

大作业 互联网出行

计 54 贾越凯 2015011335

设计框架

本项目核心算法使用 C++ 编写，前端网页使用 JavaScript 调用高德地图 API，后端 Web 服务器使用 Python 的 Django 框架编写，并实现了一个位于中间层的 Python 库，能够调用 C++ 核心算法。

构建与运行

见 README.md。

第三方库

本项目使用的下列第三方库已放入 `externals/` 目录：

- 图的划分所需的 METIS 库：<http://glaros.dtc.umn.edu/gkhome/metis/metis/overview>
- 高效查询两点最短路所需的 GPTree 库：
https://github.com/TsinghuaDatabaseGroup/GTree/tree/master/src/gtree_new_p2p

前端网页

本项目前端主要逻辑使用 JavaScript 编写，地图、乘客、车辆、路线的显示使用了高德地图的 JS API。API 的基本使用方法如下：

- 地图显示：

```
map = new AMap.Map("container", {
    center: new AMap.LngLat(center.lng, center.lat),
    resizeEnable: true,
    zooms: zooms
});
```

- 点标记显示：

```
new AMap.Marker({
    map: map,
    position: [lon, lat],
    offset: new AMap.Pixel(-8, -16),
    icon: new AMap.Icon({
        size: new AMap.Size(16, 32),
        image: "static/img/car.png",
        imageSize: new AMap.Size(16, 32)
    });
});
```

- 路线显示：

```

pathSimplifierIns = new PathSimplifier({
    zIndex: 100,
    map: map,
    getPath: function(pathData, pathIndex) {
        return pathData.path;
    },
    getHoverTitle: function(pathData, pathIndex, pointIndex) {
        return null;
    },
    clickToSelectPath: false,
    autoSetFitView: false,
    renderOptions: {
        pathLineStyle: {
            strokeStyle: "#22aa99",
            lineWidth: 6,
            dirArrowStyle: true
        },
        pathLineHoverStyle: null,
        pathNavigatorStyle: {
            width: 16,
            height: 32,
            strokeStyle: null,
            fillStyle: null,
            pathLinePassedStyle: {
                strokeStyle: "#ffaa00"
            },
        }
    }
});

pathSimplifierIns.setData([
    {
        path: car.path
    }
]);

```

- 路线巡航：

```

navi = pathSimplifierIns.createPathNavigator(0, {
    loop: true,
    speed: 10000,
});

navi.start();

```

前端的 HTML 代码位于 `server/home/template/index.html`，JavaScript 代码和 CSS 等静态文件位于 `server/home/template/static/`。

后端 Web 服务器

后端的 Web 服务器实现分为两部分：Django 服务器框架，以及调用 C++ 核心算法的 Python 库。

Django 服务器

Django 服务器部分的代码位于 `server/` 目录，其中 `server/home/views.py` 实现了 HTTP 请求的解析，`server/home/api.py` 实现了对核心算法的调用。

前端发来的两种 `GET` 请求如下：

- `GET /nearestnode`：传入的参数 `location` 为一个字符串，形如 `"116.419803,40.011865"`，表示用户输入的坐标，会返回离该点最近的路网节点编号，使用 JSON 格式。
- `GET /query`：传入的参数为 `srcId` 和 `dstId`，分别表示待接乘客起点和终点的路网节点编号，返回数据为最优的 5 辆车，以及每辆车的坐标、行驶路线、车上乘客的坐标、离带接乘客的距离和两个绕路距离等信息，使用 JSON 格式。

Python 调用 C++ 函数

本项目实现了一个 Python 库 `itrafficpy`，代码位于 `itrafficpy/` 目录。

其中，`itrafficpy.[cpp|h]` 将对核心算法库(`libitraffic.a`)的调用封装为可供 Python 识别的格式。该文件会被编译为动态链接库 `libitrafficpy.[dll|so|dylib]`。

`__init__.py` 实现了载入该动态链接库，并将对其的调用封装为 Python 函数。`solution.py` 中定义的是 Python 版的一辆车的方案类。

