中山大学计算机学院本科生实验报告

(2025 学年第 1 学期)

课程名称：数据结构与算法实验　　　　　　　　　　　　任课老师：张子臻

年级	2024 级	专业（方向）	计算机科学与技术 (人工智能与大数据)
学号	24325155	姓名	梁桂铭
电话	15817681625	Email	lianggm8@mail2.sysu.edu.cn
开始日期	2025 年 11 月 24 日	完成日期	2025 年 11 月 24 日

实验题目

给出一棵有 n 个点的以 1 为根节点的有根多叉树，请把它转成左儿子右兄弟的二叉树形式，并输出其层次遍历顺序。

注意，每个节点的左儿子一定使用它的所有儿子中的编号最小的，右兄弟一定使用比它编号大的兄弟中最小的兄弟的编号。层次遍历需严格按照自上而下，自左而右访问树的节点。

实验目的

1. 掌握有根多叉树与左儿子右兄弟二叉树之间的转换方法
2. 理解左儿子右兄弟表示法的存储结构特点
3. 熟练掌握二叉树的层次遍历算法
4. 提高对树形结构的理解和编程实现能力

算法设计

1. 数据结构设计

- 使用邻接表 `vector<vector<int>> child` 存储多叉树结构
- 使用数组 `leftchild` 存储每个节点的左儿子
- 使用数组 `rightsiblings` 存储每个节点的右兄弟

2. 算法思路

1. **输入处理**：读取节点数 n，然后读取每个节点的父节点信息，构建多叉树的邻接表表示
2. **转换过程**：对于每个节点，将其所有子节点按编号从小到大排序，最小编号的子节点作为左儿子，其余子节点按顺序建立右兄弟关系
3. **层次遍历**：使用队列进行二叉树的层次遍历，先访问左儿子，再访问右兄弟

3. 算法流程

1. 初始化邻接表 `child`，大小为 n+1
2. 读取每个节点的父节点信息，构建多叉树结构
3. 遍历每个节点，将其子节点按编号排序，建立左儿子右兄弟关系
4. 使用队列从根节点开始进行层次遍历
5. 输出层次遍历结果

4. 复杂度分析

时间复杂度分析

- **输入阶段**：读取 n 个节点的父节点信息，时间复杂度为 $O(n)$
- **转换阶段**：遍历每个节点，处理其子节点列表。由于每个节点只会被处理一次，且所有子节点的处理总和为 $O(n)$，时间复杂度为 $O(n)$
- **遍历阶段**：层次遍历二叉树，每个节点入队出队一次，时间复杂度为 $O(n)$
- **总时间复杂度**：$O(n)$

空间复杂度分析

- **邻接表存储**：使用 $O(n)$ 空间存储多叉树结构
- **指针数组**：`leftchild` 和 `rightsiblings` 数组各需要 $O(n)$ 空间
- **队列空间**：层次遍历时队列最多存储 $O(n)$ 个节点
- **总空间复杂度**：$O(n)$

程序运行与测试

程序代码

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;
```

```cpp

int n;

int main () {
    int tmp = 0;
    cin >> n;
    vector<vector<int>> child(n + 1);
    for (int i = 2; i <= n; i++) {
        cin >> tmp;
        child[tmp].push_back(i);
    }
    int* leftchild = new int[n + 1];
    int* rightsiblings = new int[n + 1];

    for (int i = 1; i < n + 1; i++) {
        if (!child[i].empty()) {
            leftchild[i] = child[i][0];
            int l = child[i].size();
            for (int j = 0; j < l - 1; j++) {
                rightsiblings[child[i][j]] = child[i][j + 1];
            }
        }
    }

    queue<int> q;
    q.push(1);

