|  |  |
| --- | --- |
| **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 632107030404

姓 名 鲁轩廷

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

## 最短路径算法

最短路径算法用于求解两点之间路径上权值之和最小的路径。在图论中，最短路径通常是指有向加权图中两个顶点之间的最短路径。

最短路径算法有多种，常用的包括Dijkstra算法，Floyd算法和Bellman-Ford算法等。这里介绍单源最短路径算法——Dijkstra，和多元最短路径算法——Floyd。

## 3.1单源最短路径算法——Dijkstra

Dijkstra算法是一种用于解决带权重无向图或有向图中单源最短路径问题的算法。该算法通过对图中的节点进行标记和松弛操作来逐步确定从源节点到所有其他节点的最短路径。

具体思路如下：

1. 初始化，将起始节点标记为已访问，并将该节点到其它所有节点的距离初始化为无穷大，将起始节点到它自身的距离初始化为0。

2. 对于所有与起始节点直接相连的节点，更新它们的距离（如果当前距离大于通过起始节点到达该节点的距离，则更新距离）。这个过程被称为“松弛操作”。

3. 从所有未标记的节点中选择当前距离最短的节点，并将其标记为已访问。

4. 重复步骤2和3，直到所有节点都被标记为已访问或者没有未标记的节点可供访问为止。

最终，Dijkstra算法会输出每个节点到起始节点的最短距离。

Dijkstra算法的时间复杂度取决于数据结构的实现。当使用优先队列（如二叉堆）来存储未被访问的节点时，时间复杂度为 O(E\*logV)，其中E为边数，V为节点数。

①邻接表法实现Dijkstra算法代码如下：

// Dijkstra（邻接表）

class AdjacencyList\_Dijkstra {

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);

}

};

// Dijkstra

vector<int> dijkstra(Graph graph, int s) {

int n = graph.n;

vector<int>dist(n, INT\_MAX); // dist[i]表示从源点s到点i的最短距离

dist[s] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>pq;

pq.push(make\_pair(0, s));

while (!pq.empty()) {

int d = pq.top().first;

int u = pq.top().second;

pq.pop();

if (dist[u] < d) { // 如果当前到源点的最短距离已经更新过，就跳过

continue;

}

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

int v = edge.to;

if (dist[u] + edge.weight < dist[v]) { // 发现一条到达v的更短路径

dist[v] = dist[u] + edge.weight; // 更新最短距离

pq.push(make\_pair(dist[v], v));

}

}

}

return dist;

}

// 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);

vector<int>res = dijkstra(graph, 0);

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

cout << "源点0到" << i << "号的最短距离为：" << res[i] << endl;

}

}

};

②邻接矩阵法实现Dijkstra算法代码如下：

// Dijkstra算法（邻接矩阵）

class AdjacencyMatrix\_Dijkstra {

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;

}

};

// Dijkstra

vector<int> dijkstra(Graph graph, int s) {

int n = graph.n;

vector<int>dist(n, INT\_MAX); // dist[i]表示从源点s到点i的最短距离

dist[s] = 0;

priority\_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>>pq;

pq.push(make\_pair(0, s));

while (!pq.empty()) {

int d = pq.top().first;

int u = pq.top().second;

pq.pop();

if (dist[u] < d) { // 如果当前到源点的最短距离已经更新过，就跳过

continue;

}

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

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

if (dist[u] + graph.g[u][v] < dist[v]) { // 发现一条到达v的更短路径

dist[v] = dist[u] + graph.g[u][v]; // 更新最短路径

pq.push(make\_pair(dist[v], v));

}

}

}

}

return dist;

}

// 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);

vector<int>res = dijkstra(graph, 0);

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

cout << "源点0到" << i << "号的最短距离为：" << res[i] << endl;

}

}

};

## 3.2多源最短路径算法——Floyd

Floyd算法是一种用于求解图中所有节点对之间的最短路径的动态规划算法。它的基本思想是利用动态规划的思想，将所有节点之间的最短路径逐步更新和优化，直到得到所有节点对之间的最短路径。

具体来说，Floyd算法使用一个二维数组来记录任意两个节点之间的最短路径长度。初始时，这个数组被初始化为邻接矩阵，即如果节点i和节点j之间有一条边，则这个数组中第i行第j列的值为这条边的权重；否则，这个数组中第i行第j列的值为无穷大。

然后，Floyd算法开始进行逐步优化。假设当前正在考虑节点k，我们要尝试更新任意两个节点i和j之间的最短路径。如果从节点i到节点j经过节点k比直接从节点i到节点j的路径更短，那么我们就更新节点i到节点j的最短路径为经过节点k的路径。

最终，当我们考虑完所有的节点k之后，二维数组中的值就是任意两个节点之间的最短路径长度。注意，如果在更新过程中发现任意两个节点之间没有路径，则这两个节点之间的最短路径长度为无穷大。

Floyd算法的时间复杂度为O(n^3)，其中n为图中节点的个数。它可以用于解决带权图中的最短路径问题，适用于任何形状的图，包括有向图和无向图。

①邻接表法实现Floyd算法代码如下：

// Floyd算法（邻接表）

class AdjacencyList\_Floyd {

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);

}

};

// Floyd

vector<vector<int>> floyd(Graph graph) {

int n = graph.n;

vector<vector<int>>dist(n, vector<int>(n));

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

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

dist[i][j] = i == j ? 0 : INT\_MAX;

}

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

dist[edge.from][edge.to] = edge.weight;

}

}

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

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

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

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX

&& dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

return dist;

}

// 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);

vector<vector<int>>dist = floyd(graph);

