算法设计—图

目录

1. 图的存储结构

邻接表

```
typedef struct ArcNode{ // 边表结点
    int adjvex; // 该弧所指向的顶点的位置
    int weight; // 弧的权值
    struct ArcNode *next; // 指向下一条弧的指针
}ArcNode;

typedef struct VNode{ // 顶点表结点
    VertexType data;  // 顶点信息
    ArcNode *first;   // 指向第一条依附该顶点的弧的指针
}VNode, AdjList[MaxVertexNum];

typedef struct{
    AdjList vertices;   // 邻接表
    int vexnum, arcnum; // 图的顶点数和弧数
}AALGraph;              // ALGraph 是以邻接表存储的图类型
```

邻接矩阵

```
typedef struct{
    VertexType Vex[MaxVertexNum]; // 顶点表
    EdgeType Edge[MaxVertexNum][MaxVertexNum]; // 邻接矩阵，边表
    int vexnum, arcnum; // 图的当前顶点数和边数
}MGraph;
```

2. 图的遍历

BFS（邻接表存储）

$$\begin{cases}\text{时间复杂度}\begin{cases}\text{邻接表存储}O(|V|+|E|)\\ \text{邻接矩阵存储}O(|V|^2)\end{cases}\\ \text{空间复杂度 }O(|V|)\end{cases}$$

【注】:（1）BFS 可以用来解决单源最短路径问题，但要求所有边的权值相等。

（2）可以用 BFS 或 DFS 来求<u>无向图</u>的连通分量。

```
bool visited[MaxVertexNum]; // 访问标记数组

void BFSTraverse(Graph G){
    memset(visited, 0, sizeof(visited));
    InitQueue(Q);
    for (i = 0; i < G.vexnum; i++){ // 对每个连通分量调用一次 BFS
        if (!visited[i])
            BFS(G, i);
    }
}
// 从顶点 V 出发，BFS 遍历图 G
```

```
void BFS(Graph G, int v){
    visit(v);
    visited[v] = true;
    EnQueue(Q, v); // 顶点入队
    while (!IsEmpty(Q)){
        DeQueue(Q, v);
        // 检测 v 的所有邻接点
        for (w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w))
            if (!visited[w]){
                visit(w);
                visited[w] = true; // 访问标记
                EnQueue(Q, w);
            }
    }// while
}
```

DFS 递归（邻接表存储）：

【注】复杂度同 BFS

```
bool visited[MaxVertexNum];

void DFSTraverse(Graph G){
    memset(visited, 0, sizeof(visited));
    for (v = 0; v < G.vexnum; v++)
        if (!visited[v])
            DFS(G, v);
}

void DFS(Graph G, int v){
    visit(v);
    visited[v] = true;
    for (w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w))
        if (!visited[w])
            DFS(G, w);
}
```

DFS 非递归(邻接表存储)：

```
void DFS(Graph &G, int v){
    int w; // 顶点序号
    InitStack(S);
    memset(visited, 0, sizeof(visited));
    Push(S, v);
    visited[v] = true;
    while (!IsEmpty(S)){
        k = Pop(s);
        visit(k);
        for (w = FirstNeighbor(G, k); w >= 0; w = NextNeighbor(G, k, w))
```

```
            if (!visited[w]){
                Push(S, w);
                visited[w] = true;
            }
    }// while
}
```

3. MST 最小生成树算法

Prim

```
//辅助数组，用来定义记录从顶点集 U 到 U-V 的权值最小的边
typedef struct {
    VertexType adjvex; // 最小边在 U 中的那个顶点
    int lowcost; // 最小边上的权值
}closeEdge[MVNum];

void Prim(MGraph G, VertexType u){
    k = locateVex(G, u);
    for (j = 0; j < G.vexnum; j++) // 对 V-U 的每一个顶点 Vj,初始化 closeEdge[j]
        if (j != k)
            closeEdge[j] = {u, G.Edge[k][j]};
    closeEdge[k].lowcost = 0; // 初始化，U = {u}
    for (i = 1; i < G.vexnum; i++){
        k = Min(closeEdge);
        u0 = closeEdge[k].adjvex; // U0 是最小边的一个顶点， U0 属于 U
        v0 = G.Vex[k]; // V0 是最小边的另一个顶点，V0 属于 V-U
        print(u0, v0); // 输出最小边
        closeEdge[k].lowcost = 0; // 第 k 个顶点并入 U
        for (j = 0; j < G.vexnum; j++)
            if (G.Edge[k][j] < closeEdge[j].lowcost)
                // 新顶点并入 U 后选择新的最小边
                closeEdge[j] = {G.Vex[k], G.Edge[k][j]};//更新 U-V 中的 closeEdge[]
    }
}
```

