# 模块设计报告

第二组 任泽高 安轶东 董佳鑫

[模块设计报告](#模块设计报告)  
 [主程序模块 main.cpp](#主程序模块-maincpp)  
 [用户登录与管理模块 login.cpp login\_page.cpp](#用户登录与管理模块-logincpp-loginpagecpp)  
 [用户登录和管理模块分析](#用户登录和管理模块分析)  
 [主要函数分析](#主要函数分析-1)  
 [1. 构造函数 login](#X91aff62e258e499c247817e398834bc46e5fd1e)  
 [2. 析构函数 ~login](#X8fc46782ed9875d96434374614e72cee11ca190)  
 [3. on\_btn\_login\_clicked 函数](#X326fbaa1ad0939d4890393d4ec0e800a24e5817)  
 [4. isAccountExists 函数](#X3527d403071ddcb383ad1bc779e2ccb9ac4f430)  
 [5. createAccount 函数](#X5367fc4e614a8a00ed00069253489f1643621e9)  
 [6. validatePassword 函数](#X19e76ded3737e5ec1c1457ad54ff18507a91fa0)  
 [算法分析](#算法分析-1)  
 [1. JSON 文件读取和解析](#Xb7429d6295a78ee57993c3f3fb256485e502107)  
 [2. 用户账户验证](#X64654b0a1c40ac1702d73457021e68c56a01342)  
 [3. 用户账户创建](#Xffbf3ea9b5e177d3959044897f7a0804fb93c4f)  
 [数据存储](#数据存储)  
 [1. JSON 文件结构](#X3c1b882894993886d8f8d73d0bd35cefb458820)  
 [2. 数据字段](#X85c9d226de3c70b797eb7d4aeedb820ea1cb385)  
 [复杂度分析](#复杂度分析)  
 [1. 账户检查和密码验证](#Xf0631752a3449d4bce867415a7bfe6009200d45)  
 [2. 账户创建](#X87658a7918d52bc852cfcd2a76fb80afd709610)  
 [总结](#总结-1)  
 [功能选择模块 functionselectepage.cpp](#功能选择模块-functionselectepagecpp)  
 [界面切换](#界面切换)  
 [信号和槽](#信号和槽)  
 [KMP算法模块 kmp.cpp](#kmp算法模块-kmpcpp)  
 [KMP 算法模块分析](#kmp-算法模块分析)  
 [1. 构建部分匹配表](#X93bb51da1a4a9357f1de9d390928d6bc89747ff)  
 [2. KMP 字符串匹配算法](#Xe02a09fb6164a17d64f68917740b3757a0f6a7a)  
 [3. 多模式串匹配](#X4e9f314ad0223da5a13d391e4febc40ddcfcb5f)  
 [具体分析](#具体分析)  
 [1. 构建部分匹配表（LPS 表）](#X725c6ae377aeb858c90af8321af57ec848872c5)  
 [2. KMP 字符串匹配](#Xb5e055319f6f476248457d683b23fcc5fb1666d)  
 [3. 多模式串匹配](#Xeeda2c21fb08d9be290bbe2417aaac496adde82)  
 [选取 KMP 算法的原因](#选取-kmp-算法的原因)  
 [Huffman编码压缩/解压模块 huffman.cpp](#huffman编码压缩解压模块-huffmancpp)  
 [Huffman 编码模块分析](#huffman-编码模块分析)  
 [主要函数分析](#主要函数分析-2)  
 [1. buildHuffmanTree 函数](#X398e698dd693ddfecb120f9a6bae5893411512e)  
 [2. generateHuffmanCodes 函数](#X7f859d20a40b9937fbacbfae279ff89f1654be1)  
 [3. encodeHuffman 函数](#X3dcca39d5deb1330bc02e13f4e98c46a5e5a336)  
 [4. decodeHuffman 函数](#X0aba70c7d053850133c92d29e25da77ca0082dc)  
 [5. saveHuffmanCodesToJSON 和 loadHuffmanCodesFromJSON 函数](#X75c90fd3498a70abf646f6ba94d95b79ac08c4b)  
 [6. writeBinaryFile 和 readBinaryFile 函数](#Xf98aa18ec7cf103dfd520cffc41a3c23883f8d4)  
 [7. compressString 函数](#X562aa49f4ced6c358510fad2202b9a55ac3ed15)  
 [8. decompressString 函数](#Xebb8dde2b6ef78a706632af37658fa53d35dbc2)  
 [总结](#总结-2)  
 [游学推荐模块 destinationrecommend.cpp](#游学推荐模块-destinationrecommendcpp)  
 [游学推荐模块算法分析](#游学推荐模块算法分析)  
 [1. 推荐功能](#X00d45c364df70dd0d3f2eb5eb8a01bbfe702f5c)  
 [2. 排序功能](#Xcc2ad89e6452cf8974aee3e0f64551159e4a480)  
 [2.1 按热度排序](#X1008b758ae51a78d80062d63256908abb5deaa8)  
 [2.2 按评分排序](#X2e61695cee1149334bbec19ef5b34c2d05c08a9)  
 [3. 搜索功能](#X6f5b7135dff4467b99e5bee10509d5dc50967f7)  
 [4. 排序算法](#X0048fb071228aba8751d53ad0f5b902a2b47140)  
 [4.1 快速排序和划分函数](#X5cc743b3e2083709d67a3ebed0e62a7843ddf85)  
 [4.2 快速选择和划分函数](#X28342a99b9e8c5370e582fb0981070682e34ec8)  
 [性能分析](#性能分析-1)  
 [地图数据读取模块 myGraph.cpp](#地图数据读取模块-mygraphcpp)  
 [地图文件读取模块算法分析](#地图文件读取模块算法分析)  
 [1. 读取景点信息](#X70b5ba82f9f4648c968fbd02c20007259f33bc4)  
 [1.1 读取景点信息并构建图结构](#Xc8f96e53d563bd60867d0841bd96e340e525339)  
 [1.2 保存更新后的景点信息](#X1006cf2b01cdafb1bf551a9871f91f6a5350c8e)  
 [主要函数分析](#主要函数分析-3)  
 [1. 加载景点信息](#X87b319eb6dc52429cb5bfe6c7e7386665fe2f86)  
 [2. 保存景点信息](#X84a97817f2d66959b88cabd6350f4f225336f15)  
 [性能分析](#性能分析-2)  
 [总结](#总结-3)  
 [路线规划算法模块 route\_plan.cpp](#路线规划算法模块-routeplancpp)  
 [路线规划模块算法分析](#路线规划模块算法分析)  
 [1. 多点路线规划](#Xdaeba228e612bf79ed1bb151852463aaab27fbd)  
 [2. Floyd-Warshall 算法](#X988019a5d3c3e36942b30eef179f769bf503926)  
 [2.1 基于距离的 Floyd-Warshall 算法](#X8ffade9bc1f2c5a97c4f35afe86c27262795f6b)  
 [2.2 基于时间的 Floyd-Warshall 算法](#X3402728958bf467200d07ef3892ec890426d7ad)  
 [3. Dijkstra 算法](#X7e1e2a6f3bd5ec3ec3c71072531e53cc50e8518)  
 [3.1 基于时间的 Dijkstra 算法](#Xea69c721e28f6aaf7168faa00c6e9c7e4ae6d81)  
 [3.2 基于距离的 Dijkstra 算法](#X18174e084ba58b18b181183e8c4368d383caafd)  
 [4. 路径获取函数](#Xdc1f7420aca04be3da8bdebae1ac507c61ed6d4)  
 [4.1 获取距离](#Xe9a0e547d5a873217427eff4aa4bdb18cf836fd)  
 [4.2 获取时间](#Xb432236dc8b22137d4d6d526668d8ebff40509a)  
 [4.3 保存路径](#X93f052677db1561f5220875cf03e284a40a3898)  
 [4.4 获取路径](#X01869dd307fd89e57e9b039c2c2febe77ad3448)  
 [5. 路径打印函数](#X95aed35207964bde9fe4686af68d2d9ca879cde)  
 [5.1 打印路径](#Xc98a321900b729bca32b0d308cab7e963270af3)  
 [5.2 打印所有点和中间点](#Xf7b97d952f974d6057b10b8e7d03c1f253b1d5f)  
 [总结](#总结-4)  
 [路线规划/设施查询/美食推荐集成模块 mymap.cpp](#路线规划设施查询美食推荐集成模块-mymapcpp)  
 [路线规划](#路线规划)  
 [1. 路线规划功能概述](#X0f40c25426ae278e6c57c2eef9b192f834ab659)  
 [2. 两点路线规划](#X0fafd70049b511562922d36961256649908d10f)  
 [3. 多点路线规划](#Xc39ff3f221ea73d0d73dcc9e1ef6d7ea4df37e5)  
 [4. 辅助函数](#X3d1c9d83ab18ee25981bba84dccd9417c2dc8b0)  
 [5. 打印路径和中间点](#Xd32a4d02a8fe2087e410f2dce5bf59299c162da)  
 [设施查询](#设施查询)  
 [1. 设施查询功能概述](#X84ccc71968ab11c19daeaf77b953d28779aa465)  
 [2. 快速排序](#X2f9445ae55c752608a9cf6ab099ae0147f3f898)  
 [美食推荐](#美食推荐)  
 [1. 美食推荐功能概述](#Xf0759ae3041db8d0efe30b18d769fa5459af0a3)  
 [2. 热度排序](#X4813e0d14cb6b33ade2ef3cbe0da958d8d0d04a)  
 [3. 距离排序](#Xe9b0d40d3f92823b42097d75719839c371bf87a)  
 [其他辅助函数](#其他辅助函数)  
 [图像化导航 myLabel 类功能分析](#图像化导航-mylabel-类功能分析)  
 [1. 类概述](#X58e26905f8c76673f6b2c6202fb05b0867aee45)  
 [2. 构造函数 myLabel](#X06422b20b8fc5a764ae1b52e294a3b2166ed61a)  
 [3. 重载 paintEvent 函数](#X858818539c61a6f16dec73fb2891eb8b419d602)  
 [4. routePaint 函数](#X2ad9abc35e215adf3836b754f5b637b83ffa526)  
 [5. PaintRoute 函数](#Xdae74ee4ee3fb93a09ee4d0e2b4972946be12e2)  
 [6. multiRoutePaint 函数](#Xd279ea8bd8021f5c3b9469e6ee4dd31ede1671a)  
 [总结](#总结-5)  
 [日记管理模块 diarymanagement.cpp](#日记管理模块-diarymanagementcpp)  
 [日记管理模块算法分析](#日记管理模块算法分析)  
 [1. 解析日记内容](#Xb8e1b664dcbb804a2edda56d05caec81fb13d0f)  
 [2. 搜索功能](#X8f060a244e6d604f0f8739f7ffa508b6847d5c6)  
 [2.1 按地点搜索](#Xdbd43dc69b66c80bcfee24db3a56a3895cdb49e)  
 [2.2 按标题搜索](#X4068b8df3ffd1132a0315a091360d70d3edba89)  
 [2.3 按内容搜索](#X7913c8fd32be436180416265f70add92f71cd4a)  
 [3. 排序功能](#Xbd7276eb9d548b956fbf0be3d643388e2c927d6)  
 [3.1 按评分排序](#Xd7d1041632619312dc733395ac6cf9e02b737ac)  
 [3.2 按人气排序](#X63e6da18db25f82678e08be32da574e83c49703)  
 [3.3 按兴趣排序](#X37074a108e508d881d656e170e49f1adcd741cc)  
 [主要函数分析](#主要函数分析-4)  
 [1. quickSort 和 partition 函数](#X742a3081a96fbbc70524198f333ba05c53b5788)  
 [2. findTopK 和 partition 函数](#Xa415656b3f35c6b2939daaf156038d138ae8a05)  
 [3. 哈希函数 complexHashFunction](#X2fc7c2df0e1b0b1f20203c6e18dbff3fd231fa5)  
 [性能分析](#性能分析-3)  
 [总结](#总结-6)  
 [日记创建模块 diarycreate.cpp](#日记创建模块-diarycreatecpp)  
 [日记创建模块分析](#日记创建模块分析)  
 [主要函数分析](#主要函数分析-5)  
 [1. on\_confirm\_clicked 函数](#Xc856b10b9bf46eb5baad92b3b02ca6035e46436)  
 [2. parseDiariesContents 函数](#X491f2507b7493fd9ad340a17733f8b81892ba1e)  
 [3. saveDiary 函数](#Xaf4652deb7b20247f05d96df5c018f003f78b25)  
 [4. compressString 和 decompressString 函数](#X108752fe55a06b93b7381d2e4956ff75276ba2e)  
 [总结](#总结-7)  
 [日记显示模块 diaryview.cpp](#日记显示模块-diaryviewcpp)  
 [DiaryView 模块分析](#diaryview-模块分析)  
 [主要函数分析](#主要函数分析-6)  
 [1. 构造函数 DiaryView](#Xfed132be7258b81eebdc96d28447742c117b4b0)  
 [2. 析构函数 ~DiaryView](#X8b17523dda274fb4aca2e0b58c578e48564d3bf)  
 [3. on\_confirmBnt\_clicked 函数](#X0bf2cfd6af11032e01899ee596b7bfa405543d4)  
 [算法分析](#算法分析-2)  
 [1. 获取评分和更新日记](#Xe4cb954fa7cf84964a560472d00bb5d88739f06)  
 [2. 读取和解析 JSON 文件](#X573ca8dcbe294cba30c83ee051e75fc8a66942e)  
 [3. 查找并更新日记条目](#Xc40e297bf07a0f9bedd04865e1d373763dec05c)  
 [4. 写回 JSON 文件](#X9f2a8afd5ce3b211257e0f9559ed9ade937adce)  
 [总结](#总结-8)  
 [模块调用关系分析](#模块调用关系分析)  
 [项目模块调用关系总结](#项目模块调用关系总结)  
 [1. 用户登录和管理模块（login.cpp）](#X1c1d93b1232ee3cd4827e2df8e2f0729085de5e)  
 [2. 功能选择界面模块（functionselectpage.cpp）](#X29cc1c358ef9e18dd89d6f058905337113a2802)  
 [3. 日记管理模块（diarymanagement.cpp）](#X9480b5a134541cbe1ac4621edae7d67b50317f7)  
 [4. 游学推荐模块（destinationrecommend.cpp）](#X96a46171c764f07b628626da1a2fff92b48beb4)  
 [5. 路线规划模块（route\_plan.cpp）](#X7b918cc966e088b865af1c57777f35d7220e190)  
 [6. 地图文件读取模块（myGraph.cpp）](#X1c9bdee42cd8cc81964aa1faeb50cb1ed1efddd)  
 [7. 地图显示模块（mymap.cpp）](#X42f13874a27ea36bcceed1c29074eaf085946a9)  
 [8. 自定义标签显示模块（myLabel.cpp）](#Xf14ebacfda01a91887ae47e0cf45b82ad742e88)  
 [总结](#总结-9)

