

《数据结构》

课程设计报告

实验名称： \_\_\_出租车轨迹分析\_\_\_

专业班级： 计算机科学与技术1班

学号： 202130442501

姓名： 祝锦红

得分：

日期： 2023.5.12

|  |  |
| --- | --- |
| 教  师  评  语 | 教师签名：  日期： |
| 成  绩  评  定 |  |
| 备  注 |  |

**一、需求分析**

根据《数据结构大作业任务书》（下称《任务书》）中“8.出租车轨迹分析”的数据集和要求，将任务分为偏向前端的“可视化”部分和偏向后端的“功能实现”部分，另外需要设计程序衔接二者。

“可视化”部分，利用高德地图的JavaScript API，编写HTML、CSS语句和JavaScript代码，重点实现接收用户需求，并将结果呈现在电子地图（网页）上，包括分类，打印点集、矩形、文字等。

“功能实现”部分，考虑C++实现。主要关注《任务书》中F3—F9部分，利用离散化、序列化等技术与标准库，依托哈希表、红黑树等数据结构，构建文件与内存的关系，在预处理数据后，建立函数逐个实现所需功能。

为了更好地衔接，在各自的程序中额外添加了关于文件读写、参数接收等内容。

**二、详细设计**

**1. 可视化**

高德地图 JS API 是一套 JavaScript 语言开发的的地图应用编程接口，移动端、PC端一体化设计，一套 API 兼容众多系统平台。在这里我们利用高德地图JS API，在JS端进行可视化。

在注册账号并申请Key后，可在本地和服务器应用高德地图API。

**（1）F1：出租车轨迹可视化**

首先在电子地图一侧放置功能菜单，包含按钮和输入框，当用户点击“显示所有车辆坐标”时（点击事件），将读取本地文件中每辆车的每个坐标；点击“显示该车坐标”时，将读取输入框中的数字（限制为整数），读取本地文件中特定车辆id的文件。



图1.1.1 F1功能栏设置

读取后，利用API提供的点标记功能Amap.Marker，自定义点样式，在地图上标记各轨迹点（setPosition），并标记出出租车号（setLabel）。



图1.1.2 F1功能实现

此处可以有个效果图

**（2）F2：地图缩放功能**

原始的高德地图API在电子地图导入时地图缩放功能默认开启，在此进行保留以完成F2.

**2. 功能实现**

**（1）类设计（taxi\_header\_use.h）**暂时写个概要，应该还要写具体的类成员函数如何实现

数据处理是一大重点和难点，因此我们设计了多个类，以不同的形式处理原始信息，并把它们存储起来。

针对不同的需求，我们设计了：

**Point类：**出租车的轨迹点类，包括时空信息和车名，能直观地将原始信息载入内存。包括出租车序号、时间、经纬度。

**Taxi类：**将一辆出租车的Point聚合在一个类中。包括出租车序号、Point数量、指向（有效）Point数组的指针、出租车在数组中的位置。

**Time\_distribution类：**即**时间分布类**。表示某段时间某范围中地图上出租车和轨迹点情况。包括一个哈希表：出租车序号—出租车在该时段的所有Point、时间、被查询的出租车号、被查询的时间值、矩形左上角和右下角经纬度、多个类对象出现出租车总数量信息、标记出租车是否被统计过的数组（避免重复统计）。查询某个范围的出租车轨迹信息用红黑树存储。

**Position\_distribution\_inBeiJing类：**即北**京内的空间分布类**。表示以0.01\*0.01的经纬度变化划为一个类，能够获取空间范围内某部（或全部）出租车的全部信息，可以按时间段查询。包括一个哈希表：出租车序号—该空间范围内全部点信息、该类对象统计的空间起始经纬度、被查询的出租车号、被查询的时间（段长自定义）、矩形区域经纬度、被统计的出租车总量、标记出租车是否被统计过的数组。查询某个范围的某时间出租车轨迹信息用红黑树存储。

**Position\_distribution\_outBeiJing类：**即**北京外的空间分布类**。作为一个整体，用一个哈希表存储信息：出租车序号—出租车位于北京外的所有点信息。

针对r区域分块的功能，我们设计了：

**Position\_distribution\_r类：**用于将r\*r的格子的各段时间的信息存储到一起，1个单位的r等效于0.001经纬度。这个类包含了：静态变量r、记录每个格子出现了哪些出租车的二维容器、记录每个格子出现的轨迹点的容器（vector套set）。