Kruskal

```
//辅助数组，表示各个顶点所属的连通分量
int Vexset[MVnum];

void Kruskal(MGraph G){
    Sort(Edge); // 将边按权值递增排序
    for (i = 0; i < G.vexnum; i++) // 各顶点自成一个连通分量
        Vexset[i] = i;
    for (i = 0; i < G.arcnum; i++){
        v1 = LocateVex(G, Edge[i].Head); // 权值最小的边的两个顶点 V1，V2
        v2 = LocateVex(G, Edge[i].Tail);
```

```
        vs1 = Vexset[v1]; // 获取连通分量
        vs2 = Vexset[v2];
        if (vs1 != vs2){
            print(Edge[i].Head, Edge[i].Tail); // 输出边
            for (j = 0; j < G.vexnum; j++)
                if (Vexset[j] == vs2)
                    vexset[j] = vs1; // 合并连通分量
        }
    }// for
}
```

4. 最短路径算法

Floyd

```
// path[][]: 记录最短路径上顶点Vj的前一顶点的序号
void Floyd(MGraph G){
    for (i = 0; i < G.vexnum; i++)
        for (j = 0; j < G.vexnum; j++){
            D[i][j] = G.Edge[i][j];i
            if (D[i][j] < MaxInt) // 如果i和j之间有弧，将j的前驱置为i
                path[i][j] = i;
            else // // 如果i和j之间无弧，将j的前驱置为-1
                path[i][j] = -1;
        }
    for (k = 0; k < G.vexnum; k++)
        for (i = 0; i < G.vexnum; i++)
            for (j = 0; j < G.vexnum; j++)
                if (D[i][k] + D[k][j] < D[i][j]){
                    D[i][j] = D[i][k] + D[k][j];
                    path[i][j] = path[k][j]; // 更新j的前驱为k
                }
}
```

Dijkstra

```
// S[]:记录从源点V0到终点Vj是否已被确定找到最短路径，即Vj是否已加入顶点集合
void Dijkstra(MGraph G, int v0){
    n = G.vexnum;
    for (v = 0; v < n; v++){
        S[v] = false; // S初始化为false
        D[v] = G.Edge[v0][v]; //将V0到各个终点的最短路径长度初始化为弧上的权值
        if (D[v] < MaxInt) // 如果v0和v之间有弧，则将v的前驱置为v0
            path[v] = v0;
        else  // 如果无弧，前驱置为-1
            path[v] = -1;
    }
    S[v0] = true; // 将v0加入S
```

```
    D[v0] = 0; // 源点到源点的距离为 0
    /* ***** 初始化结束 ***** */
    //开始主循环，每次求得 v0 到某个顶点 v 的最短距离，将 v 加到 S 集
    for (i = 1; i < n; i++){
        min = MaxInt;
        for (w = 0; w < n; w++)
            if (!S[w] && D[w] < min){
                v = w; // 选择一条当前最短路径，终点为 v
                min = D[w];
            }
        S[v] = true; // 将 v 加入 S
        for (w = 0; w < n; w++) // 更新 D[w],即前驱
            if (!S[w] && (D[v] + G.Edge[v][w] < D[w])){
                D[w] = D[v] + G.Edge[v][w];
                path[w] = v;
            }
    }
}
```

