社交网络分析系统 Report

22307140078 杨永卓 人工智能(计算机班)

一． 项目总体介绍

1. **整体思路**：本系统要求建立社交网络分析系统，涉及许多图论的知识，综合考虑了邻接矩阵、邻接表、关联矩阵以及十字链表。考虑到 data 数据未知时使用邻接矩阵有可能浪费巨大空间，且后续计算时对超大的二维矩阵计算会有很大的不方便，故使用邻接表的方法建立图并进行后续操作。
2. **数据结构的建立**：数据结构参考了书中邻接表的结构，建立了点、边、图三个数据结构，其中对边进行了简化，略去了权的操作。

```
struct Vertex {
    int data;
    Edge* adj;
};
```

点的数据机构

```
struct  Edge {
    int name;
    Edge *link;
    Edge(Edge *l=nullptr):link(l) {};
    Edge(int num) :name(num), link(nullptr) {}
};
```

边的数据结构

```
class GraphLnk {
public:
    GraphLnk(int sz);
    ~GraphLnk();
    int getValue(int i) {
        return (i >= 0 && i < numVertices) ? NodeTable[i].data : 0;
    }
    //图的基础操作
    bool insertVertex(const int& vertex);   bool removeEdge(int v1, int v2);
    bool removeVertex(int v);          bool insertEdge(int v1, int v2);
    int getFirstNeighbor(int v);   int getNextNeighbor(int v, int w);
    int getVertexPos(const int vertex) {
        for (int i = 0;i < numVertices;i++)
            if (NodeTable[i].data == vertex) return i;
        return -1;
    }
    int direct_friend_num(int v);//简略统计直接朋友
    bool areConnected(int v, int w);    void DFS(int node, bool* visited);
    //用来提取封装掉地数据
    int getmaxVertices() { return maxVertices; };  int getnumEdges() { return numEdges; }
    int getnumVertices() { return numVertices; }

    Vertex* NodeTable;//封装入private会带来诸多不便，为节约时间没有封装

private://对类中的部分信息封装，体现对用户信息的保护
    int maxVertices;
    int numEdges;
    int numVertices;
};
```

图的数据机构

3. **实现功能**：按照题目要求，实现了查询用户的直接朋友、间接朋友数量、计算两个用户的最短社交距离、计算超级连接者、朋友推荐功能。下面将介绍各个功能实现的大致思路。

(1) **查询直接朋友与间接朋友**：直接朋友是与用户之间有边连接的点，因此直接统计邻接表中与用户的关联的点的数量即可。而间接朋友是该用户的朋友的朋友，使用 BFS 的思想，统计用户所有直接朋友的直接朋友，即 BFS 树的深度为 2 时的所有用户即可。（关键点：在 BFS 统计间接朋友时，发现有些朋友已经在直接朋友中统计，为避免重复统计，使用一个叫做 look 的数组，将已经遍历到的用户标记、避免重复计数）

(2) **计算两个用户的最短社交距离**：计算最短距离之前需考虑两个用户是否在同一个连通分量上，因此先使用 areConnect()函数判断是否在同一连通分量上。此功能核心算法为 BFS 算法，即以其中一个需要观察的用户为原点，逐层向外扩张，直到找到对应的另一个用户，此时向外扩张的层数即为他们的最短社交距离。此算法中的难点为记录向外扩张的层数，为了确保将第每一层全部遍历之后再计数一次，算法使用了两个 queue，一个用于 BFS，另一个记录遍历数量，当记录到第一层全部遍历完成后，允许计数一次。

(3) **计算超级连接者**：该算法通过计算每一个用户的直接朋友数量，求具有最多直接朋友的用户即可

(4) **朋友推荐功能**：该功能可转换思路，求具有最多共同朋友，即求两个用户之间拥有最多的相同直接朋友，换个角度，即一个用户的间接朋友中出现次数最多的，就是潜在可推荐的朋友。因此统计一个用户的间接朋友出现的次数，找出其中出现次数最多即可。(该功能的时间复杂度与空间复杂度分析见后文函数具体介绍)

