|  |  |
| --- | --- |
| **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

学 号 632107030426

姓 名 屈耘毅

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

## 最小生成树算法

在一个无向连通图中，若选择一棵生成树，使得树上所有边的权值和最小，则这棵生成树称为该图的最小生成树。其中生成树是指原图中选择其中一些边和相应的节点，形成一棵树，包含原图所有节点但边数比原图少一。最小生成树是指这些生成树中，边权和最小的那一棵。这里分别介绍Kruskal算法和Prim算法。

## 3.1 Kruskal算法

Kruskal算法是求解最小生成树问题的一种常见算法。其基本思想是从图中按照边权从小到大的顺序选择边，直到选出n-1条边为止，其中n为图中节点数。

具体实现步骤如下：

1. 对所有边按照权值从小到大进行排序。

2. 从权值最小的边开始，依次选择每条边，如果这条边的加入不会形成环路，则将它加入到生成树中。

3. 重复步骤2，直到选出n-1条边为止，其中n为图中节点数。

在实现上，可以使用并查集来判断一条边的加入是否会形成环路，以保证最终生成的是一棵树。

Kruskal算法的时间复杂度为O(mlogm)，其中m为边数。

①邻接表法实现Kruskal算法代码如下：

// Kruskal算法（邻接表）

class AdjacencyList\_Kruskal {

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);

}

};

// 并查集

class UnionFind {

unordered\_map<int, int>fatherMap;

unordered\_map<int, int>sizeMap;

public:

UnionFind(int n) {

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

fatherMap[i] = i;

sizeMap[i] = 1;

}

}

int findHead(int element) {

stack<int>path;

while (fatherMap[element] != element) {

path.push(element);

element = fatherMap[element];

}

while (!path.empty()) {

fatherMap[path.top()] = element;

path.pop();

}

return element;

}

bool isSameSet(int a, int b) {

return findHead(a) == findHead(b);

}

void unionSets(int a, int b) {

int aF = findHead(a);

int bF = findHead(b);

if (aF != bF) {

int big = sizeMap[aF] >= sizeMap[bF] ? aF : bF;

int small = big == aF ? bF : aF;

fatherMap[small] = big;

sizeMap[big] = sizeMap[big] + sizeMap[small];

sizeMap.erase(small);

}

}

};

vector<Edge> generateEdges(Graph graph) {

vector<Edge>res;

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

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

res.push\_back(edge);

}

}

return res;

}

class Cmp {

public:

bool operator()(Edge& a, Edge& b) {

return a.weight < b.weight;

}

};

// Kruskal算法

vector<Edge> Kruskal(Graph graph) {

vector<Edge>edges = generateEdges(graph); // 生成所有边的信息

sort(edges.begin(), edges.end(), Cmp()); // 按边的权值从小到大排序

UnionFind uf(graph.n);

vector<Edge>res; // 组成最小生成树的边

for (Edge& edge : edges) {

int u = edge.from;

int v = edge.to;

if (!uf.isSameSet(u, v)) {

uf.unionSets(u, v);

res.push\_back(edge);

}

}

return res;

}

// for test

void test() {

Graph graph(6);

graph.addUnDirectedEdge(0, 1, 5);

graph.addUnDirectedEdge(0, 2, 1);

graph.addUnDirectedEdge(1, 2, 3);

graph.addUnDirectedEdge(1, 3, 3);

graph.addUnDirectedEdge(2, 3, 2);

//graph.addUnDirectedEdge(2, 3, 2);

graph.addUnDirectedEdge(2, 5, 4);

graph.addUnDirectedEdge(3, 4, 2);

graph.addUnDirectedEdge(4, 5, 3);

vector<Edge>res = Kruskal(graph);

cout << "该图的最小生成树所需的边有：" << endl;

for (Edge& e : res) {

cout << e.from << "——" << e.to << " 权值：" << e.weight << endl;

}

}

};

②邻接矩阵法实现Kruskal算法代码如下：

// Kruskal算法（邻接矩阵）

class AdjacencyMatrix\_Kruskal {

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;

}

};

// 并查集

class UnionFind {

unordered\_map<int, int>fatherMap;

unordered\_map<int, int>sizeMap;

public:

UnionFind(int n) {

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

fatherMap[i] = i;

sizeMap[i] = 1;

}

}

int findHead(int element) {

stack<int>path;

while (fatherMap[element] != element) {

path.push(element);

element = fatherMap[element];

}

while (!path.empty()) {

fatherMap[path.top()] = element;

path.pop();

}

return element;

}

bool isSameSet(int a, int b) {

return findHead(a) == findHead(b);

