《数据结构》实验报告

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **年级、专业、班级** | | **2018级物联网工程专业1班** | | | **姓名** | **邓露** |
| **实验题目** | **图的操作** | | | | | |
| **实验时间** | **2019.12.02** | | **实验地点** | **DS1422** | | |
| **实验成绩** |  | | **实验性质** | **□验证性** √**设计性 □综合性** | | |
| 教师评价：  □算法/实验过程正确； □源程序/实验内容提交 □程序结构/实验步骤合理；  □实验结果正确； □语法、语义正确； □报告规范；  其他：  评价教师签名： | | | | | | |
| 一、实验目的  1. 掌握图的基本概念，描述方法；遍历方法。  2. 掌握图的最短路径和最小生成树算法。 | | | | | | |
| 二、实验项目内容   1. 图的存储结构使用邻接矩阵。 2. 创建图操作类，支持BFS遍历、DFS遍历、求单源最短路径、求最小生成树、判断是否存在回路等五个功能，这些功能封装成图操作类的成员函数。   3． 输入图的节点数n（不超过10个）、边数m，节点分别用0到n-1表示。  4． 采用“起始节点，终止节点，权值”输入图的m条边，创建图。  5． 输出从节点0开始的BFS遍历、DFS遍历节点遍历顺序。  6． 输出从节点0到其余节点的最短路径及最短路径长度，如果没有路经，输出0。  7．输出图的最小生成树包含的边，边用“起始节点，终止节点，权值”表示。  注意：最后提交完整的实验报告和源程序，主程序里应对实验所要求的功能进行充分测试。 | | | | | | |
| 三、实验过程或算法（源程序）  #include<iostream>  #include<algorithm>  #include<stdio.h>  #include<math.h>  #include <vector>  #include<queue>  #include<stack>  const int INF = 9999;  using namespace std;  typedef pair<int,int> P;  struct edge { //存邻边，两点及距离  int u,v,cost;  };  bool cmp(edge& a,edge& b) {  return a.cost < b.cost;  }  class Graphm{  private:  int numvertex,numedge,flag = 0;  //the number of vertex, edge.  //flag 用于消除打印第一个元素前的空格  int \*\*matrix; //pointer to adjacency matrix  int \*mark,\*rank,\*par;  edge \*es;  int loop = 0;  public:  Graphm(int numvert,int numedge) {  init(numvert,numedge);  }  ~Graphm() {  delete []mark;  for(int i=1; i<=numvertex; i++)  delete []matrix[i];  delete []matrix;  delete []es;  delete []rank;  delete []par;  }  void init(int n,int m) {  numvertex = n;  numedge = 0;  mark = new int [n+1];  es = new edge [m+2];  for(int i=0;i<m+2;i++){  es[i].cost = 0;  es[i].u = 0;  es[i].v = 0;  }  par = new int [n+1];  rank = new int [n+1];  for(int i=1; i<=numvertex; i++)  mark[i] = 0;  matrix = new int\*[numvertex+1];  for(int i=1; i<=numvertex; i++)  matrix[i] = new int[numvertex+1];  for(int i=1; i<=numvertex; i++)  for(int j=1; j<=numvertex; j++)  matrix[i][j] = INF;  }  void setes(int s,int from,int to,int weight){  es[s].u = from;//将参数中的边的信息存入存储边的一维数组  es[s].v = to;  es[s].cost = weight;  }  void setegde(int v1,int v2,int w) {//将边的信息输入邻接矩阵中  if(matrix[v1][v2] == INF) numedge++;  matrix[v1][v2] = w;  matrix[v2][v1] = w;  }  int first(int v) {//返回的是与v节点邻接的第一个节点  for(int i=1; i<=numvertex; i++)  if(matrix[v][i] != INF) return i;  return numvertex+1;  }  int next(int v,int w) {//以w作为与v邻接的上一个节点，  //寻找与v邻接的下一个节点  for(int i=w+1; i<=numvertex; i++)  if(matrix[v][i] != INF )  return i;  return numvertex+1;  }  void reset(){//reset函数的作用是在每次执行dfs或bfs函数时将mark数组中的值重置，也就是将所有点都标记为未访问  flag = 0;  for(int i=1;i<=numvertex;i++)  mark[i] = 0;  }  void dfs(int v) {  if(flag == 0) flag = 1;//flag用于消除打印第一个元素前的空格  else cout<<" ";//对于该节点的previsit  cout<<v;  mark[v] = 1;//将该节点标记为已访问  for(int w = first(v); w<=numvertex; w = next(v,w)) {  if(mark[w] == 0)  dfs(w);  }  }  void bfs(int start) {  int v,w;  queue<int> que;  que.push(start);//初始化队列并将起点压入队列中  mark[start] = 1;//将起点标记为已访问  while(!que.empty()) {//当队列中有元素时便不断执行操作  v = que.front();  que.pop();//弹出队头元素  if(flag == 0) flag = 1;  else cout<<" ";  