@[TOC]

图相关知识点以及代码

1、基本概念

各种图

- 有向图：边是有向边
- 无向图：边是无向边
- 简单图：**不存在重复边**，**不存在顶点到自身的边**
- 多重图：某两个顶点之间的**边的个数大于1**
- 完全图：**任意两个顶点之间存在边**。有**n*(n-1)条**边的有向图是**有向完全图**
- 连通图：**无向图G**中任意两个顶点时连通（顶点v到顶点w有路径存在）的，称为连通图，**极大连通子图**为连通分量
- 强连通图：有向图，有一对顶点w和v，两点之间有路径可以互相到达，称为强连通。如果图中**任何一对顶点都是强连通，则此图为强连通图**
- 带权图：每条边上带有数值

其他概念

- 入度：以v为终点的边的个数
- 出度：以v为起点的边的个数
- 顶点的度：出度+入度
- 路径长度：路径上边的个数
- 简单路径：除了起点和重点相同，其他不同
- 简单回路：**简单路径起点与终点相同，**路径长度大于等于2

2、存储结构

(1) 领接矩阵

- 如果为有权图，aij表示对应的边<vi,vj>的权值
- 如果为非权图：
  - **aii = 0**
  - aij=1表示，当i不等于j时，<vi,vj>存在
  - aij=0表示，当i不等于j时，<vi,vj>不存在
- 求度
  - 无向图：领结矩阵的第i行（或第j列）的非零元素的个数为顶点vi的度
  - 有向图：第i行非零元素的个数为顶点vi的出度，第i列非零元素的个数为顶点vi的入度

代码表示：

```
typedef struct{
    int vex[maxsize];               //顶点表
    int edge[maxsize][maxsize];     //领接矩阵
    int vexnum, edgenum;            //顶点个数，边的个数
}MGraph;
```

(2) 领接表

主要分为三大部分

1. 边节点
2. 表节点
3. 领接表

代码表示:

```
#define maxnum 100
//边表节点
typedef struct ArcNode{
    int adjvex;
    struct ArcNode *next;
}ArcNode;

//顶点表节点
typedef struct VNode{
    int data;
    struct ArcNode *firstarc;
}VNode;

//领接表
typedef struct{
    VNode adjlist[maxnum];
    int vexnum,edgenum;
}AGraph;
```

3、遍历算法

（1） 深度优先搜索

- 递归遍历
  - 领接表

  思路: 取当前节点的firstarc指针，然后对其后序的节点进行遍历，遇到非空节点且未被访问，进行递归处理

  代码:

```
int visited[maxsize] = {0};      //标记访问数组

//深度优先遍历DFS  递归实现
void DFS_AGraph(AGraph *G, int i){//从顶点i深度遍历树

    ArcNode *p;
    printf("%d", G->adjlist[i].data);
```

```
    visited[i] = 1;
    //取领结表的表结点的第一个值，然后分别对其的边节点进行深度优先遍历
    p = G->adjlist[i].firstarc;
    while(p!=NULL){
        if(visited[p->adjvex] == 0)
            DFS_AGraph(G, p->adjvex);
        p = p->next;
    }
}

//如果是非连通图（这里可以用来计算连通图个数）
void DFS_AGraph_Travel(AGraph *G){
    int i = 0;
    for(i = 0; i < G->vexnum; i++){
        visited[i] = 0;
    }
    for(int i = 0; i < G->vexnum; i++){
        if(visited[i] == 0){
            DFS_AGraph(G, i);
        }
    }
}
```

- 领接矩阵

代码：

```
//深度优先遍历领接矩阵 递归实现 DFS
int visited[maxsize] = {0};

void DFS_MGraph(MGraph *G, int v){
    int i;

    visited[v] = 1;
    printf("%d   ", G->vex[v]);

    for(i = 0; i < G->vexnum; i++){
        if(G->edge[v][i] == 1 && visited[i] == 0){
            DFS_MGraph(G, i);
        }
    }
}
```

