图

姓名： 学号：

**1 实验目的：**

掌握图的存储结构，能够进行图的遍历，理解最小生成树算法。

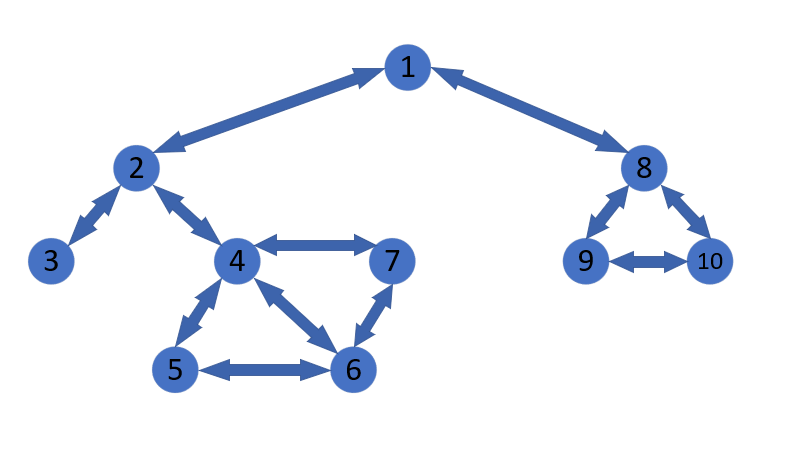
**2 实验学时：4**

**3 实验内容：**

1. 编写图类，实现基于邻接矩阵的存储；
2. 实现图的DFS、BFS算法
3. 实现图的最小生成树算法（选做）

**4实验截图：**

**实验数据连通图**





**5实验源码：**

#pragma once

typedef int elementType;

#define queuesize 100

class queue

{

public:

queue();

~queue();

void initqueuq(queue \*Q);

bool queueEmpty(queue& Q);

bool queueFull(queue& Q);

void queuefront(queue &Q,elementType &x);

void enqueue(queue\* Q, elementType x);

void outqueue(queue\* Q);

elementType data[queuesize];

int front, rear;

};

#include "queue.h"

queue::queue()

{

}

queue::~queue()

{

}

void queue::initqueuq(queue \* Q)

{

Q->front =Q->rear = 0;

}

bool queue::queueEmpty(queue& Q)

{

if (Q.front==Q.rear)

{

return true;

}

else

{

return false;

}

}

bool queue::queueFull(queue& Q)

{

if (((Q.rear+1)%queuesize)==Q.front)

{

return true;

}

else

{

return false;

}

}

void queue::queuefront(queue& Q, elementType& x)

{

if (queueEmpty(Q))

{

return;

}

else

{

x = Q.*data*[(Q.front + 1) % queuesize];

}

}

void queue::enqueue(queue\* Q, elementType x)

{

if (queueFull(\*Q))

{

return;

}

else

{

Q->rear = ((Q->rear) + 1) % queuesize;

Q->*data*[Q->rear] = x;

}

}

void queue::outqueue(queue\* Q)

{

if (queueEmpty(\*Q))

{

return;

}

else

{

Q->front = (Q->front + 1) % queuesize;

}

}

#pragma once

#include <iostream>

using namespace *std*;

#define INF 65535

#define MaxVerNum 10

typedef int elementType;

typedef int cellType;

typedef enum{UDG,UDN,DG,DN}Graphkind;

typedef struct Graphadj

{

elementType data[MaxVerNum];

cellType adj[MaxVerNum][MaxVerNum];

int VerNum;

int ArcNum;

Graphkind Gkind;

}GraphG;

class graph1

{

public:

graph1();

~graph1();

void createG(GraphG &T, int V, int A, Graphkind K,elementType data[10], cellType\*\* L);

void BFS(GraphG &T,int v);

void DFS(GraphG &T,int v);

};

#include "graph1.h"

#include "queue.h"

#include <iostream>

using namespace *std*;

bool visited[MaxVerNum] = { false };

graph1::graph1()

{

}

graph1::~graph1()

{

}

void graph1::createG(GraphG &T,int V,int A,Graphkind K,elementType data[10],cellType\*\*L)

{

T.VerNum = V;

T.ArcNum = A;

T.Gkind = K;

for (int i=0;i<MaxVerNum;i++)