cout<<v;// 对于队头元素进行previsit  for(w = first(v); w <= numvertex; w = next(v,w)) {  if(mark[w] == 0) { //当前v节点的各个邻接点也需要被访问  mark[w] = 1;//所以便将他们标记为已访问并且都压入队列  que.push(w);  }  }  }  }  void dijkstra(int start,int end) {  int dis[numvertex+1],prev[numvertex+1];  stack<int> stk;  fill(dis+1,dis+1+numvertex,INF);  cout<<endl;  fill(prev,prev + numvertex+1,-1);  dis[start] = 0;  priority\_queue<P,vector<P>,greater<P> > q;  //P(min\_distance, vertice)  q.push(P(0,start));//建立优先队列并把到起点的距离为0和起点1压入优先队列  while(!q.empty()) {  P temp = q.top();//取出优先队列的队头元素置为v点  q.pop();  int v = temp.second;//  int d = temp.first;//起点到这个v点已经确定了的最小距离为d  for(int i=1; i<=numvertex; i++) {  int e = matrix[v][i];//遍历现在的这个v点的邻接点  if( e + d < dis[i]) { //如果从起点经v点再到i点的距离  dis[i] = e + d; //比现在dis数组中存的从起点到i点  //的距离还要小的话，那就可以更新dis[i]  prev[i] = v;//更新过后记录i点的前导节点为v  q.push(P((e+d),i));//然后还要再把更新后的P(e+d,i)放入队列中。  }  }  }  for(int i=end; ;i = prev[i]){//由前面记录的前导节点数组  if(i == -1) break; //将反过来的路径依次压入栈  stk.push(i);  }  while(!stk.empty()){  cout<<stk.top()<<" "; //将栈中元素反向输出  stk.pop();  }  cout<<dis[end]; //最后输出起点到终点的最短路径距离  }  //下面的initial find unite 和same 都是并查集的初始化  //为了在kruskal算法中插边时判断两个点是否属于同一棵最小生成树  void initial(int n) {  for(int i=1; i<=n; i++) {  rank[i] = 0;  par[i] = i;  }  }  int find(int a) {  if(par[a] == a) return a;  else return par[a] = find(par[a]);  }  void unite(int a,int b) {  int ar = find(a);  int br = find(b);  if (ar == br)return;  if(rank[ar] < rank[br])  par[ar] = br;  else {  par[br] = ar;  if(rank[ar] == rank[br])rank[ar]++;  }  }  bool same(int a,int b) {  return find(a) == find(b);  }  void kruskal() {  initial(numedge);//并查集的初始化  int count = 0;  int use[numedge+1];  fill(use+1,use+1+numedge,0);//将这些边都标记为未使用  sort(es+1,es+1+numedge,cmp);//将这些边按照权重从小到大排序  for(int i=1; i<=numedge; i++) {  if(count >= numvertex-1) break;//插满numvertex-1条边即可跳出循环  edge e = es[i];  if(!same(e.u,e.v)) { // 若两个点不在同一棵树上则插入这条边  unite(e.u,e.v); //将两个点标记为在同一棵树上  use[i] = 1; //将这条边标记为已使用  count++; //边数+1，扩边完成  }  }  for(int i=1;i<=numedge;i++){  if(use[i] == 1)  cout<<es[i].u<<" "<<es[i].v<<" "<<es[i].cost<<endl;  }//将生成的最小生成树的边打印出来  }  void existloop(int v) { //以dfs遍历为基础来执行判环操作  mark[v] = 1;  for(int w = first(v); w<=numvertex; w = next(v,w)) {  if(mark[w] == 0)  existloop(w);  else if(mark[w] == 1) //如果在dfs过程中判断当前节点的下一个  //邻接点已经被访问过，则证明图中有环  loop = 1; //将图数据类型中的loop变量置为1  }  return;  }  void showloop(){//判断图是否存在环的变量loop的访问接口  if(loop == 1) cout<<"YES"<<endl;//有环则输出YES，  else cout<<"NO"<<endl;//无环则输出NO。  }  };  int main()  {  int m,n;  scanf("%d %d",&n,&m);  Graphm a1(n,m); //实例化一个图对象a1  for(int i=1; i<=m; i++) { //循环m次插入m条边  int from,to,weight;  scanf("%d %d %d",&from,&to,&weight);  edge temp;  temp.u = from;  temp.v = to;  temp.cost = weight;  a1.setegde(from,to,weight);  a1.setes(i,from,to,weight);  }  a1.reset();  a1.bfs(1);  cout<<endl;  a1.reset();  a1.dfs(1);  for(int i=2;i<=n;i++){  a1.dijkstra(1,i);  }  cout<<endl;  a1.kruskal();  a1.existloop(1);  a1.showloop();  return 0;  } | | | | | | |
| 四、实验结果及分析和（或）源程序调试过程  输入：  6 9  0 1 10  1 4 20  1 6 2  2 3 3  2 4 5  3 5 15  4 5 11  4 6 10  5 6 3  输出： | | | | | | |

OJ:

