**《数据结构实验》报告**

|  |  |  |
| --- | --- | --- |
| **第 七 次实验** | **日期：2021.12.25** | **得分：** |
| **学号：** | **姓名：*Steven*** | **专业：智能科学与技术** |

**一、实验目的**

1. 练习C语言函数的使用方法。

2. 练习使用malloc、realloc、free等函数动态分配内存的方法。

3. 练习指针变量的使用，了解其用法。

4. 理解并熟练结构体的使用方法。

5. 理解并学会邻接矩阵、邻接表的构建和操作

6. 理解并学会图的遍历方法

**二、实验内容**

1 、假设图中数据元素类型是字符型， 请采用邻接矩阵或邻接表实现图的以下基本操作：

(1) 构造图（包括有向图、有向网、无向图、无向网）

(2) 根据深度优先遍历图；

(3) 根据广度优先遍历图。

2、给定一个无向图及两种颜色，请判定能否为这个无向图的相邻顶点着不同颜色。

3、给定一个无向图及两顶点， 请判定这两个顶点之间是否有路径存在。

4、给定若干村落及村落间可能建设成公路的若干道路成本， 请计算使每个村落都有公路连通的方案所需最低成本。

**三、数据结构及算法设计**

1、

本质上就是按照输入的边集和图的类型，为邻接矩阵赋值。

2、

首先根据深度优先遍历，按照下一个点的颜色和上一个点的颜色不同的规则涂色。然后遍历邻接矩阵，即可得到某点的所有邻接点，一旦出现邻接点和当前点相同颜色的情况，就说明无法按照这种规则完成涂色；如果遍历完成了还没有这种情况，就说明这种涂色是合理的。

3、

也是利用的深度优先遍历的思想，不过在深度优先遍历的时候由于考虑到图非连通图的情况，是多起点地反复深度优先遍历；而本题中只用起始点深度优先遍历即可，若遍历过程中确实没有遍历到终点，则说明不可达。

4、

也就是构建最小生成树并计算最小生成树的带权路径和。

以下代码均使用邻接矩阵存储图

**四、核心程序代码**

1.

|  |
| --- |
| Matrix.h |
| #ifndef \_MATRIX\_H\_  #define \_MATRIX\_H\_  #include <limits.h>  #include <string.h>  #include<stdlib.h>  #include <queue>  using namespace std;  /\*  邻接矩阵在设计上是最大20\*20，但实际不一定会使用这么多，所以需要为不用的部分赋值。  对于图（无权图），邻接矩阵赋值为0即可；对于网（有权图），邻接矩阵赋值为INFINITY。  \*/  #define INFINITY INT\_MAX//最大值  #define MAX\_VERTEX\_NUM 20//最大顶点个数  #define VRType int //顶点间关系，用int表示  #define VertexType char //数据类型  //图类型标记，有向图，有向网，无向图，无向网  enum GraphKind { DG, DN, UDG, UDN };  //边的集合，为了构建图而创建的一个数据类型  struct Arc {  VertexType startPoint;  VertexType endPoint;  VRType weight = 1;  };  //邻接矩阵类型  typedef struct ArcCell {  VRType adj;//顶点关系（边或弧），图用1\0表示是否相邻，网为权值  //InfoType\* info;//边或弧的相关信息指针  }ArcCell, AdjMatrix[MAX\_VERTEX\_NUM][MAX\_VERTEX\_NUM];  //图类型，一个图包括顶点集合（点集V）、邻接矩阵（边集E）、顶点数、边/弧数、还有图类型标记  struct Mgraph {  VertexType\* vexs = (VertexType\*)malloc(MAX\_VERTEX\_NUM \* sizeof(VertexType));//顶点向量  AdjMatrix arcs;//邻接矩阵  int vexnum = 0, arcnum = 0;//图的顶点数和边或弧数  GraphKind kind;//图的类型标志  bool structState = false;  bool visited[MAX\_VERTEX\_NUM] = { false };  //构建图及邻接矩阵  void structGraphByMatrix(VertexType\* vertex, int vertexNum, Arc\* arc, int arcNum, GraphKind type) {  vexnum = vertexNum; arcnum = arcNum;//更新  kind = type;//更新  for (int i = 0; i < MAX\_VERTEX\_NUM; i++) {  vexs[i] = i < vexnum ? vertex[i] : 0;  }  //更新邻接矩阵  //1.初始化邻接矩阵  for (int i = 0; i < MAX\_VERTEX\_NUM; i++) {  for (int j = 0; j < MAX\_VERTEX\_NUM; j++) {  arcs[i][j].adj = (type == DG || type == UDG) ? 0 : INFINITY;  }  }  //2.根据边集为邻接矩阵赋值  for (int i = 0; i < arcnum; i++) {  int row = findIndex(arc[i].startPoint);  //例如<vexs[2], vexs[5]>就对应arcs[2][5]，行序为2，列序为5  int column = findIndex(arc[i].endPoint);  if (type == UDN || type == UDG) {  arcs[row][column].adj = type == UDG ? 1 : arc[i].weight;  arcs[column][row].adj = type == UDG ? 1 : arc[i].weight;  }  else {  arcs[row][column].adj = type == DG ? 1 : arc[i].weight;  }  }  structState = true;//确定已完成构建  }  //根据值，查找元素在矩阵中的位置  int findIndex(VertexType val) {  for (int i = 0; i < vexnum; i++) {  if (vexs[i] == val) {  return i;  }  }  return -1;  }  //展示邻接矩阵  void showArcs() {  if (structState) {  for (int i = 0; i < vexnum; i++) {  for (int j = 0; j < vexnum; j++) {  printf("%2d ", arcs[i][j].adj);  }  printf("\n");  }  }  else {  printf("您尚未构建矩该图\n");  }  }  //深度优先遍历  void DFS() {  memset(visited, false, sizeof(visited));  for (int i = 0; i < vexnum; i++)  if (visited[i] == false)  DFS\_Traverse(i);  }  //DFS的递归  void DFS\_Traverse(int v) {  printf("%c", vexs[v]);  visited[v] = true;  int adj = FindAdjVertex(v);  while (adj != -1) {  if (visited[adj] == false)  DFS\_Traverse(adj);  adj = FindAdjVertex(v);  }  }  //广度优先遍历  void BFS() {  memset(visited, false, sizeof(visited));  for (int i = 0; i <= vexnum; i++)  if (visited[i] == false)  BFS\_Traverse(i);  }  //BFS的递归函数  void BFS\_Traverse(int v) {  queue<int> myqueue;  int adj, temp;  printf("%c", vexs[v]);  visited[v] = true;  myqueue.push(v);  while (!myqueue.empty()) { //队列非空表示还有顶点未遍历到  temp = myqueue.front(); //获得队列头元素  myqueue.pop(); //头元素出对  adj = FindAdjVertex(temp);  while (adj != -1)  {  if (visited[adj] == false)  {  printf("%c", vexs[adj]);  visited[adj] = true;  myqueue.push(adj); //进对  }  adj = FindAdjVertex(temp);  }  }  }  };  #endif |

2.

