|  |  |
| --- | --- |
| **教师签名** | 说明: C:\Users\Administrator\AppData\Local\Microsoft\Windows\INetCache\Content.Word\我的签字.png |
| **综合性实验成绩** |  |

重庆交通大学

信息科学与工程学院

综合性实验报告

实验名称 图的存储及相关算法实验

课程名称 算法与数据结构

专业班级 电子信息类2104班

学 号 632107030401

姓 名 林忠炟

指导教师 王铁建

2023 年 5 月

|  |  |
| --- | --- |
| **评分等级** | **综合性实验评分标准** |
| 优 | 程序演示完全正确，界面美观，能正确回答90%及以上的问题；报告规范，分析清楚，严格按照要求条目书写，阐述清楚。能针对综合性题目进行分析，并设计合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。 |
| 良 | 按要求完成80%及以上功能，界面尚可，能正确回答80%及以上的问题；报告规范，分析清楚，个别条目书写不完全符合要求，阐述基本清楚。能针对综合性题目进行分析，并设计较合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。 |
| 中 | 按要求完成70%及以上功能，能回答70%及以上的问题；报告基本规范，分析基本清楚，存在30%以内条目书写不完全符合要求。在教师的指导下，能针对综合性题目进行分析，并设计较合适的解决方案，同时使用合适的编程平台进行编程、调试、测试和实现。 |
| 及格 | 按要求完成60%及以上功能，能回答老师多数问题；报告基本规范，存在40%以内条目书写不完全符合要求。在教师的指导下，能针对综合性题目进行分析和设计较合适的解决方案，并能使用合适的编程平台进行编程、调试、测试和实现，成功调试功能达60%以上。 |
| 不及格 | 存在40%以上功能未完成或抄袭；报告不规范或存在40%以上条目书写不完全符合要求。无法采用正确的方法完成项目分析和解决方案设计或成功调试测试功能超过40%未完成。 |

# 第一章 实验目的

## 1.1 图论的基本概念和算法

图论是研究图和图的性质以及在图上的计算方法和应用的学科。图论中的图是由节点和边构成的，节点表示对象，边表示对象间的关系。在图中，节点也被称为顶点或节点，边也被称为边或弧。以下是图论的一些基本概念和算法：

- 无向图和有向图：图分为无向图和有向图，无向图中的边没有方向，有向图中的边有方向。

- 连通图和不连通图：如果一张图中任意两个节点都能通过路径相连，则称该图为连通图，否则称为不连通图。

- 图的表示：图可以用邻接矩阵、邻接表等方式来表示。

- 图的遍历：图的遍历包括深度优先遍历和广度优先遍历。

- 最小生成树算法：Prim算法和Kruskal算法是常用的求无向图最小生成树的算法。

- 最短路径算法：Dijkstra算法和Floyd算法是常用的求图中最短路径的算法。

只有掌握了以上常见的基本概念和算法，才能更好地理解和运用图论。

## 1.2 应用图论解决实际问题

图论在现实生活中有广泛的应用，包括社交网络、电路设计、交通规划等方面，是许多实际问题的抽象模型。因此，学习图论相关内容，可以帮助我们培养更强大的思维，使我们更有能力理解和解决现实世界中的问题。

# 第二章 实验环境及分工

## 2.1 编程语言和开发环境

本实验中的代码所使用的编程语言均为C++，实验在Windows10操作系统上进行。

## 2.2 实验所需软件及相关配置

实验所使用到的软件有Visual Studio 2019。

## 2.3 实验分工

林忠炟：图的创建及表示方法：邻接表法、邻接矩阵法；

图的遍历算法：深度优先遍历、宽度优先遍历。

屈耘毅：最小生成树算法：Kruskal算法、Prim算法。

鲁轩廷：最短路径算法：Dijkstra算法、Floyd算法。

# 第三章 实验内容

## 3.1 图的表示方法及其创建

## 3.1.1 邻接表法

在邻接表中，每个顶点都对应一个链表，链表中存储与该顶点相邻的所有顶点。如果该图是无向图，那么每条边都对应着两个顶点之间的连边，所以需要在两个链表中分别存储这两个顶点相邻的所有顶点；如果该图是有向图，则只需要在起点对应的链表中存储终点即可。邻接表可以用一个数组来表示，数组中的每个元素对应图中的一个顶点，每个元素都是一个链表的头结点，链表中存储着该顶点相邻的所有顶点。