**Position\_distribution\_outr类:**将北京外的区域作为整体进行考虑，包含了：记录出租车是否出现过（bool数组）、每时段出现的出租车次数，以及北京外每个时段出现的全部出租车轨迹信息（vector套set）。

针对路径分析，我们设计了：

**pathNode类：**即**路径节点类**，用于存储对应路径的大致被划分到空间的信息。只包含两个信息：经度和纬度，精确到0.001.作为以下两个类的基础类。

**pathEdge类：**即**路径边类**，用于存储相邻路径的空间信息。包含以下信息：边的两个节点（无向）、经过该边所花费的最短时间（当存在多次经过时）、边的频繁度，表示该边被经过的次数、边的物理长度、达到最短时间所用的起始点和终点（有向）。

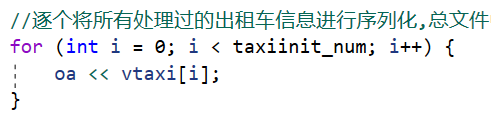
**pathGraph类：**用于存储所有节点和边，构成一个图。包含两个哈希表，其中一个用于存储全部出租车的结点信息：结点位置生成的字符串—结点形成的类信息；另外一个存储边：无向边生成的字符串—边形成的类信息。

**（2）序列化（taxi\_init\_data.cpp）**

建立了类后，我们可以将原始数据转换为各种各样的数据，为了更好、更方便地存储和利用这些数据，我们为每个类实现了序列化和反序列化的函数。

序列化和反序列化基于C++boost库。

**Taxi类序列化：**



序列化的核心代码如上，中间的过程主要在于获取原始数据、按逗号分割数据并将数据写入taxi类的vtaxi数组中。最后将数据输出到boost::archive的序列化流中，得到序列化的本地文件。

**Taxi类反序列化：**

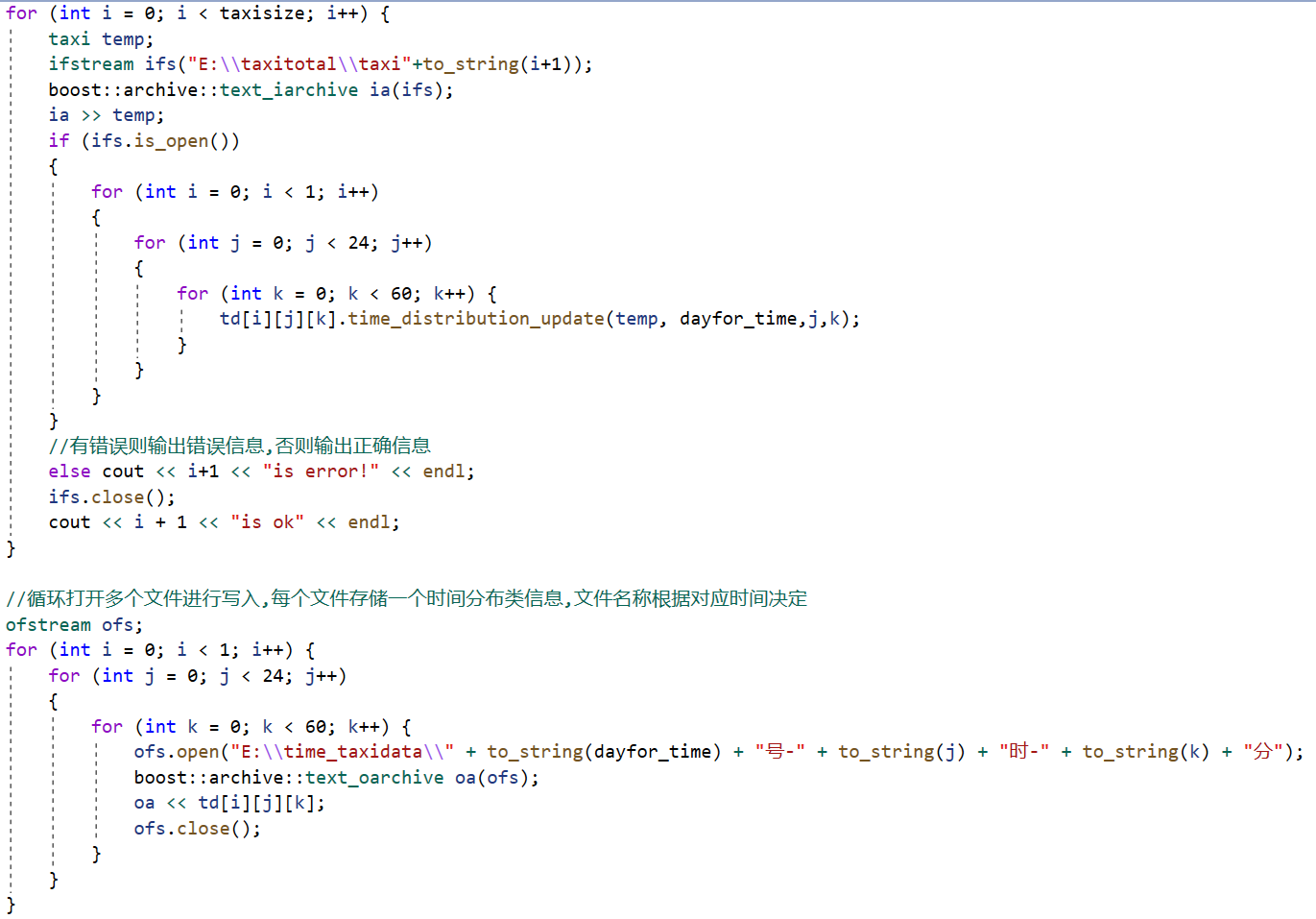
针对一辆出租车，要对其反序列化，可以采用类似的方法，将ofstream改为ifstream即可。



