**实验报告**

**学院（系）名称：**计算机科学与工程学院

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **姓名** | | 王帆 | | **学号** | | 20152180 | | | **专业** | | 计算机科学与技术 | |
| **班级** | | 15级1班 | | **实验项目** | | 实验四：图的遍历 | | | | | | |
| **课程名称** | | | | 数据结构与算法 | | | | | **课程代码** | | 0668016 | |
| **实验时间** | | | | 5 月 12 日 第 3~4 节 | | | | | **实验地点** | | 7-219 | |
| 考核标准 | 实验过程  25分 | | 程序运行  20分 | | 回答问题  15分 | | 实验报告  30分 | 特色  功能  5分 | | 考勤违纪情况  5分 | **成绩** |  |
| 成绩栏 |  | |  | |  | |  |  | |  | 其它批改意见:  教师签字： | |
| 考核内容 | 评价在实验课堂中的表现，包括实验态度、编写程序过程等内容等。 | | □功能完善,  □功能不全  □有小错  □无法运行 | | ○正确  ○基本正确  ○有提示  ○无法回答 | | ○完整  ○较完整  ○一般  ○内容极少  ○无报告 | ○有  ○无 | | ○有  ○无 |
|  | | | | | | | | | | | | |
| **一、实验目的**  通过实验使学生理解图的主要存储结构，掌握图的构造算法、图的深度优先和广度优先遍历算法。  **二、实验题目与要求**  每位同学从下面题目中至少选择2题实现：其中第1题必做，2-3题中至少选择1题。  1.输入指定的边数和顶点数建立图，并输出深度优先遍历和广度优先遍历的结果。  1）问题描述：在主程序中设计一个简单的菜单，分别调用相应的函数功能：  1 图的建立  2…深度优先遍历图  3…广度优先遍历图  0…结束  2）实验要求：在程序中定义下述函数，并实现要求的函数功能：  CreateGraph(): 按从键盘的数据建立图  DFSGrahp()：深度优先遍历图  BFSGrahp()：广度优先遍历图  3）实验提示:   图的存储可采用邻接表或邻接矩阵；   图存储数据类型定义 （邻接表存储）  # define MAX\_VERTEX\_NUM 8 //顶点最大个数  typedef struct ArcNode  { int adjvex;  struct ArcNode \*nextarc;  int weight; //边的权  }ArcNode; //表结点  # define VertexType int //顶点元素类型  typedef struct VNode  { int degree,indegree; //顶点的度，入度  VertexType data;  ArcNode \*firstarc;  }Vnode /\*头结点\*/, AdjList[MAX\_VERTEX\_NUM];  typedef struct{  AdjList vertices;  int vexnum,arcnum;//顶点的实际数，边的实际数  }ALGraph;  4）注意问题:   注意理解各算法实现时所采用的存储结构。   注意区别正、逆邻接。  2. 给出一个图的结构，输出其拓扑排序序列（顶点序列用空格隔开），要求在同等条件下，编号小的顶点在前。  3．利用最小生成树算法解决通信网的总造价最低问题  1）问题描述：若在n个城市之间建通信网络，架设n-1条线路即可。如何以最低的经济代价建设这个通信网，是一个网络的最小生成树问题。  2）实验要求：利用Prim算法求网的最小生成树。  3) 实现提示：通信线路一旦建立，必然是双向的。因此，构造最小生成树的网一定是无向网。为简单起见，图的顶点数不超过10个，网中边的权值设置成小于100。   1. 实验过程与实验结果   oj平台实验完成情况：     1. 题目一  * 数据结构定义   #define MAX\_VERTEX\_NUM 100 //最大顶点数  enum BOOLEAN{FALSE = 0,TRUE} visited[MAX\_VERTEX\_NUM];  typedef int VertexType;  typedef struct ArcNode{ //网络多一个weigth域  int adjvex; /\*该边或弧所指向的顶点的下标\*/  struct ArcNode \*nextarc; //指向下一个边节点的下标  }ArcNode;  typedef struct VNode{ //顶点节点  VertexType data; //顶点信息  ArcNode \*firstarc; //指向第一个依附顶点的边节点的指针  }VNode,AdjList;  typedef struct {  AdjList vertex[MAX\_VERTEX\_NUM]; //邻接表  int vernum,arcnum; //顶点数，边或弧数  }Graph;   * 算法设计思路简介   使用邻接链表的方式创建无向图。深度遍历时从起始顶点开始访问它的后继节点，接着访问该后继节点所指的顶点，每次访问过的顶点做一个标记，防止重复打印。整个过程利用递归实现，结束条件为访问完起始顶点所在的链表。广度遍历也从起始节点开始依次访问它的每一个后继节点，将访问过的顶点入队列，并做标记，保证访问时的优先性，当完成初始顶点所在链表的访问时，访问队头元素所指顶点的链表，结束条件为队列为空。至此，遍历完整个无向图。   * 算法描述：   C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3. 