代码如下：

// 邻接表法创建图

class AdjacencyList\_CreateGraph {

public:

// 定义边类

class Edge {

public:

int from;

int to;

int weight;

Edge(int from, int to, int weight) {

this->from = from;

this->to = to;

this->weight = weight;

}

};

// 定义图类

class Graph {

public:

int n; // 图的顶点数

vector<vector<Edge>>adjList; // 邻接表

Graph(int n) {

this->n = n;

this->adjList.resize(n);

}

// 添加有向边

void addDirectedEdge(int from, int to, int weight) {

this->adjList[from].push\_back(Edge(from, to, weight));

}

// 添加无向边

void addUnDirectedEdge(int node1, int node2, int weight) {

addDirectedEdge(node1, node2, weight);

addDirectedEdge(node2, node1, weight);

}

};

// for test

void test() {

// 创建如下这张图：

// 0——>1

// | |

// v v

// 2<——3——>4

int n = 5;

Graph graph(5);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

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

cout << "节点" << i << "的后继节点有：";

for (Edge& next : graph.adjList[i]) {

cout << next.to << "(权值" << next.weight << ") ";

}

cout << endl;

}

}

};

## 3.1.2 邻接矩阵法

邻接矩阵法是图的一种表示方法，用一个二维数组来表示图中各个节点之间的连通关系。

假设图G有n个节点，我们可以使用一个n×n的矩阵A来表示这个图，其中A[i][j]表示节点i和节点j之间的连通关系。如果节点i和节点j之间有一条边，则A[i][j]的值为1，否则为0。

在有向图中，如果边的方向是从节点i指向节点j的，则A[i][j]的值为1，否则为0。

邻接矩阵法还可以用来表示带权图，即在边上标记权值。这时A[i][j]的值就表示从节点i到节点j的边的权值。

代码如下：

// 邻接矩阵法创建图

class AdjacencyMatrix\_CreateGraph {

public:

class Graph {

public:

int n; // 图的大小

vector<vector<int>>g; // g[i][j]：i号节点与j号节点之间有一条路径，权值为g[i][j]

Graph(int n) {

this->n = n;

this->g.resize(n, vector<int>(n));

}

// 添加有向边

void addDirectedEdge(int from, int to, int weight) {

g[from][to] = weight;

}

// 添加无向边

void addUnDirectedEdge(int node1, int node2, int weight) {

g[node1][node2] = weight;

g[node2][node1] = weight;

}

};

// for test

void test() {

// 创建如下这张图：

// 0——>1

// | |

// v v

// 2<——3——>4

Graph graph(5);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

for (int i = 0; i < graph.n; i++) {

for (int j = 0; j < graph.n; j++) {

if (graph.g[i][j] != 0) {

cout << i << "->" << j << " 权重：" << graph.g[i][j] << endl;

}

}

}

}

};

## 3.2 图的深度优先遍历

图的深度优先遍历（Depth First Search，DFS）是指从图的某一个顶点出发，尽可能“深”的搜索图的算法。其具体实现是从该顶点开始，遍历与该顶点相连的所有顶点，然后递归地对每个未被访问过的相邻顶点继续进行深度优先遍历，直到遍历完整张图。

DFS可以用递归或者栈来实现。在递归实现中，我们对当前节点的相邻节点递归调用DFS函数，而在栈实现中，我们首先把起始节点压入栈中，然后重复以下步骤：从栈中弹出一个节点，并把该节点标记为已访问；遍历该节点的邻居节点，并把未被访问过的邻居节点压入栈中。

DFS的时间复杂度是O(|V|+|E|)，其中V是图的顶点集合，E是图的边集合。

①邻接表法实现深度优先遍历代码如下：

// 图的深度优先遍历（邻接表）

class AdjacencyList\_DFS {

public:

// 深度优先遍历

// Graph结构请看3.1.1

void dfs(Graph graph) {

int n = graph.n;

vector<bool>vis(n); // 标记某个节点是否访问过

process(graph, 0, vis);

}

void process(Graph graph, int i, vector<bool>& vis) {

vis[i] = true;

cout << i << " ";

for (Edge& next : graph.adjList[i]) {

if (!vis[next.to]) { // vis[next.to] == false，说明next节点没有访问过

process(graph, next.to, vis);

}

}

}

// for test

void test() {

// 创建如下这张图：

// 0——>1——>5

// | |

// v v

// 2<——3——>4

Graph graph(6);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(1, 5, 5);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

dfs(graph);

}

};

