数 据 结 构

实习报告

题　　目：数据结构实习报告

班　　级：191181

姓　　名：常文瀚

完成日期：2020 年 6 月 10 日

目　录

一、带环、相交链表问题

1.1 问题描述

构造单链表后，将元素值为 m 和 n（从键盘输入，如有多个相同元素值，仅考虑首个出现的元素）的节点建立连接，注意判断节点出现的先后关系，将后面出现的节点（假设为 n）的链域连到先出现的节点（假设为 m），将原 n 节点的后续节点搬迁到原单链表的头部，形成以下双头相交链表(如果使用带头结点的链表，搬迁过程中请自行额外增加一个头节点)；

利用课程 ppt 中关于判断链表是否有环的方法，判断链表是否有环路，并求出环路出现的位置，即 m，n 节点的指针，请使用环路判断函数（输入为链表头指针），不可直接查找 m,n，支持使用 2 个不同的链表头部指针进行查询；

将环路链接取消，恢复成单链表，并根据表头指针（输入）求解链表的中间节点（输出中间节点的值并返回指针），注意：不要将链表转换成数组来求解；

编写函数，判断两个链表是否相交，函数输入为双头相交链表的两个头指针并输出相交的指针，如没有相交则返回 NULL。注意：判断相交是基于指针的比较（判断是否有相同的节点指针出现在两个链表中）而不是节点的值的比较。

1.2 需求分析

（1）为保证数据输入的准确性与安全性，采用单个数据输入的方式录入单链表节点数据；

（2）在录入数据后能选取 m，n 节点的指针对单链表进行剪切，并且能够复原；

（3）利用快慢指针法，判断链表是否有环路，并求出环路出现的位置，即 m，n 节点的指针；

（4）在手动取消环路后，可以检测出两条链表是否相交，若相交则返回两个链表香蕉的指针,若不想交则返回 NULL；

（5）在程序运行时直接初始化有头结点的单链表；

1.3 概要设计

采用单链表的数据结构与 C++语言进行编程，完成题目要求的功能，采用 Qt 对题目最终程序进行可视化

1.3.1 数据结构定义

程序将会涉及如下单链表的数据结构，用 C++语言描述如下：

（1） 定义单链表以及对其操作的函数

```
typedef struct LNode
```

```
{
    ElemType data;//数据域
    struct LNode* next;//指针域
}LNode, * LinkList;
```

基本操作：

static int InitList(LinkList& L,LinkList& T)

初始条件：L 与 T 均为链表初始化函数定义出的两个头节点，其中之后插入新节点将从头结点 L 后面插入，T 节点用于作为剪切链表后的另一个头结点。

操作结果：初始化单链表，生成头结点 L，头结点 T；

static int ListLength(LinkList L)

初始条件：成功初始化链表 LinkList，并且存在头结点 T 与 Y；

操作结果：返回单链表 LinkList 长度；

static bool ListInsert(LinkList& L, int i, ElemType e)

初始条件：成功初始化链表 LinkList，并且存在头结点 T 与 Y；

操作结果：采用后插法，将新元素插入到第 i(1<=i<=length+1)个位置 即 i-1 之后；

static bool ListDelete(LinkList& L, int i)

初始条件：链表 LinkList 存在，并且不只有头结点，应当已经有插入进去的节点；

操作结果：将第 i 个位置的节点删除，即删除 i-1 之后的结点；

static LNode* LocateElem(LinkList L, ElemType e)

初始条件：链表 LinkList 中存在已经插入的数据；

操作结果：按值查找 查找第一个等于 e 的结点 成功返回该结点指针，否则返回 NULL；

功能实现操作：

static void PrintList(LinkList L)

初始条件：链表 LinkList 初始化成功；

操作结果：遍历 LinkList 中的节点，并将他们输出；

static void Insert(LinkList& L,ElemType e)

初始条件：链表初始化成功；

操作结果：封装插入函数，将指定数据生成节点对链表进行后插；

static void Delete(LinkList L)

初始条件：链表初始化成功，且已经插入了若干节点；

操作结果：删除指定的节点；

static void Search(LinkList L,ElemType e)

初始条件：链表初始化成功，且已经插入了若干节点；

操作结果：查找目标节点所在位置，可根据使用场景修改为 bool 类型或返回目标节点；

static LNode* EntryNode(LinkList h)

初始条件：链表初始化成功

操作结果：通过快慢法判断出选择的单链表是否存在环；

1.3.2 模块设计

（1）主程序操作界面；

（2）链表遍历显示、剪切后链表的遍历现实；

（3）新元素的添加模块；

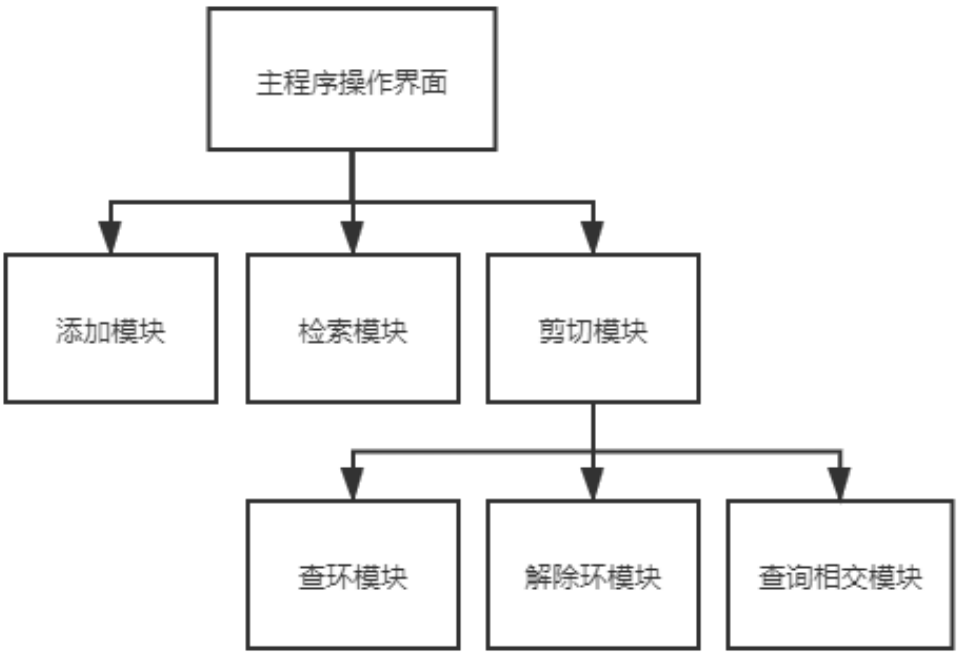
（4）题目中 m，n 的选择，与剪切模块，剪切后可以把剪切的每个段显示在界面中；

（5）查环模块，当查询到环后可以弹出提示框提示环的位置；

（6）解除环路模块，点击后，若环路存在，则可以将环路解除；

（7）查询相交模块，可以将是否存在环路以弹窗的形式告知操作者，以缓解布局冲突问题；

1.3.3 各模块间的调用关系



1.4 详细设计

1.4.1 存储结构设计

```
//单链表结点数据结构
typedef struct LNode
{
    ElemType data;//数据域
    struct LNode* next;//指针域
}LNode, * LinkList;
```

1.4.2 主要算法设计

```
void MainWindow::cutm(int m)
{
    //对 m 前的数据进行剪切
    LinkList p = L->next;//跳过头结点
    QString string_1;

    //剪切头结点到 m
    while (p->data!=m)
    {
        QString element; //转化为字符串
        element = QString::number(p->data);
        string_1 = string_1 + element + " ";

        p = p->next;
    }
    ui->textBrowser_2->setText(string_1);
}



void MainWindow::cutmn(int m,int n)
```

```
{
    LNode* p;
    QString string;
    QString element;
    QString tail;
    p=LocateElem(L,m);  //定位到m节点

    //剪切从m到n
    while(p->data!=n){
        element = QString::number(p->data);
        string = string + element + " ";
        p=p->next;
    }
    element = QString::number(p->data);
    string = string + element + " ";
    ui->textBrowser_4->setText(string);

    //剪切从n到结尾
    while(p->next!=nullptr){
        element = QString::number(p->next->data);
        tail = tail + element + " ";
        p=p->next;
    }
    ui->textBrowser_3->setText(tail);
}


void MainWindow::EntryNodeOfLoop(LinkList L)
{
    LNode* walker = L->next;
```

```
LNode* runner = L->next;
QString msg="从原始链表头 L 检查：发现存在环";
QString prompt="环路出现的位置在";
while (runner != nullptr && runner->next != nullptr)
{
    walker = walker->next;
    runner = runner->next->next;
    if (walker == runner)
    {
        LNode* p = L->next;
        LNode* q = walker;
        while (p != q)
        {
            p = p->next;
            q = q->next;
        }
        if (p == q)
        {
            while (p->next != q)
            {
                p = p->next;
            }
            QString p_data = QString::number(p->data);
            QString q_data = QString::number(q->data);
            QMessageBox msgbox;
            msgbox.setText(msg+"\n"+prompt+q_data+" "+p_data);
            msgbox.addButton("确定",QMessageBox::AcceptRole);
            msgbox.exec();
            break;
        }
```

```
        }
    }
}
void MainWindow::reBuild(LinkList L,LinkList T,int m,int n)
{
    //定位 m
    LNode* p = LocateElem(L, m);
    LNode* q = p;
    while (q->data != n)
    {
        q = q->next;
    }

    //定位 n
    LNode* r = q;
    T->next = q->next;
    while (r->next != nullptr)
    {
        r = r->next;    //定位尾节点 r
    }

    r->next = p;//尾巴指向 p
    q->next = p;//
}


LNode* MainWindow::IsIntersect(LinkList L, LinkList T)
{
    LNode* p = L; LNode* q = T;
    int lenL = ListLength(L); int lenT = ListLength(T);
```

```
if (lenL > lenT) {
    for (int i = 0; i < lenL - lenT; i++) {
        p = p->next;
        i++;
    }
}
else if (lenT > lenL) {
    for (int i = 0; i < lenT - lenL; i++) {
        q = q->next;
        i++;
    }
}

while (p->next != NULL && q->next != NULL)
{
    p = p->next;
    q = q->next;
    if (&p->data == &q->data)
    {
        //qDebug() << "相交在" <<p->data<< endl;
        QString msg="相交在：";
        QString p_data = QString::number(p->data);
        QMessageBox msgbox;
        msgbox.setText(msg+" "+p_data);
        msgbox.addButton("确定",QMessageBox::AcceptRole);
        msgbox.exec();
        return p;
    }
}
```

```
    QString msg="未发现相交：";
    QMessageBox msgbox;
    msgbox.setText(msg+" ");
    msgbox.addButton("确定",QMessageBox::AcceptRole);
    msgbox.exec();
    return nullptr;
}
```