5. 有向图，是否存在以 V0 为起点的包含所有顶点的简单路径。

使用 DFS。设图的顶点信息就是图的编号，用 num 记录访问顶点的个数，当 num 等于图的顶点个数,输出所访问的顶点序列,顶点序列存在 path 中。path 和 visited 数组,顶点计数器 num,均是全局变量,都已初始化。

```
void getPath(Graph G, int v0){
    visited[v0] = true;
    path[++num] = v0;
    if (num == G.vexnum) { // 有一条简单路径，输出
        for (i = 1; i <= num; i++)
            cout << path[i];
        cout << endl;
        exit(0);
    }
    ArcNode p = G.vertices[v0].first;
    while (p){
        if (!visited[p ->adjvex]) // DFS 遍历下一个结点
            getPath(G, p ->adjvex);
        p = p ->next;
    }
    visited[v0] = false; // 取消访问标记
    num--;
}
```

6. 求距离定点 V0 的最短路径为 K 的所有顶点。

用广度优先遍历求解。以 $v0$ 作生成树的根为第 1 层，则距顶点 $v0$ 最短路径长度为 K 的顶点均在第 K+1 层。可用队列存放顶点，将遍历访问顶点的操作改为入队操作。队列中设头尾指针为 f 和 r，用 level 表示层数。

```
void getVex_K(Graph G, int v0){
    r = f = 0;  level = 1;
    Q[++r] = v0;  t = r;
    bool flag = false; // flag：false 表示查找结果不存在
    visited[v0] = true;
    while (f < r && level <= k+1){
        v = Q[++f];
        w = FirstNeighbor(G, v);
        while (w != 0){ // w != 0 表示邻接点存在
            if (!visited[w]){
                Q[++r] = w; // 邻接点入队
                visited[w] = true;
                if (level == k+1){
                    cout << "距离顶点 V0 最短距离为 k 的顶点：" << w;
                    flag = true;
                }
                w = NextNeighbor(G, v, w);
            }
        }// while
        if (f == t){ // 当前层处理结束，进入下一层
            level++;
            t = r;
        }
    }// while
    if (!flag)
        cout << "图中无距 V0 顶点距离最短距离为 K 的顶点";
}
```

7. 无向图，邻接表，求 Vi 与 Vj 之间是否存在一条长度为 K 的简单路径。

采用 DFS，找到 i 的第一个邻接点 adj，在从 adj 出发递归地求出是否存在 adj 到 j 的长度为 k-1 的简单路径。简言之，在 DFS 中加上深度参数。

```
bool path_K(Graph G, int i, int j, int k){
    if (i == j && k == 0) // 找到一条路径，并且长度符合
        return true;
    else if (k > 0){
        visited[i] = true;
        for (p = G.vertices[i].first; p != NULL; p = p ->next){
            adj = p ->adjvex;
            if (!visited[adj])
                if (path_K(G, adj, j, k-1)) // 剩余路径长度减一
```

```
                return true;
        }
        visited[i] = false; // 恢复状态
    }
    return false; // 没有符合条件的路径
}
```

8. 邻接矩阵，有向图，是否存在简单有向回路，并输出。

实质是判断图是否有环，可利用拓扑排序进行判断，若存在环，将未输出的结点依次输出，即为环的路径。

```
// indegree[]存放各顶点的入度值
// 并用值为 0 的入度域当栈，用 top（初值-1）指向栈顶元素
void topo(MGraph g){
    bool visited[] = {false};
    for (i = 0; i < n; i++)
        if (indegree[i] == 0){
            indegree[i] = top;
            top = i;
            visited[i] = true; // 进入拓扑序列
        }
    while (top != -1){
        m++; // m 表示拓扑序列中的顶点数目
        j = top; // j 记录排序顶点
        top = indegree[top]; // 出栈
        for (i = 0; i < n; i++)
        if (g[j][i] == 1){
            indegree[i]--;
            if (indegree[i] == 0){ // 入度为 0 的顶点入栈
                indegree[i] = top;
                top = i;
                visited[i] = true;
            }
        }
    }//while
    if (m == n)
        cout << "无环";
    else {
        cout << "有环";
        for (i = 0; i < n; i++)
            if (!visited[i])
                printCycle(i, i); // 输出环
    }
}
```

```
// 输出以 start 为起始点的环
void printCycle(int v, int start){
    for (i = 0; i < n; i++){
        if (g[v][i] != 0 && !visited[i]){
            cout << v;
            if (i != start) // 直到回到起始点，即输出环
                printCycle(i, start);
            break;
        }

    }
}
```