②邻接矩阵法实现深度优先遍历代码如下：

// 深度优先遍历（邻接矩阵法）

class AdjacencyMatrix\_DFS {

public:

class Graph {

public:

int n; // 图的大小

vector<vector<int>>g; // g[i][j]：i号节点与j号节点之间有一条路径，权值为g[i][j]

Graph(int n) {

this->n = n;

this->g.resize(n, vector<int>(n));

}

// 添加有向边

void addDirectedEdge(int from, int to, int weight) {

g[from][to] = weight;

}

// 添加无向边

void addUnDirectedEdge(int node1, int node2, int weight) {

g[node1][node2] = weight;

g[node2][node1] = weight;

}

};

// DFS

void dfs(Graph graph) {

int n = graph.n;

vector<bool>vis(n, false);

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

if (!vis[i]) {

process(graph, i, vis);

}

}

}

void process(Graph graph, int i, vector<bool>& vis) {

vis[i] = true;

cout << i << " ";

for (int j = 0; j < graph.n; j++) {

if (graph.g[i][j] != 0 && !vis[j]) {

process(graph, j, vis);

}

}

}

// for test

void test() {

// 创建如下这张图：

// 0——>1——>5

// | |

// v v

// 2<——3——>4

Graph graph(6);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(1, 5, 5);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

dfs(graph);

}

};

## 3.3 图的宽度优先遍历

图的宽度优先遍历（BFS，Breadth-First Search）是一种用于遍历或搜索图形或树数据结构的算法。在遍历过程中，从一个源节点开始，宽度优先遍历先遍历源节点的所有邻居节点，然后遍历邻居节点的所有邻居节点，以此类推，直到图中所有节点都被遍历到。

具体的思路如下：

1. 创建一个空队列 Q 和一个空的 visited 集合，用来记录已经遍历过的节点。

2. 将源节点加入队列 Q 中，并将源节点加入 visited 集合中。

3. 当 Q 队列不为空时，从队列头部取出一个节点，并对其所有未被访问过的邻居节点进行遍历。

4. 对于每个邻居节点，如果其未被访问过，将其加入 Q 队列和 visited 集合中，并标记为已经访问。

5. 重复执行步骤 3 和步骤 4，直到 Q 队列为空。

①邻接表法实现宽度优先遍历代码如下：

// 图的宽度优先遍历（邻接表）

class AdjacencyList\_BFS {

public:

// 定义边类

class Edge {

public:

int from;

int to;

int weight;

Edge(int from, int to, int weight) {

this->from = from;

this->to = to;

this->weight = weight;

}

};

// 定义图类

class Graph {

public:

int n; // 图的顶点数

vector<vector<Edge>>adjList; // 邻接表

Graph(int n) {

this->n = n;

this->adjList.resize(n);

}

// 添加有向边

void addDirectedEdge(int from, int to, int weight) {

this->adjList[from].push\_back(Edge(from, to, weight));

}

// 添加无向边

void addUnDirectedEdge(int node1, int node2, int weight) {

addDirectedEdge(node1, node2, weight);

addDirectedEdge(node2, node1, weight);

}

};

// 宽度优先遍历

void bfs(Graph graph) {

queue<int>queue;

vector<bool>vis(graph.n); // 标记某个节点是否访问过

queue.push(0);

vis[0] = true;

while (!queue.empty()) {

int cur = queue.front();

queue.pop();

cout << cur << " ";

for (Edge& next : graph.adjList[cur]) {

if (!vis[next.to]) {

queue.push(next.to); // vis[next.to] == false，说明next节点没有访问过

vis[next.to] = true;

}

}

}

}

// for test

void test() {

// 创建如下这张图：

// 0——>1——>5——>6

// | | |

// v v v

// 2<——3——>4——>7

Graph graph(8);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(1, 5, 5);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

graph.addDirectedEdge(4, 7, 6);

graph.addDirectedEdge(5, 6, 7);

graph.addDirectedEdge(6, 7, 8);

bfs(graph);

}

};

②邻接矩阵法实现宽度优先遍历代码如下：

// 宽度优先遍历（邻接矩阵法）

class AdjacencyMatrix\_BFS {

public:

class Graph {

public:

int n; // 图的大小

vector<vector<int>>g; // g[i][j]：i号节点与j号节点之间有一条路径，权值为g[i][j]

Graph(int n) {

this->n = n;

this->g.resize(n, vector<int>(n));

}

// 添加有向边

void addDirectedEdge(int from, int to, int weight) {

g[from][to] = weight;

