最优路径规划

设计文档

目录

1 介绍和背景

1.1 简介

随着智能化技术的不断发展，人们对于交通出行的需求越来越高，尤其是在大城市中，人们的出行需求更加迫切。安徽大学磬苑校区作为一所具有一定规模的高校，每天有大量的学生和教职工在校内行走，如何进行高效、便捷的出行成为了一个亟待解决的问题。因此，开发一款校园路径规划系统，为校园内的行人和车辆提供更加高效、智能化的出行方式，对于优化校园交通管理、提高出行效率、降低交通事故率等方面具有重要意义。

1.2 项目的背景和思路

本项目的背景是基于上述需求，开发一款针对安徽大学磬苑校区的路径规划系统。该系统旨在通过智能算法的应用，为用户提供准确的路线规划、导航、出行建议等功能，以便用户更加高效地完成出行任务。

项目的整体思路如下，项目实现过程可以分为对每个步骤的实现：

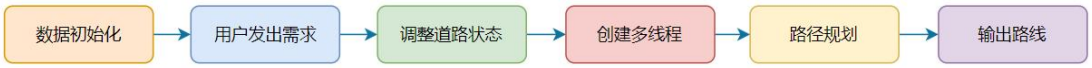


图 1 最优路径规划算法思路

（1）数据初始化：

在实现最优路径规划模拟之前，需要先对校园地图进行处理，将各个道路以及道路上的节点转化为计算机可以处理的数据结构，如图数据、邻接表等。同时，需要读取和处理一些与道路状态相关的数据，如车速、拥堵程度、限速等，以便后续使用。

（2）用户发出需求：

在实际使用中，用户需要输入起点、终点、出行时间等信息。这些信息需要在程序中进行处理，以便后续使用。用户还可以选择出行方式（步行、骑行、驾车等），程序需要根据用户的选择进行相应的路径规划。

（3）调整道路状态：

在实际出行中，道路的状态会随着时间和交通流量的变化而不断变化，如车流量、限速、拥堵程度等。因此，程序需要在实现最优路径规划时，实时调整道路状态，并根据当前状态计算最优路径。这个过程可以通过多线程实现，提高程序的运行效率。

（4）创建多线程：

为了提高程序的运行效率，可以采用多线程技术。程序可以同时处理多个路径规划请求，提高程序的响应速度。同时，多线程技术还可以实现道路状态的实时更新，保证路径规划结

果的准确性。

（5）路径规划：

路径规划是整个最优路径规划模拟的核心部分。程序需要根据起点、终点、出行时间、出行方式等信息，计算最短路径或最优路径，并考虑到多种因素，如道路拥堵程度、限速、天气等，以及用户的个性化需求，如步行偏好、安全性偏好等。常见的路径规划算法有Dijkstra 算法、A*算法、蚁群算法等。

（6）输出路线：

程序需要将最优路径输出给用户，包括路径线路、路径长度、出行时间、出行方式等信息，并展示在用户界面上。同时，程序还需要输出一些与路径规划相关的信息，如道路状态、拥堵情况等，以便用户更好地了解实时路况。

1.3 项目的范围和约束条件

本项目的范围包括但不限于以下方面：

（1）仅针对安徽大学磬苑校区的路径规划需求进行设计和开发。

（2）提供步行、骑行和驾车三种出行方式的最优路径规划。

（3）界面友好、操作简单、易于使用。

约束条件包括但不限于以下方面：

（1）系统需要考虑安全问题，确保用户的个人信息和隐私不会泄露。

（2）系统需要根据实际情况进行道路状态的调整和更新，以保证路线规划的准确性。

（3）系统需要考虑用户出行时间、天气、气温、上下课高峰等因素，提供最优的出行建议。

1.4 目标用户

本项目的目标用户主要是安徽大学磬苑校区的学生、教职工以及其他校园内的行人和车辆。随着社会对于出行效率和智能化的需求不断提高，本系统也有可能被其他高校或城市所采用。

2 需求概述

本节将详细阐述项目的需求概况，包括项目的核心功能、可选功能、假设和约束条件，以及用户案例。

2.1 需求摘要

最优路径规划模拟项目是为了在校园内提供便捷、快速、高效的路径规划服务，使得用户能够更加方便地出行。本项目旨在为用户提供三种不同的出行方式（步行、骑行、驾车）

的最优路径规划，并考虑到各种限制因素（例如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好），以便用户能够选择最适合自己的出行方式。

2.2 项目的核心功能

本项目的核心功能是为用户提供三种出行方式（步行、骑行、驾车）的最优路径规划，包括以下具体功能：

（1）用户输入出发点和目的地，以及出行时间；

用户可以在交互式地图界面上选择出发点和目的地，并输入出发时间。应用程序需要验证用户输入的信息是否合法，并将其作为计算最优路径的参数之一。

（2）根据不同出行方式和用户输入的信息，计算最优路径；

该应用程序需要根据用户选择的出行方式（步行、骑行、驾车）和用户输入的出发点、目的地以及出发时间，计算出最优路径。最优路径应该是最短的，并且考虑了各种限制因素，如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好等。

（3）考虑各种限制因素，如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好等，为用户提供最优的出行方案；

在计算最优路径时，应用程序需要考虑各种限制因素，例如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好等。例如，在下雨天，用户可能更愿意选择驾车出行，而不是步行或骑行。应用程序需要根据这些因素，为用户提供最优的出行方案。

（4）用户可以在界面上查看最优路径和推荐出行方案。

应用程序需要在交互式地图界面上展示用户的最优路径，并提供推荐的出行方案。用户可以查看这些信息，并根据自己的需要做出决策。

2.3 可选功能

除了核心功能外，本项目还可以添加以下可选功能：

（1）用户登录注册；

（2）系统反馈用户的使用记录，以及出行时长、花费等相关信息；

（3）提供用户历史记录和个性化推荐。

2.4 假设和约束条件

本项目的开发需要满足以下假设和约束条件：

（1）假设用户在输入出发点和目的地时能够准确无误地输入相关信息；

（2）假设系统能够获取到校园地图和各种限制因素的相关数据；

（3）假设系统的算法和数据结构能够处理大规模的地图和用户请求。

约束条件包括：

（1）系统响应时间应该在可接受的范围内；

（2）系统的精度应该达到可接受的标准；

（3）系统的数据安全和用户隐私保护应该得到充分保障。

3 系统设计

3.1 系统架构

3.1.1 数据结构

（1）地图数据结构

该数据结构需要存储校园地图的各个节点、边和其它相关信息，比如节点之间的距离、路段是否可通行等。可以使用图论中的邻接表或邻接矩阵。如果使用邻接表，每个节点的信息包括节点编号、节点名称、相邻节点编号及相邻节点到该节点的距离等。如果使用邻接矩阵，则需要使用一个矩阵来表示节点之间的距离或权重，如果两个节点之间不连通，则在矩阵中对应的元素为无穷大。

（2）用户输入数据结构

该数据结构需要存储用户输入的出发点、目的地和出行时间等信息。用户输入数据结构包括出发点、目的地和出行时间等信息。可以使用一个包含各种属性和方法的对象来表示。

（3）路径规划数据结构

该数据结构需要存储路径规划过程中的中间状态，如已访问节点、当前最短路径、起点到当前节点的最短路径等。可以使用堆或优先队列来实现，以快速获取当前距离起点最近的未访问节点，并且在访问节点时更新当前最短路径。

（4）推荐方案数据结构

该数据结构需要存储每种出行方式的最优路径及其缘由，如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好等。可以使用哈希表或映射表来实现，以便快速获取最优路径及其缘由。

（5）界面数据结构

该数据结构需要存储界面相关信息，如用户输入框、地图显示区域、推荐方案列表等。可以使用面向对象的方式来实现，比如定义一个包含各种属性和方法的地图对象、一个包含各种属性和方法的路径对象、一个包含各种属性和方法的用户输入对象等。这些对象可以通过面向对象的继承、组合等方式来实现。

3.1.2 分析建模

（1）功能模型——数据流图

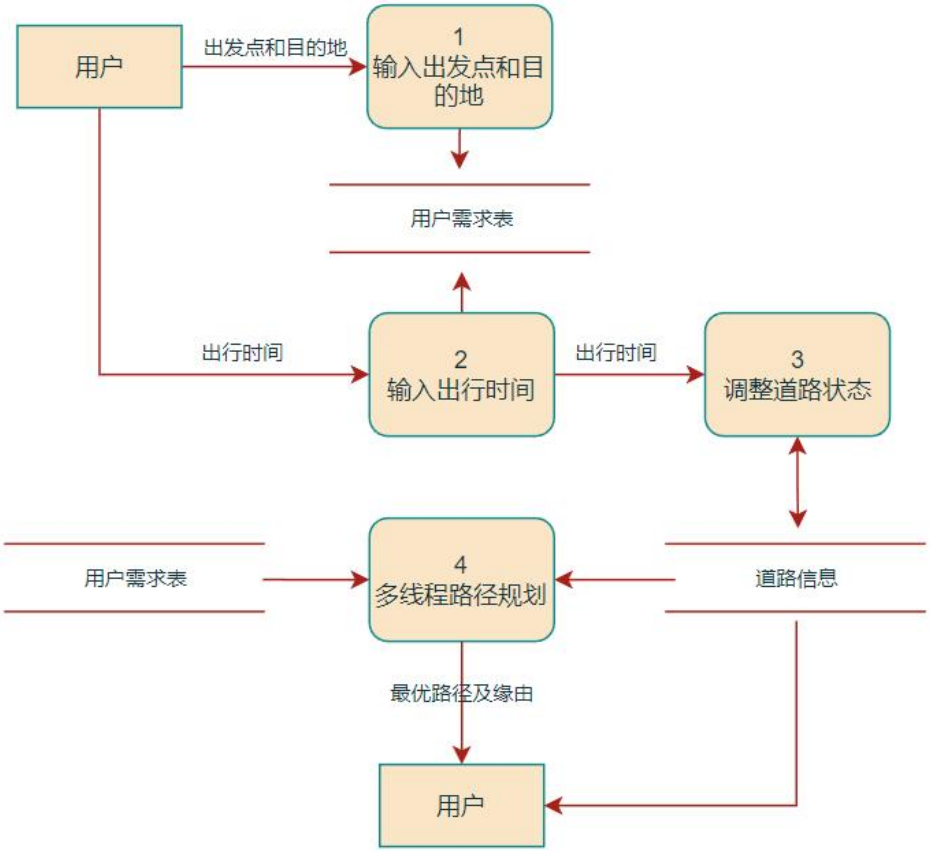


图 2 最优路径规划的数据流图

数据流图通过展示数据流向以及处理过程，清晰地展示了路径规划系统的整体架构，同时也能够更加具体地描述各个模块之间的关系和作用。

在路径规划系统中，用户首先输入出发点和目的地，然后输入出行时间。这些信息都需要被保存在系统中以供后续的处理使用。出行时间被送入调整道路状态的模块中，此模块会利用用户选择的出行时间来收集各种限制因素，例如天气、气温、上下课高峰等，并结合停车点、限速点、个性化偏好等，以此来调整道路状态。

调整好的道路状态将会被储存，与用户需求一同流入多线程路径规划模块。在此模块中，系统会根据用户的起讫点、出行时间等需求，结合道路状态，采用多线程的方式规划出步行、骑行和驾车三种出行方式的最优路径。为每种出行方式建立一个安全线程，可以更加有效地保证规划的准确性和速度。

在多线程路径规划完成后，系统会将最优路径输出给用户，供其选择。在此过程中，用户可以通过界面查看最优路径和推荐出行方案。整个系统流程清晰明了，使用户的出行体验更加愉悦和高效。同时，该系统还可以根据不同情况进行优化和升级，以适应不同用户的需求和变化的环境因素。

（2）数据模型——ER 图

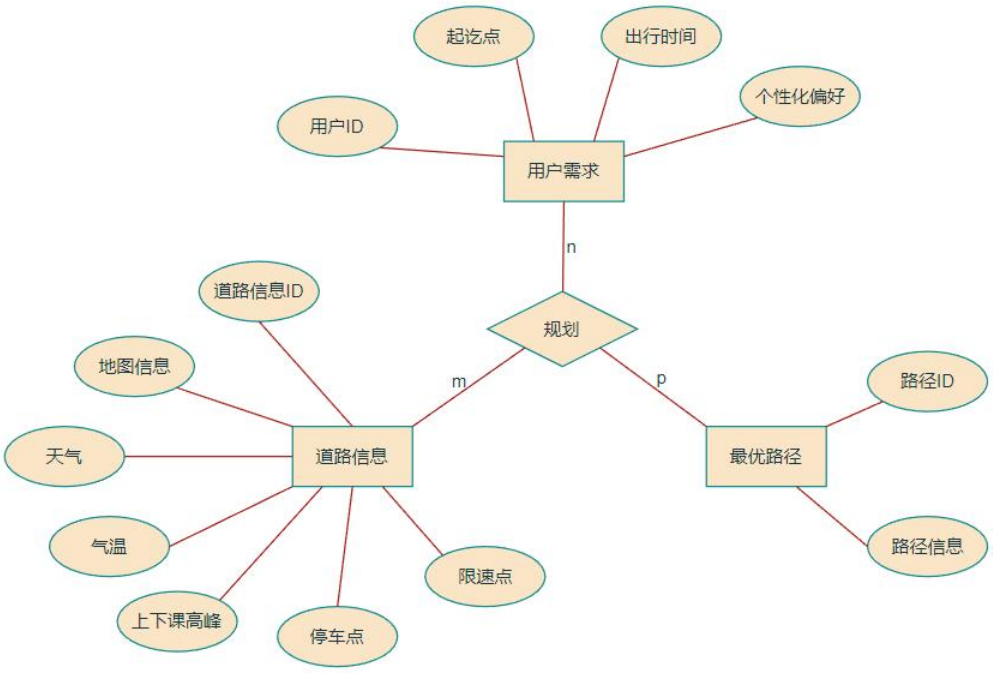


图 3 最优路径规划的 ER 图

在一个复杂的系统中，实体是指系统中的各种对象，可以是人、事物、概念等，而实体联系图就是用来描述这些实体之间的关系的一种工具。在本系统中，有三个主要的实体，分别是用户需求、道路信息和最优路径。下面将分别对这三个实体进行更详细的描述。

① 用户需求：用户 ID、起讫点、出行时间、个性化偏好

首先是用户需求。这个实体包含了四个属性，分别是用户 ID、起讫点、出行时间和个性化偏好。这些属性都是用户输入的信息，用来帮助系统进行路径规划。其中，起讫点和出行时间是最基本的信息，用来确定用户的出行需求；而个性化偏好则是指用户在选择出行方式时的特殊要求，比如偏好步行或者不喜欢坐地铁等。

② 道路信息：道路信息 ID、地图信息、天气、气温、上下课高峰、停车点、限速点

第二个实体是道路信息。这个实体包含了七个属性，分别是道路信息 ID、地图信息、天气、气温、上下课高峰、停车点和限速点。这些属性都是道路信息的特征，可以用来帮助系统进行最优路径的规划。具体来说，地图信息包含了道路的长度、形状和位置等基本信息，天气和气温则可以影响道路的通行情况，上下课高峰可以影响道路的拥堵程度，停车点和限速点则是指在道路上的具体限制和便利设施。

③ 最优路径：路径 ID、路径信息

最后是最优路径这个实体，包含了路径 ID 和路径信息两个属性。路径 ID 是用来标识这

条路径的唯一标识符，路径信息则是指该路径的具体细节，比如起讫点、所需时间、路线等。需要注意的是，一条最优路径可以对应多个用户需求和道路信息，因为不同用户有不同的出行需求，而不同的道路信息可能会影响到路径规划的结果。