1.5 调试报告

1.5.1 程序在设计过程中主要遇到了如下几个主要的问题：

（1）在我构建剪切函数 reBuild(LinkList L,LinkList T,int m,int n)时，我发现剪切链表后元素与应取得的结果不对应，并且这个现象也会影响到构建出的另一条链表，经过检查是在进行循环时，循环条件由 r->next!=nullptr 写成了 r!=nullptr 导致了节点的分割位置出现问题，后通过单步调试已经解决（参考下图）。

```
//定位n
LNode* r = q;
T->next = q->next;
while (r->next != nullptr)
{
    r = r->next;    //定位尾节点r
}

qDebug() <<"p节点为"<< p->data << endl;
qDebug() <<"q连接"<< q->next->data << endl;
qDebug() <<"r节点为"<< r->data << endl;
qDebug() <<"T节点为"<< T->data << endl;
qDebug() <<"T连接"<< T->next->data << endl;

qDebug() << "length" << ListLength(L) << endl;

r->next = p;//尾巴指向p
q->next = p;//

qDebug() <<"p节点为"<< p->data << endl;
qDebug() <<"q连接"<< q->next->data << endl;
qDebug() <<"r节点为"<< r->data << endl;
qDebug() <<"r连接"<< r->next->data << endl;
qDebug() <<"T节点为"<< T->data << endl;
qDebug() <<"T连接"<< T->next->data << endl;
```

（2）剪切链表时无法从连接到 m 节点前的一系列节点中的第一个节点开始进行遍历以及查环。在编写程序时，无法从新构成的第二个链表开始进行一系列功能的操作，因为针对没有进行剪切的单链表进行操作的函数都是在有头结点的情况下发挥功能，因此我的选择是在初始化链表的时候，构建两个头结点 L 和 T，其中 T 作为第二头结点，在将 n 后面的元素剪切下来后连接到 T 的后面，之后再连接 m 节点，这样就解决了问题。

（3）在编写头文件时，Qt 会报错，经过查询需要将头文件 LinkList.h 中的函数定义为静态函数，添加 static 关键词。在调用该函数的时候，static 函数在内存中只维持一份，而普通函数每次调用都会维持一份拷贝。当然这个静态函数也可以在其他文件中调用了，只是要包含它所在的头文件了。除了在头文件中定义多次引用该头文件不会出现重复定义的，静态函数能保证其他文件调用该函数速度很快，用来输出一些统计信息还是很不错的选择的。如果涉及的计数和多线程，还是要加锁的应该。

1.5.2 程序的扩展方向

（1）用户体验优化的完善。虽然在整个程序中我已经加入了一些保护链表的以及保护程序的机制，以防止操作者的误触导致程序崩溃，但是还有可以进一步完善的问题，例如如果没有对链表进行操作时可以针对解除环路按钮进行保护，以免程序进入死循环。

（2）代码的简化。在 Qt 中，从 LineEdit 控件中获取的文本可以直接本应用于函数的操作，但是我在进行编写时选择了定义 QString 类型的变量，在之后的操作中如果不是需要对用户输入的数据进行修改等操作可以直接以“ui->控件变量名->text()”的形式使用获取的数据。

（3）控件布局的美化。因为本题将一条单链表分解成为了成为了三个部分，所以我采用的界面布局并非左右对称或上下对称等标准美观的布局，为了可以使得数据的展示较为直观，所以采用了这种较为妥协的布局方法，但目前并没有想到很好的解决方法。具体模块及其功能可参考上文。

1.6 体会心得

（1）理解分析问题的能力得到提高。设计一个应用程序关键是对要求做最准确的把握，也就是说弄清楚需求分析是很重要的。单链表的知识点本身难度并不大 ，但是这次课程设计的第一题对单链表的操作与在学校时遇到的操作并不同，因此需要更加认真理解题目需求，以防出现错误操作。首先需要通读完整题目，将所有需求整理，按照操作顺序，逐一解决整个设计框架也逐渐清晰，功能实现的算法也更加容易被想出。

（2）编程能力托到了提升。在经过对题目的分析后，我选择了用可视化的方法解题，并参考互联网上很多实际互联网公司的产品思维对程序进行设计，分别以一个程序员和一个产品工作者的视角对用户的需求以及体验效果进行思考，但因为是第一次采用这样的设计方式，我的设计能力还需要进行更长时间的锻炼和完善，与此同时，代码不够简洁的问题也亟待解决。（程序效果图如下）

（3）对程序设计语言与数据结构的细微之处又了更深刻的理解。在对链表的构建中，我参考了一些网上已经存在的代码，并且综合他们的优点编写了较为简洁整齐的链表类并将其放置在头文件中加以调用，同时在对链表操作的过程中编写新的操作函数完善功能，整个过程让我切身体会到了在自己的程序运行成功时的快乐与成就感，并且更加的热爱编程。

1.7 测试结果

（1）测试用例设计：单链表数据为 1 4 3 2 8 0 7，设 m=3，n=2；

（2）测试结果：

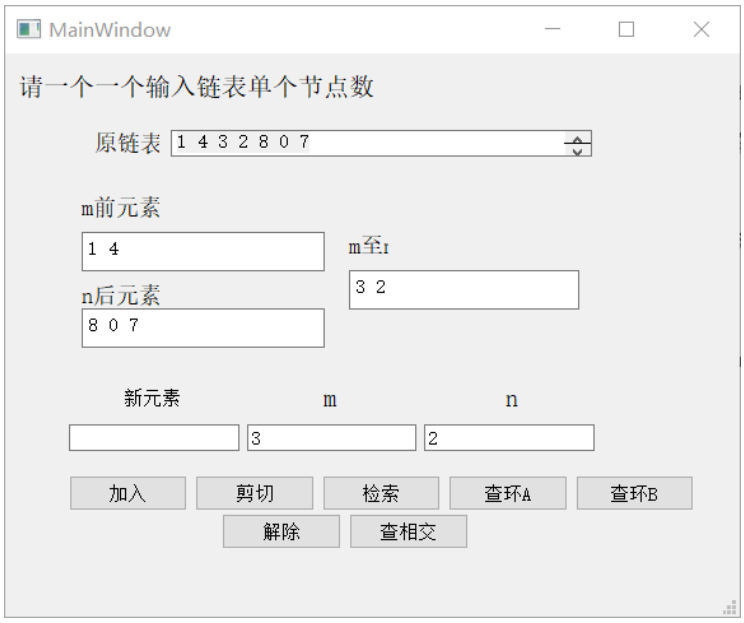


图 1 插入链表信息并进行剪切

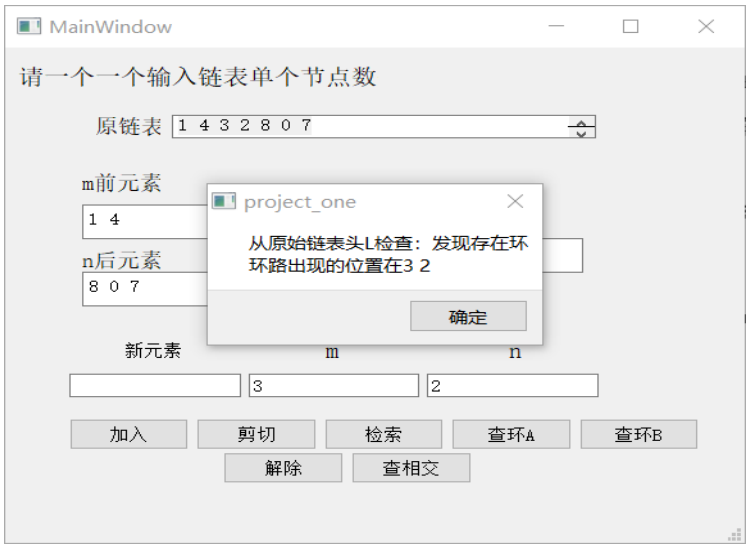


图 2 进行剪切并查看是否有环存在

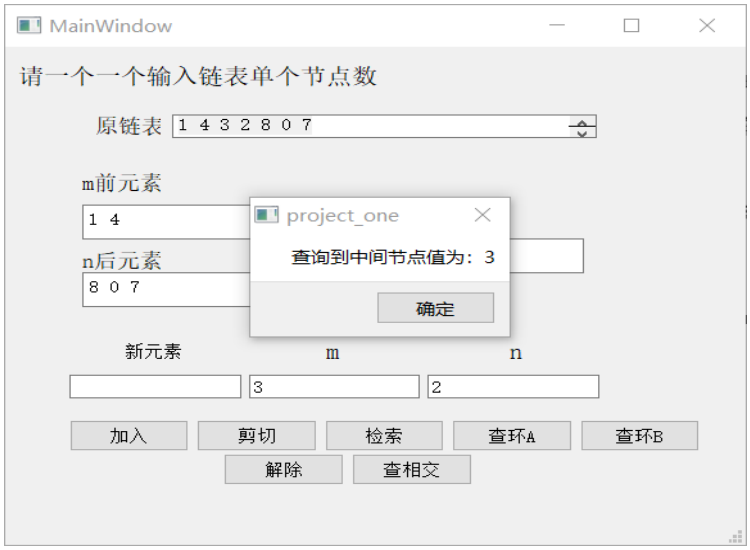


图 3 解除环路并查询其中间节点

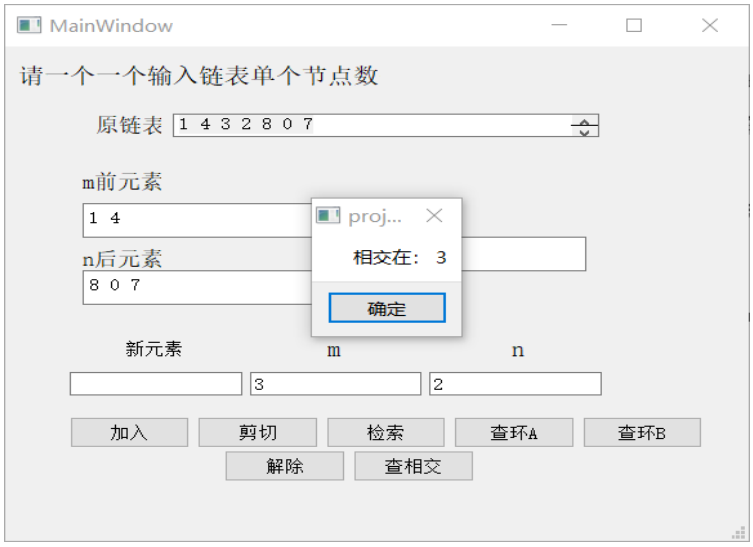


图 4 查询两个链表的相交节点

二、疫情控制

2.1 问题描述

H 国有 n 个城市，这 n 个城市用 n-1 条双向道路相互连通构成*一棵树*，1 号城市是首都，也是树中的根节点。

H 国的首都爆发了一种危害性极高的传染病。当局为了控制疫情，不让疫情扩散到边境城市（叶子节点所表示的城市），决定动用军队在一些城市建立检查点，使得从首都到边境城市的每一条路径上都至少有一个检查点,边境城市也可以建立检查点。但特别要注意的是，首都是不能建立检查点的。

