\*\*\*\*\*\*\*\*Kruskal（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. **heap.h：**

// Heap class 堆类

template <typename E, typename Comp> class heap {

private:

E\* Heap; // Pointer to the heap array 堆指针

int maxsize; // Maximum size of the heap 堆容量

int n; // Number of elements now in the heap 堆大小

// Helper function to put element in its correct place

//辅助插入元素到正确位置的函数

void siftdown(int pos) {

while (!isLeaf(pos)) { // Stop if pos is a leaf 当pos是叶节点时停止

int j = leftchild(pos); int rc = rightchild(pos);

if ((rc < n) && Comp::prior(Heap[rc], Heap[j]))

j = rc; // Set j to greater child's value 设置j为更大的子结点值

if (Comp::prior(Heap[pos], Heap[j])) return; // Done 完成

swap(Heap, pos, j);

pos = j; // Move down

}

}

public:

heap(E\* h, int num, int max) // Constructor 构造函数

{ Heap = h; n = num; maxsize = max; buildHeap(); }

int size() const // Return current heap size 返回当前堆大小

{ return n; }

bool isLeaf(int pos) const

{ return (pos >= n/2) && (pos < n); }

int leftchild(int pos) const

{ return 2\*pos + 1; } // Return leftchild position 返回左孩子位置

int rightchild(int pos) const

{ return 2\*pos + 2; } // Return rightchild position 返回右孩子位置

int parent(int pos) const // Return parent position 返回父结点位置

{ return (pos-1)/2; }

void buildHeap() // Heapify contents of Heap

{ for (int i=n/2-1; i>=0; i--) siftdown(i); }

// Insert "it" into the heap 插入it入堆

void insert(const E& it) {

Assert(n < maxsize, "Heap is full");

int curr = n++;

Heap[curr] = it; // Start at end of heap 先插入堆尾

// Now sift up until curr's parent > curr

//现在将it上拉直到curr's parent > curr

while ((curr!=0) &&

(Comp::prior(Heap[curr], Heap[parent(curr)]))) {

swap(Heap, curr, parent(curr));

curr = parent(curr);

}

}

// Remove first value 删除第一个元素

E removefirst() {

Assert (n > 0, "Heap is empty");

swap(Heap, 0, --n); // Swap first with last value 首元素与尾元素交换

if (n != 0) siftdown(0); // Siftdown new root val 下拉新root值维护堆性质

return Heap[n]; // Return deleted value 返回删除的元素

}

// Remove and return element at specified position

//删除并返回pos位置的元素

E remove(int pos) {

Assert((pos >= 0) && (pos < n), "Bad position");

if (pos == (n-1)) n--; // Last element, no work to do如果是尾元素，不需要做太多工作

else

{

swap(Heap, pos, --n); // Swap with last value 与尾元素交换

while ((pos != 0) &&

(Comp::prior(Heap[pos], Heap[parent(pos)]))) {

swap(Heap, pos, parent(pos)); // Push up large key 大值上拉

pos = parent(pos);

}

if (n != 0) siftdown(pos); // Push down small key 小值下拉

}

return Heap[n];

}

};

1. **uf.h**

// General tree representation for UNION/FIND

//一般树表示并查集

class ParPtrTree {

private:

int\* array; // Node array 顶点数组

int size; // Size of node array

int FIND(int) const; // Find root查找该顶点的根

public:

ParPtrTree(int); // Constructor构造函数

~ParPtrTree() { delete [] array; } // Destructor析构函数

void UNION(int, int); // Merge equivalences合并

bool differ(int, int); // True if not in same tree判断是否属来自同一棵树

};

ParPtrTree::ParPtrTree(int sz) { // Constructor构造函数

size = sz;

array = new int[sz]; // Create node array建立顶点数组

for(int i=0; i<sz; i++) array[i] = ROOT;

}

// Return True if nodes are in different trees

//若不是同一棵树返回true

bool ParPtrTree::differ(int a, int b) {

int root1 = FIND(a); // Find root of node a.找a的根

int root2 = FIND(b); // Find root of node b.找b的根

return root1 != root2; // Compare roots.比较两根

}

void ParPtrTree::UNION(int a, int b) { // Merge subtrees.合并树

int root1 = FIND(a); // Find root of node a

int root2 = FIND(b); // Find root of node b

if (root1 != root2) array[root2] = root1; // Merge

}

// FIND with path compression 返回根的值

int ParPtrTree::FIND(int curr) const {

if (array[curr] == ROOT) return curr; // At root已经是根则返回

array[curr] = FIND(array[curr]);

return array[curr];