三个实体之间的联系是通过规划关系进行连接的。规划关系是指道路信息和用户需求之间的联系，以及用户需求和最优路径之间的联系。具体来说，道路信息会根据用户需求进行调整，以适应不同的出行需求，但是不同用户的需求可能对应同一种道路信息。同时，用户需求和道路信息会一起用于最优路径的规划，而一条最优路径可能会对应不同的用户需求和道路信息，因为不同用户有不同的出行需求，而不同的道路信息也会影响到路径规划的结果。

（3）行为模型——状态转换图

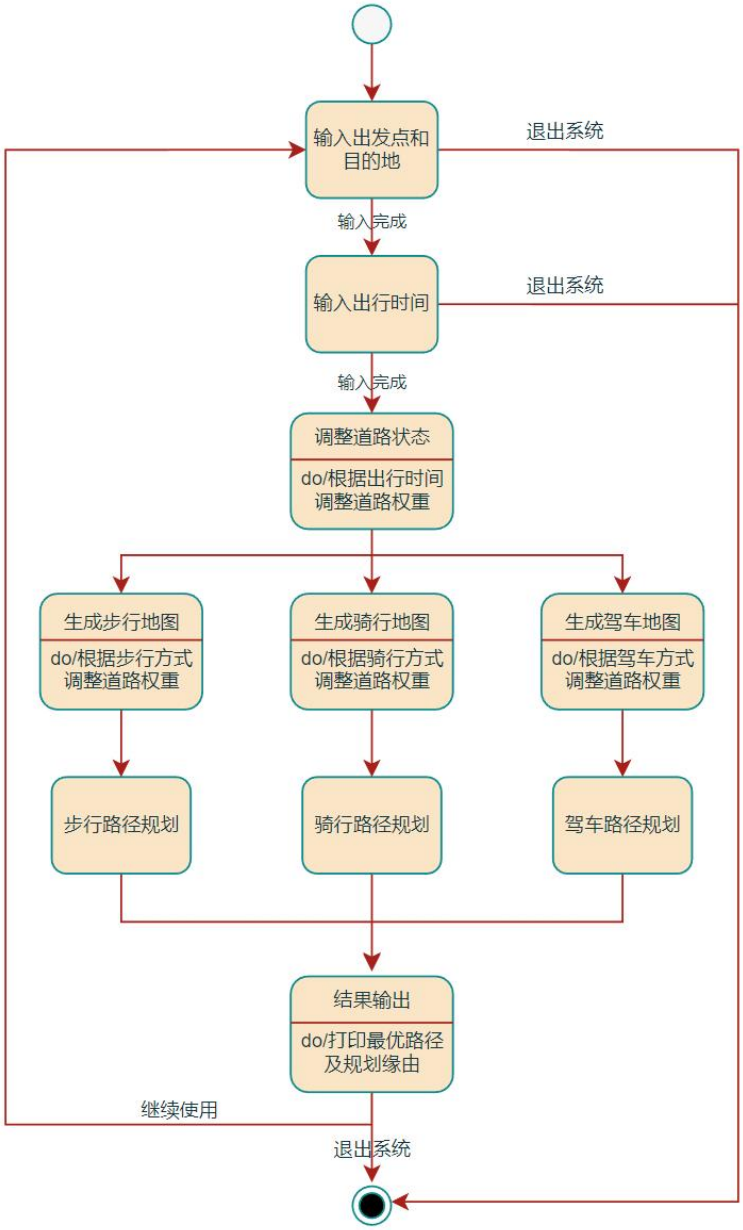


图 4 最优路径规划的状态转换图

在状态转换图中，系统会依次要求用户输入出发点、目的地和出行时间。在这个过程中，系统会根据用户的输入来收集必要的信息，如起讫点坐标和出行时间等，这些信息将作为路径规划的输入。同时，系统允许用户随时选择退出系统，以提高用户的交互体验。

接下来，系统进入调整道路状态，这个过程中，系统会根据用户输入的需求收集影响因素，如天气、气温等。根据这些影响因素，系统会调整道路权重，以便在后续的路径规划过程中更准确地反映出道路的实际情况。具体来说，系统会将道路长度乘以一个权重因子，这个权重因子反映了道路的通行状况，由此改变该道路在路径规划时被选中的概率。调整完毕后，系统会建立三个安全线程分别进行步行、骑行、驾车的路径规划。

在规划之前，系统会根据不同出行方式再次调整道路，即为每一种出行方式制作出一个专属的地图。这个地图会反映出该出行方式下道路的通行情况，例如有些道路适合步行，但是并不适合骑行或驾车。紧接着，各个线程会在这个专属地图上做出路径规划，给出最优路径并生成规划缘由。在路径规划时，系统还会考虑用户的个性化偏好，例如有的用户喜欢走风景优美的路线，而有的用户则更注重时间和速度。

最后，所有路径规划完毕后，系统会将结果进行输出。此时用户可以再次选择是否退出系统，如果不退出则可以继续使用，系统回到等待用户输入需求的状态。整个过程中，系统会根据不同实体之间的关系，以及用户输入的信息来进行数据处理和路径规划，最终给出最优的路径和规划缘由。这个系统能够提供高质量的路径规划服务，让用户出行更加便捷和安全。

3.1.3 软件结构图

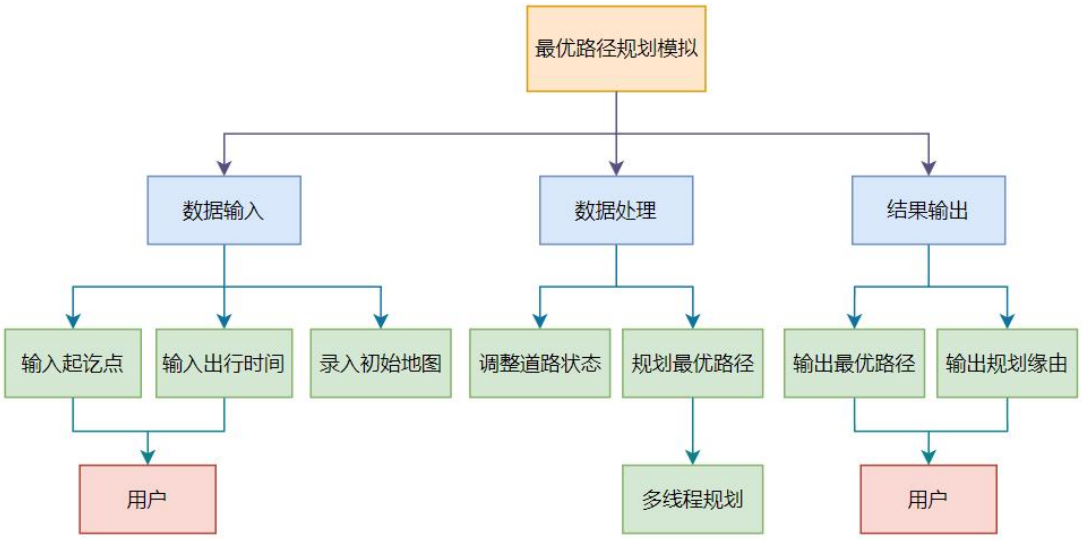


图 5 最优路径规划的软件结构图

软件结构图是一种表示软件系统组成部分及其相互关系的图形化工具，它可以用于描述软件系统的模块、子系统和组件之间的关系和通信方式。

在本项目的软件结构图中，将系统分为了数据输入、数据处理、结果输出三大部分。

其中，数据输入一部分由用户输入的起讫点和出行时间组成，另一部分由系统录入的初始地图组成。用户输入的起讫点和出行时间是系统进行路径规划的基本信息，而系统录入的初始地图则是路径规划的基础数据。

数据处理部分主要包括两大模块，一个是调整道路状态，该部分通过用户提出的需求对初始地图上各个道路的权重进行更新，注意这里理应生成三个不同的地图，用于对应步行、骑行、驾车三种不同的出行方式；另一块是规划最优路径，这一部分需要牵扯到多线程服务。系统会为三种出行方式分别建立一个安全线程，三个线程同时服务，同时规划，规划完毕后共同输出。

结果输出是在三种路径都规划完毕后进入的状态，系统会将已经规划好的路径以及规划缘由一起输出给用户。在这一部分中，需要考虑输出信息的可读性和可视化效果，让用户能够清晰地了解路径规划结果。同时，还需要考虑输出信息的安全性和隐私保护，避免用户的个人信息泄露。

对用户来说，中间所有的处理过程都是透明的。用户只能看到自己选择好起讫点和出行时间后，系统生成已经规划好的路线，中间的过程都在系统后台完成，让用户使用起来方便快捷。

3.2 模块设计

3.2.1 数据初始化模块

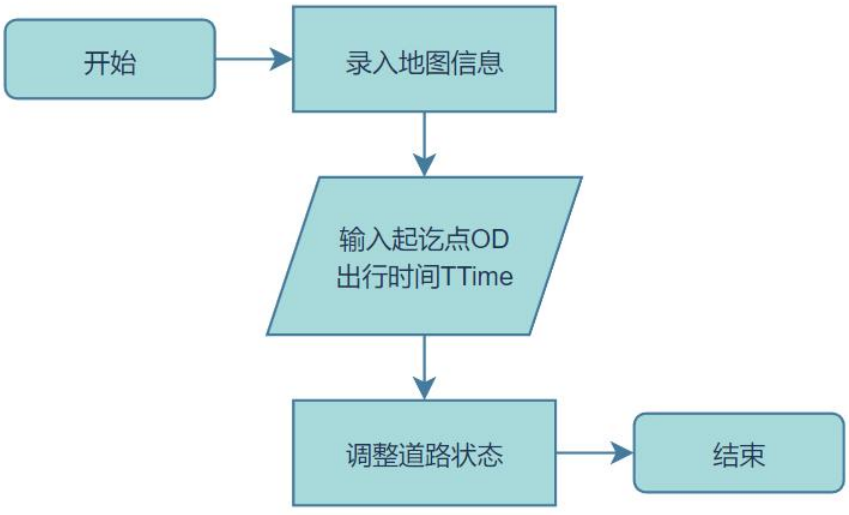


图 6 数据初始化模块流程图

该部分主要用于生成路径规划前的地图信息。

首先系统会录入初始的地图信息，同时软件也会将该信息显示给用户。在用户输入起讫点和出行时间后，系统会根据出行时间收集影响因素，并根据影响因素调整道路状态。最后调整后的道路状态将送入路径规划部分用于规划最优路径。

具体而言，系统会为每条道路乘以一个权重，用于改变道路在路径中的占比，进而影响到被选中的概率。

影响因素包括以下几个方面（表格内数据为权重因子）：

（1）天气：

	步行	骑行	驾车
晴	1	1	1
小雨	1.2	1.2	1.2
中雨或大雨	1.5	2	1.5
微风	1	1	1
大风	1.2	1.5	1
小雪	1.2	1.2	1.2
中雪或大雪	1.5	2	1.5
多云	1	1	1

（2）气温：

	步行	骑行	驾车
寒冷（T<=0℃）	1.5	2	1
低温（0℃<T<=15℃）	1.2	1.5	1
舒适（15℃<T<=25℃）	1	1	1
高温（25℃<T<=35℃）	1.2	1	1
炎热（35℃<T）	1.5	1	1

（3）上下课高峰：

	步行	骑行	驾车
高峰期	1.2	1.5	2
空闲期	1	1	1

（4）停车点：

	步行	骑行	驾车
有停车点	1	1	1
无停车点	1	1.2	1.5

（5）限速点：

	步行	骑行	驾车
有限速点	1	1	1
无限速点	1	1	1.5

（6）出行方式

该部分决定了某些道路是否可以同行，因为有些道路只能步行而不能骑行或驾车；或只能步行、骑行而不能驾车。具体如下：

	步行	骑行	驾车
不能步行	∞		
不能骑行		∞	
不能驾车			∞

3.2.2 路径规划模块

路径规划是一个典型的图论问题，可以转变成求图中的最短路径。对于这类问题，我们常选用的算法有两种，分别是 Dijkstra 算法和 A*算法。

（1）Dijkstra 算法

Dijkstra 算法是一种经典的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。它的基本思想是以起点为中心向外层层扩展，直到扩展到终点为止。下面将对 Dijkstra 算法进行详细说明：

1. 初始化：

将起点 s 到各个点的距离 d[s]初始化为 0，其他点到起点的距离 d[v]初始化为无穷大（即不存在这条路径），s 作为集合 S 的唯一元素，V-S 中的节点 v 加入集合 Q。

2. 迭代过程：

重复进行以下操作，直到集合 Q 为空：

① 在集合 Q 中找到距离起点 s 最近的节点 u。

② 将 u 加入集合 S 中，并从集合 Q 中删除。

③ 对于 u 的每个邻居节点 v，如果 d[u]+w(u,v)<d[v]，则更新 d[v]的值，表示通过 u

可以到达 v 的距离更短。

3. 最终结果：

当集合 S 包含终点 t 时，算法结束。此时 d[t]就是起点 s 到终点 t 的最短路径长度，同时通过记录每个节点的前驱节点，可以回溯出 s 到 t 的最短路径。

Dijkstra 算法的优点是算法思路简单，实现较为容易，且能够处理带权图的单源最短路径问题。但是，该算法要求所有边的权值都为非负数，否则结果可能不正确。此外，该算法只能用于有向无环图（DAG）或者正权有向图。本项目所面临环境正好满足该条件。

总之，Dijkstra 算法是一种十分经典的最短路径算法，具有广泛的应用场景，在网络路由、交通运输等领域都有着重要的作用。下面是其算法流程图：

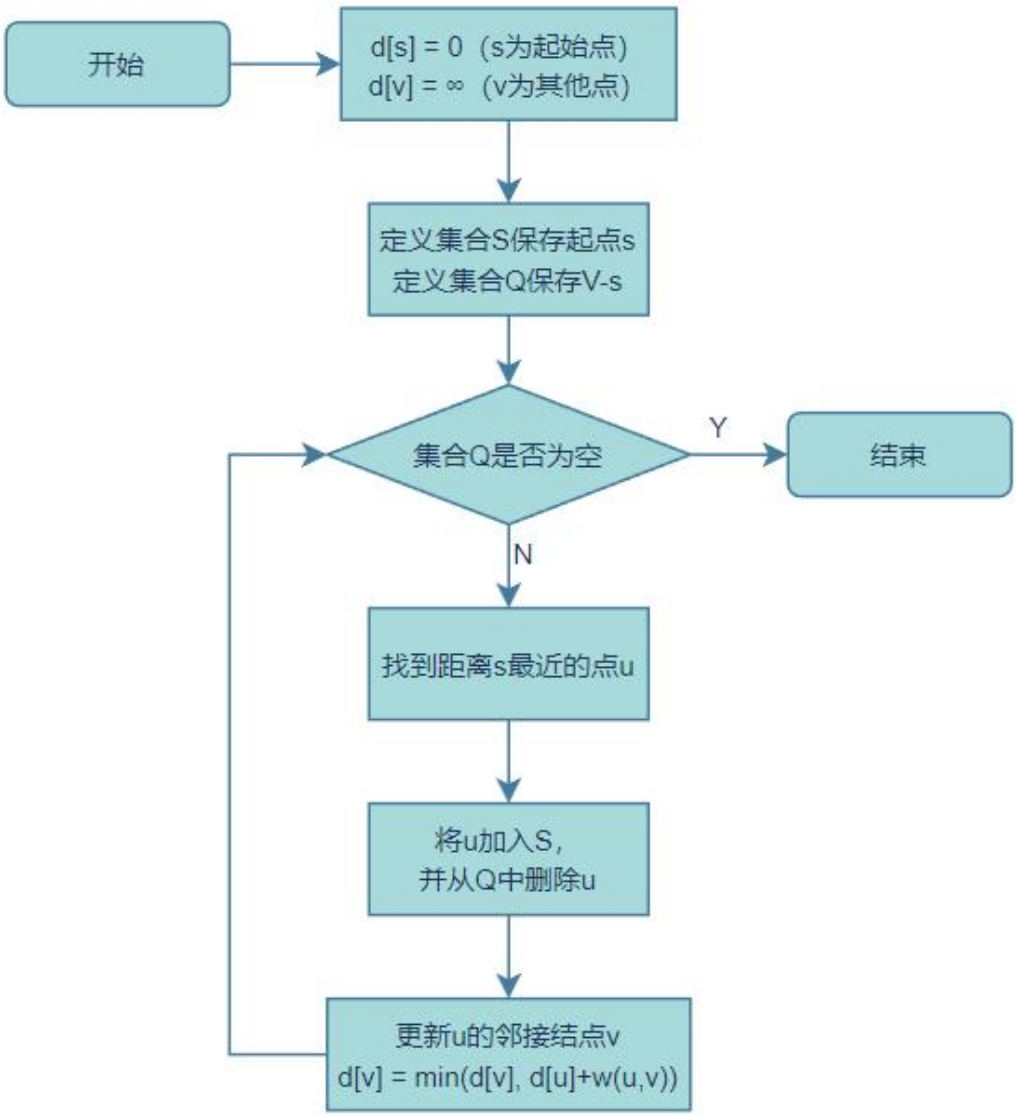


图 7 Dijkstra 算法流程图

（2）A*算法

A*算法是一种启发式搜索算法，用于解决带权有向图的单源最短路径问题。它在Dijkstra算法的基础上，利用启发式函数估计从当前节点到目标节点的距离，以此来加速搜索过程。下面将对A*算法进行详细说明：