{

T.data[i] = data[i];

//cout << T.data[i];

}

for (int j=0;j<MaxVerNum;j++)

{

for (int k = 0; k <MaxVerNum; k++)

{

T.adj[j][k] = \*((int\*)L + j \* MaxVerNum + k);

//cout << T.adj[j][k];

}

}

//cout << T.data[0];

}

void graph1::DFS(GraphG &T, int v)

{

*cout* <<T.data[v - 1]<<" ";

visited[v - 1] = true;

for (int w =0;w<T.VerNum;w++)

{

if ((T.adj[v-1][w]>=1)&&(T.adj[v-1][w]<INF)&&(!visited[w]))

{

DFS(T, w + 1);

}

}

}

void graph1::BFS(GraphG &T,int v)

{

int u;

bool visited[MaxVerNum] = {false};// 顶点是否被访问的标记

queue Q;

Q.initqueuq(&Q);

*cout* << T.data[v-1] << " ";

visited[v-1] = true;

Q.enqueue(&Q, v);

while (!Q.queueEmpty(Q))

{

Q.queuefront(Q, u);

Q.outqueue(&Q);

for (int j = 0; j <T.VerNum; j++)

{

if ((T.adj[u-1][j] >= 1) &&( T.adj[u-1][j]<INF)&&(!visited[j]))

{

*cout* <<T.data[j] << " ";

visited[j] = true;

Q.enqueue(&Q, j + 1); // 邻接点入队

}

}

}

*cout* << *endl*;

}

#include <iostream>

#include "graph1.h"

#include "queue.h"

int main()

{

elementType tst0[10] = {1,2,3,4,5,6,7,8,9,10};

cellType\* c[10];

cellType tst1[][10] = { {0,1,0,0,0,0,0,1,0,0},

{1,0,1,1,0,0,0,0,0,0},

{0,1,0,1,1,0,0,0,0,0},

{0,1,0,0,1,1,1,0,0,0},

{0,0,0,1,0,1,0,0,0,0},

{0,0,0,1,1,0,1,0,0,0},

{0,0,0,1,0,1,0,0,0,0},

{1,0,0,0,0,0,0,0,1,1},

{0,0,0,0,0,0,0,1,0,1},

{0,0,0,0,0,0,0,1,1,0}

};

c[0] = &tst1[0][0]; c[1] = tst1[1]; c[2] = tst1[2]; c[3] = tst1[3]; c[4] = tst1[4]; c[5] = tst1[5]; c[6] = tst1[6]; c[7] = tst1[7]; c[8] = tst1[8]; c[9] = tst1[9];

int v = 10, a = 12;

graph1 G; GraphG T; Graphkind k =UDG ;

G.createG(T, v, a, k, tst0,(int\*\*)tst1);

*cout* << "深度优先遍历算法" << *endl*;

G.DFS(T, 1); *cout* << *endl*;

*cout* << "广度优先遍历算法" << *endl*;

G.BFS(T, 1);

return 0;

}

**4 心得体会：**

这次图实验在图的构建方面由于输入数据太多改为直接用数组定义，由于时间关系没有进行编写能读取文本文件的输入部分以及对数据进行分类处理的分类部分，以及没有完成对于最小生成树的解决。在邻接矩阵传入函数时，由于二维数组的传递问题，修改了好久，这个地方真的是一言难尽，我不知道找到的方法是不是最简单的，总感觉有点复杂。DFS遍历算法由于是深度优先遍历，所以使用递归就可以很好的达到效果，但是BFS部分由于要先对一个顶点遍历后，再遍历其下一层次的其他顶点，所以要先确保某一层次全部顶点遍历完全，就要用到队列，由于队列是一端进行插入，一端进行删除，很好的对应了需求，某一顶点出队后，访问其邻接点并入队，直到全部顶点遍历完成，在两种遍历算法中都要用到一个bool型判断标志来标识遍历过的顶点，在我进行实际操作中由于DFS递归时，设定为局部变量的判断数组出现了bug，在BFS编写过程中，由于需要用到队列，但是之前对于栈的应用比较多，对于队列应用却很生疏，又去补习队列的知识。此外本实验中的图是无向图，对于有向图不知道是否还适用。