【Experiment3-3】二叉树层次遍历.png  深度遍历  C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3. 【Experiment3-3】二叉树层次遍历 (1).png  广度遍历     * 算法的实现和测试结果   深度遍历  void DFSTraverse(Graph \*G){ //对图G做深度游先搜索  int v;  for(v = 0;v < G->vernum;v++){  visited[v] = FALSE; //初始化访问标志数组visited  }  for(v = 0;v < G->vernum;v++){  if(!visited[v]){  DFS(G,v,1); //对尚未访问的顶点 v 调用DFS  }  }  }  void DFS(Graph \*G,int v,int n){ //从顶点 v 出发对 v 所在的连通分量进行深度优先搜索  visited[v] = TRUE;  Visit(G,v,n); //访问顶点 v  int w;  for(w = FirstAdjVex(G,v);w > -1;w = NextAdjVex(G,v,w)){ //对 v 的未访问的邻接点 w 递归调用 DFS  if(!visited[w]){  DFS(G,w,n+1);  }  }  }    广度遍历  void BFSTraverse(Graph \*G){  int w,u,n = 1;  LinkQueue \*Q;  Q = InitQueue();  for(int v = 0;v < G->vernum;++v){  visited[v] = FALSE; //初始化访问标志数组visited  }  for(int v = 0;v < G->vernum;++v){  if(!visited[v]){  visited[v] = TRUE;  Visit(G,v,n++);  EnQueue(Q,v); //顶点节点入队列  while(!QueueEmpty(Q)){ //当队列不为空，即还有未访问的节点时  u = DeQueue(Q);  for(w = FirstAdjVex(G,u);w > -1;w = NextAdjVex(G,u,w)){  if(!visited[w]){  visited[w] = TRUE;  Visit(G,w,n++);  EnQueue(Q,w); //w 的未访问过的邻接节点入队列  }  }  }  }  }  }          邻接表    输出结果  题目二   * 数据结构定义   #define INFINIITY INT\_MAX //最大值  #define MAX\_VERTEX\_NUM 30 //最大顶点数  #define TRUE 1  #define FALSE 0  int stack[MAX\_VERTEX\_NUM];  int top = 0;  typedef int VertexType; //数据类型  typedef struct ArcNode {  int adjvex; //该边或弧所指向的顶点的下标  struct ArcNode \*nextarc; //指向下一个边节点的下标  }ArcNode;  typedef struct VNode { //顶点节点  VertexType data; //顶点信息  ArcNode \*firstarc; //指向第一个依附顶点的边节点的指针  }VNode,AdjList;  typedef struct {  AdjList vertices[MAX\_VERTEX\_NUM]; //邻接表  int vexnum,arcnum; //顶点数，边或弧数  }Graph;   * 算法设计思路简介   首次对各顶求入度，按顺序访问顶点 对入度为 0 的顶点打印并将关联的顶点入度 – 1,一直循环直到打印完所有入度为0的顶点。  FindInDegree (G,indegree); 求所有顶点的入度  TopologicalSort (Graph \*G) 对入度为 0 的顶点打印并修改入度  for(j = 1;j <= G->vexnum;j++) {  if(indegree[j] == 0) { //如果入度为 0 打印  Visit(G,j,count);  indegree[j] = -1;  count++;  for(p = G->vertices[j].firstarc; p ;p = p->nextarc){  --indegree[p->adjvex]; //关联的顶点入度减 1 ,}  j = 0;}  }   * 算法描述：   C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3. 【Experiment3-3】二叉树层次遍历 (2).png   * 算法的实现和测试结果：   void FindInDegree (Graph \*G,int indegree[]) { //求顶点的入度  int i;  ArcNode \*p;  for(i = 1;i <= G->vexnum; i++){  indegree[i] = 0; //赋初值  }  for(i = 1;i <= G->vexnum; i++){  p = G->vertices[i].firstarc;  while(p){  indegree[p->adjvex]++;  p = p->nextarc;  }  }  }  int TopologicalSort (Graph \*G){  int j,count,indegree[MAX\_VERTEX\_NUM];  ArcNode \*p;  //对各顶点求入度  FindInDegree (G,indegree);  //按顺序访问顶点 对入度为 0 的顶点打印并将关联的顶点入度 - 1  count = 0;    for(j = 1;j <= G->vexnum;j++) {  if(indegree[j] == 0) { //如果入度为 0 打印  Visit(G,j,count);  indegree[j] = -1;  count++;  for(p = G->vertices[j].firstarc; p ;p = p->nextarc){  --indegree[p->adjvex]; //关联的顶点入度减 1 ,  }  j = 0;  }  }  if(count < G->vexnum) {  printf("此有向图有回路\n");  return 0;  }  else {  printf("为一个拓扑排序\n");  return 1;  }  }    图示：        输出结果    题目三：   * 数据结构定义：   #define VertexNum 30  #define INFINITY 1000  typedef struct  {  char vertex[VertexNum]; //顶点表  int AdjMatrix[VertexNum][VertexNum]; //邻接矩阵,可看做边表  int n,e; //图中当前的顶点数和边数  }Graph;    typedef struct arc  {  int u; //边的起始顶点  int v; //边的终止顶点  int weigth; //边的权值  }arc;   * 算法设计思路简介:   这里使用的是Kruskal算法，最初先构造一个只有n个顶点，没有边的非连通图，图中每个顶点自成一个连通分量。当选到一条具有最小权值的边时，若该边的两个顶点分别落在不同的两个连通分量上，则将此边加入到集合T中；否则将此边舍去，重新选择一条满足上述条件的权值最小边。如此重复，直到所有顶点在同一连通分量上为之。   * 算法描述：   C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3. 【Experiment3-3】二叉树层次遍历 (3)1.png  C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\3. 【Experiment3-3】二叉树层次遍历 (3)2.png   * 算法的实现和测试结果：   void heapsort(arc E[],int k){  arc temp;  for(int i = 0;i < k - 1;i++){  for(int j = 0;j < k - i -1;j++){  if(E[j].weigth > E[j+1].weigth){  temp = E[j];  E[j] = E[j+1];  E[j+1] = temp;  }  }  }  void kruskal(Graph G)  {  int i,j,u1,v1,sn1,sn2,k,min;  int vset[VertexNum]; //辅助数组，判定两个顶点是否连通  arc E[G.e]; //存放所有的边  k=0; //E数组的下标从0开始  //让边进数组  for (i=1;i<=G.n;i++) {  for (j=1;j<=G.n;j++)  {  if (G.AdjMatrix[i][j]!=0 && G.AdjMatrix[i][j]!=INFINITY)  {  E[k].u=i;  E[k].v=j;  E[k].weigth=G.AdjMatrix[i][j];  k++;  }  }  }    heapsort(E,k); //排序，按权值从小到大排列    for (i=1;i <= G.n;i++) //初始化辅助数组  {  vset[i]=i;  }  k=0; //生成的边数，最后要刚好为总边数  j=0; //E中的下标  min = 0; // 最小生成树的权值和  while (k < G.n - 1 && j < G.e)  {  sn1=vset[E[j].u];  sn2=vset[E[j].v]; //得到两顶点属于的集合编号  if (sn1 != sn2) //不在同一集合编号内的话，把边加入最小生成树  {  min += E[j].weigth;  k++;  for (i=1;i <= G.n;i++)  {  if (vset[i] == sn2)  {  vset[i] = sn1;  }  }  }  j++;  }  printf("%d",min);  }  图示：  C:\Users\mjt89\AppData\Local\Microsoft\Windows\INetCache\Content.Word\无标题.png    输出结果     1. **收获与体会**   通过本实验，理解了图的定义，掌握了图的存储思想及其存储实现。在以上实验操作中，实现了邻接表的建立和邻接表的输出，邻接矩阵的建立和输出。拓扑排序的思想和实现，最小生成树的实际应用等。但是，还有一些知识不是很理解，在操作中出现一些问题，解决问题不是很熟练，在接下来的学习中，将多加练习和钻研，实现深刻掌握图的遍历思想、图的应用的编程实现。   