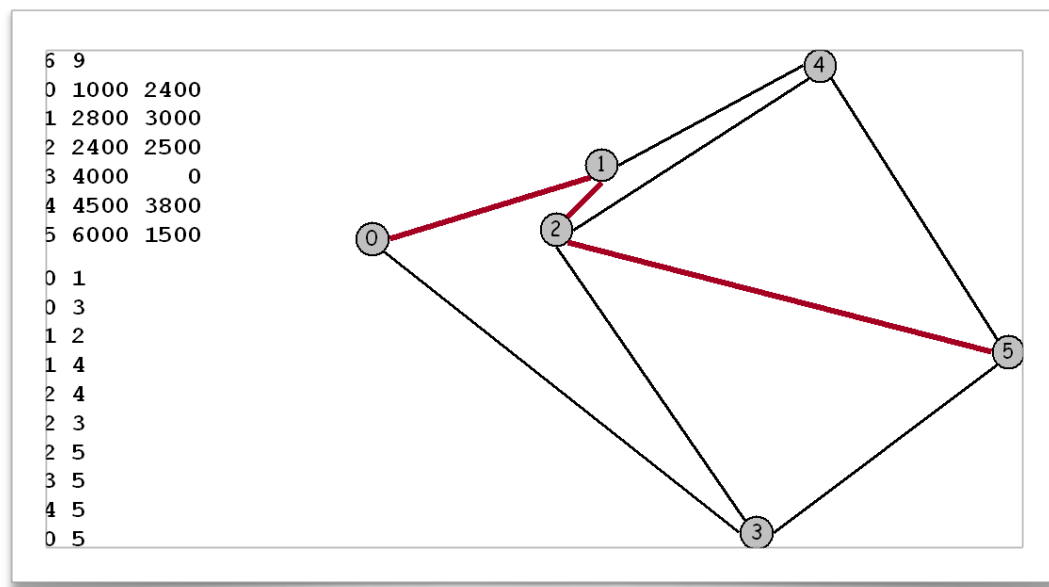


# 地图路由 (Map Routing)

## 一、题目

实现经典的 Dijkstra 最短路径算法，并对其进行优化。

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：



**Dijkstra 算法。** 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。

**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。

## 二、解决想法

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  $\infty$  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。

测试。美国大陆文件 [usa.txt](#) 包含 87,575 个交叉点和 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、代码实现

### 1. 建地理位置的坐标点。

```
public class ShortestPathDemo {  
    protected static int[] V;  
    protected static int[] X;  
    protected static int[] Y;  
  
    public static void formatArray(int v) {  
        ShortestPathDemo.V = new int[v];  
        ShortestPathDemo.X = new int[v];           //点的x坐标  
        ShortestPathDemo.Y = new int[v];           //点的y坐标  
    }  
}
```

## 2.构建地图

```
public static void main(String[] args) throws InterruptedException {
    Scanner input = null;
    try {
        input = new Scanner(new File( pathname: "usa.txt"));
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }

    int v = input.nextInt(); //读入顶点数
    int e = input.nextInt(); //读入边数

    //初始化数组
    formatArray(v);

    /*使用EdgeWeightedDigraph类的第一个构造函数，传入顶点个数v这一个参数
    * 构造出含有v个顶点的加权有向图
    * 此时图内仅有顶点，没有边*/
    EdgeWeightedDigraph map = new EdgeWeightedDigraph(v); //使用该类的第一个构造函数

    /*用三个数组分别存入文件中的数据，分别为
    * V[] 顶点的集合
    * X[] 顶点的横坐标
    * Y[] 顶点的纵坐标*/
    for (int i = 0; i < v; i++) {
        V[i] = input.nextInt();
        X[i] = input.nextInt();
        Y[i] = input.nextInt();
    }

    /*通过计算每个边的权值，向加权有向图map中加入所有边
    * 此时图就形成了*/
    for (int i = 0; i < e; i++) {
        int one = input.nextInt();
        int theOther = input.nextInt();
        double weight = getWeight(one, theOther);
        DirectedEdge directedEdge = new DirectedEdge(one, theOther, weight); //生成 (有向) 加权边
        map.addEdge(directedEdge); //向图中添加边
    }
}
```