现在，在 H 国的一些城市中已经驻扎有军队，且一个城市可以驻扎多个军队。一支军队可以在有道路连接的城市间移动，并在除首都以外的任意一个城市建立检查点，且只能在一个城市建立检查点。一支军队经过一条道路从一个城市移动到另一个城市所需要的时间等于道路的长度（单位：小时）。

请问最少需要多少个小时才能控制疫情。注意：不同的军队可以同时移动。

2.2 需求分析

（1）输入格式：

第一行一个整数 n，表示城市个数。

接下来的 n-1 行，每行 3 个整数，u,v,w，每两个整数之间用一个空格隔开，表示从城市 u 到城市 v 有一条长为 w 的道路。数据保证输入的是一棵树，且根节点编号为 1。

接下来一行一个整数 m，表示军队个数。

接下来一行 m 个整数，每两个整数之间用一个空格隔开，分别表示这 m 个军队所驻扎的城市的编号。

（2）输出格式：

一个整数，表示控制疫情所需要的最少时间。如果无法控制疫情则输出-1。

2.3 概要设计

采用邻接表的数据结构与 C++语言进行编程，完成题目要求的功能，代码运行采取的是控制台运行

2.3.1 数据结构定义

（1）定义邻接表

```
Int fi[100001], ne[100001], w[100001], v[100001];
```

基本操作：

bool cmp(node u, node v)

初始条件：临接表内存在数据

操作结果：比较节点 u 与节点 v 所存储的距离的大小 ，因为是 bool 类型，所以若 u 节点存储的距离大于 v 所存储的节点的距离则返回 true，反之返回 false，在后面的排序函数会使用。

void addedge(int u, int vv, int z)

初始条件：程序运行，建立了邻接表即可；

操作结果：向邻接表中添加元素，可以是双向的；

void dfs(int u)

初始条件：邻接表构建成功，其中含有用户插入的信息；

操作结果：首先对对整个树形图进行倍增预处理；

void maketi(int u)

初始条件：邻接表构建成功，其中含有用户插入的信息；

操作结果：从根节点开始遍历，军队在时间限制内的已知距离，无法走出根节点则记为时间；

void makeaa(int u)

初始条件：maketi(int u)运行完成，记录完成军队在时间限制内的已知距离或时间；

操作结果：记录可用来弥补的距离及其现在位置(根节点直系子节点)，记录每个子节点军队的最小弥补值的序号；

void makecc()

初始条件：已经存储了需要弥补的根节点直系子节点；

操作结果：将距离从大到小排序；

bool ans()

初始条件：执行完成了 void makeaa(int u)与 void makecc()两个函数，完成了军队距离与城市的排序；

操作结果：将用来弥补的军队与需要弥补的距离进行匹配；

int erfen()

初始条件：匹配的函数运行完成，这是可以求去二分时间；

操作结果：求出二分时间；

void getho(int u)

初始条件：军队与距离的匹配均完成

操作结果：记录该点跳转直系祖先或者该点父节点，距根节点距离；

2.3.2 模块设计

（1）主函数 main；

（2）主函数中要包括预处理倍增模块，对整个树的军队进行预处理记录信息；

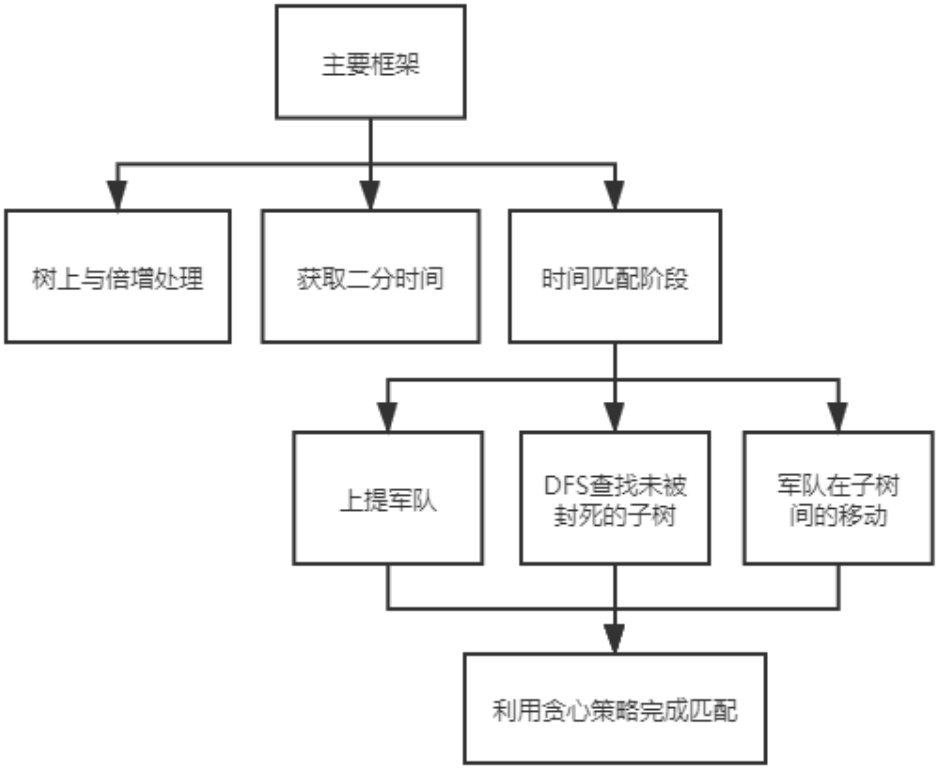
（3）二分答案。军队可以同时移动，说明我们要控制传染病的时间是军队移动到位时，移动时间最长的军队的移动时间。而我们要求最小值，即要求最大化最小值。；

（4）上提“军队，使用倍增的方法将军队在二分出的答案限制内尽力往上”提“，不过不可以到根节点；

（5）dfs 找未被封死的子树，如果一个节点建立了检查点或者它的所有子树都设立了检查点，则说明以这个节点为根的子树已经被“封死”。记录根节点的所有子树中，未被“封死”的子树；

（6）军队在子树间转移；

2.3.3 各模块间的调用关系



2.4 详细设计

2.4.1 存储结构设计

```
struct node {
    int x, v;
}arre[50001], coo[50001];  //多出来的，缺少的
```

2.4.2 主要算法

```
bool cmp(node u, node v)
{
    return u.v > v.v;
}
void addedge(int u, int vv, int z)
{
    w[++cnt] = vv; ne[cnt] = fi[u]; fi[u] = cnt; v[cnt] = z;
}
void dfs(int u)
{
    int maxx = -1, now1 = 0, now2 = 0;
    for (int i = fi[u]; i; i = ne[i])
        if (w[i] != ho[u])  //保证向下走
        {
            dfs(w[i]);
            if (ti[w[i]] == -1) now2 = 1;  //该点需要军队弥补
            if (ti[w[i]] >= v[i]) now1 = 1;  //可以走到该点
            maxx = max(maxx, ti[w[i]] - v[i]);
        }
    if (u != 1 && ne[fi[u]])  //不是根节点且不是独苗（即有多个子节点的非根节点）
    {
        if (now1) ti[u] = max(ti[u], maxx);  //所有点中走得最长的一个记为该点剩下的长度
```

```
        else if (now2) ti[u] = max(ti[u], -1);  //没有军队到达子节点，最低记为-1
        else ti[u] = max(ti[u], 0);  //每个子节点都有军队，但在该点的子节点与该点之间结束
    }
}
void maketi(int u)
{
    for (int i = 1; i <= m; i++)
        if (dis[arm[i]] >= u) ti[arm[i]] = u;  //军队在时间限制内的已知距离，无法走出根节点则记为时间
    dfs(1);  //从根节点开始遍历
}
void makeaa(int u)
{
    for (int i = 1; i <= m; i++)
        if (dis[arm[i]] < u) arre[++na].x = gra[arm[i]], arre[na].v = u - dis[arm[i]];  //记录可用来弥补的距离及其现在位置(根节点直系子节点)
    sort(arre + 1, arre + na + 1, cmp);  //从大到小排序
    for (int i = 1; i <= na; i++)
    {
        if (!cal[arre[i].x]) cal[arre[i].x] = i;  //每个子节点军队的最小弥补值的序号
        else if (arre[cal[arre[i].x]].v > arre[i].v) cal[arre[i].x] = i;
    }
}
void makecc()
{
    for (int i = fi[1]; i; i = ne[i])  //所有需要弥补的根节点直系子节点
        if (ti[w[i]] == -1) coo[++nc].x = w[i], coo[nc].v = v[i];
    sort(coo + 1, coo + nc + 1, cmp);  //距离从大到小排序
```

```
}
bool ans()
{
    if (na < nc) return 0;  //绝对无法弥补
    memset(b, 0, sizeof(b));
    int i = 1, j = 1;  //用来弥补的军队，须弥补的距离
    for (; i <= nc; i++)
        if (!b[cal[coo[i].x]] && cal[coo[i].x]) b[cal[coo[i].x]] = 1;  //有军队刚好可以弥补，无需经过根节点
        else
        {
            while (b[j] && j <= na) j++;
            if (j > na || arre[j].v < coo[i].v) return 0;  //无法弥补
            b[j] = 1; j++;
        }
    return 1;
}
bool pan(int u)
{
    memset(cal, 0, sizeof(cal)); na = nc = 0;
    memset(ti, -1, sizeof(ti));
    maketi(u);
    makeaa(u);
    makecc();
    return ans();
}
int erfen()  //二分答案
{
    int l = -1, r = 999999999;
    while (l + 1 < r)
```

```
    {
        int mid = (l + r) / 2;
        if (pan(mid)) r = mid;
        else l = mid;
    }
    return r;
}
void getho(int u)
{
    for (int i = fi[u]; i; i = ne[i])
        if (ho[u] != w[i])
        {
            if (u == 1) gra[w[i]] = w[i];  //记录该点跳转直系祖先（该祖先为1号节点的子节点）
            else gra[w[i]] = gra[u];
            ho[w[i]] = u; dis[w[i]] = dis[u] + v[i];  //该点父节点，距根节点距离
            getho(w[i]);
        }
}
```

