\*\*\*\*\*\*\*\*\*\*BFS（基于邻接矩阵实现Graph)\*\*\*\*\*\*\*\*\*

1. **book函数(book.h)：**

#include <iostream>

#include <cstdlib>

#include<cstring>

#include <time.h> // Used by timing functions

using namespace std;

//comp的模板函数

template <typename T>

inline bool Comp(T t1,T t2){

if(t1==t2)

return true;

return false;

}

//模板特殊化

template<>

bool Comp<string>(string s1,string s2){

if(strcmp(s1.c\_str(),s2.c\_str())==0)

return true;

return false;

}

// Assert: If "val" is false, print a message and terminate

// the program

void Assert(bool val, string s) {

if (!val) { // Assertion failed -- close the program 检查失败，关闭程序

cout << "Assertion Failed: " << s << endl;

exit(-1);

}

}

1. **graph抽象类(graph.h)：**

#ifndef GRAPH

#define GRAPH

template <typename VertexType>

class Graph {

private:

void operator =(const Graph&) {} // Protect assignment

Graph(const Graph&) {} // Protect copy constructor

public:

Graph() {} // Default constructor默认构造函数

virtual ~Graph() {} // Base destructor 析构函数

// Initialize a graph of n vertices 初始化一有n个顶点的图

virtual void Init(int n) =0;

// Return: the number of vertices and edges 返回图的顶点数、边数

virtual int n() =0;

virtual int e() =0;

// Return v's first neighbor 返回顶点v的第一个邻居

virtual int first(int v) =0;

// Return v's next neighbor 返回在w点之后的邻居（与物理存储中的存放位置有关）

virtual int next(int v, int w) =0;

//设置图的类型（有向图或无向图）

virtual void setType(bool flag)=0;

//获取图的类型

virtual bool getType()=0;

//找到(包含实际信息的）顶点在图中的位置

virtual int locateVex(VertexType u) =0;

//返回某个顶点的值(实际信息)

virtual VertexType getVex(int v)=0;

//给某个顶点赋值

virtual void putVex(int v,VertexType value) =0;

// Set the weight for an edge 为边(v1,v2)设置权值

virtual void setEdge(int v1, int v2, int wght) =0;

// Delete an edge删除边(v1,v2)

virtual void delEdge(int v1, int v2) =0;

// Determine if an edge is in the graph 判断边(i,j)是否在图中

virtual bool isEdge(int i, int j) =0;

// Return an edge's weight 返回边的权值

virtual int weight(int v1, int v2) =0;

// Get and Set the mark value for a vertex 取得和设置顶点的标志位

virtual int getMark(int v) =0;

virtual void setMark(int v, int val) =0;

};

#endif

1. **graph的邻接矩阵实现(graphm.h)：**

#include <iostream>

#include "graph.h"

#include "book.h"

#define MAX\_VERTEX\_NUM 40

#define UNVISITED 0

#define VISITED 1

using namespace std;

template <typename VertexType>

class Graphm : public Graph<VertexType> {

private:

int numVertex, numEdge; //顶点数和边数

bool undirected; // true表示无向图 false表示有向图

VertexType vexs[MAX\_VERTEX\_NUM]; //存储顶点信息

int \*\*matrix; // Pointer to adjacency matrix 指向邻接矩阵matrix

int \*mark; // Pointer to mark array 指向mark数组

public:

Graphm(int numVert) // Constructor 构造函数

{ Init(numVert); }

~Graphm() { // Destructor 析构函数

cout<<"gramat delete";

delete [] mark; // Return dynamically allocated memory回收动态分配内存

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

delete [] matrix[i];

delete [] matrix;

}

void Init(int n) { // Initialize the graph 初始化图

int i;

numVertex = n;

numEdge = 0;

mark = new int[n]; // Initialize mark array 初始化mark数组

for (i=0; i<numVertex; i++)

mark[i] = UNVISITED;

matrix = (int\*\*) new int\*[numVertex]; // Make matrix 初始化邻接矩阵

for (i=0; i<numVertex; i++)

matrix[i] = new int[numVertex];

for (i=0; i< numVertex; i++) // Initialize to 0 weights 初始化权值为0

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

matrix[i][j] = 0;

