**北京邮电大学课程设计报告**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **课 程 设**  **计 名 称** | **数 据 结 构** | | **学 院** | **计 算 机** | **指导教师** | **王尊亮** |
| **班 级** | **班内序号** | **学 号** | | **学生姓名** | **成 绩** | |
| 2023211313 | 01 | 2023210887 | | 孙明皓 |  | |
| 2023211320 | 02 | 2023210890 | | 胡振宇 |  | |
| 2023211313 | 03 | 2023210976 | | 刘梓宸 |  | |
| **课**  **程**  **设**  **计**  **内**  **容** | 设计并实现一个个性化旅游系统，帮助大家管理自己的旅游活动，具备旅游地点推荐、旅游路线规划、旅游场所查询、旅游日记交流等功能。旅游前：需先按照旅游目的地的热度、评价和个人兴趣选择旅游目的地；旅游中：在景点（包括学校）内部也需要根据游览的目标规划最优的参观线路，在游览过程中还可以给出相应的景点介绍和场所查询；旅游后：可以根据所拍照片和游览经历生成旅游日记。  需完成四项基本功能及三个选做功能中的若干项。具体要求详见课程设计任务说明 PPT。 | | | | | |
| **学 生**  **课程设计**  **报 告**  （附 页） | 详见文档 | | | | | |
| **课**  **程**  **设**  **计**  **成**  **绩**  **评**  **定** | **评语**:  、  **成绩:**  指导教师签名：  2025年 6 月 20 日 | | | | | |

**数据结构课程设计报告**

目录

[1. 需求介绍 3](#_Toc201150535)

[1.1. 旅游推荐 3](#_Toc201150536)

[1.2. 旅游路线规划 3](#_Toc201150537)

[1.3. 场所查询 4](#_Toc201150538)

[1.4. 旅游日记管理和交流 4](#_Toc201150539)

[2. 系统架构 6](#_Toc201150540)

[2.1. 技术基础与开发环境 6](#_Toc201150541)

[2.2. 系统架构设计 6](#_Toc201150542)

[3. 实现功能 8](#_Toc201150543)

[3.1. 前置功能 8](#_Toc201150544)

[3.2. 旅游推荐 8](#_Toc201150545)

[3.3. 校园导航 9](#_Toc201150546)

[3.4. 旅游日记 11](#_Toc201150547)

[4. 成员分工 12](#_Toc201150548)

[5. 各模块设计 13](#_Toc201150549)

[5.1. 用户管理和数据库模块 13](#_Toc201150550)

[5.2. 旅游推荐模块 13](#_Toc201150551)

[5.3. 校园导航模块 14](#_Toc201150552)

[5.4. 旅游日记模块 14](#_Toc201150553)

[6. 算法分析 16](#_Toc201150554)

[6.1. 字符串查找算法 16](#_Toc201150555)

[6.2. 排序算法 17](#_Toc201150556)

[6.3. 路线规划算法 21](#_Toc201150557)

[6.4. 场所查询算法 23](#_Toc201150558)

[6.5. 压缩算法 25](#_Toc201150559)

[7. 测试评价 29](#_Toc201150560)

# 需求介绍

当前大家经常利用假期去各地旅游。个性化旅游系统可以帮助大家管理自己的旅游活动，具备旅游地点推荐、旅游路线规划、旅游场所查询、旅游日记交流等功能。

**（1）旅游前**：需先按照旅游目的地的热度、评价和个人兴趣选择旅游目的地；

**（2）旅游中**：在景点（包括学校）内部也需要根据游览的目标规划最优的参观线路，在游览过程中还可以给出相应的景点介绍和场所查询；

**（3）旅游后**：可以根据所拍照片和游览经历生成旅游日记。

## 旅游推荐

1. **景点排序**：

用户可以根据自己的喜好选择不同的景点和学校作为旅游目的地，在推荐页面，系统会向用户推荐旅游景点和学校，可以按照不同指标进行排序；

推荐算法基础要求为排序算法，可以按照设计好的不同排序标准进行排序（考虑到用户通常只看前10个景点或者学校，要求不经过完全排序可以排好前10的景点或者学校，并且考虑数据动态变化）