2.5 调试报告

2.5.1 程序在设计过程中主要遇到了如下几个主要的问题：

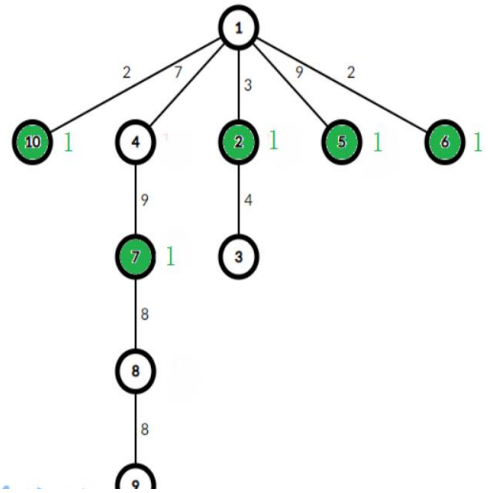
（1）题目理解问题，在解决第一题后阅读这一题感觉难度顿时提高，题目看了很多遍也不是很有思路，同时对一些题目中的点也不能完全理解，在参考查阅了一些文献后，我对该题的解题思路有了初步的认识，在结合某竞赛网站给出的示例图片后，我对本题应当选择的数据结构有了进一步的认识，以下是我参考学习的一部分思路分析和网站上的示例图片。

实现流程：树上倍增预处理+二分答案+贪心

思路：

因为若一个时间限制满足题意，则所有比它大的
案求解。

那么接下来的重点就是check函数了，如何判断
停留的节点深度越小越好，这样可以控制最多的
点，则令其暂时停在根节点的子节点；否则走到
优化。



（2）邻接表知识点遗忘较多，在整理这道题目的资料是，我对上学期邻接表部分的知识遗忘较多，因此花了较多时间去重新复习邻接表数据结构。

（3）参考程序重构较为复杂，在网络上可供参考的多篇代码注释较少，定义的数据结构与算法无法直接看出起到的作用，因此需要花费较长时间阅读理解代码，将代码与算法进行匹配，从而根据思路构建出自己的框架实现需求。

（4）函数逻辑错误，在进行向邻接表添加变得操作时，我没有注意到本题需要在双向边的条件下完成操作，使得军队可以移动，在主函数中添加了修改得以解决该问题。

2.6 体会心得

近一星期的学习与尝试，过程十分曲折，几乎每天都是对着电脑，不然就是翻阅资料。在此期间我失落过，也曾一度热情高涨。点点滴滴令我回味无穷。第二题的解答与学习使我体会到只有做到细心耐心，恒心才能做好事情。

这一题使我懂得了理论与实际相结合是很非常重要的，只有理论知识是远远不够的，只有把所学的理论知识与实践相结合起来,从理论中得出结论，才能提高自己的实际动手能力和独立思考的能力。在整个设计过程中，构思是很花费时间的。调试时经常会遇到这样那样的错误，有的是因为粗心造成的语法错误。当然，也会有很多时候用错了方法，总是实现不了功能。同时在设计的过程中发现了自己的不足之处,对以前所学过的知识理解得不够深刻，掌握得不够牢固。

这一道竞赛题目巩固和加深了对数据结构的理解,提高综合运用本课程所学知识的能力以及培养独立思考，深入研究，分析问题、解决问题的能力。做课程设计同时也是对课本知识的巩固和加强，平时看课本时，有些问题就不是很能理解，做完课程设计，那些问题就迎刃而解了。

2.7 测试结果

（1）测试用例设计

4

1 2 1

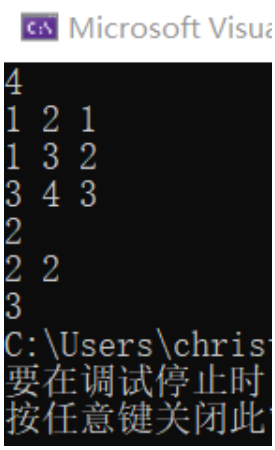
1 3 2

2 4 3

2

2　2

（2）测试结果



三、交通咨询系统设计

3.1 问题描述

设计一个交通咨询系统，通过读取全国城市距离（http://pan.baidu.com/s/1jIauHSE，请在程序运行时动态加载到内存），获取从某一城市到另一城市的最短路径。

1、请验证全国其他省会城市（不包括海口）到武汉中间不超过 2 个省（省会城市）是否成立？（正是因为武汉处于全国的中心位置，此次疫情才传播的如此广）；

2、允许用户查询从任一个城市到另一个城市之间的最短路径（两种算法均要实现，界面上可自行选择）以及所有不重复的可行路径（可限制最多经过 10 个节点），并利用快速排序对所有路径方案依据总长度进行排序输出（输出到文件），每一条结果均需包含路径信息及总长度，试比较排序后的结果与迪杰斯特拉算法和费洛伊德算法输出的结果

3、假设在求解 2 个城市间最短路径时需要绕过某个特定的城市（用户输入或者选择，例如武汉），请问应该如何实现？

3.2 需求分析

（1）验证全国其他省会城市（不包括海口、台北、香港、澳门）到武汉中间不超过 2 个省（省会城市）是否成立

（2）允许用户用两种算法查询从任一个城市到另一个城市之间的最短路径，以及所有不重复的可行路径（可限制最多经过 10 个节点），并利用快速排序对所有路径方案依据总长度进行排序输出（输出到文件），每一条结果均需包含路径信息及总长度。

（3）可以求出绕过某个特定城市的最短路径

3.3 概要分析

程序涉及 C++编程语言，并且采用带权临接矩阵的数据结构，最终用 Qt 进行可视化

3.3.1 数据结构定义

该题目的题解采用了邻接矩阵的数据结构,结合无向图 unordered_map 和队列对数据进行处理。

（1）初始化的参数

```
typedef long long ll;
typedef std::pair<int, int> p;   //想想元组，配对，就把他简化一点吧
const int inf = 0x3f3f3f3f;     //设置一个超大值预存
const int maxn = 50;
const int maxm = 5e3;
bool init_done=false, floyd_done;     //初始化是否完成，弗洛伊德是否进行完成
bool  vis[maxn], adj[maxn][maxn];
int num[maxn][maxn];
int ecnt, head[maxn], dis[maxn], pre[maxn], sta[maxn], g[maxn][maxn]; //g 为带权邻接矩阵
struct edge { int v, w, nxt; } e[maxm];     //定义节点
QVector<QVector<int>> res;    //二维容器
std::unordered_map<QString, int> id; //unordered_map,有点类似哈希表
QString city[maxn];
```

（2）基本操作

```
void MainWindow::addEdge(int u, int v, int w)
```

初始条件：全局变量定义成功即可操作

操作结果：向邻接表中添加城市与距离

void MainWindow::**init**()

初始条件：读取文件，将两个 txt 文件中的数据添加到邻接表中；

操作结果：将给出的文件中的数据添加到邻接表，初始化完成

void MainWindow::**print_wuhan**()

初始条件：邻接表构建成功

操作结果：判断全国其他省会城市（不包括海口、台北、香港、澳门）到武汉中间不超过 2 个省（省会城市）是否成立

void MainWindow::**dfs**(int now, int des, int step, int sum, QVector<int> storage)

初始条件：邻接链表初始化成功；

操作结果：使用 DFS 遍历 A 城市到 B 城市的所有路径，并将他们统一的存储到一个容器中；

bool MainWindow::**compare**(const QVector<int>& va, const QVector<int>& vb)

初始条件：有两个路径存在，并且已经存储了长度可以被用来比较长度；

操作结果：比较两条路径大小，返回 true 或 false；

void MainWindow::**getPath**()

初始条件：邻接表初始化成功，有数据可以使用；

操作结果：获取十步以内的所有从 A 城市到 B 城市的路径；

void MainWindow::**dijkstra**(bool flag)

初始条件：邻接表初始化完成

操作结果：通过迪杰斯特拉算法获取城市 A 到城市 B 的最短路径；

void MainWindow::**floyd**(int n)

