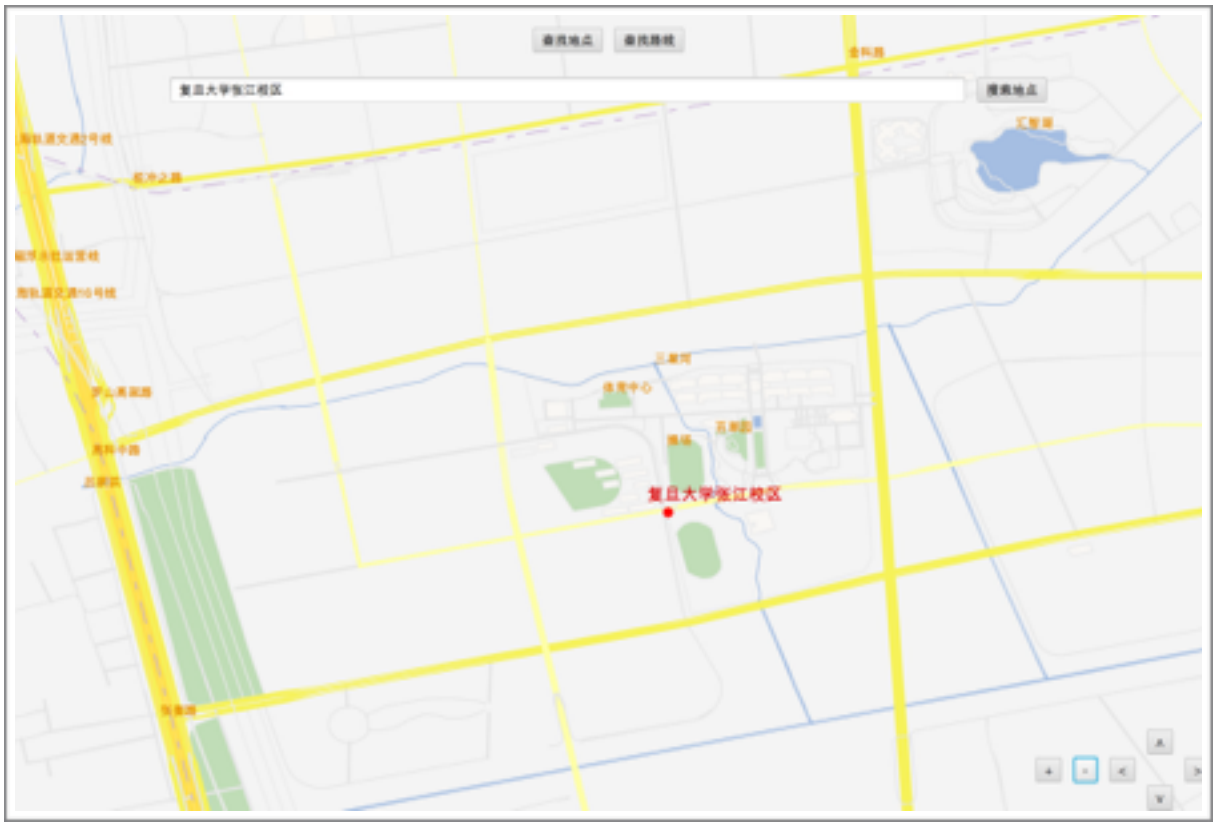
基于OpenStreetMapXML的地图



基于OpenStreetMapXML的地图

综述

本项目以Java语言实现，能够跨平台运行。界面没有使用传统的Java Swing工具包，因为Swing在不同平台上界面风格差异较大（且不美观），选取JavaFX作为地图的GUI工具包使得程序界面在不同平台上具有相同的Look & Feel，而且许多绘图函数较以往更加丰富。

本地地图数据利用OpenStreetMap导出的XML文件，程序实现了地图的移动、缩放、地点查找、最短路搜索。

XML文件中包含数以十万记的节点，要通过恰当的数据结构高效地实现视野内点的查找及绘制是非常具有挑战性的工作。本程序通过对不同点进行分类存入不同的数据结构，通过Map储存节点ID与对象的关系，通过地图的分块化处理实现视野范围内节点的查找及绘制已达到较高效率的地图显示。

文章组织

一、基本的数据结构

1. node对象在程序当中的存储方式
2. way对象在程序当中的存储方式

二、预处理过程

1. 文件的读取
2. 数据的分层存放
3. 地图区域的分块处理

三、地图绘制

1. 不同分层级别下的地图显示
2. 具体点和路线的处理
3. 地点名称的显示

四、程序界面

1. 绘图工具包的选择
2. 界面的设计

五、程序功能的实现

1. 单个顶点的查找
2. 最短路的查找

基本数据结构

类node

node的类定义中包含了顶点的id、x坐标（经度）、y坐标（纬度）以及顶点相关的tag，需要注意的是tag的存储使用的是一个HashMap，这样只要输入需要的tag名称就可快速获得对应的值（不过实际中由于除了name以外的tag并未发挥实际用处，因此未作存储）。

```
//node.java
String id;//储存ID
double x,y;//x = lon , y = lat
HashMap<String, String> tag = new HashMap<>();//储存tag
public node(String ID){}//构造函数
public node(String ID,double lat,double lon){}//构造函数
public node(String ID,double lat,double lon,String Name){}//构造函数
public double getx(){}//返回x
public double gety(){}//返回y
public HashMap<String, String> getTag(){}//返回tag
```