二． 具体函数介绍

(一). 类函数：

GraphInk：构造函数
~GraphInk：析构函数
insertVertex：插入用户
insertEdge：插入边
getValue：得到用户的姓名
getVertexPos：得到用户在邻接表 NodeTable 中的位置
direct_friend_num：用类函数的方式得到一个用户的相邻结点个数。通过直接遍历对应的结点的链表直接计数即可

getFirstNeighbor：得到第一个邻居
getNextNeighbor: 得到下一个邻居
getmaxVertices：得到被封装的最大用户的数量
getnumVertices：得到被封装的现有用户的数量
getnumEgde: 得到被封装的现有关系数量

areConnected：判断图中两个结点是否是连通的。该函数中建立了一个与图中节点个数相同大小的布尔类型的数组，调用 DFS 函数来遍历整个图，若从一点出发经过深度遍历之后发现另一点被标记过，则说明两点在同一连通分量，否则两点不在同一连通分量。
DFS：深度遍历图。通过递归深度遍历一个图，遍历到的点在对应的 visit 数组中被标记。

注：这两个函数的设计是为了在计算用户之间的最短社交距离时排除两个用户不在同一连通分量中的情况。

```
bool GraphInk::areConnected(int v, int w) {
    bool* visited = new bool[numVertices];
    for (int i = 0; i < numVertices; ++i) {
        visited[i] = false;
    }
    DFS(node: v, visited);
    bool result = visited[w];
    delete[] visited;
    return result;
}

void GraphInk::DFS(int node, bool* visited) {
    if (visited[node]) {
        return;
    }
    visited[node] = true;
    Edge* neighbor = NodeTable[node].adj;
    while (neighbor != nullptr) {
        DFS(node: neighbor->name, visited);
        neighbor = neighbor->link;
    }
}
```

这里展示两个 areConnected 以及 DFS 的实现

(二). 功能函数：

1.friend_numbers：该函数用于实现查找用户的直接朋友与间接朋友数量，直接朋友通过遍历用户所连接的链表的数量求得，而间接朋友需要用二重循环，求用户的直接朋友的直接朋友数量。此函数中需要考虑重复的问题，因此使用一个布尔型数组 look，用来记录已经被遍历到的用户（如下图所示）。

```
bool* look = new bool[human.getmaxVertices()];//用来标记用户是否被遍历
for (int i = 0;i < human.getmaxVertices();i++) {
    look[i] = false;
}
```

```
//检查用户是否存在，避免出现野指针等问题
if (temp<0 || temp>human.getnumVertices()) {
    outputFile << "用户不存在" << endl;
    delete look;
    return;
}
```

避免出现野指针导致程序无法运行的问题，加入一个判断

该函数在计算间接朋友时，只遍历了部分用户，时间复杂度很难用总结点数 n 和边数 e 来表示。在计算时建立的 look 数组，大小为图中用户数量，所以空间复杂度 O(n)。

2.shortest_road：该函数用于计算两个用户之间的最小距离，使用 BFS 的方法逐层向外扩张寻找另一个用户。函数开始时先调用 areConnect 函数来判断两个用户是否在两个用户之间是否存在路，然后使用 BFS 进行层序遍历，直到找到另一个用户停止。时间复杂度与空间复杂度与具体用户有关，难以表示。这里比较特殊在于使用了两个队列来记录总的层数，record_queue 用来回溯上一层的节点，record_son_num 用来记录上一层中邻接点的数量，当技术达到上一层邻接点的数量时，就将 record_son_num 中的上一层节点数抛出，同时记录层数加一，代码实现如下

```
record_queue.push(_Val:v);
record_son_num.push(_Val:human.direct_friend_num(v));
```

