**作业 HW4\* 实验报告**

日期：2024年12月1日

1. **涉及数据结构和相关背景**

**1.1定义**

* 图G=(V,E)由顶点集V和边集E组成
* 顶点(Vertex)：图中的数据元素
* 边(Edge)：顶点之间的关系
* 有向图：边有方向
* 无向图：边无方向
* 带权图：边具有权值

### 1.2基本术语

* 度(Degree)：与顶点相连的边数
* 路径(Path)：顶点间的边序列
* 连通图(ConnectedGraph)：任意两点间存在路径
* 生成树(SpanningTree)：包含所有顶点的无环连通子图

## 1.3.图的存储结构

### 邻接矩阵

### 邻接表

## 1.4.图的遍历

### 广度优先搜索(BFS)

### 深度优先搜索(DFS)

## 1.5.最小生成树算法

### Kruskal算法（适用于稀疏图）

### Prim算法（适用于稠密图）

## 1.6.拓扑排序

### AOV网络

### AOE网络

## 1.7.最短路算法

### Dijkstra算法（非负权图）

### Floyd算法（全源最短路）

**2. 实验内容**

**2.1 图的遍历**

**2.1.1 问题描述**

本题给定一个无向图，需要使用DFS和BFS两种方法找出图中的所有连通分量。图的顶点编号从0到n-1，要求搜索时总是从编号最小的顶点开始遍历。可以使用邻接矩阵或邻接表(尾插法)存储图的结构。

**2.1.2 基本要求**

输入:

* 第一行输入两个整数n和m,分别表示顶点数和边数
* 接下来m行,每行输入一条边的两个顶点编号

输出:

* 第一行输出DFS遍历结果
* 第二行输出BFS遍历结果
* 连通分量以{v1 v2...}的格式输出,分量内顶点用空格分隔,分量之间无空格

数据范围:

* 20%数据: 0<n≤15
* 40%数据: 0<n≤100
* 100%数据: 0<n≤1000
* 所有数据: 0.5n≤m≤1.5n

**2.1.3 数据结构设计**

struct grmap

{

    bool exit = 0;      // 表示两点间是否存在边

    int visit\_time = 0; // 遍历次数标记

};

// 核心数据结构定义

grmap \*\*gra; // 邻接矩阵

int \*num\_;   // 每个顶点的度数

int \*min\_;   // 每个顶点相邻的最小顶点编号

bool \*out;   // 访问标记数组

**2.1.4功能说明（函数、类）**

**1. 图的建立**

void input(grmap\*\* gra, int\* num\_, int\* min\_, const int edge\_num) {

    for (edge\_num次) {

        读入边的两个顶点start, end

        // 无向图需要双向标记

        gra[start][end].exit = true

        gra[end][start].exit = true

        // 更新顶点的度数

        num\_[start]++

        num\_[end]++

        // 更新最小相邻顶点编号

        min\_[start] = min(min\_[start], end)

        min\_[end] = min(min\_[end], start)

    }

}

**2. DFS遍历实现**

void dfs\_help(int now\_v, grmap\*\* gra, const int\* num\_, const int\* min\_) {

    if (顶点已访问) return

    // 处理当前顶点

    if (度数 > 1)

        将顶点压入栈

    if (度数 == 0)

        从栈顶取出新的顶点继续遍历

    标记当前顶点已访问

    // 遍历相邻顶点

    for (从最小相邻点遍历到最大点) {

        if (顶点未访问 且 存在边连接) {

            标记边已访问

            输出相邻顶点

            递归遍历相邻顶点

        }

    }

    if (栈不为空)