## 主程序模块 main.cpp

主函数主要执行以下操作：

1. **加载地图数据**：调用 loadAttractionArray 函数，从 data\_map.json 文件中加载地图和景点信息。
2. **初始化应用程序**：创建 QApplication 对象，初始化 Qt 应用程序。
3. **显示登录界面**：创建并显示登录界面窗口，用户可以在此登录或注册。
4. **进入事件循环**：进入 Qt 的事件循环，等待并处理用户交互事件。

## 用户登录与管理模块 login.cpp login\_page.cpp

### 用户登录和管理模块分析

用户登录和管理模块负责处理用户登录、账户验证和账户创建等操作。该模块通过读取和修改 user\_accounts.json 文件来实现用户账户的管理。以下是对该模块中各个算法和函数的详细分析。

#### 主要函数分析

#### 1. 构造函数 login

login::login(QWidget \*parent)  
 : QWidget(parent)  
 , ui(new Ui::login)  
{  
 ui->setupUi(this);  
}

* **功能**：初始化登录界面。
* **实现**：
  + 初始化界面元素。
* **复杂度**：时间复杂度为 O(1)，因为只是简单的对象初始化和属性赋值。

#### 2. 析构函数 ~login

login::~login()  
{  
 delete ui;  
}

* **功能**：释放 login 对象占用的资源。
* **实现**：
  + 删除 UI 对象，释放内存。
* **复杂度**：时间复杂度为 O(1)。

#### 3. on\_btn\_login\_clicked 函数

void login::on\_btn\_login\_clicked()

* **功能**：处理用户点击登录按钮的操作，包括用户账户验证和账户创建。
* **实现**：
  + 从 UI 界面获取用户输入的账号和密码。
  + 调用 isAccountExists 函数检查账户是否存在。
  + 如果账户存在，调用 validatePassword 函数验证密码是否正确。
  + 如果密码正确，显示登录成功信息并进入功能选择界面。
  + 如果账户不存在，调用 createAccount 函数创建新账户，并进入功能选择界面。
* **复杂度**：时间复杂度为 O(n)，其中 n 为 JSON 文件的大小。

#### 4. isAccountExists 函数

bool isAccountExists(const string& account)

* **功能**：检查用户账户是否存在于 JSON 文件中。
* **实现**：
  + 读取 user\_accounts.json 文件并解析 JSON 数据。
  + 检查 JSON 数据中是否存在指定账户。
* **复杂度**：时间复杂度为 O(n)，其中 n 为 JSON 文件的大小。

#### 5. createAccount 函数

void createAccount(const string& account, const string& password)

* **功能**：在 JSON 文件中创建新的用户账户。
* **实现**：
  + 读取现有的 JSON 文件或创建新的 JSON 结构。
  + 在 JSON 结构中添加新账户。
  + 将更新后的 JSON 数据写回文件。
* **复杂度**：时间复杂度为 O(n)，其中 n 为 JSON 文件的大小。

#### 6. validatePassword 函数

bool validatePassword(const string& account, const string& enteredPassword)

* **功能**：验证输入的密码是否与 JSON 文件中存储的密码匹配。
* **实现**：
  + 读取 user\_accounts.json 文件并解析 JSON 数据。
  + 检查 JSON 数据中指定账户的密码是否与输入的密码匹配。
* **复杂度**：时间复杂度为 O(n)，其中 n 为 JSON 文件的大小。

### 算法分析

#### 1. JSON 文件读取和解析

* **功能**：读取 user\_accounts.json 文件并解析其中的 JSON 数据。
* **实现**：使用 ifstream 读取文件内容，并使用 cJSON\_Parse 解析 JSON 数据。
* **复杂度**：时间复杂度为 O(n)，其中 n 为 JSON 文件的大小。

#### 2. 用户账户验证

* **功能**：检查用户账户是否存在，并验证密码是否正确。
* **实现**：
  + 调用 isAccountExists 函数检查账户是否存在。
  + 调用 validatePassword 函数验证密码是否正确。
* **复杂度**：时间复杂度为 O(n)。