1. **源代码清单**   **题目一：**  #include <stdio.h>  #include <stdlib.h>  #define MAX\_VERTEX\_NUM 100 //最大顶点数  //图的种类{有向图，有向网络，无向图AG，无向网络}  enum BOOLEAN{FALSE = 0,TRUE} visited[MAX\_VERTEX\_NUM];  typedef int VertexType;  typedef struct ArcNode{ //网络多一个weigth域  int adjvex; /\*该边或弧所指向的顶点的下标\*/  struct ArcNode \*nextarc; //指向下一个边节点的下标  }ArcNode;  typedef struct VNode{ //顶点节点  VertexType data; //顶点信息  ArcNode \*firstarc; //指向第一个依附顶点的边节点的指针  }VNode,AdjList;  typedef struct {  AdjList vertex[MAX\_VERTEX\_NUM]; //邻接表  int vernum,arcnum; //顶点数，边或弧数  }Graph;  typedef struct QNode{ //队列节点  int data;  struct QNode \*next;  }QueueNode;  typedef struct LinkQueue{ //队头队尾节点  QueueNode \*front;  QueueNode \*rear;  }LinkQueue;  LinkQueue \*InitQueue(){  LinkQueue \*Q = (LinkQueue \*)malloc(sizeof(LinkQueue));  QueueNode \*p = (QueueNode \*)malloc(sizeof(QueueNode));  if(p){  Q->front = Q->rear = p;  Q->front->next = NULL;  return Q;  }  else {  printf("创建队列失败！"); exit(0);  }  }  int QueueEmpty(LinkQueue \*Q){  return (Q->front == Q->rear) ? 1 : 0;  }  LinkQueue \*EnQueue(LinkQueue \*Q,int x){  QueueNode \*s = (QueueNode \*)malloc(sizeof(QueueNode));  if(s){  s->data = x;  Q->rear->next = s;  s->next = NULL;  Q->rear->next = s;  Q->rear = s;  }  else {  printf("创建队列节点失败！"); exit(0);  }  return Q;  }  int DeQueue(LinkQueue \*Q){  QueueNode \*p;  int e;  if(!QueueEmpty(Q)){  p = Q->front->next; //指向队头节点  e = p->data;  Q->front->next = p->next;  if(p == Q->rear){  Q->rear = Q->front;  }  free(p);  }  return e;  }  /\*创建无向图的邻接表\*/  Graph \*CreateAG(Graph \*G,int n,int m) {  int i,j,a,b;  G = (Graph \*)malloc(sizeof(Graph));  G->vernum = n;  G->arcnum = m;  for(i = 0;i < G->vernum;i++){ //初始化顶点节点，这里 data 为 1,2,3.....  G->vertex[i].data = i + 1;  G->vertex[i].firstarc = NULL;  }  for(i = 0;i < G->arcnum;i++){  scanf("%d%d",&a,&b); //输入边(a,b)  ArcNode \*s;  a--;  b--;  s = (ArcNode \*)malloc(sizeof(ArcNode));  s->adjvex = b;  s->nextarc = NULL; //链入顶点 a 对应的边节点  s->nextarc = G->vertex[a].firstarc;  G->vertex[a].firstarc = s;  s = (ArcNode \*)malloc(sizeof(ArcNode));  s->adjvex = a;  s->nextarc = NULL; //链入顶点 b 对应的边节点  s->nextarc = G->vertex[b].firstarc;  G->vertex[b].firstarc = s;  }    return G;  }  int FirstAdjVex(Graph \*G,int v){  if(G->vertex[v].firstarc)  return G->vertex[v].firstarc->adjvex;  else  return -1;  }  int NextAdjVex(Graph \*G,int v,int w){  ArcNode \*p;  if(G->vertex[v].firstarc==NULL) {  return -1;  }  else {  p = G->vertex[v].firstarc;  while(p->adjvex !