}

int n() { return numVertex; } // Number of vertices 返回节点数

int e() { return numEdge; } // Number of edges 返回边数

// Return first neighbor of "v" 返回v的第一个邻居

int first(int v) {

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

if (matrix[v][i] != 0) return i;

return numVertex; // Return n if none 如果没有邻居返回节点数

}

// Return v's next neighbor after w 返回v的在w后的邻居

int next(int v, int w) {

for(int i=w+1; i<numVertex; i++)

if (matrix[v][i] != 0)

return i;

return numVertex; // Return n if none如果没有邻居返回节点数

}

//设置图的类型（有向图或无向图）

void setType(bool flag){

undirected=flag;

}

//获取图的类型

bool getType(){

return undirected;

}

/\*\*返回顶点在图中的位置\*\*/

int locateVex(VertexType u){

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

if(Comp(u,vexs[i])) //Comp模板函数写在book.h中

return i;

}

return -1;

}

/\*\*返回某个顶点的值(实际信息) \*\*/

VertexType getVex(int v){

return vexs[v];

}

/\*\*给某个顶点赋值\*\*/

void putVex(int v,VertexType value){

vexs[v]=value;

}

// Set edge (v1, v2) to "wt" 设置边(v1,v2)的权值为wt

void setEdge(int v1, int v2, int wt) {

Assert(wt>0, "Illegal weight value");

if (matrix[v1][v2] == 0)

numEdge++;

matrix[v1][v2] = wt;

if(undirected){

matrix[v2][v1] = wt;

}

}

void delEdge(int v1, int v2) { // Delete edge (v1, v2) 删除边(v1,v2)

if (matrix[v1][v2] != 0){

numEdge--;

matrix[v1][v2] = 0;

if(undirected){

matrix[v2][v1] = 0;

}

}

}

bool isEdge(int i, int j) // Is (i, j) an edge? 判断边(i,j)是图中的一条边吗？

{ return matrix[i][j] != 0; }

int weight(int v1, int v2) { return matrix[v1][v2]; }

int getMark(int v) { return mark[v]; }

void setMark(int v, int val) { mark[v] = val; }

};

1. **create.h(建图)：**

#include<iostream>

#define LINELEN 80

using namespace std;

char\* getl(char\* buffer, int n, FILE\* fid) {

char\* ptr;

ptr = fgets(buffer, n, fid);

while ((ptr != NULL) && (buffer[0] == '#'))

ptr = fgets(buffer, n, fid);

return ptr;

}

/\*构建图\*/

Graph<string>\* createGraph(FILE\* fid) {

char buffer[LINELEN+1]; // Line buffer for reading 将从文件读取内容缓存到buffer

int i, v1, v2, dist;

/\*【读取顶点个数】 \*/

if (getl(buffer, LINELEN, fid) == NULL) // Unable to get number of vertices无法读取节点数

{ cout << "Unable to read number of vertices\n";

return NULL;

}

int num=atoi(buffer);

/\*【建图初始化】 \*/

Graph<string>\* G = new Graphm<string>(num);

/\*【读取图的顶点信息并存储】 \*/

if (getl(buffer, LINELEN, fid) == NULL)

{ cout << "Unable to read info of vertices\n";

return NULL ;

}

char\* cbuff=buffer;

string ver=strtok(cbuff," ");//空格分割各顶点信息

//存储前n-1个

for(i=0;i<num-1;i++){

G->putVex(i,ver);//存储顶点信息

ver=strtok(NULL," ");

}

//最后一个顶点信息之后的字符可能是\t之类的字符故不能和上面一样

//直接用空格分割

int k=0;

while(ver[k]>=33&&ver[k]<=126){

k++;

}

ver=ver.substr(0,k);//获取最后一个顶点信息

G->putVex(i,ver);//存储顶点信息

/\*【读取图的类型】 \*/

if (getl(buffer, LINELEN, fid) == NULL) // Unable to get graph type 读取图类型错误

{ cout << "Unable to read graph type\n";

return NULL ;