#### 3. 用户账户创建

* **功能**：在 JSON 文件中创建新的用户账户。
* **实现**：调用 createAccount 函数，在 JSON 数据中添加新账户，并将更新后的 JSON 数据写回文件。
* **复杂度**：时间复杂度为 O(n)。

### 数据存储

#### 1. JSON 文件结构

* **文件路径**：user\_accounts.json
* **结构**：
* {  
   "accounts": {  
   "username1": "password1",  
   "username2": "password2"  
   }  
  }

#### 2. 数据字段

* **accounts**：顶层字段，存储所有用户账号和密码的集合。
* **账号（Key）**：用户账号的字符串表示，唯一标识一个用户。
* **密码（Value）**：用户密码的字符串表示，用于用户登录时的身份验证。

### 复杂度分析

#### 1. 账户检查和密码验证

* **时间复杂度**：O(n)，其中 n 为 JSON 文件的大小。因为需要读取整个文件并解析 JSON 数据。

#### 2. 账户创建

* **时间复杂度**：O(n)，其中 n 为 JSON 文件的大小。因为需要读取文件、更新 JSON 数据并写回文件。

### 总结

用户登录和管理模块通过读取和修改 user\_accounts.json 文件，实现了用户账户的验证和创建功能。该模块通过 JSON 数据结构存储用户信息，使用简单易懂的算法实现了账户的管理。虽然读取和写入 JSON 文件的操作在大规模数据情况下可能会带来性能瓶颈，但对于一般应用场景来说已经足够。

## 功能选择模块 functionselectepage.cpp

功能选择模块主要是用户登录之后选择功能的界面。

### 界面切换

FunctionSelectPage 模块的核心功能是根据用户选择的功能选项，进行界面切换。通过下拉菜单选择功能，用户可以选择进入不同的子模块，如目的地推荐和日记管理。

* 界面切换的实现
  + 使用 Qt 的 QComboBox 组件获取用户的选择。
  + 使用 switch 语句判断选择，创建相应的子模块界面对象，并调用 show() 函数显示新界面。

### 信号和槽

FunctionSelectPage 模块使用了 Qt 的信号和槽机制，在用户选择某一功能模块时，隐藏当前界面，并在子模块界面关闭时重新显示功能选择界面。

* 信号和槽的实现
  + 在创建子模块界面对象时，使用 connect 函数连接子模块的 quit\_function 信号到当前对象的 show 槽。
  + 在子模块界面关闭时，发射 quit\_function 信号，重新显示功能选择界面。

## KMP算法模块 kmp.cpp

### KMP 算法模块分析

KMP 算法模块实现了字符串的高效匹配功能。该模块的核心是 kmpSearch 函数，通过构建部分匹配表（LPS 表），实现字符串的快速匹配。以下是对该模块中各个算法的详细分析。

#### 1. 构建部分匹配表

vector<int> buildPartialMatchTable(const string& pattern)

* **功能**：构建部分匹配表（LPS 表），用于记录模式串中每个位置的最长前缀后缀匹配长度。
* **算法**：遍历模式串，通过比较字符，记录每个位置的部分匹配长度。
* **复杂度**：O(m)，其中 m 为模式串的长度。
* **原因**：LPS 表用于加速 KMP 算法的匹配过程，避免不必要的重复比较。

#### 2. KMP 字符串匹配算法

bool kmpSearch(const string& text, const string& pattern)

* **功能**：使用 KMP 算法在文本中查找模式串，返回匹配结果。
* **算法**：通过 LPS 表加速匹配过程，避免重复比较。
* **复杂度**：O(n + m)，其中 n 为文本长度，m 为模式串长度。
* **原因**：KMP 算法在最坏情况下的时间复杂度为线性时间，适合处理大规模文本匹配问题。

#### 3. 多模式串匹配

bool kmpPatternMatch(const vector<string>& patterns, const string& text)

* **功能**：在文本中查找多个模式串中的任意一个，返回匹配结果。
* **算法**：遍历模式串集合，使用 KMP 算法进行匹配。
* **复杂度**：O(k \* (n + m))，其中 k 为模式串的数量，n 为文本长度，m 为单个模式串的平均长度。
* **原因**：在实际应用中，可能需要匹配多个模式串，组合 KMP 算法能高效处理多个模式串匹配问题。

### 具体分析

#### 1. 构建部分匹配表（LPS 表）

部分匹配表记录了模式串中每个位置的最长前缀后缀匹配长度。在构建 LPS 表时，逐个字符比较模式串中的字符，通过移动指针和记录长度，构建出完整的 LPS 表。

* **算法实现**：
  + 初始化长度 len 为 0，指针 i 为 1。
  + 遍历模式串：
    - 如果模式串的当前字符与前缀字符匹配，则增加长度，并记录在 LPS 表中。
    - 如果不匹配，根据 LPS 表调整长度，继续比较。
  + 直到遍历完整个模式串，完成 LPS 表的构建。
* **算法性能**：
  + 时间复杂度为 O(m)，空间复杂度为 O(m)。
  + 在模式串长度较长时，能显著减少匹配过程中的重复比较，提高匹配效率。

#### 2. KMP 字符串匹配

KMP 算法通过 LPS 表加速模式串的匹配过程。在匹配过程中，通过移动指针和调整匹配位置，快速找到模式串在文本中的位置。

* **算法实现**：
  + 初始化指针 i 和 j，分别指向文本和模式串的起始位置。
  + 遍历文本：
    - 如果当前字符匹配，则移动指针。
    - 如果模式串匹配完成，返回匹配成功。
    - 如果不匹配，根据 LPS 表调整指针位置，继续比较。
  + 如果遍历完整个文本，返回匹配失败。
* **算法性能**：
  + 时间复杂度为 O(n + m)，空间复杂度为 O(m)。
  + 能在线性时间内完成匹配，适合大规模文本处理。

#### 3. 多模式串匹配

多模式串匹配在文本中查找多个模式串中的任意一个。通过遍历模式串集合，依次使用 KMP 算法进行匹配。

* **算法实现**：
  + 遍历模式串集合：
    - 对每个模式串，使用 KMP 算法进行匹配。
    - 如果任意一个模式串匹配成功，返回匹配成功。
  + 如果遍历完所有模式串未匹配成功，返回匹配失败。
* **算法性能**：
  + 时间复杂度为 O(k \* (n + m))，空间复杂度为 O(m)。
  + 能处理多个模式串的匹配需求，提高算法的实用性。

### 选取 KMP 算法的原因

KMP 算法通过预处理模式串，构建 LPS 表，在匹配过程中避免了不必要的重复比较。相比于暴力匹配算法，KMP 算法能在线性时间内完成匹配，性能优越。在处理大规模文本匹配问题时，KMP 算法是一个高效的选择。

## Huffman编码压缩/解压模块 huffman.cpp

### Huffman 编码模块分析

Huffman 编码模块主要用于对字符串数据进行压缩和解压缩，通过构建 Huffman 树来生成字符的编码表，并利用该编码表对数据进行编码和解码。以下是对该模块中各个算法和函数的详细分析。

#### 主要函数分析

#### 1. buildHuffmanTree 函数

HuffmanNode\* buildHuffmanTree(const unordered\_map<char, int>& frequencyMap)

* **功能**：根据字符频率表构建 Huffman 树。
* **算法**：Huffman 算法是一种贪心算法，通过频率最低的两个节点合并成一个新节点，重复该过程直到只剩下一个节点，即 Huffman 树的根节点。
* **复杂度**：时间复杂度为 O(n log n)，其中 n 为字符种类数。空间复杂度为 O(n)，需要存储所有字符的节点。
* **原因**：Huffman 算法能够最优地编码字符，减少编码后的数据长度，是一种经典的无损压缩算法。

#### 2. generateHuffmanCodes 函数

void generateHuffmanCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& huffmanCodes)

* **功能**：根据 Huffman 树生成字符的 Huffman 编码表。
* **算法**：递归遍历 Huffman 树，左子树路径编码为 "0"，右子树路径编码为 "1"，直到叶子节点，将路径编码存储到编码表中。
* **复杂度**：时间复杂度为 O(n)，其中 n 为字符种类数。空间复杂度为 O(n)，需要存储所有字符的编码。
* **原因**：递归遍历简单直观，适合树结构的操作。

#### 3. encodeHuffman 函数

string encodeHuffman(string input, const unordered\_map<char, string>& huffmanCodes)

* **功能**：将输入字符串编码为 Huffman 编码。
* **算法**：遍历输入字符串，对于每个字符，查找其对应的 Huffman 编码，并将编码拼接到结果字符串中。
* **复杂度**：时间复杂度为 O(m)，其中 m 为输入字符串长度。空间复杂度为 O(m)，存储编码后的字符串。
* **原因**：通过查表法进行编码，效率高且实现简单。

#### 4. decodeHuffman 函数

string decodeHuffman(const unordered\_map<char, string>& huffmanCodes, string encodedString)

* **功能**：将 Huffman 编码的字符串解码为原始字符串。
* **算法**：遍历编码字符串，逐位拼接形成当前编码片段，在编码表中查找对应字符，找到后将字符加入结果字符串并重置当前编码片段。
* **复杂度**：时间复杂度为 O(m)，其中 m 为编码字符串长度。空间复杂度为 O(m)，存储解码后的字符串。
* **原因**：逐位查找编码表，保证解码过程高效准确。

#### 5. saveHuffmanCodesToJSON 和 loadHuffmanCodesFromJSON 函数

void saveHuffmanCodesToJSON(const unordered\_map<char, string>& huffmanCodes, const string& fileName)  
unordered\_map<char, string> loadHuffmanCodesFromJSON(const string& fileName)