初始条件：邻接表初始化完成

操作结果：通过 Floyd 算法获取城市 A 到城市 B 的最短路径；

3.3.2 模块设计

（1）主要操作面板的设计

（2）查询从 A 城市到 B 城市所有步数小于 10 的路径；

（3）验证全国其他省会城市（不包括海口、台北、香港、澳门）到武汉中间不超过 2 个省（省会城市）是否成立；

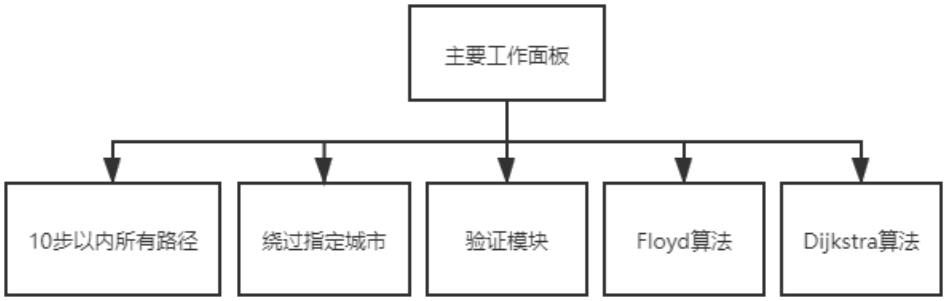
（4）运用迪杰斯特拉算法，求出两个城市间的最短路径；

（5）运用弗洛伊德算法，求出两个城市间的最短路径

（6）选择城市 C，求出城市 A 与城市 B 之间绕过 C 的最短路径；

3.3.3 各模块间的调用关系

主要操作界面下五种功能可以分别使用，并将最终结果显示在 textBrowser 控件内，具体结构图参见第 26 页图片



3.4 详细设计

3.4.1 存储结构设计

```
typedef long long ll;
typedef std::pair<int, int> p;   //想想元组，配对，就把他简化一点吧
const int inf = 0x3f3f3f3f;     //设置一个超大值预存
const int maxn = 50;
const int maxm = 5e3;
bool init_done=false, floyd_done;     //初始化是否完成，弗洛伊德是否进行完成
bool  vis[maxn], adj[maxn][maxn];
int num[maxn][maxn];
int ecnt, head[maxn], dis[maxn];
int pre[maxn], sta[maxn], g[maxn][maxn];   //g 为带权邻接矩阵
struct edge { int v, w, nxt; } e[maxm];     //定义节点
QVector<QVector<int>> res;    //二维容器
std::unordered_map<QString, int> id; //有点类似哈希表
QString city[maxn];
```

3.4.2 主要算法

```
void MainWindow::addEdge(int u, int v, int w)
```

```
{
    e[ecnt].v = v;
    e[ecnt].w = w;
    e[ecnt].nxt = head[u];
    head[u] = ecnt++;
}

void MainWindow::init()
{
    //先进行初始化判断，防止二次初始化
    if (init_done)
    {
        qDebug() << "无需重复初始化" << endl << endl;
        return;
    }
    memset(head, -1, sizeof(head));
    memset(g, 0x3f, sizeof(g));

    QFile in_adj("D:/file_read/boo.txt");    //读取文件
    qDebug()<<in_adj.exists();
    QFile in_dis("D:/file_read/dis.txt");    //读取文件
    qDebug()<<in_dis.exists();

    if(!in_dis.open(QIODevice::ReadOnly)){
        qDebug()<<"fail";
        return;
    }

    if(!in_adj.open(QIODevice::ReadOnly)){
        qDebug()<<"afail";
```

```
    return;
}
QTextStream adjfile(&in_adj);
QTextStream disfile(&in_dis);

//录入城市及其编号
for (int i = 1; i <= 30; i++)
{
    adjfile >> city[i];
    id[city[i]] = i;
}
for (int i = 1; i <= 30; i++)
{
    QString city_now;
    adjfile >> city_now;              //获取城市名
    int id_now = id[city_now];  //获取城市编号
    for (int j = 1; j <= 30; j++)
    {
        //录入1,0，构造对称
        //adjfile >> adj[id_now][j];
        adjfile >> num[id_now][j];
        adj[id_now][j]=num[id_now][j];
        adj[j][id_now] = adj[id_now][j];
    }
}

QString tmp;
for (int i = 1; i <= 29; i++)
{
    disfile >> tmp;
```

```
        qDebug()<<tmp;
    }

    for (int i = 1; i <= 30; i++)
    {
        QString city_now;
        disfile >> city_now;
        int id_now = id[city_now];
        for (int j = 1; j <= i - 1; j++)    //斜着的
        {
            int w;
            disfile >> w;    //w 是距离
            qDebug()<<w;
            if (!adj[id_now][j]) continue;  // 如果为空就 continue
            g[id_now][j] = g[j][id_now] = w;    //对称处理
            qDebug()<<"hello"<<g[id_now][j];
            addEdge(id_now, j, w);
            addEdge(j, id_now, w);  //添加
        }
    }
    qDebug() << "初始化成功" << endl << endl;
    init_done = true;

}

//疯狂遍历
int MainWindow::bfs(int src, int des)
{
    memset(vis, false, sizeof(vis));    //全部设为 FALSE
    queue<p> q;     //定义队列 q
```

```
    vis[src] = true;    //起始城市设置为 true
    q.push(p(src, 0));  //入队
    while (!q.empty())
    {
        int u = q.front().first, s = q.front().second;
        if (u == des) return s - 1;     //相同跳出
        q.pop();

        for (int i = head[u]; ~i; i = e[i].nxt)
        {
            int v = e[i].v;
            if (!vis[v])    //没进来过继续
            {
                vis[v] = true;
                q.push(p(v, s + 1));
            }
        }
    }
    return inf;
}

//去武汉的函数
void MainWindow::print_wuhan()
{
    bool flag;

    for (int i = 1; i <= 30; i++)
    {
        if (city[i] != "武汉" )
        {
```

```
                QString str = "到武汉至少中转";
                QString ci = "次";
                QString value = QString::number(bfs(i, id["武汉"]));
                QString final = city[i] + str + value + ci;
                string conl = final.toStdString();
                //cout << final << endl << endl;
                //zhongzhuan << conl << endl;
            }
        }
        for (int i = 1; i <= 30; i++)
        {
            if (city[i] != "武汉")
            {
                int a = bfs(i, id["武汉"]);
                if (a <= 2) {
                    flag = true;
                }
                else {
                    flag = false;
                }
            }
        }
        if (flag) {
            this->ui->textBrowser->append("全国其他省会城市（不包括海口）到武汉中间不超过 2 个省（省会城市）成立");
        }
        else {
            this->ui->textBrowser->append("全国其他省会城市（不包括海口）到武汉中间不超过 2 个省（省会城市）不成立");
            this->ui->textBrowser->append("具体可查看文件  zhongzhuan.txt");
```

```
        }
    }

    //遍历
    void MainWindow::dfs(int now, int des, int step, int sum, QVector<int> storage)
    {
        if (step > 10) {
            return;         //可以修改步数
        }
        vis[now] = true;
        storage.push_back(now);      //将路径加入 vactor
        if (now == des)
        {
            storage.push_back(sum);      //结束，结算总长度
            res.push_back(storage);        //整个遍历的顺序 vector 存入 res
            return;      //当遍历到了自己说明一圈都遍历了，直接返回就好，把自己这个容器加进 res
        }
        for (int i = head[now]; ~i; i = e[i].nxt)
        {
            int nxt = e[i].v;
            if (vis[nxt])
            {
                continue;
            }
            //采用递归的方法去遍历
            dfs(nxt, des, step + 1, sum + e[i].w, storage);
            vis[nxt] = false;
        }
    }
```

```
//用于比较最后总长度大小
bool MainWindow::compare(const QVector<int>& va, const QVector<int>& vb)
{
    return va[va.size() - 1] < vb[vb.size() - 1];
}

void MainWindow::getPath()
{
    res.clear();    //res 二维容器，清除错误状态
    memset(vis, false, sizeof(vis));    //设置为 FALSE
    QString begin = this->ui->begin->text();
    QString end = this->ui->end->text();

    this->ui->textBrowser->append("请等待，正在计算中");

    //判断输入
    if (id.find(begin) == id.end() || id.find(end) == id.end())
    {
        qDebug() << "地点不存在" << endl << endl;
        this->ui->textBrowser->append("地点不存在");
        return;
    }
    int a = id[begin];  //获取编号
    int b = id[end];
    dfs(a, b, 0, 0, {});    //深度遍历
    sort(res.begin(), res.end(), compare);      //用总长度对 res 中的所有遍历
结果排序
    int cnt = 0;
```

```
        string cong="从",dao="到",zong="总路程为 ",str1,str2;
        QString total = QString::fromStdString(zong),key1;
        int count;
        for (auto vec : res)    // for each 类型的 loop
        {
            QString number = QString::number(++cnt);
            this->ui->textBrowser->append("第"+number+" 条路径");
            for (int i = 0; i < vec.size() - 2; i++){
                count = vec[vec.size() - 1];
                str1 = city[vec[i]].toStdString(),str2 =
city[vec[i+1]].toStdString();
                key1 = QString::fromStdString(cong+str1+dao+str2);
                this->ui->textBrowser->append(key1);

            }
            QString index = QString::number(count);
            this->ui->textBrowser->append(total+index+"\n");
        }
        this->ui->textBrowser->append("限制步数：10，调整函数内部参数可以修改");
    }

    void MainWindow::dijkstra(bool flag)
    {
        memset(dis, 0x3f, sizeof(dis));
        memset(vis, false, sizeof(vis));
        QString begin = this->ui->begin->text();
        QString end = this->ui->end->text();

        if (id.find(begin) == id.end() || id.find(end) == id.end())
        {
```

```
            cout << "地点不存在" << endl << endl;
            return;
        }
        int id_site = 0;
        if (flag)   //做绕开处理
        {
            QString out = this->ui->dont->text();
            if (id.find(out) == id.end())
            {
                cout << "地点不存在" << endl << endl;
                return;
            }
            id_site = id[out];
        }
        priority_queue<p, vector<p>, greater<p> > q;
        /*
            使用优先队列，数据类型 pair(理解为元组不知是否恰当)
            Container 容器类型，Container 必须是用数组实现的容器，比如
vector,deque 等等
            容器类型选择的 vector，vector 类型选择了 pair
            greater 表示了升序队列
            优先队列与队列操作基本相同，只是在这基础上添加了内部的一个排序，它本
质是一个堆实现的
        */
        int a = id[begin];
        int b = id[end];
        q.push(p(dis[a] = 0, a));      //入队
        while (!q.empty())
        {
            int u = q.top().second;
```

```
            q.pop();
            if (vis[u]) continue;
            vis[u] = true;
            for (int i = head[u]; ~i; i = e[i].nxt)
            {
                int v = e[i].v, w = e[i].w;
                if (v != id_site && dis[v] > dis[u] + w)
                {
                    pre[v] = u;
                    dis[v] = dis[u] + w;
                    q.push(p(dis[v], v));
                }
            }
        }

        QString dao = "到",path = "的最短路径如下:",cong="从";
        this->ui->textBrowser->append(begin+dao+end+path+" ");
        int top = 0;
        for (int i = b; i != a; i = pre[i]) sta[top++] = i;
        for (int i = top - 1, last = a; i >= 0; i--)
        {
            int u = last, v = sta[i];
            string chufa2=city[u].toStdString(),zhongdian2 =
city[v].toStdString();
            this->ui->textBrowser->append(cong+city[u]+dao+city[v]);
            qDebug()<<"从"<<city[u]<<"到"<<city[v];
            last = v;
        }
        QString mindis = QString::number(dis[b]);
        this->ui->textBrowser->append("总路程为: "+mindis+"\n");
```

```
}

void MainWindow::floyd(int n)
{
    QString begin = this->ui->begin->text();
    QString end = this->ui->end->text();

    if (!floyd_done)    //查找最小距离并修改矩阵对应值
    {
        for (int k = 1; k <= n; k++)
        {
            for (int i = 1; i <= n; i++)
            {
                for (int j = 1; j <= n; j++)
                {
                    g[i][j] = min(g[i][j], g[i][k] + g[k][j]);
                }
            }
        }
    }
    floyd_done = true;      //修改判别，只运算一次最短路径

    if (id.find(begin) == id.end() || id.find(end) == id.end())
    {
        cout << "地点不存在" << endl << endl;
        return;
    }
    int a = id[begin];
    int b = id[end];     //获取矩阵中的行列坐标
    QString dis = QString::number(g[a][b]);
```

```
        this->ui->textBrowser->append(begin+"到"+end+"的最短路径为:"+dis+"\n");
        qDebug() << endl << begin<< " 到 " << end<< " 的最短路径为: " << dis << endl
<< endl;
    }
```