核心算法

本项目的核心算法使用 C++ 实现，位于 `src/` 目录。各文件的作用如下：

- `Node.[cpp|h]`：定义了路网节点类，实现了根据经纬度计算地球上两点的地表最短距离(两点所在的大圆上的弧长)；
- `Car.[cpp|h]`：定义了车辆类，实现了对于一辆车，求将乘客全部送达的最短距离、绕路距离以及送达顺序；
- `Map.[cpp|h]`：定义了地图类，实现了路网数据的读入、直接用地表最短距求计算两点距离、调用 GPTree 使用路网数据求两点距离等函数；
- `Solution.[cpp|h]`：定义了对于一辆车的方案类，方案中包含所选车辆、送达顺序、行驶路线、绕路距离等数据；
- `InternetTraffic.[cpp|h]`：本项目主类，用于加载读入车辆数据、处理查询、返回最优的 5 辆车等；
- `main.cpp`：实现大作业功能的命令行程序，用于测试。

该部分代码会被编译为静态链接库 `libitraffic.a` 以及可执行文件 `main`，可分别被 `itrafficpy` 模块链接或直接在命令行中运行。

算法描述

核心算法的实现主要位于 `src/InternetTraffic.c` 和 `src/Car.c` 中。

绕路的定义

原问题中绕路的定义不是很精确，为了更符合实际，我修改两种绕路距离的定义为：

- 车上乘客绕路：去接待接乘客后将所有乘客送达需要的最短距离，减去不去接待接乘客时将所有乘客送达需要的最短距离，可以认为是司机的绕路距离。
- 待接乘客绕路：待接乘客上车后实际坐车的距离，减去该乘客起点与终点间的最短路。

其中将所有乘客送达需要的最短距离是一个旅行商问题，存在最优解。

最优的车辆的定义

原问题对选出的 5 辆车没有显示，不过实际中显然应该选出最优的 5 辆车，于是就需要有个“最优车辆”的定义。

结合了实际与实现难度，我给出的“最优车辆”的定义如下：该车从初始位置开始，直到将待接乘客送达目的地时，已行驶的距离最短，当车速不变时可认为是让待机乘客到达目的地的时间最早。将所有可行车辆按这个距离从小到大排序，前 5 辆即被选出的车。

算法

题目中要求所有车辆离待接乘客的距离不超过 10km，且上面定义的需要最优化的距离很大程度上与当时车辆和待接乘客的距离有关，所以可先将所有车辆按到待接乘客起点的最短路从小到大排序，只保留最短路小于等于 10km 的车辆，既大大减小了搜索范围，又有助于先找到最优的车辆。由于一次询问中待接乘客起点不变，这是个单源最短路问题，可使用 Dijkstra 算法在 $O(|V| \log_2(|V| + |E|))$ 的复杂度内求得($|V|$ 为路网点数， $|E|$ 为路网边数)。