}

if (buffer[0] == 'U')//无向图

G->setType(true);//undirected = true;

else if (buffer[0] == 'D')//有向图

G->setType(false);//undirected = false;

else {

cout << "Bad graph type: |" << buffer << "|\n";

return NULL;

}

/\*【读取边数和边】 \*/

if (getl(buffer, LINELEN, fid) == NULL)

{ cout << "Unable to read num of edges\n";

return NULL ;

}

int num\_edge=atoi(buffer);

for(int m=0;m<num\_edge;m++){

getl(buffer, LINELEN, fid);

string sbuff=buffer;

string tvalue;

int loc1=sbuff.find(' ');//找到第一个分割空格的位置

tvalue=sbuff.substr(0,loc1);//读取第一个顶点信息到tvalue

//找到第一个顶点在图中的位置

v1 = G->locateVex(tvalue);

int loc2=sbuff.find(' ',loc1+1);//找第二个分割空格的位置

tvalue=sbuff.substr(loc1+1,loc2-loc1-1);//读取第二个顶点信息到tvalue

//找到第二个顶点在图中的位置

v2 = G->locateVex(tvalue);

//获取dist边权值

i=loc2+1;

dist=atoi(&buffer[i]);

//存储边

G->setEdge(v1,v2,dist);

}

return G;

}

/\*打印图\*/

void Gprint(Graph<string>\* G) {

int i, j;

cout << "顶点数：" << G->n() << "\n";

cout << "边 数: " << G->e() << "\n";

cout << "图类型: "<<(G->getType()?"无向图":"有向图")<<endl;

cout << "顶点信息：\n";

for(i=0;i<G->n();i++){

cout<<G->getVex(i)<<" ";

}

cout<<endl;

cout<<"边信息：\n";

if(G->getType()){

for (i=0; i<G->n(); i++) {

for(j=i; j<G->n(); j++){

if(G->weight(i, j)!=0){

cout<<G->getVex(i)<<" <--> "<<G->getVex(j)<<":"<<G->weight(i, j)<<endl;

}

}

}

}

else{

for (i=0; i<G->n(); i++) {

for(j=0; j<G->n(); j++){

if(G->weight(i, j)!=0){

cout<<G->getVex(i)<<" --> "<<G->getVex(j)<<":"<<G->weight(i, j)<<endl;

}

}

}

}

cout << "邻接表为:\n";

G->printGraphl();

}

1. **queue抽象类(queue.h)**

template <typename E> class Queue {

private:

void operator =(const Queue&) {} // Protect assignment

Queue(const Queue&) {} // Protect copy constructor

public:

Queue() {} // Default 构造函数

virtual ~Queue() {} // Base destructor 析构函数

// Reinitialize the queue. The user is responsible for reclaiming the storage used by the //queue elements. 用户要负责回收用于存储队列的内存

virtual void clear() = 0;

// Place an element at the rear of the queue.在队尾添加元素”it”

// it: The element being enqueued.

virtual void enqueue(const E&) = 0;

// Remove and return element at the front of the queue.删除并返回队首元素

// Return: The element at the front of the queue.

virtual E dequeue() = 0;

// Return: A copy of the front element.返回队首元素

virtual const E& frontValue() const = 0;

// Return: The number of elements in the queue.返回队列长度

virtual int length() const = 0;

};

1. **queue的顺序队列实现(aqueue.h)**

#include "queue.h"

#define defaultSize 100

// Array-based queue implementation顺序队列的实现

template <typename E> class AQueue: public Queue<E> {

private:

int maxSize; // Maximum size of queue 队列最大容量

int front; // Index of front element 首元素位置

int rear; // Index of rear element 尾元素位置

E \*listArray; // Array holding queue elements 数组存放队列元素

public:

AQueue(int size =defaultSize) { // Constructor 构造函数

// Make list array one position larger for empty slot

maxSize = size+1;

rear = 0; front = 1;

listArray = new E[maxSize];

}

~AQueue() { delete [] listArray; } // Destructor 析构函数

void clear() { rear = 0; front = 1; } // Reinitialize 重新初始化

void enqueue(const E& it) { // Put "it" in queue 入队

Assert(((rear+2) % maxSize) != front, "Queue is full");

rear = (rear+1) % maxSize; // Circular increment 循环增加

listArray[rear] = it;