* **功能**：将 Huffman 编码表保存到 JSON 文件中，或从 JSON 文件加载 Huffman 编码表。
* **算法**：遍历编码表，将每个字符及其对应编码保存到 JSON 对象中，并写入文件或从文件读取并解析为编码表。
* **复杂度**：时间复杂度为 O(n)，其中 n 为字符种类数。空间复杂度为 O(n)，存储编码表。
* **原因**：JSON 格式通用性好，适合存储和传输结构化数据。

#### 6. writeBinaryFile 和 readBinaryFile 函数

void writeBinaryFile(const string& fileName, const string& binaryData)  
string readBinaryFile(const string& fileName)

* **功能**：将二进制数据写入文件，或从文件读取二进制数据。
* **算法**：将二进制字符串转换为实际的二进制数据并写入文件，或从文件读取二进制数据并转换为二进制字符串。
* **复杂度**：时间复杂度为 O(m)，其中 m 为二进制数据长度。空间复杂度为 O(m)，存储二进制数据。
* **原因**：二进制文件存储效率高，适合保存编码后的数据。

#### 7. compressString 函数

void compressString(string inputString, int ID)

* **功能**：压缩输入字符串，生成 Huffman 编码表和编码后的二进制数据，并分别保存到 JSON 文件和二进制文件中。
* **算法**：统计字符频率，构建 Huffman 树，生成 Huffman 编码表，将输入字符串编码为 Huffman 编码，保存编码表和编码数据。
* **复杂度**：时间复杂度为 O(m log n)，其中 m 为输入字符串长度，n 为字符种类数。空间复杂度为 O(m + n)，存储编码表和编码数据。
* **原因**：Huffman 编码能有效压缩数据，减少存储空间。

#### 8. decompressString 函数

string decompressString(int ID)

* **功能**：解压缩 Huffman 编码的数据，从 JSON 文件和二进制文件中分别加载 Huffman 编码表和编码数据，解码得到原始字符串。
* **算法**：加载 Huffman 编码表和编码数据，将编码数据解码为原始字符串。
* **复杂度**：时间复杂度为 O(m)，其中 m 为编码数据长度。空间复杂度为 O(m)，存储解码后的字符串。
* **原因**：解压缩恢复原始数据，保证数据完整性。

### 总结

Huffman 编码模块通过合理的算法设计，实现了字符串数据的高效压缩和解压缩。该模块利用 Huffman 树、哈希表、递归遍历、查表法等技术，在处理大规模数据时表现出色，能够有效减少存储空间，提高数据传输效率。通过将编码表和编码数据分别存储在 JSON 文件和二进制文件中，实现了数据的持久化和高效存储。

## 游学推荐模块 destinationrecommend.cpp

### 游学推荐模块算法分析

在 destinationRecommend 类中，实现了游学推荐的相关功能，包括根据用户偏好推荐地点、搜索地点、按热度和评分排序等。以下是对该模块中各个算法的详细分析。

#### 1. 推荐功能

推荐功能主要通过 recommend\_mode 函数实现。

void destinationRecommend::recommend\_mode()

* **功能**：根据用户选择的类别推荐相关的景点。
* **算法**：首先根据用户选择的类别生成相应的关键词集合，然后遍历所有景点，通过 KMP 算法进行字符串匹配，找到符合关键词的景点。
* **复杂度**：关键词匹配为 O(n\*m)，其中 n 为景点数量，m 为关键词数量。
* **原因**：KMP 算法能够高效地处理字符串匹配问题，通过关键词匹配提高推荐的准确性和相关性。

#### 2. 排序功能

排序功能分为按热度和按评分进行排序，分别对应 sortAndPrintPopTop10 和 sortAndPrintRateTop10 函数。

##### 2.1 按热度排序

void destinationRecommend::sortAndPrintPopTop10(vector<int>& indices)

* **功能**：按热度从高到低对景点进行排序，并展示前十个热度最高的景点。
* **算法**：使用快速排序和选择排序进行 Top-K 问题的解决。
* **复杂度**：快速排序平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)，Top-K 选择排序为 O(n)。
* **原因**：快速排序在大多数情况下性能优越，结合选择排序能高效解决 Top-K 问题。

##### 2.2 按评分排序

void destinationRecommend::sortAndPrintRateTop10(vector<int>& indices)

* **功能**：按评分从高到低对景点进行排序，并展示前十个评分最高的景点。
* **算法**：与按热度排序类似，使用快速排序和选择排序。
* **复杂度**：快速排序平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)，Top-K 选择排序为 O(n)。
* **原因**：快速排序在大多数情况下性能优越，结合选择排序能高效解决 Top-K 问题。

#### 3. 搜索功能

搜索功能通过 search\_mode 函数实现。

void destinationRecommend::search\_mode()

* **功能**：根据用户输入的关键词搜索景点。
* **算法**：使用 KMP 算法进行字符串匹配，找到符合关键词的景点。
* **复杂度**：KMP 算法复杂度为 O(n\*m)，其中 n 为景点名称长度，m 为用户输入的关键词长度。
* **原因**：KMP 算法能够高效地进行字符串匹配，适合处理长文本的匹配问题。

#### 4. 排序算法

排序算法使用快速排序和选择排序来解决 Top-K 问题。

##### 4.1 快速排序和划分函数

void destinationRecommend::quickPopularityFullSort(vector<int>& arr, int low, int high)  
int destinationRecommend::partitionPopularity(vector<int>& arr, int low, int high)

* **功能**：实现快速排序，用于对景点按热度进行排序。
* **算法**：快速排序是一种分治算法，通过选择一个枢纽元素，将数组分为两部分，使得一部分所有元素大于枢纽元素，另一部分所有元素小于枢纽元素，然后递归排序。
* **复杂度**：平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)。
* **原因**：快速排序在大多数情况下性能优越，适合大规模数据排序。

##### 4.2 快速选择和划分函数

void destinationRecommend::findPopularityTopK(vector<int>& arr, int low, int high, int k)  
int destinationRecommend::partitionPopularity(vector<int>& arr, int low, int high)

* **功能**：找到前 K 个热度最高的景点。
* **算法**：使用快速选择算法，通过选择一个枢纽元素，将数组分为两部分，使得一部分所有元素大于枢纽元素，另一部分所有元素小于枢纽元素，然后递归选择。
* **复杂度**：平均时间复杂度为 O(n)，最坏情况为 O(n^2)。
* **原因**：快速选择算法在大多数情况下性能优越，适合解决 Top-K 问题。

### 性能分析

游学推荐模块通过使用高效的算法，实现了景点推荐、搜索和排序功能。KMP 算法用于字符串匹配，能够高效处理关键词匹配问题。快速排序和选择排序用于解决 Top-K 问题，能够快速找到前 K 个热度或评分最高的景点。KMP 算法的使用，提高了搜索和推荐的效率。

## 地图数据读取模块 myGraph.cpp

### 地图文件读取模块算法分析

在 myGraph 模块中，实现了从 JSON 文件中读取地图信息并构建图结构，主要功能包括读取景点信息、构建顶点和边、保存更新后的景点信息等。以下是对该模块中各个算法的详细分析。

### 1. 读取景点信息

#### 1.1 读取景点信息并构建图结构

void loadAttractionArray(string filename)

* **功能**：从 JSON 文件中读取景点信息并构建图结构。
* **算法思想**：
  + 打开并读取 JSON 文件内容，将其解析为 JSON 对象。
  + 依次遍历 JSON 数组中的每个景点对象，提取景点的基本属性（如名称、是否是学校、热度、评分等）。
  + 解析并构建景点的顶点和边信息，将其存储在图结构中。
* **算法特点**：
  + 使用 cJSON 库解析 JSON 文件，便于处理复杂的 JSON 数据结构。
  + 将景点信息存储在全局数组 attr 中，便于后续操作。
* **复杂度**：时间复杂度为 O(n + m)，其中 n 为景点数量，m 为边的数量。
* **原因**：逐个解析景点对象，并构建其顶点和边，确保地图信息的完整性和正确性。

#### 1.2 保存更新后的景点信息

void saveAttractionArray(string filename)

* **功能**：将更新后的景点信息保存回 JSON 文件。
* **算法思想**：
  + 打开并读取现有的 JSON 文件内容，将其解析为 JSON 对象。
  + 依次遍历全局数组 attr 中的每个景点对象，创建新的 JSON 对象，并添加更新后的景点信息。
  + 将新的 JSON 数据写入文件，覆盖原有的文件内容。
* **算法特点**：
  + 使用 cJSON 库创建和操作 JSON 对象，确保数据的正确性和一致性。
  + 将更新后的景点信息保存回 JSON 文件，便于后续操作。
* **复杂度**：时间复杂度为 O(n + m)，其中 n 为景点数量，m 为边的数量。
* **原因**：逐个处理景点对象，并将其更新后的信息写入文件，确保数据的持久化。

### 主要函数分析

#### 1. 加载景点信息

void loadAttractionArray(string filename)

* **功能**：从 JSON 文件中加载景点信息，并构建图结构。
* **算法实现**：
  + 打开 JSON 文件并读取其内容。
  + 使用 cJSON 库解析 JSON 数据，遍历 JSON 数组中的每个景点对象。
  + 提取景点的基本属性（如名称、热度、评分等），并存储在全局数组 attr 中。
  + 解析景点的顶点和边信息，构建图结构。
* **复杂度**：时间复杂度为 O(n + m)，其中 n 为景点数量，m 为边的数量。
* **原因**：确保从文件中正确读取景点信息，并构建完整的图结构，便于后续的路径规划和展示。