1. 初始化：

将起点s加入开放列表openSet，起点到起点的距离g(s)为0，f(s)=h(s)表示启发式函数值，h(s)表示起点到终点的估算距离（常用欧氏距离或曼哈顿距离），起点没有父节点。

2. 迭代过程：

重复进行以下操作，直到终点t被加入到开放列表中，或者开放列表为空：

① 在开放列表openSet中找到f值最小的节点n。

② 如果n为终点t，则算法结束，从终点回溯得到最短路径。

③ 将n从开放列表中删除，并将其加入到关闭列表closedSet中。

④ 对n的所有邻居节点v，如果v不在关闭列表中，则对于v进行以下操作：

如果v不在开放列表中，则将v加入到开放列表，并设置v的父节点为n，计算g(v)和f(v)；

如果v已经在开放列表中，则比较从n到v的路径是否更优，如果更优则更新v的父节点和g值，并重新计算f(v)。

3. 最终结果：

当终点t被加入到开放列表中时，算法结束。此时从终点t开始回溯得到的路径就是起点s到终点t的最短路径。

A*算法的优点是在Dijkstra算法的基础上，加入了启发式函数来指导搜索方向，因此在搜索过程中，能够更快地接近终点，从而加快搜索速度。此外，A*算法也能够处理存在负权边的情况，只要启发式函数满足一定条件，就能保证搜索的正确性。

总之，A*算法是一种十分高效、广泛应用的最短路径搜索算法，能够在多种领域中得到应用，如游戏AI、机器人路径规划、地图导航等。下面是其算法流程图：

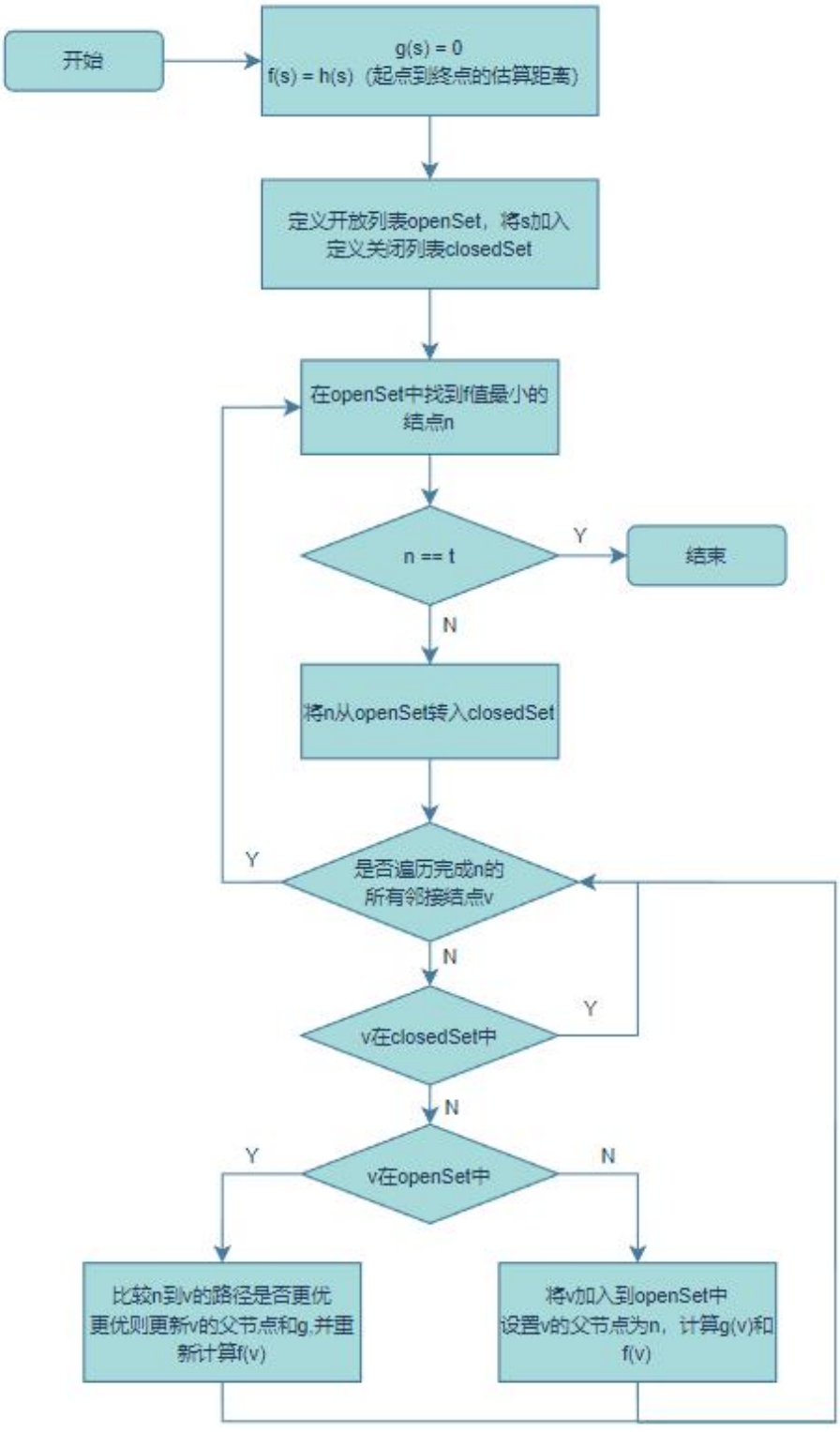


图 8 A*算法流程图

3.2.3 多线程设计模块

该模块用于系统中对多个线程的设计。本系统中，由于需要对步行、骑行、驾车三种出行方式的最优路径进行规划，想要提高运行效率，不可避免地会运用到多线程的技术。

系统中，我们为每种路径单独生成了一个安全线程供其进行路径规划，在生成之前，需要注意读取到数据初始化模块中调整后的对应地图。因为每种方式背后的道路状态都是不同的。算法流程图如下：

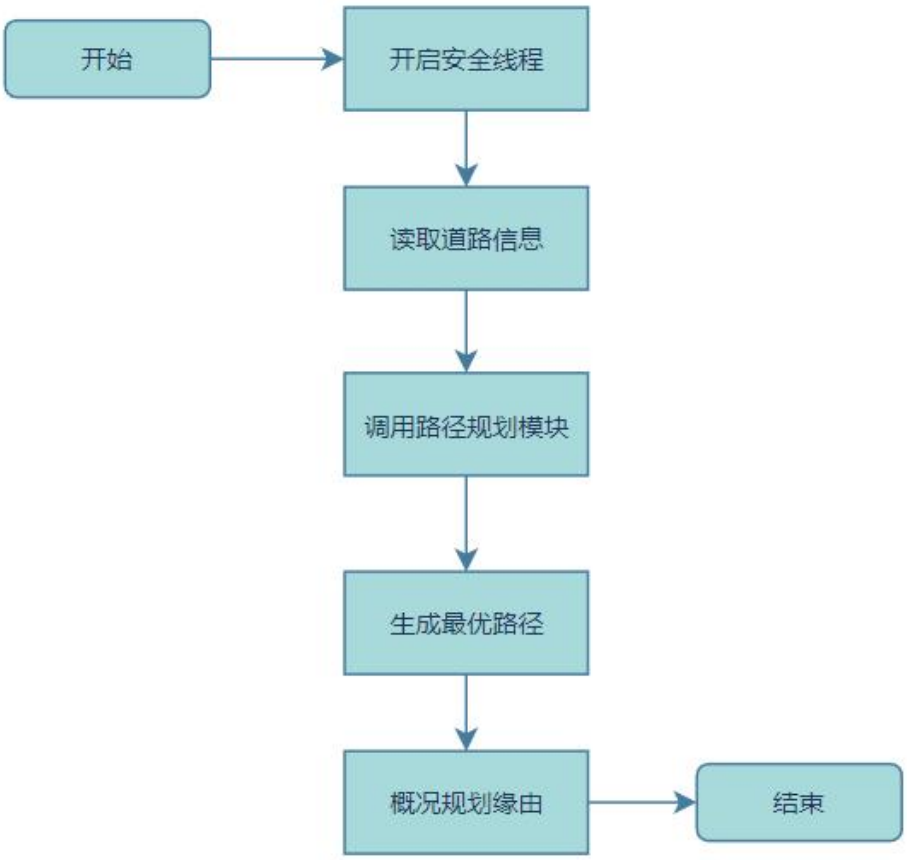


图 9 多线程设计算法流程图

3.2.4 显示模块

显示模块贯穿于系统始终，用户在使用软件的过程中无时无刻不在读取屏幕上的信息。该模块的设计虽不会影响系统后台的运行，但直接关系到用户的使用体验，一个好的内容显示模块应当将信息准确、清晰、有条理地进行显示。

在该系统中，显示的内容具体包括如下部分：

（1）地图信息

该部分是用户一直查阅的内容，后续所有的算法或内容显示都是基于此之上的。系统会将地图直接展示在屏幕上，让用户对地图的信息，包括地点、路径、名称等等有一个最直观的了解。地图中，名称会之间标注在地点之上；路口用蓝色标点，路径则用白色，整体地图简约整洁。

（2）用户需求

系统会将用户所提出的起讫点以及出行时间展示在屏幕上，这部分内容直接放置在文本框内，用于用户审核系统是否正确地捕获了自己的需求，当发现有错误时，可以及时调整。

（3）影响因素

系统会将包括天气、气温、上下课高峰等等出行影响因素显示出来，以文本方式让用户知道当前时刻道路上的状态。包括后续最优路径中的推荐缘由，也是基于此部分向用户表明。

（4）最优路径

最优路径包括两部分，一是路径本身，二是推荐缘由。对于路径本身，系统会在地图上直接利用不同颜色将路径绘制出来，用户可以基于此查阅到系统为其推荐的各个最优路径；对于推荐缘由，可以采用文本框、图表等多种方式，最主要的目的是让用户能够准确直接地知道系统为什么会这样规划，以帮助用户做出最终决策。

3.2.5 界面设计模块

系统界面通过 Vs+Qt 实现，包括各个窗口的生成，标签、按钮、文本框等各个部件的添加和组合，以及前后端的连接和页面的刷新等。

具体流程如下：

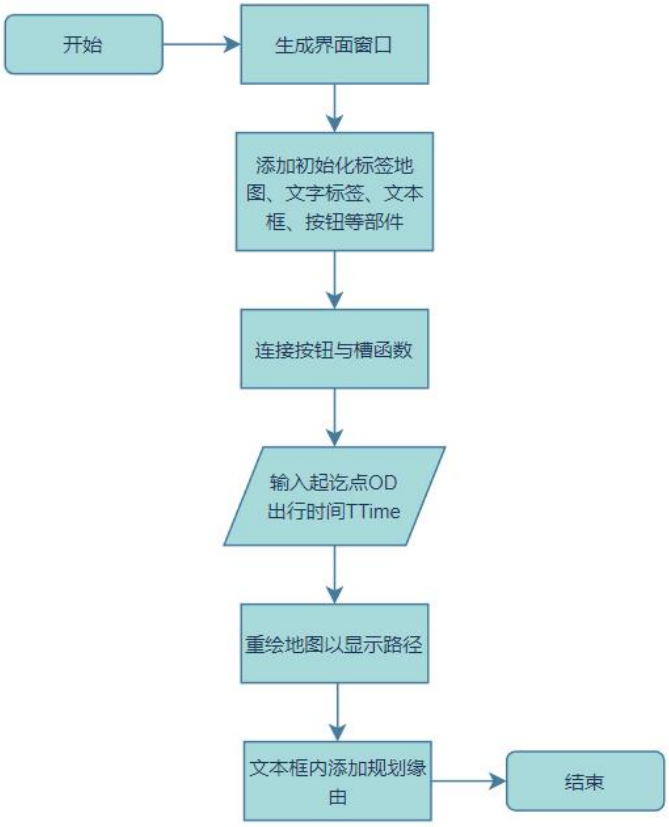


图 10 界面设计模块算法流程图

3.3 界面设计

（1）地图样貌

（2）显示路径、路标等信息

（3）显示最优路径

下图显示了一条从竹园到博北的最优路径：

4　系统规格说明

4.1　功能编号和优先级

表 1　功能编号和优先级

功能编号	功能	优先级
1	接受用户数据	中
2	系统初始化	中
3	创建安全线程	中
4	规划路径	高
5	内容显示	低

4.2　加工说明

表 2　各功能对应 IPO

功能	输入	处理	输出
接受用户数据	起讫点和出行时间	系统接受用户数据	
系统初始化	用户数据	系统根据输入信息生成初始化数据	道路信息
创建安全线程		为三种出行方式创建规划线程	①步行规划线程 ②骑行规划线程 ③驾车规划线程
规划路径	①规划线程	为每种出行方式规	最优路径与缘由