### 3.使用改进的Dijkstra 算法 ( OptimizedDijkstra类 ) 进行测试

```
Scanner userInput = new Scanner(System.in);
StdOut.println("Input the starting point & destination point & times: ");

int usrStart = userInput.nextInt();
int usrEnd = userInput.nextInt();
int times = userInput.nextInt(); //运行次数

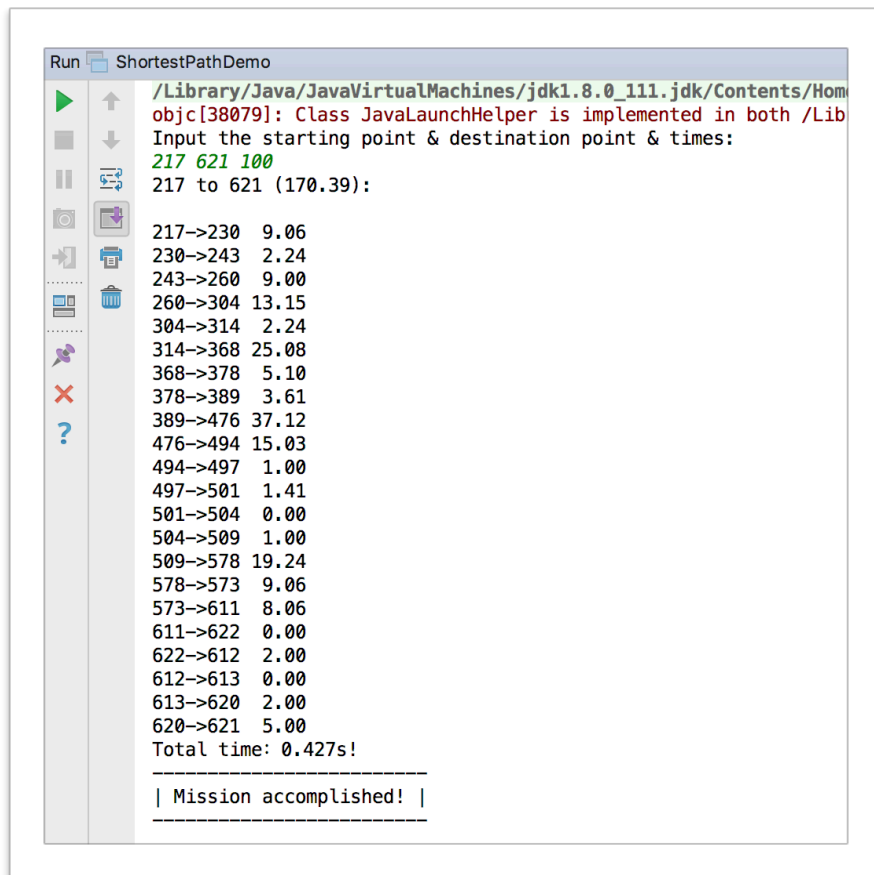
Stopwatch start = new Stopwatch(); //计时开始
int i = 0;

OptimizedDijkstra optimizedDijkstra = null;
for (; i++ < times; ) {
    optimizedDijkstra = new OptimizedDijkstra(map, usrStart, usrEnd);
}
double elapsedTime = start.elapsedTime();
StdOut.print(usrStart + " to " + usrEnd);
StdOut.printf(" (%.2f): \n\n", optimizedDijkstra.distTo(usrEnd));
if (optimizedDijkstra.hasPathTo(usrEnd)) {
    for (DirectedEdge de : optimizedDijkstra.pathTo(usrEnd)) {
        StdOut.println(de + " ");
    }
} else {
    StdOut.println(usrStart + " cannot reach " + usrEnd);
}

StdOut.println("Total time: " + elapsedTime + "s!");
StdOut.println("-----\n| Mission accomplished! |\n-----");
```

