北京邮电大学课程设计报告

<table>
<tr><td>课 程 设
计 名 称</td><td colspan="2">数据结构
课程设计</td><td>学 院</td><td>计 算 机</td><td>指导教师</td><td>郭岗</td></tr>
<tr><td>班 级</td><td>班内序号</td><td colspan="2">学 号</td><td>学生姓名</td><td colspan="2">分 工</td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211683</td><td>张晨阳</td><td colspan="2">数据：爬取全国景区、校园数据；
后端：实现路线规划、场所查询模块；整合所有模块；
前端：实现所有模块前后端交互的逻辑；
报告：完成部分报告；</td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211637</td><td>廖轩毅</td><td colspan="2">后端：实现游学推荐模块、游学日记模块部分内容；
前端：美化前端界面；
报告：完成部分报告；</td></tr>
<tr><td>2022211305</td><td></td><td colspan="2">2022211644</td><td>徐路</td><td colspan="2">数据：根据真实地图构建建筑、道路信息；
后端：实现登录模块；实现日记压缩下载解压功能；
报告：完成部分报告；</td></tr>
<tr><td>课程设计内容</td><td colspan="6">1、 基本内容：开发一款学生游学系统，以满足学生利用假期进行游学活动的管理需求。该系统旨在为学生提供便捷的游学活动管理功能，包括游学推荐、游学路线规划、场所查询以及游学日记管理等核心功能。通过该系统，学生可以更好地组织和记录自己的游学经历，提开游学活动的效率和体验。
2、 设计方法：小组合作采用 C++、Vue.js3.0 等主流语言进行系统开发。
3、 项目成果：实现游学系统各模块的的全部基本功能，并在此基础上，使用模拟退火算法优化了多途径点的求解算法，实现了用户友好的全图形化界面，实现了不同交通工具选择的选做功能。</td></tr>
</table>

学　生课程设计报　告 （附　页）	
课程设计成绩评定	遵照实践教学大纲并根据以下四方面综合评定成绩： 1、课程设计目的任务明确，选题符合教学要求，份量及难易程度 2、团队分工是否恰当与合理 3、综合运用所学知识，提高分析问题、解决问题及实践动手能力的效果 4、是否认真、独立完成属于自己的课程设计内容，课程设计报告是否思路清晰、文字通顺、书写规范 **评语**: **成绩:** 指导教师签名： 2024 年　月　日

注：评语要体现每个学生的工作情况，可以加页。

目录

1.项目概述

1.1. 项目背景

本项目旨在开发一款学生游学系统，以满足学生利用假期进行游学活动的管理需求。

该系统旨在为学生提供便捷的游学活动管理功能，包括游学推荐、游学路线规划、场所查询以及游学日记管理等核心功能。

通过该系统，学生可以更好地组织和记录自己的游学经历，提开游学活动的效率和体验。

1.2. 项目需求说明

实现一个具备游学推荐、游学路线规划、场所查询、游学日记管理等功能的学生游学系统，具体需求如下：

1) 游学前准备：

游学推荐：根据游学热度、评价和个人兴趣推荐游学目的地。

游学查询：输入名称、类别、关键字等信息来查询景点和学校；

2) 游学中体验：

参观线路规划：在校园和景点内部规划最优参观路线。

景点介绍与场所查询：在游览过程中提供景点介绍和场所信息查询。

3) 游学后回顾：

游学日记管理：根据照片和游览经历生成游学日记。

游学日记下载：游学日记数据应进行无损压缩存储，以优化存储空间。

2.系统架构

2.1. 开发环境与工具

2.1.1.操作系统

Windows 11.

2.1.2.集成开发工具

Visual Studio Code 1.90

在前后端调试过程中，前后端的服务需要分开运行，而 VSCode 作为一个文本编辑器，可以使多种不同文件出现在一起管理，且可以多开终端进入不同文件夹启动服务，作为本项目的开发工具十分合适。

2.1.3.Node.js 环境

使用 npm（Node Package Manager）来管理项目依赖和自动化工作流是现代前端和 Node.js 应用开发的标准做法。

2.1.4.Express.js 框架

Express.js 是一个基于 Node.js 平台的极简且灵活的 Web 应用开发框架，它使得 Web 服务器的搭建和 API 路由的设计变得快速而简单。

2.1.5.MySQL 8.0

本项目选择 MySQL8.0 作为数据库系统，因其安装配置简单，性能优越。

2.2. 编程语言选择

2.2.1.C++17

C++ 作为本项目后端开发的主要语言，用于实现各模块的主要功能、算法、数据结构，以及与数据库的连接。

本项目使用了 C++17 的 template、<auto>、模板参数等新特性。

2.2.2.JavaScript

JavaScript 在本项目中扮演了服务器端编程语言的角色，通过 Node.js 平台实现。

在本项目中，负责为前端应用程序提供数据接口和逻辑处理能力；与后端程序交互，调用编译好的 C++程序，从而实现前后端的解耦和功能的专业分工。

2.2.3.Vue 3.0

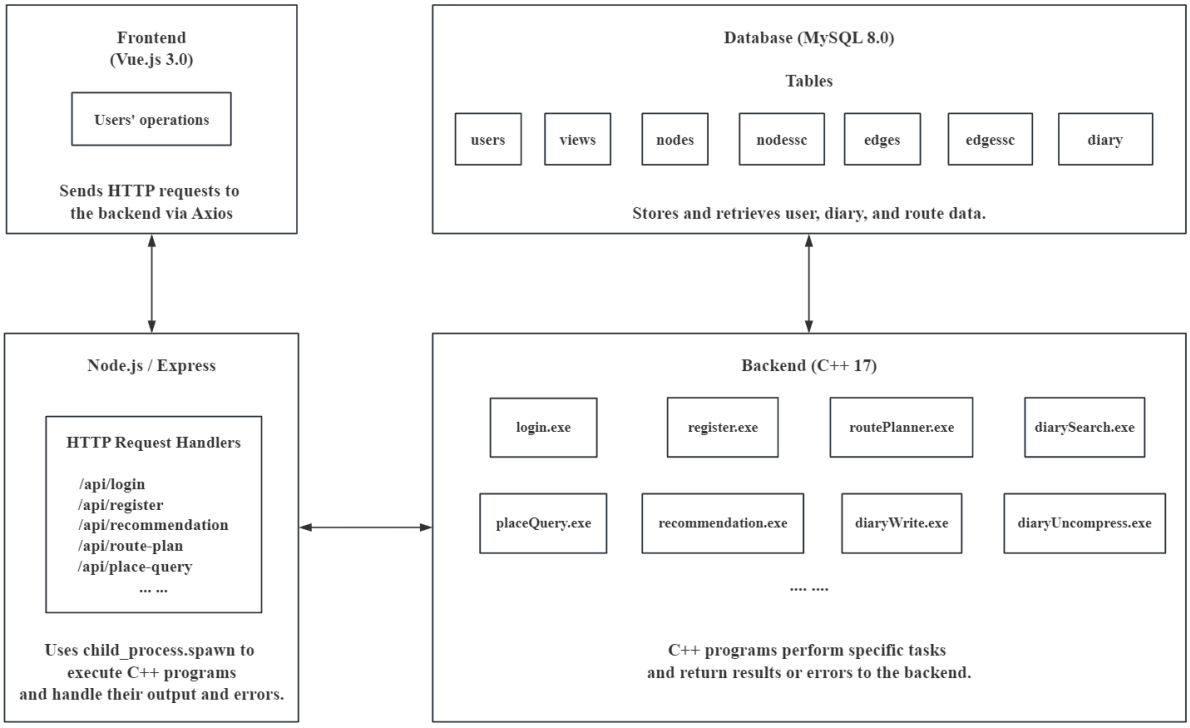
Vue 3.0 在本项目中扮演的角色是构建用户界面（UI）的前端 JavaScript 框架。