|  |
| --- |
| Matrix.h（这几个函数是属于Matrix.struct Mgraph中的） |
| //查找该点的邻接点  int FindAdjVertex(int x) {  if (kind == UDG) {  for (int i = 0; i <= vexnum; i++)  if (arcs[x][i].adj == 1 && visited[i] == false)  return i;  }  return -1;//表示不存在邻接点  }  //判断能否填色  bool IfCouldAddColor() {  int color[MAX\_VERTEX\_NUM];//每个顶点的颜色标记，0记为颜色1，1记为颜色2，-1记为未填色状态  memset(visited, false, sizeof(visited));  memset(color, -1, sizeof(color));//记录颜色，同时也起到visited数组的作用  int satisfy = 1;  for (int i = 0; i < vexnum; i++)  if (color[i] == -1)  addColor(i, color, 0);  memset(visited, false, sizeof(visited));  for (int i = 0; i < vexnum; i++)  if (visited[i] == false)  satisfy \*= checkColor(i, color);  return satisfy;  }  //递归函数:填色  void addColor(int index, int\* arr, int color) {  arr[index] = color;//给当前点填色  visited[index] = true;  int adj = FindAdjVertex(index);//获取到当前点的邻接点  while (adj != -1) {//存在这个邻接点  if (visited[adj] == false)//如果邻接点和当前点的颜色不同，则继续递归  addColor(adj, arr, abs(1 - color));//邻接点换个颜色  adj = FindAdjVertex(index);  }  }  //检查颜色  bool checkColor(int index, int\* colorArr) {  queue<int> adjVertex;  for (int i = 0; i < vexnum; i++) {  if (arcs[index][i].adj == 1) {  if (colorArr[index] == colorArr[i] && index != i)  return false;  }  }  return true;  } |

3.

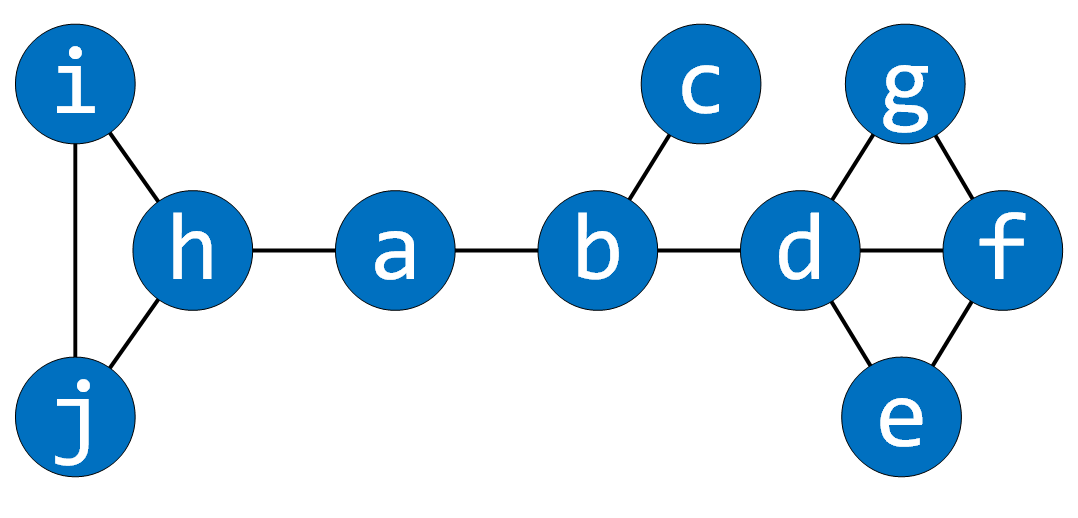
|  |
| --- |
| Matrix.h（这几个函数是属于Matrix.struct Mgraph中的） |
| bool findArcs(int startVertex, int endVertex) {  memset(visited, false, sizeof(visited));  DFS\_Traverse(startVertex);  return (visited[endVertex] == true);  } |

4.

|  |
| --- |
| Matrix.h（这几个函数是属于Matrix.struct Mgraph中的） |
| int Prime(int v)///最小生成树的起点v  {  memset(visited, false, sizeof(visited));  int d[MAX\_VERTEX\_NUM];//记录从当前树到各的最小权值  int sum = 0;//最小路径和  for (int i = 0; i < vexnum; i++)  d[i] = arcs[v][i].adj;  d[v] = 0;  visited[v] = true;//标记遍历过  for (int i = 1; i < vexnum; i++)///一共n个点，要找n-1条边  {  int minn = INFINITY, k=v;//minn是最小权边，k是其对应的index  for (int j = 0; j < vexnum; j++)///在所有u∈U,w∈V-U的边(u,w)∈E中找到一条权值最小的边  {  if (minn > d[j] && visited[j] == false)  {  k = j;  minn = d[j];  }  }  visited[k] = true;///将此点加入最小生成树中  sum += d[k];  for (int j = 0; j < vexnum; j++)//更新不在最小生成树的点的边权值，以便找到下一个权值最小的边  {  if (visited[j] == false && d[j] > arcs[k][j].adj)  d[j] = arcs[k][j].adj;  }  }  return sum;///返回生成的最小生成树的权值和  } |

