图

郑悟强 PB22051082

2023.11.27

1 题目 1：图的遍历

1.1 问题描述

输入一个无向图，输出图的深度优先搜索遍历顺序与广度优先搜索遍历顺序。显然的，最后的答案会有多种可能，这里统一要求：当有多个节点可以搜索时，优先去节点编号最小的那个。

1.2 算法的描述

1.2.1 数据结构的描述

使用邻接矩阵存储方式存储图，可以方便实现多个节点时有限搜索编号最小的（按顺序遍历邻接矩阵的对应行即可）。

```
//使用图的邻接矩阵方式存储
#define MAX_VERTIX_NUM 20       //最多结点数量
int visited[MAX_VERTIX_NUM];    //visited数组用于遍历时判断是否遍历过

typedef struct {

}MGraph;

class Graph{
private:
    int vexes[MAX_VERTIX_NUM];    //顶点向量
    int ArcCell[MAX_VERTIX_NUM][MAX_VERTIX_NUM];    //邻接矩阵
    int vexnum,arcnum;            //图的当前顶点数和弧数
    int visited[MAX_VERTIX_NUM];
```

1.2.2 程序结构的描述

DFS 算法的基本逻辑为：

从开始位置结点开始遍历，接着遍历结点连接到的结点，遍历结束后，再遍历下一个连接到的结点，实现效果为，沿一个路径一直走到结尾，再不断回退找另外的路径。具体代码为：

```
void Graph::DFSTraverse(){
    //初始化一下visited数组
    for(int i=1;i<=vexnum;i++){
        visited[i]=0;
    }
    int start;
    cout<<"DFS遍历起点为: ";
    cin>>start;
    DFS(start);
    cout<<endl;
}

void Graph::DFS(int start){
    visited[start] = 1;
    cout<<start<<" ";
    for(int i = 1;i<=vexnum;i++){
        if(ArcCell[start][i]>0&&visited[i]==0){
            DFS(i);
        }
    }
}
```

BFS 算法的基本逻辑为：

借助辅助队列，每次将对应结点入队，然后遍历所有队头结点连接的结点，将其全部入队，重复操作。实现效果为，依次找到 n 步能走到的全部结点，按顺序输出。具体代码为：

```
void Graph::BFSTraverse(){
    //初始化一下visited数组
    for(int i=1;i<=vexnum;i++){
        visited[i]=0;
    }
    int start;
    cout<<"请输入BFS遍历起点: ";
    cin>>start;
    BFS(start);
}

void Graph::BFS(int start){
//使用队列（队列太麻烦写了就用个数组代替了）
    int queue[100];
    int front = 0;
    int back = 0;

    queue[back] = start;
    back++;
    visited[start] = 1;
    cout<<start<<" ";
    while(back>front){//队列非空
        for(int i=1;i<=vexnum;i++){
            if(ArcCell[front][i]>0 && visited[i]==0){
                queue[back++] = i;
                visited[i] = 1;
                cout<<i<<" ";
            }
        }
        front++;//相当于出队
    }
}
```

1.3 算法时空分析

1.3.1 时间复杂度

DFS 算法由于每次会遍历整行的邻接矩阵，共 n 个结点，所以复杂度为 O(n^2)(n 为结点数量)。

BFS 算法同样会遍历整行的邻接矩阵，n 个结点要遍历 n 遍，复杂度为 O(n^2)。

1.3.2 空间复杂度

DFS 算法需要辅助数组 visited 来判断是否遍历过，复杂度为 O(n)。

BFS 算法需要辅助数组 visited 和辅助队列，复杂度为 O(n)。

2 题目 2：求通讯网最小代价生成树

2.1 问题描述

输入一个无向铁通讯网图，用 Prim 和 Kruskal 算法计算最小生成树并输出。

2.2 算法的描述

2.2.1 数据结构的描述

使用邻接链表存储方式存储图

```
//使用图的邻接链表方式存储
#define MAX_VERTIX_NUM 20//最多结点数量
int visited[MAX_VERTIX_NUM];    //visited数组用于遍历时判断是否遍历过
typedef struct ArcNode{
    int adjvex;                 //该弧所指向的顶点的位置
    int weight;                 //弧的权重
    struct ArcNode *nextarc;    //指向下一条弧的指针
}ArcNode;
typedef struct VNode{
    int data;                   //顶点的内容
    ArcNode *firstarc;          //顶点的第一条弧
}VNode,AdjList[MAX_VERTIX_NUM+1];
typedef struct ALGraph{
    AdjList vertices;
    int vexnum,arcnum;          //顶点数量和弧的数量
}ALGraph;
```