3.5 调试报告

3.5.1 程序在设计过程中主要遇到了如下几个主要的问题：

（1）首要的问题，是在实现可视化的时候，对信息输出的形式如何设计与处理，在经过思考后，我的选择是使用 Qt 自带的控件 textBrowser 并采用不删除某操作之前操作得到的信息，直接在 textBrowser 下面插入信息的模式(append),这样使操作者能够运用不同功能并保存，以免遗忘，以及能够让操作者能够对不同算法实现的不同路径进行对比。

（2）文件初始化失败的问题。在这个程序中，文件的读取被我设置在了主界面初始化的位置，这样就可以让文件在程序初始化时直接被读取，但是在我调整了文件夹的位置后程序提示初始化失败，经检查将文本文件的位置路径从相对路径改为绝对路径即可解决。

（3)对数据类型的处理。Qt 中存在一些与在运用 C++11 编程时不同的数据类型,例如 QString 等。我先在控制台程序中运行验证算法，再将算法转移到 Qt 中进行可视化，这时就会遇到大量的数据类型转换问题，可参考下图。

```
int a = id[begin];
int b = id[end];      //获取矩阵中的行列坐标
QString dis = QString::number(g[a][b]);
this->ui->textBrowser->append(begin+"到"+end+"的最短路径为: "+dis+"\n");
```

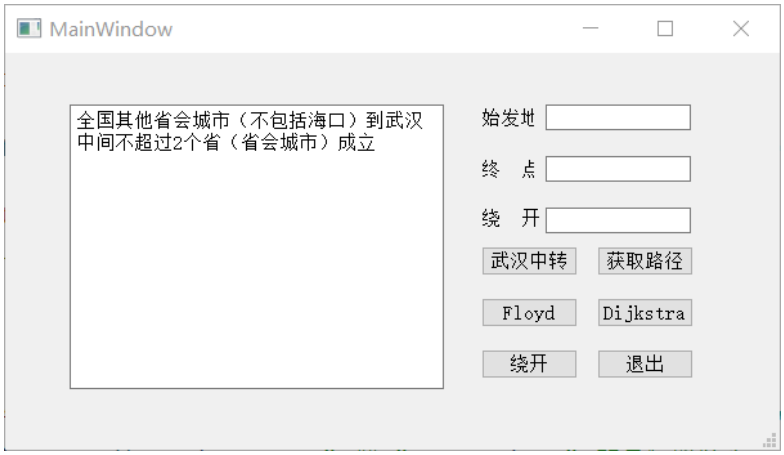
3.6 体会心得

通过本次课程设计，对图的概念有了一个新的认识，在学习离散数学的时候，总觉得图是很抽象的东西，但是在学习了《数据结构与算法》这门课程之后，我慢慢地体会到了其中的奥妙，图能够在计算机中存在，首先要捕捉他有哪些具体化、数字化的信息，比如说权值、顶点个数等,这也就说明了想要把生活中的信息转化到计算机中必须用数字来完整的构成一个信息库。

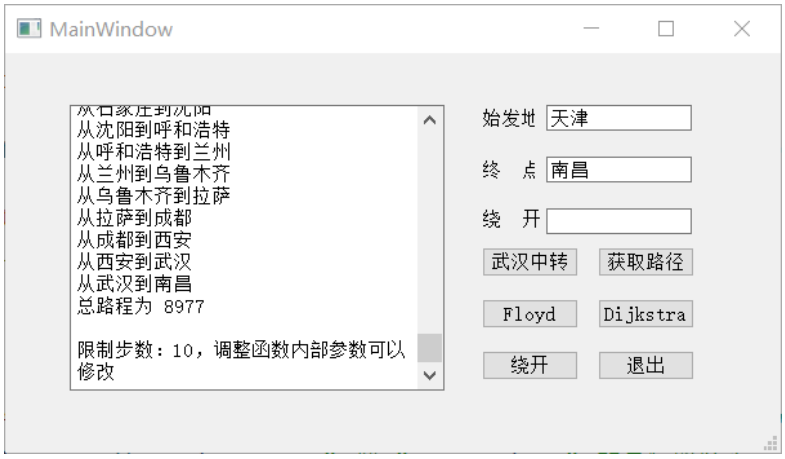
这一题让我更加了解到数据结构的重要性。以及它对我们专业的发展发挥的作用。对我们而言，知识上的收获很重要，但精神上的丰收更加可喜，让我知道了学无止境的道理。我们每一一个人永远不能满足于现有的成就，人生就像在爬山，一座山峰的后面还有更高的山峰在等着你。挫折是一份财富，经历是一份拥有。这一题让我在明白做课程设计时要能够从

多方面去考虑，去研究，用多种算法去实现要求。我学到了很多课内学不到的东西，比如独立思考解决问题，出现差错的随机应变，这些都让我受益非浅，今后的制作应该能够更轻松，自己也都能够解决并高质量的完成项目。

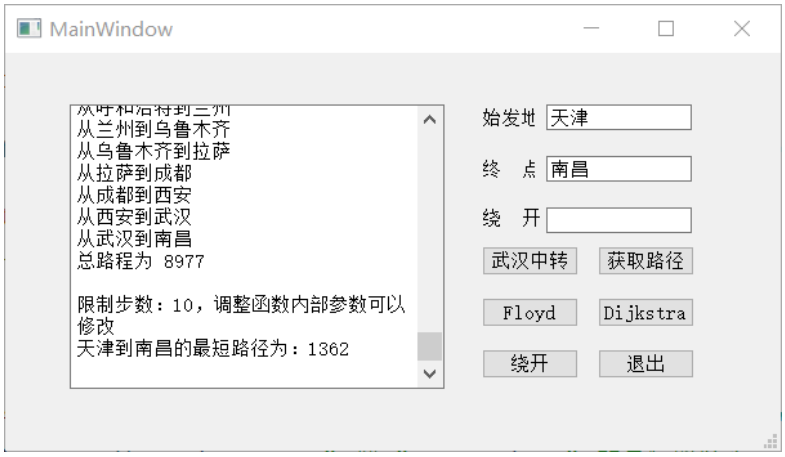
3.7 测试结果



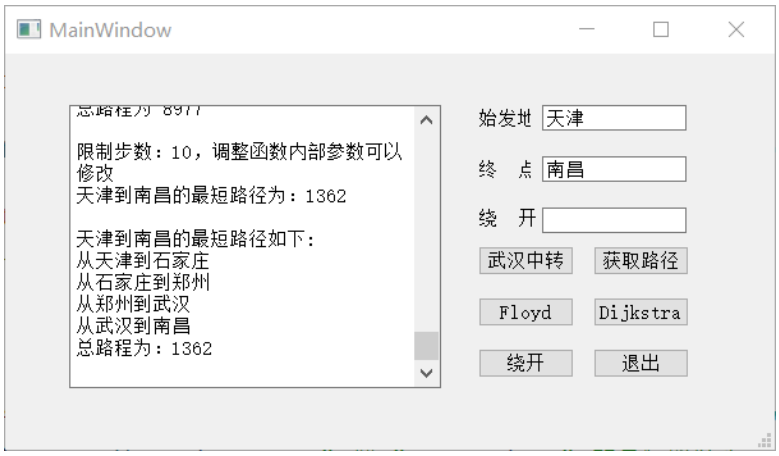
图一 判断其他省会到武汉中间不超过两个省成立



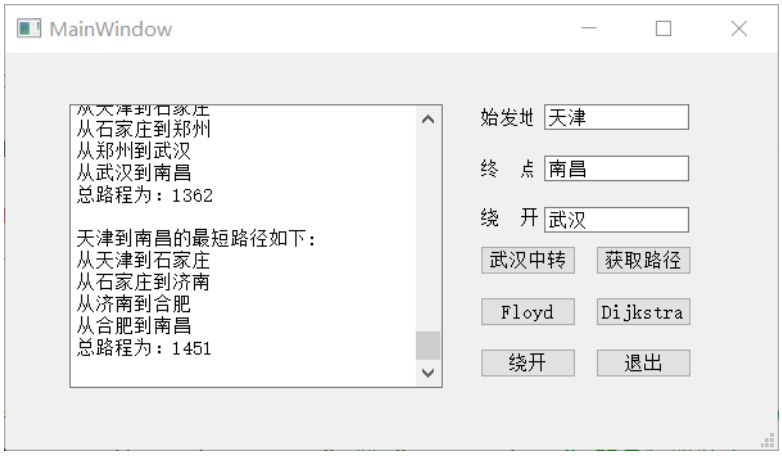
图二 获取指定的十步之内的所有路径



图三 通过 Floyd 算法获取最短路径



图四 通过 Dijkstra 算法获取最短路径



图五 绕开指定城市的最短路径

四、简单搜索引擎系统

4.1 问题描述

给定 n 个文本文件，具体需求如下:

（1）依次读取 n 个文本文件并利用分词函数建立倒排索引；

（2）自行设计查找算法，支持从倒排索引中检索给定的字符串，请提供用户输入查找字符串的页面，相当于百度首页的输入框；

（3）自行设计排序算法，即需要考虑当查询字符串出现在多个文档中时，哪个文档在输出后的结果中排第一 or 第二 or...，可参考 pagerank 算法（可查找相关开源代码并调用），也可自行设计任何可用、有一定意义的算法，简单/复杂均可；

（4）自行设计搜索结果展示界面，要求能将关键字出现的位置附近的上下文显示出来，请

参考百度的搜索结果展示页面，注意，不能在检索后对所有检索结果用字符串模式匹配函数进行关键字查找！

4.2 需求分析

（1）实现爬虫爬取网页数据并保存到本地，并建立邻接矩阵在爬取过程中记录入度出度，用于 pagerank；

（2）对爬取到的内容使用开源代码 jieba 进行分词；

（3）分词后建立倒排索引；

（4）对用户输入进行解析搜索并展示。

4.3 项目分工

高天翔：QT 界面设计、爬虫实现、pagerank 算法实现

常文瀚：资料收集整理与模块的设计、调用 jieba 对数据进行分词、倒排索引实现

4.4 概要设计

以主界面为主体，较慢的爬虫和初始化倒排索引为子线程，较快的分词和搜索直接在主线程中调用， 实现了在线爬取网页、本地载入网页、对网页源代码进行处理、分词、建立倒排索引、根据 PageRank 进行排序等功能。

4.4.1 数据结构的定义

数据结构采用了红黑树、链表、散列表、线性表

使用红黑树进行数据的组织，采用线性表虽然存储简单但是会极大的增加搜寻的时间长度，而采用 AVL 树则会造成维护上的极大困难，所以采用折中的办法，进行初步倒排索引的建立。

每个红黑树的节点除了用来连接上下文关系的部分，其中的数据部分主要分为两个部分，字符串和倒排索引的指针。由于添加操作比较多，所以每个倒排索引的元素指向的索引节采用链表的结构。每个链表记录提及该词条的网站的代号。

读写文件是通过经过处理过的初步倒排索引，建立起以文件为基础的倒排索引库，于是，文件的生成格式以及读取方式也是这个模块的核心。

4.4.2 模块设计

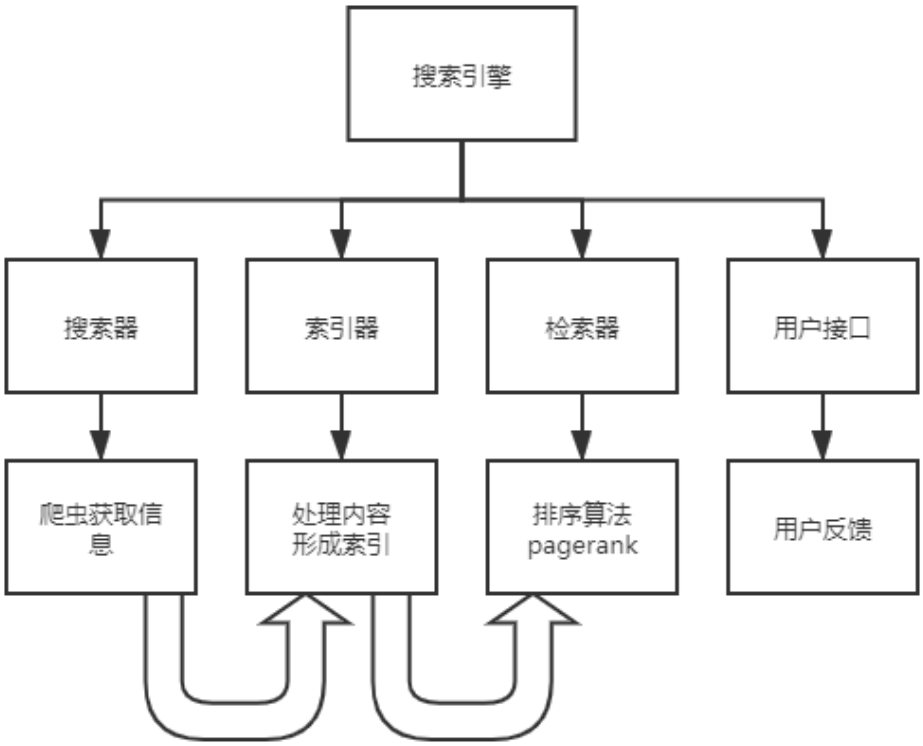
搜索器：爬虫爬取网页数据并保存，将内容送给索引器；

索引器：对爬虫爬取到的内容使用 jieba 分词 进行处理，并建立倒排索引；

检索器：根据用户的输入，对数据进行检索并排序；

用户接口：将查询到的内容以一定的样式展示，将关键字出现位置的上下文显示出来；

4.4.3 各模块间的调用关系



4.5 详细设计

4.5.1 存储结构设计

```
section_entry(std::streampos position, std::fstream &fn) {
    maxSize = (~(0xffffffff << maxEntry_idx));
    numCurrent = 0;
    startpos = position;
    entryConfig = sizeNode;
    H_abandon = 1;
    RID_En flushentry[0x000000ff + 1];
    for (auto &initflush : flushentry) {
        initflush.Valid = initflush.Type = initflush.Fort = 0;
        initflush.tag = 0;
        initflush.init_off = initflush.init_off_H = 0;
```

```
            initflush.name_off = initflush.name_off_H = 0;
            initflush.num_node = initflush.num_node_H = 0;
        }
        fn.seekp(startpos);
        for (unsigned i = 0; i < (0x00001fff + 1) / (0x000000ff + 1); i++) {
            fn.write((char *)(void *)&flushentry, sizeof(flushentry));
        }
        security = true;
    }
    void flush(std::fstream &fn);
    void write(uint32_t name_offset, std::pair<int, int> listInfo,
               uint16_t afterhash, std::fstream &fn);
    uint32_t isAbandon() const;
    std::streampos position() const;
    uint32_t numEntry() const;
    void disArm();
};
class list_section {
    RID64_Nd *section_buf;
    const int bufSize_const;
    unsigned maxSize;
    unsigned bufSize;
    unsigned bufNum_max;
    std::streampos listSection_pos;
    int accCount;
    int ptrCurrent;
    int numCurrent;
    unsigned sizeCurrent;
    uint32_t store(std::fstream &fn, uint32_t size);
```

```
    public:
     list_section(std::streampos position) : bufSize_const(0x00000fff + 1) {
         bufSize = bufSize_const;
         maxSize = (~(0xffffffff << maxList_idx));
         bufNum_max = bufSize / sizeNode;
         section_buf = new RID64_Nd[bufNum_max + 1];
         accCount = 0;
         ptrCurrent = 0;
         numCurrent = 0;
         listSection_pos = position;
         sizeCurrent = 0;
     }
     ~list_section() { delete[] section_buf; }
     std::pair<int, int> write(infoCollector_node *head, std::fstream &fn);
     std::streampos positionCurrent() const;
};
class name_section {
     char *section_buf;
     unsigned maxSize;
     int ptrCurrent;
     int numCurrent;

    public:
     name_section() {
         maxSize = (~(0xffffffff << maxName_idx));
         section_buf = new char[maxSize + 1];
         section_buf[maxSize] = '\0';
         ptrCurrent = numCurrent = 0;
     }
     ~name_section() { delete[] section_buf; }
```

```
    uint32_t write(std::string request);
    std::string read(uint32_t ask_offset);
    uint32_t end() const;
    uint32_t num() const;
    uint32_t store(std::fstream &fn);
    uint32_t store_align_4k(std::fstream &fn);
};

// integral operations
extern std::string INDEX_DIR_PATH;  // 索引文件路径
extern std::string filenameRID;     // RID 文件名
extern int init_stat;               //状态
extern int compulsory_write_RID;    //强制写入符号
```