- 非递归遍历
  - 领接表

思路： 这里需要注意的时，我们只需要取当前的节点的第一个未被访问的节点，其他节点后序再访问

代码：

```
//深度优先遍历DFS 非递归实现

int visited[maxsize] = {0};
```

```
void DFS_AGraph2(AGraph *G, int v){
    int stack[maxsize];
    int top = -1, k;
    ArcNode *p = NULL;

    printf("%d  ", G->adjlist[v].data);         //访问节点，并入栈
    stack[++top] = v;
    visited[v] =  1;

    while(top != -1){
        k = stack[top];      //取栈顶元素，然后取其顶点的第一条边
        p = G->adjlist[k].firstarc;

        while(p != NULL && visited[p->adjvex] != 0){//一直往后找到第一个未被访问的节点
            p = p->next;
        }
        if(!p){      //如果该顶点的邻接表都未找到，说明已经全部访问完，出栈
            top--;
        }
        else{           //找到了则入栈
            printf("%d  ", G->adjlist[p->adjvex].data);
            stack[++top] = p->adjvex;
            visited[p->adjvex] = 1;
        }
    }
}
```

- 领结矩阵

代码：

```
//深度优先遍历 非递归算法
int visited[maxsize] = {0};

void DFS_MGraph2(MGraph *G, int v){
    int stack[maxsize];
    int top = -1, i,k;

    visited[v] = 1;
    stack[++top] = v;
    printf("%d ", G->vex[v]);

    while(top != -1){
        k = stack[top];
        for(i = 0; i < G->vexnum; i++){
            if(G->edge[k][i] == 1 && visited[i] == 0){
                stack[++top] = i;
                visited[i] = 1;
                printf("%d ", G->vex[i]);
                break;                    //找到之后跳出循环 ，不然就不是深搜了
            }
        }
```

```
            if(i == G->vexnum) top--;   //这里说明顶点k的所有相邻节点都遍历过，直接弹出栈
        }
    }
```

(2）广度优先搜索

- 领接表

代码：

```
//领接表图的广度优先遍历  BFS
int visited[maxsize] = {0};     //标记访问数组

void BFS_AGraph(AGraph *G, int i){
    int j;
    ArcNode *p;
    int queue[maxsize], front, rear;    //通过队列实现
    front = rear = -1;

    printf("%d  ", G->adjlist[i].data); //访问当前节点并入队
    visited[i] = 1;
    queue[++rear] = i;

    while(rear != front){//当队列不为空时

        j = queue[++front];     //出队并对其边节点进行访问
        p = G->adjlist[j].firstarc;
        while(p){
            if(visited[p->adjvex] == 0){
                printf("%d  ", G->adjlist[p->adjvex].data);
                queue[++rear] = p->adjvex;
                visited[p->adjvex] = 1;
            }
            p = p->next;
        }
    }

}

//如果是非连通图
void BFS_AGraph_Travel(AGraph *G){
    int i = 0;
    for(i = 0; i < G->vexnum; i++){
        visited[i] = 0;
    }
    for(int i = 0; i < G->vexnum; i++){
        if(visited[i] == 0){
            BFS_AGraph(G, i);
        }
    }
}
```

- 领结矩阵

代码：

```
//广度优先遍历 领接矩阵 BFS
void BFS_MGraph(MGraph *G, int v){
    int queue[maxsize];
    int rear = -1, front = -1;
    int j,i;

    printf("%d   ", G->vex[v]);      //访问节点并入队
    visited[v] = 1;
    queue[++rear] = v;

    while(rear != front){//队列不为空时
        i = queue[++front];      //取队首节点，然后不断访问其所在行不为0且未访问过的值
        for(j = 0; j < G->vexnum; j++){
            if(G->edge[i][j] == 1 && visited[j] == 0){
                visited[j] = 1;
                printf("%d   ", G->vex[j]);
                queue[++rear] = j;
            }
        }
    }
}
```

4、拓扑排序

概念

AOV网：**顶点**表示**活动**，**有向边**表示活动之间的先后关系

拓扑序列： 把AOV网中的所有节点的顶点排成一个线性序列，使每个活动的所有前驱节点都排在该活动的前边，**求不出拓扑排序说明有环**，拓扑排序未必唯一

拓扑排序实现

- 领接表

思路：

1. 首先利用领接表求各个顶点的入度indegree，print数组存储拓扑序列
2. 利用栈，将当前节点入度为0的节点入栈
3. 当栈不为空时，循环遍历栈中元素，找到与栈中相连的节点，将其的相连的节点入度减1，并将入度为0的元素入栈
4. 判断print中数组元素个数是否与领结表元素相同，相同则是拓扑排序，不相同存在环