	②道路信息	划最优路径	
内容显示	①道路信息 ②最优路径与缘由	打印内容	

4.3 性能需求

4.3.1 数据精度

表 3 系统各字段及精度

字段	精度
起讫点 B/EPoint	String 型
出行时间 BTime	int 型

4.3.2 时间特性

（1）响应时间：用户在任意的操作后，系统能在 1s 内做出反应

（2）数据处理时间：视系统运行状态决定

4.3.3 灵活性

当用户提出的起讫点和出行时间发生改变时，整体系统的结构、操作流程、运行环境等不会发生改变，只会修改系统运行中的数据。

4.4 运行需求

4.4.1 软件接口

（1）操作系统：Microsoft Windows 10、11

（2）软件设备：Visual Studio 2022

4.4.2 硬件接口

（1）内存：512M 以上

（2）磁盘空间：40G 以上

（3）CPU：333Mhz 以上

（4）硬盘空间：1.5G 以上

4.5 其他需求

4.5.1 质量属性

（1）可用性：用户与开发人员很容易上手使用，用户界面符合操作习惯

（2）可靠性：在给定时间内系统可以大致上满足无错运行的要求

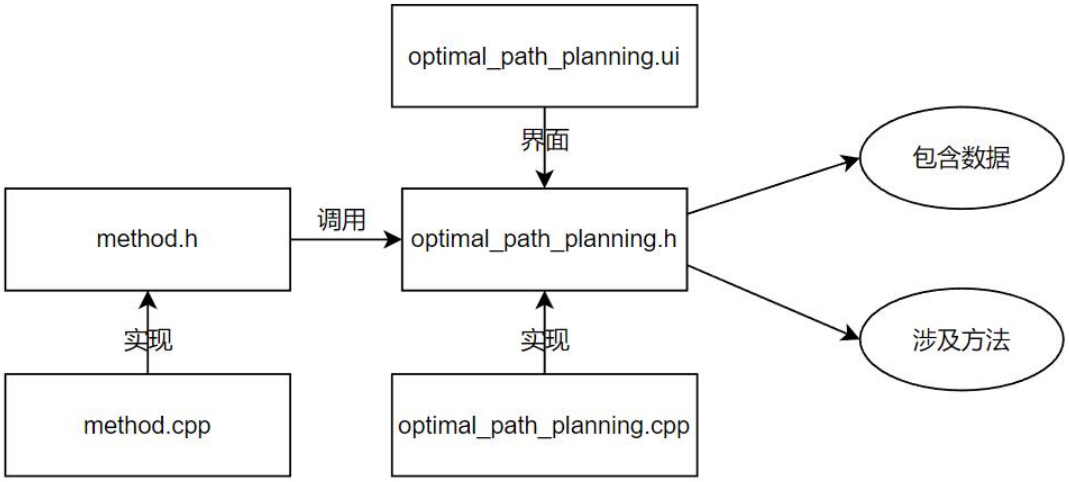
（3）可维护性：系统运行会将行为写进日志以供维护工作的进行

（4）安全性：定时对关键数据、关键程序进行备份，防止数据丢失

（5）可移植性：移动端之间可以互相移植

5 系统实现

5.1 程序框架



整个代码分为两个主要部分。首先是"optimal_path_planning"，负责对界面事件做出相应。当用户需要规划路线时，"optimal_path_planning"会调用"method"方法开始路线规划。

"method"方法主要考虑当前的气候、天气条件以及出入高峰时段、停车点分布和限速路段分布等因素，以规划出一条最优路线。它综合考虑这些因素，并返回规划结果给"optimal_path_planning"，最后将结果显示在界面上。

5.2 程序总览

5.2.1 optimal_path_planning.h

```h
#pragma once

#include <QtWidgets/QMainWindow>
#include "ui_optimal_path_planning.h"
#include <Qwidget>
#include "QMouseEvent"
#include <QPainter>
#include <map>
#include <string>
#include "algorithm"
#include "random"
#include "method.h"
using namespace std;

class optimal_path_planning : public QMainWindow
{
    Q_OBJECT

public:
    optimal_path_planning(QWidget *parent = nullptr);
    ~optimal_path_planning();

    void Draw(void);
    void Init(void);

    string GetShow(int mode);

protected:
    void mousePressEvent(QMouseEvent* event) override;

private:
    Ui::optimal_path_planningClass ui;

    //初始数据
    map<int, QPoint> Point; // 路口
    // 键值={id, (路口,路口)}
    map<int, pair<int, int>> Route;  // 道路
    // 键值={(路口,路口), 距离}
    map<pair<int, int>, int> map_method; //地图 (用于传给method)
    map<int, QPoint> Location; // 建筑
    map<int, string> Name;     // 路口-名字
    map<int, int> L2M;         // 建筑-路口

    map<int, string> Weather2Name;   //气候-名字
    map<int, int> Time2T;            //时间-气温

    //出行信息
    int Weather = 0; // 保存气候
    int T = 0;       // 保存气温
    int hour = 0;    // 出行时间
    int minute = 0;  // 出行时间
    vector<int> Way = { -1, -1 }; // 出行路线

    //规划信息
    vector<vector<vector<int>>> Planning; // 规划路线
    int mode_Plan = 0; // 设置显示规划模式

private slots://相应功能槽函数
    void Adjust(void);       //调整设置模式
    void Delete(void);       //删除中途地点
    void ShowLandmark(void); //显示地标
    void updateWT(void);     //更新气候
    void affirm(void);       //确认 (开始规划)
    void change(void);       //更改显示规划
};
```

"optimal_path_planning.h"中设计了一个名为"optimal_path_planning"的类，继承自QMainWindow，用于实现最优路径规划的功能。该类通过用户界面与用户进行交互，并调用"method"类来进行路线规划。

（1）代码中的主要成员函数包括：

Draw()：绘制界面的函数。

Init()：初始化数据的函数。

GetShow(int mode)：根据给定的模式参数返回相应的显示信息。

mousePressEvent(QMouseEvent* event)：处理鼠标点击事件的函数。

（2）代码中的成员变量包括：

ui：用户界面相关的对象。

Point：存储路口的坐标信息。

Route：存储道路的起点和终点。

map_method：用于传递给"method"类的地图数据。

Location：存储建筑的坐标信息。

Name：存储路口的名称。

L2M：存储建筑与路口之间的关系。

Weather2Name：存储气候与名称之间的对应关系。

Time2T：存储时间与气温之间的对应关系。

Weather：保存当前的气候信息。

T：保存当前的气温信息。

hour 和 minute：保存出行时间。

Way：保存出行路线的起点和终点。

mode：地点模式设置。

isShowLandmark：是否显示地标的设置。

Planning：存储规划的路线信息。

mode_Plan：显示规划模式设置。

代码中还包括一些槽函数（slots），用于响应界面上的各种功能按钮的点击事件，例如调整设置模式、删除中途地点、显示地标、更新气候、确认开始规划和更改显示规划等。

总体来说，这段代码实现了这个最优路径规划的整体系统，用户可以通过界面进行设置，包括选择起点和终点、调整设置模式、显示地标等，然后系统根据用户的设置和当前的气候、

时间等信息，使用"method"类进行路线规划，并将结果显示在界面上。

5.2.2 method.h

```
#pragma once

#include "vector"
#include "string"
#include "windows.h"
#include "map"
#include <queue>
#include <iostream>
#include "thread"
#include "mutex"
using namespace std;

//出行信息表(Travel information table)
typedef struct TIT {
    map<pair<int, int>, int> map_Walk;  //步行地图
    map<pair<int, int>, int> map_Ride;  //骑行地图
    map<pair<int, int>, int> map_Drive; //驾车地图

    vector<int> stopRide;  // 停车点
    vector<int> stopDrive; // 停车点
    vector<pair<int, int>> Limit; // 限速路段
    vector<pair<pair<int, int>, pair<int, int>>> Peak; // 高峰时间段

    vector<int> Way;   // 出行路线

    vector<vector<vector<int>>> Planning; //规划路线
    int flagWalk = 0, flagRide = 0, flagDrive = 0;

    int Weather; // 天气
    int T;       // 气温
    int hour;    // 出行时间
    int minute;  // 出行时间

    // 当前时间是否在某个时间段中
    bool isInPeriod(pair<pair<int, int>, pair<int, int>> b) const { ... }
    // 当前时间是否在高峰时间段中
    bool isInPeak() const { ... }
    // 该道路是否是限速路段
    bool isInLimit(pair<int, int> a) const { ... }
    // 该道路周围是否含有停车点 (flag = 1判断骑行; flag!=1判断驾车)
    bool isInStop(pair<int, int> a, int flag) const { ... }
}TIT;

class method {
public:
    method(int hour, int minute, vector<int> Way, int Weather, int T, map<pair<int, int>, int> map_method);
    ~method();
    void init(void);    //数据初始化 (包括调整地图信息)

    //规划两点之间的最优路径
    static void Dijkstra(int Start, int End, map<pair<int, int>, int>& map_method, vector<int>& ans, int& flag);
    static void createThread(method*);              //创建安全线程
    void route_Planning(void);                      //路径规划
    vector<vector<vector<int>>> get_Planning(void); //返回规划路线
    bool get_flag(void);

private:
    TIT tit;
};
```

"method.h"定义了一个名为"method"的类，用于执行最优路径规划的具体算法。该类包含了一些成员变量和成员函数来实现路径规划的功能。

（1）代码中的主要成员函数包括：

init()：数据初始化函数，包括对地图信息的调整。

Dijkstra(int Start, int End, map<pair<int, int>, int>& map_method, vector<int>& ans, int& flag)：执行 Dijkstra 算法来计算两点之间的最优路径。

createThread(method*)：创建安全线程。

route_Planning()：路径规划函数，用于执行最优路径规划的具体步骤。

get_Planning()：获取规划路线的函数，返回规划的路线结果。

get_flag()：获取标志位的函数,返回规划是否完成的标志。

（2）代码中还定义了一个名为"TIT"的结构体，用于存储出行信息表。

该结构体包含了多个成员变量，包括步行地图、骑行地图、驾车地图、停车点、限速路段、高峰时间段、出行路线等信息。结构体中还定义了一些成员函数，用于判断当前时间是否在某个时间段中、判断当前时间是否在高峰时间段中、判断某道路是否是限速路段、判断某道路周围是否含有停车点等。

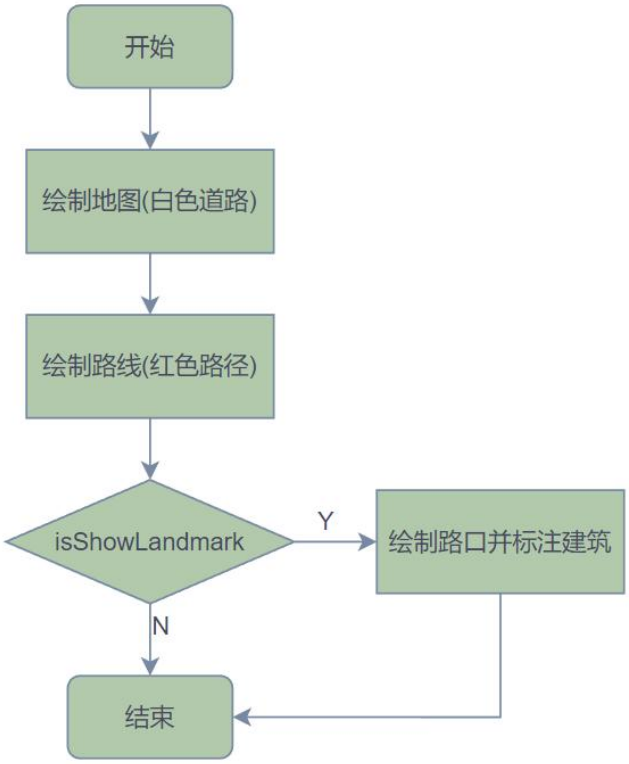
通过使用这个"method"类，可以实现根据给定的出行信息和地图数据进行最优路径规划，并返回规划的路线结果。

5.3 代码实现

5.3.1 optimal_path_planning.cpp

（1）void optimal_path_planning::Draw(void)

绘制函数用于在界面中将地图、规划路线、地标等信息显示出来。由于绘制图层的缘故，所以需要设置特定的绘制过程。如下：先绘制地图，即各个道路。之后绘制红色的路线，这会将一部分白色道路进行覆盖。最后，判断是否需要显示地标，如果需要，则将路口和建筑标注出来。



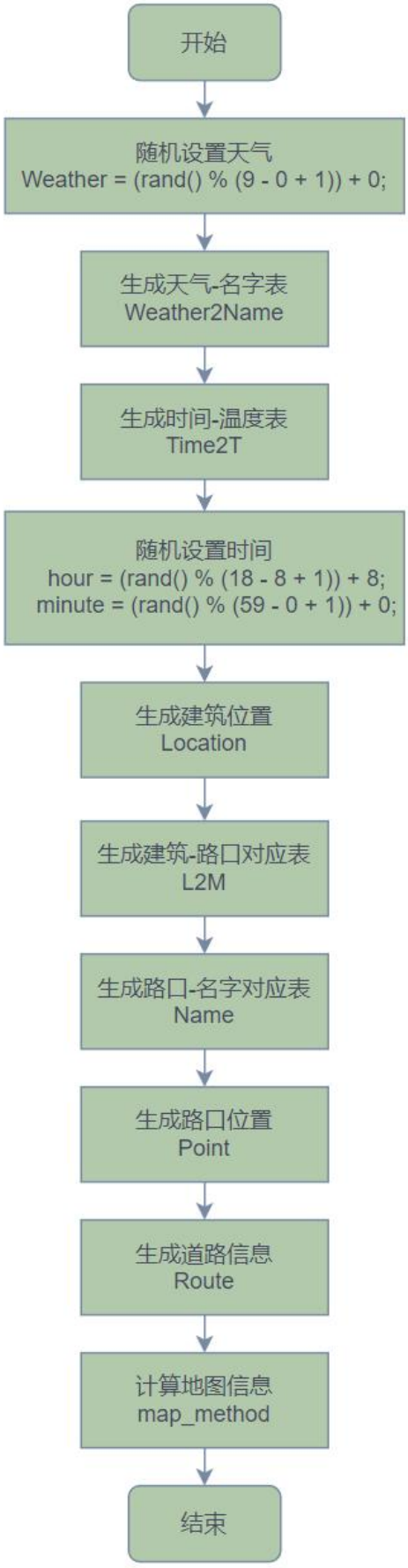
```
//绘制
void optimal_path_planning::Draw(void)
{
    QPixmap pixmap(":/optimal_path_planning/1.jpg");
    QPainter painter(&pixmap);
    if (isShowLandmark) {
        // 绘制地图
        painter.setPen(QPen(Qt::white, 2));
        for (int i = 0; i < Route.size(); i++) {
            painter.drawLine(Point[Route[i].first], Point[Route[i].second]);
        }
    }
    // 绘制路线
    if (!Planning.empty()) {
        painter.setPen(QPen(Qt::red, 2));
        for (auto& v : Planning[mode_Plan]) {
            for (auto it = v.begin(); it < v.end() - 1; ++it) {
                painter.drawLine(Point[*it], Point[*(it+1)]);
            }
        }
    }
    if (isShowLandmark) {
        // 绘制路口
        painter.setPen(QPen(Qt::blue, 4));
        for (int i = 0; i < Point.size(); i++) {
            painter.drawPoint(Point[i]);
        }
        // 标注建筑
        QFont font("Arial", 8);
        painter.setFont(font);
        painter.setPen(Qt::yellow);
        for (int i = 0; i < Location.size(); i++) {
            painter.drawText(Location[i], QString::fromLocal8Bit(Name[L2M[i]]));
        }
    }
    ui.label->setPixmap(pixmap);
}
```

（2）void optimal_path_planning::Init(void)

初始化数据函数中，首先需要随机设置天气，并生成响应的天气-名字表，这一部分用于后续在界面上将天气信息显示出来。之后，系统会生成时间-温度表，紧接着随机设置初始的出行时间，当然时间在后续可以由用户自由更改。

完成上述工作后，系统会依次生成建筑位置、建筑-路口对应表、路口-名字对应表和路口位置。这些数据通常作为静态数据在后台运行，并且需要开发人员在系统开发前进行调查和收集。建筑位置和路口位置可以用坐标表示，以便后续在地图上进行展示和定位。建筑-路口对应表记录了每个建筑物对应的最近的路口，以便在路径规划过程中进行参考。而路口-名字对应表则用于在界面上显示路口的名称，使用户可以更加直观地了解地理位置。

最后，系统会生成道路信息，并将其存储在地图信息中。地图信息通常使用 map 数据结构来存储，其中的键是每条道路，值是每条道路的距离。这样，在后续的路径规划过程中，系统可以根据地图信息进行路线计算和优化。


开始
随机设置天气
Weather = (rand() % (9 - 0 + 1)) + 0;
生成天气-名字表
Weather2Name
生成时间-温度表
Time2T
随机设置时间
hour = (rand() % (18 - 8 + 1)) + 8;
minute = (rand() % (59 - 0 + 1)) + 0;
生成建筑位置
Location
生成建筑-路口对应表
L2M
生成路口-名字对应表
Name
生成路口位置
Point
生成道路信息
Route
计算地图信息
map_method
结束