类way

way的类定义中除了同样包含路径的id和tag（不同于node，way的所有Tag都进行了储存）外，还包括所有经过顶点的id，存放于一个Vector<String>中，记为ndid。这样同时就记录了顶点间的顺序。出于面向对象程序设计的考虑，无论是node或是way，都进行了妥善的封装，对象内的成员变量以private的形式进行隐藏，需要时通过接口函数进行访问。

```
//way.java
String id;//储存ID
Vector<String> ndid = new Vector<>();//按顺序储存way上node的id
HashMap<String, String> tag = new HashMap<>();//储存tag
```

public way(String ID){}//构造函数

public void insertNode(String ID){}//添加node的id

public void insertTag(String k, String v){}//添加tag

public HashMap<String, String> getTag(){}//返回tag

public Vector<String> getNdid(){}//返回way上node的id

public String getId(){}//返回way的id

Relation

由于在地图绘制过程中relation并未发挥实际用处，因而虽然原本也生成了相关的类和对象，但在后续版本中被废弃。

预处理过程

主程序运行前，先使用mapFileProcessor类进行文件处理。该类读入原始的map.osm文件，将多余信息去除后，得到精简版的data.osm文件。由于地图构建过程中并未用到relation的信息，所原来处理的relation信息在后期维护中直接删除了。该文件有利于开发时进行数据分析和搜集。其中bound、node、way分别精简如下：

<bounds minlat="31.1740000" minlon="121.5218000" maxlat="31.2433000" maxlon="121.6475000"/>

<node

<id 61104104

<lon 121.5791369

<lat 31.2442536

>//没有name的node

<node

<id 301639413

<lon 121.4851416

<lat 31.2036493

<tag

<name 西藏南路

>//有name的node

<way

<id 275420538

<nd 2800478890

<nd 2800478891

<tag bridge yes

<tag highway secondary

<tag int_name Zhangheng Lu

<tag layer 1

<tag name 张衡路

<tag name:en Zhangheng Road

<tag name:zh 张衡路

<tag name:zh-hant 張衡路

<tag name:zh_pinyin Zhānghéng Lù

<tag oneway yes

>//way

处理完原始文件之后，prepare类的构造函数会读入（已处理完毕的）data文件，首先读入位于文件第一行的bound信息，bound即该文件包含的区域范围，而后对于每一个node，记录下其id、经纬度，如果有name信息则一并存储。在处理经纬度时用一个strdouble()函数将经纬度转化为double类型。

在读取data文件时，有一些需要注意的点：文件的结构是先bound、后node、再way，因而无需在每次读到一段新信息时都判断此时的类型。对于每一个way，提取其中每一个node的id，由于xml文件本身的问题，一些出现在way中的点实际上并不存在于前述的所有node记录中，遇到这样的点应将其从way的点列中剔除，避免出现“断路”的情况（所谓断路，即读到该点，却无法获知该点的经纬度，导致绘图时出现错误）。

本程序的亮点之一即在于地图的分层存放，即将不同级别的路分置于不同存储结构（实现时每一层的具体存储结构为Vector<way>）中存放，如将道路根据highway的tag信息，分为motorway、trunk、primary、secondary等等，另有natural、waterway、railway等的标记，也应做类似处理。这样在地图缩放时，可以通过规定哪些结构仅在哪些缩放级别上显示，避免杂乱的绘制。如此，即提高了程序的执行效率，又提高了用户的使用体验。

当way全部读入完成后，一旦遇到relation标签，即跳出循环，prepare过程结束。

一些情况下（比如，当搜索最短路时，若起点或终点不在连通的路中，则需要找到邻近的在路中的点），为了便于查找某些小区域内的点而避免全图查找，将整张地图进行了分块处理。具体一些说，即是将地图分成了很多个0.0001 x 0.0001大小的小方格形区域，将每个区域对应的点的信息存储在HashMap<String, Vector<String>>中，前一个String的格式为“X坐标值 && Y坐标值”，其中X、Y坐标值都是该小格左下角的经纬度值（保留

小数点后四位小数），对应的Vector，则是存放了该区域内所有顶点的ID信息。为了节约存储空间，每次读到该区域中第一个点时才生成Vector，之后再遇到同一区域的点，则可直接插入Vector中。

存储点与点间的直接连接关系时，需要忽略的有以下两类：首尾点id相同的封闭路径（这样的路径一般代表的是建筑物或公园、空地等的轮廓）、没有highway信息的路径（一般是水路、铁路，甚至有时有可能只是电线）。

另外，有namedNode专门用于存储所有有name属性的点或路径。实际上其中存储的皆为顶点，若是需要存储有name属性的路径，则将该路径的第一个点作为该路径的代表顶点进行存储。

```
//prepare.java
java.io.File file = new java.io.File("data.osm");
Scanner scan = new Scanner(file);
HashMap<String, node> id_node = new HashMap<>();//记录从Node的id到node的关系
String inLine, id, name, k, v, type, role, dfs;
DecimalFormat df = new DecimalFormat("#.0000");//用于将经纬度转化为0.0001精确度的string
node n;
Vector<String> s;
Vector<way> w = new Vector<>();//储存所有way
Vector<way> motw = new Vector<>();//储存motorway等级的way
Vector<way> truw = new Vector<>();//储存trunk等级的way
Vector<way> priw = new Vector<>();//储存primary等级的way
Vector<way> secw = new Vector<>();//储存secondary等级的way
Vector<way> terw = new Vector<>();//储存tertiary等级的way
HashMap<String, String> namedNode = new HashMap<>();//记录从node或way的name到该node或way的id的关系
```