}

// 添加无向边

void addUnDirectedEdge(int node1, int node2, int weight) {

g[node1][node2] = weight;

g[node2][node1] = weight;

}

};

// BFS

void bfs(Graph graph, int start) {

int n = graph.n;

vector<bool>vis(n, false);

queue<int>queue;

queue.push(start);

vis[start] = true;

while (!queue.empty()) {

int cur = queue.front();

queue.pop();

cout << cur << " ";

// 遍历与当前节点相邻的节点

for (int next = 0; next < n; next++) {

if (graph.g[cur][next] != 0 && !vis[next]) {

queue.push(next);

vis[next] = true;

}

}

}

}

// for test

void test() {

// 创建如下这张图：

// 0——>1——>5——>6

// | | |

// v v v

// 2<——3——>4——>7

Graph graph(8);

graph.addDirectedEdge(0, 1, 1);

graph.addDirectedEdge(0, 2, 2);

graph.addDirectedEdge(1, 3, 3);

graph.addDirectedEdge(1, 5, 5);

graph.addDirectedEdge(3, 2, 4);

graph.addDirectedEdge(3, 4, 5);

graph.addDirectedEdge(4, 7, 6);

graph.addDirectedEdge(5, 6, 7);

graph.addDirectedEdge(6, 7, 8);

bfs(graph, 0);