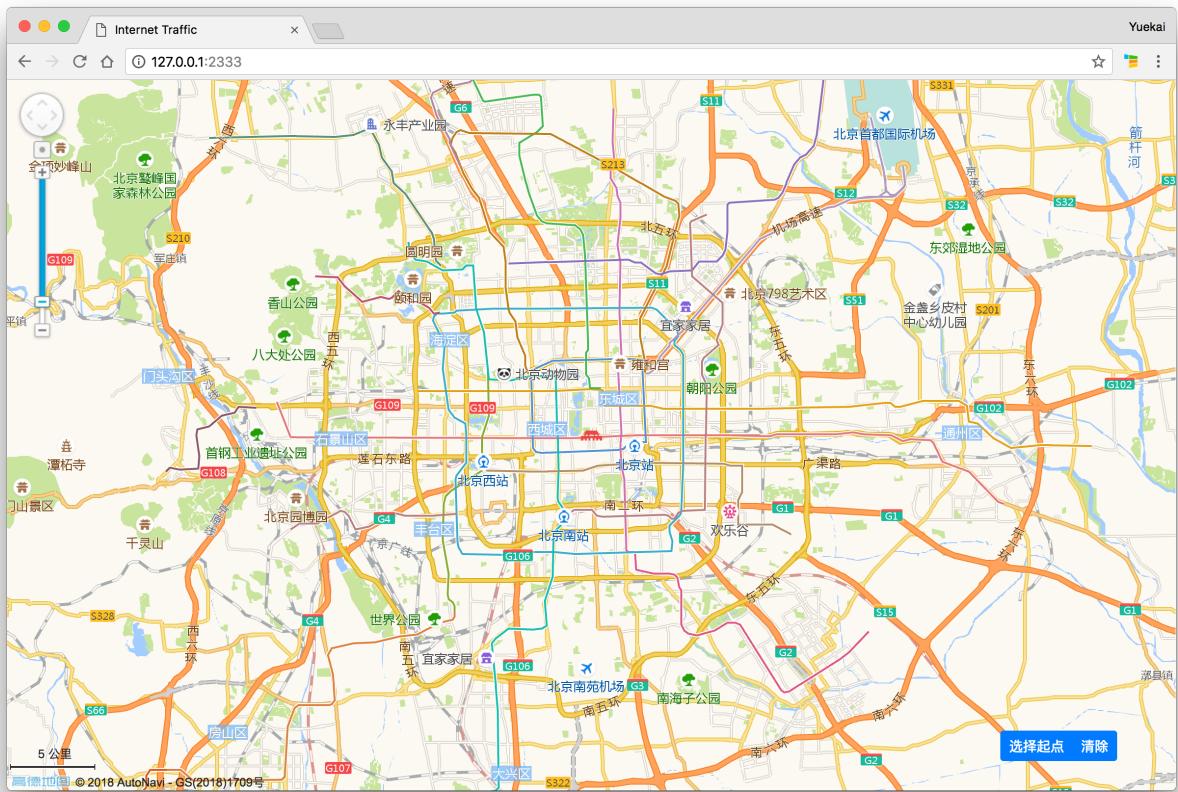
然后对于剩下的每辆车，分别计算去接待接乘客和不去接待接乘客时，将所有乘客送达的最短距离，以便算出绕路距离。这是个经典的旅行商问题，相关的节点有车上乘客终点、待接乘客起点、终点，最多 5 个点，所以即使用暴力枚举经过点的顺序，也需要 $5! = 120$ 的复杂度。为了高效计算两点间的最短路，我使用了第三方库 GPTree。实际上可先预处理出 6 个点(算上车辆的当前位置)两两之间的最短路，只需计算最多 $6 \times 5 / 2 = 15$ 次两点间的最短路即可。为了更加符合实际，我还枚举了乘客起点的顺序，即车辆可以选择不直接去接待接乘客，而先在路上放下一位乘客，以达到最优。

现在已经求出了所有满足离待接乘客的距离不超过 10km、两个绕路距离不超过 10km 的车辆，将它们按最优车辆的定义排序，前 5 辆即被选出的车。这时再使用 GPTree 恢复出最短路径的方案。