9. 有向无环图，邻接表，非递归，权值均为 1，求每个定点出发的最长路径的长度。

对于每个顶点出发，利用 BFS，当队列为空前，最后一个顶点的层次减一，就是从该定点出发的最长路径长度。

10. 邻接表，有向图，输出 Vi→Vj 的所有简单路径

采用 DFS，从 Vi 出发，递归遍历图中顶点，设置一个 path 数组，用于存储路径，当访问到 Vj 时，则输出该搜索路径上的结点。

```
int path[]; // 存放路径
// 查找图 G 中 vi 到 vj 的路径，len 表示路径长度
void FindPath(Graph G, int vi, int vj, int len){
    int w;
    ArcNode *p;
    path[++len] = vi; // 当前结点加入路径
    visited[vi] = true; // 标记已访问
    if (vi == vj) // 找到路径，并输出
        print(path[]);
    p = G ->vertices[vi].first; // 指向邻接点的第一个相邻点
    while (p){
        w = p ->adjvex;
        if (!visited[w]) // 递归查找下一个邻接点
            NextNeighbor(G, w, vj, len);
        p = p ->next;
    }
    visited[vi] = false;
}
```

11. 邻接矩阵，有向图，BFS，是否存在 Vi→Vj 的路径

```
int Exist_Path_BFS(ALGraph G, int i, int j){
    bool visited[] = {false};
    InitQueue(Q);
    EnQueue(Q, i);
```

```
    while (!isEmpty(Q)){
        DeQueue(Q, u);
        visited[u] = true;
        for (p = FirstNeighbor(G, i); p; p = NextNeighbor(G, i, p)){
            k = p.adjvex;
            if (k == j) // 查找成功
                return true;
            if (!visited[k])
                EnQueue(Q, k);
        }
    }
    return false; // 查找失败
}
```

12. 邻接表，有向图，是否存在回路。

拓扑排序，或者 DFS

13. 邻接矩阵，有向无环图，求图中的最长路径长度

在第 9 题的基础上，选出最长路径即可。

14. 求自由树（无环连通图）的直径（最短距离中最长的）

两次 BFS。第一次 BFS，从任意一点 v 出发，找到距离 v 最远的一点 u。第二次 BFS，从找到的 u 出发，找到距离 u 最远的一点 w。其中 u 与 w 之间的路径长度，即为自由树的直径。

15. 求图的关节结点（割点），邻接表，链接表，联通有向图

提供两个思路。

（1）**暴力法**（时间复杂度较高）：暴力法的原理就是通过定义求解割点和割边。在图中去掉某个顶点，然后进行 DFS 遍历，如果连通分量增加，那么该顶点就是割点。

（2）**Tarjan 算法**（时间复杂度 O（V+E））：假设 DFS 中我们从顶点 U 访问到了顶点 V（此时顶点 V 还未被访问过），那么我们称顶点 U 为顶点 V 的父顶点，V 为 U 的孩子顶点。在顶点 U 之前被访问过的顶点，我们就称之为 U 的祖先顶点。