=w ){  p = p->nextarc;  }    if(p->nextarc == NULL){  return -1;  }  else {  return p->nextarc->adjvex;  }  }  }  void Visit(Graph \*G,int u,int n){  if(n < G->vernum){  printf("%d ",G->vertex[u].data);  }  else {  printf("%d",G->vertex[u].data);  }  }  void DFS(Graph \*G,int v,int n){ //从顶点 v 出发对 v 所在的连通分量进行深度优先搜索    visited[v] = TRUE;    Visit(G,v,n); //访问顶点 v    int w;  for(w = FirstAdjVex(G,v);w > -1;w = NextAdjVex(G,v,w)){ //对 v 的未访问的邻接点 w 递归调用 DFS  if(!visited[w]){  DFS(G,w,n+1);  }  }  }  void test(Graph \*G){  for(int v = 0;v < G->vernum;v++){  printf("%d :",G->vertex[v].data);  ArcNode \*p = G->vertex[v].firstarc;  while(p){  printf("%d -- ",p->adjvex+1);  p = p->nextarc;  }  printf("null\n");  }  }  void DFSTraverse(Graph \*G){ //对图G做深度游先搜索  int v;  for(v = 0;v < G->vernum;v++){  visited[v] = FALSE; //初始化访问标志数组visited  }  for(v = 0;v < G->vernum;v++){  if(!visited[v]){  DFS(G,v,1); //对尚未访问的顶点 v 调用DFS  }  }  }  void BFSTraverse(Graph \*G){  int w,u,n = 1;  LinkQueue \*Q;  Q = InitQueue();  for(int v = 0;v < G->vernum;++v){  visited[v] = FALSE; //初始化访问标志数组visited  }  for(int v = 0;v < G->vernum;++v){  if(!visited[v]){  visited[v] = TRUE;  Visit(G,v,n++);  EnQueue(Q,v); //顶点节点入队列  while(!QueueEmpty(Q)){ //当队列不为空，即还有未访问的节点时  u = DeQueue(Q);  for(w = FirstAdjVex(G,u);w > -1;w = NextAdjVex(G,u,w)){  if(!visited[w]){  visited[w] = TRUE;  Visit(G,w,n++);  EnQueue(Q,w); //w 的未访问过的邻接节点入队列  }  }  }  }  }  }  int main(){  int n,m;  Graph \*G;  scanf("%d",&n);  scanf("%d",&m);  G = CreateAG(G,n,m);  DFSTraverse(G);  printf("\n");  BFSTraverse(G);  return 0;  }  **题目二：**  #include <stdio.h>  #include <stdlib.h>  #define INFINIITY INT\_MAX //最大值  #define MAX\_VERTEX\_NUM 30 //最大顶点数  #define TRUE 1  #define FALSE 0  int stack[MAX\_VERTEX\_NUM];  int top = 0;  typedef int VertexType; //数据类型  typedef struct ArcNode {  int adjvex; //该边或弧所指向的顶点的下标  struct ArcNode \*nextarc; //指向下一个边节点的下标  }ArcNode;  typedef struct VNode { //顶点节点  VertexType data; //顶点信息  ArcNode \*firstarc; //指向第一个依附顶点的边节点的指针  }VNode,AdjList;  typedef struct {  AdjList vertices[MAX\_VERTEX\_NUM]; //邻接表  int vexnum,arcnum; //顶点数，边或弧数  }Graph;  /\*创建无向图的邻接表\*/  Graph \*CreateAG(Graph \*G,int n,int m) {  int i,j,a,b;  G = (Graph \*)malloc(sizeof(Graph));  G->vexnum = n;  G->arcnum = m;  for(i = 1;i <= G->vexnum;i++){  G->vertices[i].data = i;  G->vertices[i].firstarc = NULL;  }  for(i = 1;i <= G->arcnum;i++){  scanf("%d %d",&a,&b); //输入边（a,b）、权值w和info所代表的信息  ArcNode \*s;  s = (ArcNode \*)malloc(sizeof(ArcNode));  s->adjvex = b;  s->nextarc = NULL; //链入顶点 a 对应的边节点  s->nextarc = G->vertices[a].firstarc;  G->vertices[a].firstarc = s;  }  return G;  }    void FindInDegree (Graph \*G,int indegree[]) { //求顶点的入度  int i;  ArcNode \*p;  for(i = 1;i <= G->vexnum; i++){  indegree[i] = 0; //赋初值  }  for(i = 1;i <= G->vexnum; i++){  p = G->vertices[i].firstarc;  while(p){  indegree[p->adjvex]++;  p = p->nextarc;  }  }  /\*test  for(i = 1;i <= G->vexnum; i++){  printf("%d的入度 = %d\n",i,indegree[i]);  } end\*/  }  void Visit(Graph \*G,int v,int N){  if(N == 0){  printf("%d",G->vertices[v].data);  }  else {  printf(" %d",G->vertices[v].data);  }  }  int TopologicalSort (Graph \*G){  int j,count,indegree[MAX\_VERTEX\_NUM];  ArcNode \*p;  //对各顶点求入度  FindInDegree (G,indegree);  //按顺序访问顶点 对入度为 0 的顶点打印并将关联的顶点入度 - 1  count = 0;    for(j = 1;j <= G->vexnum;j++) {  if(indegree[j] == 0) { //如果入度为 0 打印  Visit(G,j,count);  indegree[j] = -1;  count++;  for(p = G->vertices[j].firstarc; p ;p = p->nextarc){  --indegree[p->adjvex]; //关联的顶点入度减 1 ,  }  j = 0;  }  }    if(count < G->vexnum) {  printf("此有向图有回路\n");  return 0;  }  else {  printf("为一个拓扑排序\n");  return 1;  }    }  int main(){  Graph \*G;  int n,m;  int indegree[MAX\_VERTEX\_NUM];  scanf("%d%d",&n,&m);  G = CreateAG(G,n,m);  TopologicalSort (G);  return 0;  }  **题目三：**  #include <stdio.h>  #define VertexNum 30  #define INFINITY 1000  typedef struct  {  char vertex[VertexNum]; //顶点表  int AdjMatrix[VertexNum][VertexNum]; //邻接矩阵,可看做边表  int n,e; //图中当前的顶点数和边数  }Graph;    typedef struct arc  {  int u; //边的起始顶点  int v; //边的终止顶点  int weigth; //边的权值  }arc;  Graph Initgraph(Graph G){  int i,j,w;  scanf("%d",&G.n);  scanf("%d",&G.e);  for(i = 1;i <= G.n;i++){  for(j = 1;j <= G.n;j++){  if(i != j){ G.AdjMatrix[i][j] = INFINITY;  }  else {  G.AdjMatrix[i][j] = 0;  }  }  }  for(int k = 1;k <= G.e;k++){  scanf("%d",&i);  scanf("%d",&j);  scanf("%d",&w);  if(w < G.AdjMatrix[i][j]){  G.AdjMatrix[i][j] = w;  }  }  return G;  }  void heapsort(arc E[],int k){  arc temp;  for(int i = 0;i < k - 1;i++){  for(int j = 0;j < k - i -1;j++){  if(E[j].weigth > E[j+1].weigth){  temp = E[j];  E[j] = E[j+1];  E[j+1] = temp;  }  }  }  void kruskal(Graph G)  {  int i,j,u1,v1,sn1,sn2,k,min;  int vset[VertexNum]; //辅助数组，判定两个顶点是否连通  arc E[G.e]; //存放所有的边  k=0; //E数组的下标从0开始  for (i=1;i<=G.n;i++) //让边进数组  {  for (j=1;j<=G.n;j++) {  if (G.AdjMatrix[i][j]!=0 && G.AdjMatrix[i][j]!=INFINITY)  {  E[k].u=i;  E[k].v=j;  E[k].weigth=G.AdjMatrix[i][j];  k++;  }  }  }  heapsort(E,k); //排序，按权值从小到大排列  for (i=1;i <= G.n;i++) //初始化辅助数组  {  vset[i]=i;  }  k=0; //生成的边数，最后要刚好为总边数  j=0; //E中的下标  min = 0; // 最小生成树的权值和  while (k < G.n - 1 && j < G.e)  {  sn1=vset[E[j].u];  sn2=vset[E[j].v]; //得到两顶点属于的集合编号  if (sn1 != sn2) //不在同一集合编号内的话，把边加入最小生成树  {  min += E[j].weigth;  k++;  for (i=1;i <= G.n;i++)  {  if (vset[i] == sn2) {  vset[i] = sn1;  }  }  }  j++;  }  printf("%d",min);  }  int main(){  Graph G;  G = Initgraph(G);  kruskal(G);  return 0;  } | | | | | | | | | | | | |