在本项目中，Vue 3.0 提供了一个强大、灵活且高效的前端开发解决方案，用于构建交互性强、用户友好的 Web 应用。

2.3. 系统架构设计

2.3.1.系统完整架构

系统的完整架构图如下：



在本系统中，前端采用 Vue.js 3.0 技术开发，为用户提供直观的操作界面。

用户在前端界面上执行操作，例如登录、注册、搜索游学日记等，这些操作会触发前端通过 Axios 库发送 HTTP 请求到后端服务器。

后端服务器基于 Node.js 和 Express.js 构建，专门负责处理这些 HTTP 请求。

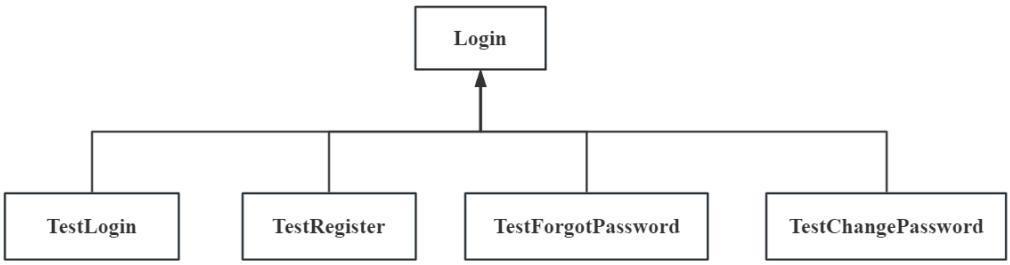
后端服务器接收到前端的请求后，会根据请求的类型调用相应的 API 端点，如 /api/login、/api/recommendation、/api/register 等。

这些 API 端点实际上是 Node.js 使用 child_process.spawn 方法调用的 C++ 编写的后端程序。这些程序直接与 MySQL 8.0 数据库交互，存储和检索用户信息、游学日记和路线规划数据。

任务执行完毕后，C++ 程序将结果或错误信息返回给 Node.js 服务器，服务器再将这些信息封装成 HTTP 响应发送回前端。前端接收到响应后，根据内容更新用户界面，完成用户操作的闭环。