}

1. **manager.cpp(主函数 测试文件)：**

#include<iostream>

#include<cstring>

#include "graphm.h"

#include "graph.h"

#include "create.h"

#include "heap.h"

#define ROOT -1

#include "uf.h"

#define INFINITY 9999

using namespace std;

void AddEdgetoMST(Graph<string>\* G,int v1, int v2) {

cout << "增加边 " << G->getVex(v1) << " <--> " << G->getVex(v2) << ":"<<G->weight(v1,v2)<<endl;

}

// Simple class to store data in the heap: edge and its distance 将边和权值存入堆的类

class KruskElem { // An element for the heap堆的元素

public:

int from, to, distance; // The edge being stored 存边要用的值

KruskElem() { from = -1; to = -1; distance = -1; }

KruskElem(int f, int t, int d)

{ from = f; to = t; distance = d; }

};

// Comparator to compare two KruskElem's比较器：比较两个堆元素

class Comp {

public:

static bool prior(KruskElem x, KruskElem y)

{ return x.distance < y.distance; }

};

void Kruskel(Graph<string>\* G) { // Kruskal's MST algorithm构造最小生成树算法

ParPtrTree A(G->n()); // Equivalence class array等价类数组

KruskElem E[G->e()]; // Array of edges for min-heap最小堆的边的数组

int i;

int edgecnt = 0;

for (i=0; i<G->n(); i++) // Put the edges on the array添加边到数组中

for (int w=G->first(i); w<G->n(); w = G->next(i,w)) {

if(w>i){//无向图 保证A->B B->A值存一次

E[edgecnt].distance = G->weight(i, w);

E[edgecnt].from = i;

E[edgecnt++].to = w;

}

}

// Heapify the edges

heap<KruskElem, Comp> H(E, edgecnt, edgecnt);

int numMST = G->n(); // Initially n equiv classes初始化n个等价类

for (i=0; numMST>1; i++) { // Combine equiv classes合并等价类

KruskElem temp;

temp = H.removefirst(); // Get next cheapest edge取下一条最短边

int v = temp.from; int u = temp.to;

if (A.differ(v, u)) { // If in different equiv classes如果在不同等价类里

A.UNION(v, u); // Combine equiv classes合并等价类

AddEdgetoMST(G,temp.from, temp.to); // Add edge to MST在MST里添加边

numMST--; // One less MST等价类数目减一

}

}

}

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);

cout<<"--------------------------------\n"

<<"调用Kruskal求解最小生成树：\n";

Kruskel(G);

cout<<"MST构建完成！\n";

return 0;

}

1. **test.gph(输入文件)：**

# 第三版课本P262 图11.20

6 # Number of vertices

A B C D E F # 输入时每个顶点信息之间要以空格分割

U # Undirected graph

8 # 边数

A C 7

A E 9

B C 5

B F 6

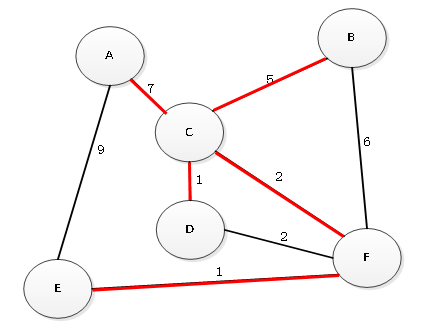
C D 1

C F 2

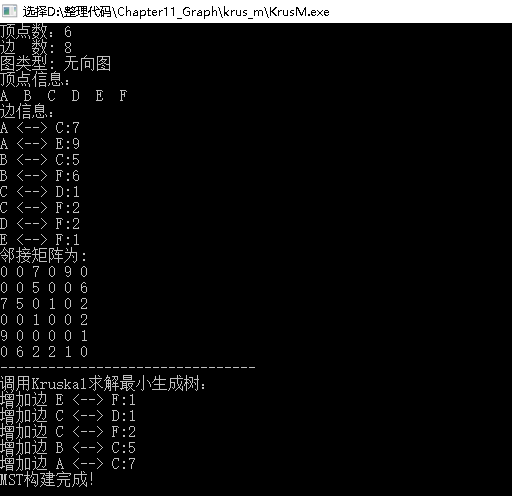
D F 2

E F 1

**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