代码：

```
//拓扑排序

//小算法，求各顶点入度
```

```
void Count_InDegree_AGraph(AGraph *G, int *indegree){
    ArcNode *p;          //用来扫描每个顶点所发出的边
    int i, j,sum=0,k;
    for(i = 0; i < G->vexnum; i++){//计算每个顶点的入度
        sum = 0;
        for(j = 0; j < G->vexnum; j++){ //计算k这个节点的入度  遍历整个领接表

            p = G->adjlist[j].firstarc;
            while(p != NULL){
                if(p->adjvex == i){     //在某个顶点找到了，sum++,并跳出循环
                    sum++;
                    break;
                }
                p = p->next;            //遍历该节点的所有弧
            }
        }
        indegree[i] = sum;          //遍历完整个图后，sum便是该顶点的入度
    }
}

bool TopologicalSort(AGraph *G, int print[]){
    int stack[maxsize];
    int indegree[G->vexnum]={0};
    ArcNode *p = NULL;
    int top = -1, count = 0,i = 0;      //count计算当前已经输出的顶点数

    Count_InDegree_AGraph(G,indegree);//计算入度，并将入度为0的顶点入栈
    for(i = 0; i < G->vexnum; i++){
        if(indegree[i] == 0){
            stack[++top] = i;
        }
    }

    while(top != -1){
        i = stack[top--];           //取当前栈顶元素并出栈
        print[count++] = i;         //记录当前拓扑序列

        p = G->adjlist[i].firstarc;
        while(p != NULL){           //  判断当前节点所连接的节点入度减1，并且将入度为0的
元素入栈
            if(!(--indegree[p->adjvex])){
                stack[++top] = p->adjvex;
            }
            p = p->next;
        }
    }

    if(count < G->vexnum){  //  排序失败，存在回路
        return false;
    }
    else{                       //排序成功
        return true;
    }
}
```

- 领接矩阵

代码：

```
// 拓扑排序

//小算法：求各顶点的入度
void Count_InDegree_MGraph(MGraph *G, int indegree[]){
    int i,j;
    memset(indegree,0,sizeof(indegree));
    for(i = 0; i < G->vexnum; i++){
        for(j = 0; j < G->vexnum; j++){
            if(G->edge[j][i] == 1)
                indegree[i]++;
        }
    }
}

bool TopologicalSort(MGraph *G,int* print){
    int stack[maxsize];
    int top = -1, i,count = 0,k;
    int indegree[G->vexnum]={0};

    Count_InDegree_MGraph(G, indegree);
    for(i = 0; i < G->vexnum; i++){
        if(indegree[i] == 0){
            stack[++top] = i;
        }
    }

    while(top != -1){
        k = stack[top--];
        print[count++] = k;
        for(i = 0; i < G->vexnum; i++){
            if(G->edge[k][i] == 1){
                indegree[i]--;
                if(indegree[i] == 0){
                    stack[++top] = i;
                }
            }
        }
    }

    if(count == G->vexnum) return true;
    else return false;
}
```

5、关键路径

概念

AOE网： 有向边表示一个工程中的各项**活动**，**边上的权值**表示活动的**持续时间**，**顶点表示事件。**

源点：表示整个工程的开始（入度为0）

汇点：表示整个工程的结束（出度为0）

关键路径：从源点到汇点的最长路径，关键路径决定了完成整个工程的所需时间

路径长度：路径上的各边权值之和

关键活动：关键路径上的活动

事件Vj最早发生时间ve(j):　从源点v0到vj的最长路径长度

事件Vj最晚发生时间vl(j):　保证汇点的最早发生时间不推迟的前提下事件vj允许的最迟开始时间，等于ve（n-1)减去vj到vn-1的最长路径长度

活动ai最早发生时间e(ai)： e(ai) = ve(j)

事件ai最早发生时间l(ai): l(ai) = vl(k)-weight(<j,k>

求关键路径

1. 对AOE网进行拓扑排序，按拓扑排序求出各顶点事件的最早发生时间ve
2. 按拓扑逆序求出各顶点事件的最迟发生时间vl
3. 根据ve和vl的值，求出各活动的最早开始时间e(i)与最迟开始时间l(i),**如果e(i)=l(i),则i是关键活动**

6、最短路径问题

(1) 单源最短路径

BFS 无权图

思路：

- 整理思路与BFS思路一致，先将源点入队列，然后对相邻的边进行遍历，随之入队
- 此外增加三个数组
  - dist[]保存源点到某点的最短路径
  - path[]保存当前节点的最短路径中前一个节点的信息
  - visited[]判断当前节点是否并入最短路径
- 整体实现方法为：
  1. 初始化数组dist,path,visited
  2. 源点能够到其他节点的边进行判断，更新dist，path,visited
  3. 源点入队列，循环判断
  4. 取队头元素，判断其相邻的边是否存在，如果存在是否被访问过，如果都没有，则并入最短路径，更新dist,path,visited数组，并且将当前节点入队，继续循环判断