显然如果顶点 U 的所有孩子顶点可以不通过父顶点 U 而访问到 U 的祖先顶点，那么说明此时去掉顶点 U 不影响图的连通性，U 就不是割点。相反，如果顶点 U 至少存在一个孩子顶点，必须通过父顶点 U 才能访问到 U 的祖先顶点，那么去掉顶点 U 后，顶点 U 的祖先顶点和孩子顶点就不连通了，说明 U 是一个割点。

```
// visited 数组的下标表示顶点的编号，数组中的值表示该顶点在 DFS 中的遍历顺序(或者说时间戳)，每访问到一个未访问过的顶点，访问顺序的值（时间戳）就增加 1
// low 数组的下标表示顶点的编号，数组中的值表示 DFS 中该顶点不通过父顶点能访问到的祖先顶点中最小的顺序值（或者说时间戳）。
int visited[], low[];
int count;
void dfs_articul(AdjList g, vertype v0){
    count++;
```

```
    visited[v0] = count; // 访问顺序号放入 visited[]
    min = visited[v0]; // 初始化访问初值
    ArcNode p = g[v0].first;
    while (p){
        w = p ->adjvex;
        if (!visited[v0]){
            dfs_articul(g, w);
            if (low[w] < min)
                min = low[w];
            if (low[w] >= visited[v0]) // 找到割点
                print(g[v0].vertex);
        } else { // w 已被访问，是 v 的祖先
            if (visited[w] < min)
                min = visited[w];
        }
        p = p ->next;
    }// while
    low[v0] = min;
}

// 主调用函数
void get_articul(){
    // 初始化
    memset(visited, 0, sizeof(visited));
    count = 1;
    visited[1] = 1;
    // 设邻接表上第一个结点是生成树的根，并从此处开始访问
    p = g[1].first;
    v = p ->adjvex;
    dfs_articul(g, v);
    if (count < n){ // 生成树的根有两棵以上的子树
        prinf(g[1].vertex); // 根是割点
        while (p ->next){
            p = p ->next;
            v = p ->adjvex;
            if (!visited[v])
                dfs_articul(g, v);
        }
    }
}
```

16. 无向连通图，求半径最小的生成树。（根到叶子的最大距离，称为树的半径）

用邻接表作为存储结构。依次删去树叶(度为 1 的结点)，将与树叶相连的结点度数减

1。设在第一轮删去原树 T 的所有树叶后，所得树为 T;再依次做第二轮删除，即删除所有 T 的叶子:如此反复，若最后剩下-一个结点，则树直径应为删除的轮数 X2,具体算法如下:

```
int MAX_D(){
  m=0;
  for(i=1;i≤n;i++)
    if(du(vei1)-1){
      Enqueue(Q, v[i]); //叶子 vi 入队
      m=m+1; //m 记录当前一轮叶子数
    }
   r=0; //表示删除叶子轮数
   while(m>=2){//当前叶子轮数
    j=0; //j 计算新一轮叶子数目
    for(i=1;i<=m;i++){
      dequeue(Q, v); //出队，表示删去叶子 v 将与 v 相邻的结点 w 的度数减 1
      if(du(w)==1){//w 是新一轮的叶子
        j=j+1;
        Enqueue(Q, w);//w 入队
        r=r+1; //轮数加 1
        m=j; //新一轮叶子总数
       }

     }//for
   }//while
        if(m==0)
            return 2*r-1; //m=0，直径为轮数*2-1
        else
            return 2*r; //m=1，直径为轮数*2
 }
```

17. 求图的连通分量数

遍历完一个图，需要调用 DFS 或者 BFS 的次数。

18. 删除边，无向图，邻接表

注意：无向图的邻接表中，每条边出现两次。

19. 删除顶点，邻接表，有向图（邻接矩阵，无向图）

注意：邻接矩阵删除顶点时，矩阵变成 n-1 阶。

邻接表删除顶点时，要区分删除的是否是第一个邻接顶点，顶点表大小还要减一。

20. 判断给定序列书否是一个图的拓扑排序序列，邻接表

设“任意给定序列 $v_1, v_2, \cdots, v_n$”存在一队列中，再设一栈存放入度为 0 的顶点。队列中元素出队，与栈中元素比较（尽管栈具有后进先出的性质，但是入度为 0 的顶点并无先后次序，故栈中所有元素都可以和队头元素比较），若比较相等，则出队，删除栈中匹配元素，被删元素要按拓扑排序办法处理（即从其发出的所有弧头顶点的入度均减 1，减成 0 则弧头顶点入栈）。如此进行下去，直至所有队空和栈空，结论为给定序列是该有向图的一个拓扑序列。若在比较过程中，任意阶段发生队头与栈中元素不等，则给定序列不是该有向图的一个拓扑序列。

21. 邻接表存储->邻接矩阵存储　|　邻接矩阵存储->邻接表存储
22. 判断一个图是否是一棵树(王道-5.3.4-2)
23. 用 DFS 得到图的一个拓扑序列（王道-5.4.5-9）