}

void unionSets(int a, int b) {

int aF = findHead(a);

int bF = findHead(b);

if (aF != bF) {

int big = sizeMap[aF] >= sizeMap[bF] ? aF : bF;

int small = big == aF ? bF : aF;

fatherMap[small] = big;

sizeMap[big] = sizeMap[big] + sizeMap[small];

sizeMap.erase(small);

}

}

};

vector<pair<int, pair<int, int>>> generateEdges(Graph graph) {

vector<pair<int, pair<int, int>>>res;

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

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

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

res.push\_back(make\_pair(graph.g[i][j], make\_pair(i, j)));

}

}

}

return res;

}

// Kruskal

// pair<int, pair<int, int>>：<边的权值, <边的起点, 边的终点>>

vector<pair<int, pair<int, int>>> kruskal(Graph graph) {

int n = graph.n;

auto edges = generateEdges(graph); // 生成所有边的信息

sort(edges.begin(), edges.end(), less<pair<int, pair<int, int>>>()); // 按边的权值从小到大排序

UnionFind uf(n);

vector<pair<int, pair<int, int>>>res; // 组成最小生成树的边

for (auto& it : edges) {

int from = it.second.first;

int to = it.second.second;

if (!uf.isSameSet(from, to)) {

uf.unionSets(from, to);

res.push\_back(it);

}

}

return res;

}

// for test

void test() {

Graph graph(6);

graph.addUnDirectedEdge(0, 1, 5);

graph.addUnDirectedEdge(0, 2, 1);

graph.addUnDirectedEdge(1, 2, 3);

graph.addUnDirectedEdge(1, 3, 3);

graph.addUnDirectedEdge(2, 3, 2);

//graph.addUnDirectedEdge(2, 3, 2);

graph.addUnDirectedEdge(2, 5, 4);

graph.addUnDirectedEdge(3, 4, 2);

graph.addUnDirectedEdge(4, 5, 3);

auto res = kruskal(graph);

cout << "该图的最小生成树所需的边有：" << endl;

for (auto& e : res) {

cout << e.second.first << "——" << e.second.second << " 权值：" << e.first << endl;

}

}

};

## 3.2 Prim算法

Prim算法是一种用于构造无向带权图的最小生成树的贪心算法，其基本思想是从一个顶点开始，每次将一个未被选择的且与当前生成树相邻的权值最小的顶点加入生成树中。下面是Prim算法的详细步骤：

1. 选择一个起点作为生成树的根节点，将其标记为已访问。

2. 将与起点相邻的边加入一个优先队列中，队列中的元素按照边的权值从小到大排列。

3. 取出队列中权值最小的边，并判断其所连接的另一个顶点是否已经被访问。如果已经被访问，则将该边舍弃；否则将该边加入生成树中，并将该顶点标记为已访问。

4. 重复执行步骤2和步骤3，直到所有的顶点都被访问过。

在实际应用中，Prim算法通常使用堆来实现优先队列，以提高算法的效率。

①邻接表法实现Prim算法代码如下：

// Prim算法（邻接表）

class AdjacencyList\_Prim {

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);

}

};

// 定义边的比较函数

class CompareEdge {

public:

bool operator()(const Edge& e1, const Edge& e2) const {

return e1.weight > e2.weight;

}

};

vector<Edge> Prim(Graph graph) {

vector<Edge> res;

vector<bool> vis(graph.n, false);

priority\_queue<Edge, vector<Edge>, CompareEdge>pq;

// 将0号顶点作为起点

vis[0] = true;

for (Edge& e : graph.adjList[0]) { // 把0顶点相邻的顶点入队

pq.push(e);

}

while (!pq.empty()) {

Edge e = pq.top();

pq.pop();

if (vis[e.to]) {

continue;

}

vis[e.to] = true;

res.push\_back(e);

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

if (!vis[next.to]) {

pq.push(next);

}

}

}

return res;

}

// for test

void test() {

Graph graph(6);

graph.addUnDirectedEdge(0, 1, 5);

graph.addUnDirectedEdge(0, 2, 1);

graph.addUnDirectedEdge(1, 2, 3);

graph.addUnDirectedEdge(1, 3, 3);

graph.addUnDirectedEdge(2, 3, 2);

//graph.addUnDirectedEdge(2, 3, 2);

graph.addUnDirectedEdge(2, 5, 4);

graph.addUnDirectedEdge(3, 4, 2);

graph.addUnDirectedEdge(4, 5, 3);

vector<Edge>res = Prim(graph);

cout << "该图的最小生成树所需的边有：" << endl;

for (Edge& e : res) {

cout << e.from << "——" << e.to << " 权值：" << e.weight << endl;