**五、测试及结果**

前三题的测试用例均为如下图所示的无向图



代码如下：

|  |
| --- |
| GraphKind type = UDG;//先假定一个无向图      VertexType vertexs[10] = { 'a','b','c','d','e','f','g','h','i','j' };//10个点      Arc arcs[12];//12条边      arcs[0].startPoint = 'a'; arcs[0].endPoint = 'b'; arcs[0].weight = 1;      arcs[1].startPoint = 'a'; arcs[1].endPoint = 'h'; arcs[1].weight = 1;      arcs[2].startPoint = 'b'; arcs[2].endPoint = 'c'; arcs[2].weight = 1;      arcs[3].startPoint = 'b'; arcs[3].endPoint = 'd'; arcs[3].weight = 1;      arcs[4].startPoint = 'd'; arcs[4].endPoint = 'e'; arcs[4].weight = 1;      arcs[5].startPoint = 'd'; arcs[5].endPoint = 'f'; arcs[5].weight = 1;      arcs[6].startPoint = 'd'; arcs[6].endPoint = 'g'; arcs[6].weight = 1;      arcs[7].startPoint = 'e'; arcs[7].endPoint = 'f'; arcs[7].weight = 1;      arcs[8].startPoint = 'f'; arcs[8].endPoint = 'g'; arcs[8].weight = 1;      arcs[9].startPoint = 'h'; arcs[9].endPoint = 'i'; arcs[9].weight = 1;      arcs[10].startPoint = 'h'; arcs[10].endPoint = 'j'; arcs[10].weight = 1;      arcs[11].startPoint = 'i'; arcs[11].endPoint = 'j'; arcs[11].weight = 1;      map.structGraphByMatrix(vertexs, 10, arcs, 12, type); |

1. 测试结果：

(1)邻接矩阵

文本

描述已自动生成

(2)深度优先遍历

图形用户界面, 文本

中度可信度描述已自动生成

(3)广度优先遍历

图形用户界面, 文本

描述已自动生成

2. 测试结果：

本题中应是不可填色的，true为1，false为0，结果正确。

图形用户界面, 文本

中度可信度描述已自动生成：

3. 测试用例：

bool res = map.findArcs(1,5);