2.3.2.登录模块架构

登录模块设计如图：



各子模块主要功能如下：

Login：实现各种登录有关的操作；

TestLogin：响应前端的登录请求；

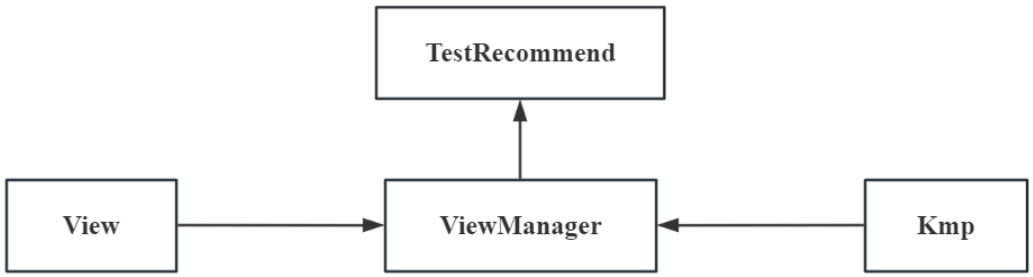
TestRegister：响应前端的注册请求；

TestForgotPassword：响应前端的找回密码请求；

TestChangePassword：响应前端的修改密码请求。

2.3.3.游学推荐模块架构

游学推荐模块设计如图：



各子模块主要功能如下：

TestRecommend：响应前端的各种游学推荐请求；

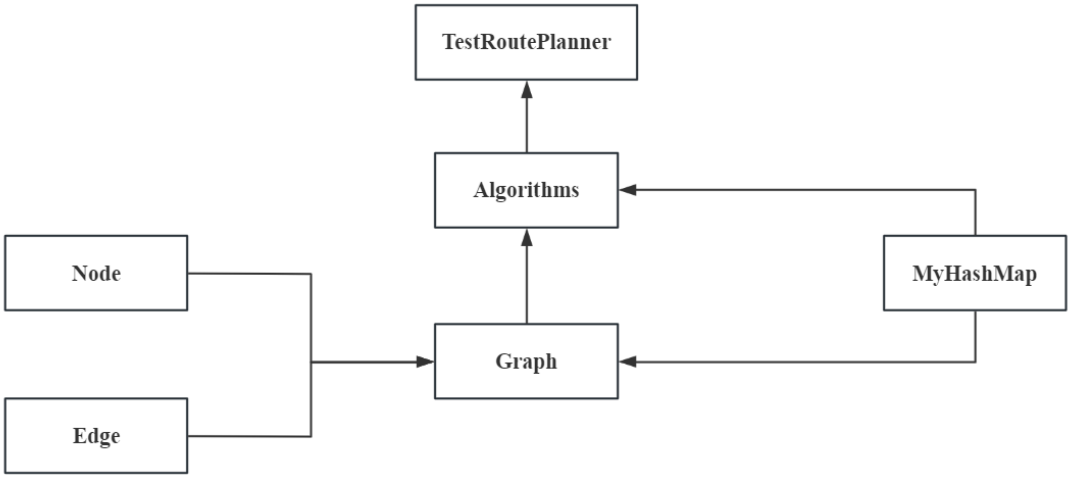
ViewManager：实现游学推荐的主要功能：排序、搜索等；

View：存储各景区、校园的数据结构；

Kmp：实现 Kmp 字符串匹配算法；

2.3.4.路线规划模块架构

路线规划模块设计如图：



各子模块主要功能如下：

TestRoutePlanner：响应前端的各种路线规划请求；

Algorithms：实现多种路线规划的核心算法；

Graph：存储当前地图的数据结构；

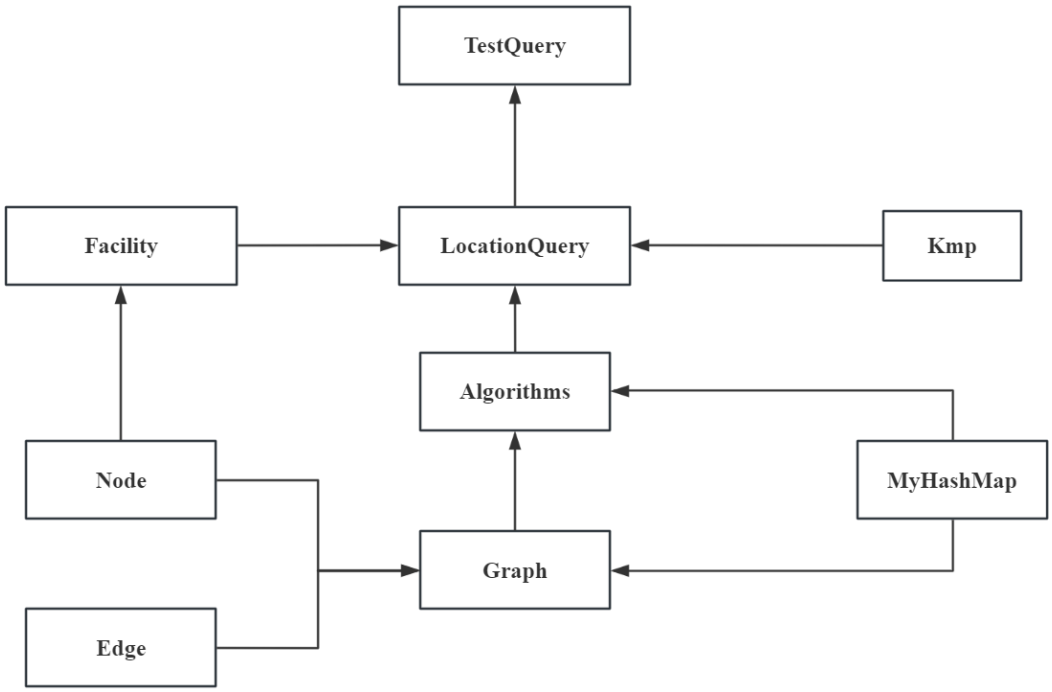
MyHashMap：自己实现的哈希表数据结构；

Node：用于存储不同建筑、设施的数据结构；

Edge：用于存储不同道路的数据结构。

2.3.5.场所查询模块架构

场所查询模块设计如图：



各子模块主要功能如下：

TestQuery：响应前端的各种场所查询请求；

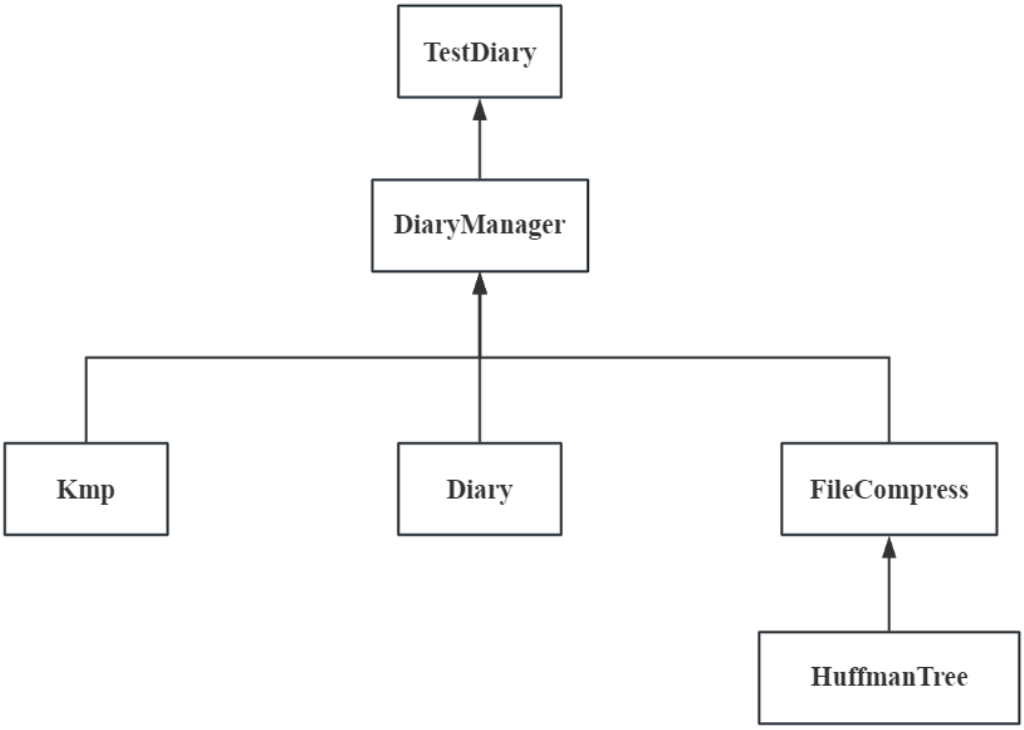
Facility：用于存储设施的数据结构；

LocationQuery：实现查询设施、计算距离并排序等核心功能；

剩余子模块与前文相同。

2.3.6.游学日记模块架构

游学日记模块设计如图：



各子模块主要功能如下：

TestDiary：响应前端的各种游学日记请求；

DiaryManager：实现游学日记模块的主要功能、算法；

Diary：用于存储日记的数据结构；

FileCompress：实现日记压缩、解压的核心算法；

HuffmanTree：实现压缩功能所需要的哈夫曼树；

剩余子模块与前文相同。

3.功能实现

3.1. 主要功能列表

3.1.1.注册登录

1) 用户可以注册账号；
2) 用户可以登录账号；
3) 用户可以修改账号密码；
4) 用户可以找回密码；

3.1.2.游学推荐

1) 个性化推荐：

(1) 学生可以根据个人喜好选择景点和学校。

(2) 系统会根据游学热度、评价和个人兴趣向学生推荐景点和学校。

2) 查询与排序：

(1) 输入名称、类别、关键字等信息来查询景点和学校；

(2) 查询结果应能根据热度和评价进行排序；

(3) 可选择前十展示或全部展示。

3.1.3.游学路线规划

1) 点到点游学路线规划：

(1) 输入目标地点后，系统提供从当前位置出发到达目的地的最优线路。

2) 途经多点的游学线路规划：

(1) 输入多个目标地点时，系统规划一条从当前位置出发，经过所有目标点并返回当前位置的最优线路。

3) 线路规划策略：

(1) 最短距离策略：以最短的地理距离作为规划依据。

(2) 最快时间策略：考虑道路拥挤度，以最短的旅行时间作为规划依据。

(3) 交通工具选择：在校区内可以选择自行车和步行，在景区内可以选择步行和电瓶车。

4) 路线展示：

(1) 提供用户友好的界面，使学生能够轻松输入目标信息，并查看规划的线路。

3.1.4.场所查询

1) 附近设施查询：

(1) 用户在景区或学校内部选中某个景点或场所后，系统能找出该地点附近一定范围内的超市、卫生间等设施。

2) 类别过滤：

(1) 用户可以通过选择类别对查询结果进行过滤，以便找到特定类型的设施。

(2) 用户可以输入类别名称，系统根据此输入查找附近服务设施，并进行距离排序。

3.1.5.游学日记管理

1) 游学日记撰写与编辑:

(1) 学生可以在游学过程中或结束后撰写日记，记录游学体验。

2) 日记浏览与评分：

(1) 学生可以浏览其他学生的游学日记，日记的浏览量作为热度指标。

(2) 学生浏览后可以对日记进行评分。

3) 日记查询：

(1) 学生可以输入游学目的地、日记名称、内容等，系统将筛选出相关日记并根据热度和评分排序。

4) 日记压缩存储：

(1) 游学日记数据可以进行无损压缩下载。

3.2. 功能实现特点及系统优点

1) 用户名支持中文、英文、标点符号混搭；
2) 支持多用户并发，且用户数量无上限；
3) 景区校园数量突破 300；
4) 多途径点导航没有途径点数量限制，用户可以途径足够多的景点建筑；
5) 本系统提供景区/校园详情主页，帮助用户全面了解目的地，做出更好的游学规划；
6) 用户查询到某个设施后，可一键进行当前位置到该设施的路线规划；
7) 当用户选择多途径点的路线规划功能时，本系统会根据用户的途径点数选择不同的算法提供路线；
8) 实现所有功能的图形化展示，增强了和用户之间的交互性，使用起来更加便捷直观。系统界面各功能清晰，操作逻辑性强，为用户提供了流畅的体验。

4.数据结构设计

4.1. Algorithms 类

```
class Algorithms {
   public:
    // 定义结构体存储 Dijkstra 算法的结果，包括路径和路径长度
    struct PathResult {
        std::vector<int> path;  // 存储最短路径的途径点序列
        double length;          // 存储最短路径的总长度
        double time;            // 存储最快路径的总时间

        PathResult()
            : length(0), time(0) {}
    };
};
```

4.2. Diary 类

```
class Diary {
   public:
    std::string title;              // 日记标题
    std::string author;             // 日记作者
    std::string destination;        // 日记描述对象/地点
    std::string content;            // 日记内容
    int popularity;                 // 日记热度
    int rating;                     // 日记评分
    int score;                      // 日记综合评分

    // 构造函数
    Diary(std::string title, std::string author, std::string destination, std::string content);

    // 用于打印日记内容
    void DiaryPrint();

    // 用于压缩下载日记
    void DiaryWriteintoFile();
};
```

4.3. DiaryManager 类

```
class DiaryManager {
   private:
    std::vector<Diary> diaries;  //日记数组

   public:
    // 用于打印所有日记
    void printAllDiaries();

    // 用于添加日记
    void addDiary(Diary diary);

    // 用于获取日记的综合评分
    void getScore(int a, int b);

    // 用于通过综合评分快速排序日记
    void q_sort(int left, int right);

    // 用于搜索日记
    void diarySearch(std::string search_title, std::string search_author, std::string
search_destination, std::string search_content, int search_mode);

    // 用于下载日记
    void diaryDownload();

    // 用于解压日记
    void diaryUncompress(std::string path);

    // 用于增加日记的热度
    int up_popularity(std::string content);

    // 用于更新日记的评分
    int update_rate(std::string content, int new_rating);
};
```

4.4. Edge 类

```
class Edge {
   public:
    // 交通工具类型
    enum type {
        WALK = 1,           // 步行
        EBIKE = 2,       // 电动车
        BIKE = 3         // 自行车
    };

    Node* source;        // 边的起点节点
    Node* destination;   // 边的终点节点
    double distance;     // 边的长度
    double congestion;   // 边的拥挤度
    type transportMode;  // 边的交通工具类型
    double speed;        // 边的交通工具速度

    // 构造函数
    Edge(Node* source, Node* destination, double distance, double congestion, type transportMode,
double speed);

    // 获取边的起点节点
    Node* getFrom() const;

    // 获取边的终点节点
    Node* getTo() const;

    // 获取边的长度
    double getLength() const;

    // 获取边的拥挤度
    double getCongestion() const;

    // 获取边的交通工具速度
    double getSpeed() const;

    // 获取交通工具类型
    type gettype() const;
};
```