}

}

};

②邻接矩阵法实现Prim算法代码如下：

// Prim算法（邻接矩阵）

class AdjacencyMatrix\_Prim {

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;

}

};

// Prim

// pair<int, pair<int, int>>：<边的权值, <边的起点, 边的终点>>

vector<pair<int, pair<int, int>>> prim(Graph graph) {

int n = graph.n;

vector<bool>vis(n, false);

vector<pair<int, pair<int, int>>>res;

priority\_queue<pair<int, pair<int, int>>,

vector<pair<int, pair<int, int>>>,

greater<pair<int, pair<int, int>>>>pq;

// 将0号顶点作为起点

vis[0] = true;

for (int i = 0; i < n; i++) { // 把0顶点相邻的顶点入队

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

pq.push(make\_pair(graph.g[0][i], make\_pair(0, i)));

}

}

while (!pq.empty()) {

auto cur = pq.top();

pq.pop();

int from = cur.second.first;

int to = cur.second.second;

if (vis[to]) {

continue;

}

vis[to] = true;

res.push\_back(cur);

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

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

pq.push(make\_pair(graph.g[to][next], make\_pair(to, next)));

}

}

}

return res;

}

// for test

void test() {

Graph graph(6);

graph.addUnDirectedEdge(0, 1, 5);

graph.addUnDirectedEdge(0, 2, 1);

graph.addUnDirectedEdge(1, 2, 3);

graph.addUnDirectedEdge(1, 3, 3);

graph.addUnDirectedEdge(2, 3, 2);

graph.addUnDirectedEdge(2, 5, 4);

graph.addUnDirectedEdge(3, 4, 2);

graph.addUnDirectedEdge(4, 5, 3);

auto res = prim(graph);

cout << "该图的最小生成树所需的边有：" << endl;

for (auto& e : res) {

cout << e.second.first << "——" << e.second.second << " 权值：" << e.first << endl;

}

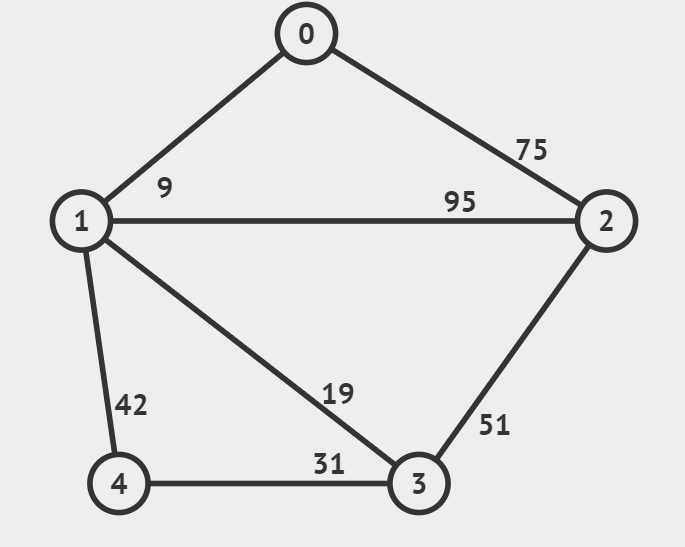
}

};

# 第四章 实验结果

## 4.1 Kruskal算法实验结果展示

建立如下图结构：

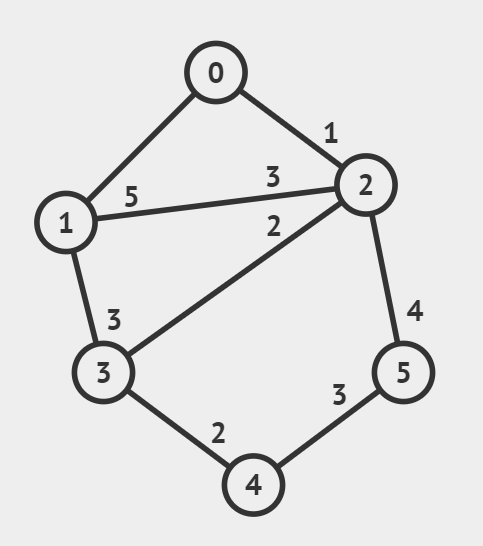


代码运行结果如下：



## 4.2 Prim算法实验结果展示

建立如下图结构：



代码运行结果如下：



# 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，同时对于图的算法的使用有了一定的经验，并且加深了我对图的各种算法的印象，以及重新温习了书中对图的算法的相关知识。

## 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. 对于图的相关算法的使用缺少经验，此次实验只是对图的算法的最基本使用，并未学习到其更深层次的使用方法。