DijkstraUndirectedSP(EdgeWeightedGraph G, node2D[] node) {
82.         nodes = node;
83.         mGraph = G;
84.         for (Edge e : G.edges()) {
85.             if (e.weight() < 0)
86.                 throw new IllegalArgumentException("edge " + e + " has negative
    weight");
87.         }
88.         distTo = new double[G.V()];
89.         edgeTo = new Edge[G.V()];
90.         for (int v = 0; v < G.V(); v++)
91.             distTo[v] = Double.POSITIVE_INFINITY;
92.         distTo[s] = 0.0;
93.         pq = new IndexMultiwayMinPQ<Double>(G.V(), 2);//(语句 1)优化 3 进行优化
94.         //pq = new indexPQ<Double>(G.V());//(语句 2)
95.     }
96.
97.
98.     private void relax(Edge e, int v) {
99.         int w = e.other(v);
100.      /*    double weight = distTo[v] + e.weight() + nodes[v].euDist(nodes[d]) -
    nodes[w].euDist(nodes[d]);*///（语句 1）优化 2 进行优化
101.         double weight = distTo[v] + e.weight();//（语句 2）优化 2 不进行优化
102.         if (distTo[w] > weight) {
103.             distTo[w] = weight;
104.             edgeTo[w] = e;
105.             if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
106.             else pq.insert(w, distTo[w]);
107.         }
108.     }
109.
110.     public void setSource(int s) {
111.         initDijkstra();
112.         this.s = s;
113.         distTo[s] = 0.0;
114.         pq.insert(s, distTo[s]);
115.     }
116.
117.
118.     public double distTo(int v) {
119.         return distTo[v];
120.     }
121.
122.     public boolean hasPathTo(int v) {
123.         while (!pq.isEmpty()) {
124.             int x = pq.delMin();
125.         /*    if (x == v) {//优化 1 不进行优化
126.                 return true;
127.
```

```
128.             }*/
129.             for (Edge e : mGraph.adj(x))
130.                 relax(e, x);
131.         }
132.         return distTo[v] < Double.POSITIVE_INFINITY;
133.     }
134.
135.     public void initDijkstra() {
136.         for (int i = 0; i < mGraph.V(); i++) {
137.             if (pq.contains(i)) {
138.                 pq.delete(i);
139.             }
140.             if (edgeTo[i] != null) {
141.                 edgeTo[i] = null;
142.             }
143.             if (!Double.isInfinite(distTo(i))) {
144.                 distTo[i] = Double.POSITIVE_INFINITY;
145.             }
146.         }
147.     }
148.
149.     public Iterable<Edge> pathTo(int v) {
150.         if (!hasPathTo(v)) return null;
151.         Stack<Edge> path = new Stack<Edge>();
152.         int x = v;
153.         for (Edge e = edgeTo[v]; e != null; e = edgeTo[x]) {
154.             path.push(e);
155.             x = e.other(x);
156.         }
157.         return path;
158.     }
159. }
160.
161. class node2D {
162.     private double x;
163.     private double y;
164.
165.     public node2D(double x, double y) {
166.         this.x = x;
167.         this.y = y;
168.     }
169.
170.     public double euDist(node2D that) {//欧氏距离
171.         double delta_x = this.x - that.x;
172.         double delta_y = this.y - that.y;
173.         return Math.sqrt(delta_x * delta_x + delta_y * delta_y);
174.     }
175. }
```

五、 运行结果比较

	Run1	Run2	Run3	Run4	Run5	Run6	Run7	Run8	Run9	Run10	Average
无优化	3253	2444	3452	2873	3352	2749	2661	2809	2818	2558	2896.9
优化一	2666	2625	2796	2471	2696	2763	2787	2762	2473	2430	2646.9
优化二	2440	2638	2578	2775	2709	2505	2843	2617	2397	2732	2623.4
优化三	2813	2680	2382	2691	2577	2979	2777	2441	2297	2831	2646.8
全优化	2501	2249	2253	2217	2490	2302	2442	2491	2302	2292	2353.9

表 5.1 各种优化方法运行时间比较表（单位：ms）

从表中可以看出，相较于无优化的情况，在运行时间上，优化一性能提

高了 8.63%，优化二性能提高了 9.44%，优化三性能提高了 8.63%。若同时使用三种优化方式，则性能提高高达 18.74%！三种优化方式优化性能显著。

六、 实验总结

本次实验主要是针对稀疏图优化 Dijkstra 算法。三种优化方法分别针对点、边及数据结构进行优化，实际操作并不难，只需对相应代码段进行优化即可。这次的实验让我感受到了对算法的优化的探求是永无止境的，一个小的优化的想法可能会带来性能上的巨大的提升。我们在学习算法的过程中也要秉承着批判精神，多思考，去尝试探求更好的算法！