#### 4.测试结果

测试1：起始点编号217，目的点编号621，测试100次。获得路径和总时间。结果为可到达。

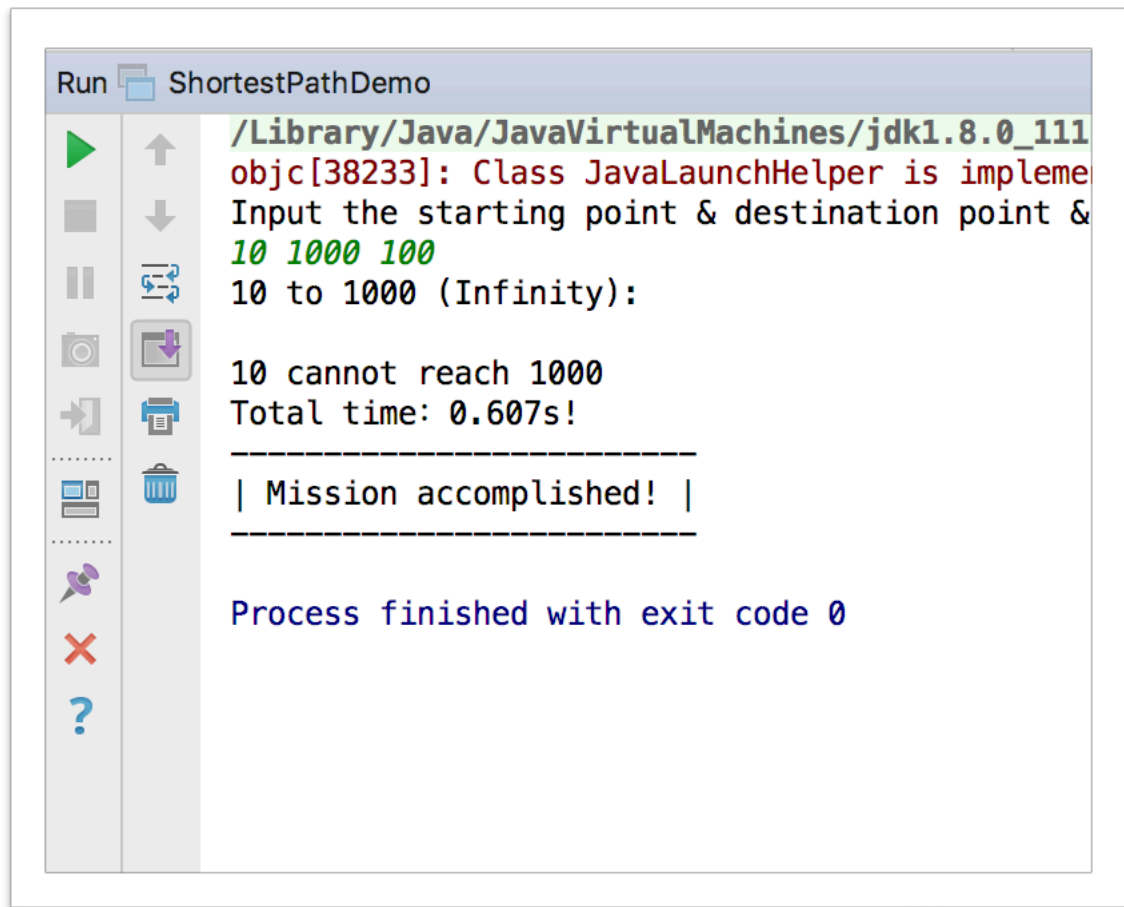


```
Run ShortestPathDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_111.jdk/Contents/Home
objc[38079]: Class JavaLaunchHelper is implemented in both /Lib
Input the starting point & destination point & times:
217 621 100
217 to 621 (170.39):

217->230 9.06
230->243 2.24
243->260 9.00
260->304 13.15
304->314 2.24
314->368 25.08
368->378 5.10
378->389 3.61
389->476 37.12
476->494 15.03
494->497 1.00
497->501 1.41
501->504 0.00
504->509 1.00
509->578 19.24
578->573 9.06
573->611 8.06
611->622 0.00
622->612 2.00
612->613 0.00
613->620 2.00
620->621 5.00
Total time: 0.427s!

-----
| Mission accomplished! |
-----
```

测试2：起始点编号10，目的点编号1000，测试100次。获得路径和总时间。结果为不可达



```
Run ShortestPathDemo
/Library/Java/JavaVirtualMachines/jdk1.8.0_111
objc[38233]: Class JavaLaunchHelper is implemented...
Input the starting point & destination point &
10 1000 100
10 to 1000 (Infinity):
10 cannot reach 1000
Total time: 0.607s!
-----
| Mission accomplished! |
-----

Process finished with exit code 0
```