领接表实现

```
//BFS实现单源最短路径
void BFS_MIN_Distance(AGraph *G, int v, int dist[], int path[]){
    int visited[G->vexnum] = {0};
    int queue[G->vexnum];
    int front = -1, rear = -1;
    int i,j;
    ArcNode *p;

    for(i=0; i<G->vexnum; i++){      //初始化dist和path数组
        dist[i] = maxnum;
```

```
        path[i] = -1;
    }

    dist[v] = 0;                    //源点入队列
    queue[++rear] = v;
    visited[v] = 1;

    while(rear != front){

        j = queue[++front];      //队头出队
        p = G->adjlist[j].firstarc; //取领接表的第一个节点

        while(p){                           //遍历
            if(visited[p->adjvex] == 0){    //未被访问加入队列，加入最短路径
                visited[p->adjvex] = 1;
                dist[p->adjvex] = dist[j] + 1;
                path[p->adjvex] = j;
                queue[++rear] = p->adjvex;
            }
            p= p->next;                //继续广度优先遍历
        }
    }
}
```

领接矩阵实现

```
//最短路径算法，BFS实现
void BFS_MIN_Distance1(MGraph *G, int v, int dist[], int path[]){
    int visited[G->vexnum] = {0};
    int queue[G->vexnum];
    int front = -1, rear = -1;
    int i,j,k;

    for(i = 0; i < G->vexnum; i++){ //初始化dist和path数组
        dist[i] = maxnum;
        path[i] = -1;
    }

    dist[v] = 0;                //顶点v入队
    queue[++rear] = v;
    visited[v] = 1;

    while(rear != front){
        k = queue[++front];      //出队
        for(i=0; i < G->vexnum; i++){
            if(G->edge[k][i] != 0 && visited[i] == 0){  //存在边，并且未被访问，加入队列以及最短路径
                queue[++rear] = i;
                dist[i] = dist[k]+1;
                path[i] = k;
                visited[i] = 1;
            }
        }
    }
}
```

Dijstra带权图\无权图 对于负权图无法求解

思路：

- 增加三个数组
  - dist[]保存源点到某点的最短路径
  - path[]保存当前节点的最短路径中前一个节点的信息
  - visited[]判断当前节点是否并入最短路径
- 实现思路：
  1. 先初始化dist，path，visited数组
  2. 然后将源点的邻接边信息加入到dist数组中并跟新path数组，标记当前节点已经并入最短路径
  3. 循环n-1，先从dist数组中选择最小的路径并入最短路径当中
  4. 然后在加入前面选择的边之后是否需要更新当前的dist数组，和path数组

领接表实现

```
// dijstra单源最短路径，领接表实现
void Dijstra_AGraph(AGraph *G, int v, int dist[], int path[]){
    int visited[G->vexnum] = {0};
    int i,j,k,min,temp;
    ArcNode *p = G->adjlist[v].firstarc;

    for(i=0; i<G->vexnum; i++){     //初始化
        dist[i] = maxnum;
        path[i] = -1;
    }

    while(p!=NULL){                 //第一次先将顶点v的相邻节点并入最短路径
        dist[p->adjvex] = p->info;
        path[p->adjvex] = v;
        p = p->next;
    }
    visited[v] = 1;

    for(i=0; i<G->vexnum-1; i++){       //然后再依次判断n-1次
        min = maxnum;
        for(j = 0; j < G->vexnum; j++){ //在目前最短路径中找到最小值
            if(visited[j]==0 && dist[j] < min){
                min = dist[j];
                k = j;
            }
        }
        visited[k] = 1;                 //并入最短路径
        p = G->adjlist[k].firstarc;     //更新权值
        while(p){
            temp  = p->adjvex;
            if(visited[temp]==0 && dist[k]+p->info < dist[temp]){
                dist[temp] = dist[k]+p->info;
                path[temp] = k;
            }
            p = p->next;
```

```
        }
    }
}
```