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

for (int j = 0; j < dist[i].size(); j++) {

cout << i << "号节点与" << j << "号节点间的最短路径长度为：" << dist[i][j] << endl;

}

}

}

};

②邻接矩阵法实现Floyd算法代码如下：

// Floyd算法（邻接矩阵）

class AdjacencyMatrix\_Floyd {

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;

}

};

// Floyd

vector<vector<int>> floyd(Graph graph) {

int n = graph.n;

vector<vector<int>>dist(n, vector<int>(n));

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

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

dist[i][j] = i == j ? 0 : INT\_MAX;

}

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

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

dist[i][j] = graph.g[i][j];

}

}

}

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

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

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

if (dist[i][k] != INT\_MAX && dist[k][j] != INT\_MAX

&& dist[i][k] + dist[k][j] < dist[i][j]) {

dist[i][j] = dist[i][k] + dist[k][j];

}

}

}

}

return dist;

}

// 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);

vector<vector<int>>dist = floyd(graph);

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

for (int j = 0; j < dist[i].size(); j++) {

cout << i << "号节点与" << j << "号节点间的最短路径长度为：" << dist[i][j] << endl;

}

}

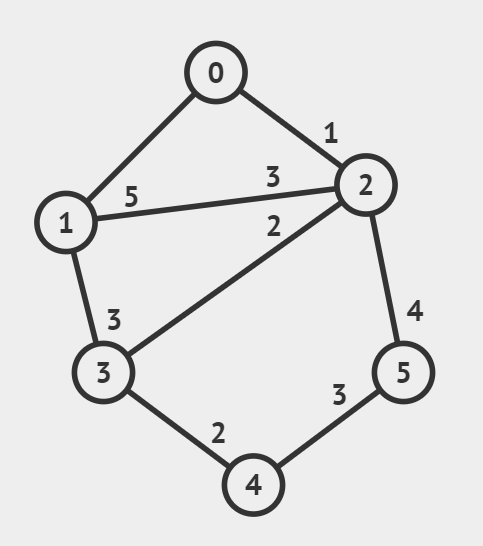
}

};

# 第四章 实验结果

## 4.1 Dijkstra算法实验结果展示

建立如下图结构：

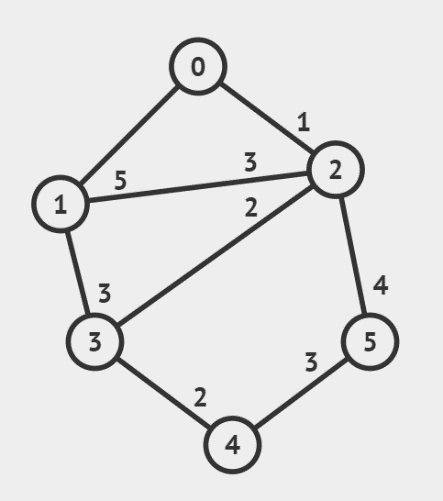


代码运行结果如下：

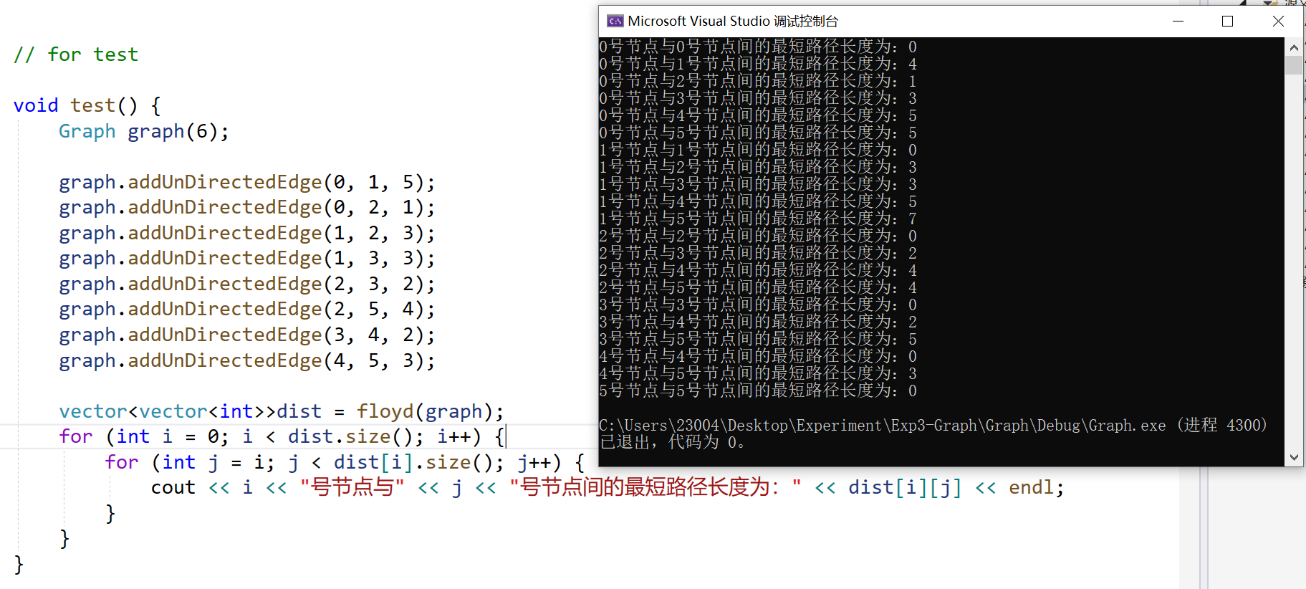


## 4.2 Floyd算法实验结果展示

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。