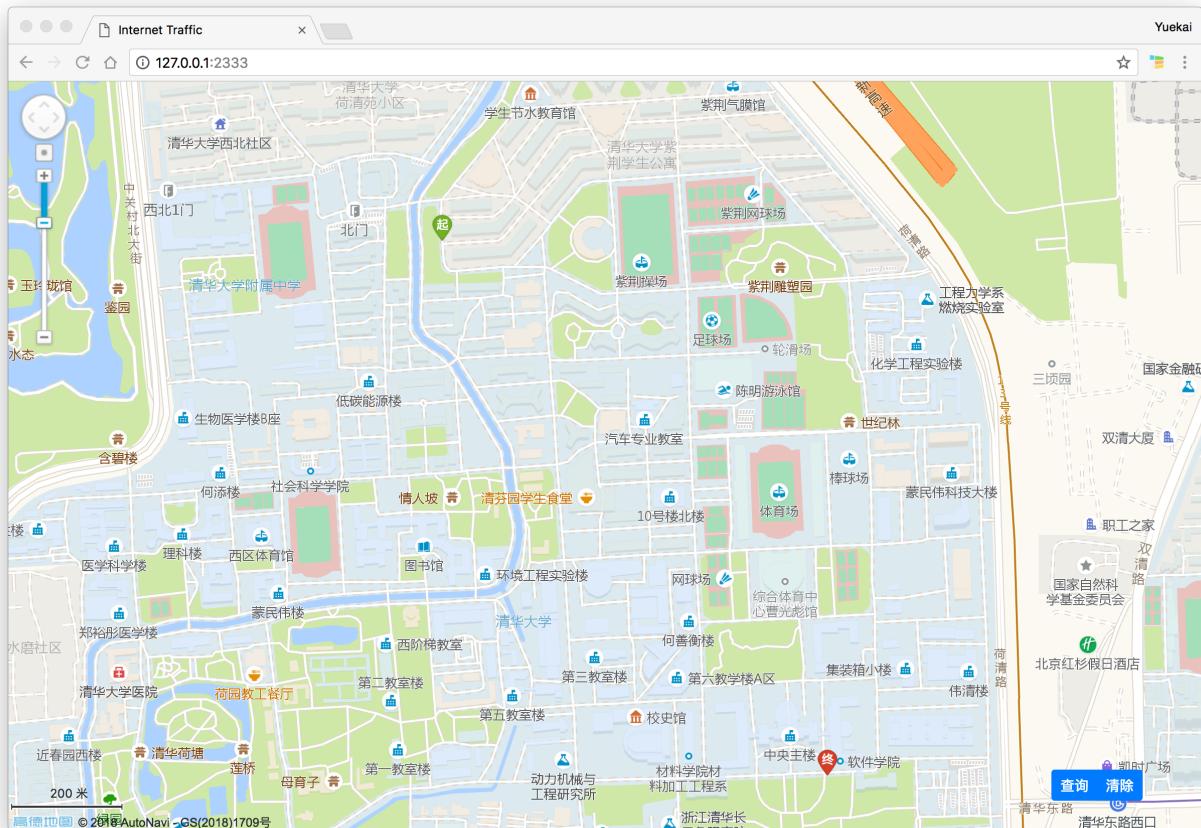
在极端情况下所有符合要求的车辆数仍然可能会很多，计算量庞大。不过在将所有车先按到待接乘客的距离排序后，排在后面的车辆不太可能成为最优车辆。所以可规定一个阈值或时限，当符合要求的车辆数达到该阈值或超过时限直接返回，这时一般也能获得最优的 5 辆车。