**Time\_distribution类序列化：**

该类的序列化需要出租车类序列化和反序列化支持。

假设我们已经将出租车信息写入了本地序列化文件，那么可以将其反序列化出来，然后按照time\_distribution的格式将数据写入到time\_distribution的类对象数组中，最后以同样的方式（ofstream和boost::archive::text\_oarchive）将对象信息序列化到本地文件中。



**Time\_distribution反序列化：**

传入时间参数（时、分）和出租车号后，用类似的方式（ofstream改ifstream）反序列化time\_distribution文件到time\_distribution对象。

其他类均实现了类似的序列化和反序列化操作，在此不多赘述。

**（3）功能函数设计**

**任务一：F3：区域范围查找**

实现函数：int function\_findtaxinum\_somewhere(set<Point, CMP>& s,int day, int hour, int minute, int time\_span , bool output)

set<Point, CMP>& s：存储区域内点信息，内部按车名和时间排序

double\* pos：4个元素，矩形西经度、东经度、南纬度、北纬度

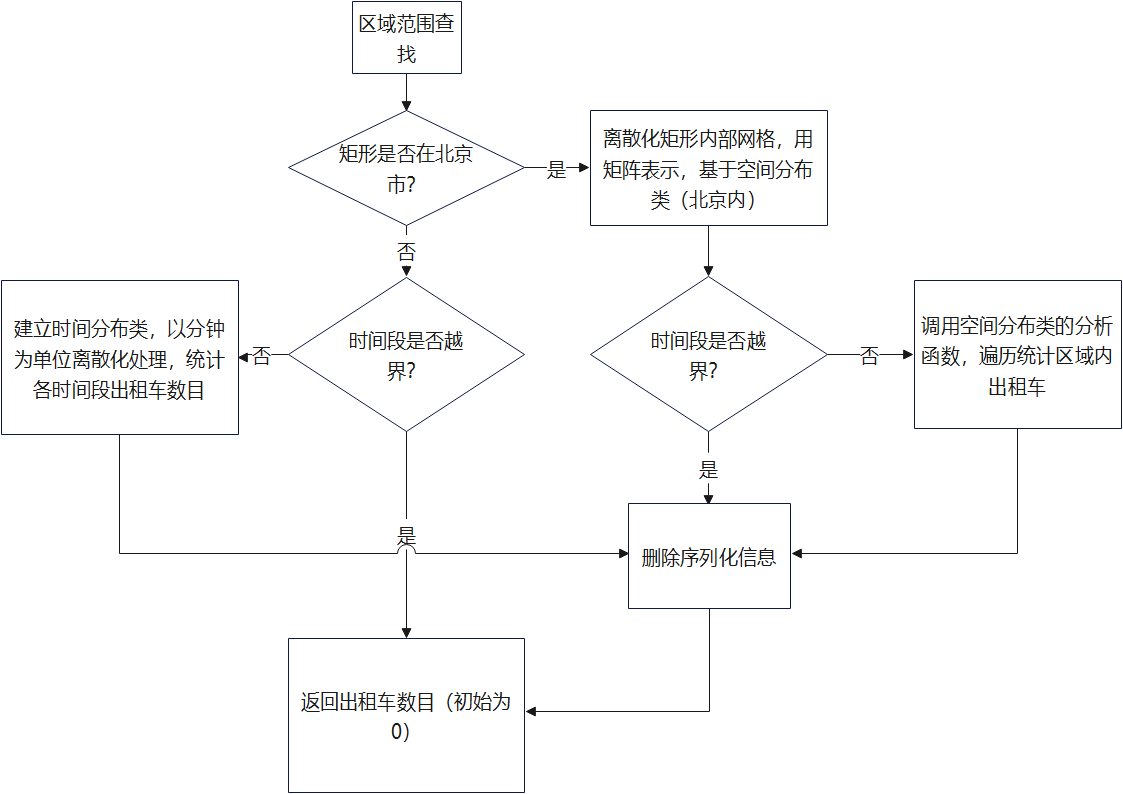
int day, hour, minute：时间

int time\_span：时间跨度（步长）

bool output：是否输出过程结果

返回值：返回矩形区域内出租车数目

具体流程如下：

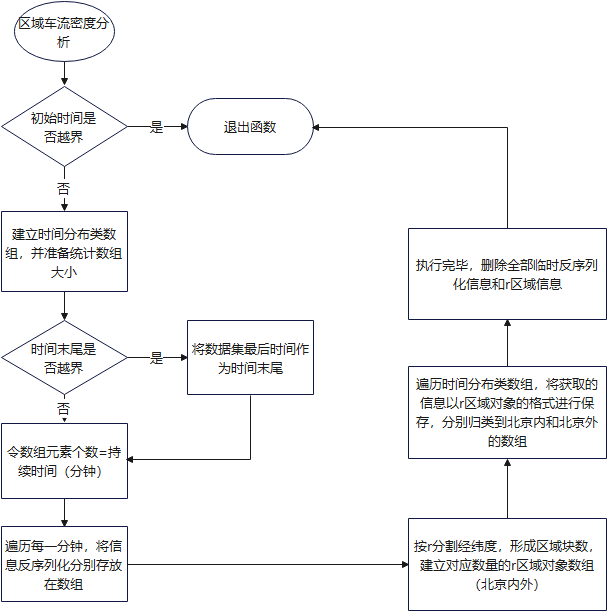


**任务二：F4：区域车流密度分析**

在我们的设计中，一段时间的某个区域（格子）的“车流密度”以出租车数量为标准，车流密度的变化就是在两段时间上这个区域出租车数量的变化情况。

实现函数：void function\_regional\_density\_analysis(int r, int day, int hour, int minute, int time\_span, bool output)

具体流程如下：



运行结果在中间过程（“执行完毕前”）输出到文件中，函数本身不返回结果。

**任务三：F5：区域关联分析1**

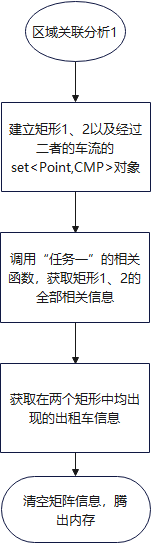
这个任务需要统计不同时间段往来两个矩形区域的车流量的变化。在这里，我们定义：当同一辆车在两个矩形区域均出现时，对往来这两个区域之间的车流量贡献增加。反之，如果有车在其中一个区域出现但另一个区域没出现，或者在两个区域都没出现，就不算车流量。