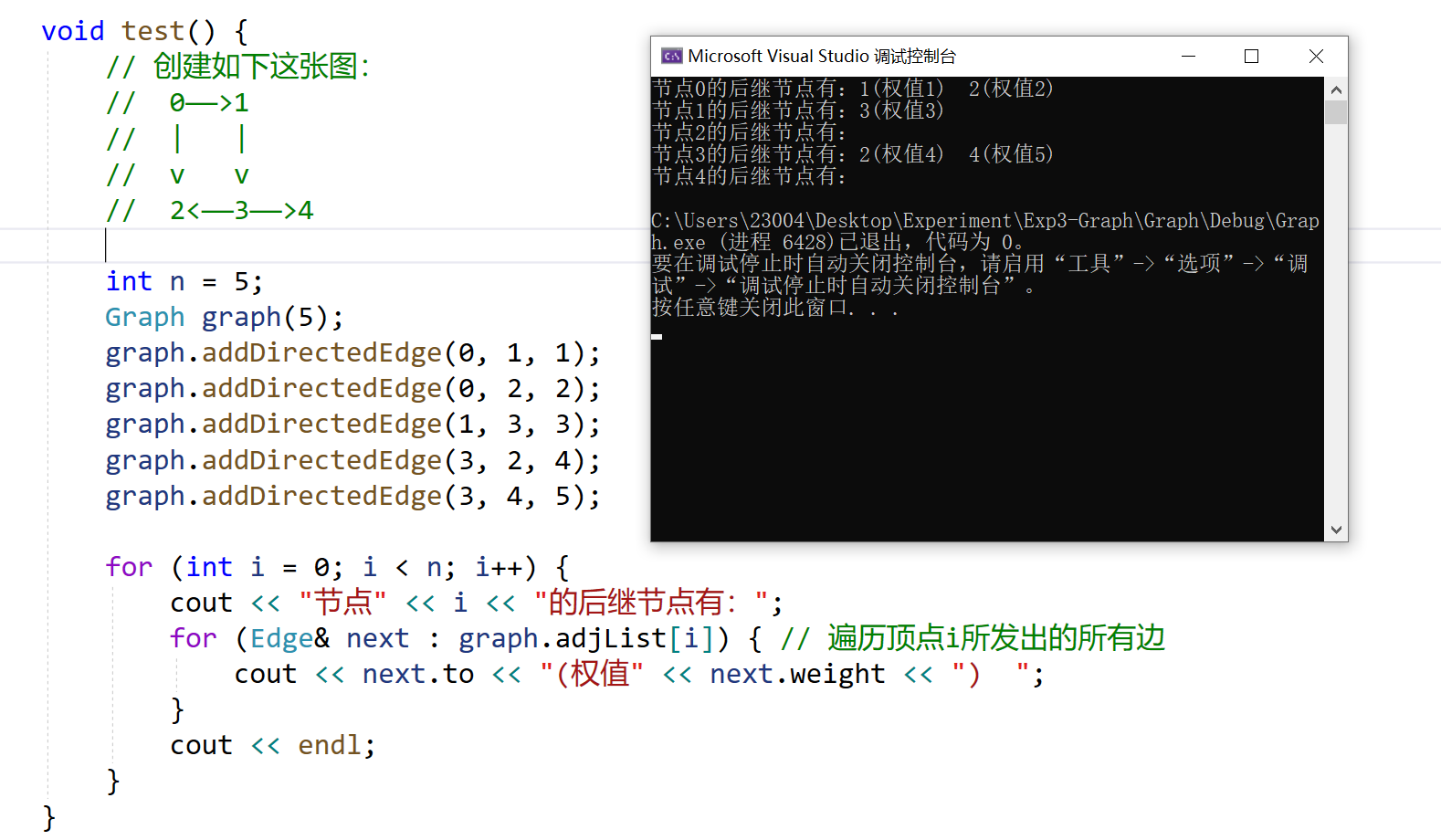
}

};

# 第四章 实验结果

## 4.1图的创建实验结果展示

邻接表：



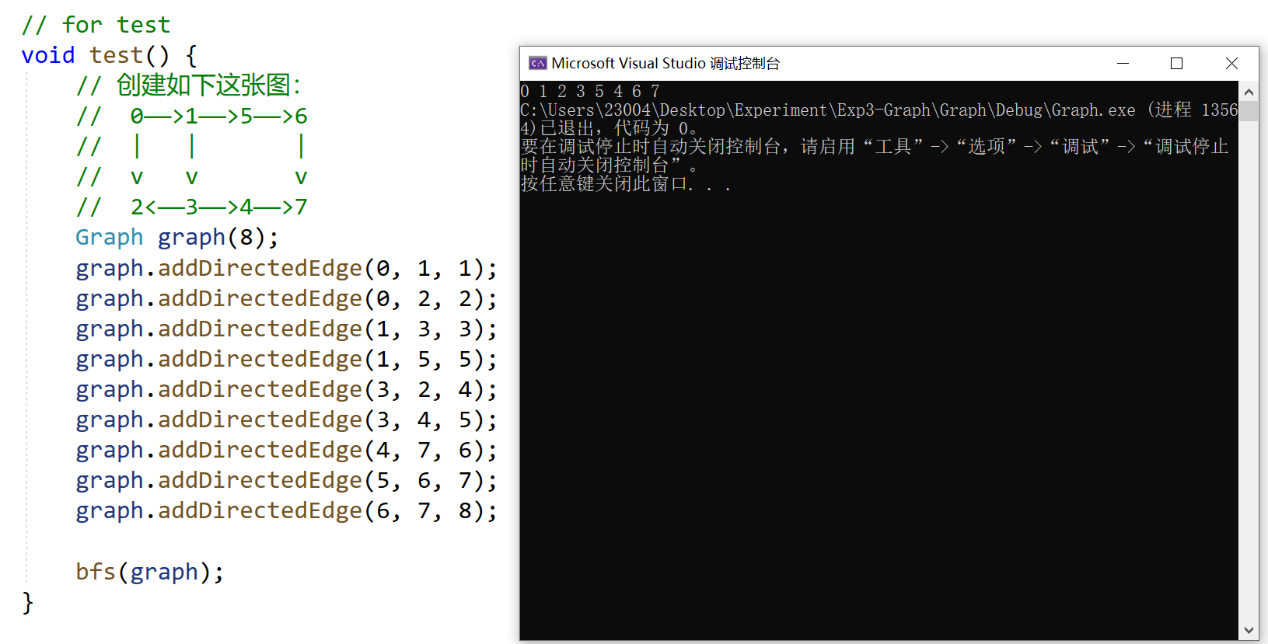
邻接矩阵：



## 4.2图的深度优先遍历实验结果展示



## 4.3图的宽度优先遍历实验结果展示



# 第五章 实验总结

## 5.1 实验结果的总结

通过本次实验，我们较为详细和深刻地学习和总结了图论的知识，对图的相关算法有了系统、全面的认识，实验内容包括图的基本概念和表示方法（邻接表和邻接矩阵），图的遍历（深度优先遍历和广度优先遍历），最小生成树算法（Kruskal算法和Prim算法）和最短路径算法（Dijkstra算法和Floyd算法）。

## 5.2 实验的不足和改进方向

1. 学习完图论的相关算法后，缺少对实际问题的思考与应用，无法将实验内容运用到实际问题中。

2. 代码只用了C++语言来实现，以后将尝试用其他编程语言再实现一遍实验中的算法。