4.5. Facility 类

```
class Facility {
   private:
    Node location;  // 使用 Node 类型来代表位置
    std::string name;  // 设施名称
    std::string type;  // 设施类别

   public:
    // 构造函数
    Facility(const Node& loc, const std::string& name, const std::string& type);

    // 获取设施名称
    std::string getName() const;

    // 获取设施类别
    std::string getType() const;

    // 获取设施位置
    const Node& getLocation() const;
};
```

4.6. CodeInfo 结构体

```
// 定义一个名为 CodeInfo 的结构体，用于存储字符的编码和频率信息。
struct CodeInfo {
    // CodeInfo 的构造函数，初始化字符编码为默认值，字符出现次数为 0。
    CodeInfo()
        : code(), cnt(0) {}

    // 重载大于运算符，用于比较两个 CodeInfo 对象的频率 cnt。
    friend bool operator>(const CodeInfo& left, const CodeInfo& right);

    // 重载不等于运算符，用于比较两个 CodeInfo 对象的频率 cnt 是否不相等。
    friend bool operator!=(const CodeInfo& left, const CodeInfo& right);

    // 重载加法运算符，用于合并两个 CodeInfo 对象的频率 cnt。
    friend CodeInfo operator+(const CodeInfo& left, const CodeInfo& right);

    // 字符本身，使用 unsigned char 类型，可以存储 0 到 255 范围内的值。
    unsigned char ch;

    // 该字符的哈夫曼编码，使用 std::string 类型存储。
    std::string code;

    // 该字符出现的次数，使用 long long 类型存储，可以存储非常大的数值。
    long long cnt;
};
```

4.7. LocationQuery 类

```
class LocationQuery {
   private:
    Graph& graph;  // 指向图的指针
    std::vector<Node*> facilities;  // 用于存储设施节点

   public:
    // 构造函数
    explicit LocationQuery(Graph& graph);
    void loadFacilities();
    std::vector<Node*> findNearbyFacilities(Node* location, double radius);
    std::vector<Node*> filterResultsByCategory(std::vector<Node*> results, std::string& category);
    std::vector<Node*> sortFacilitiesByDistance(std::vector<Node*>& facilities, int low, int high);
};
```

4.8. FileCompress 类

```
// 该类包含压缩和解压缩文件的方法
class FileCompress
{
   public:
    // 用于压缩文件
    void Compress(const std::string& FilePath);

    // 用于解压缩文件
    void UnCompress(const std::string& FilePath);

   private:
    // 用于从文件路径中获取文件名
    void GetFileName(const std::string& FilePath, std::string& output);

    // 用于获取扩展名（后缀）
    void GetPostfixName(const std::string& FilePath, std::string& output);

    // 用于填充 info 信息，读取源文件并填充字符频率信息
    void FillInfo(FILE* src);

    // 用于填充编码信息，根据哈夫曼编码填充字符的编码
    void FillCode(const HuffmanTreeNode<CodeInfo>* pRoot);

    // 核心压缩函数
    void CompressCore(FILE* src, FILE* dst, const std::string& FilePath);

    // 用于保存编码信息至压缩文件首部
    void SaveCode(FILE* dst, const std::string& FilePath);

    // 用于从文件中获取一行元素
    void GetLine(FILE* src, unsigned char* buf, int size);

    // 用于从解压缩文件中获取头部编码信息
    void GetHead(FILE* src, std::string& Postfix);

    // 核心解压函数
    void UnCompressCore(FILE* input, FILE* output, HuffmanTreeNode<CodeInfo>* pRoot);

   private:
    CodeInfo info[256];  // CodeInfo 类型数组
};
```

4.9. Graph 类

```
class Graph {
   public:
    int size;  // 图的大小
    HashMap<int, Node*, HashFunc> nodes;  // 存储图里的所有节点

    // 构造函数
    Graph(int size);

    // 用于添加节点
    void addNode(int id, Node::Type type, const std::string& name, const std::string& description);

    // 用于添加边
    void addEdge(const int& from, const int& to, double distance, double congestion, double speed,
Edge::type transportMode);

    // 用于获取节点
    Node* getNode(int id);
};
```

4.10. HuffmanTreeNode 结构体

```
// 定义哈夫曼树节点结构体
template <typename T>
struct HuffmanTreeNode
{
    // 构造函数
    HuffmanTreeNode(const T& data);

    T _weight;      // 节点权重，通常表示字符出现频率
    HuffmanTreeNode* pLeft;  // 指向左子节点的指针
    HuffmanTreeNode* pRight; // 指向右子节点的指针
    HuffmanTreeNode* pParent; // 指向上父节点的指针
};
```

4.11. greateer 结构体

```
// 定义比较结构体，用于优先队列中比较节点权重
template <typename T>
struct greater
{
    bool operator()(const T& left, const T& right);
};
```

4.12. HuffmanTree 类

```
// 定义哈夫曼树类
template <typename T>
class HuffmanTree
{
public:
    // 构造函数
    HuffmanTree(const T* weight, int size, const T& invalid);

    // 析构函数，释放哈夫曼树占用的内存
    ~HuffmanTree();

    // 层序遍历哈夫曼树
    void LevelTraverse();

    // 获取哈夫曼树的根节点
    HuffmanTreeNode<T>* GetRoot();

private:
    // 销毁哈夫曼树的辅助函数
    void _Destroy(HuffmanTreeNode<T>*& pRoot);

    // 创建哈夫曼树的辅助函数
    void _Create(const T* weight, int size);

private:
    HuffmanTreeNode<T>* pRoot; // 哈夫曼树的根节点
    T _invalid;               // 无效的权重值，用于标记不使用的字符
};
```

4.13. Node 类

```
class Node {
   public:
    // 节点类型枚举
    enum Type {
        BUILDING = 1,  // 建筑、景点、场所
        FACILITY = 2,  // 设施
        NONE = 0
    };

    int id;                    // 节点的唯一标识
    Type type;                 // 节点的类型
    std::string name;          // 节点的名称
    std::vector<Edge*> edges;  // 与该节点相连的边
    std::string description;   // 节点的描述信息
    double distance;           // 与当前搜索位置的距离

    // 构造函数
    Node(int id, Type type, const std::string& name, const std::string& description);

    // 析构函数
    ~Node();  // 释放 edges 中的边指针

    // 用于获取节点 ID
    int getId() const;

    // 用于获取节点名称
    const std::string& getName() const;

    // 用于获取节点类型
    Type getType() const;

    // 用于获取节点描述
    const std::string& getDescription() const;

    // 用于设置节点描述
    void setDescription(const std::string& description);

    // 用于获取与当前位置的距离
    double getDistance() const;

    // 用于设置与当前位置的距离
```

```
    void setDistance(double distance);

    // 用于添加边
    void addEdge(Edge* edge);

    // 静态方法，用于创建表示“空”的 Node 实例
    static Node emptyNode();
};
```

4.14. View 类

```
class View {
   public:
    int LocationID;    // 景点 ID
    std::string Name;  // 景点名字
    std::string Type;  // 景点类型
    int Popularity;    // 景点热度
    double Ratings;    // 景点评分
    int Score;         // 景点综合评分
};
```

4.15. ViewManager 类

```
class ViewManager {
   private:
    std::vector<View> views;  // 景区数组

   public:
    void getViews();
    void Recommendation(int obj, int quan, int mo, std::string s_s);
    void getScore(int a, int b);
    void q_sort(int left, int right);
    std::vector<View> selectSort(int length,int obj,std::string search_string);
};
```

5.算法设计与性能分析

5.1. 快速排序算法

当用户选择展示全部时，我们采用快速排序算法进行排序展示，因此时数据量较大，而快速排序大部分情况下效率较高。

算法流程：

1) 选择一个中间元素作为基准。
2) 初始化左右指针进行分区操作。
3) 对基准值左边和右边的子数组递归地进行快速排序。

算法时间复杂度：在最坏情况下的复杂度$O(nlogn)$。

```
1. /*排序算法*/
2. void ViewManager::q_sort(int left, int right) {
3.     int p = views[(left + right) / 2].Score;
4.     int i = left;
5.     int j = right;
6.     while (i <= j) {
7.         while (views[i].Score > p)
8.             i++;
9.         while (views[j].Score < p)
10.            j--;
11.        if (i <= j) {
12.            std::swap(views[i], views[j]);
13.            i++;
14.            j--;
15.        }
16.    }
17.    if (j > left)
18.        q_sort(left, j);
19.    if (right > i)
20.        q_sort(i, right);
21. }
```