实现函数：set<Point, CMP> function\_region\_association\_analysis\_one(double\* pos1, double\* pos2, int day, int hour, int minute, int time\_span, bool output)

double\* pos1, pos2 ：两个矩形的方位

返回值：点集，可以反映车流量

流程如下：

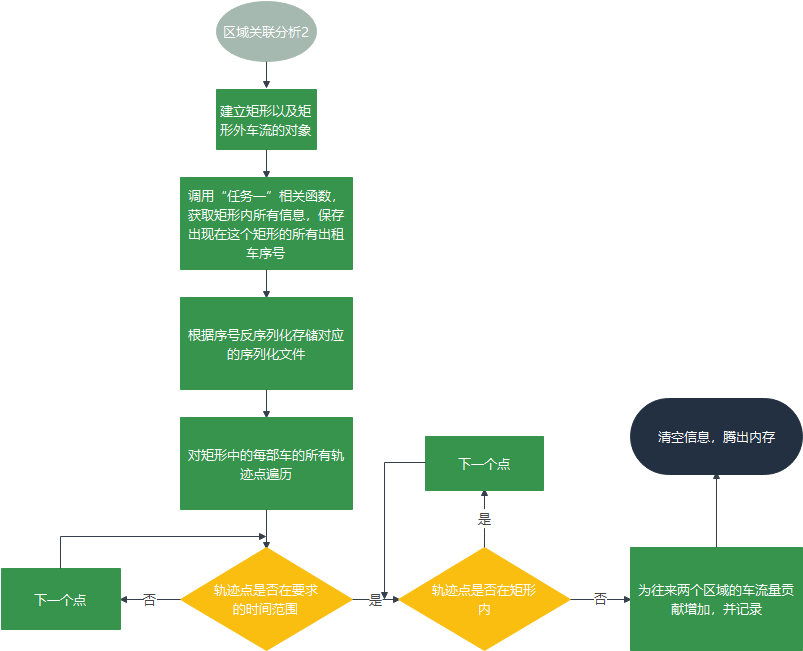


**任务四：F6：区域关联分析2**

这个任务和F5的区别在于：其中一个矩形区域变成了另一个矩形区域的“补集”，即以外的所有地区，因此处理方式相差不大。

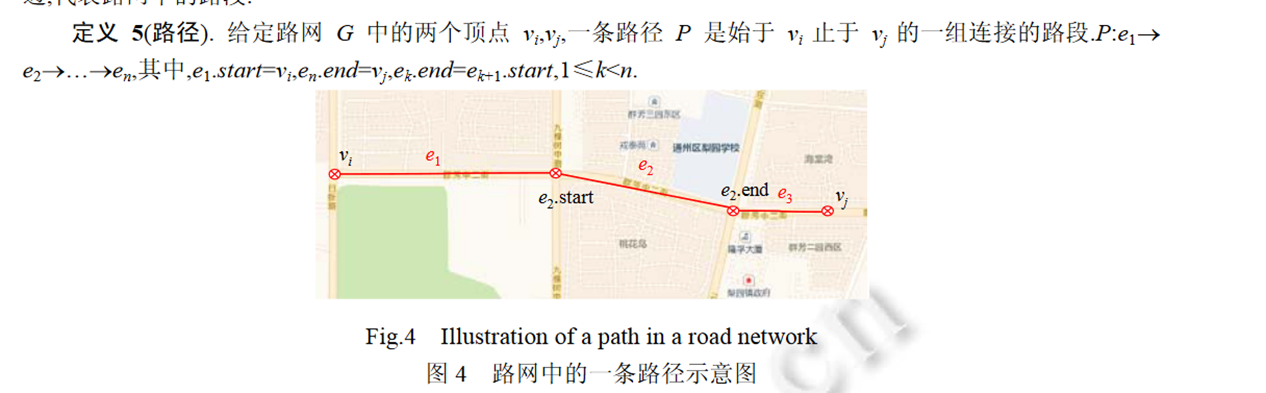
实现函数：void function\_region\_association\_analysis\_two(double\* pos, int day, int hour, int minute, int time\_span, bool output)