        继续遍历栈顶顶点

}

**2.1.5 调试分析（遇到的问题和解决方法）**

1. **连通分量的划分**

* 每次从未访问的最小编号顶点开始遍历
* 遍历完一个连通分量后，寻找下一个未访问的最小编号顶点
* 使用visit\_time区分DFS的访问标记

1. **遍历顺序控制**

* DFS使用栈结构确保回溯时能选择正确的顶点
* 每次选择编号最小的相邻顶点优先遍历

1. **输出格式处理**

* 连通分量边界的特殊处理
* 空格的添加时机控制
* 标记符号的位置控制

**2.1.6 总结和体会**

1. 算法设计思路：

* 分别实现DFS和BFS两种遍历方式
* 通过连通分量的概念理解图的结构特性
* 理解不同遍历方式的特点和应用场景

1. 实现要点：

* 合理设计数据结构，降低空间复杂度
* 正确处理图的边界情况
* 注意代码的可维护性和可读性

1. 改进方向：

* 可以考虑使用邻接表存储优化空间复杂度
* 遍历算法可以通过迭代方式代替递归
* 输出格式处理可以封装成独立函数

**2.2 六度空间**

**2.2.1 问题描述**

六度空间理论(小世界理论)认为世界上任意两个人之间间隔的人不会超过6个人。本题要求对给定的社交网络图,计算每个节点在"六度空间"理论(即经过不超过6个人)可以认识的人数占总人数的百分比。

**2.2.2 基本要求**

输入:

* 第一行输入N和M,分别表示节点数和边数(1<N≤2000, M≤33×N)
* 接下来M行,每行输入两个整数,表示有直接联系的两个节点编号(1到N)

输出:

* 对每个节点输出一行,格式为"节点编号: 百分比%"
* 百分比精确到小数点后2位
* 按float类型计算以避免精度误差

**2.2.3 数据结构设计**

// 使用邻接表存储图结构

vector<vector<int>> input\_gramap; // 存储图的邻接表

bool \*visit;                      // 访问标记数组

queue<int> que;                   // 用于BFS的队列

**2.2.4功能说明（函数、类）**

**1.BFS层次遍历函数:**

static int bfs(const vector<vector<int>>& input\_gramap,

              const int num\_point, const int start\_point) {

    // 初始化访问数组和队列

    bool\* visit = new bool[num\_point] {false};

    queue<int> que;

    int num\_return = 0;  // 记录可达节点数

    // 从起始点开始BFS

    que.push(start\_point);

    // 限制6层遍历

    for(int layer = 0; layer < 6 && !que.empty(); layer++) {

        int size = que.size();  // 当前层节点数

        // 遍历当前层所有节点

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

            int curr = que.front();

            que.pop();

            // 遍历相邻节点

            for(int next : input\_gramap[curr]) {

                if(!visit[next]) {

                    visit[next] = true;

                    que.push(next);

                    num\_return++;

                }

            }

        }

    }

    delete[] visit;

    return num\_return;  // 返回可达节点数

}

**2. 主函数**

int main() {

    // 读入节点数和边数

    输入n,m

    // 建立邻接表

    创建大小为n+1的邻接表

    for(m次) {

        读入边的两端点x,y

        在邻接表中添加双向边

    }

    // 计算每个节点的六度空间百分比

    for(每个节点) {

        调用bfs得到可达节点数

        计算并按格式输出百分比

    }

}

**2.2.5 调试分析（遇到的问题和解决方法）**

1. BFS深度控制

* 问题: 需要限制BFS的遍历深度为6层
* 解决: 使用两层循环,外层控制层数,内层处理当前层节点

1. 百分比计算

* 问题: 需要保证计算精度
* 解决: 使用float类型计算,并使用setprecision控制输出精度

1. 内存管理

* 问题: 动态分配的访问数组需要及时释放
* 解决: 使用智能指针或确保delete及时执行

**2.2.6 总结和体会**

1. 算法设计要点:

* 使用BFS实现层次遍历
* 通过队列size控制层次
* 注意图的无向性质

1. 实现收获:

* 加深了对BFS的理解
* 掌握了层次遍历的控制方法
* 提高了对精度控制的认识

1. 可优化方向:

* 可以使用邻接矩阵优化空间
* 可以使用并查集优化连通性判断
* 可以添加剪枝优化遍历效率

**2.3 村村通**

**2.3.1 问题描述**

在N个村庄之间建设道路,使得任意两个村庄之间都能互相到达。已知部分村庄之间已经建有道路,需要在剩余村庄之间新建一些道路,使得所有村庄连通,并且新建道路的总长度最小。这本质上是一个最小生成树的变种问题。

**2.3.2 基本要求**

输入：

* 第一行输入整数n(3≤n≤100),表示村庄数量
* 接下来n行,每行n个数,表示村庄i和j之间的距离(范围[1,1000])
* 然后输入整数m,表示已建道路数量
* 接下来m行,每行两个数a,b(1≤a<b),表示村庄a和b之间已建有道路

输出：

* 输出一个整数,表示需要新建道路的最小总长度

**2.3.3 数据结构设计**

// 存储村庄间距离的邻接矩阵

vector<vector<int>> all;

// 存储已建道路的连通性矩阵

vector<vector<bool>> connected;

// 存储每个村庄所属的连通分量

vector<int> father\_;

**2.3.4功能说明（函数、类）**

**1. 深度优先搜索标记连通分量:**

void bfs(int row, int the\_id, vector<vector<bool>>& connected,

         vector<int>& father\_, const int num\_country) {

    // 标记当前村庄属于当前连通分量

    father\_[row] = the\_id;

    // 遍历所有可能相连的村庄

    for (int i = 0; i < num\_country; i++) {

        // 如果存在已建道路且未被标记

        if (connected[row][i] && father\_[i] == -1) {

            // 递归标记

            bfs(i, the\_id, connected, father\_, num\_country);

        }

    }

}

**2. 主算法实现:**

int main() {

    // 读入基础数据并初始化数据结构

    输入村庄数量n

    初始化距离矩阵all

    初始化连通矩阵connected

    // 标记现有连通分量

    for (遍历所有村庄) {

        if (村庄未被标记) {

            从该村庄开始进行深度优先搜索标记

        }

    }

    // 寻找最小代价连接方案

    while (true) {

        初始化最小距离和最佳连接点

        // 遍历所有可能的连接

        for (所有村庄对(i,j)) {

            if (i和j属于不同连通分量) {

                更新最小距离和最佳连接点

            }

        }

        if (找到最佳连接点) {

            更新总代价

            合并连通分量

            检查是否已全部连通

        }

    }

}

**2.3.5 调试分析（遇到的问题和解决方法）**

1. 连通分量识别

* 问题：需要正确识别已有道路形成的连通分量
* 解决：使用深度优先搜索标记连通分量

1. 最小代价选择

* 问题：在不同连通分量间找到最小代价的连接方式
* 解决：遍历所有可能的连接,维护最小值

1. 连通性判断

* 问题：判断是否所有村庄都已连通
* 解决：检查所有村庄是否属于同一连通分量

**2.3.6 总结和体会**

1. 算法设计思路：

* 将问题转化为最小生成树的变种
* 利用已有道路对原图进行预处理
* 通过贪心策略选择最小代价连接

1. 关键技术点：

* 深度优先搜索标记连通分量
* 贪心选择最小代价连接
* 并查集思想管理连通性

1. 优化方向：

* 可以使用并查集优化连通性管理
* 可以使用最小堆优化边的选择
* 可以使用邻接表优化空间复杂度

**2.4 给定条件下构造矩阵**

**2.4.1 问题描述**

需要构造一个k×k的矩阵，使得1到k的每个数字恰好出现一次，其余位置填0。矩阵需要满足两组条件：行条件(rowConditions)规定某些数字的上下相对位置关系，列条件(colConditions)规定某些数字的左右相对位置关系。

**2.4.2 基本要求**

输入：

* 第一行包含三个整数k、n、m，分别表示矩阵大小、行条件数和列条件数
* 接下来n行，每行两个整数表示行的先后关系
* 接下来m行，每行两个整数表示列的左右关系