HashMap<String, Vector<String>> inWayNode = new HashMap<>();//将DecimalFormat处理过的点的经纬度用“&&”连接后的string作为key，将在这个区域内的在way上的node放在Vector<String>里作为value

int i,j;

double x,y;

static double minlat, maxlat, minlon, maxlon;//记录该XML文件的范围

public double strDouble(String ss){}//将String转化为double

public prepare() throws Exception{}//构造函数，处理文件

public Vector<way> getw(){}//返回w

public Vector<way> getMotw(){}//返回motw

public Vector<way> getTruw(){}//返回truw

public Vector<way> getPriw(){}//返回priw

public Vector<way> getSecw(){}//返回secw

public Vector<way> getTerw(){}//返回terw

public HashMap<String, String> getNamedNode(){}//返回namedNode

public HashMap<String, Vector<String>> getInWayNode(){}//返回inWayNode

public HashMap<String, node> getIdNode(){}//返回id_node

地图的绘制

主窗口打开时，绘制的是根据bound信息得到的地图的中心点周边的区域，长为0.04， 宽为0.02。地图的重绘使用drawMap函数，绘制时需要的信息有：中心点的X、Y坐标，以及显示的范围大小。

绘制地图时，分层存储的优势得以充分发挥。每一种缩放尺度下，仅有适合于该尺寸的道路级别进行显示，且无需进行额外的遍历、计算等。类似于百度、腾讯等多家在线地图，缩放的尺度并不是随意的，而是分为了具体几个层级。

尺度（长）小于0.08时，绘制所有层级的道路；大于0.08时，只绘制secondary以上的路径；再其后则只绘制trunk以上的路，再则只绘制motorway……以此类推。每一条路的绘制具体则通过drawLine函数实现，且在不同层级、不同缩放尺度上，道路宽度、颜色等信息均会发生变化。

其他的一些具体点：如为waterway且为闭区域，则用蓝色填充多边形；如为waterway但不是闭区域，则用蓝色绘制线条；natural的Tag下若有water同样填为蓝色；不是water的natural轮廓内填充绿色；subway使用紫色虚线绘制；非subway的railway统一使用灰黑色虚线绘制；标记为building的封闭轮廓填充灰色。

没有路名标记的地图总归是不完整的。但是，若是由于分层问题导致明明对应的路没有在当前尺度下进行绘制，却仍在对应位置给出了名称，则地图的使用者将造成困惑。

因此，地图的名称标记也依赖于当前的缩放尺度，只标记当前已经绘制的路中较高的几个层级。标记路径名称时，名称标记在这条路的某一点上，若该点已经有其他道路标记，则选取该路径的下一点进行标记。当前显示区域内，相同的路名只出现一次。

程序界面

本程序的UI使用JavaFX制作，选择JavaFX的主要原因是因其作为Java本身的一部分存在于运行时环境中，而无需引入第三方依赖，并且JavaFX十分美观，可制作较为现代化的界面。但相比于AWT及Swing，JavaFX本身作为较为年轻的库，自身仍有一些不足，如内存管理机制仍有问题等。

在主程序启动时，首先从prepare过程中接过预先准备好的一系列Vector及Map，存储于自身对象内。

本程序的界面设计十分美观，由于JavaFX的先进性，界面一改以往Swing界面丑陋的模样，十分现代化。且参照最新的Google Map的形式，使得所有按钮及文本框都“飘浮”于地图绘制层之上。有查找地点及查找路线两个主要功能，选择不同功能时，屏幕上侧将只显示对应于该功能的控制界面，十分简洁且直观。

具体功能的实现

本程序通过地点的名称找到对应的在地图上的位置，既可用于单个点的定位，又可用于找最短路时起点和终点的确定。匹配名称时，目前的设定是需要程序中存储的点和用户请求的某个点名称精确匹配。在精确匹配的前提下，由于在预处理阶段已经将有名称的点（包括有名称的路，取其中的第一个点作为对应点）与其ID存入了一个HashMap中，因此只需将所请求的地名作为key输入，即可得到对应的ID作为返回值。在得到返回值后，只需在存储Node的结构中同样通过Hash找到对应顶点，即可得经纬度信息。

找单个点时，一旦可确定经纬度，即以该点为显示区域的中心，重新绘图。若未找到对应点，则不作处理。

在找最短路时，用户需要输入起点与终点的名称，根据上述找单个点的方法，可找到起点与终点的位置。若该点在图中的度数为零（即没有连在大的道路网络中，需要注意的是在预处理时已经排除了封闭路径、水路等的干扰），则查找邻近点中位于道路网络中的点。

预处理时已经记录了每个点到其他点的有效直连边，因此判断点的度数是十分便捷的。另外，由于对整张地图已经进行了每一小格的分格化处理，每一小格是一个仅有0.0001 x 0.0001大小的区域，因此在该区域中如果找到度数不为0的点，是十分理想的。若未找到符合要求的点，则在与该小格相邻的外围格中继续寻找，一层层向外拓展，直至找到为止。在实际实验中，任何点都能在拓展两次之内找到位于路中的对应点。