效果演示

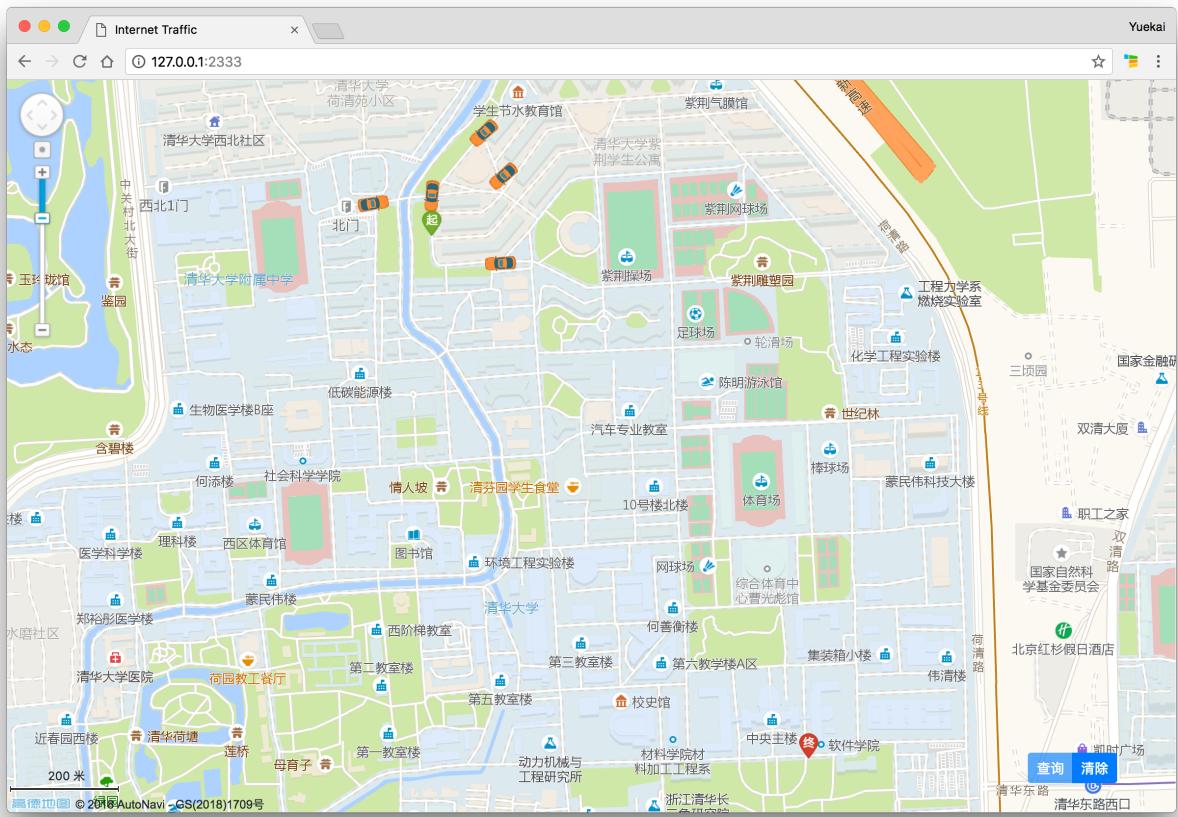
启动 Web 服务器，打开网页后的初始画面如下图所示：



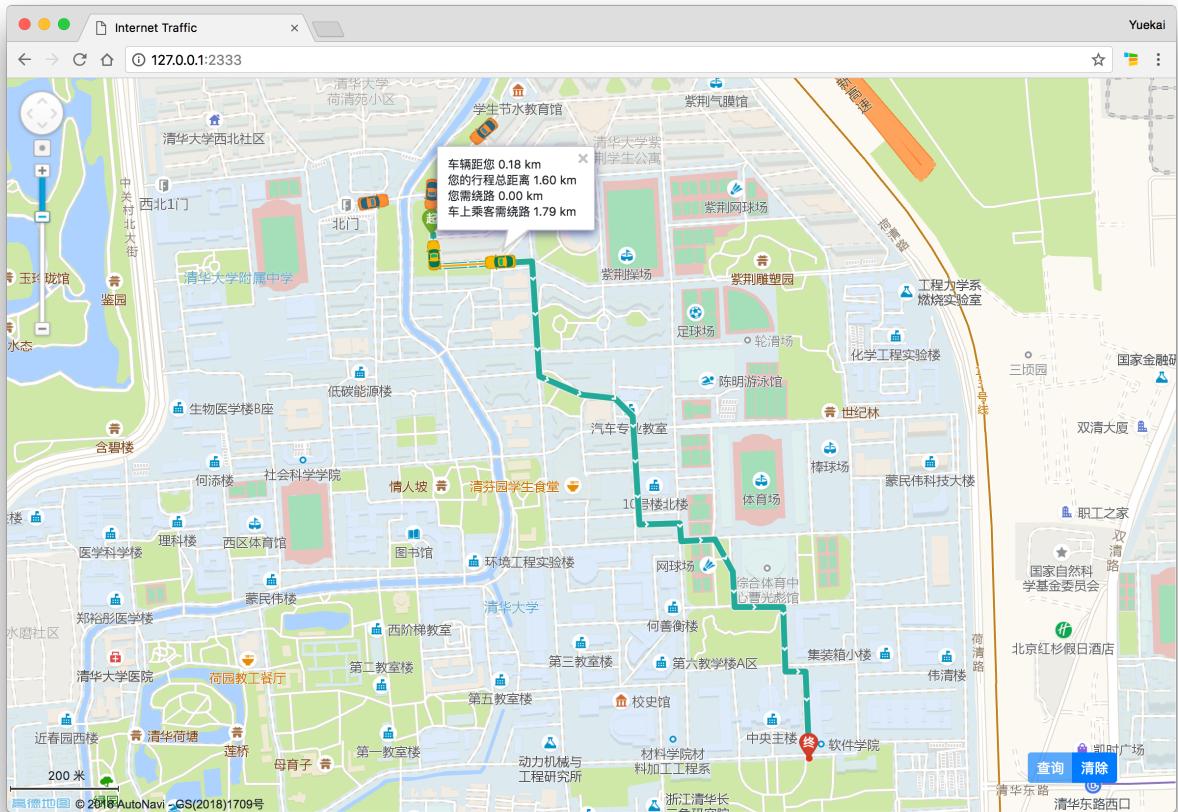
点击选择按钮，然后点击地图选择起点和终点，会自动转换为最近的路网节点：