5.2. 选择排序算法

当用户选择展示前十时，我们采用选择排序的变体进行不完全排序展示，即只排出符合条件的前十个内容。

采用这样的设计，可以大大提高效率，避免算力的浪费。

算法流程如下：

1) 遍历列表，根据类型和搜索条件筛选符合条件的景点。
2) 使用 kmp 算法检查景点名称是否包含搜索字符串。
3) 对筛选出的景点列表进行排序。
4) 重复以下步骤，直到列表排序完成或达到前十个元素：
5) 寻找当前未排序部分中得分最高的景点。
6) 将该景点与未排序部分的第一个元素交换位置。
7) 返回排序后的前十个景点列表。

算法时间复杂度：$O(n)$，由于这里只进行最多 10 次选择排序操作，因此其时间复杂度可以近似为$O(1)$。

```
1. // 选择排序，实现非全排列展示前十个
2. std::vector<View> ViewManager::selectSort(int length, int object, std::string
search_string) {
3.     std::vector<View> filteredViews;
4.     if (object == 0) {
5.         for (int i = 0; i < length; i++)
6.             if (views[i].Type=="attraction" && kmp(search_string, views[i].Name))
7.                 filteredViews.push_back(views[i]);
8.     } else if (object == 1) {
9.         for (int i = 0; i < length; i++)
10.            if (views[i].Type == "school" && kmp(search_string, views[i].Name))
11.                filteredViews.push_back(views[i]);
12.    } else if (object == 2) {
13.        for (int i = 0; i < length; i++)
14.            if (kmp(search_string, views[i].Name))
15.                filteredViews.push_back(views[i]);
16.    }
17.    int index;
18.    for (int i = 0; i < filteredViews.size() && i < 10; i++) {
19.        index = i;
20.        for (int j = i + 1; j < filteredViews.size(); j++) {
```

```
            if (filteredViews[j].Score > filteredViews[index].Score)
                index = j;
        }
        std::swap(filteredViews[i], filteredViews[index]);
    }
    return filteredViews;
}
```

5.3. KMP 算法

本系统采用 KMP（Knuth-Morris-Pratt）字符串匹配算法，用于在用户进行关键词搜索等操作时，进行内容的匹配查找。

相比朴素算法，该算法利用少量的空间牺牲大幅提高了时间效率。

算法流程：

1) 如果 t 等于 "-1"，返回 true；
2) 初始化 nextval 数组，长度为 t 的长度；
3) 调用 get_nextval 函数计算模式字符串 t 的 nextval 数组；
4) 初始化变量 i 和 j 为 0；
5) 进入 while 循环；
6) 如果 j 大于或等于模式字符串长度，返回 true，表示匹配成功；否则返回 false。

算法时间复杂度：$O(n)$。

```
void get_nextval(std::string t, int nextval[]) {
    int j = 0, k = -1;
    int t_len = t.length();
    nextval[0] = -1;
    while (j < t_len)
        if (k == -1 || t[j] == t[k]) {
            j++;
            k++;
            if (t[j] != t[k])
                nextval[j] = k;
            else
                nextval[j] = nextval[k];
        } else
            k = nextval[k];
}
```

```
16.
17. bool kmp(std::string t, std::string s) {
18.     if (t == "-1")
19.         return 1;
20.     int line_limit;
21.     line_limit = t.length();
22.     int nextval[line_limit];
23.     int i = 0, j = 0;
24.     int s_len = s.length(), t_len = t.length();
25.     get_nextval(t, nextval);
26.     while (i < s_len && j < t_len)
27.         if (j == -1 || s[i] == t[j]) {
28.             i++;
29.             j++;
30.         } else
31.             j = nextval[j];
32.     if (j >= t_len)
33.         return true;
34.     else
35.         return false;
36. }
```

5.4. Dijkstra 算法

本项目采用 Dijkstra 算法寻找图中两点间的最短路径。以此来满足用户进行点到点路线规划的操作。经测试，Dijkstra 算法足以满足本系统的数据量，且效果较好。

包括最短时间策略在内，算法流程基本相同，只改变了比较的值（从距离改为时间），故不多赘述，只介绍最短距离路径的算法实现。

算法流程如下：

1) 初始化距离、前驱节点和访问标记的哈希表。
2) 设置起始节点的距离为 0。
3) 循环直到找到目标节点或所有节点被访问。
4) 使用辅助函数 findMinDistanceNode 找到最小距离节点。
5) 更新邻居节点的距离和前驱节点。
6) 重建路径并返回结果。

算法时间复杂度：$O(n^2)$，在最坏情况下每个节点都可能成为最小距离节点。

```
// 辅助函数，用于找到尚未访问的具有最小距离（时间）的节点
int findMinDistanceNode(HashMap<int, double, HashFunc>& distance, HashMap<int, bool, HashFunc>& visited, int size) {
    double minDistance = std::numeric_limits<double>::max();
    int minNode = -1;
    for (int i = 0; i < size; i++) {
        double nodeDistance = *(distance.find(i));
        if (!*(visited.find(i)) && nodeDistance < minDistance) {
            minDistance = nodeDistance;
            minNode = i;
        }
    }
    return minNode;
}

// 寻找最短路径算法
Algorithms::PathResult Algorithms::findShortestPath(Graph& graph, int startNodeID, int endNodeID) {
    int numNodes = graph.size;  // 图中节点总数
    // 初始化距离、前驱节点和访问标记的哈希表
    HashMap<int, double, HashFunc> distances(numNodes);
    HashMap<int, int, HashFunc> predecessors(numNodes);
    HashMap<int, bool, HashFunc> visited(numNodes);
```

```
    PathResult result;

    // 初始化
    for (int i = 0; i < numNodes; ++i) {
        distances.insert(i, std::numeric_limits<double>::max());
        predecessors.insert(i, -1);
        visited.insert(i, false);
    }
    distances[startNodeID] = 0;

    for (int i = 0; i < numNodes; ++i) {
        int u = findMinDistanceNode(distances, visited, numNodes);
        if (u == -1)
            break;  // 所有节点都已访问
        if (u == endNodeID)
            break;  // 找到最短路径

        visited[u] = true;

        for (const auto& edge : graph.getNode(u)->edges) {
            int v = edge->getTo()->id;
            double alt = distances[u] + edge->distance;
            if (alt < distances[v]) {
                distances[v] = alt;
                predecessors[v] = u;
            }
        }
    }

    // 重建从 endNodeId 到 startNodeId 的路径
    std::stack<int> pathStack;
    for (int at = endNodeID; at != -1; at = predecessors[at])
        pathStack.push(at);

    if (!pathStack.empty() && pathStack.top() == startNodeID) {  // 如果路径存在
        while (!pathStack.empty()) {
            result.path.push_back(pathStack.top());
            pathStack.pop();
        }
        result.length = distances[endNodeID];
    }

    return result;
}
```

5.5. 基于回溯法的排列生成算法

当用户选择多途径点的路线规划功能时，本系统会根据用户的途径点数选择不同的算法提供路线。

当途径点数量 ＜5 时，采用基于回溯法的排列生成算法进行暴力求解 TSP 问题。

经实际测试，该算法在途径点 ＞8 后，会出现＞10min 的求解时长，对用户体验极不友好，但少量途径点时，效率不低且路线精确，故保留了该算法。

算法流程如下：

1) 初始化结果，设置初始路径长度为无穷大。
2) 调用 permutations 函数生成所有排列并计算路径长度。
3) 返回最短路径结果。

其中 permutations 函数算法流程如下：

1) 如果左右指针相遇，计算路径长度和路径。
2) 如果当前路径长度小于已知最短路径，则更新结果。
3) 对数组中的每个元素，交换当前元素与左侧元素，递归生成排列。

