**《数据结构》实验报告（第七章）**

**学号： \_58122231\_ \_ 姓名 ：\_\_\_\_\_\_\_陆文韬\_\_\_\_**

**实验题号： P352 3 5，P359 3 实验日期：\_2023.12.19\_\_\_\_**

**实验一**

**1．问题描述：**描述实验内容和要求以及需要解决的问题。

假定图采用邻接表表示，写一个完整的深度优先搜索的C++函数，用适当的图检测函数的正确性

假定图采用邻接表表示，写一个完整的广度优先搜索的C++函数，用适当的图检测函数的正确性

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

深度优先搜索：采用递归的思想，对图进行递归检索

广度优先搜索：采用队列来维护，访问当前的节点，然后将它的所有邻接点全部放入队尾

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

addEdge：向无向图中加入边

DFSOutput：递归执行深度优先搜索

DFS：深度优先搜索

BFS：广度优先搜索

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

Graph g(9);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 5);

g.addEdge(2, 6);

g.addEdge(3, 7);

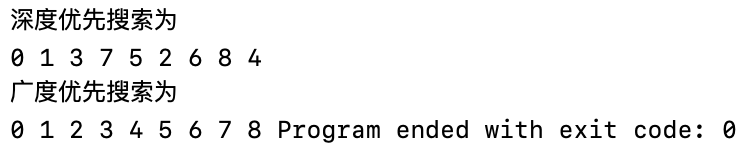
g.addEdge(5, 7);

g.addEdge(6, 7);

g.addEdge(7, 8);

g.addEdge(6, 8);

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

在实验编写的过程中，要对深度优先搜索和广度优先搜索的算法思想非常了解，根据算法选择不同的实现方法，在深度优先搜索中采用递归方法，在广度优先搜索中采用队列来实现。

**7. 源代码：**

#include <iostream>

#include <list>

#include <vector>

#include<queue>

**using** **namespace** std;

**class** Graph//表示一个无向图

{

**private**:

**int** V; //表示图中顶点数

list<**int**>\* adj; //表示邻接表，存储图的边信息

**public**:

//构造函数

Graph(**int** V)

{

**this**->V = V;

adj = **new** list<**int**>[V];

}

//向图中增加一条边

**void** addEdge(**int** v, **int** w)

{

//在顶点v和w之间增加一条边，并且是无向图

adj[v].push\_back(w);

adj[w].push\_back(v);

}

//递归执行深度优先搜索

**void** DFSOutput(**int** v, vector<**bool**>& visited)

{

visited[v] = **true**;

cout << v << " ";

**for** (**int** neighbor : adj[v])

{

//如果邻接顶点还没有被访问过就进行递归输出，知道所有邻接顶点都被访问过

**if** (!visited[neighbor])

DFSOutput(neighbor, visited);

}

}

//深度优先搜索

**void** DFS(**int** v)

{

vector<**bool**> visited(V, **false**); //所有顶点没有被访问过

DFSOutput(v, visited);

}

//广度优先搜素

**void** BFS(**int** s)

{

vector<**bool**> visited(V, **false**); //所有顶点没有被访问过

queue<**int**> queue;

//记录顶点s为访问过的

visited[s] = **true**;

queue.push(s);

**while** (!queue.empty())

{

s = queue.front();

cout << s << " ";

queue.pop();

//如果一个顶点的邻接顶点没有被访问过就记录它们并让其进入队列

**for** (**int** neighbor : adj[s])

{

**if** (!visited[neighbor])

{

visited[neighbor] = **true**;

queue.push(neighbor);

}

}

}

}

};

**int** main()

{

//创建一个有9个顶点的图

Graph g(9);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(2, 5);

g.addEdge(2, 6);

g.addEdge(3, 7);

g.addEdge(5, 7);

g.addEdge(6, 7);

g.addEdge(7, 8);

g.addEdge(6, 8);

cout << "深度优先搜索为" << endl;

g.DFS(0);

cout << "\n广度优先搜索为" << endl;

g.BFS(0);