（2）**景点搜索**：

用户可以输入景点和学校的名称、类别、关键字等进行查询，查询结果有多项时，要进行排序。

## 旅游路线规划

（1）**点对点查询**

用户可以输入目标景点或者场所信息，系统会为用户规划从当前位置出发到达景点或者场所的最优旅游线路；

核心算法为最短路径算法。

（2）**多途径点路线规划**

用户可以输入多个目标景点或者场所信息，系统会为用户规划从当前位置出发，参观多个景点或者场所的最优旅游线路

核心算法为途经多点最短路径算法，从当前位置出发，参观完返回当前位置。

对于最短的定义要能多样化：最短距离/最短时间/交通工具的最短时间策略

（3）**图形化结果展示**

设计导航功能的图形界面，包括地图展示和输出路径展示；

## 场所查询

（1）**设施的自动搜索**

选中某个景点或者场所，会找出附近一定范围内的超市、卫生间等设施，并根据距离进行排序；

核心算法为距离排序，可以通过选择类别对结果进行过滤；

（2）**设施的指定查询**

可以由用户输入类别名称查找某个地点附近的服务设施，并根据距离进行排序；

核心算法为查找和排序

## 旅游日记管理和交流

（1）**旅游日记的撰写**

用户旅游过程中或者旅游结束时可以撰写旅游日记，通过文字、片和等方式记录旅游内容；

需要对所有用户的旅游日记进行统一的管理（数据库进行统一存储）

（2）**旅游日记的浏览**

用户可以根据浏览和查询所有用户的旅游日记，旅游日记的浏览量即为该日记的热度，每位同学浏览完可以对旅游日记进行评分

用户还可以按日记内容进行全文检索（核心算法为文本搜索）

（3）**旅游日记的推荐**

用户在浏览所有旅游日记时，可以按照不同指标进行推荐

推荐算法基础要求为排序算法，可以根据热度和评分进行排序（核心算法为排序算法）

（4）**旅游日记的搜索**

景点查询：用户可以输入旅游目的地，查找目的地相关的旅游日记

标题查询：用户可以输入旅游日记的名称进行精确查询

1. **旅游日记的压缩存储**

可以对旅游日记进行压缩存储

核心算法为无损压缩：使用哈夫曼编码算法

# 系统架构

## 技术基础与开发环境

* 采用 **C++17** 作为主要开发语言，借助其现代特性（如结构化绑定、if constexpr、更强的模板推导等）提高了代码的可读性与可维护性。
* 基于 **Qt 6.8.3** 框架开发，该版本为 LTS（长期支持）版本，具有出色的稳定性与跨平台特性。
* 使用的操作系统为Windows 11.
* 选用**Qt Creator** 作为开发工具，其作为官方 IDE，天然支持 Qt 项目管理，开发体验优于通用编辑器。
* 使用内置的 **qmake** 构建系统，配置简洁直观。编译器遵循 C++17 标准，兼顾语法现代化与主流平台的兼容性，利于代码重用与后续维护。
* 选用 **MySQL 8.0** 作为后端数据库系统，结合 Qt 中的 SQL 模块，可以在保持类型安全的前提下，实现 SQL 查询、数据绑定与结果处理，避免繁杂的第三方库集成。

## 系统架构设计

本系统采用“主页面 + 多功能模块”式结构。主页面作为登录之后的功能入口，用户可选择进入景点推荐、校园导航或旅游日记任一模块，模块之间彼此逻辑独立，不存在功能调用依赖。

景点推荐、旅游模块以数据库为数据支撑，调用统一封装的数据库操作模块进行信息读取、写入与更新。校园导航模块由于地图数据为静态数据，只需程序初始化时读取，使用基于 JSON，专门用于表示地理空间数据的GeoJSON格式文件存储场所道理信息。该架构提高了系统的内聚性和可维护性，后续模块的添加与扩展无需修改其他模块逻辑，符合高内聚低耦合的设计原则。

项目架构如图所示：

- `school\_map.qrc`, `travel-system.pro`

- `data/`:

`icon.png`, `map.png`, `map\_of\_bupt.geojson`

- `include/`:

`campus\_nav.h`, `database\_connection.h`, `diaryread.h`, `diarywindow.h`, `kmp\_search.h`, `location\_choose.h`, `login\_widget.h`, `mainwindow.h`, `register\_widget.h`, `route\_plan.h`, `site\_recommend.h`, `top\_k\_algorithm.h`, `writewidget.h`

- `simple\_class/`: `diary.h`, `location.h`, `map\_elements.h`, `user.h`

- `tool\_class/`: `HuffmanCoding.h`, `read\_data.h`, `update\_database.h`

- `src/`:

`campus\_nav.cpp`, `database\_connection.cpp`, `diaryread.cpp`, `diarywindow.cpp`, `kmp\_search.cpp`, `location\_choose.cpp`, `login\_widget.cpp`, `main.cpp`, `mainwindow.cpp`, `register\_widget.cpp`, `route\_plan.cpp`, `site\_recommend.cpp`, `writewidget.cpp`

- `tool\_class/`: `HuffmanCoding.cpp`, `read\_data.cpp`, `road\_calcu.cpp`, `update\_database.cpp`

- `ui/`:

`campus\_nav.ui`, `diaryread.ui`, `diarywindow.ui`, `location\_choose.ui`,

`login\_widget.ui`, `mainwindow.ui`, `register\_widget.ui`, `site\_recommend.ui`, `writewidget.ui`

# 实现功能

## 前置功能

* 进入系统后，用户需先进行登录，未注册用户可以先注册随后登录。



* 登陆成功后即进入首页，用户可以选择想进入的页面。



## 旅游推荐

* 用户可以按照“学习，美食，旅游，运动，热度，评分”任意一个标准进行TopK优先队列排序。
* 用户可以搜索景点名称进行查找，进行**模糊匹配。**



* 用户双击选择景点可以直接跳转到该景点对应旅游日记的搜索结果，便于查阅。



## 校园导航

* 用户可以输入场所名称定位自己在校园中的当前位置和添加想去的目的地。
* 用户确定当前位置和目的地后，可以选择规划策略（距离优先/步行时间优先/混合时间优先）规划路线。
* 计算出的路径可以可视化的呈现，绘制在地图中。



* 用户可以自定义查询范围和选择场所类型，查询附近的场所，范围排序基准为**实际步行距离。**
* 查询得到的场所同样可以可视化地标明位置，展示在地图中。
* 用户可以一键添加查询得到的场所至目的地列表，后续可直接导航计算出的路径可以可视化的呈现，绘制在地图中。