两个队列

```
while (!record_queue.empty()) {
    int temp = record_queue.front();
    Edge* q = human.NodeTable[temp].adj;
    record_queue.pop();
    while (q != nullptr) {
        if (look[q->name] != true) {
            look[q->name] == true;
            record_queue.push(_Val:q->name);
            record_son_num.push(_Val:human.direct_friend_num(v:q->name));
        }
        if (q->name == w) {
            outputFile<<v<<"和"<<w << "的最短距离为" << record << std::endl;
            return;
        }
        q = q->link;
    }
    record_friend++;
    if (record_friend == record_son_num.front()) {
        record++;
        record_friend = 0;
    }
}
```

输出
显示输出来源(S): 调试
"big_homework.exe"
"big_homework.exe"
"big_homework.exe"
"big_homework.exe"

shortset_road 具体实现

3.super_friend：该函数用于实现寻找超级连接者，思路为将每一个节点的直接朋友数求出，然后记录最大者。此函数需要遍历所有的用户，并对每一个用户都遍历所有相邻节点，故时间复杂度为 $O(n^2)$，其中 n 为用户数量。

4.findMax：该函数是 suggest_friend 中调用的一个辅助函数，用于找出 suggest_friend 函数中记录间接朋友出现次数的数组中最大的元素标号。

5.suggest_friend：该函数先记录一个用户的直接朋友，记录于 direct_friend 中，然后从这些结点出发，统计这些结点的直接朋友的数量，并记录于 suggest_friend 数组中，随后调用 findMax 函数来统计出现次数最多的结点，这些结点即为可推荐的朋友数量。

```
int* direct_friend = new int[human.getmaxVertices()];
int* suggest_friend = new int[human.getmaxVertices()];
for (int i = 0;i < human.getmaxVertices();i++) direct_friend[i] = -1;
for (int i = 0;i < human.getmaxVertices();i++) suggest_friend[i] = 0;
```

两个记录数组

```
Edge* p = human.NodeTable[v].adj;
while (1) {
    if (p != nullptr) {
        direct_friend[p->name] = 1;
        p = p->link;
    }
    else break;
}
```

记录所有的直接朋友

```
for (int i = 0;i < human.getmaxVertices();i++) {
    if (direct_friend[i] == 1) {
        Edge* p = human.NodeTable[i].adj;
        while (1) {
            if (p->link != nullptr) {
                if(p->name!=v) suggest_friend[p->name]++;
                p = p->link;
            }
            else break;
        }
    }
}
```

记录直接朋友的直接朋友出现的次数

时间复杂度：时间复杂度主要由三部分构成：第一个 while 循环，第二个 for 与 while 的嵌套，第三个 findMax 的调用。第一个 while 循环遍历了用户的所有直接朋友，假设记为 n_1，第二个 for 循环遍历了每一个直接朋友的相邻结点，因此，记作 $n_{11}, n_{12},\cdots$，故遍历次数为 $n_1*(n_{11}+n_{12}+\cdots)$，findMax 中遍历了所有的结点 n，故时间复杂度为 $O(n^2+n+n)$,即为 $O(n^2)$。

空间复杂度：本函数空间消耗较大，建立了三个大小为 n 的数组（函数本身两个，findMax 中一个），因此空间复杂度为 O(n)。

三． 测试样例及结果

1. 查询用户的直接朋友与间接朋友数：

用户 5 的直接朋友有 13 个,间接朋友有 334 个
用户 321 的直接朋友有 3 个,间接朋友有 344 个
用户 85 的直接朋友有 14 个,间接朋友有 333 个
用户 47 的直接朋友有 2 个,间接朋友有 345 个
用户 96 的直接朋友有 9 个,间接朋友有 338 个
用户 12 的直接朋友有 1 个,间接朋友有 346 个
用户 98 的直接朋友有 49 个,间接朋友有 298 个
用户 485 的直接朋友有 5 个,间接朋友有 240 个
用户 1,247 的直接朋友有 24 个,间接朋友有 1,021 个
用户 2,458 的直接朋友有 99 个,间接朋友有 694 个