4.5.2 主要算法

```
/* 对 text 目录下的文本文件进行分词 */
void Widget::divide_text() {
    qDebug() << "pageNumber=" << pageNumber;
    ui->textBrowser->clear();
    ui->labelStatus->setText("正在分词...");

    ui->progressBar->setMinimum(0);
    ui->progressBar->setMaximum(pageNumber);
    ui->progressBar->setValue(0);

    string indexDir = entry.toStdString();
    indexDir += "/text-index";
    //文件存在则进行覆盖
    if (_access(indexDir.c_str(), 0) != -1) {
        _rmdir(indexDir.c_str());
        _mkdir(indexDir.c_str());
```

```
} else  //文件夹不存在新建文件夹
    _mkdir(indexDir.c_str());

/* 对 text 目录下的所有文件进行分词 */
for (int fileNum = 0; fileNum < pageNumber; fileNum++) {
    ui->textBrowser->append(
        QObject::tr("解析%1/text/%2.txt...\n").arg(entry).arg(fileNum));

    /* 打开网页文本文件 */
    QString filename =
        QObject::tr("%1/text/%2.txt").arg(entry).arg(fileNum);
    QFile f(filename);
    if (!f.open(QIODevice::Text | QIODevice::ReadOnly)) {
        QMessageBox::warning(
            this, tr("打开网页文本"),
            tr("打开网页文本文件%1 错误：").arg(filename) + f.errorString());
        qDebug() << tr("Open failed.") << endl;
        return;
    }

    /* 读入文件并分词，存在 wordcounts 容器中 */
    QTextStream txtInput(&f);
    txtInput.setCodec("UTF-8");
    QString lineStr;
    vector<WordCount> wordcounts;
    while (!txtInput.atEnd()) {
        lineStr = txtInput.readLine();
        // ui->textBrowser->append(QObject::tr("%1<Line
        // End>\n").arg(lineStr));
```

```
    vector<string> words;
    jieba.CutForSearch(lineStr.toStdString(), words, true);

    int sz = words.size();
    // qDebug() << tr("words.size()=%1").arg(words.size()) << endl;
    for (int i = 0; i < sz; i++) {
        if (words[i] == " ") continue;

        bool flag = false;
        for (int k = 0; k < wordcounts.size(); k++)
            if (wordcounts[k].word == words[i]) {
                wordcounts[k].count++;
                flag = true;
            }

        if (!flag) {
            WordCount wc(words[i], 1);
            wordcounts.push_back(wc);
        }
    }
    // qDebug() << tr("wordcounts.size()=%1").arg(wordcounts.size()) <<
    // endl;
}
f.close();

ui->textBrowser->append(QObject::tr("Cut Finished\n"));

/* 打开分词目标文件 */
QString indexfilename =
    QObject::tr("%1/text-index/%2-index.txt").arg(entry).arg(fileNum);
```

```
    QFile indexfile(indexfilename);
    if (!indexfile.open(
            QIODevice::Text |
            QIODevice::WriteOnly))  //若要输出换行，需使用 QIODevice::Text
    {
        QMessageBox::warning(
            this, tr("打开分词目标文件"),
            tr("打开分词目标文件%1 错误：").arg(indexfilename) +
                indexfile.errorString());
        qDebug() << tr("Open failed.") << endl;
        return;
    }

    /* 将 wordcounts 写入分词目标文件 */
    // qDebug() << tr("wordcountSize=%1").arg(wordcounts.size()) << endl;
    int wordcountsSize = wordcounts.size();
    QTextStream wcOutput(&indexfile);
    wcOutput.setCodec("UTF-8");
    for (int i = 0; i < wordcountsSize; i++) {
        QString result =
            QObject::tr("%1\t\t\t%2\n")
                .arg(QString::fromStdString(wordcounts[i].word))
                .arg(wordcounts[i].count);
        wcOutput << result;
        // ui->textBrowser->append(result);
    }
    indexfile.close();

    ui->progressBar->setValue(fileNum + 1);
}
```

```
    ui->labelStatus->setText("分词完成");
    QMessageBox::information(this, tr("分词"), tr("分词已完成"));
}

/* 显示单个网页的搜索结果 */
void Widget::show_result(const vector<string>& targetWords, int fileNum) {
    int leftpace = 20;
    int rightpace = 20;

    /* 打开网页并读入 */
    QString fileName = QObject::tr("%1/text/%2.txt").arg(entry).arg(fileNum);
    QFile f(fileName);
    if (!f.open(QIODevice::Text | QIODevice::ReadOnly)) {
        QMessageBox::warning(
            this, tr("打开网页"),
            tr("打开网页文本文件%1 错误：").arg(fileName) + f.errorString());
        qDebug() << tr("Open failed.") << endl;
        return;
    }
    QTextStream txtInput(&f);
    txtInput.setCodec("UTF-8");
    QString text = txtInput.readAll();
    // qDebug() << fileName << " " << QString::fromStdString(get_url(fileNum));
    // qDebug() << text;
    f.close();

    /* 将元素类型为 string 的 targetWords 转为元素为 QString 的 keywords 以便后续使
用 */
    int szTargetWords = targetWords.size();
```

```
vector<QString> keywords;
for (int i = 0; i < szTargetWords; i++)
    keywords.push_back(QString::fromStdString(targetWords[i]));

/* 对本网页分词，结果在 textWords 中 */
vector<Word> textWords;
jieba.CutForSearch(text.toStdString(), textWords, true);
int szTextWords = textWords.size();
//按照 offset 先后排序
for (int i = 0; i < szTextWords; i++)  {
    int least = i;
    for (int j = i; j < szTextWords; j++)
        if (textWords[j].unicode_offset < textWords[least].unicode_offset)
            least = j;
    if (least != i) swap(textWords[i], textWords[least]);
}

/* 遍历 textWords，将关键词相关的语句放到 body 中 */
QString body;
int begin = -1;
int end = 0;
// qDebug() << "targetWords.size()=" << targetWords.size();
for (int j = 0; j < textWords.size(); j++) {
    // qDebug() << "textWords.size()=" << textWords.size();
    for (int i = 0; i < targetWords.size(); i++)
        if (targetWords[i] == textWords[j].word) {
            int offset = textWords[j].unicode_offset;
            int length = textWords[j].unicode_length;
            // qDebug() << "word:" << QString::fromStdString(targetWords[i])
            // << ",offset=" << offset << ",length=" << length;
```

```
                if (begin <= offset &&
                    offset + length <= end)  //若本词已经显示，则直接跳过
                    continue;
                begin = (offset - leftpace >= end) ? (offset - leftpace) : end;
                end = (offset + length + rightpace < text.size())
                          ? (offset + length + rightpace)
                          : text.size();

                body += text.mid(begin, end - begin);
                body += "...";
                body += "\n";
                // qDebug() << "begin=" << begin << "end:";
                // qDebug() << body << endl;
            }
    }
    /* 搜索结果的标题为该网页的URL */
    QString title =
        QObject::tr("http://%1").arg(QString::fromStdString(get_url(fileNum)));

    /* 向textBroswer输出标题和搜索结果 */
    ui->textBrowser->append(str_to_html(title, keywords, fileNum, TITLETYPE));
    ui->textBrowser->append(str_to_html(body, keywords, fileNum, BODYTYPE));
}
/* 初始化倒排索引，调用initRidThread子线程 */
void Widget::init_rid() {
    entry = "www.thepaper.cn";
    ui->labelStatus->setText("正在建立倒排索引...");
    ui->progressBar->setRange(NOT_START, RID_COMPLETE);
    ui->progressBar->setValue(NOT_START);
```

```
    working = true;
    emit sendInitRidParameter(entry, pageNumber);
    initRidThread->start();
}

/* 显示所有搜索到的网页 */
void Widget::show_results(int* resultPages, int szResultPages,
                          vector<string> targetWords) {
    sort_by_pagerank(resultPages, szResultPages);   //根据 PageRank 排序
    sort_by_in_degree(resultPages, szResultPages);  //根据入度排序
    for (int i = 0; i < szResultPages; i++) {
        show_result(targetWords, resultPages[i]);
        ui->progressBar->setValue(ui->progressBar->value() + 1);
    }
}
```

4.6 调试报告

4.6.1 程序在设计过程中主要遇到了如下几个主要的问题

（1）首先，在进行各模块学习的过程中遇到了对倒排索引理解不够透彻的问题，加之受到惯性思维的影响且急于完成任务，我总是把正向索引和倒排索引的结构搞混，在反复记忆后才能够区分二者。

（2）对建立倒排索引理解不够透彻，导致建立倒排索引的函数较为冗杂，需要进一步学习理解，对函数结构做好规划。

（3）在最开始设计完善各模块的时候，没有查询更多的信息做好准备进行构思，导致结构的设计更迭多次，对小组进度有较大影响，准备工作应当深思熟虑后完成，我应当吸取这次教训。

4.6.2 程序的扩展方向

该程序可扩展的地方还有很多，值得我们更深入理解其中的分词原理等一系列算法知识，其中分词和建立倒排索引的地方我曾考虑过多线程处理，但因为这方面知识了解甚少，无奈只好暂时放弃，并且在用户搜索是没有处理好突出信息重点的问题也值得努力处理。

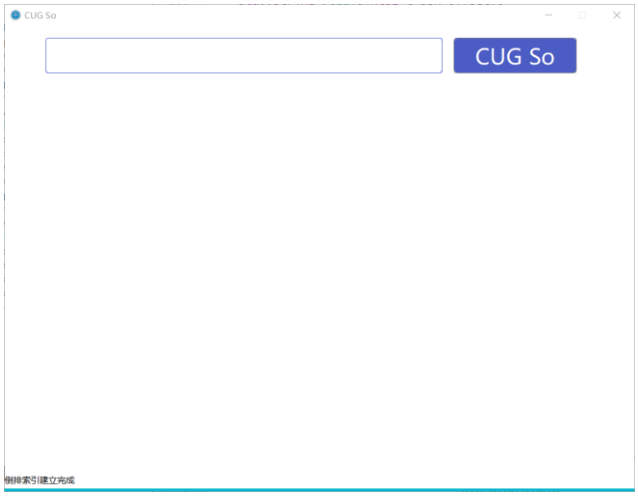
4.7 体会心得

自从上大学以来，这道题目堪称难度最高的一道课程设计题目，从前期准备到不同模块的构建与实现，再到最终可视化效果的优化，每一步我们小组都付出了大量的努力，并且为之付出了许多个白天与夜晚，在搜索引擎能够跑通时的喜悦又难以言表。经过这次数据结构的课程设计后，我更加深刻的认识到了数据结构在现实生产中对整个任务的架构与完成的重要性，以及对推进整个任务进度时的核心地位。在我们完成这项任务后，我们也在不断反思在整个过程中需要有哪些地方在今后的学习中完善，例如从数据结构出发，如果时间更加充裕，我们或许可以尝试用更多的思路来对某一模块进行实现，这样就可以比较不同思路下代码的简洁性与实现需求时的难易程等。

我们也跟深刻的注意到了学习并锻炼使用课堂上不会学到的知识与技能的重要性，在很多招聘网站中用人单位大部分的需求或许在校园内并不会学到，例如 Vue.Js,TensorFlow，SpringBoot 等目前开发程序会用到的主流框架，这就要求我们在课后用更多时间去学习并掌握最近的技能，锻炼自己快速掌握一项技能的能力。

最终，我也要感谢我的队友和老师，感谢我的队友高天翔可以督促我快速完成前三题并对第一、三题进行可视化，以及他在这段时期时时刻刻保持着高度的热情，带动我投身到这一项课程设计的完成工作中。同时更不能忘记老师对我们的辛苦教育以及姜老师在上机实验时的严格，通过四次上机的验收让我更深刻的意识到了要同时保持对自己程序结构与代码的深刻理解，无疑这将会是我在大学时期获得的宝贵财富之一，若非上过这门数据结构，我想我不会明白数据结构在计算机科学这个世界中独一无二的地位。

4.8 测试结果



图一 初始化界面

图二 搜索界面 1

图三 搜索界面 2

五、参考资料

1. https://time.geekbang.org/column/article/234839

2. https://time.geekbang.org/column/article/88774

3. https://time.geekbang.org/column/article/98998

4. https://www.cnblogs.com/ysjxw/archive/2008/04/11/1148961.html

5. https://www.cnblogs.com/ysjxw/archive/2008/04/11/1148963.html

6. https://www.cnblogs.com/ysjxw/archive/2008/04/11/1148964.html

7. https://www.cnblogs.com/ysjxw/archive/2008/04/11/1148965.html

8. 姜鑫维，赵岳松. Topic PageRank: a Search Engine Based on Topic%Topic PageRank——一种基于主题的搜索引擎[J]. 计算机技术与发展, 2007, 017(005):238-241.