地图的xml数据文件中，并未给出任何点与点之间的距离信息，因而需要时应通过两点的经纬度关系及勾股定理进行计算。然而计算距离时所需的开方运算效率是十分低下的，因而需要尽可能减少其使用的次数。为此，

任何点与点之间的直连边（edge类型的对象），其初始化时的距离都为-1，-1代表尚未进行计算。一旦需要计算该两点间距离时，便在计算后将该距离储存（替代-1），则之后一旦读到该距离的值不为-1，便无需重复计算，节约时间。

确定可行的起点与终点后，最短路的实际算法类似于Dijkstra，然而结合地图的具体情况进行了一些优化，即是将所有可能被搜索到的点的范围框定在包含起点到终点的一个矩形区域之中，而不是通常意义上的整张地图，有效地减少了搜索的范围及可能涉及到的点的个数，大大提高了效率。

具体地矩形范围框定策略如下：连接起点与终点，以此为对角线可确定一矩形，出于健壮性考虑，将矩形的边界左右各扩展0.2倍的长、上下各扩展0.2倍的宽。同时，考虑到一些小范围找路时的不确定性，再将上下左右的范围各自扩展0.01的经纬度，这样的一个扩展范围对于较大的区域而言无足轻重，但在小范围找最短路时，对于健壮性是很有帮助的。

//mapDraw.java

prepare pre;//

String wname, pname1, pname2, start, end;//pname1是搜索地点或搜索路线的起点的名字，pname2是搜索路线的终点的名字，start是搜索路线时离起点最近的点的id，end是搜索路线时离终点最近的点的id

HashMap<String, node> id_node;//从prepare类中接过id_node的对象

Vector<way> w;//从prepare类中接过w的对象

Vector<way> vw;//记录本次画地图时需要遍历的way的集合

Vector<way> motw;//从prepare类中接过motw的对象

Vector<way> truw;//从prepare类中接过truw的对象

Vector<way> priw;//从prepare类中接过priw的对象

Vector<way> secw;//从prepare类中接过secw的对象

Vector<way> terw;//从prepare类中接过terw的对象

Vector<String> names = new Vector<>();

Vector<String> nodes = new Vector<>();

```
HashMap<String, String> namedNode;//从prepare类中接过namedNode的对象
HashMap<String, Vector<String>> inWayNode;//从prepare类中接过inWayNode的对象
HashMap<String, String> Pre;
HashMap<String, Double> dist;
public class Edge{
    public String id;
    public double w;
    Edge(String ID){
        id = ID;
        w = -1;
    }
};//记录边的另一个端点和边的长度的class，开始时w为-1，在找路过程中如果用到就计算w，计算之后无需反复计算
Vector<Edge> e;
HashMap<String, Vector<Edge>> edge = new HashMap<>();//key是node的id，value是所有与这个node有边的点及边长的集合
Vector<String> vs;
Pane pane3;
BorderPane pane;
Path path, spath;
Text text;
GridPane pane1;
GridPane pane4;
FlowPane pane2;
BorderPane pane51;
BorderPane pane5;
FlowPane pane52;
Circle circle;
TextField tf1, tf2, tf3;
Button l = new Button("<");
Button r = new Button(">");
Button u = new Button("^");
Button d = new Button("v");
```

Button b = new Button("+");

Button s = new Button("-");

Button fp = new Button("查找地点");

Button fw = new Button("查找路线");

Button fp1 = new Button("搜索地点");

Button fw1 = new Button("搜索路线");

Label lbal1 = new Label(" ");

Label lbal2 = new Label(" → ");

Group g = new Group();

Group gc = new Group();//以上全为地图的布局所需变量

DecimalFormat df = new DecimalFormat("#.0000");

double xx, yy, xnow, ynow, mnow, nnow;//xnow、ynow、mnow、nnow分别是当前界面的中心的（x, y）以及中心到边界的长和宽

boolean bool;

public void drawline(way way1, String nn, double x, double y, double m, double n){}//画出way1并根据地图缩放大小决定是否标出路名

public void drawmap(double x, double y, double m, double n){}//画出以（x, y）为中心以及m、n为中心到边界的长和宽的地图

public void findPlace(String str){}//找到名字是str的点，另xnow为其经度，ynow为其纬度

public String findNP(double x, double y){}//找到（x, y）附近的点

public void Dijkstra(double bx, double sx, double by, double sy){}//最短路算法，只需要考虑以起点到终点的连线为对角线的矩形并以其中心向四周扩充0.01的矩形范围内的路，超出范围就丢掉该路