输出：

* 如果存在满足条件的矩阵，输出该矩阵
* 如果不存在解，输出-1
* 矩阵元素间用空格分隔

**2.4.3 数据结构设计**

// 存储最终构造的矩阵

vector<vector<int>> res;

// 存储每个数字的行号和列号

vector<int> up\_down\_id;    // 行号

vector<int> left\_right\_id; // 列号

// 标记行列是否被使用

vector<bool> row; // 行使用标记

vector<bool> col; // 列使用标记

// 存储图的邻接表

vector<vector<int>> father; // 存储前驱节点

vector<vector<int>> ch;     // 存储后继节点

**2.4.4功能说明（函数、类）**

**1.拓扑排序处理行/列约束：**

void input(const int in\_num, vector<int> &no, const int n) {

    // 初始化数据结构

    vector<int> child(n, 0);                    // 入度统计

    vector<vector<int>> father(n), ch(n);       // 邻接表

    vector<vector<bool>> appear(n, vector<bool>(n, false)); // 重边检查

    // 构建有向图

    for (每组约束条件) {

        if (!已存在该边) {

            添加边到邻接表

            更新入度统计

            标记该边已存在

        }

    }

    // 寻找入度为0的起始点

    找到入度为0的点start

    if (不存在入度为0的点) return;

    // 拓扑排序过程

    no[start] = n - 1;  // 安排最高位置

    for (遍历所有点) {

        处理当前点的所有前驱节点

        更新前驱节点的位置编号

        更新入度信息

        选择新的起始点

    }

}

**2.矩阵构造：**

int main() {

    // 读入数据

    读入k, n, m

    初始化结果矩阵和辅助数组

    // 分别处理行列约束

    input(up\_down, up\_down\_id, n);

    input(left\_right, left\_right\_id, n);

    // 构造矩阵

    for (每个数字i) {

        if (位置无效 || 位置已被占用) {

            输出-1

            退出程序

        }

        在确定的位置放置数字i+1

        标记该行列已使用

    }

    // 输出结果矩阵

    打印矩阵

}

**2.4.5 调试分析（遇到的问题和解决方法）**

1. 约束关系处理

* 问题：需要正确处理数字间的相对位置关系
* 解决：使用拓扑排序确定每个数字的行列位置

1. 重边处理

* 问题：输入可能包含重复的约束关系
* 解决：使用appear数组标记已存在的边

1. 矩阵有效性检验

* 问题：需要确保每个数字都有合法的位置且不冲突
* 解决：使用row和col数组检查位置占用情况

**2.4.6 总结和体会**

1. 算法设计思路：

* 将位置约束转化为有向图
* 使用拓扑排序确定相对位置
* 通过位置的合法性验证解的存在性

1. 实现难点：

* 正确处理重复约束条件
* 确保位置分配不冲突
* 验证解的合法性

1. 可优化方向：

* 可以使用优先队列优化拓扑排序
* 可以优化空间复杂度
* 可以添加更多的提前判断以加快求解速度

**2.1 必修课**

**2.5.1 问题描述**

计算机系的课程体系包含n门必修课程。每门课程需要特定学时完成,且可能依赖于其他课程的完成。学生可以在满足前置课程要求的情况下,随时选修任意数量的课程。系统需要预测每门课程的最早完成时间,并分析单个课程学时变动对总体毕业时间的影响。

**2.5.2 基本要求**

输入内容：

* 第一行输入课程总数n
* 接下来n行每行包含:
  + 课程学时ti
  + 前置课程数量ci
  + ci个前置课程编号

输出要求：

* 共n行,每行两个整数:
  + 该课程最早完成时间
  + 增加该课程学时是否延长毕业时间(1表示会,0表示不会)

**2.5.3 数据结构设计**

class Solution

{

private:

    int n;                     // 课程总数

    vector<vector<int>> graph; // 课程依赖图(邻接表表示)

    vector<int> times;         // 各课程学时

    // 辅助数组

    vector<int> inDegree; // 入度统计

    vector<int> earliest; // 最早完成时间

};