2.2.2 程序结构的描述

Prim 算法的基本逻辑为：

每次找到路径最小且不在已有联通分量的路径，一共找 vexnum-1 次，具体代码为：

```
void Graph::MinSpanTree_PRIM(){
    int sum = 0;
    //从第一个顶点开始
    //初始化辅助数组
    for(int i=1;i<=G.vexnum;i++){
        closeedge[i] = {0,0};
    }
    for(ArcNode *p = G.vertices[1].firstarc;p!=nullptr;p = p->nextarc){
        closeedge[p->adjvex] = {1,p->weight};
    }
    //一步步连这vexnum-1条弧
    for(int j = 2;j<=G.vexnum;j++){
        int k = min();

        sum += closeedge[k].lowcost;

        //更新辅助数组
        closeedge[k] = {0, 0};
        for(ArcNode *p = G.vertices[k].firstarc;p!=nullptr;p = p->nextarc){
            if(p->weight < closeedge[p->adjvex].lowcost){
                closeedge[p->adjvex] = {k,p->weight};
            }
        }
    }
    cout<<sum;
}
```

核心操作：min 函数：

```
int Graph::min(){
    int pos;
    //定位第一个pos
    for(int i=1;i<=G.vexnum;i++){
        if(closeedge[i].lowcost != 0){
            pos = i;
            break;
        }
    }
    //找到最小的pos
    for(int j = pos+1;j<=G.vexnum;j++){
        if(closeedge[j].lowcost > 0 && closeedge[j].lowcost < closeedge[pos].lowcost){
            pos = j;
        }
    }
    return pos;
}
```

Kruskal 算法的基本逻辑为：

初始时每个结点属于自己的联通分量，并且将所有的弧按照权重排序，每次选中一个弧后，将两个结点及所在联通分量归为一个，每次找两个顶点在不同联通分量的最小的弧。具体代码为：

辅助数组：

```
struct{
    int start;
    int end;
    int weight;
}arcs[MAX_ARCS_NUM];        //记录所有弧的两头和权重
int nodes[MAX_VERTIX_NUM];  //记录所有顶点所在的联通分量
```

核心算法：

```
void Graph::MinSpanTree_Kruskal(){//用kruskal算法构造最小生成树
    //先将所有的弧排序
    sort();

    //初始化所有联通分量数组
    for(int t=1;t<=G.vexnum;t++){
        nodes[t] = t;
    }

    int sum = 0;
    int pos = 1;
    for(int i=1;i<G.vexnum;i++){//一共找vexnum-1个弧
        //找到第一个两个结点联通分量不同的弧
        while(nodes[arcs[pos].start] == nodes[arcs[pos].end]){
            pos++;
        }
        //选中这个弧
        sum += arcs[pos].weight;
        //这个弧两端所属联通分量相同
        for(int j = 1;j<=G.arcnum;j++){
            if(nodes[j]==arcs[pos].end){
                nodes[j] = nodes[arcs[pos].start];
            }
        }
        for(int m = 1;m<=G.vexnum;m++){
            cout<<nodes[m]<<" ";
        }
        cout<<endl;
    }
    cout<<sum;
}
```

排序算法：

```
void Graph::sort(){//用冒泡排序
    for(int i=1;i<G.arcnum;i++){
        for(int j=1;j<G.arcnum-i;j++){
            if(arcs[j].weight>arcs[j+1].weight){
                int w = arcs[j].weight;
                arcs[j].weight = arcs[j+1].weight;
                arcs[j+1].weight = w;

                int s = arcs[j].start;
                arcs[j].start = arcs[j+1].start;
                arcs[j+1].start = s;

                int e = arcs[j].end;
                arcs[j].end = arcs[j+1].end;
                arcs[j+1].end = e;
            }
        }
    }
}
```

2.3 算法时空分析

2.3.1 时间复杂度

Prim 算法核心为每次找到所在联通分量不同的最短的弧，需要遍历所有的弧，时间复杂度为 O(ne)(n 为所有结点数量，e 为所有弧的数量)

Kruskal 算法首先要进行排序，冒泡排序复杂度为 O(e^2)，再进行循环，循环中要更新数组，需要遍历所有结点所在的连通分量，复杂度为 O(n^2)，所以总的复杂度为 O(n^2+e^2)。

2.3.2 空间复杂度

DFS 算法需要辅助数组 closeedge 来判断所在连通分量，复杂度为 O(e)。

BFS 算法需要辅助数组 nodes 来判断所在连通分量，复杂度为 O(n)。

3 题目 3：铁路交通网的最短路径

3.1 问题描述

输入一个无向铁路交通图、始发站和终点站，用 Dijkstra 算法计算从始发站到终点站的最短路径。

3.2 算法的描述

3.2.1 数据结构的描述

使用邻接链表存储方式存储图

```
//使用图的邻接链表方式存储
#define MAX_VERTIX_NUM 20//最多结点数量
int visited[MAX_VERTIX_NUM];    //visited数组用于遍历时判断是否遍历过
typedef struct ArcNode{
    int adjvex;                 //该弧所指向的顶点的位置
    int weight;                 //弧的权重
    struct ArcNode *nextarc;    //指向下一条弧的指针
}ArcNode;
typedef struct VNode{
    int data;                   //顶点的内容
    ArcNode *firstarc;          //顶点的第一条弧
}VNode,AdjList[MAX_VERTIX_NUM+1];
typedef struct ALGraph{
    AdjList vertices;
    int vexnum,arcnum;          //顶点数量和弧的数量
}ALGraph;
```