public void findWay(){}//找路函数

public void getEdge(){}//得到HashMap<String, Vector<Edge>> edge

public void start(Stage primaryStage) throws Exception{}//画出地图

class lHandler implements EventHandler<ActionEvent>{}//左移，向左移动界面1/3

class uHandler implements EventHandler<ActionEvent>{}//右移，向右移动界面1/3

class rHandler implements EventHandler<ActionEvent>{}//上移，向上移动界面1/2

class dHandler implements EventHandler<ActionEvent>{}//下移，向下移动界面1/2

class bHandler implements EventHandler<ActionEvent>{}//放大，将地图的长宽改为原来的1/2

class sHandler implements EventHandler<ActionEvent>{}//缩小，将地图的长宽改为原来的2倍

class fwHandler implements EventHandler<ActionEvent>{}//上方按钮显示查找地点

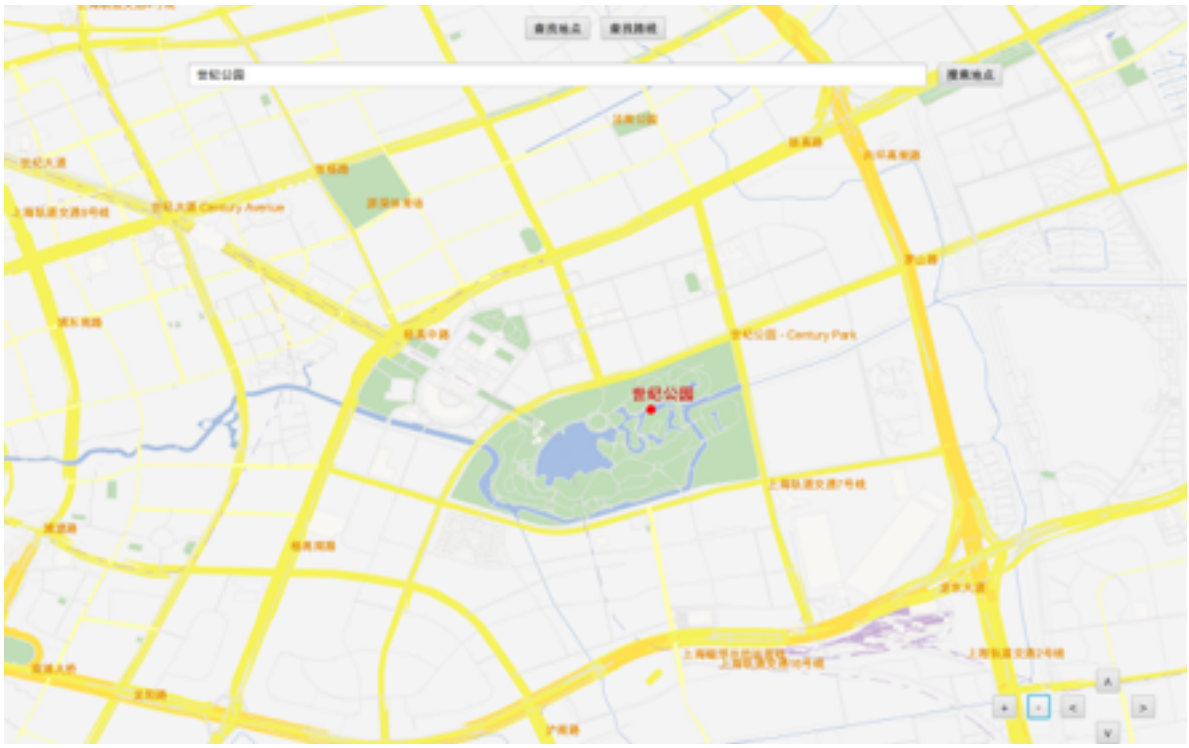
class fpHandler implements EventHandler<ActionEvent>{}//上方定点显示查找路线

class fp1Handler implements EventHandler<ActionEvent>{}//搜索地点，以该地点为中心画新地图

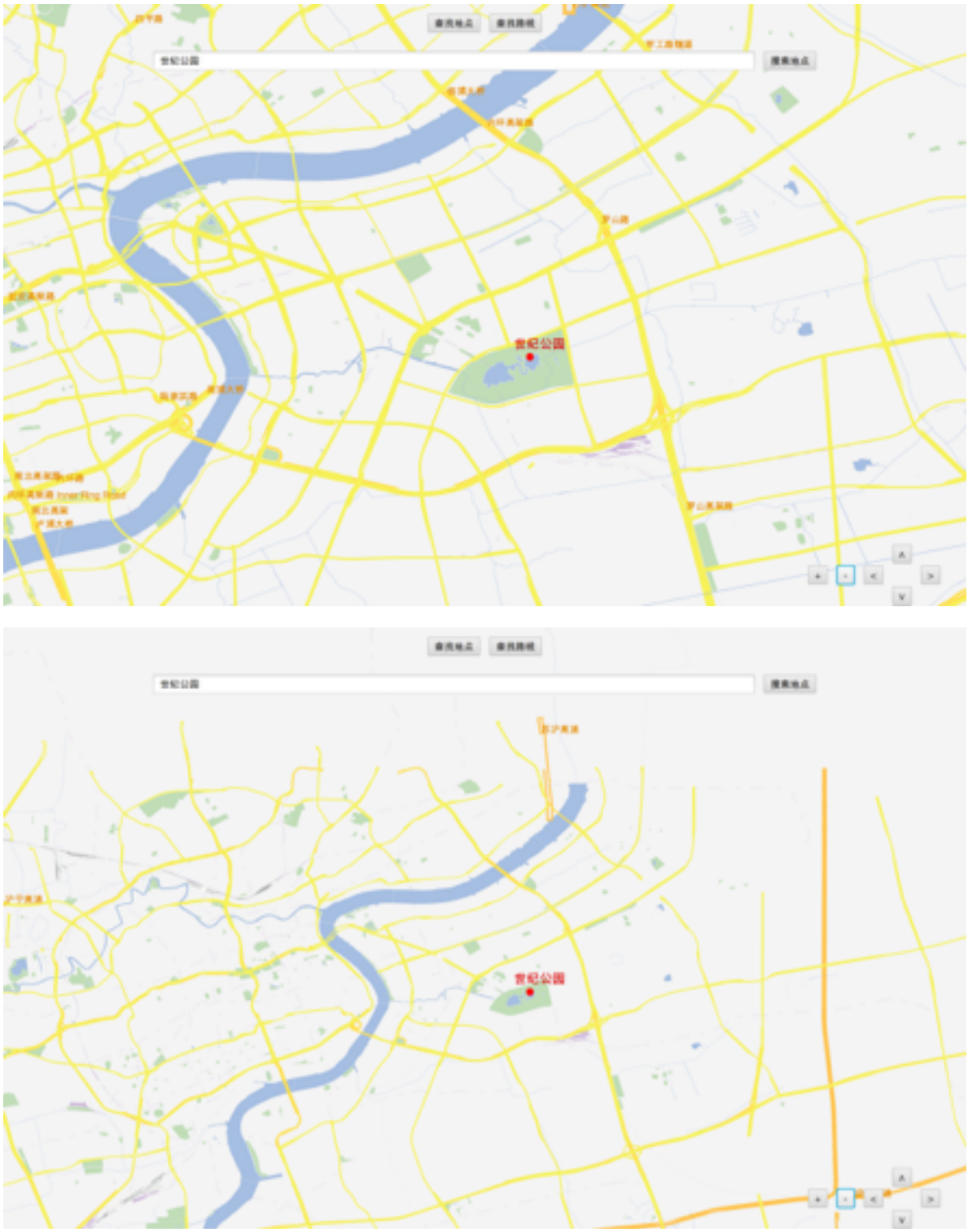
class fw1Handler implements EventHandler<ActionEvent>{}//搜索路线