**2.5.4功能说明（函数、类）**

**1. 计算课程完成时间：**

vector<int> cal\_times(const vector<int>& times) {

    // 初始化数据

    vector<int> inDegree(n, 0);    // 统计每门课的前置课程数

    vector<int> earliest(n, 0);     // 记录最早完成时间

    queue<int> q;                   // 用于拓扑排序的队列

    // 统计每门课的前置课程数

    for (课程 i = 0 到 n-1) {

        for (课程 i 的后续课程 next) {

            inDegree[next]++;

        }

    }

    // 将所有无前置课程的课程加入队列

    for (课程 i = 0 到 n-1) {

        if (无前置课程) {

            加入队列;

            设置初始完成时间;

        }

    }

    // 拓扑排序计算完成时间

    while (队列非空) {

        取出当前课程;

        for (当前课程的后续课程 next) {

            更新完成时间;

            减少入度;

            if (所有前置课程已完成) {

                将课程加入队列;

            }

        }

    }

    return earliest;

}

**2. 主要计算流程：**

void solve() {

    // 构建课程依赖图

    读取课程数量和学时信息;

    构建课程依赖关系;

    // 计算原始完成时间

    vector<int> ori\_times = cal\_times(times);

    int ori\_max\_time = findMax(ori\_times);

    // 分析每门课程学时增加的影响

    for (每门课程) {

        增加该课程学时;

        计算新的完成时间;

        判断是否影响总体毕业时间;

        恢复课程学时;

    }

}

**2.5.5 调试分析（遇到的问题和解决方法）**

1. 课程依赖处理

* 问题：需要正确处理课程之间的前后依赖关系
* 解决：使用邻接表存储依赖关系,通过拓扑排序确保依赖顺序

1. 完成时间计算

* 问题：需要考虑所有前置课程的完成时间
* 解决：在拓扑排序过程中动态更新完成时间

1. 影响分析

* 问题：判断单个课程变化对总体的影响
* 解决：通过比较变化前后的最大完成时间

**2.5.6 总结和体会**

1. 算法设计思路：

* 使用拓扑排序处理课程依赖关系
* 动态规划思想计算最早完成时间
* 增量分析方法评估变化影响

1. 核心技术点：

* 图的邻接表表示
* 基于队列的拓扑排序实现
* 动态更新的时间计算方法

**2.6小马吃草**

**2.6.1 问题描述**

在一个带权无向图G中,有N个点和M条边,点的编号从1到N,第i条边的长度为w\_i。其中H个点上有可供食用的牧草。现有R匹小马,第j匹小马的起始位置在start\_j点,需要先到任意一个有牧草的点进食,然后前往终点end\_j。问题要求计算每匹小马需要走过的最短距离。**2.6.2 基本要求**

输入格式：

* 第一行：顶点数N和边数M
* 接下来M行：每行三个整数x\_i、y\_i、w\_i,表示一条边的两个端点和权重
* 下一行：有牧草的点数量H和小马数量R
* 接下来一行：H个整数表示有牧草的点的编号
* 最后R行：每行两个整数start\_j和end\_j表示每匹小马的起点和终点

输出格式：

* R行整数,表示每匹小马需要走过的最短距离

**2.6.3 数据结构设计**

// 边的结构定义

class Edge

{

public:

    int to;     // 目标顶点

    int weight; // 边的权重

    Edge(int t, int w) : to(t), weight(w) {}

};

// 解决方案类

class Solution

{

private:

    vector<vector<Edge>> graph; // 图的邻接表表示

    vector<int> grass\_points;   // 存储牧草点的位置

    int n;                      // 顶点数量

};

**2.6.4功能说明（函数、类）**

**1. Dijkstra最短路径算法实现：**

vector<long long> dijkstra(int start)