    while (!q.empty()) {
        int tmp = q.front();
        cout << tmp << " ";
        q.pop();
        if (leftchild[tmp]) q.push(leftchild[tmp]);
        if (rightsiblings[tmp]) q.push(rightsiblings[tmp]);
    }
    return 0;
}
```

实验总结

通过本次实验，我深入理解了有根多叉树与左儿子右兄弟二叉树之间的转换关系。主要收获如下：

1. 掌握了左儿子右兄弟表示法的核心思想：左指针指向第一个子节点，右指针指向下一个兄弟节点
2. 理解了多叉树到二叉树的转换规则，能够正确处理子节点间的兄弟关系
3. 熟练掌握了二叉树的层次遍历算法，理解了队列在层次遍历中的应用
4. 提高了对树形结构的处理能力，能够灵活运用不同的数据结构表示树

实验题目

某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。规定：x 和 y 是亲戚，y 和 z 是亲戚，那么 x 和 z 也是亲戚。如果 x,y 是亲戚，那么 x 的亲戚都是 y 的亲戚，y 的亲戚也都是 x 的亲戚。（人数 5000，询问亲戚关系次数 5000）。

实验目的

1. 掌握并查集 (Union-Find) 数据结构的原理和实现方法
2. 学习使用并查集解决连通性判断问题
3. 理解路径压缩优化在并查集中的重要作用
4. 提高处理大规模数据关系的能力

算法设计

1. 数据结构设计

本问题采用并查集 (Disjoint Set Union, DSU) 数据结构来解决亲戚关系判断问题：

- 使用数组 dis[] 来表示并查集，数组下标对应人员编号
- 数组元素值为-1 表示该节点是根节点，否则存储其父节点的编号

2. 核心算法

查找操作 (find)：

- 功能：查找元素所在集合的代表元 (根节点)
- 实现：采用路径压缩优化，在查找过程中将路径上的所有节点直接连接到根节点
- 时间复杂度：近似 O(1)

合并操作 (unite)：

- 功能：合并两个元素所在的集合
- 实现：先找到两个元素的根节点，如果不同则将其中一个根节点连接到另一个根节点
- 时间复杂度：主要取决于查找操作，近似 O(1)

3. 算法流程

1. 初始化并查集数组，所有元素设为-1(各自为独立的集合)
2. 读入 m 对亲戚关系，对每对关系执行合并操作
3. 读入 q 次查询，对每次查询通过查找操作判断两个元素是否属于同一集合
4. 根据查找结果输出"Yes" 或"No"

4. 复杂度分析

0.0.1 时间复杂度分析

- **初始化阶段**：初始化并查集数组需要 $O(n)$ 时间
- **合并操作**：m 次合并操作，每次合并的时间复杂度为 $O(\alpha(n))$，其中 $\alpha(n)$ 是反阿克曼函数，增长极其缓慢，可视为常数级别
- **查询操作**：q 次查询操作，每次查询的时间复杂度同样为 $O(\alpha(n))$
- **总体复杂度**：$O(T \times (n + m \times \alpha(n) + q \times \alpha(n))) \approx O(T \times (n + m + q))$

0.0.2 空间复杂度分析

- **主要空间**：并查集数组需要 $O(n)$ 的空间
- **辅助空间**：递归调用栈的深度最多为 $O(n)$，但由于路径压缩优化，实际深度很小
- **总体复杂度**：$O(n)$

程序运行与测试

程序代码

```cpp
#include <iostream>
using namespace std;

int find (int* dis, int x) {
    if (dis[x] == -1) {
        return x;
    }
    dis[x] = find(dis, dis[x]);
    return dis[x];
}

void unite(int* dis, int x, int y) {
    x = find(dis, x);
    y = find(dis, y);
    if (x == y) return;
    dis[y] = x;
}
```

```cpp
int main () {
    int T;
    cin >> T;
    for (int i = 0; i < T; i++) {
        int n, m, q;
        cin >> n >> m;
        int* dis = new int[n + 1];
        for (int j = 0; j < n + 1; j++) {
            dis[j] = -1;
        }
        for (int j = 0; j < m; j++) {
            int x, y;
            cin >> x >> y;
            unite(dis, x, y);
        }
        cin >> q;
        for (int k = 0; k < q; k++) {
            int u, v;
            cin >> u >> v;
            if (find(dis, u) == find(dis, v)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
    }
}
```

实验总结

通过本次实验，我深入理解了并查集数据结构的原理和应用：

1. 并查集在处理动态连通性问题时具有很高的效率，查找和合并操作的时间复杂度都接近常数级别
2. 路径压缩技术显著提高了查找效率，避免了树形结构退化为链表的情况
3. 算法只需要 O(n) 的额外空间，非常适合处理大规模数据

实验题目

关系 R 具有对称性和传递性。数对 p q 表示 pRq,p 和 q 是 0 或自然数,p 不等于 q。要求写一个程序将数对序列进行过滤，如果一个数对可以通过前面数对的传递性得到，则将其滤去。

实验目的

1. 掌握并查集 (Union-Find) 在关系过滤中的应用
2. 理解对称性和传递性关系的特点

3. 学习处理大规模数对序列的高效算法
4. 提高对连通性问题的分析和解决能力

算法设计

1. 问题分析

给定一个具有对称性和传递性的关系 R，需要过滤数对序列：

- 如果数对 (p,q) 可以通过前面数对的传递性得到，则过滤掉
- 只输出那些建立新连通关系的数对
- 关系具有对称性：pRq 意味着 qRp
- 关系具有传递性：pRq 且 qRr 意味着 pRr

2. 数据结构设计

采用并查集数据结构来维护连通分量：

- 使用全局数组 dis[100001] 表示并查集
- 数组初始化为-1，表示每个元素都是独立的集合
- 数组下标对应数对中的数字 (0 到 100000)

3. 核心算法

查找操作 (find)：

- 功能：查找元素所在连通分量的根节点
- 实现：采用路径压缩优化，递归查找并更新父节点指针
- 时间复杂度：$O(\alpha(n))$，近似常数级别

合并操作 (unite)：

- 功能：合并两个连通分量
- 实现：先找到两个元素的根节点，如果不同则进行合并
- 时间复杂度：$O(\alpha(n))$，主要取决于查找操作

4. 复杂度分析

0.0.3 时间复杂度分析

- **初始化阶段**：初始化并查集数组需要 O(n) 时间，n=100000
- **处理每个数对**：对于每个输入数对，执行一次查找和可能的合并操作
- **单次操作复杂度**：查找和合并操作的时间复杂度为 $O(\alpha(n))$，其中 $\alpha(n)$ 是反阿克曼函数
- **总体复杂度**：$O(m \times \alpha(n))$，其中 m 为输入数对数量 (最多 1000000)

0.0.4 空间复杂度分析

- **主要空间**：并查集数组需要 $O(n) = O(100000)$ 的空间
- **辅助空间**：递归调用栈深度很小，由于路径压缩优化
- **总体复杂度**：$O(n)$

5. 算法流程

1. 初始化并查集数组，所有元素设为-1
2. 循环读取每个数对 (x,y)
3. 使用 find 操作检查 x 和 y 是否已经在同一连通分量中
4. 如果不在同一分量，输出该数对并执行 unite 操作合并两个分量
5. 如果在同一分量，说明该关系可通过传递性得到，直接跳过

程序运行与测试

程序代码