（3）string optimal_path_planning::GetShow(int mode_Plan)

```
//显示字符串 (-1显示路线, 0-2显示提示信息)
string optimal_path_planning::GetShow(int mode_Plan) {
    if (mode_Plan == -1) {
        string ShowMid = "";
        for (auto it = Way.begin() + 1; it != Way.end() - 1; ++it) {
            ShowMid += (Name[*it] + " ");
        }
        return ShowMid;
    }
    else {
        string Show = "";
        if (mode_Plan == 0)
            Show += "步行规划:";
        else if (mode_Plan == 1)
            Show += "骑行规划:";
        else
            Show += "驾车规划:";
        // 加入路线
        int j = 1;
        for (auto& v : Planning[mode_Plan]) {
            Show += "\n" + to_string(j) + ": ";
            auto it = v.begin();
            for (; it < v.end() - 1; ++it) {
                // 如果有此地点，则加入显示
                int i = 0;
                for (; i < L2M.size(); i++) {
                    if (L2M[i] == *it) {
                        Show += (Name[*it] + "->");
                        break;
                    }
                }
            }
            j += 1;

            Show += Name[*it];
        }
        Show += "\n规划说明:\n";
        if (mode_Plan == 0) {
            Show += "已为您规划距离最短的路线";
        }
        else if (mode_Plan == 1) {
            Show += "在保证距离相对较短的前提下，已为您规划途中含有更多停车点的路线";
        }
        else {
            Show += "在保证距离相对较短的前提下，已为您规划途中含有更多停车点和更少限速路段的路线";
        }
        Show += "\n温馨提示:";
        Show += "\n 气 温 : ";
        if (T <= 0)
            Show += "寒冷天气，出行注意保暖哦！";
        else if (T <= 15)
            Show += "低温天气，一件外套足矣！";
        else if (T <= 25)
            Show += "温度舒适！";
        else if (T <= 35)
            Show += "高温天气，出行注意遮阳！";
        else
            Show += "炎热天气，尽量减少出行哦！";
        Show += "\n 天 气 : ";
        if (Weather == 1)
            Show += "晴朗天空，出行畅通！";
        else if (Weather == 2 || Weather == 3)
            Show += "中到大雨，注意脚下湿滑，记得带伞！";
        else if (Weather == 4)
            Show += "小风拂面！";
        else if (Weather == 5)
            Show += "刮大风啦，骑行谨慎！";
        else if (Weather == 6)
            Show += "下小雪啦，可以欣赏雪景，但也要注意脚下安全哦！";
        else if (Weather == 7 || Weather == 8)
            Show += "中到大雪，注意路面积雪，谨防摔倒或车辆滑行！";
        else
            Show += "多云天气，出行舒适！";
        return Show;
    }
}
```

该代码是`optimal_path_planning`类中的一个成员函数，函数名为`GetShow`，返回类型为`string`。函数根据传入的参数`mode_Plan`来生成路径规划的展示信息，并将结果以字符串的形式返回。

首先，函数会检查`mode_Plan`的值是否为-1。如果是-1，表示当前不是进行路径规划，而是中途添加了地点。在这种情况下，函数会遍历`Way`列表中的元素（除去起点和终点），将对应路口的名称添加到字符串`ShowMid`中，并用空格进行分隔。最后，返回`ShowMid`作为展示信息。

如果`mode_Plan`的值不是-1，表示正在进行路径规划，那么函数会根据不同的规划模式生成相应的展示信息。首先，根据`mode_Plan`的值判断是步行规划、骑行规划还是驾车规划，并将相应的文本添加到字符串`Show`中。

然后，函数会遍历`Planning`列表中指定模式的元素，其中每个元素代表一条路径规划的结果。对于每条规划路径，函数会生成一个编号，并将其添加到`Show`中，接着遍历路径中的每个路口。如果该路口在`L2M`（建筑-路口对应表）中存在，则将路口的名称添加到`Show`中，并在路口之间加入"->"的连接符。最后，将当前路径的规划结果添加到`Show`中。

接下来，函数添加一些规划说明的文本，根据`mode_Plan`的值不同，显示不同的规划说明信息。然后，根据`T`（气温）的值选择相应的温馨提示文本，将其添加到`Show`中。接着，根据`Weather`（天气）的值选择相应的天气提示文本，将其添加到`Show`中。

最后，函数返回生成的展示信息字符串`Show`，供界面显示和用户查看。

（4）void optimal_path_planning::mousePressEvent(QMouseEvent* event)

```
void optimal_path_planning::mousePressEvent(QMouseEvent* event)
{

    QPoint pos = event->pos();  // 获取鼠标点击的位置
    if (pos.x() <= 817 && pos.y() <= 435) {
        int index = -1;
        int dis = 0;
        for (int i = 0; i < Location.size(); i++) {
            int deltaX = pos.x() - Location[i].x();
            int deltaY = pos.y() - Location[i].y();
            int distance = round(sqrt(deltaX * deltaX + deltaY * deltaY));

            if (index == -1) {
                index = i;
                dis = distance;
                continue;
            }

            if (distance < dis) {
                index = i;
                dis = distance;
            }
        }
        if (mode == 0) {
            Way.front() = L2M[index];
            ui.label_7->setText(QString::fromLocal8Bit(Name[Way.front()]));
        }
        else if(mode == 1) {
            Way.back() = L2M[index];
            ui.label_8->setText(QString::fromLocal8Bit(Name[Way.back()]));
        }
        else {
            if (Way[Way.size() - 2] != L2M[index]) {
                Way.insert(Way.end() - 1, L2M[index]);
                ui.textBrowser_4->setText(QString::fromLocal8Bit(GetShow(-1)));
            }
        }
    }
}
```

该代码是一个鼠标点击事件的处理函数，用于处理用户在界面上点击的操作。

当用户点击鼠标时，函数首先获取鼠标点击的位置坐标，保存在变量`pos`中。

接下来，函数会判断点击位置的坐标是否在一个特定的区域范围内（判断条件为`pos.x() <= 817 && pos.y() <= 435`）。如果在该区域内，表示用户点击的是地图上的某个位置。

然后，函数会遍历保存建筑位置信息的`Location`列表，计算点击位置与每个建筑位置之间的距离，并找到距离最近的建筑位置。遍历过程中，计算距离使用了欧几里得距离公式。

找到最近的建筑位置后，根据当前的模式`mode`进行不同的处理。如果`mode`为 0，表示当前是设置起点的模式，则将起点设置为最近的建筑位置，并更新界面上对应的文本显示。如果`mode`为 1，表示当前是设置终点的模式，则将终点设置为最近的建筑位置，并更新界面上对应的文本显示。如果`mode`不为 0 或 1，表示当前是添加中途地点的模式，会在路径中插入最近的建筑位置，并更新界面上的文本显示。

最后，如果在特定区域外点击，函数不执行任何操作，代码结束。

（5）void optimal_path_planning::Adjust(void)

```
void optimal_path_planning::Adjust(void)
{
    mode = (mode + 1) % 3;
    if (mode == 0) {
        ui.label_12->setText(QString::fromLocal8Bit("起始位置"));
    }
    else if (mode == 1) {
        ui.label_12->setText(QString::fromLocal8Bit("目标位置"));
    }
    else {
        ui.label_12->setText(QString::fromLocal8Bit("中途位置"));
    }
}
```

该代码是一个函数，用于调整路径规划的模式。

函数首先对变量`mode`进行修改，将其值增加 1 并取余 3，以实现循环切换模式的效果。`mode`表示当前的路径规划模式，可以取 0、1、2 三个值，分别表示起始位置模式、目标位置模式和中途位置模式。

接下来，根据`mode`的值对界面上的文本进行更新。如果`mode`为 0，表示当前是起始位置模式，则界面上的文本显示为"起始位置"。如果`mode`为 1，表示当前是目标位置模式，则界面上的文本显示为"目标位置"。如果`mode`为 2，表示当前是中途位置模式，则界面上的文本显示为"中途位置"。

通过调用该函数，可以循环切换路径规划的模式，以便用户可以在界面上选择设置起始位置、目标位置或中途位置。

（6）void optimal_path_planning::Delete(void)

```
//删除中途地点
void optimal_path_planning::Delete(void)
{
    if (Way.size() > 2) {
        Way.erase(Way.end() - 2);
        ui.textBrowser_4->setText(QString::fromLocal8Bit(GetShow(-1)));
    }
}
```

该代码是一个函数，用于删除路径规划中的中途位置。

函数首先检查路径的长度是否大于 2，这是为了确保路径中至少存在起始位置和目标位置。如果路径长度大于 2，则执行删除操作。

删除操作使用`erase`函数从`Way`向量中删除倒数第二个元素，即删除倒数第二个中途位置。

接下来，通过调用`GetShow`函数获取更新后的路径信息，并将其转换为`QString`类型，然后将其设置为界面上的`textBrowser_4`文本框的显示内容。

通过调用该函数，可以删除路径规划中的中途位置，并更新界面上的显示。

（7）void optimal_path_planning::ShowLandmark(void)

```
//显示地标
void optimal_path_planning::ShowLandmark(void)
{
    isShowLandmark = !isShowLandmark;
    if (isShowLandmark) {
        ui.pushButton_3->setText(QString::fromLocal8Bit(" 关 闭 "));
    }
    else {
        ui.pushButton_3->setText(QString::fromLocal8Bit(" 开 启 "));
    }
    Draw();
}
```

该代码是一个函数，用于显示或隐藏地标。

函数首先通过取反操作改变`isShowLandmark`变量的值，用于切换显示或隐藏地标的状态。如果`isShowLandmark`为`true`，则表示当前为显示状态，否则为隐藏状态。

接下来，根据`isShowLandmark`的值，更新界面上按钮`pushButton_3`的文本。如果`isShowLandmark`为`true`，将按钮文本设置为"关闭"；如果`isShowLandmark`为`false`，将按钮文本设置为"开启"。

最后，调用`Draw`函数，根据当前的显示状态重新绘制界面，以显示或隐藏地标。

通过调用该函数，可以切换地标的显示或隐藏状态，并更新界面上的按钮文本和图形显示。

（8）void optimal_path_planning::updateWT(void)

```
//更新气候
void optimal_path_planning::updateWT(void)
{
    hour = ui.spinBox->value();
    minute = ui.spinBox_2->value();
    T = Time2T[hour];
    ui.label_9->setText(QString::fromLocal8Bit(Weather2Name[Weather] + to_string(T) + "度 "));
}
```

该代码是一个函数，用于更新时间和天气信息。

函数首先从界面上获取用户设置的小时和分钟，分别存储在`hour`和`minute`变量中。

接下来，根据`hour`的值，从`Time2T`映射表中获取对应的气温值，并将其存储在变量`T`中。

然后，通过拼接字符串的方式，将当前的天气名称、气温和单位（度）信息组合成一个字符串，并设置给界面上的`label_9`标签，用于显示天气和气温信息。

通过调用该函数，可以根据用户设置的小时和分钟更新时间信息，并根据映射表获取对应的气温值，最后更新界面上的天气和气温显示。

（9）void optimal_path_planning::affirm(void)

```
//确认（开始规划）
void optimal_path_planning::affirm(void)
{
    if (Way.front() >= 0 && Way.back() >= 0) {
        Planning.clear();
        method Q = method(hour, minute, Way, Weather, T, map_method);
        Q.route_Planning();
        Planning = Q.get_Planning();
        Draw();
        mode_Plan = 0;
        ui.textBrowser->setText(QString::fromLocal8Bit(GetShow(mode_Plan)));
    }
}
```

该代码是一个函数，用于确认用户输入的起始位置和目标位置，并进行路径规划。

首先，函数检查起始位置和目标位置是否有效，即判断`Way.front()`和`Way.back()`是否大于等于 0。如果两者都大于等于 0，表示起始位置和目标位置已经设置。

然后，函数执行以下操作：

1. 清空`Planning`，即清空之前的路径规划结果。

2. 创建一个`method`对象`Q`，并使用用户设置的小时、分钟、起始位置、目标位置、天气和气温信息进行初始化。

3. 调用`Q.route_Planning()`方法进行路径规划。

4. 获取路径规划的结果，将其存储在`Planning`中。

5. 调用`Draw()`方法更新界面上的地图显示，显示新的路径规划结果。

6. 将`mode_Plan`设置为 0，表示显示步行规划的结果。

7. 将路径规划结果显示在界面上的`textBrowser`中，通过调用`GetShow(mode_Plan)`函数获取要显示的文本信息，并将其设置给`textBrowser`。

通过调用该函数，可以确认起始位置和目标位置，并进行路径规划，最后更新地图和显示规划结果的文本框。

（10）void optimal_path_planning::change(void)

```
//更改显示规划
void optimal_path_planning::change(void)
{
    if (!Planning.empty()) {
        mode_Plan = (mode_Plan + 1) % 3;
        ui.textBrowser->setText(QString::fromLocal8Bit(GetShow(mode_Plan)));
        Draw();
    }
}
```

该代码是一个函数，用于切换显示路径规划结果的模式。

首先，函数检查`Planning`是否为空，即判断是否已经进行了路径规划。如果`Planning`不为空，表示已经进行了路径规划。

然后，函数执行以下操作：

1. 将`mode_Plan`的值加 1 并取余 3，实现循环切换模式。`mode_Plan`的值表示当前显示的路径规划模式，取值范围为 0、1、2。

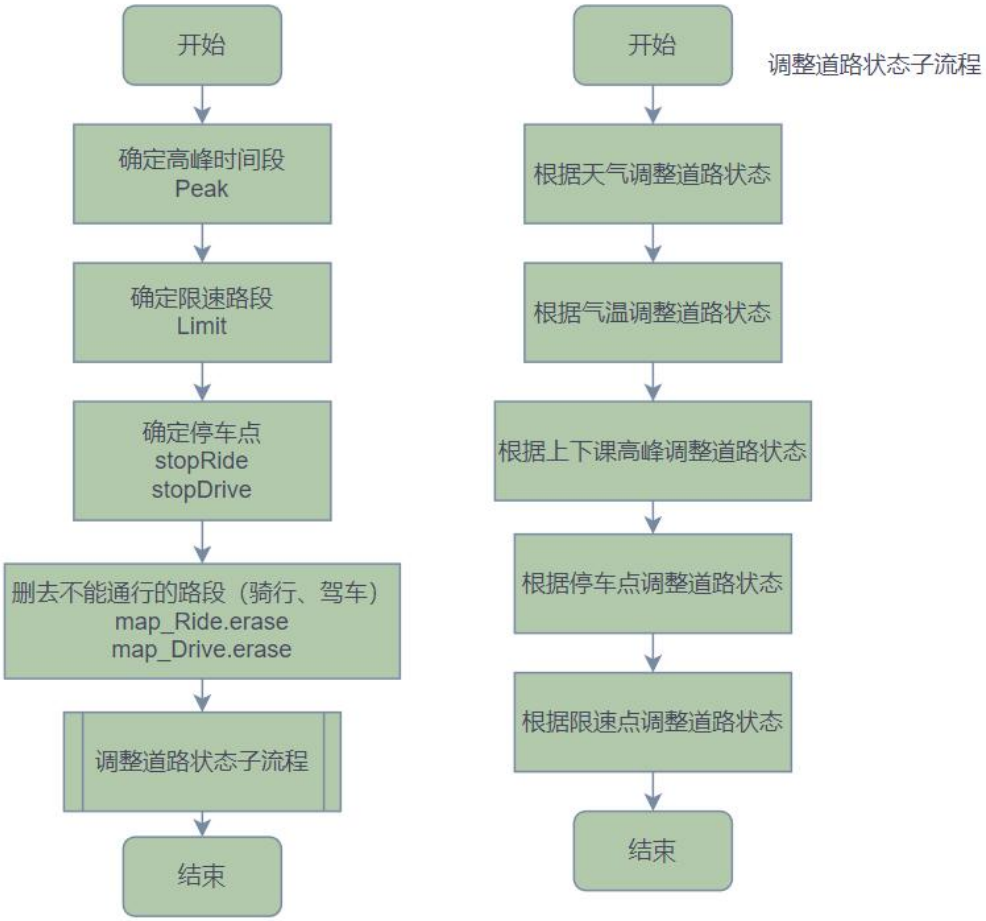
2. 将切换后的模式对应的文本信息通过调用`GetShow(mode_Plan)`函数获取，并将其设置给界面上的`textBrowser`，以更新显示的文本信息。