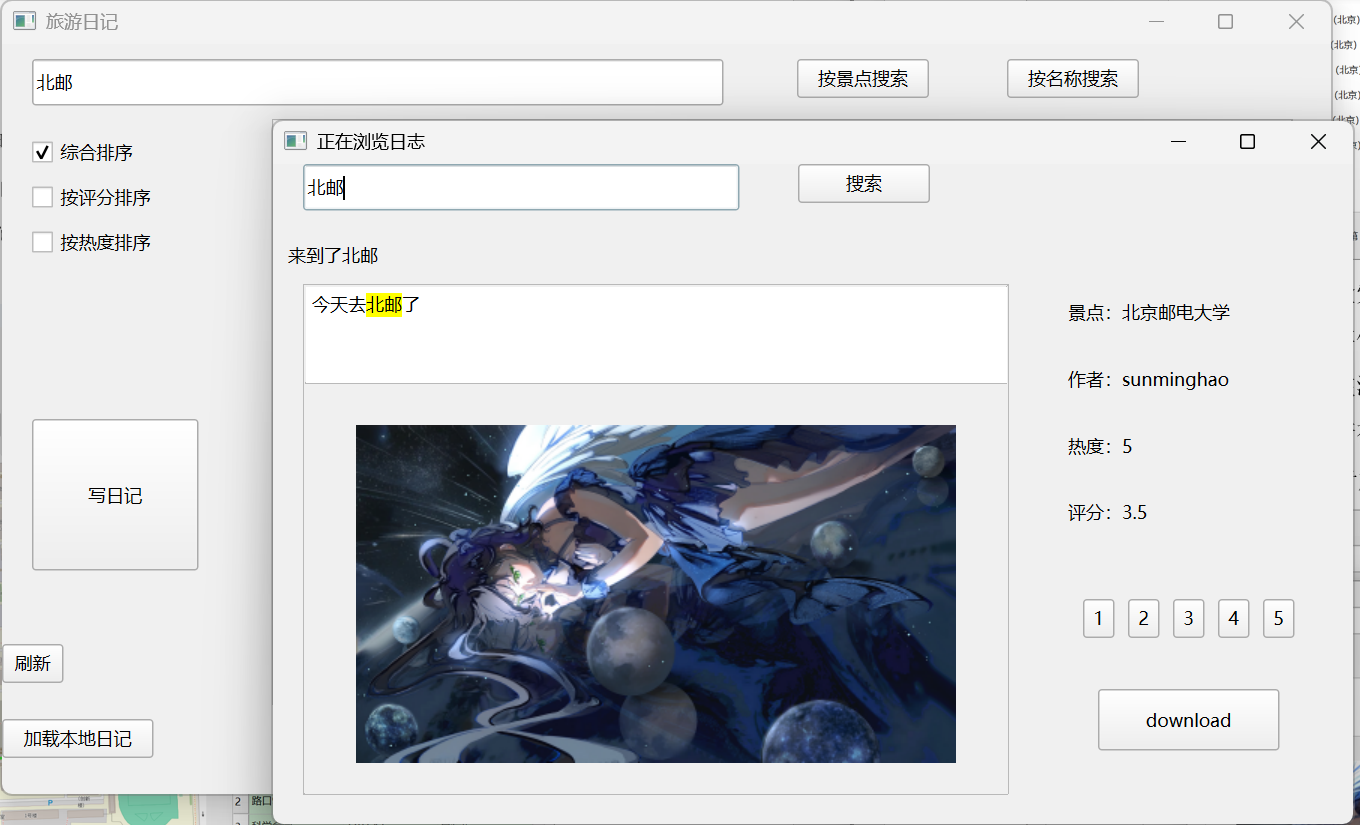


## 旅游日记

* 用户可以输入旅游目的地，对目的地相关的旅游日记根据热度，评分以及日记的综合得分进行排序。
* 用户可以按景点名称或者日记名称进行模糊搜索。



* 可以按日记内容进行全文检索。
* 可以对旅游日记进行压缩存储下载，并且可以通过客户端以本地浏览的形式打开。
* 可以自己撰写对应景点的旅游日记，并且可以**上传图片。**
* 用户可以根据浏览和查询所有用户的旅游日记，旅游日记的浏览量即为该日记的热度，每位同学浏览完可以对旅游日记进行评分（0-5分制）。



# 成员分工

刘梓宸：

* 爬取、编写景区和校园数据以及旅游日记内容；
* 完成关键公用算法，如自实现topK排序（使用堆排序），KMP算法；
* 搭建旅游推荐界面。

胡振宇：

* 设计初期预想的整体数据结构；
* 搭建数据库，存储景区和校园，旅游日记数据以及用户信息；
* 代码实现QT连接数据库，搭建用户登录注册界面；
* 完成旅游日记模块的算法编写和界面搭建。