下面是地图界面的截图：

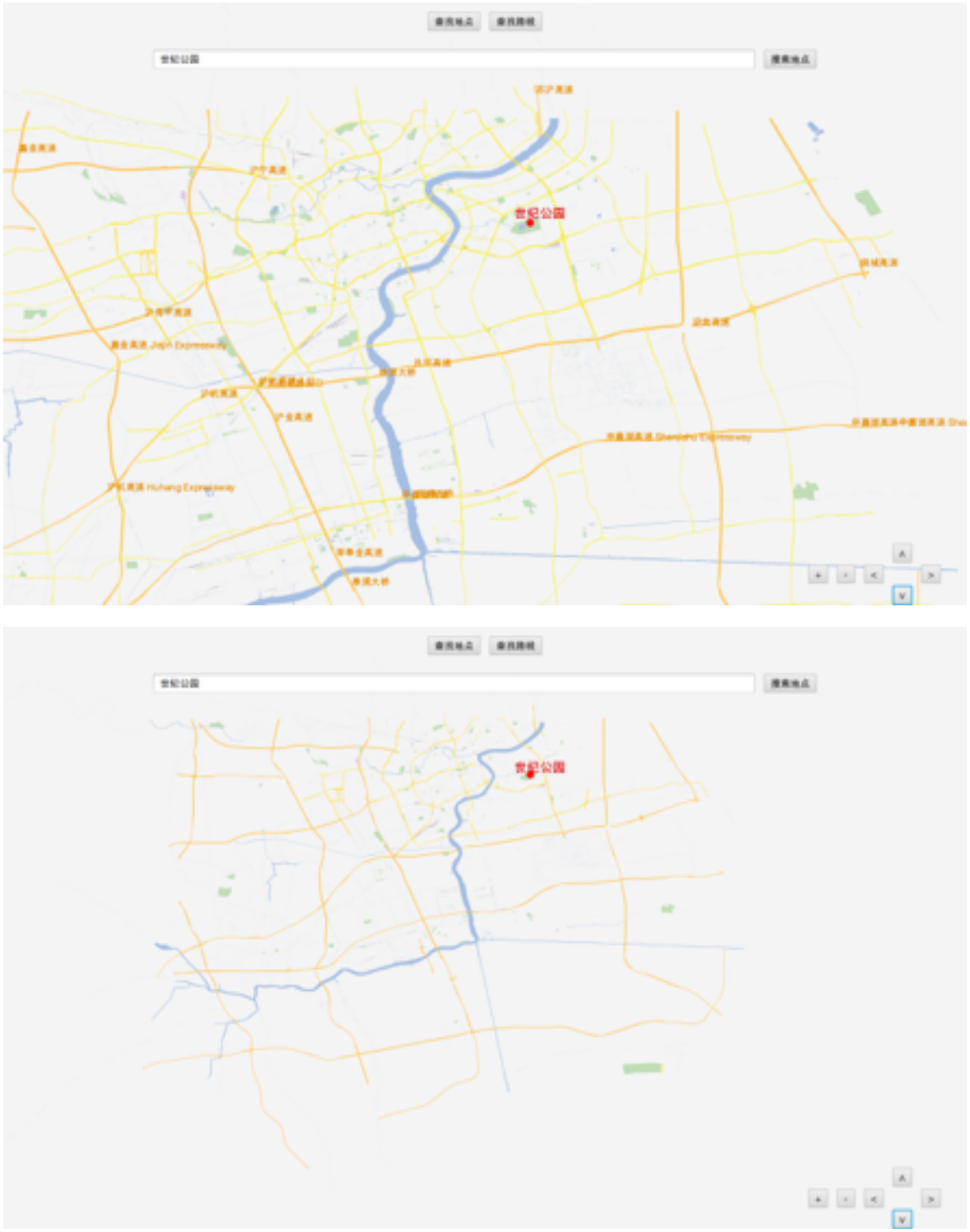
查找地点



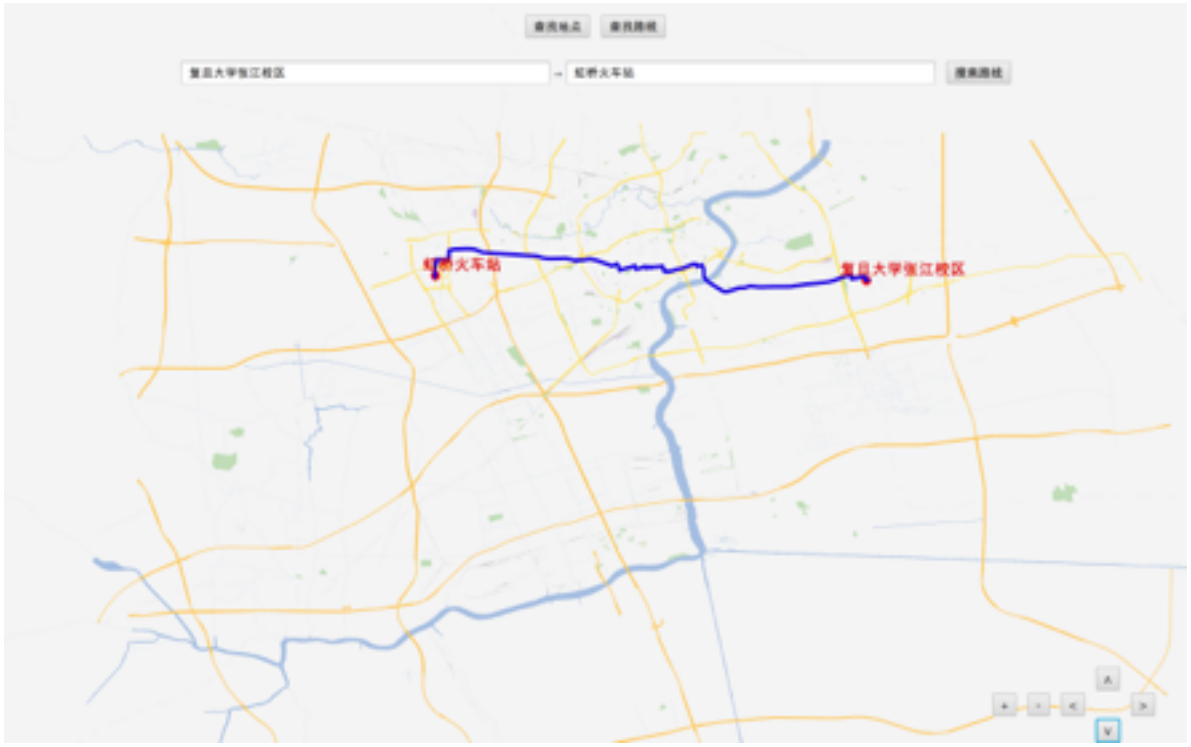
地图缩放与分级显示