```cpp
#include <iostream>
using namespace std;
int dis[100001] = {-1};
int find (int x) {
    if (dis[x] == -1) return x;
    dis[x] = find(dis[x]);
    return dis[x];
}
void unite(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    dis[y] = x;
}
int main () {
    for (int i = 0; i < 100001; i++) {
        dis[i] = -1;
    }
    int x, y;
    while (cin >> x >> y) {
        if (find(x) != find(y)) cout << x << " " << y << endl;
        unite(x, y);
    }
    return 0;
}
```

实验总结

通过本次实验，我深入理解了并查集在关系过滤中的应用：

1. 并查集能够高效处理大规模的关系过滤问题，时间复杂度接近线性
2. 将关系过滤问题转化为连通分量维护问题，体现了问题抽象能力的重要性
3. 使用固定大小的数组，空间复杂度为 O(n)，适合处理大规模数据

附录、提交文件清单

第一题

```cpp
#include <iostream>
#include <queue>
#include <vector>

using namespace std;

int n;

int main () {
    int tmp = 0;
    cin >> n;
    vector<vector<int>> child(n + 1);
    for (int i = 2; i <= n; i++) {
        cin >> tmp;
        child[tmp].push_back(i);
    }
    int* leftchild = new int[n + 1];
    int* rightsiblings = new int[n + 1];

    for (int i = 1; i < n + 1; i++) {
        if (!child[i].empty()) {
            leftchild[i] = child[i][0];
            int l = child[i].size();
            for (int j = 0; j < l - 1; j++) {
                rightsiblings[child[i][j]] = child[i][j + 1];
            }
        }
    }

    queue<int> q;
    q.push(1);

    while (!q.empty()) {
        int tmp = q.front();
```

```cpp
        cout << tmp << "␣";
        q.pop();
        if (leftchild[tmp]) q.push(leftchild[tmp]);    // 注意vector的扩容机制
        if (rightsiblings[tmp]) q.push(rightsiblings[tmp]);
    }
    return 0;
}
```

第二题

```cpp
#include <iostream>
using namespace std;

int find (int* dis, int x) {
    if (dis[x] == -1) {
        return x;
    }
    dis[x] = find(dis, dis[x]);
    return dis[x];
}

void unite(int* dis, int x, int y) {
    x = find(dis, x);
    y = find(dis, y);
    if (x == y) return;
    dis[y] = x;
}
int main () {
    int T;
    cin >> T;
    for (int i = 0; i < T; i++) {
        int n, m, q;
        cin >> n >> m;
        int* dis = new int[n + 1];    // 不能用new int[n + 1]{1}初始化
        for (int j = 0; j < n + 1; j++) {
            dis[j] = -1;
        }
        for (int j = 0; j < m; j++) {
            int x, y;
            cin >> x >> y;
            unite(dis, x, y);
        }
        cin >> q;
        for (int k = 0; k < q; k++) {
            int u, v;
            cin >> u >> v;
            if (find(dis, u) == find(dis, v)) cout << "Yes" << endl;
            else cout << "No" << endl;
        }
```

```
    }
}
```

第三题

```
#include <iostream>
using namespace std;
int dis[100001] = {-1};
int find (int x) {
    if (dis[x] == -1) return x;
    dis[x] = find(dis[x]);
    return dis[x];
}
void unite(int x, int y) {
    x = find(x);
    y = find(y);
    if (x == y) return;
    dis[y] = x;
}
int main () {
    for (int i = 0; i < 100001; i++) {
        dis[i] = -1;
    }
    int x, y;
    while (cin >> x >> y) {
        if (find(x) != find(y)) cout << x << " " << y << endl;
        unite(x, y);
    }
    return 0;
}
```