#### 2. 保存景点信息

void saveAttractionArray(string filename)

* **功能**：将更新后的景点信息保存回 JSON 文件。
* **算法实现**：
  + 打开现有的 JSON 文件并读取其内容。
  + 使用 cJSON 库解析 JSON 数据，遍历全局数组 attr 中的每个景点对象。
  + 创建新的 JSON 对象，并添加更新后的景点信息。
  + 将新的 JSON 数据写入文件，覆盖原有的文件内容。
* **复杂度**：时间复杂度为 O(n + m)，其中 n 为景点数量，m 为边的数量。
* **原因**：确保更新后的景点信息能够正确持久化，便于后续操作。

### 性能分析

在该模块中，主要使用了 cJSON 库来解析和操作 JSON 数据，通过遍历 JSON 数组和全局数组 attr，实现了景点信息的读取和保存功能。解析和保存 JSON 文件的时间复杂度主要取决于景点数量和边的数量，在实际应用中性能较为优越，能够高效处理大规模的景点信息数据。

### 总结

地图文件读取模块通过使用 cJSON 库实现了从 JSON 文件中读取景点信息并构建图结构，以及将更新后的景点信息保存回 JSON 文件的功能。通过详细的算法设计和实现，确保了系统的高效性和正确性，为后续的路径规划和展示功能提供了坚实的数据基础。

## 路线规划算法模块 route\_plan.cpp

### 路线规划模块算法分析

在 route\_plan 模块中，实现了多点和两点路线规划功能，包括基于距离和时间的路径规划、Floyd-Warshall 算法、Dijkstra 算法等。以下是对该模块中各个算法的详细分析。

### 1. 多点路线规划

多点路线规划通过 Multi\_point\_route\_planning 函数实现。

void Multi\_point\_route\_planning(ALGraph\* graph, bool isSchool)

* **功能**：规划从起点到多个目的地的最短路径。
* **算法**：用户输入起点和多个目的地的 ID，通过 Floyd-Warshall 算法预处理所有点之间的最短路径，然后使用全排列方式计算所有路径的总距离，找到最短路径。
* **复杂度**：Floyd-Warshall 算法复杂度为 O(n^3)，全排列复杂度为 O((n+1)!)。
* **原因**：Floyd-Warshall 算法适用于稠密图，全排列方式可以确保找到所有可能路径中的最短路径。

### 2. Floyd-Warshall 算法

#### 2.1 基于距离的 Floyd-Warshall 算法

void FloydByDist(ALGraph\* graph, int start, int num, int place[])

* **功能**：计算所有点之间的最短距离，并找到起点和多个目的地的最短路径。
* **算法思想**：
  + 初始化图的权重矩阵，对于没有直接边相连的点设置为无穷大。
  + 对于每一个顶点作为中间点，尝试更新所有顶点对之间的最短路径。如果通过中间点可以找到更短的路径，则更新路径和前驱节点。
* **算法特点**：
  + 适用于稠密图，能够计算所有点对之间的最短路径。
  + 使用动态规划的思想，通过不断迭代优化路径。
* **复杂度**：时间复杂度为 O(n^3)。
* **原因**：Floyd-Warshall 算法适用于计算稠密图中所有点对之间的最短路径。

#### 2.2 基于时间的 Floyd-Warshall 算法

void FloydByTime(ALGraph\* graph, int start, int num, int place[], int flag)

* **功能**：计算所有点之间的最短时间，并找到起点和多个目的地的最短路径。
* **算法思想**：
  + 初始化图的权重矩阵，根据交通工具（步行、自行车、电动车）不同，设置不同的拥堵系数。
  + 对于每一个顶点作为中间点，尝试更新所有顶点对之间的最短时间。如果通过中间点可以找到更短的时间路径，则更新路径和前驱节点。
* **算法特点**：
  + 适用于稠密图，能够计算所有点对之间的最短时间路径。
  + 使用动态规划的思想，通过不断迭代优化时间路径。
* **复杂度**：时间复杂度为 O(n^3)。
* **原因**：Floyd-Warshall 算法适用于计算稠密图中所有点对之间的最短路径。

### 3. Dijkstra 算法

#### 3.1 基于时间的 Dijkstra 算法

void DjikstraByTime(ALGraph\* graph, int start, int flag)

* **功能**：计算从起点到所有点的最短时间。
* **算法思想**：
  + 初始化所有顶点的最短时间为无穷大，起点的最短时间为0。
  + 使用贪心算法，每次选择未访问的最近节点，更新其邻居节点的最短时间。
  + 通过优先队列不断选择未访问的最近节点，更新其邻居节点的最短时间。
* **算法特点**：
  + 适用于计算单源最短路径，能够高效找到起点到所有点的最短时间。
  + 使用优先队列优化选择最近节点的效率。
* **复杂度**：时间复杂度为 O(n^2)。
* **原因**：Dijkstra 算法适用于计算单源最短路径，能够高效找到起点到所有点的最短时间。

#### 3.2 基于距离的 Dijkstra 算法

void DjikstraByDist(ALGraph\* graph, int start)

* **功能**：计算从起点到所有点的最短距离。
* **算法思想**：
  + 初始化所有顶点的最短距离为无穷大，起点的最短距离为0。
  + 使用贪心算法，每次选择未访问的最近节点，更新其邻居节点的最短距离。
  + 通过优先队列不断选择未访问的最近节点，更新其邻居节点的最短距离。
* **算法特点**：
  + 适用于计算单源最短路径，能够高效找到起点到所有点的最短距离。
  + 使用优先队列优化选择最近节点的效率。
* **复杂度**：时间复杂度为 O(n^2)。
* **原因**：Dijkstra 算法适用于计算单源最短路径，能够高效找到起点到所有点的最短距离。

### 4. 路径获取函数

#### 4.1 获取距离

int getdis(int num, int place[])

* **功能**：计算一条路径的总距离。
* **算法**：遍历路径上的所有点，累加它们之间的距离。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：遍历路径上的所有点可以准确计算路径的总距离。

#### 4.2 获取时间

int gettimelen(int num, int place[])

* **功能**：计算一条路径的总时间。
* **算法**：遍历路径上的所有点，累加它们之间的时间。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：遍历路径上的所有点可以准确计算路径的总时间。

#### 4.3 保存路径

void savpath(int place[], int num)

* **功能**：将一条路径保存到 anspath 数组中。
* **算法**：遍历路径上的所有点，将它们依次保存到 anspath 数组中。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：保存路径便于后续的路径展示和分析。

#### 4.4 获取路径

void getpath(int start, int end)

* **功能**：获取从起点到终点的路径，并保存到 anspath 数组中。
* **算法**：通过递归方式，找到起点到终点的所有中间点，将它们依次保存到 anspath 数组中。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：递归方式可以有效获取路径上的所有中间点，便于路径展示和分析。

### 5. 路径打印函数

#### 5.1 打印路径

void printPath(ALGraph\* graph, int start, int end)

* **功能**：打印从起点到终点的路径。
* **算法**：通过递归方式，找到起点到终点的所有中间点，依次打印它们。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：递归方式可以有效获取路径上的所有中间点，便于路径展示和分析。

#### 5.2 打印所有点和中间点

void printMidpoint(ALGraph\* graph, int start, int end)

* **功能**：打印从起点到终点的所有点和中间点。
* **算法**：通过链表存储路径上的所有点和中间点，依次打印它们。
* **复杂度**：时间复杂度为 O(n)。
* **原因**：链表可以有效存储路径上的所有点和中间点，便于路径展示和分析。

### 总结

路线规划模块通过使用 Floyd-Warshall 算法和 Dijkstra 算法，实现了多点和两点的路径规划功能。通过全排列方式解决多点路径的最短距离问题，使用递归和链表打印路径上的所有点和中间点，为用户提供了高效的路线规划和展示功能。通过详细的算法设计和实现，确保了系统的高效性和准确性。

## 路线规划/设施查询/美食推荐集成模块 mymap.cpp

### 路线规划

#### 1. 路线规划功能概述

void myMap::on\_routePlanBtn\_clicked()

* **功能**：实现两点路线规划、多点路线规划以及交通策略的路线规划。
* **算法思想**：
  + 检查用户是否选择了多点路线规划。
  + 根据用户选择的规划类别（最短距离或最短时间）和交通工具（步行、自行车或电瓶车）调用相应的路线规划算法。
* **算法特点**：
  + 使用Dijkstra算法和Floyd-Warshall算法实现最短路径计算。
  + 支持多点路线规划，结合回溯算法计算所有排列组合，找到最优路径。
* **复杂度**：
  + Dijkstra算法：O(V^2)，其中V是顶点数量。
  + Floyd-Warshall算法：O(V^3)，适用于全局最短路径计算。
  + 多点路线规划：O(n!)，因为需要计算所有排列组合。

#### 2. 两点路线规划

void DjikstraByDist(ALGraph\* graph, int start)

* **功能**：实现从起点到终点的最短距离路线规划。
* **算法实现**：
  + 使用Dijkstra算法计算从起点到所有其他顶点的最短距离。
  + 初始化所有顶点的距离为无穷大，起点的距离为0。
  + 重复以下步骤，直到所有顶点都被访问：
    - 找到未访问顶点中距离起点最近的顶点，将其标记为已访问。
    - 更新其所有邻接顶点的距离，如果通过该顶点到邻接顶点的距离更短，则更新邻接顶点的距离和前驱节点。
* **复杂度**：O(V^2)，适用于稠密图。

void DjikstraByTime(ALGraph\* graph, int start, int flag)

* **功能**：实现从起点到终点的最短时间路线规划，支持不同交通工具。
* **算法实现**：
  + 与Dijkstra最短距离算法类似，只是根据不同交通工具计算时间。
  + 初始化所有顶点的时间为无穷大，起点的时间为0。
  + 根据交通工具选择合适的拥堵系数，计算最短时间。
* **复杂度**：O(V^2)，适用于稠密图。

#### 3. 多点路线规划

void FloydByDist(ALGraph\* graph, int start, int num, int place[])

* **功能**：实现从起点经过多个中间点再返回起点的最短距离路线规划。
* **算法实现**：
  + 使用Floyd-Warshall算法计算所有顶点间的最短距离。
  + 初始化m\_dist和m\_prevNode数组，m\_disti表示顶点i到顶点j的最短距离，m\_prevNodei表示路径中从i到j的前驱节点。
  + 遍历所有顶点的所有边，更新m\_dist和m\_prevNode数组。
  + 通过回溯算法计算所有可能的排列组合，找到最优路径。
* **复杂度**：O(V^3)（Floyd-Warshall） + O(n!)（多点路径），适用于较小规模的多点路线规划。

void FloydByTime(ALGraph\* graph, int start, int num, int place[], int flag)

* **功能**：实现从起点经过多个中间点再返回起点的最短时间路线规划，支持不同交通工具。
* **算法实现**：
  + 与FloydByDist类似，只是根据不同交通工具计算时间。
  + 初始化m\_timelen数组，m\_timeleni表示顶点i到顶点j的最短时间。
  + 根据交通工具选择合适的拥堵系数，计算最短时间。
* **复杂度**：O(V^3)（Floyd-Warshall） + O(n!)（多点路径），适用于较小规模的多点路线规划。

#### 4. 辅助函数

int getdis(int num, int place[])

* **功能**：计算多点路线的总距离。
* **算法实现**：
  + 遍历所有顶点，累加路径总距离。
  + 如果某段路径不存在，返回无穷大。
* **复杂度**：O(n)，其中n为中间点的数量。

int gettimelen(int num, int place[])

* **功能**：计算多点路线的总时间。
* **算法实现**：
  + 与getdis类似，只是计算时间。
* **复杂度**：O(n)，其中n为中间点的数量。

void savpath(int place[], int num)

* **功能**：保存多点路线的最优路径。
* **算法实现**：
  + 遍历所有顶点，保存路径。
* **复杂度**：O(n)，其中n为中间点的数量。

void getpath(int start, int end)

* **功能**：获取从起点到终点的路径。
* **算法实现**：
  + 递归调用，获取路径。
* **复杂度**：O(n)，其中n为路径中的顶点数量。

#### 5. 打印路径和中间点

void printPath(ALGraph\* graph, int start, int end)

* **功能**：打印从起点到终点的路径。
* **算法实现**：
  + 递归调用，打印路径中的每个顶点。
* **复杂度**：O(n)，其中n为路径中的顶点数量。

void printMidpoint(ALGraph\* graph, int start, int end)

* **功能**：打印路径中的所有中间点。
* **算法实现**：
  + 递归调用，打印路径中的每个中间点。
* **复杂度**：O(n)，其中n为路径中的中间点数量。

### 设施查询

#### 1. 设施查询功能概述

void myMap::on\_facilitySearch\_clicked()

* **功能**：根据用户输入的设施名称和排序条件进行设施查询。
* **算法思想**：
  + 使用KMP算法进行字符串匹配，查找设施名称。
  + 使用快速排序对匹配结果按距离、类别等条件排序。
* **算法特点**：
  + 使用KMP算法提高字符串匹配效率。
  + 快速排序保证查询结果按用户选择的排序条件排序。
* **复杂度**：
  + KMP算法：O(n + m)，其中n为文本长度，m为模式串长度。
  + 快速排序：平均时间复杂度O(n log n)。

#### 2. 快速排序

void myMap::quickSort(vector<int> &indices, int low, int high)

* **功能**：对设施查询结果按距离排序。
* **算法实现**：
  + 使用快速排序算法，对设施查询结果按距离排序。
  + 选择枢轴，将元素划分为小于枢轴和大于枢轴的两部分，递归排序。
* **复杂度**：平均时间复杂度O(n log n)，最坏情况下为O(n^2)。

### 美食推荐

#### 1. 美食推荐功能概述

void myMap::on\_foodSearch\_clicked()

* **功能**：根据用户输入的美食名称和排序条件进行美食推荐。
* **算法思想**：
  + 使用KMP算法进行字符串匹配，查找美食名称。
  + 根据用户选择的排序条件（按热度、按距离、按类别、按餐厅）进行排序和筛选。
* **算法特点**：
  + 使用KMP算法提高字符串匹配效率。
  + 使用快速排序和选择排序对匹配结果按不同条件排序。
* **复杂度**：
  + KMP算法：O(n + m)，其中n为文本长度，m为模式串长度。
  + 快速排序：平均时间复杂度O(n log n)。
  + 选择排序：O(n^2)。

#### 2. 热度排序

void myMap::findPopularityTopK(vector<int> &arr, int low, int high, int k, int selectedSpotIndex, int selectedBuildIndex)

* **功能**：对美食查询结果按热度排序。
* **算法实现**：
  + 使用快速选择算法找到前K个热度最高的美食。
  + 使用快速排序对前K个美食进行排序。
* **复杂度**：快速选择算法O(n)，快速排序平均时间复杂度O(n log n)。

#### 3. 距离排序

void myMap::findDistanceTopK(vector<int> &arr, int low, int high, int k, int selectedSpotIndex, int selectedBuildIndex)

* **功能**：对美食查询结果按距离排序

。

* **算法实现**：
  + 使用快速选择算法找到前K个距离最近的美食。
  + 使用快速排序对前K个美食进行排序。
* **复杂度**：快速选择算法O(n)，快速排序平均时间复杂度O(n log n)。

### 其他辅助函数

bool myMap::exactMatch(const string &text, const string &pattern)

* **功能**：实现字符串的精确匹配。
* **算法实现**：
  + 比较字符串的每个字符，如果所有字符都匹配，则返回true。
* **复杂度**：O(n)，其中n为字符串长度。

总结来看，这些函数共同实现了一个复杂的导航和查询系统，涵盖了路线规划、设施查询和美食推荐等功能。通过合理使用各种算法，如Dijkstra算法、Floyd-Warshall算法、KMP算法、快速排序和快速选择，系统能够高效地处理大规模数据，并提供良好的用户体验。

## 图像化导航 myLabel 类功能分析

### 1. 类概述

myLabel 类继承自 QLabel，用于在地图上绘制路径。该类的主要功能包括加载地图图像、绘制单点路径和多点路径，以及处理绘制事件。

### 2. 构造函数 myLabel

myLabel::myLabel(QWidget \*parent)  
 :QLabel(parent)  
{  
 m\_Img = QPixmap("./bupt\_map.bmp");  
 setPixmap(m\_Img);  
 setGeometry(10, 10, m\_Img.width(), m\_Img.height());  
 m\_flag = 0;  
}

* **功能**：构造函数用于初始化 myLabel 对象，加载地图图像，并设置标签的大小和位置。
* **初始化**：
  + 加载地图图像 bupt\_map.bmp。
  + 设置图像为标签的显示内容。
  + 设置标签的位置和大小。
  + 初始化标志变量 m\_flag 为 0，表示初始状态不绘制路径。

### 3. 重载 paintEvent 函数

void myLabel::paintEvent(QPaintEvent \*e)  
{  
 QLabel::paintEvent(e);  
 if(m\_flag == 1) PaintRoute(m\_start, m\_end);  
 if(m\_flag == 2) multiRoutePaint();  
}

* **功能**：重载 paintEvent 函数，用于处理绘制事件。
* **绘制逻辑**：
  + 调用父类的 paintEvent 函数。
  + 根据 m\_flag 的值决定绘制单点路径或多点路径：
    - m\_flag == 1：调用 PaintRoute 函数绘制单点路径。
    - m\_flag == 2：调用 multiRoutePaint 函数绘制多点路径。

### 4. routePaint 函数

void myLabel::routePaint(Attraction a, int flag, int start, int end)  
{  
 m\_attract = a;  
 m\_flag = flag;  
 m\_start = start;  
 m\_end = end;  
 update();  
}

* **功能**：设置绘制路径的相关参数，并更新绘图区域。
* **参数设置**：
  + m\_attract：设置当前景点对象。
  + m\_flag：设置绘制标志（1 表示单点路径，2 表示多点路径）。
  + m\_start 和 m\_end：设置起点和终点索引。
  + 调用 update 函数触发重绘事件。

### 5. PaintRoute 函数

void myLabel::PaintRoute(int start, int end);

* **功能**：递归绘制从起点到终点的单点路径。
* **绘制逻辑**：
  + 递归调用 PaintRoute 绘制从起点到终点的所有路径段。
  + 计算起点、终点和中间点的坐标，并转换为像素值。
  + 使用 QPainter 绘制路径段，并在路径终点绘制箭头。

### 6. multiRoutePaint 函数

void myLabel::multiRoutePaint();

* **功能**：绘制多点路径，包括所有路径段和箭头。
* **绘制逻辑**：
  + 遍历路径中的所有顶点，绘制每段路径。
  + 计算起点、终点和中间点的坐标，并转换为像素值。
  + 使用 QPainter 绘制路径段，并在路径终点绘制箭头。