孙明皓：

* 爬取，设计地图数据；
* 完成校园导航模块（路线规划+场所查询）的算法编写和界面搭建。
* 进行最终整体系统调试，优化算法和用户体验。

# 各模块设计

## 用户管理和数据库模块

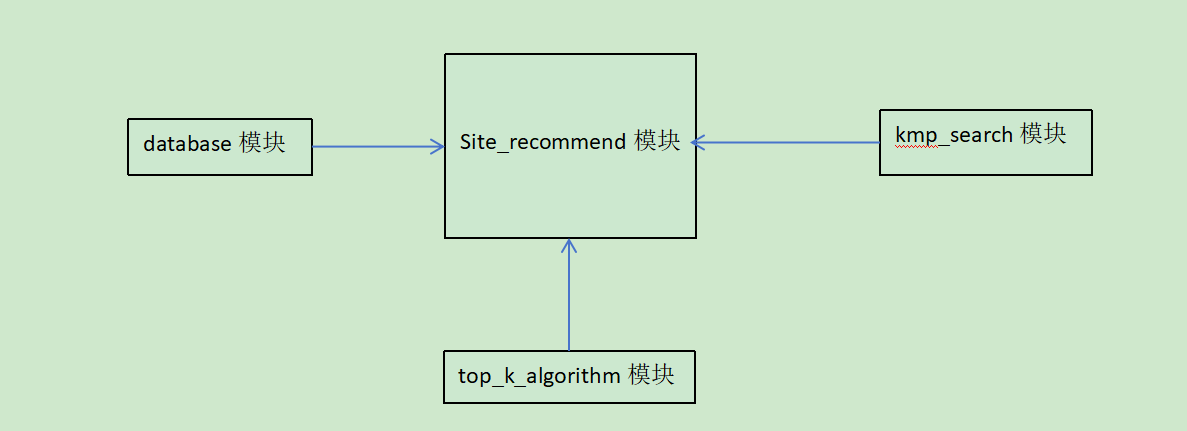
各子模块功能：

database\_connection模块：该模块用于与远程数据库建立连接，采用单例类，仅具有数据库的连接和断开功能。

update\_database模块：该模块负责对数据库数据的CRUD操作。

login模块和register模块：读写数据库中的用户表，实现进入系统时的用户登陆注册基本功能。

## 旅游推荐模块



各子模块功能如下：

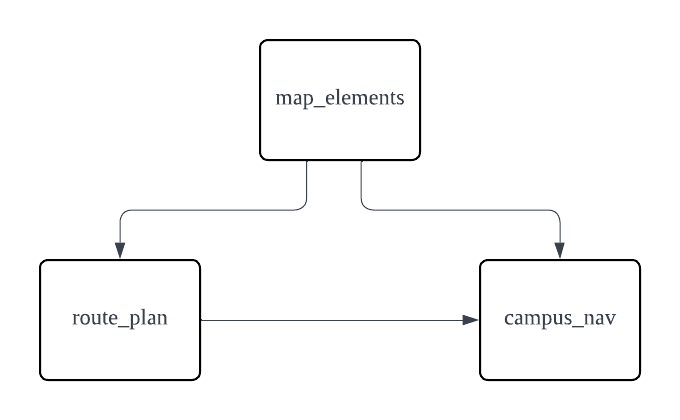
kmp\_search：KMP匹配算法

top\_k\_algorithm：使用堆排序实现的TopK的功能

site\_recommend：实现根据分类标准进行景点排序，支持查找功能（可进行模糊匹配），可只显示部分结果，也可以显示全部的查找结果

database相关模块：包含了database\_connection, read\_data 和 update\_databse模块，负责数据库的连接和crud功能