算法时间复杂度：$O(n!\,n^2)$，因为它们生成所有可能的排列，且每个排列都要执行 Dijkstra 算法。

```
1. void permutations(Algorithms::PathResult& result, Graph& graph, int startNodeID,
std::vector<int> arr, int l, int r) {
2.     if (l == r) {
3.         // 基础条件：如果左右指针相遇
4.         double tempLength = 0;
5.         std::vector<int> tempPath;
6.         for (int i = 0; i < arr.size(); i++) {
7.             if (i == 0) {
8.                 Algorithms::PathResult temp = Algorithms::findShortestPath(graph,
startNodeID, arr[i]);
9.                 tempLength += temp.length;
10.                for (int j = 0; j < temp.path.size(); j++) {
11.                    tempPath.push_back(temp.path[j]);
12.                }
13.            } else {
14.                Algorithms::PathResult temp = Algorithms::findShortestPath(graph, arr[i -
1], arr[i]);
15.                tempLength += temp.length;
```

```
16.                 for (int j = 1; j < temp.path.size(); j++) {
17.                     tempPath.push_back(temp.path[j]);
18.                 }
19.             }
20.         }
21.         Algorithms::PathResult temp = Algorithms::findShortestPath(graph, arr.back(),
startNodeID);
22.         tempLength += temp.length;
23.         for (int j = 1; j < temp.path.size(); j++) {
24.             tempPath.push_back(temp.path[j]);
25.         }
26.
27.         if (tempLength < result.length) {
28.             result.length = tempLength;
29.             result.path = tempPath;
30.         }
31.     } else {
32.         // 对于数组中的每个元素，将其与左侧元素交换，然后递归打印右侧子数组的排列
33.         for (int i = l; i <= r; i++) {
34.             // 交换 arr[l] 和 arr[i]
35.             std::swap(arr[l], arr[i]);
36.             // 递归打印右侧子数组的排列
37.             permutations(result, graph, startNodeID, arr, l + 1, r);
38.             // 回溯：交换回来，恢复原样
39.             std::swap(arr[l], arr[i]);
40.         }
41.     }
42. }
43.
44. // 暴力算法，全排列
45. Algorithms::PathResult Algorithms::findBruteForcePath(Graph& graph, int startNodeID,
std::vector<int>& targets) {
46.     PathResult result;
47.     int size = targets.size();
48.     result.length = INF;
49.     permutations(result, graph, startNodeID, targets, 0, size - 1);
50.
51.     return result;
52. }
```

5.6. 模拟退火算法

如上一段介绍，当途径点＞8 时，暴力算法无法快速求解出最佳路径，故我们引入了新的算法——模拟退火算法。

经实际测试，在我们小型地图（景区、校园）的条件下，该算法可以在多项式时间内求解出最优路线，不存在长时间求解不出答案的问题。

即使途径点数量非常多（几十个），也可以在退火过程中求解出近似最优解。

下面介绍模拟退火算法：

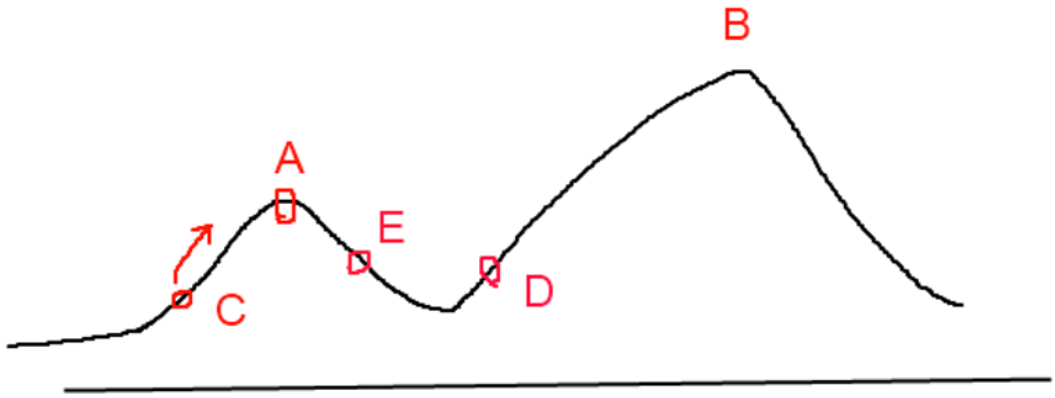
与遗传算法、粒子群优化算法和蚁群算法等不同，模拟退火算法不属于群优化算法，不需要初始化种群操作。

模拟退火算法 (SA) 来源于固体退火原理，是一种基于概率的算法。

固体退火原理：将固体加温至充分高的温度，再让其徐徐冷却，加温时，固体内部粒子随温升变为无序状，内能增大，分子和原子越不稳定。而徐徐冷却时粒子渐趋有序，能量减少，原子越稳定。在冷却（降温）过程中，固体在每个温度都达到平衡态，最后在常温时达到基态，内能减为最小。

模拟退火算法原理：以一定的概率来接受一个比当前解要差的解，因此有可能会跳出这个局部的最优解，达到全局的最优解。

以下图为例，模拟退火算法在搜索到局部最优解 A 后，会以一定的概率接受到 E 的移动。也许经过几次这样的不是局部最优的移动后会到达 D 点，于是就跳出了局部最大值 A。



模拟退火算法描述：

若$Y(i+1) \geq Y(i)$ ：(即移动后得到更优解)，则总是接受该移动；

若 $Y(i+1) < Y(i)$ ：(即移动后的解比当前解要差)，则以一定的概率接受移动，而且这个概率随着时间推移逐渐降低（逐渐降低才能趋向稳定）。

这里的“一定的概率”的计算参考了金属冶炼的退火过程，这也是模拟退火算法名称的由来。

根据热力学的原理，在温度为 T 时，出现能量差为 dE 的降温的概率为 P(dE)，表示为：

$$P(dE) = exp\frac{dE}{kT}$$

其中 k 是一个常数，exp 表示自然指数，且 dE<0。

即：温度越高，出现一次能量差为 dE 的降温的概率就越大；温度越低，则出现降温的概率就越小。又由于 dE 总是小于 0，因此$\frac{dE}{kT} < 0$ ，所以 P(dE)的函数取值范围是(0,1) 。

随着温度 T 的降低，P(dE)会逐渐降低。

我们将一次向较差解的移动看做一次温度跳变过程，我们以概率 P(dE)来接受这样的移动。

在本项目中，每一次变换路线顺序即可看作一次温度跳变过程。

算法完整过程如下：

1) 根据目标节点构造一个完全图，其中包含起点和所有目标点。
2) 在完全图中添加节点和边，边的权重是原始图中相应路径的最短长度。
3) 使用 simulatedAnnealing 函数在这个完全图上寻找最优路径：
   (1) 初始化当前结果和最佳结果，设置起始温度和冷却率。
   (2) 在温度高于结束温度时，进行循环，每次循环中：
      a) 随机扰动当前路径，交换两个节点的位置。
      b) 计算新路径的长度，并与当前结果进行比较，根据退火概率接受新路径。
      c) 如果新路径更优，则更新最佳结果。
   (3) 每次循环后，降低温度。
   (4) 返回最佳结果。
4) 根据模拟退火的结果，构造最终的路径。这涉及到将模拟退火过程中的节点顺