3. 调用`Draw()`方法更新界面上的地图显示，以显示新的路径规划结果。

通过调用该函数，可以循环切换显示路径规划结果的模式，包括步行规划、骑行规划和驾车规划。

5.3.2　method.cpp

（1）void method::init(void)



在初始化函数中，系统首先明确了一些基本信息，如高峰时间段、限速路段和停车点等。此外，系统还会直接删除一些骑行和驾车不能通行的道路。一旦所有基本信息都明确好了，系统会调用道路状态子程序来对步行、骑行和驾车三个地图的道路状态进行调整。

在调整道路状态子程序中，系统会根据权重因子（具体细节请参考 3.2 模块设计中的数据初始化模块）依次从天气、气温、上下课高峰、停车点和限速点这五个方面更新道路的权重。通过考虑这些因素，系统能够动态调整道路的通行能力，以反映实际情况。最终，系统形成了一份用于路径规划的出行地图。

这样的初始化过程确保了路径规划时考虑了多个因素的影响，如天气、温度、交通状况等，从而使路径规划更加准确和实用。通过权重的调整，系统能够根据实时的情况为用户提供最优的路径规划方案。

（2）void method::Dijkstra(int Start, int End, map<pair<int, int>, int>& map_method, vector<int>&ans, int& flag)

```
mutex mutex1;
void method::Dijkstra(int Start, int End, map<pair<int, int>, int>& map_method, vector<int>&ans, int& flag)
{
    vector<int> distance(106, INT_MAX); //距离
    vector<int> previous(106, -1);      //前驱节点
    // 最小堆处理顶点
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;
    // 将起点加入优先队列，距离设置为0
    pq.push(make_pair(0, Start));
    distance[0] = 0;
    while (!pq.empty()) {
        int current = pq.top().second;
        pq.pop();
        // 遍历当前顶点的邻居
        for (int i = 0; i < 106; ++i) {
            if (i == current) {
                continue;
            }
            int newDistance = -1;
            if (map_method.find(make_pair(current, i)) != map_method.end()) {
                newDistance = distance[current] + map_method[make_pair(current, i)];
            }
            else if (map_method.find(make_pair(i, current)) != map_method.end()) {
                newDistance = distance[current] + map_method[make_pair(i, current)];
            }
            else {
                continue;
            }
            // 如果找到了更短的路径，则更新距离和前驱节点
            if (newDistance < distance[i]) {
                distance[i] = newDistance;
                previous[i] = current;
                pq.push(make_pair(newDistance, i));
            }
        }
    }
    // 根据前驱节点数组构建最短路径
    int current = End;
    ans.clear();
    while (current != Start) {
        ans.push_back(current);
        current = previous[current];
    }
    ans.push_back(Start);
    // 将最短路径反转，使其按照起点到终点的顺序
    reverse(ans.begin(), ans.end());
    mutex1.lock();
    flag += 1;
    mutex1.unlock();
}
```

这段代码实现了 Dijkstra 算法用于最短路径的计算。代码解释如下：

mutex mutex1：声明了一个互斥量，用于实现线程安全的访问和操作。

void method::Dijkstra(int Start, int End, map<pair<int, int>, int>& map_method, vector<int>&ans, int& flag)：定义了一个名为 Dijkstra 的函数，它接受起点和终点的索引、存储地图信息的 map、存储最短路径的向量和一个标志变量作为参数。

vector<int> distance(106, INT_MAX)：创建一个长度为 106 的整型向量，用于存储起点到每个顶点的距离，默认初始化为无穷大。

vector<int> previous(106, -1)：创建一个长度为 106 的整型向量，用于存储每个顶点的前驱节点，默认初始化为-1。

priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq：创建一个优先队列，用于按照距离的升序处理顶点。

pq.push(make_pair(0, Start))：将起点加入优先队列，距离设置为 0，表示起点到自身的距离为 0。

while (!pq.empty()) { ... }：使用 while 循环，直到优先队列为空，即遍历完所有顶点。

int current = pq.top().second：获取当前队列中距离最小的顶点。

pq.pop()：弹出队列中距离最小的顶点。

for (int i = 0; i < 106; ++i) { ... }：遍历当前顶点的邻居。

if (map_method.find(make_pair(current, i)) != map_method.end()) { ... }：判断当前顶点与邻居之间是否存在道路连接。

newDistance = distance[current] + map_method[make_pair(current, i)]：计算通过当前顶点到达邻居的新距离。

if (newDistance < distance[i]) { ... }：如果找到了更短的路径，则更新距离和前驱节点。

int current = End：将终点作为当前节点。

while (current != Start) { ... }：根据前驱节点数组构建最短路径。

ans.push_back(Start)：将起点添加到最短路径中。

reverse(ans.begin(), ans.end())：将最短路径反转，使其按照起点到终点的顺序。

mutex1.lock()和 mutex1.unlock()：在更新共享变量`flag`之前加锁，确保线程安全。

以下是代码的实现逻辑：

1. 首先，创建了两个向量`distance`和`previous`，用于存储每个顶点的距离和前驱节点。

2. 创建了一个优先队列`pq`，用于按照距离的升序处理顶点。

3. 将起点加入优先队列，并将其距离设置为 0。

4. 进入循环，直到优先队列为空：

从优先队列中取出距离最小的顶点作为当前顶点。

遍历当前顶点的邻居：

如果当前顶点与邻居之间存在道路连接：

计算通过当前顶点到达邻居的新距离。

如果新距离小于当前存储的距离，则更新距离和前驱节点，并将邻居加入优先队列。

5. 根据前驱节点数组构建最短路径：

从终点开始，依次回溯前驱节点，将每个顶点添加到最短路径的向量`ans`中。

将最短路径反转，使其按照起点到终点的顺序。

6. 最后，对共享变量`flag`进行加锁，增加其值，然后释放锁。

总体而言，该代码通过使用 Dijkstra 算法，逐步更新顶点之间的距离和前驱节点，从而找到起点到终点的最短路径。它使用优先队列来选择距离最小的顶点，并通过循环遍历邻居顶点来更新距离和前驱节点。最终，它构建出一条从起点到终点的最短路径，并将其存储在`ans`向量中。同时，为了保证线程安全，使用互斥量对共享变量进行加锁和解锁操作。

（3）void method::init(void)

```
void method::createThread(method* W)
{
    vector<vector<int> >wayWalk(W->tit.Way.size() - 1);
    vector<vector<int> >wayRide(W->tit.Way.size() - 1);
    vector<vector<int> >wayDrive(W->tit.Way.size() - 1);
    W->tit.flagWalk = W->tit.flagRide = W->tit.flagDrive = 0;
    for (int i = 0; i < W->tit.Way.size() - 1; i++) {
        thread t1(W->Dijkstra, W->tit.Way[i], W->tit.Way[i + 1], ref(W->tit.map_Walk), ref(wayWalk[i]), ref(W->tit.flagWalk));
        t1.detach();
        thread t2(W->Dijkstra, W->tit.Way[i], W->tit.Way[i + 1], ref(W->tit.map_Ride), ref(wayRide[i]), ref(W->tit.flagRide));
        t2.detach();
        thread t3(W->Dijkstra, W->tit.Way[i], W->tit.Way[i + 1], ref(W->tit.map_Drive), ref(wayDrive[i]), ref(W->tit.flagDrive));
        t3.detach();
    }
    W->tit.Planning.clear();
    W->tit.Planning.resize(3);

    while (W->tit.flagWalk < W->tit.Way.size() - 1);
    W->tit.Planning[0] = wayWalk;
    while (W->tit.flagRide < W->tit.Way.size() - 1);
    W->tit.Planning[1] = wayRide;
    while (W->tit.flagDrive < W->tit.Way.size() - 1);
    W->tit.Planning[2] = wayDrive;
}
```

这段代码实现了一个创建线程的函数，用于并行地调用`Dijkstra`方法来计算不同出行方式下的最短路径。以下是代码的实现逻辑：

1. 首先，创建了三个二维向量`wayWalk`、`wayRide`和`wayDrive`，用于存储步行、骑行和驾车的最短路径。

2. 将步行、骑行和驾车的计数器`flagWalk`、`flagRide`和`flagDrive`设置为 0，用于在后

续判断计算是否完成。

3. 使用循环遍历`W->tit.Way`的前 N-1 个元素（N 为`W->tit.Way`的长度），每次迭代都创建三个线程：

第一个线程使用`W->Dijkstra`方法计算从`W->tit.Way[i]`到`W->tit.Way[i + 1]`的步行最短路径，并将结果存储在`wayWalk[i]`中。

第二个线程使用`W->Dijkstra`方法计算从`W->tit.Way[i]`到`W->tit.Way[i + 1]`的骑行最短路径，并将结果存储在`wayRide[i]`中。

第三个线程使用`W->Dijkstra`方法计算从`W->tit.Way[i]`到`W->tit.Way[i + 1]`的驾车最短路径，并将结果存储在`wayDrive[i]`中。

每个线程使用`detach()`将其设置为分离线程，使其在后台运行。

4. 清空`W->tit.Planning`并将其大小设置为 3，以便存储步行、骑行和驾车的规划路径。

5. 使用循环和条件判断等待步行、骑行和驾车的计算完成：

在循环中，通过比较计数器`flagWalk`、`flagRide`和`flagDrive`与`W->tit.Way`的大小，判断计算是否完成。

等待步行计算完成后，将步行的最短路径存储在`W->tit.Planning[0]`中。

等待骑行计算完成后，将骑行的最短路径存储在`W->tit.Planning[1]`中。

等待驾车计算完成后，将驾车的最短路径存储在`W->tit.Planning[2]`中。

总体而言，该代码使用线程并行地调用`Dijkstra`方法来计算不同出行方式下的最短路径。它创建多个线程来同时计算步行、骑行和驾车的最短路径，并在计算完成后将结果存储在相应的向量中。最后，它将最短路径存储在`W->tit.Planning`中，以供后续使用。

需要注意的是，使用`detach()`将线程设置为分离线程，意味着主线程不会等待这些线程的完成。因此，在等待计算完成时，使用了一个循环和条件判断的方式，持续检查计数器是否达到预期值，以确定计算是否完成。一旦计算完成，将最短路径存储在相应的位置。

这种并行计算的方式可以提高计算效率，因为不同出行方式的最短路径可以同时计算，而不需要按顺序逐个计算。这在处理大规模地图或复杂的路径规划时尤为重要，可以减少计算时间并提升系统的响应速度。

（4）void method::route_Planning(void)

```
void method::route_Planning(void)
{
    thread t(createThread, this);
    t.join();
}
```

在`route_Planning`函数中，首先创建了一个线程`t`，并将当前对象`this`作为参数传递给`createThread`函数。然后使用`t.join()`语句，主线程会等待线程`t`的执行完成。

`createThread`函数是一个在上文中已经解释过的函数，用于并行计算步行、骑行和驾车的最短路径。通过创建线程并在后台执行最短路径计算，可以提高计算效率。

因此，`route_Planning`函数的目的是通过调用`createThread`函数来实现并行计算最短路径，并在主线程中等待计算完成。这样可以确保在执行后续操作之前，最短路径已经计算完成并存储在相应的数据结构中。

（5）vector<vector<vector<int>>> method::get_Planning(void)

```
vector<vector<vector<int>>> method::get_Planning(void)
{
    return tit.Planning;
}
```

这段代码定义了一个成员函数`get_Planning`，它返回一个`vector<vector<vector<int>>>`类型的值。

函数内部直接通过`return`语句返回了`tit.Planning`，`tit`是当前对象的一个成员变量，它的类型是 `struct Tit`。而`tit.Planning`是 `Tit` 结构体中的一个成员变量，它是一个三维向量。

所以，`get_Planning`函数的作用是获取当前对象的`tit.Planning`数据，并将其作为结果返回。这样其他代码可以调用该函数来获取路径规划的结果，以便进行后续操作。

（6）vector<vector<vector<int>>> method::get_Planning(void)

```
bool method::get_flag(void)
{
    return ((tit.flagWalk == tit.Way.size() - 1) && (tit.flagRide == tit.Way.size() - 1) && (tit.flagDrive == tit.Way.size() - 1));
}
```

这段代码定义了一个成员函数 `get_flag`，它返回一个布尔值。

函数内部使用逻辑与运算符 `&&` 来组合多个条件表达式，判断三个条件是否同时成立。这三个条件分别是：

1. `tit.flagWalk == tit.Way.size() - 1`：判断 `tit` 结构体中的 `flagWalk` 是否等于 `Way` 的大小减去 1。

2. `tit.flagRide == tit.Way.size() - 1`：判断 `tit` 结构体中的 `flagRide` 是否等于 `Way` 的大小减去 1。

3. `tit.flagDrive == tit.Way.size() - 1`：判断 `tit` 结构体中的 `flagDrive` 是否等于 `Way` 的大小减去 1。

如果这三个条件同时成立，即表示 `flagWalk`、`flagRide` 和 `flagDrive` 都已经达到了 `Way` 大小减去 1 的值，那么函数返回 `true`，否则返回 `false`。

因此，`get_flag` 函数的作用是判断当前对象的三个标志值是否全部达到指定的条件，以便其他代码可以通过调用该函数来检查是否完成了路径规划的所有操作。

6 软件介绍

6.1 界面介绍

6.1.1 初始界面

初始界面是用户打开系统时的界面，为用户提供方便的操作和设置选项。系统已随机生成时间和温度，并展示在界面上。用户可以调整时间，温度将根据时间变化，变化内容根据时间-温度表 Time2T 确定。

界面主要包含五个按键，分别是地标显示设置、位置调整设置、删除中途点、确认出行和更改规划。它们的功能如下：

1. 关闭/开启地标：该按键允许用户选择是否显示地标。点击关闭地标时，地图上的道路和建筑信息会隐藏，展示最基本的卫星地图界面；点击开启地标时，道路以白色显示，建筑以黄色文字标注。这样的功能设计为用户提供了个性化的地图显示选择，根据需要可以切换地标的显示与隐藏，以满足用户不同的使用场景和偏好。

2. 位置调整设置：该按键用于调整起始位置、目标位置和中途位置。界面上会显示当前设置的地点类型，用户可以点击调整设置来更改地点类型。通过在地图上点击，用户可以设置位置，系统会计算最近的建筑并保存在出行路线中。这个功能为用户提供了灵活的位置调整能力，可以根据具体需求随时更改起始位置、目标位置或添加/删除中途位置，以满足个性化的出行需求。

3. 删除中途点：该按键用于删除中途位置。每次点击该按键，系统将删除最后一个中途点，如果没有中途点，则按键失效。这个功能为用户提供了撤销错误或不需要的中途点的便利，避免了重新设置整个出行路线的麻烦。用户可以根据需要灵活地删除中途点，优化出行方案。

4. 确认出行：点击该按键，系统会检查是否设置了起始位置和目标位置。如果位置不完整，则按键无效；如果位置完整，系统会调用规划模块开始路径规划，并将规划好的路线显示在界面上供用户参考。这个功能为用户提供了出行方案的确认操作，确保了起始位置和目标位置的正确性，并根据用户的需求生成最佳的出行路线，提高出行效率和舒适度。