## 校园导航模块



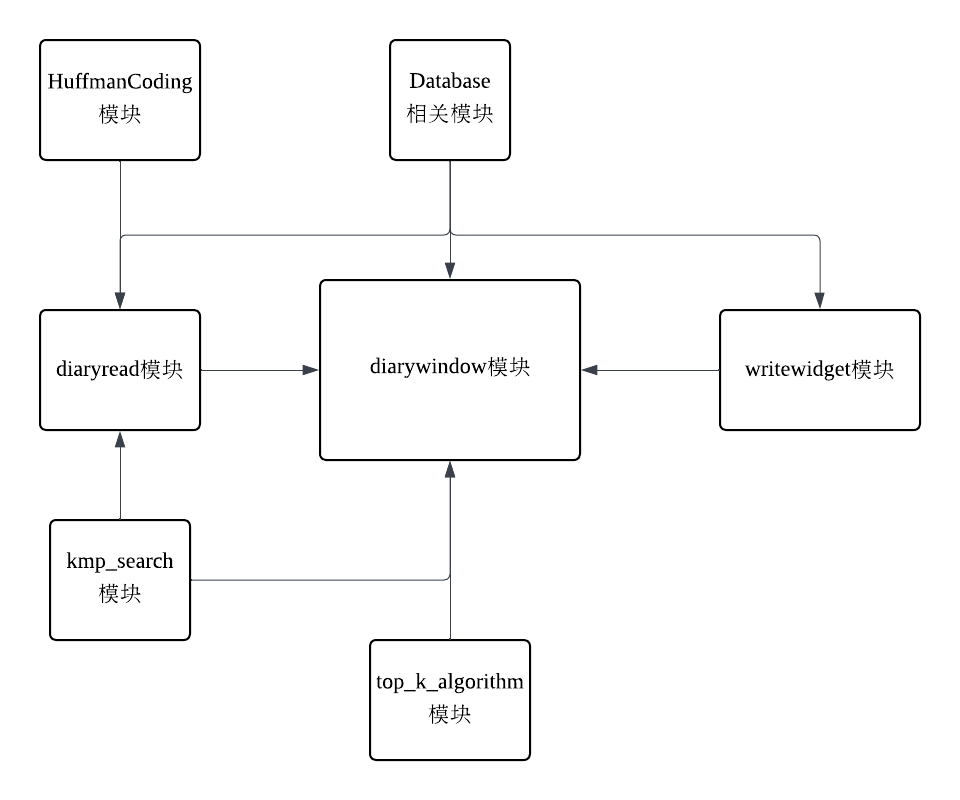
各子模块功能：

map\_elements: 基本的场所、道路等数据结构定义

route\_plan: 底层算法模块，包含地图数据读取、图建立、路线规划、场所查询

campus\_nav: 负责前端页面展示和用户操作交互

## 旅游日记模块



各模块功能：

diarywindow模块：负责旅游日记浏览页面的呈现的功能

diaryread模块：负责旅游日记内容页面的呈现和功能

writewidget模块：负责旅游日记写作页面的呈现和功能

HuffmanCoding模块：包含哈夫曼编码/解码功能，用于提供给旅游日记的下载和本地下载文件的展示

kmp\_search模块：kmp算法实现模块，用于实现文章的全文搜索和文章的标题/景点搜索的功能

top\_k\_algorithm模块：堆排序算法的实现，用于实现旅游日记的不同种类的排序功能

database相关模块：包含了database\_connection, read\_data 和 update\_databse模块，负责数据库的连接和crud功能。

# 算法分析

## 字符串查找算法

本算法用于日记内容的查询，以及旅游推荐和旅游日记模块的模糊搜索（逐一条目遍历匹配标题）。使用*KMP*算法，时间复杂度为 *O*(*m*+*n*)，分为预处理和与模板串匹配两个过程，空间复杂度来源于部分匹配表的 *O*(*m*)。

算法大致思路：首先构建模式串的next数组，然后初始化文本指针i=0和模式指针j=0

逐个比较字符。当字符不匹配时，利用next数组跳过一些比较；当字符匹配时，两个指针都向前移动。如果j=模式串长度，则匹配成功。

算法详细代码：

next数组构建函数：

// 实现部分匹配表构建

std::vector<int> KMP::buildNext(const std::string& pattern) {

int n = pattern.length();

std::vector<int> next(n, 0); // 初始化next数组，全部设为0

for (int i = 1, j = 0; i < n; ++i) // 当字符不匹配时，回退j到next[j-1]的位置

{

while (j > 0 && pattern[i] != pattern[j]) {

j = next[j - 1];

}

// 如果字符匹配，j向前移动

if (pattern[i] == pattern[j]) {

++j;

}

// 记录当前位置的next值

next[i] = j;

}

return next;

}

匹配函数：

// KMP匹配算法实现

bool KMP::kmpMatch(const std::string& text, const std::string& pattern) {

std::vector<int> next = buildNext(pattern); // 构建next数组

int n = text.length();

int m = pattern.length();

for (int i = 0, j = 0; i < n; ++i) // 当不匹配时，利用next数组调整j的位置

{

while (j > 0 && text[i] != pattern[j]) {

j = next[j - 1];

}

// 字符匹配时，j向前移动

if (text[i] == pattern[j]) {

++j;

}

// 完全匹配成功

if (j == m) {

return true;

}

}

return false;

}

算法优缺点：

* 搜索功能：kmp算法能提高匹配效率，但是对于这个功能的场景，用户输入的内容不长，要匹配的目标字符串也不长，实际上并不能发挥出算法的优势。
* 日记内容检索功能：日记内容一般字符量大，kmp的优势可以更好地体现。

## 排序算法

本算法用于旅游推荐和旅游日记界面的条目顺序展示。对于旅游推荐功能，默认只排序和展示前10个条目，点击“加载全部数据”后才排序所有条目并启用翻页功能。此举是为了提高搜索响应的速度，在数据量较大时可以更好的体现。

算法使用自实现堆排序。我们打造了一个通用函数和最小堆，利用C++的模板特性实现。开发人员可以通过传入待排序的向量和相关的比较器，实现任意类，任意比较依据的排序。

算法思路：首先将前k个元素加入最小堆，然后不断弹出堆顶元素，拉入下一元素直至遍历完向量，最后再将堆内元素依次添加到新向量，进行一次逆序后输出。算法时间复杂度为*O*(*n\*logk*)，空间复杂度来源于大小为k的堆为*O*(*k*)

算法详细代码：

TopK排序函数：

template<typename T, typename criteria>

std::vector<T> getTopK(const std::vector<T>& items, int k, criteria cmp) {

my\_pq<T, criteria> minHeap(cmp);

// 初始化堆

for (int i = 0; i < std::min(k, (int)items.size()); ++i) {

minHeap.push(items[i]);

}

// 处理剩余元素

for (size\_t i = k; i < items.size(); ++i) {

if (cmp(items[i], minHeap.top())) {

minHeap.pop();

minHeap.push(items[i]);

}

}

// 提取结果

std::vector<T> result;

while (!minHeap.empty()) {

result.push\_back(minHeap.top());

minHeap.pop();

}

std::reverse(result.begin(), result.end());

return result;

}

自实现堆类：

template<typename T, typename criteria>

class my\_pq {

private:

std::vector<T> heap;

criteria cmp;

// 自定义上浮操作（用于push）

void sift\_up(size\_t index) {

while (index > 0) {

size\_t parent = (index - 1) / 2;

if (cmp(heap[parent], heap[index])) {

std::swap(heap[parent], heap[index]);

index = parent;

} else {

break;