### 总结

* myLabel 类的主要功能是加载地图图像并在地图上绘制路径。
* 通过重载 paintEvent 函数实现路径的动态绘制。
* 提供 PaintRoute 和 multiRoutePaint 函数分别用于单点路径和多点路径的绘制。
* 使用 QPainter 类实现路径和箭头的绘制，确保路径在地图上的准确显示。

## 日记管理模块 diarymanagement.cpp

### 日记管理模块算法分析

在 diaryManagement 类中，实现了各种关于日记管理的功能，包括解析 JSON 文件、搜索日记、按不同标准排序等。以下是对该模块中各个算法的详细分析。

#### 1. 解析日记内容

std::vector<Diary> diaryManagement::parseDiariesContents(cJSON \*diaries)

* **功能**：将 JSON 文件中的日记内容解析为 Diary 结构体，并存储在 std::vector<Diary> 中。
* **算法**：遍历 JSON 数组，逐个提取日记条目的各个字段并填充到 Diary 结构体中。
* **复杂度**：O(n)，其中 n 为 JSON 文件中日记条目的数量。
* **原因**：解析 JSON 文件并转换为结构体便于后续处理和操作。

#### 2. 搜索功能

搜索功能分为按地点、标题和内容进行搜索，分别对应 searchByDest、searchByTitle 和 searchByContent 函数。

##### 2.1 按地点搜索

void diaryManagement::searchByDest()

* **功能**：根据用户输入的地点进行日记的查找。
* **算法**：使用哈希表将日记按照地点哈希值进行分组，再根据用户输入的地点哈希值查找匹配的日记。
* **复杂度**：哈希表构建为 O(n)，查找为 O(1)。
* **原因**：哈希表查找速度快，能高效处理大量数据。

##### 2.2 按标题搜索

void diaryManagement::searchByTitle()

* **功能**：根据用户输入的标题进行日记的查找。
* **算法**：与按地点搜索类似，使用哈希表将日记按照标题哈希值进行分组，再根据用户输入的标题哈希值查找匹配的日记。
* **复杂度**：哈希表构建为 O(n)，查找为 O(1)。
* **原因**：哈希表查找速度快，能高效处理大量数据。

##### 2.3 按内容搜索

void diaryManagement::searchByContent()

* **功能**：根据用户输入的内容进行日记的查找。
* **算法**：使用 KMP 算法进行字符串匹配查找。
* **复杂度**：KMP 算法复杂度为 O(m+n)，其中 m 为日记内容长度，n 为用户输入字符串长度。
* **原因**：KMP 算法适合处理长文本的字符串匹配问题，能高效地进行部分匹配。

#### 3. 排序功能

排序功能分为按评分、按人气和按兴趣进行排序，分别对应 browseByRating、browseByPopularity 和 browseByInterest 函数。

##### 3.1 按评分排序

void diaryManagement::browseByRating(std::vector<Diary> &matchedDiary)

* **功能**：按评分从高到低对日记进行排序，并展示前十条评分最高的日记。
* **算法**：使用快速排序进行排序，并结合选择排序进行 Top-K 问题的解决。
* **复杂度**：快速排序平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)，Top-K 选择排序为 O(n)。
* **原因**：快速排序是一种高效的排序算法，结合选择排序能高效解决 Top-K 问题。

##### 3.2 按人气排序

void diaryManagement::browseByPopularity(std::vector<Diary> matchedDiary)

* **功能**：按人气从高到低对日记进行排序，并展示前十条人气最高的日记。
* **算法**：与按评分排序类似，使用快速排序和选择排序。
* **复杂度**：快速排序平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)，Top-K 选择排序为 O(n)。
* **原因**：快速排序是一种高效的排序算法，结合选择排序能高效解决 Top-K 问题。

##### 3.3 按兴趣排序

void diaryManagement::browseByInterest(std::vector<Diary> sortedDiaries)

* **功能**：根据用户选择的兴趣标签筛选日记，并按人气排序展示前十条日记。
* **算法**：首先根据兴趣标签筛选日记，然后使用快速排序和选择排序进行排序。
* **复杂度**：筛选复杂度为 O(n)，快速排序平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)，Top-K 选择排序为 O(n)。
* **原因**：快速排序是一种高效的排序算法，结合选择排序能高效解决 Top-K 问题，兴趣标签筛选能提高用户体验。

### 主要函数分析

#### 1. quickSort 和 partition 函数

void diaryManagement::quickRatingSort(std::vector<Diary>& arr, int low, int high)  
int diaryManagement::partitionRating(std::vector<Diary>& arr, int low, int high)

* **功能**：实现快速排序，用于对日记按评分进行排序。
* **算法**：快速排序是一种分治算法，通过选择一个枢纽元素，将数组分为两部分，使得一部分所有元素小于枢纽元素，另一部分所有元素大于枢纽元素，然后递归排序。
* **复杂度**：平均时间复杂度为 O(n log n)，最坏情况为 O(n^2)。
* **原因**：快速排序在多数情况下性能优越，适合大规模数据排序。

#### 2. findTopK 和 partition 函数

void diaryManagement::findRatingTopK(std::vector<Diary>& arr, int low, int high, int k)  
int diaryManagement::partitionRating(std::vector<Diary>& arr, int low, int high)

* **功能**：找到前 K 个评分最高的日记。
* **算法**：使用快速选择算法，通过选择一个枢纽元素，将数组分为两部分，使得一部分所有元素大于枢纽元素，另一部分所有元素小于枢纽元素，然后递归选择。
* **复杂度**：平均时间复杂度为 O(n)，最坏情况为 O(n^2)。
* **原因**：快速选择算法在多数情况下性能优越，适合解决 Top-K 问题。

#### 3. 哈希函数 complexHashFunction

size\_t diaryManagement::complexHashFunction(const string& str)

* **功能**：对输入字符串进行哈希，生成哈希值。
* **算法**：使用 DJB 哈希算法，对字符串中的每个字符进行哈希运算，生成哈希值。
* **复杂度**：时间复杂度为 O(m)，其中 m 为字符串长度。DJB 哈希算法简单高效，冲突率低，适用于大规模字符串哈希。
* **原因**：DJB 哈希算法具有较好的均匀分布性，适用于快速查找和去重。

### 性能分析

在该模块中，主要使用了快速排序和选择排序来处理日记的排序和筛选问题，这两种算法在大多数情况下性能优越，适合处理大规模数据。哈希表用于地点和标题的搜索，可以实现常数时间的查找，大大提高了搜索效率。KMP 算法用于内容的字符串匹配，能够高效处理长文本的匹配问题。

### 总结

日记管理模块通过合理的数据结构和高效的算法，实现了日记的解析、搜索和排序功能。使用快速排序和选择排序解决排序和 Top-K 问题，通过哈希表和 KMP 算法提高搜索效率，为用户提供了高效的日记管理和浏览体验。

## 日记创建模块 diarycreate.cpp

### 日记创建模块分析

DiaryCreate 模块主要用于创建新日记并将其保存到 JSON 文件中。该模块通过用户输入的日记信息构建新的日记条目，并调用 Huffman 编码函数对日记内容进行压缩。以下是对该模块中各个算法和函数的详细分析。

#### 主要函数分析

#### 1. on\_confirm\_clicked 函数

void DiaryCreate::on\_confirm\_clicked()

* **功能**：用户确认创建日记后，读取现有日记文件，生成新的日记条目，压缩日记内容，并将新日记保存到文件中。
* **算法**：
  + 读取 JSON 文件：将文件内容解析为 cJSON 对象。
  + 解析现有日记：调用 parseDiariesContents 函数，将 JSON 数据转换为 Diary 对象的向量。
  + 构建新日记条目：根据用户输入填充 Diary 结构体。
  + 压缩日记内容：调用 compressString 函数对日记内容进行 Huffman 编码。
  + 保存新日记：调用 saveDiary 函数，将新日记添加到 JSON 文件中。
* **复杂度**：
  + 读取 JSON 文件和解析日记：时间复杂度为 O(n)，其中 n 为日记条目数。
  + 构建新日记条目和压缩内容：时间复杂度为 O(m log m)，其中 m 为日记内容长度。
  + 保存新日记：时间复杂度为 O(n)，需要重新写入整个 JSON 文件。
* **原因**：通过上述步骤保证了新日记的正确创建和保存，使用 Huffman 编码压缩内容，提高存储效率。

#### 2. parseDiariesContents 函数

std::vector<Diary> DiaryCreate::parseDiariesContents(cJSON \*diaries)

* **功能**：将 JSON 数据解析为 Diary 对象的向量。
* **算法**：
  + 解析 JSON 数据：通过 cJSON 库解析日记数组。
  + 转换为 Diary 对象：遍历 JSON 数组，将每个日记条目转换为 Diary 结构体，并调用 decompressString 函数解压日记内容。
* **复杂度**：时间复杂度为 O(n)，其中 n 为日记条目数。空间复杂度为 O(n)，存储解析后的 Diary 对象。
* **原因**：将 JSON 数据转换为 Diary 对象便于后续处理和操作。

#### 3. saveDiary 函数

void DiaryCreate::saveDiary(const Diary& diary)

* **功能**：将新的日记条目保存到 JSON 文件中。
* **算法**：
  + 读取现有 JSON 文件：将文件内容解析为 cJSON 对象。
  + 添加新日记条目：构建新的 cJSON 对象表示日记条目，并添加到日记数组中。
  + 写入 JSON 文件：将更新后的 JSON 对象转换为字符串，并写入文件。
* **复杂度**：
  + 读取和解析 JSON 文件：时间复杂度为 O(n)，其中 n 为日记条目数。
  + 添加新日记条目和写入文件：时间复杂度为 O(n)，需要重新写入整个 JSON 文件。
* **原因**：通过读取、更新和重写 JSON 文件，保证了新日记的正确保存，确保数据的一致性和完整性。