5. 更改规划：系统界面一次只显示一种路径的相关信息。点击更改规划按键，系统将按顺序切换显示步行、骑行和驾车的路径信息。这个功能为用户提供了对不同出行方式路径的比较和选择的能力。用户可以通过轻松点击按钮切换不同的规划结果，以便对比各种出行方式的路线选择和相关信息。这样的设计使用户能够直观地了解不同出行方式的优势和劣势，便于根据实际需求做出最佳的决策。

需要注意的是，点击更改规划按键时，系统会先检查是否已经规划好了路径信息，即确保存在有效的路径规划结果。如果尚未规划路径，则按键无效，避免了用户在没有有效信息的情况下进行不必要的操作。

综上所述，初始界面为用户提供了基本的操作和设置选项，以便用户根据个人需求进行出行规划。通过关闭/开启地标功能，用户可以根据实际需要选择地图上的显示内容；通过位置调整设置，用户可以灵活调整起始位置、目标位置和中途位置；删除中途点功能方便用户撤销冗余中途点；确认出行按键确保出行路线完整并进行路径规划；更改规划按键允许用户对不同出行方式的路线进行比较和选择。

这些功能的设计旨在提供用户友好的操作界面和个性化的出行体验。用户可以根据自身需求和偏好进行设置和调整，从而得到最优的出行方案。系统通过简洁明了的界面和直观的按键设计，提供了便捷的出行规划和路径选择功能，为用户提供了高效、准确和舒适的出行

体验。

6.1.2 规划界面

这是一条从北体到槐园的路线规划示例，途中经过了快递中心和文典阁两个地点。通过我们的规划路线系统，用户可以方便地查看并参考这条路线规划。

在地图上，我们使用红色标注来表示规划好的路线，与地图上的白色道路形成鲜明对比，使用户能够清晰地辨识出所选择的路径。

在规划信息中，系统首先会告知用户当前的规划是哪种出行方式，图中显示的是步行规划。接着，系统会按照用户指定的出行路线，逐段展示最优路径。在这个示例中，我们展示了三段路径，第一段是从起始位置北体到中途位置 1 快递中心的路径，第二段是从中途位置 1 快递中心到中途位置 2 文殿阁的路径，第三段是从中途位置 2 文典阁到目标位置槐园的路径。

当最优路线显示完毕后，系统会提供当前规划的说明。在图中的步行规划示例中，系统直接选择了距离最短的路线作为最优路线。这样的说明有助于用户了解系统的规划策略和选

择依据。

最后，根据当前的天气和气候条件，系统会给出出行建议，即温馨提示。例如，如果当前的天气是多云，气温为 11 度，系统可能会提醒用户“低温天气，一件外套足矣！”或者“多云天气，出行舒适！”。这样的提示有助于用户在出行前做好相应的准备，保证出行的舒适和安全。

通过这样的路线规划示例，用户可以清楚地了解到系统提供的路线规划功能和相关信息展示。用户可以根据自身需求和条件，选择合适的出行方式和最优路线，提前了解天气和气候情况，为出行做好准备。这样的路线规划系统旨在为用户提供便利、快捷且个性化的出行体验，使其能够轻松规划并享受到最佳的出行方案。

（1）骑行规划如下：

在骑行规划中，规划偏好是基于寻找距离相对较短的路线的前提下，同时侧重于规划途中包含更多停车点的考虑。这意味着系统会倾向于选择那些在骑行过程中提供更多停车机会和便利的道路，以满足骑行者对停车的需求。通过优先考虑停车点密度较高的路径，骑行规

划可以提供更多停车选择，方便骑行者在需要时进行停车、休息或处理其他事务。

（2）驾车规划如下：

在驾车规划中，规划偏好是基于寻找距离相对较短的路线的前提下，同时侧重于规划途中包含更多停车点和较少限速路段的考虑。这意味着系统会倾向于选择那些在驾车过程中提供更多停车机会和减少限速路段的道路，以提供更便利和顺畅的驾车体验。

6.1.3 关闭地标

关闭地标功能后，系统会隐藏地图上的道路信息和建筑物信息，以提供用户一个更为简洁的卫星地图界面。关闭地标不会对系统的其他功能和性能产生任何影响，用户可以在不受干扰的情况下流畅地进行操作和浏览。

关闭地标功能并不会影响其他系统功能的可用性，用户仍然可以使用系统的其他按键和设置选项。例如，用户可以继续调整位置设置、添加或删除中途点、确认出行和更改规划。只是在关闭地标状态下，地图上不再显示道路的具体走向和建筑物的位置，这可能会对一些用户提供更清爽和沉浸式的使用体验。

起始位置: 请点击地图　中途位置:　出行气候: 微风，8度

目标位置: 请点击地图　更改规划

出行时间: 16 时 16 分

当前设置: 起始位置

开 启 地 标　调整设置

删除中途点　确认出行

起始位置: 北体　中途位置: 快递中心 文典阁　出行气候: 多云，11度

目标位置: 槐园　更改规划

出行时间: 10 时 32 分

当前设置: 中途位置

开 启 地 标　调整设置

删除中途点　确认出行

驾车规划:
1: 北体->快递中心
2: 快递中心->桔园食堂->桔园->桃园->文典阁
3: 文典阁->竹园->槐园
规划说明:
在保证距离相对较短的前提下，已为您规划途中含有更多停车点和更少限速路段的路线
温馨提示:

6.2 软件主要特点

（1）用户界面良好

该软件注重用户体验，具有友好的用户界面设计，使用户能够轻松地理解和操作软件功能。界面布局合理，图标清晰，颜色搭配和谐，提供直观的操作指引和反馈，以提供良好的使用体验。

（2）操作安全性高

无论用户何时点击按钮、输入命令或执行其他操作，软件都能正确地响应，并提供与操作目的相符合的结果。软件的界面和功能都经过了充分测试和验证，确保它们在各种操作环境下都能正常工作。按钮、菜单、输入框等用户界面元素可以按预期工作，不会导致界面冻结、崩溃或产生错误行为。

（3）多线程开发

软件采用多线程开发技术，以提高系统的并发性和响应速度。通过合理地利用多线程，软件可以同时执行多个任务，提高用户的操作效率和响应性。例如，在路径规划功能中，可以同时进行多条路径的计算，加快规划结果的生成。

（4）简易流畅

软件注重简易性和流畅性，致力于提供简化的操作流程和流畅的用户体验。通过简化操作步骤、提供直观的界面布局和交互方式，软件使用户能够快速上手并顺利完成所需任务。同时，软件经过优化和性能调整，确保系统的运行流畅，避免卡顿和延迟，提供快速响应和操作的体验。

7 软件测试

7.1 任务完成度

任务	是否完成
界面友好	完成
应用场景：安徽大学磬苑校区	完成
起讫点、出行时间随机设置	完成
给出步行、骑行和驾车三种出行方式的最优路径、以及缘由（例如天气、气温、停车点、限速点、上下课高峰、出行时间点、以及个性化偏好）	完成

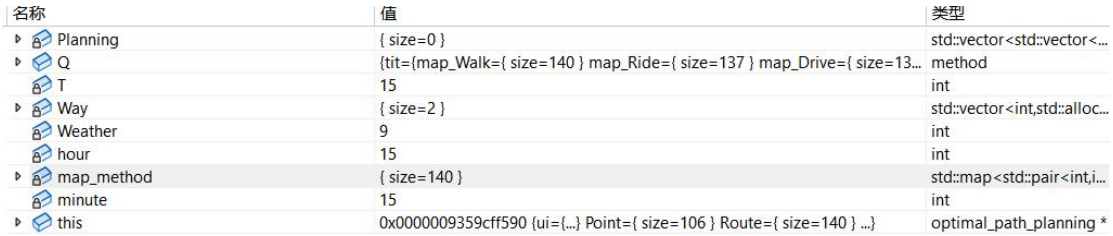
相关文档完整	完成
算法实现过程中开启多个安全线程计算特征嵌入	完成
模型能够在三种出行方式中通用	完成

7.2 各模块测试

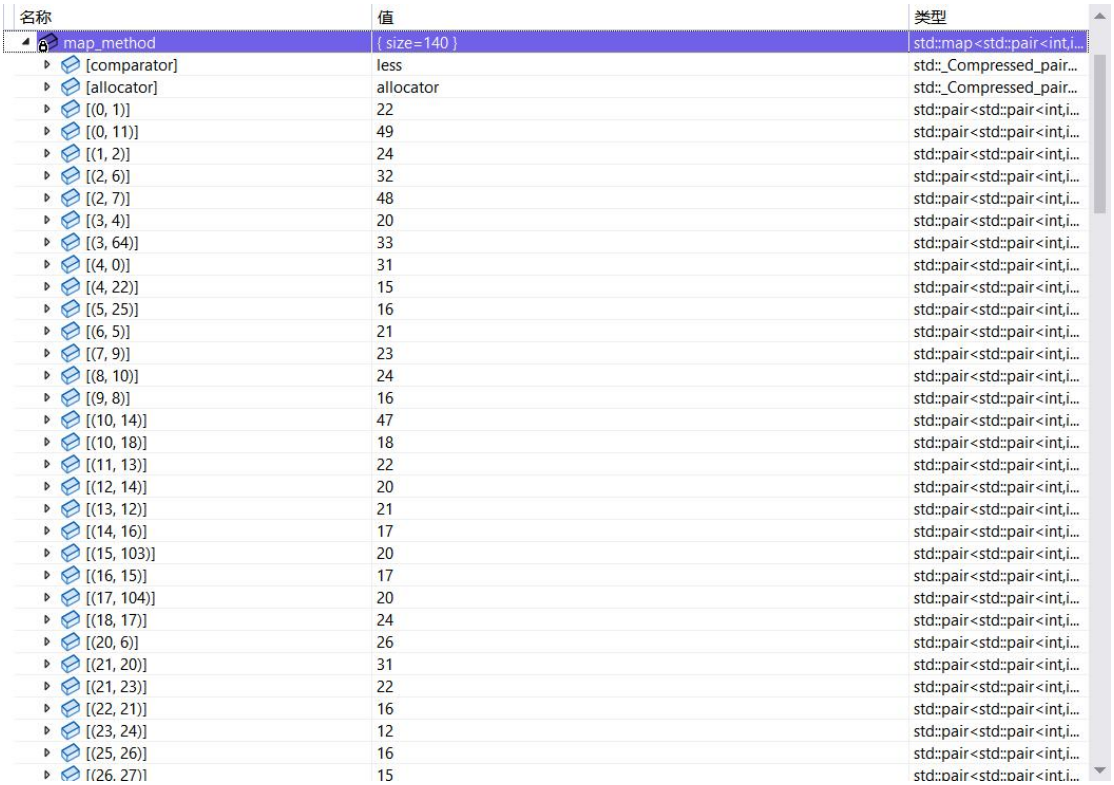
7.2.1 数据初始化模块

测试思路：为了确保系统的稳定性和数据的准确性，我们采用设置断点的方式，在用户确认规划后、系统规划路线之前插入断点。通过这种方式，我们能够检测系统后台数据是否正常，是否存在非法数据的情况。这样可以及时发现潜在的问题，并进行修复，确保系统运行的可靠性和安全性。断点的设置可以帮助开发人员深入了解程序的执行流程，方便调试和优化代码，从而提升系统的性能和用户体验。

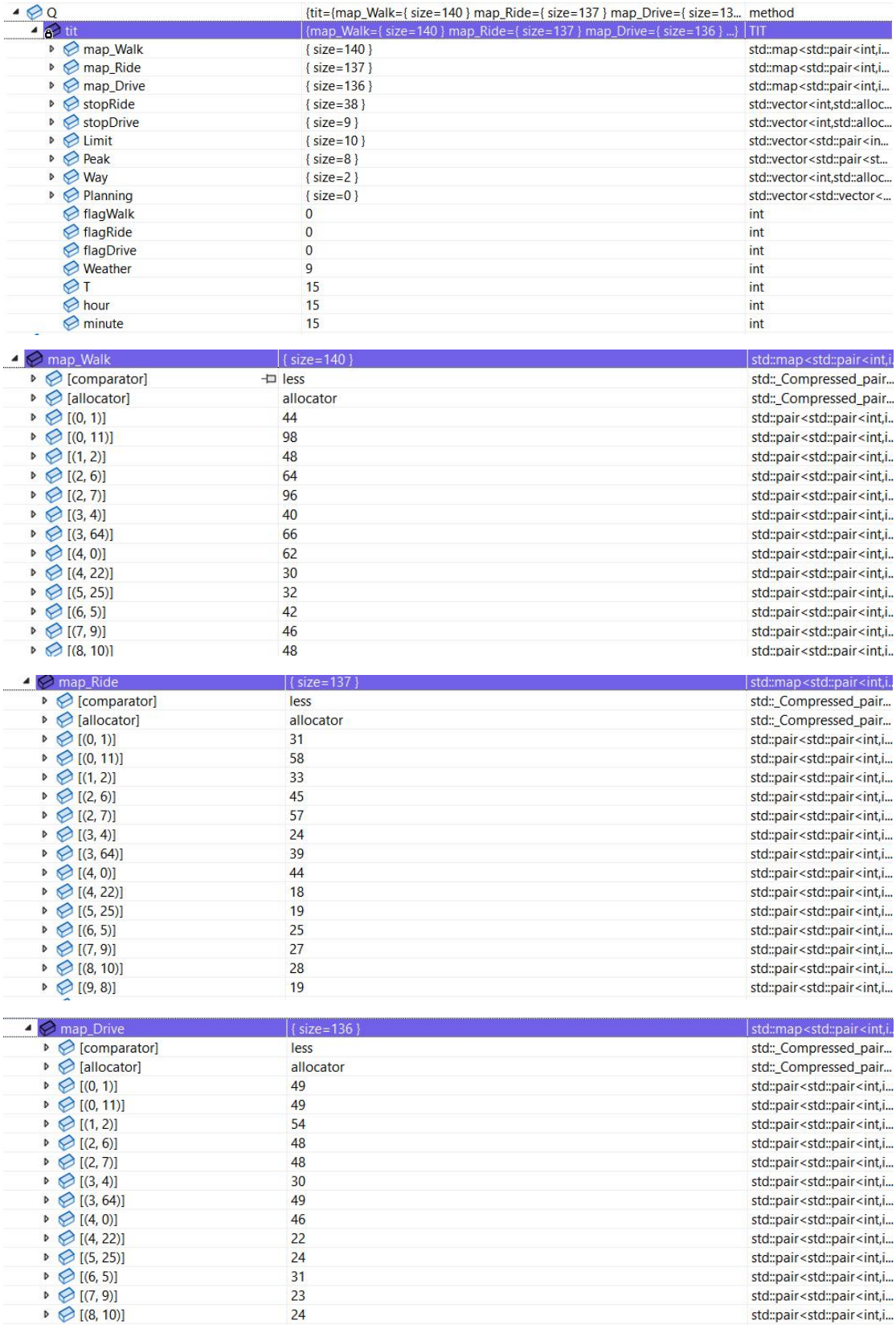
测试界面如下：



名称	值	类型
Planning	{ size=0 }	std::vector<std::vector<...
Q	{tit={map_Walk={ size=140 } map_Ride={ size=137 } map_Drive={ size=13...	method
T	15	int
Way	{ size=2 }	std::vector<int,std::alloc...
Weather	9	int
hour	15	int
map_method	{ size=140 }	std::map<std::pair<int,i...
minute	15	int
this	0x0000009359cff590 {ui={...} Point={ size=106 } Route={ size=140 } ...}	optimal_path_planning *



名称	值	类型
map_method	{ size=140 }	std::map<std::pair<int,i...
[comparator]	less	std::_Compressed_pair...
[allocator]	allocator	std::_Compressed_pair...
[(0, 1)]	22	std::pair<std::pair<int,i...
[(0, 11)]	49	std::pair<std::pair<int,i...
[(1, 2)]	24	std::pair<std::pair<int,i...
[(2, 6)]	32	std::pair<std::pair<int,i...
[(2, 7)]	48	std::pair<std::pair<int,i...
[(3, 4)]	20	std::pair<std::pair<int,i...
[(3, 64)]	33	std::pair<std::pair<int,i...
[(4, 0)]	31	std::pair<std::pair<int,i...
[(4, 22)]	15	std::pair<std::pair<int,i...
[(5, 25)]	16	std::pair<std::pair<int,i...
[(6, 5)]	21	std::pair<std::pair<int,i...
[(7, 9)]	23	std::pair<std::pair<int,i...
[(8, 10)]	24	std::pair<std::pair<int,i...
[(9, 8)]	16	std::pair<std::pair<int,i...
[(10, 14)]	47	std::pair<std::pair<int,i...
[(10, 18)]	18	std::pair<std::pair<int,i...
[(11, 13)]	22	std::pair<std::pair<int,i...
[(12, 14)]	20	std::pair<std::pair<int,i...
[(13, 12)]	21	std::pair<std::pair<int,i...
[(14, 16)]	17	std::pair<std::pair<int,i...
[(15, 103)]	20	std::pair<std::pair<int,i...
[(16, 15)]	17	std::pair<std::pair<int,i...
[(17, 104)]	20	std::pair<std::pair<int,i...
[(18, 17)]	24	std::pair<std::pair<int,i...
[(20, 6)]	26	std::pair<std::pair<int,i...
[(21, 20)]	31	std::pair<std::pair<int,i...
[(21, 23)]	22	std::pair<std::pair<int,i...
[(22, 21)]	16	std::pair<std::pair<int,i...
[(23, 24)]	12	std::pair<std::pair<int,i...
[(25, 26)]	16	std::pair<std::pair<int,i...
[(26, 27)]	15	std::pair<std::pair<int,i...