}

}

}

// 自定义下沉操作（用于pop）

void sift\_down(size\_t index) {

size\_t size = heap.size();

while (true) {

size\_t left = 2 \* index + 1;

size\_t right = 2 \* index + 2;

size\_t largest = index;

if (left < size && cmp(heap[largest], heap[left])) {

largest = left;

}

if (right < size && cmp(heap[largest], heap[right])) {

largest = right;

}

if (largest != index) {

std::swap(heap[index], heap[largest]);

index = largest;

} else {

break;

}

}

}

public:

my\_pq(criteria cmp) : cmp(cmp) {}

void push(const T& item) {

heap.push\_back(item);

sift\_up(heap.size() - 1); // 上浮操作

}

void pop() {

if (heap.empty()) return;

std::swap(heap.front(), heap.back());

heap.pop\_back();

if (!heap.empty()) {

sift\_down(0); // 下沉操作

}

}

const T& top() const {

return heap.front();

}

bool empty() const {

return heap.empty();

}

};

算法优缺点：堆排序为不稳定排序，不同的排序量的时候排序结果不完全一致，这导致实际系统中点击“加载全部数据”后首页数据有时会发生变化，但是优点是可以不进行完全的排序，直接获取顺序靠前的结果。

## 路线规划算法

首先进行预处理，在进入页面时，读取地图数据（点+边）并对应三种策略（距离优先/步行时间优先/混合时间优先）计算不同的权值，构建出三张邻接表，便于后续用户操作直接调用。用户导航时调用shortest\_path函数，根据传入的起点编号，目的点编号数组，导航策略对应的邻接表计算最短路径，返回值为最短度量（距离/时间），具体路线信息记录在record向量中。

算法大框架为贪心算法，循环内部采用Dijkstra算法的思路，每次搜索到一个目的地后将其设为新的起点搜索下一个，直至全部目的地均已遍历到。在Dijkstra算法内部，使用最小堆记录尚未找到最短路径的点的当前距离值，时间复杂度优化为*O*(*k\*nlogn*)，其中k为目的地数量。

算法详细代码：

double route\_plan::shortest\_path(int start, const vector<int>& end,

vector<vector<place\_info>>& graph, vector<place\_info>& record) {

const double INF = numeric\_limits<double>::max(); // 无穷大表示不可达

int cur = start; // 当前所在节点

double all\_dist = 0; // 累计总距离

vector<bool> visited(end.size(), false); // 标记每个目标点是否已被访问

// 遍历目标点集合 end，依次找到从当前点 cur 到某个未访问目标点的最短路径

for(int i = 0; i < end.size(); ++i) {

vector<double> dist(graph.size(), INF); // 起点到各节点的最短距离

vector<place\_info> prev(graph.size()); // 最短路径的前驱节点信息

priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;

dist[cur] = 0; // 起点自身距离为 0

pq.push({0, cur}); // 初始化优先队列（最小堆）

int prior; // 用于记录上一轮的起点（以便路径回溯）

// 执行 Dijkstra 算法

while (!pq.empty()) {

auto [d, u] = pq.top(); // 当前距离最短的节点

pq.pop();

// 检查 u 是否是当前未访问的目标点之一

for(int j = 0; j < end.size(); ++j) {

if (u == end[j] && !visited[j]) {

prior = cur;

cur = end[j]; // 更新当前点为新的目标点

visited[j] = true;

goto next; // 找到一个目标点后立即终止当前 Dijkstra 搜索

}

}

if (d > dist[u]) continue; // 跳过非最短路径

for (const auto &neighbor : graph[u]) {

int v = neighbor.getId(); // 相邻节点编号

double weight = neighbor.getWeight(); // 边权（距离）

// 松弛操作

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

// 记录路径来源

prev[v] = place\_info(u, weight, neighbor.getType());

pq.push({dist[v], v});

}

}

}

next:

// 回溯路径：从当前目标点 cur 回溯到 prior（上一次所在点）

int next = cur;

vector<place\_info> temp;

while (next != prior) {

temp.emplace\_back(next, all\_dist + dist[next], prev[next].getType());

next = prev[next].getId(); // 向上追溯前驱节点

}

// 将 temp 中的路径逆序写入 record，确保路径顺序正确

for(auto it = temp.rbegin(); it != temp.rend(); ++it){

record.push\_back(\*it);

}

all\_dist += dist[cur]; // 累加当前段路径距离

}

return all\_dist; // 返回总路径距离

}

算法优缺点：

* + - * 相对于传统TSP问题求法非常高效，只需要线性对数级时间即可求得。
      * 贪心求出的路径不一定为真正的最短路径，可能陷入局部最优陷阱，但实际需求并非需要真正最优的线路，只需要一个大致近似解，考虑到程序性能和实现困难度，此算法可以接受

## 场所查询算法

查询附近的场所时范围排序基准为实际步行距离。search\_place函数会使用Dijkstra算法计算附近各点距离添加到数组中，此时即已按距离远近顺序添加，在达到指定范围时退出循环返回数组，再按场所类型做一次筛选即为最终结果。时间复杂度为*O*(*nlogn*)。