序转换为原始图中的实际路径。

5) 返回包含最终路径和长度的 PathResult 对象。

算法时间复杂度：近似于$O(n^4)$，由于模拟退火是一个概率性算法，其确切的时间复杂度难以精确计算，但通常会在合理的时间内收敛到一个解。

```
// 模拟退火算法相关参数
const double START_TEMPERATURE = 50000.0;
const double END_TEMPERATURE = 1.0;
const double COOLING_RATE = 0.998;

// 模拟退火算法
Algorithms::PathResult Algorithms::simulatedAnnealing(Graph& completeGraph, int startNodeID, std::vector<int>& nodes) {
    // 随机数生成器
    std::random_device rd;
    std::mt19937 gen(rd());
    std::uniform_real_distribution<> dis(0.0, 1.0);

    // 初始化路径和路径长度
    Algorithms::PathResult currentResult;
    currentResult.path = nodes;
    currentResult.path.insert(currentResult.path.begin(), startNodeID);
    currentResult.path.push_back(startNodeID);

    for (int i = 0; i < currentResult.path.size() - 1; ++i)
        for (auto edg : completeGraph.getNode(currentResult.path[i])->edges)
            if (edg->getTo()->id == currentResult.path[i + 1]) {
                currentResult.length += edg->getLength();
                break;
            }

    Algorithms::PathResult bestResult = currentResult;

    double temperature = START_TEMPERATURE;

    // 退火过程
    while (temperature > END_TEMPERATURE) {
        std::vector<int> newPath = currentResult.path;

        // 扰动
        int i = 1 + rand() % (newPath.size() - 2);  // 避免改变起点和终点
        int j = 1 + rand() % (newPath.size() - 2);
        std::swap(newPath[i], newPath[j]);
```

```
38.
39.         Algorithms::PathResult newResult;
40.         newResult.path = newPath;
41.         for (int i = 0; i < newResult.path.size() - 1; ++i)
42.             for (auto edgs : completeGraph.getNode(newResult.path[i])->edges)
43.                 if (edgs->getTo()->id == newResult.path[i + 1]) {
44.                     newResult.length += edgs->getLength();
45.                     break;
46.                 }
47.
48.         double delta = newResult.length - currentResult.length;
49.         if (delta < 0 || dis(gen) < exp(-delta / temperature))
50.             currentResult = newResult;
51.
52.         if (currentResult.length < bestResult.length)
53.             bestResult = currentResult;
54.
55.         temperature *= COOLING_RATE;
56.     }
57.
58.     return bestResult;
59. }
60.
61. // 构造完全图并使用模拟退火算法
62. Algorithms::PathResult Algorithms::findOptimalPath(Graph& graph, int startNodeID,
std::vector<int>& targets) {
63.     // 构造完全图
64.     int n = targets.size();
65.     Graph completeGraph(n + 1);                              // 包含起点和所有目标点
66.     std::map<std::pair<int, int>, std::vector<int>> paths;  // 存放完全图路径
67.
68.     // 完全图节点添加
69.     Node* t = graph.getNode(startNodeID);
70.     completeGraph.addNode(startNodeID, t->getType(), t->getName(), t->getDescription());
71.     for (int i = 0; i < n; i++) {
72.         Node* temp = graph.getNode(targets[i]);
73.         completeGraph.addNode(targets[i], temp->getType(), temp->getName(),
temp->getDescription());
74.     }
75.
76.     // 完全图路径添加
77.     for (int i = 0; i < n; ++i) {
78.         for (int j = i + 1; j < n; ++j) {
```

```
79.             Algorithms::PathResult result = Algorithms::findShortestPath(graph,
targets[i], targets[j]);
80.             completeGraph.addEdge(targets[i], targets[j], result.length, 0, 0,
Edge::type::WALK);
81.             std::pair<int, int> key1(targets[i], targets[j]);
82.             std::pair<int, int> key2(targets[j], targets[i]);
83.             paths.insert(std::make_pair(key1, result.path));
84.             std::reverse(result.path.begin(), result.path.end());
85.             paths.insert(std::make_pair(key2, result.path));
86.         }
87.         Algorithms::PathResult resultStart = Algorithms::findShortestPath(graph,
startNodeID, targets[i]);
88.         completeGraph.addEdge(startNodeID, targets[i], resultStart.length, 0, 0,
Edge::type::WALK);
89.         std::pair<int, int> key1(startNodeID, targets[i]);
90.         std::pair<int, int> key2(targets[i], startNodeID);
91.         paths.insert(std::make_pair(key1, resultStart.path));
92.         std::reverse(resultStart.path.begin(), resultStart.path.end());
93.         paths.insert(std::make_pair(key2, resultStart.path));
94.     }
95.
96.     // 模拟退火求解
97.     Algorithms::PathResult ret = simulatedAnnealing(completeGraph, startNodeID, targets);
98.
99.     // 根据途径点的顺序，添加中间的 node
100.    std::vector<int> finalpath;
101.    for (int i = 0; i < ret.path.size() - 1; i++) {
102.        std::pair<int, int> key(ret.path[i], ret.path[i + 1]);
103.        auto it = paths.find(key);
104.        finalpath.insert(finalpath.end(), it->second.begin(), it->second.end() - 1);
105.    }
106.    finalpath.push_back(ret.path.back());
107.    ret.path = finalpath;
108.
109.    return ret;
110. }
```

5.7. 哈夫曼编码

本系统采用哈夫曼编码对日记进行压缩下载。

算法流程：

1) 打开输入文件；
2) 调用 FillInfo 函数填充字符出现次数信息：
   (1) 初始化 info 数组；
   (2) 统计每个字符在文件中出现的次数；
   (3) 构建哈夫曼树；
   (4) 生成每个字符的哈夫曼编码。
3) 获取压缩后的文件名，并打开输出文件；
4) 调用 CompressCore 函数执行压缩核心逻辑：
   (1) 将输入文件的指针重置到文件开头；
   (2) 保存压缩文件的编码头信息；
   (3) 读取输入文件并进行编码转换，逐字节写入输出文件。
5) 关闭文件。

算法时间复杂度：$O(nlogn)$

该算法实现包括压缩、解压两部分，内容过多，可见源代码附件中的 HuffmanTree.h，FileCompress.cpp。

6.系统测试结果

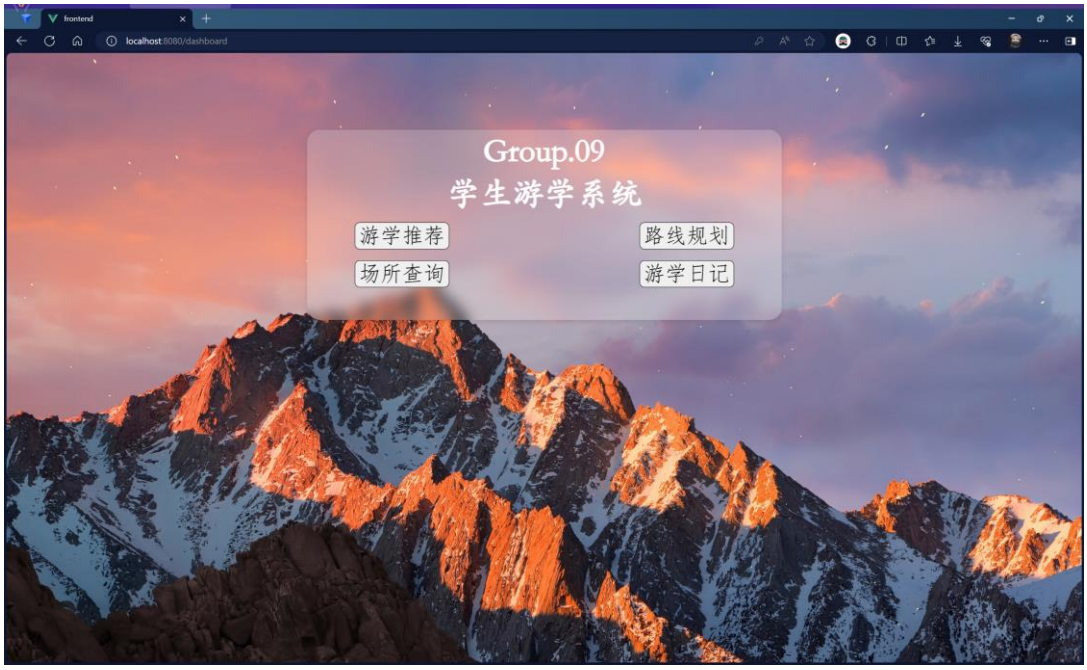
全部测试结果可见附件：开发文档/测试报告，本报告仅展示主要功能的测试。

6.1. 登录

在学生游学系统首页点击页面右下角的登录按钮，即可进入登录界面。用户需要输入用户名及其对应的密码，并点击页面下方登录按钮即可登录。

6.2. 主页

用户登录后即可进入主界面。



用户可以点击不同按钮以进入不同的界面。点击游学推荐按钮即可进入游学推荐界面，点击路线规划按钮即可进入路线规划界面，点击场所推荐按钮即可进入场所推荐界面，点击游学日记按钮即可进入游学日记界面。

6.3. 游学推荐

进入游学推荐界面，该界面旨在通过各游学地点的热度和评分为用户推荐游学地点。该页面可通过筛选景区或学校，通过热度排序、评分排序或综合排序为用户列出游学地点。

用户可以通过页面上方的搜索框输入想要查找的游学地点名称，并点击右侧的搜索按钮进行搜索。搜索结果依旧可以通过筛选景区或学校，通过热度排序、评分排序或综合排序为用户列出。