{

    // 初始化距离数组

    vector<long long> dist(n + 1, LLONG\_MAX);

    dist[start] = 0;

    queue<int> pq;

    // 从起点开始搜索

    pq.push(start);

    // 广度优先搜索过程

    while (!pq.empty())

    {

        // 取出当前顶点

        int current = pq.front();

        pq.pop();

        // 遍历所有相邻顶点

        for (const Edge &edge : graph[current])

        {

            // 如果找到更短路径则更新

            if (dist[edge.to] > dist[current] + edge.weight)

            {

                dist[edge.to] = dist[current] + edge.weight;

                pq.push(edge.to);

            }

        }

    }

    return dist;

}

**2.** **主要解决流程：**

void solve() {

    // 读取图结构

    读入顶点数N和边数M;

    构建邻接表;

    // 读取牧草点和马的信息

    读入牧草点数量H和马的数量R;

    // 处理每匹马的路径

    for (每匹马) {

        // 计算最短路径

        计算从起点到所有点的最短距离;

        计算从终点到所有点的最短距离;

        // 寻找经过牧草点的最短总路径

        遍历所有牧草点;

        更新最短总距离;

        输出最短距离;

    }

}

**2.6.5 调试分析（遇到的问题和解决方法）**

1. 路径计算优化

* 使用Dijkstra算法计算最短路径
* 分别计算起点到牧草点和牧草点到终点的距离
* 通过组合路径找到最优解

1. 内存管理

* 采用邻接表存储图结构以优化空间使用
* 使用long long类型处理大数值
* 合理复用距离数组以减少内存分配

1. 性能优化

* 实现基于队列的Dijkstra算法
* 避免重复路径计算
* 在找到最优路径后及时终止搜索

**2.6.6 总结和体会**

1. 算法选择 选择Dijkstra算法的原因：

* 能够高效处理带权图
* 保证找到最短路径
* 适合多点路径查找问题

1. 实现要点 代码实现需要平衡多个因素：

* 通过合适的数据结构实现内存效率
* 通过优化算法提高处理速度
* 保持代码结构清晰便于维护

1. 改进方向 可能的优化方案：

* 使用优先队列改进Dijkstra算法性能
* 对重复计算进行缓存

**3. 实验总结**

通过本次图论算法实验，我们深入研究了图的遍历、最小生成树、最短路径以及拓扑排序等经典问题，获得了丰富的实践经验和深刻的理论认识。现将主要收获总结如下：

在实验中，不同的数据结构展现出各自的优势和局限性。邻接矩阵在图的遍历问题中表现出查询效率高的特点，但在村村通问题等稀疏图场景下会造成较大的空间浪费。邻接表则在小马吃草这类需要频繁访问邻接点的问题中体现出更好的性能。深度优先搜索中使用栈结构，而广度优先搜索采用队列，这些辅助数据结构的选择都直接影响了算法的实现效率。

问题抽象和建模是算法设计的首要环节。在课程依赖分析中，我们将课程关系抽象为有向无环图，通过拓扑排序有效解决了时间计算问题。六度空间理论的验证则通过限制广度优先搜索的层数来实现，体现了对实际问题的巧妙简化。在算法效率方面，村村通问题中采用并查集优化连通性判断，小马吃草问题中使用优先队列优化Dijkstra算法，都取得了显著的性能提升。

在具体实现过程中，积累了大量实用技巧。例如在图的遍历中使用访问标记数组避免重复访问，在最短路径算法中采用动态规划思想优化计算过程。调试过程中注意到边界情况的处理尤为重要，如处理重边、自环等特殊情况。同时，通过模块化设计、清晰的代码结构和完善的注释文档，确保了程序的可维护性和可扩展性。

本次实验不仅加深了对图论算法的理解，更重要的是提高了将理论知识应用于解决实际问题的能力。通过综合运用多种数据结构和算法，成功处理了不同类型的图论问题，为今后的算法学习和工程实践奠定了坚实的基础。