算法详细代码：

vector<place> route\_plan::search\_place(int start, double max\_dist) {

const double INF = numeric\_limits<double>::max(); // 初始化为“无限远”

vector<place> result; // 用于返回满足条件的结果点集

vector<double> dist(graph\_d.size(), INF); // 存储起点到每个点的最短距离

// 最小堆：按当前到某节点的最短距离升序排列

priority\_queue<pair<double, int>, vector<pair<double, int>>, greater<pair<double, int>>> pq;

dist[start] = 0; // 起点距离设为 0

pq.push({0, start}); // 将起点压入堆中

while (!pq.empty()) {

auto [d, u] = pq.top(); // 当前最近的点 u 及其距离 d

pq.pop();

// 如果最短路径已超过 max\_dist，终止搜索（剪枝）

if (d > max\_dist) break;

// 如果 d 不是 u 的最短路径（堆中有旧信息），跳过

if (d > dist[u]) continue;

else {

places[u].dist = d; // 更新该点的距离字段（用于展示等用途）

result.push\_back(places[u]); // 加入结果列表

}

// 遍历所有邻接点，进行 Dijkstra 的松弛操作

for (const auto &neighbor : graph\_d[u]) {

int v = neighbor.getId(); // 相邻节点编号

double weight = neighbor.getWeight(); // 边的权重（距离）

// 如果通过 u 到 v 的路径更短，更新并入堆

if (dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

pq.push({dist[v], v});

}

}

}

return result; // 返回所有在 max\_dist 范围内的点

}

## 压缩算法

压缩算法基于哈夫曼编码，用于日记下载和上传。时间复杂度 *O*(*n*+*k*log*k*)，分为统计频率和构建哈夫曼树两个过程，空间复杂度*O*(*k*)，取决于字符个数（字符频率的存储）。

算法详细代码：

压缩部分算法：

// 生成字符频率映射

std::map<QChar, int> HuffmanCoding::generateFrequencyMap(const QString& input) {

    std::map<QChar, int> frequencies;

    for (QChar ch : input) {

        frequencies[ch]++;

    }

    return frequencies;

}

// 创建哈夫曼树

void HuffmanCoding::createTree() {

    while (q.size() > 1) {

        HaffumanNode\* left = q.top(); q.pop();

        HaffumanNode\* right = q.top(); q.pop();

        HaffumanNode\* parent = new HaffumanNode(left->value + right->value, left, right);

        q.push(parent);

    }

    root = q.top(); // 根节点

}

// 生成编码

std::map<QChar, QString> HuffmanCoding::codeList(const std::map<QChar, int>& data) {

    std::map<QChar, QString> codes;

    for (const auto& pair : data) {

        QString code = searchCode(pair.first, root, "");

        codes.insert({pair.first, code});

    }

    return codes;

}

// 哈夫曼编码搜索

QString HuffmanCoding::searchCode(QChar ch, HaffumanNode\* node, QString s) {

    if (!node) return "";

    if (node->kind == 1) {

        if (node->ch == ch) return s;

        return "";

    }

    QString leftResult = searchCode(ch, node->ltree, s + "0");

    if (!leftResult.isEmpty()) return leftResult;

    return searchCode(ch, node->rtree, s + "1");

}

// 文本压缩

void HuffmanCoding::compress(const diary d) {

    this->d = d;

    QString output;

    data = generateFrequencyMap(QString::fromStdString(d.context)); // 获取字符频率

    for(auto pair : data)

        qDebug() << pair.first << " " << pair.second;

    qDebug() << "1231231";

    for (const auto& pair : data) {

        HaffumanNode\* node = new HaffumanNode(pair.first, pair.second);

        q.push(node);

    }

    createTree(); // 构建哈夫曼树

    codes = codeList(data); // 获取编码

    output.clear(); // 清空输出字符串

    for (QChar ch : QString::fromStdString(d.context)) {

        output += codes.at(ch); // 连接编码

    }

    this->codestirng = output;

}

    return codes;

}

// 哈夫曼编码搜索

QString HuffmanCoding::searchCode(QChar ch, HaffumanNode\* node, QString s) {

    if (!node) return "";

    if (node->kind == 1) {

        if (node->ch == ch) return s;

        return "";

    }

    QString leftResult = searchCode(ch, node->ltree, s + "0");

    if (!leftResult.isEmpty()) return leftResult;

    return searchCode(ch, node->rtree, s + "1");

}

// 文本压缩

void HuffmanCoding::compress(const diary d) {

    this->d = d;

    QString output;

// 获取字符频率

    data = generateFrequencyMap(QString::fromStdString(d.context));

    for(auto pair : data)

        qDebug() << pair.first << " " << pair.second;

    qDebug() << "1231231";

    for (const auto& pair : data) {

        HaffumanNode\* node = new HaffumanNode(pair.first, pair.second);

        q.push(node);

    }

    createTree(); // 构建哈夫曼树

    codes = codeList(data); // 获取编码

    output.clear(); // 清空输出字符串

    for (QChar ch : QString::fromStdString(d.context)) {

        output += codes.at(ch); // 连接编码

    }

    this->codestirng = output;

}

解压缩部分算法：

// 文本解压缩

diary HuffmanCoding::load\_diary(const QString& fileName) {

//打开文件代码

......