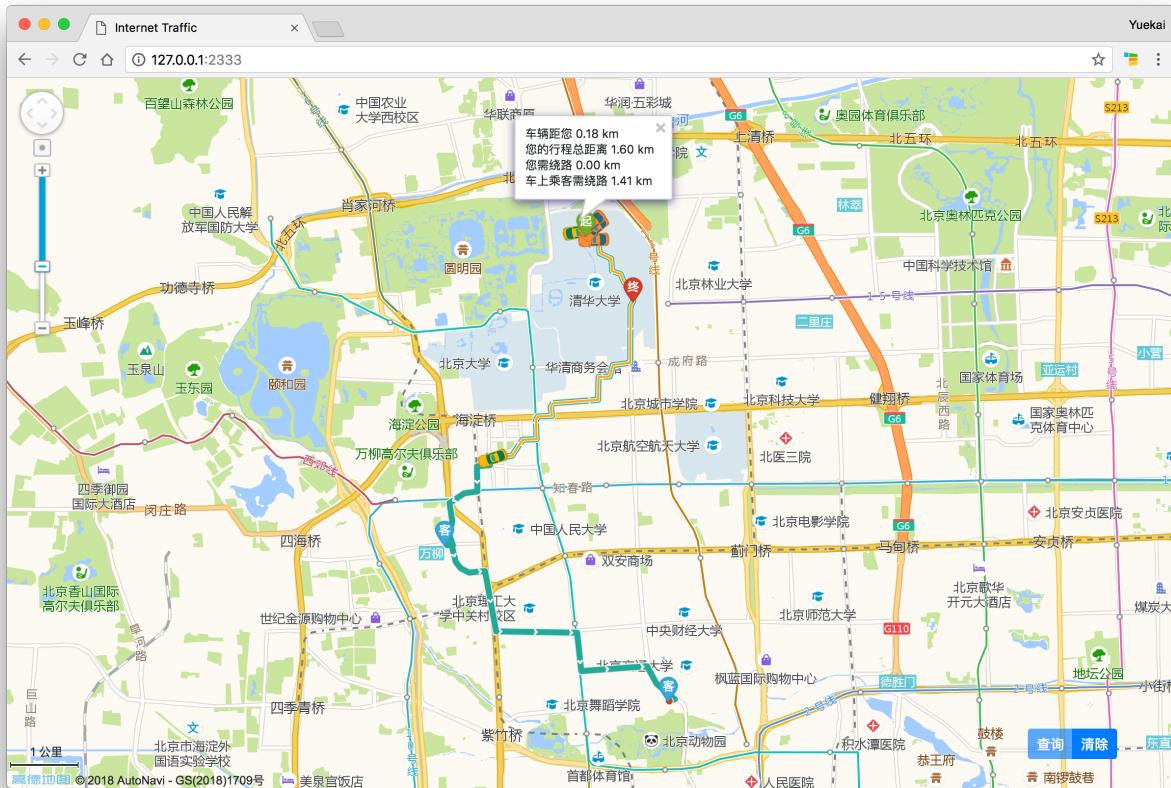
点击查询，等待大约 2 秒(取决于所选的阈值)后显示最优的 5 辆车：



点击车辆可显示计划行驶路线、绕路距离等信息，并有一个动画演示：



还可以显示车上其它乘客的目的地：



点击清除可重新开始。