3.2.2 程序结构的描述

Dijkstra 算法的基本逻辑为：

(1) 集合 S 初始为空，D[] 初始为起点到每个结点的距离。

(2) 从顶点集合 V-S 中找到最短的路径，即 D[j] = MinD[i]|$V_i \in V-S$，再令 S = S∪j。

(3) 修改最短路径长度：if D[j] + Edge[j][k] < D[k],then D[k] = D[j] + Edge[j][k]。

(4) 重复 (2),(3) 操作 n-1 次。

具体代码为：

```
void Graph::ShortestPath_DIJ(){
    int D[G.vexnum+1];
    int S[G.vexnum+1];//用于表示是否在已经选过的里面
    for(int k = 1;k<=G.vexnum;k++){
        D[k] = 0;S[k] = 0;
    }
    int begin,end;
    cout<<"请输入起点和终点：";
    cin>>begin>>end;
    S[begin] = 1;           //起点属于S集
    D[begin] = 0;
    //先把起点的对应的弧连上
    for(ArcNode *p = G.vertices[begin].firstarc;p;p = p->nextarc){
        D[p->adjvex] = p->weight;
    }
    //一共循环vexnum-1次
    for(int i=1;i<G.vexnum;i++){
        int min;        //找到V-S集合中路径最短的
        for(int j = 1;j<=G.vexnum;j++){
            if(S[j]==0 && D[j]>0){
                min = j;
                break;
            }
        }
        for(int t = min+1;t<=G.vexnum;t++){
            if(S[t] == 0 && D[t] < D[min] && D[t]>0){
                min = t;
            }
        }
        S[min] = 1;//这个点属于S了
        for(ArcNode *p = G.vertices[min].firstarc;p != nullptr;p = p
            if(D[p->adjvex] == 0 || D[min] + p->weight < D[p->adjvex
                D[p->adjvex] = D[min]+p->weight;
            }
        }
    }
    cout<<D[end];
```

3.3 算法时空分析

3.3.1 时间复杂度

算法中需遍历 n-1 次，同时每一次中需要找到最小的值，也需遍历 n 次，所以总时间复杂度为 $O(n^2)$。

3.3.2 空间复杂度

有 D 数组和 S 数组为辅助数组，空间复杂度为 O(n)。

4 附加题：显示图

4.1 问题描述

可以直接使用题目 1 的输入数据画出图的情况则为满足要求，要求图整体美观，可以体现出顶点的编号，边的交叉尽量少。

4.2 算法的描述

4.2.1 程序结构的描述

(1) 深度优先搜索时，同一个结点连接的各个结点纵坐标相同，为根节点 +1，横坐标从 0 开始依次 +1，记录每个结点坐标。

(2) 建立画框，根据坐标位置画圆，代表结点。

(3) 遍历所有的边，进行连线。

具体代码为：

```
void Graph::DFS(int start) {
    visited[start] = 1;
    cout << start << " ";
    int x = pos[start][1];
    for (ArcNode* p = G.vertices[start].firstarc; p != nullptr; p = p->nextarc) {
        if (visited[p->adjvex] == 0)
        {
            //记录深度有限搜索次序，来记录坐标
            pos[p->adjvex][1] = x;
            x++;
            pos[p->adjvex][0] = pos[start][0] + 1;
            DFS(p->adjvex);
        }
    }
}
```

```
void Graph::DrawGraph() {
    //  创建一个宽度为WINDOW_WIDTH、高度为WINDOW_HEIGHT的窗体，并用白色背景色刷新窗体。
    initgraph(WINDOW_WIDTH, WINDOW_HEIGHT);
    setbkcolor(WHITE);
    setlinecolor(BLACK);
    settextcolor(BLACK);
    cleardevice();


    //用dfs遍历得到的顺序画树
    for (int i = 1; i <= G.vexnum; i++) {
        setfillcolor(YELLOW);
        fillcircle(200+pos[i][0]*150, 200+pos[i][1]*150, 30);
        char nu = i+48;
        outtextxy(200 + pos[i][0] * 150, 200 + pos[i][1] * 150, nu);
        for (ArcNode* p = G.vertices[i].firstarc; p; p = p->nextarc) {
            line(200 + pos[i][0] * 150, 200 + pos[i][1] * 150, 200 + pos[p->adjvex][0] * 150, 200 + pos[p->adjvex][1] * 150);
        }
    }
    system("pause");
    closegraph();
}
```

4.3 算法时空分析

4.3.1 时间复杂度

算法复杂度及 DFS 复杂度加上遍历所有的弧，复杂度为 O(ne)。

4.3.2 空间复杂度

DFS 算法中需 visited 数组记录是否遍历过，还需要坐标 pos 数组记录坐标，空间复杂度为 O(n)。

5 实验体会与分析

通过本次实验，我熟练了图的基本存储方式及一些图有关的基本算法。同时附加题还让我学会了 easyx 库的基本绘图方法，以及如何实现基本的画图显示操作。