#### 4. compressString 和 decompressString 函数

void compressString(string inputString, int ID)  
string decompressString(int ID)

* **功能**：对日记内容进行压缩和解压缩。
* **算法**：
  + 压缩：统计字符频率，构建 Huffman 树，生成 Huffman 编码表，对字符串进行编码，保存编码表和编码后的二进制数据。
  + 解压缩：加载 Huffman 编码表，读取二进制数据，解码得到原始字符串。
* **复杂度**：
  + 压缩：时间复杂度为 O(m log m)，其中 m 为输入字符串长度。空间复杂度为 O(m)，存储编码表和编码数据。
  + 解压缩：时间复杂度为 O(m)，其中 m 为编码数据长度。空间复杂度为 O(m)，存储解码后的字符串。
* **原因**：Huffman 编码能够有效压缩数据，减少存储空间，提高数据传输效率。

### 总结

DiaryCreate 模块通过用户输入创建新的日记条目，并利用 Huffman 编码对日记内容进行压缩，保证了数据的高效存储和传输。该模块设计合理，算法选择得当，能够处理大规模日记数据，确保数据的一致性和完整性。通过对 JSON 文件的读取、更新和重写，实现了日记的持久化存储，为用户提供了便捷的日记管理功能。

## 日记显示模块 diaryview.cpp

### DiaryView 模块分析

DiaryView 模块用于显示和更新日记内容，包括显示日记标题、目的地和内容，以及用户对日记的评分和人气度的更新。以下是对该模块中各个算法和函数的详细分析。

#### 主要函数分析

#### 1. 构造函数 DiaryView

DiaryView::DiaryView(QWidget \*parent, std::vector<Diary> diaries, int id)

* **功能**：初始化 DiaryView 界面，并加载指定 ID 的日记内容。
* **实现**：
  + 初始化界面元素，设置标题、目的地和内容的显示。
  + 通过传入的日记向量 diaries 和日记 ID id，加载对应日记的详细信息。
  + 将日记内容设置到界面元素中。
* **复杂度**：时间复杂度为 O(1)，因为直接访问向量中的元素，设置界面元素的值。

#### 2. 析构函数 ~DiaryView

DiaryView::~DiaryView()

* **功能**：释放 DiaryView 对象占用的资源。
* **实现**：
  + 删除 UI 对象，释放内存。
* **复杂度**：时间复杂度为 O(1)。

#### 3. on\_confirmBnt\_clicked 函数

void DiaryView::on\_confirmBnt\_clicked()

* **功能**：用户确认对日记的评分后，更新日记的人气和评分，并保存到 JSON 文件中。
* **实现**：
  + 获取用户选择的评分，并计算新的评分和人气。
  + 读取 JSON 文件并解析为 cJSON 对象。
  + 查找并更新指定 ID 的日记条目的评分和人气。
  + 将更新后的 JSON 对象写回文件。
* **复杂度**：
  + 读取和解析 JSON 文件：时间复杂度为 O(n)，其中 n 为日记条目数。
  + 查找并更新指定 ID 的日记条目：时间复杂度为 O(n)，需要遍历 JSON 数组查找条目。
  + 写入更新后的 JSON 文件：时间复杂度为 O(n)。
* **原因**：通过读取、更新和写回 JSON 文件，保证了日记评分和人气的持久化存储，确保数据的一致性和完整性。

### 算法分析

#### 1. 获取评分和更新日记

在 on\_confirmBnt\_clicked 函数中，用户选择的评分通过单选按钮获取，然后更新日记的评分和人气。

* **计算新的评分**：
  + 使用加权平均的方法计算新的评分：
  + double rating = ((m\_diaries[m\_id].rating \* m\_diaries[m\_id].popularity) + r) / popularity;
  + 复杂度为 O(1)。

#### 2. 读取和解析 JSON 文件

* **读取 JSON 文件**：
  + 使用 QFile 类读取 JSON 文件内容，并通过 QTextStream 将文件内容读取为字符串。
  + 复杂度为 O(n)，其中 n 为文件大小。
* **解析 JSON 文件**：
  + 使用 cJSON 库将 JSON 字符串解析为 cJSON 对象。
  + 复杂度为 O(n)，其中 n 为 JSON 字符串长度。

#### 3. 查找并更新日记条目

* **查找指定 ID 的日记条目**：
  + 遍历 cJSON 对象中的日记数组，查找 ID 匹配的条目。
  + 复杂度为 O(n)，其中 n 为日记条目数。
* **更新日记条目**：
  + 更新匹配条目的评分和人气。
  + 复杂度为 O(1)。

#### 4. 写回 JSON 文件

* **将 cJSON 对象转换为字符串**：
  + 使用 cJSON\_Print 将 cJSON 对象转换为 JSON 字符串。
  + 复杂度为 O(n)，其中 n 为 JSON 对象的大小。
* **写入 JSON 文件**：
  + 使用 QFile 类将 JSON 字符串写回文件。
  + 复杂度为 O(n)，其中 n 为文件大小。

### 总结

DiaryView 模块通过界面显示日记内容，并允许用户对日记进行评分和人气更新。模块中使用了高效的 JSON 解析和操作库 cJSON，确保了数据的一致性和完整性。通过快速的加权平均计算和哈希匹配算法，保证了评分和人气更新的效率。整体设计合理，算法选择得当，能够高效处理日记数据的显示和更新操作。

## 模块调用关系分析

### 项目模块调用关系总结

#### 1. 用户登录和管理模块（login.cpp）

* **功能**：处理用户登录和账户管理。
* **调用关系**：
  + FunctionSelectPage：成功登录后，调用FunctionSelectPage以选择功能。

#### 2. 功能选择界面模块（functionselectpage.cpp）

* **功能**：用户在登录成功后选择要执行的功能。
* **调用关系**：
  + destinationRecommend：选择游学推荐功能后，调用destinationRecommend。
  + diaryManagement：选择日记管理功能后，调用diaryManagement。

#### 3. 日记管理模块（diarymanagement.cpp）

* **功能**：实现日记的创建、管理、排序和搜索等功能。
* **调用关系**：
  + parseDiariesContents：解析 JSON 文件中的日记内容。
  + searchByDest、searchByTitle、searchByContent：提供按地点、标题和内容搜索日记的功能。
  + browseByRating、browseByPopularity、browseByInterest：提供按评分、人气和兴趣排序日记的功能。
  + quickRatingSort 和 partitionRating：用于按评分排序。
  + findRatingTopK 和 partitionRating：用于找到评分前 K 名的日记。

#### 4. 游学推荐模块（destinationrecommend.cpp）

* **功能**：根据用户选择的类别和排序方式推荐景点。
* **调用关系**：
  + recommend\_mode：推荐模式，根据类别推荐景点。
  + search\_mode：搜索模式，根据用户输入搜索景点。
  + sortAndPrintPopTop10 和 sortAndPrintRateTop10：按热度和评分排序推荐景点。
  + findPopularityTopK 和 partitionPopularity：用于找到前 K 名的热门景点。
  + findRatingTopK 和 partitionRating：用于找到评分前 K 名的景点。

#### 5. 路线规划模块（route\_plan.cpp）

* **功能**：提供两点路线规划、多点路线规划以及根据交通工具选择最优路线的功能。
* **调用关系**：
  + Multi\_point\_route\_planning 和 two\_point\_route\_planning：多点和两点路线规划功能。
  + FloydByDist 和 FloydByTime：分别基于距离和时间的弗洛伊德算法。
  + DjikstraByDist 和 DjikstraByTime：分别基于距离和时间的迪杰斯特拉算法。
  + printPath 和 printMidpoint：打印最优路径及中间节点信息。

#### 6. 地图文件读取模块（myGraph.cpp）

* **功能**：读取和解析地图 JSON 文件，将景点及其属性加载到程序中。
* **调用关系**：
  + loadAttractionArray：从 JSON 文件加载景点数据。
  + saveAttractionArray：将景点数据保存到 JSON 文件。

#### 7. 地图显示模块（mymap.cpp）

* **功能**：显示地图及规划路线，并提供设施和美食的搜索与推荐功能。
* **调用关系**：
  + on\_routePlanBtn\_clicked：根据用户选择的起点、终点及规划模式调用DjikstraByDist、DjikstraByTime、FloydByDist或FloydByTime。
  + on\_editRouteBtn\_clicked：调用EditMultiPoint进行多点路线编辑。
  + on\_facilitySearch\_clicked：调用DjikstraByDist进行设施搜索，并按距离排序设施。
  + on\_foodSearch\_clicked：调用findPopularityTopK、quickPopularityFullSort、findDistanceTopK或quickDistanceFullSort进行美食推荐和排序。

#### 8. 自定义标签显示模块（myLabel.cpp）

* **功能**：显示地图及绘制规划路线。
* **调用关系**：
  + paintEvent：调用PaintRoute或multiRoutePaint绘制单条或多条路线。
  + routePaint：根据传入参数调用PaintRoute或multiRoutePaint。

### 总结

该项目由多个模块组成，分别负责用户登录、功能选择、日记管理、游学推荐、路线规划、地图文件读取、地图显示和自定义标签显示等功能。各模块之间通过函数调用相互交互，形成完整的功能链条。例如，用户登录成功后，进入功能选择界面，用户选择不同的功能后，调用相应的模块进行处理。各模块内部也通过函数调用完成具体的功能实现，如路线规划模块调用不同的算法函数实现路线规划。