**return** 0;

}

**实验二**

**1．问题描述：**描述实验内容和要求以及需要解决的问题。

将程序6.7完善成为C++函数，用于找到一个最小代价生成树。函数的复杂度应该为O(n\*n)，这里n是输入图的顶点数。证明确实达到了这样的复杂度。

**2．算法思想：**详细描述解决相应问题所需要的算法设计思想。

采用Prim算法，首先确定一个顶点，每次选择未访问节点中距离已知集合最近的点加入集合，然后更新未访问节点到已知集合的距离。

**3．功能函数：**描述所设计的功能函数。如果有多个函数，需要描述它们之间的关系。

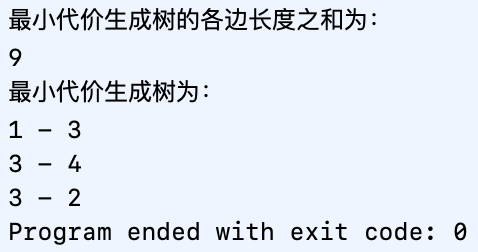
Prim：根据prim算法生成最小代价生成树

**4．测试数据：**设计测试数据，或具体给出测试数据。

提示：要求测试数据能全面地测试所设计程序的功能。

4 5 1 2 5 1 3 2 1 4 8 2 3 4 3 4 3

**5．测试情况：**给出程序的测试情况，分析运行结果，显示实验结果截图。

****

**6．实验总结：**写出实验过程中遇到的问题，以及问题的解决过程。分析算法的时间复杂度和空间复杂度，总结实验心得体会。

要对prim算法有充分的理解，在程序编写的过程中，需要维护的数组非常多，包括dist，TV，parent等，在循环的执行过程中，要注意parent数组的更新。

**7. 源代码：**

#include<iostream>

#include<vector>

**using** **namespace** std;

**const** **int** INF = 0x3f3f3f3f;//表示无穷大。

**int** TV[100][100];//二维数组，表示图中各顶点之间的边的权重。如果两个顶点之间没有直接的边，则权重为 INF（无穷大）

**int** dist[100];//一维数组，存储每个顶点到集合 S（已经加入到 MST 中的顶点集合）的最小距离

**int** parent[100];//一维数组，记录每个顶点在 MST 中的父顶点

**int** result;//变量，用于存储 MST 的总边权重

**bool** visited[100];//一维布尔数组，标记每个顶点是否已经被加入到 MST 中

vector<pair<**int**, **int**>> mstEdges; //向量，用于存储 MST 中的边

**void** prim(**int** vertex, **int** edge)

{

**for** (**int** i = 1; i <= vertex; i++)

{

dist[i] = TV[1][i];//更新各顶点的dist[]为到顶点1的距离

visited[i] = **false**;

parent[i] = i == 1 ? -1 : 1;//初始化父顶点，除了顶点1外，所有顶点的父顶点暂时都设为1

}

visited[1] = **true**;

result = 0;

**for** (**int** i = 2; i <= vertex; i++)

{

**int** temp = INF;//初始化距离

**int** subscript = -1;//接下来去寻找离集合S最近的点加入到集合中，用subscript记录这个点的下标。

**for** (**int** j = 1; j <= vertex; j++)

{

**if** (!visited[j] && dist[j] < temp)//如果这个点没有加入集合S，而且这个点到集合的距离小于temp就将下标赋给subscript

{

temp = dist[j];//更新集合V到集合S的最小值

subscript = j;//把点赋给subscript

}

}

**if** (subscript == -1)//如果subscript==-1，意味着在集合V找不到边连向集合S，生成树构建失败，将result赋值正无穷表示构建失败，结束函数

{

result = INF;

**return**;

}

visited[subscript] = **true**;//如果找到了这个点，就把它加入集合S

result += dist[subscript];//加上这个点到集合S的距离

mstEdges.push\_back(make\_pair(parent[subscript], subscript)); //记录最小生成树的边

**for** (**int** j = 1; j <= vertex; j++)

{

**if** (!visited[j] && TV[subscript][j] < dist[j])

{

dist[j] = min(dist[j], TV[subscript][j]);//用新加入的点更新dist[]

parent[j] = subscript;//更新其他顶点的父顶点为新加入的顶点

}

}

}

cout << endl;

}

**int** main()

{

cout << "请输入图的点数和边数:" << endl;

**int** vertex, edge;

cin >> vertex >> edge;//读入这个图的点数n和边数m

**for** (**int** i = 1; i <= vertex; i++)

{

**for** (**int** j = 1; j <= vertex; j++)

{

TV[i][j] = INF;//初始化任意两个点之间的距离为正无穷（表示这两个点之间没有边）

}

dist[i] = INF;//初始化所有点到集合S的距离都是正无穷

}

**for** (**int** i = 1; i <= edge; i++)

{

cout << "请输入两条边和它们之间的距离:" << endl;

**int** vertex1, vertex2, width;

cin >> vertex1 >> vertex2 >> width;//读入a，b两个点之间的边

TV[vertex1][vertex2] = TV[vertex2][vertex1] = width;//由于是无向边，我们对TV[vertex1][vertex2]和TV[vertex2][vertex1]都要赋值

}

prim(vertex, edge);//调用prim函数

**if** (result == INF)//如果result的值是正无穷，表示不能该图不能转化成一棵树，输出orz

cout << "orz";

**else**

{

cout << "最小代价生成树的各边长度之和为：\n" << result << endl;//否则就输出结果result

cout << "最小代价生成树为：" << endl;

**for** (**const** **auto**& edge : mstEdges)

{

cout << edge.first << " - " << edge.second << endl;

}

}

**return** 0;

// 4 5 1 2 5 1 3 2 1 4 8 2 3 4 3 4 3

}