Q	{tit={map_Walk={ size=140 } map_Ride={ size=137 } map_Drive={ size=13...	method
tit	{map_Walk={ size=140 } map_Ride={ size=137 } map_Drive={ size=136 } ...}	TIT
map_Walk	{ size=140 }	std::map<std::pair<int,i...
map_Ride	{ size=137 }	std::map<std::pair<int,i...
map_Drive	{ size=136 }	std::map<std::pair<int,i...
stopRide	{ size=38 }	std::vector<int,std::alloc...
stopDrive	{ size=9 }	std::vector<int,std::alloc...
Limit	{ size=10 }	std::vector<std::pair<in...
Peak	{ size=8 }	std::vector<std::pair<st...
Way	{ size=2 }	std::vector<int,std::alloc...
Planning	{ size=0 }	std::vector<std::vector<...
flagWalk	0	int
flagRide	0	int
flagDrive	0	int
Weather	9	int
T	15	int
hour	15	int
minute	15	int

map_Walk	{ size=140 }	std::map<std::pair<int,i
[comparator]	less	std::_Compressed_pair...
[allocator]	allocator	std::_Compressed_pair...
[(0, 1)]	44	std::pair<std::pair<int,i..
[(0, 11)]	98	std::pair<std::pair<int,i..
[(1, 2)]	48	std::pair<std::pair<int,i..
[(2, 6)]	64	std::pair<std::pair<int,i..
[(2, 7)]	96	std::pair<std::pair<int,i..
[(3, 4)]	40	std::pair<std::pair<int,i..
[(3, 64)]	66	std::pair<std::pair<int,i..
[(4, 0)]	62	std::pair<std::pair<int,i..
[(4, 22)]	30	std::pair<std::pair<int,i..
[(5, 25)]	32	std::pair<std::pair<int,i..
[(6, 5)]	42	std::pair<std::pair<int,i..
[(7, 9)]	46	std::pair<std::pair<int,i..
[(8, 10)]	48	std::pair<std::pair<int,i..

map_Ride	{ size=137 }	std::map<std::pair<int,i
[comparator]	less	std::_Compressed_pair...
[allocator]	allocator	std::_Compressed_pair...
[(0, 1)]	31	std::pair<std::pair<int,i...
[(0, 11)]	58	std::pair<std::pair<int,i...
[(1, 2)]	33	std::pair<std::pair<int,i...
[(2, 6)]	45	std::pair<std::pair<int,i...
[(2, 7)]	57	std::pair<std::pair<int,i...
[(3, 4)]	24	std::pair<std::pair<int,i...
[(3, 64)]	39	std::pair<std::pair<int,i...
[(4, 0)]	44	std::pair<std::pair<int,i...
[(4, 22)]	18	std::pair<std::pair<int,i...
[(5, 25)]	19	std::pair<std::pair<int,i...
[(6, 5)]	25	std::pair<std::pair<int,i...
[(7, 9)]	27	std::pair<std::pair<int,i...
[(8, 10)]	28	std::pair<std::pair<int,i...
[(9, 8)]	19	std::pair<std::pair<int,i...

map_Drive	{ size=136 }	std::map<std::pair<int,i
[comparator]	less	std::_Compressed_pair...
[allocator]	allocator	std::_Compressed_pair...
[(0, 1)]	49	std::pair<std::pair<int,i...
[(0, 11)]	49	std::pair<std::pair<int,i...
[(1, 2)]	54	std::pair<std::pair<int,i...
[(2, 6)]	48	std::pair<std::pair<int,i...
[(2, 7)]	48	std::pair<std::pair<int,i...
[(3, 4)]	30	std::pair<std::pair<int,i...
[(3, 64)]	49	std::pair<std::pair<int,i...
[(4, 0)]	46	std::pair<std::pair<int,i...
[(4, 22)]	22	std::pair<std::pair<int,i...
[(5, 25)]	24	std::pair<std::pair<int,i...
[(6, 5)]	31	std::pair<std::pair<int,i...
[(7, 9)]	23	std::pair<std::pair<int,i...
[(8, 10)]	24	std::pair<std::pair<int,i...

上述测试说明，系统所有后台数据正常，未出现违法数据，表明数据初始化模块工作符合要求。为了提高测试的正确性，我进行了多次测试，如下：

测试用例	预期结果	测试结果
15 时 15 分；多云 13 度	内容生成正确	内容生成正确
8 时 47 分；大风 16 度	内容生成正确	内容生成正确
17 时 36 分；晴 26 度	内容生成正确	内容生成正确
21 时 19 分；大雪 12 度	内容生成正确	内容生成正确
9 时 57 分；小雨 10 度	内容生成正确	内容生成正确
16 时 38 分；晴 17 度	内容生成正确	内容生成正确

7.2.2 路径规划模块

测试思路：为了确保路径规划的准确性，我们在不同地点之间进行多次规划，并通过规划返回的结果来判断路径规划的正确与否。通过多次规划，我们可以比较不同结果之间的一致性，确保路径规划算法的可靠性和正确性。如果多次规划的结果相同或非常接近，我们可以认为路径规划是正确的；如果结果存在明显差异，我们需要进一步检查和调整算法，以提高规划的准确性。这种多次规划的方法可以帮助我们验证路径规划的可靠性，并为用户提供准确的导航服务。

测试界面如下：

上述测试说明，从北篮球场到文典阁经过榴园食堂和北池塘距离最短，经过手工比较，此条规划正确，说明路径规划模块工作符合要求。为了提高测试的正确性，我进行了多次测

试，如下：

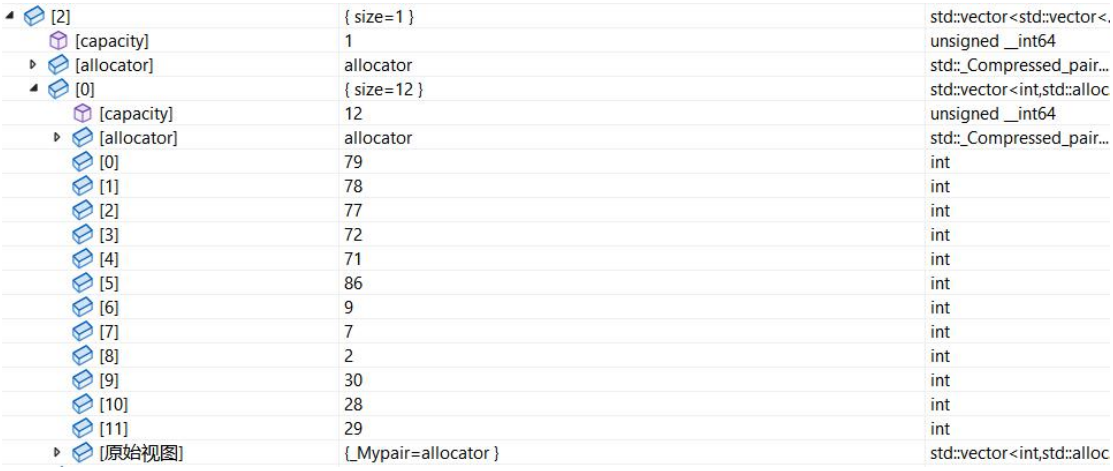
测试用例	预期结果	测试结果
杏园到南体	路径规划正确	路径规划正确
北池塘到东门	路径规划正确	路径规划正确
蕙园到西门	路径规划正确	路径规划正确
桔园食堂到博南	路径规划正确	路径规划正确
西门超市到艺术楼	路径规划正确	路径规划正确
桔园到行政楼	路径规划正确	路径规划正确

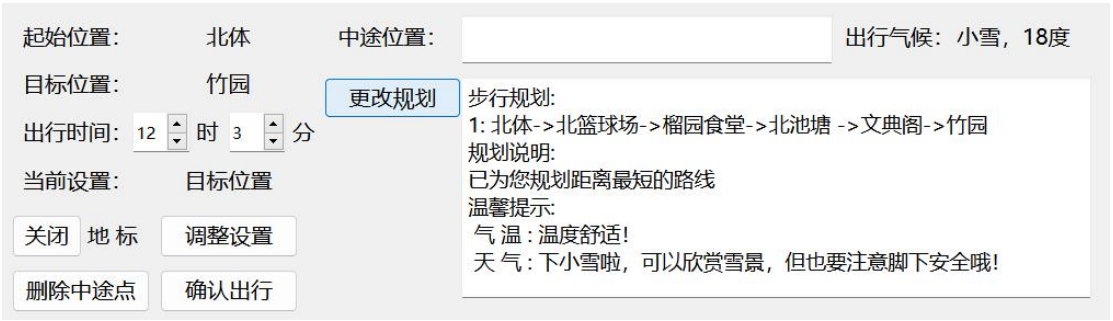
7.2.3 显示模块

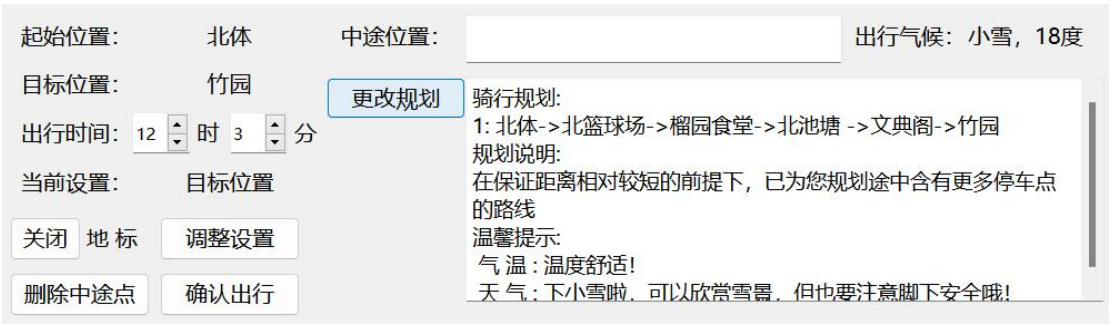
测试思路：为了确保系统的数据一致性和正确性，我们可以通过添加断点的方式在界面将后台数据显示之后插入断点。在断点处，我们可以比对界面数据和后台数据的一致性，以测试系统是否正确地将后台数据传递给了界面。通过比对数据，我们可以检测潜在的错误或非法数据，并及时进行修复和调整。这种断点测试的方法可以帮助我们确保系统的数据处理过程准确无误，提高系统的可靠性和安全性。

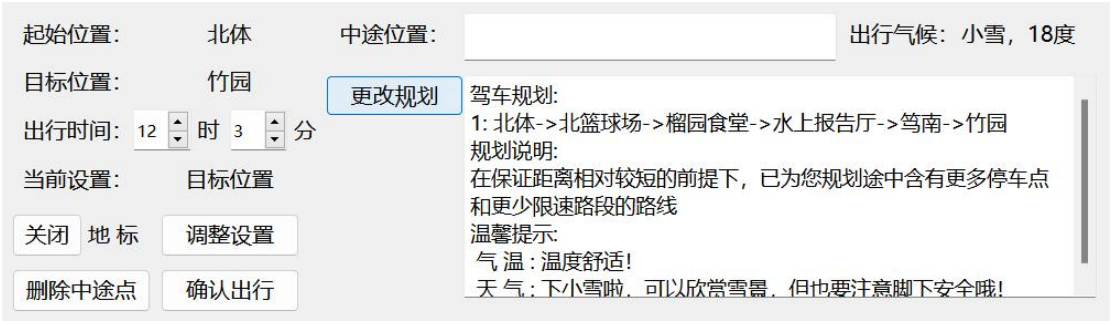
测试界面如下：

Planning	{ size=3 }	std::vector<std::vector<..
[capacity]	3	unsigned __int64
[allocator]	allocator	std::_Compressed_pair...
[0]	{ size=1 }	std::vector<std::vector<..
[capacity]	1	unsigned __int64
[allocator]	allocator	std::_Compressed_pair...
[0]	{ size=12 }	std::vector<int,std::alloc..
[capacity]	12	unsigned __int64
[allocator]	allocator	std::_Compressed_pair...
[0]	79	int
[1]	78	int
[2]	77	int
[3]	72	int
[4]	71	int
[5]	70	int
[6]	0	int
[7]	1	int
[8]	2	int
[9]	30	int
[10]	28	int
[11]	29	int
[原始视图]	{_Mypair=allocator }	std::vector<int,std::alloc..
[原始视图]	{_Mypair=allocator }	std::vector<std::vector<..
[1]	{ size=1 }	std::vector<std::vector<
[capacity]	1	unsigned __int64
[allocator]	allocator	std::_Compressed_pair...
[0]	{ size=12 }	std::vector<int,std::alloc
[capacity]	12	unsigned __int64
[allocator]	allocator	std::_Compressed_pair...
[0]	79	int
[1]	78	int
[2]	77	int
[3]	72	int
[4]	71	int
[5]	70	int
[6]	0	int
[7]	1	int
[8]	2	int
[9]	30	int
[10]	28	int
[11]	29	int
[原始视图]	{_Mypair=allocator }	std::vector<int,std::alloc









上述测试说明系统界面显示信息和后台数据匹配，说明显示模块工作符合要求。为了提高测试的正确性，我进行了多次测试，如下：

测试用例	预期结果	测试结果
梅园食堂到体育馆	路径显示正确	路径显示正确
李园到东门	路径显示正确	路径显示正确
蕙园到西门	路径显示正确	路径显示正确
桔园到人文楼	路径显示正确	路径显示正确

西门超市到艺术楼	路径显示正确	路径显示正确
枣园到行政楼	路径显示正确	路径显示正确

8 项目管理和进度安排

进度安排	规划时间	完成度
完成需求分析	7 天	已完成
收集地图信息	3 天	已完成
搭建系统初始框架	3 天	已完成
设计前端界面	2 天	已完成
设计后端算法	8 天	已完成
编写项目文档	3 天	已完成