在此给出函数流程图：



**任务五：F7：频繁路径分析1**

这个任务的需求描述并不直观，在此我们定义两个量。其一是“路径”，我们采用“路网匹配算法综述”（高文超等）的定义，如下：



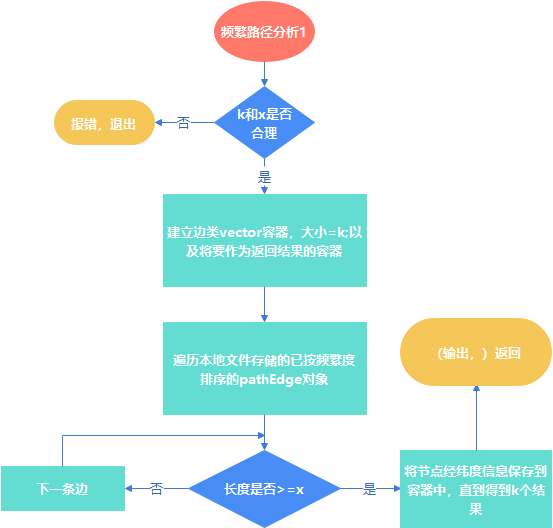
注意路径是有向边两两首尾相连的结果。

其二是“频繁度”，我们定义：一条路径的频繁度是这条路径上通行的汽车的总数。

实现函数：vector<vector<string>> function\_path\_frequent\_analysis1(double x,int k, bool output)

返回值：第一层vector:“边”；第二层vector：四个元素，表示两个节点的经纬度

流程如下：



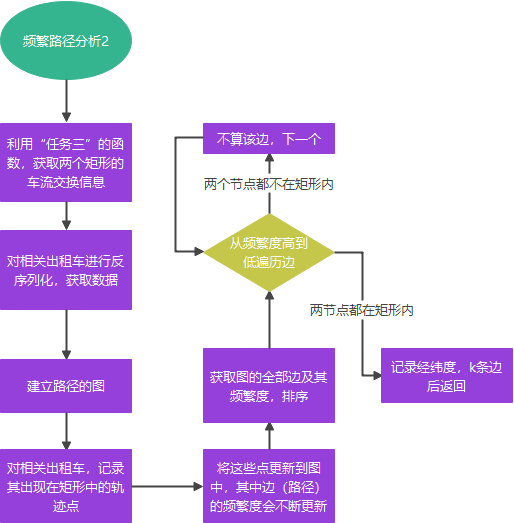
**任务六：F8：频繁路径分析2**

这个任务涉及到两个矩形，也涉及到前面关于频繁路径的实现。因此我们将二者结合来实现。

实现函数：vector<vector<double>> function\_path\_frequent\_analysis2(double\* pos1, double\* pos2, int k, bool output)

返回值：节点经纬度信息数组

过程：



前面的任务中我们对频繁度进行了预处理，这里因为限定了区域，我们不能直接读取频繁度文件，只能通过一步步建图来更新路径的频繁度。

考虑到k条边，实际上可以将排序的步骤改为建立大顶堆来存频繁度，取前k个堆顶元素即可。实际上都需要去获取图的全部边，因此时间复杂度区别不大，为了让程序更“显而易见”，采用排序的方法。

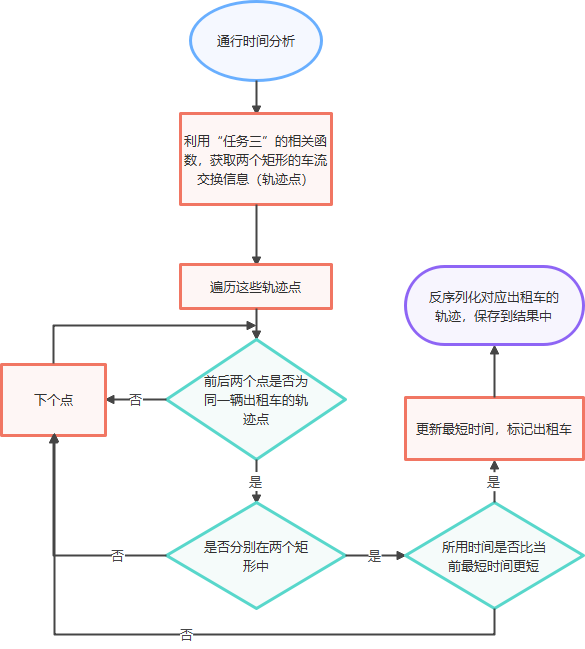
我们还设计了仅仅分析相邻点的路径情况，这时，只需要考虑依次读取的每条在矩形的边（外部存储时已经按频繁度排序），读取k条即可。

**任务七：F9：通信时间分析**

实现函数：vector<Point> function\_pathuse\_shortesttime(double\* pos1, double\* pos2, int day, int hour, int minute, int time\_span, bool output)

返回值：一系列点，依次相连后得到最短通行时间的路径

实现流程如下：



三**、软件测试**

（正文宋体小四）

**四、总结**

（正文宋体小四）

**五、参考文献**

（格式随意）