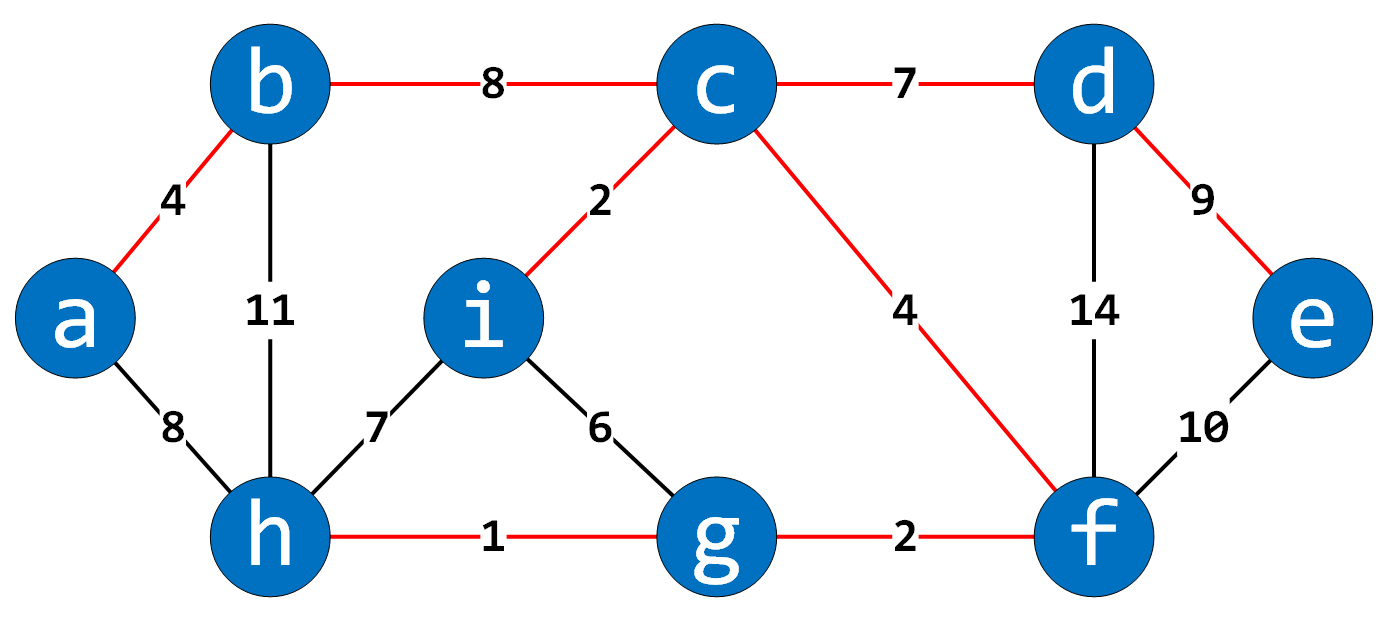
printf("\n%d",res);

文本

描述已自动生成

由于本题调用了深度优先遍历，所以第一行是遍历结果，第二行是1表示存在从b到f的路。

最后一题的测试用例为如下图所示的带权无向图（无向网）



代码如下：

|  |
| --- |
| GraphKind type = UDN;//先假定一个无向图      VertexType vertexs[9] = { 'a','b','c','d','e','f','g','h','i' };//假定有9个点      Arc arcs[14];//假定有14条边      arcs[0].startPoint = 'a'; arcs[0].endPoint = 'b'; arcs[0].weight = 4;      arcs[1].startPoint = 'a'; arcs[1].endPoint = 'h'; arcs[1].weight = 8;      arcs[2].startPoint = 'b'; arcs[2].endPoint = 'h'; arcs[2].weight = 11;      arcs[3].startPoint = 'b'; arcs[3].endPoint = 'c'; arcs[3].weight = 8;      arcs[4].startPoint = 'c'; arcs[4].endPoint = 'i'; arcs[4].weight = 2;      arcs[5].startPoint = 'c'; arcs[5].endPoint = 'd'; arcs[5].weight = 7;      arcs[6].startPoint = 'c'; arcs[6].endPoint = 'f'; arcs[6].weight = 4;      arcs[7].startPoint = 'd'; arcs[7].endPoint = 'f'; arcs[7].weight = 14;      arcs[8].startPoint = 'd'; arcs[8].endPoint = 'e'; arcs[8].weight = 9;      arcs[9].startPoint = 'e'; arcs[9].endPoint = 'f'; arcs[9].weight = 10;      arcs[10].startPoint = 'f'; arcs[10].endPoint = 'g'; arcs[10].weight = 2;      arcs[11].startPoint = 'g'; arcs[11].endPoint = 'h'; arcs[11].weight = 1;      arcs[12].startPoint = 'g'; arcs[12].endPoint = 'i'; arcs[12].weight = 6;      arcs[13].startPoint = 'h'; arcs[13].endPoint = 'i'; arcs[13].weight = 7;      map.structGraphByMatrix(vertexs, 9, arcs, 14, type); |

4. 测试结果：

图形用户界面, 文本

描述已自动生成