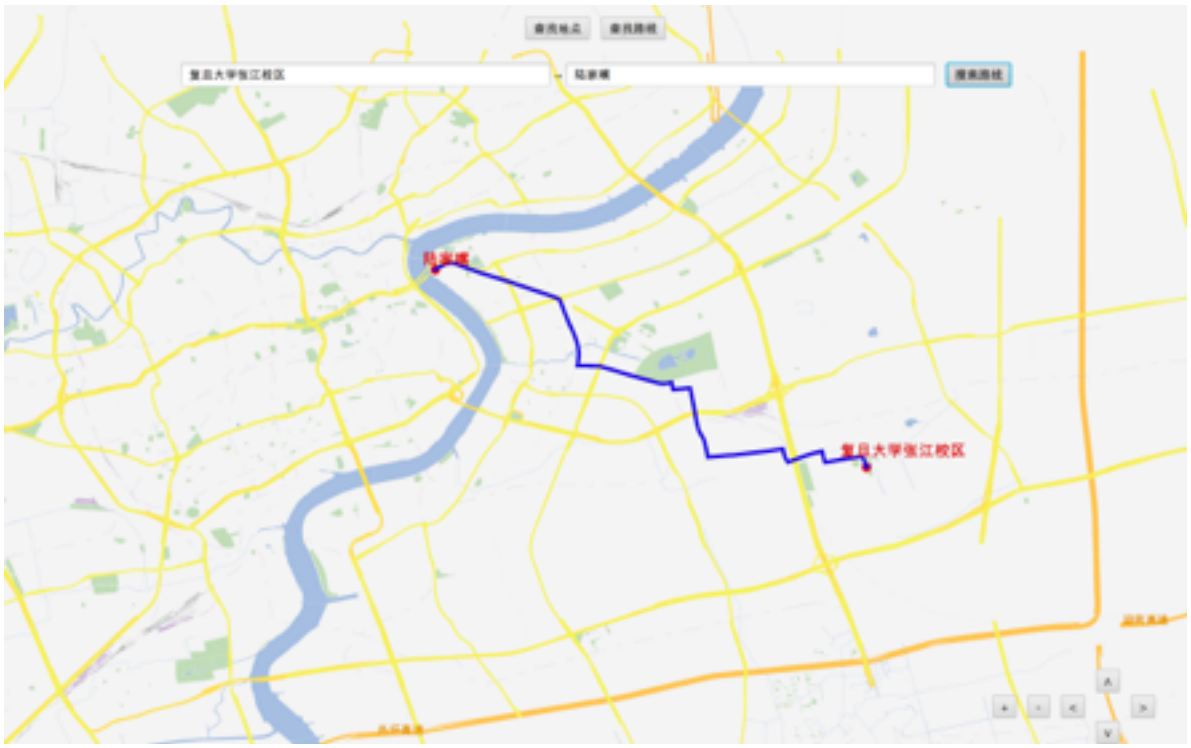


地图缩放与分级显示

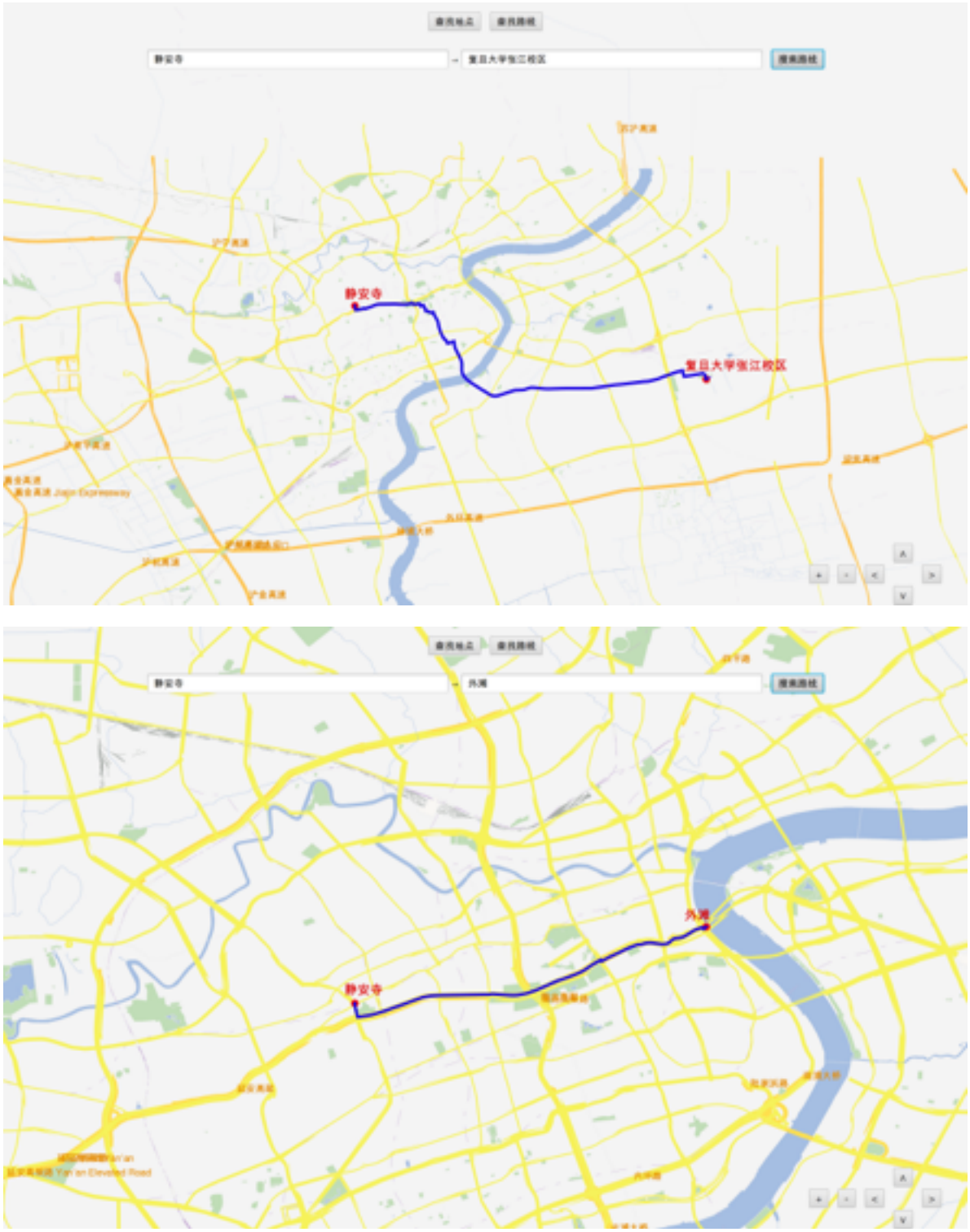


地图缩放与分级显示


查找地点
查找路线
复旦大学张江校区
→
虹桥火车站
搜索路线
虹桥火车站
复旦大学张江校区


查找地点
查找路线
复旦大学张江校区
→
陆家嘴
搜索路线
陆家嘴
复旦大学张江校区

查找路线



查找路线