点击对应游学地点的详情即可进入该游学地点的详情页面。该页面包含游学地点的名称（双语）及介绍。同时页面下方还有开始路线规划按钮，点击即可进入路线规划界面。

6.4. 路线规划

进入路线规划界面，该界面旨在为用户提供游学地点内部的导航，包括点到点导航和多途径点导航。

在点到点导航模式中，在下方的文本框需要输入当前地点和目的地点的序号（序号如页面右侧的图片所示），并可选择对应的交通工具。在景区中，可以选择步行或电动车；在校园中，可以选择步行或自行车。点击景区（或校园）路线规划下方的搜索按钮，即可得到导航的路线信息，包括最短路径和最快路径。

点击景区（或校园）路线规划下方的切换到多途径点按钮，即可切换到多途径点模式。点击相同位置的切换到点到点按钮，即可再次切换到点到点模式。

在多途径点导航模式中，在下方的文本框需要输入当前地点和途径地点的序号（序号如页面右侧的图片所示）。点击下方的添加途径点按钮，即可增加途径地点。点击景区（或校园）路线规划下方的搜索按钮，即可得到导航的路线信息。

6.5. 场所查询

进入场所查询界面，该界面旨在为用户提供游学地点内部所在位置周围的各场所信息，包括场所名称、场所种类、所在位置与场所的距离等信息。

用户需要输入当前所在位置的序号（序号如页面右侧的图片所示），并设置查询范围。查询范围可以通过手动输入设置或移动拖拽条设置。用户还可以通过手动输入场所类型或选择场所类型来筛选查询结果，点击下方的查询按钮即可显示查询结果。

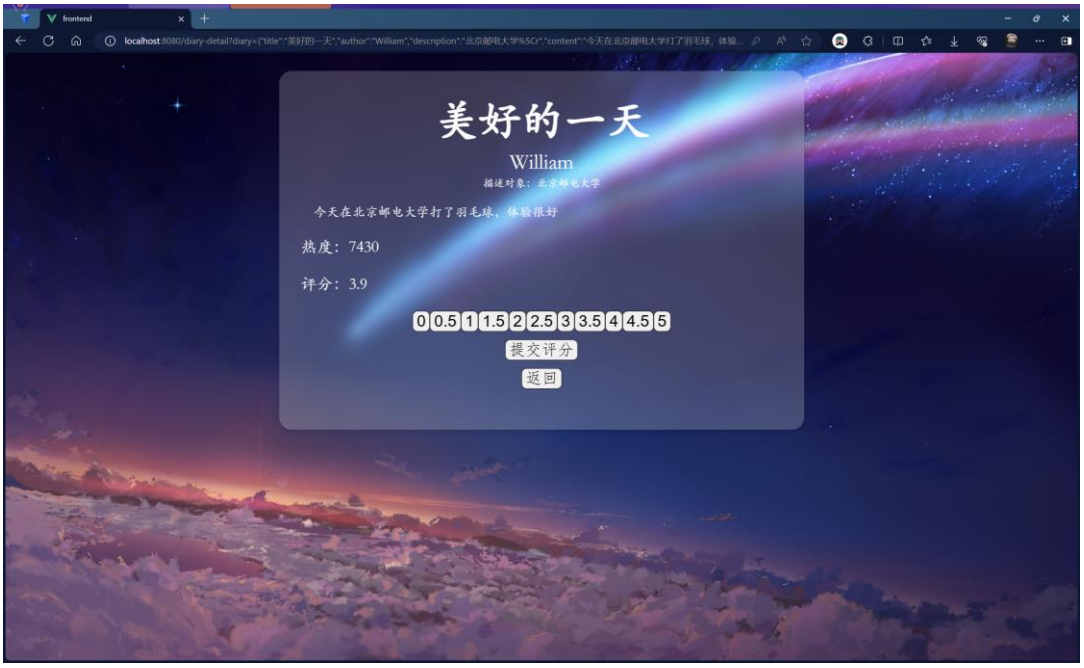
此时点击相应场所的路线按钮将直接进入点到点的路线规划界面。

6.6. 游学日记管理

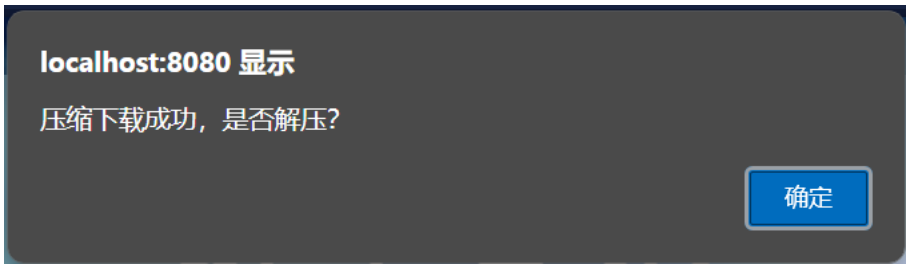
进入游学日记界面，该页面旨在为用户提供游学日记的搜索、下载以及撰写功能。

用户可以通过搜索游学日记的标题、作者名、目的地或内容来搜索游学日记。搜索结果可以选择游学日记评分排序或游学日记热度排序。点击下方的搜索按钮即可显示搜索结果。

点击对应游学日记的详情即可进入该游学日记的详情页面。该页面包含游学日记的标题、作者、描述对象、内容、热度和评分。同时页面下方还有评分系统，用户可以选择相应评分来为该日记打分，点击下方的提交评分按钮即可。

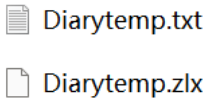


在游学日记界面搜索完成后，可以点击页面左侧的下载日记按钮下载日记，此时会提示日记“压缩下载成功，是否解压？”，选择确定。

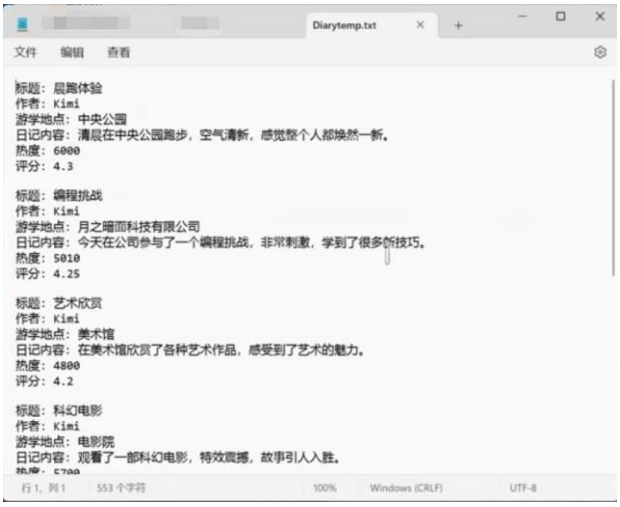


此时会提示用户输入刚才下载压缩文件的路径.

点击确认后该路径下会有两个文件，其中 zlx 后缀的为压缩文件，txt 后缀的为解压文件。



其中，txt 后缀的文件即为下载的日记。



点击游学日记界面下方的写日记按钮，即可进入写日记界面。用户需要输入日记的标题、作者、描述对象/地点和日记内容，并点击上传按钮，即可完成日记的撰写。

7.项目总结

7.1. 总结体会

在本次数据结构课程设计中，我们小组负责开发《学生游学系统》，旨在为学生提供一个全面的游学体验管理平台。

在此过程中，我们深刻体会到了数据结构在实际应用中的重要性。通过将数据结构的理论知识应用于实际项目开发中，我们加深了对图、树、排序等数据结构的理解。除此之外，我们更学会了如何写出更适合开发人员的代码，如何添加合理的注释，组织模块结构。

对于算法知识，我们学习了模拟退火算法，这种概率算法不同于课堂上的算法，它具有更多的复杂变化和模拟应用，既开拓了我们的眼界，也提高了我们的能力。

项目开发过程中，我们还学会了如何在团队中分工合作，有效沟通，共同解决问题。

本次课程设计不仅锻炼了我们的编程能力和项目开发能力，也提高了我们解决实际问题的能力。我们相信，通过这次经历，我们为未来的学习和工作打下了坚实的基础。

7.2. 后续改进

1) 考虑在路线规划时能实时在地图上进行导航，并更新当前状态信息;
2) 考虑能根据不同用户的不同兴趣实现个性化推荐，即协同过滤等等;

8.附录

相关源代码、可执行文件、开发文档见附件。