    QStringList code = line.split(' ', Qt::SkipEmptyParts);

    int count = 0;

    root = new HaffumanNode(0, nullptr, nullptr); // 创建根节点

    //创建哈夫曼树

    while(count < code.length()){

        QChar character = code[count].at(0);

        QString code1 = code[++count];

        qDebug() << character << " " << code1;

        codes[character] = code1;

        HaffumanNode\* current = root;

        // 遍历编码中的每个字符

        for (QChar bit : code1) {

            if (bit == '0') {

                // 创建左子节点

                if (!current->ltree) {

                    current->ltree = new HaffumanNode(0, nullptr, nullptr);

                }

                qDebug() << "0";

                current = current->ltree; // 移动到左子节点

            } else if (bit == '1') {

                // 创建右子节点

                if (!current->rtree) {

                    current->rtree = new HaffumanNode(0, nullptr, nullptr);

                }

                qDebug() << "1";

                current = current->rtree; // 移动到右子节点

            }

        }

        // 在叶子节点处设置字符

        current->ch = character;

        current->kind = 1; // 设置为带字符的叶子节点

        count++;

    }

    d.author\_name = in.readLine().toStdString();

    d.site\_name = in.readLine().toStdString();

    d.title = in.readLine().toStdString();

    line = in.readLine();

    d.context = getDecompress(line).toStdString();

    d.image\_path = "0";

    file.close();

    return d;

}

QString HuffmanCoding::getDecompress(QString str){

    QString output;

    HaffumanNode\* node = root;

    for (QChar ch : str) {

        node = (ch == '0') ? node->ltree : node->rtree;

        if (node->kind == 1) {

            output += node->ch; // 添加字符到输出

            node = root; // 返回根节点

        }

    }

    return output;

}

    for (const auto& pair : data) {

        QString code = searchCode(pair.first, root, "");

        codes.insert({pair.first, code});

    }

    return codes;

}

// 哈夫曼编码搜索

QString HuffmanCoding::searchCode(QChar ch, HaffumanNode\* node, QString s) {

    if (!node) return "";

    if (node->kind == 1) {

        if (node->ch == ch) return s;

        return "";

    }

    QString leftResult = searchCode(ch, node->ltree, s + "0");

    if (!leftResult.isEmpty()) return leftResult;

    return searchCode(ch, node->rtree, s + "1");

}

// 文本压缩

void HuffmanCoding::compress(const diary d) {

    this->d = d;

    QString output;

    data = generateFrequencyMap(QString::fromStdString(d.context)); // 获取字符频率

    for(auto pair : data)

        qDebug() << pair.first << " " << pair.second;

    qDebug() << "1231231";

    for (const auto& pair : data) {

        HaffumanNode\* node = new HaffumanNode(pair.first, pair.second);

        q.push(node);

    }

    createTree(); // 构建哈夫曼树

    codes = codeList(data); // 获取编码

    output.clear(); // 清空输出字符串

    for (QChar ch : QString::fromStdString(d.context)) {

        output += codes.at(ch); // 连接编码

    }

    this->codestirng = output;

}

        // 在叶子节点处设置字符

        current->ch = character;

        current->kind = 1; // 设置为带字符的叶子节点

        count++;

    }

......

    line = in.readLine();

    d.context = getDecompress(line).toStdString();

......

    return d;

}

QString HuffmanCoding::getDecompress(QString str){

    QString output;

    HaffumanNode\* node = root;

    for (QChar ch : str) {

        node = (ch == '0') ? node->ltree : node->rtree;

        if (node->kind == 1) {

            output += node->ch; // 添加字符到输出

            node = root; // 返回根节点

        }

    }

    return output;

}

算法优缺点：哈夫曼编码是一种无损压缩，而且实现原理简单，但是该功能面对的是中文场景，中文文章的不同字符数量大，在相同文字量的时候与英文比较无法体现哈夫曼编码在数据压缩上的优势。

# 测试评价

系统的设计从课程设计的各个要求出发，考虑了游学过程中的各个环节。整体开发注重对功能的简洁化和人性化。

系统启动是简洁明了的注册和登录窗口，确保用户能够快速开始使用。

登录之后是主界面，用户可以轻松访问所有核心功能，包括旅游推荐、路线规划、场所查询和游学日记。

旅游推荐界面通过多个维度的标签为用户推荐游学地点，同时支持对景点带自动补全的模糊搜索，使用户能够根据自己的偏好快速找到合适的游学目的地。

路线规划界面和场所查询界面在设计中被结合在一起，用户首先可以设定自己所在的位置。对于场所查询，可以设置要查询的场所和范围，得到查询结果。对于导航，而可以设置单个或多个目的地和路线规划策略，实现灵活的路线规划功能。

游学日记界面则允许用户记录和分享自己的游学经历。用户可以标题，景点，以及不同信息进行日记库的搜索排序和筛选，并可以通过评分的方式和别人进行互动，同时还支持了日记文件压缩下载的功能。整体功能全面，且易于上手。

本系统通过图形化界面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。

优点：

* 场景查询和路线规划结合，简化了系统的复杂度，可以更容易理解功能的使用
* 路线规划依据现实的道路情况，且考虑到不同交通方式，更具实际意义
* 搜索功能都配备了自动补全和模糊搜索功能，方便用户的查询
* 排序的标签多样，可以便于用户挑选到感兴趣的景点/旅游日记
* 旅游日记的下载文件可以通过客户端以本地浏览形式打开，增强了互动性。
* 图形化界面简洁，易于上手。

可改进方面：

* 缺少了个性化推荐功能，导致用户兴趣点的搜索更多需要手动获得。
* 地图样本数据少，具有局限性。
* 界面美化程度低，仅具有功能实用性
* 日记的图片上传功能对图片数量存在限制，可能影响用户体验