2. 超级连接者查询

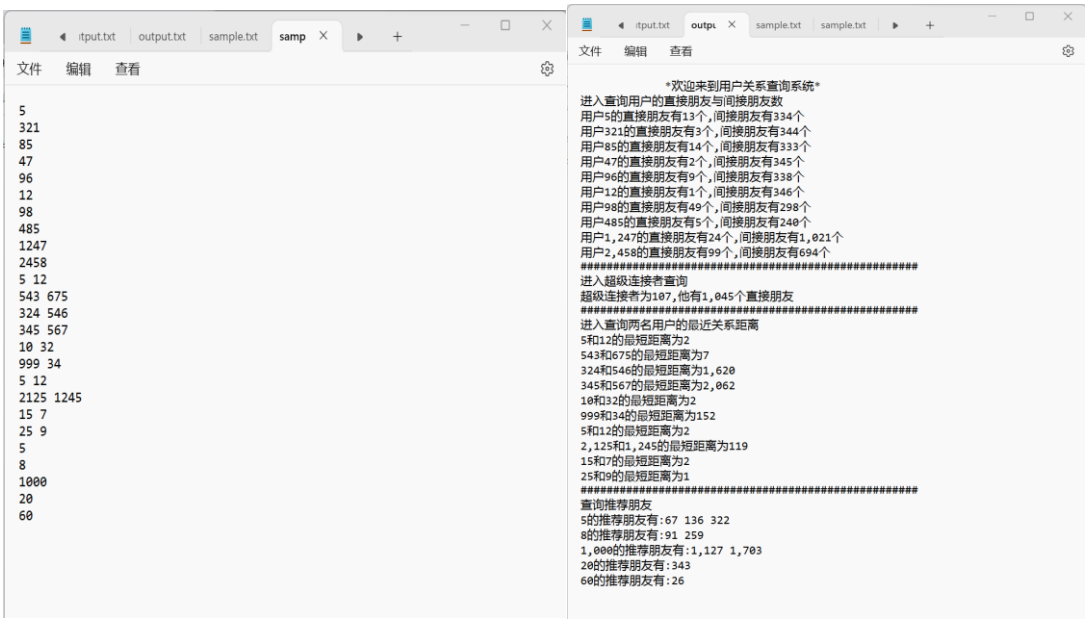
超级连接者为 107,他有 1,045 个直接朋友

3. 查询两名用户的最近关系距离

5 和 12 的最短距离为 2
543 和 675 的最短距离为 7
324 和 546 的最短距离为 1,620
345 和 567 的最短距离为 2,062
10 和 32 的最短距离为 2
999 和 34 的最短距离为 152
5 和 12 的最短距离为 2
2,125 和 1,245 的最短距离为 119
15 和 7 的最短距离为 2
25 和 9 的最短距离为 1

4. 查询推荐朋友

5 的推荐朋友有:67 136 322
8 的推荐朋友有:91 259
1,000 的推荐朋友有:1,127 1,703
20 的推荐朋友有:343
60 的推荐朋友有:26



具体实现如图

四． 反思与不足

本次 project 虽然功能都已全部实现，但是有很多不足，其中最大的不足在于对于性能的考虑不够充分，很多操作需要很大的时间复杂度，同时会产生很多冗余的空间，如何提高代码的效率是我未来应该着重努力的地方。同时，如何合理的运用数据结构，建立适合问题的数据结构也是我所需要考虑的，这些都需要在日后加强。另一方面，对代码的健壮性认识不足，很多出现问题之后才能意识到，对于野指针等问题的考虑还有待加强。

五． 收获

本次 project 一方面让我感受到了数据结构在实际中的广泛运用，意识到我们生活中的各种程序都离不开数据结构，另一方面在实现过程中对图的各种操作以及对图的结构有了深刻的认识，对深度遍历等有了更深入的理解。同时，在处理众多数据时，调试也成为我的一大问题，在不断解决 bug 的同时，也锻炼了我的代码调试能力，学会了使用输出、断点等方式来验证自己的猜想并解决 bug。

注：关于图的结构代码参考书中代码。