}

E dequeue() { // Take element out 删除队首元素

Assert(length() != 0, "Queue is empty");

E it = listArray[front];

front = (front+1) % maxSize; // Circular increment 循环增加

return it;

}

const E& frontValue() const { // Get front value 取队首元素

Assert(length() != 0, "Queue is empty");

return listArray[front];

}

virtual int length() const // Return length 返回队列长度

{ return ((rear+maxSize) - front + 1) % maxSize; }

};

1. **manager.cpp(主函数 测试文件)：**

#include <iostream>

#include <cstring>

#include "graphm.h"

#include "graph.h"

#include "create.h"

#include "aqueue.h"

using namespace std;

Queue<int>\* Q;

void PreVisit(Graph<string>\* G, int v) {

cout << "PreVisit vertex " << G->getVex(v) << "\n";

}

void PostVisit(Graph<string>\* G, int v) {

cout << "PostVisit vertex " << G->getVex(v) << "\n";

}

void BFS(Graph<string>\* G, int start, Queue<int>\* Q) {

int v, w;

Q->enqueue(start); // Initialize Q 初始化队列Q

G->setMark(start, VISITED);

while (Q->length() != 0) { // Process all vertices on Q 处理再Q中的所有节点

v = Q->dequeue();

PreVisit(G, v); // Take appropriate action 前向遍历

for (w=G->first(v); w<G->n(); w = G->next(v,w))

if (G->getMark(w) == UNVISITED) {

G->setMark(w, VISITED);

Q->enqueue(w);

}

//PostVisit(G,v); 后向遍历 与前向遍历结果相反

}

}

void doTraverse(Graph<string>\* G, int v) {

cout<< "--------------------------------"<<endl

<<"从顶点" << G->getVex(v) << "起BFS遍历: \n";

BFS(G, v, Q);

}

void graphTraverse(Graph<string>\* G) {

int v;

for (v=0; v<G->n(); v++)

G->setMark(v, UNVISITED); // Initialize mark bits初始化标志位

for (v=0; v<G->n(); v++)

if (G->getMark(v) == UNVISITED)

doTraverse(G, v);

}

int main(){

FILE \*fid;

fid = fopen("test.gph", "rt");

Graph<string>\* G;

//创建图（create.h)

G = createGraph(fid);

//打印图（create.h)

Gprint(G);

Q = new AQueue<int>(G->n());

//遍历图

graphTraverse(G);

return 0;

}

1. **test.gph(输入文件)：**

5 # Number of vertices

A B C D E # 输入时每个顶点信息之间要以空格分割

U # Undirected graph

7 # 边数

A B 10

B D 5

A D 20

A C 3

C B 2

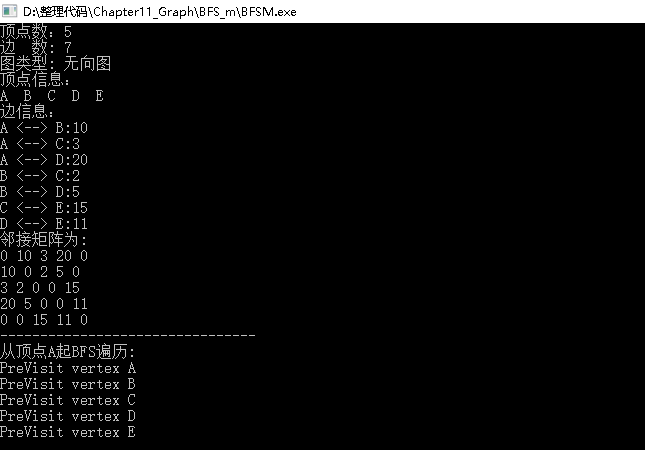
C E 15

D E 11

**如下图：**



1. **测试结果：**



【附录】

版本信息声明：

Dev-C++ 5.11

TDM-GCC 4.9.2 64-bit Release

部分代码来源：

<http://people.cs.vt.edu/~shaffer/Book/>

资料整理人：

信息科学与工程学院 物联1402班 201408080203 宁静仪

信息科学与工程学院 物联1402班 201408080222 吴彦彦