领接矩阵实现

```
//最短路径算法，dijstra算法实现
void Dijstra(MGraph *G, int v, int dist[], int path[]){
    int visited[G->vexnum] = {0};
    int i,j,k,min;

    for(i = 0; i < G->vexnum; i++){     //初始化dist数组和path数组
        dist[i] = G->edge[v][i];
        if(dist[i] < max){
            path[i] = v;
        }
        else{
            path[i] = -1;
        }
    }

    visited[v] = 1;
    for(i = 0; i < G->vexnum-1; i++){           //遍历n-1次
        min = max;
        for(j = 0; j < G->vexnum; j++){         //从dist数组中找到最小并入最短路径
            if(visited[j] == 0 && dist[j] < min){
                min = dist[j];
                k = j;
            }
        }
        visited[k] = 1;
        for(j = 0; j < G->vexnum; j++){         //判断是否需要更新路径
            if(visited[j]==0 && dist[k]+G->edge[k][j] < dist[j]){
                dist[j] = dist[k]+G->edge[k][j];
                path[j] = k;
            }
        }
    }
}
```

（2）各顶点之间最短路径

Floy 不能解决带有负权回路的图

思路：

1. 增加两个二维辅助数组path[][]，A[][]，path最短路径之间经过的节点，A存放最短路径
2. 初始化path和数组A
3. 每次增加一个节点，判断是否需要更新路径，三重循环

代码：

```
//Floy算法实现
```

```
void Floy(MGraph *G, int path[][maxsize], int A[][maxsize]){
    int i,j,k;
    for(i = 0; i < G->vexnum; i++){      //初始化
        for(j = 0; j < G->vexnum; j++){
            path[i][j] = -1;
            A[i][j] = G->edge[i][j];
        }
    }
    for(k=0;k < G->vexnum; k++){        //floy算法关键步骤
        for(i = 0; i < G->vexnum; i++){
            for(j = 0; j < G->vexnum; j++){
                if(A[i][j] > A[i][k]+A[k][j]){      //增加k作为中间节点是否会增加最短路径长度
                    path[i][j] = k;
                    A[i][j] = A[i][k]+A[k][j];
                }
            }
        }
    }
}
```

7、最小支撑树

(1) Prim算法

思路:

1. 基本与dijstra算法一致，先确定两个数组，一个visited，lowcost数组
2. 先对起点所连接的边进行初始化赋值，然后遍历n-1次，分别从lowcost中找到最小边权值，然后加入到最小生成树之中，再判断是否需要对最小生成树进行更新。
3. 更新的判断条件与最短路径的dijstra算法不一样，这里只需要比较的权值大小即可。

代码:

邻接表实现

```
// Prim 算法，最小支撑树，领结表实现
void Prim_AGraph(AGraph *G, int v, int &sum){
    int lowcost[G->vexnum];
    int visited[G->vexnum];
    int i,j,k,min,temp;

    for(i=0; i < G->vexnum; i++){       //初始化lowcost数组 和visited数组
        lowcost[i] = maxnum;
        visited[i] = 0;
    }

    ArcNode *p = G->adjlist[v].firstarc;
    while(p!=NULL){              //将源点所连接的边赋值给lowcost数组
        temp = p->adjvex;
        lowcost[temp] = p->info;
        p = p->next;
    }
    visited[v] = 1;
```

```
    for(i = 0; i < G->vexnum-1; i++){
        min = maxnum;
        for(j=0; j <G->vexnum; j++){
            if(visited[j]==0 && lowcost[j] < min){
                min = lowcost[j];
                k = j;
            }
        }
        sum+=lowcost[k];
        visited[k] = 1;
        p = G->adjlist[k].firstarc;
        while(p!=NULL){
            int temp = p->adjvex;
            if(visited[temp]==0 && p->info < lowcost[temp]){
                lowcost[temp] = p->info;
            }
            p = p->next;
        }
    }
    printf("最小支撑树的权值之和为%d",sum);
}
```

领接矩阵实现

```
//最小生成树算法，prim算法实现 领接矩阵 基本和dijstra一样
void Prim(MGraph *G, int v, int &sum){
    int lowcost[G->vexnum];
    int visited[G->vexnum];
    int i,j,k,min;
    sum = 0;
    for(i=0; i < G->vexnum; i++){       //初始化lowcost数组 和visited数组
        lowcost[i] = G->edge[v][i];
        visited[i] = 0;
    }
    visited[v] = 1;

    for(i = 0; i < G->vexnum-1; i++){   //遍历n-1次找到n-1条最短边构成最小支撑树
        min = maxnum;
        for(j=0; j < G->vexnum; j++){       //在目前的lowcost中找到最小的权值加入最小支
撑树
            if(visited[j]==0 && lowcost[j] < min){
                min = lowcost[j];
                k = j;
            }
        }
        sum += lowcost[k];                  //更新最小支撑树的权值和，标记已经加入最小支
撑树的顶点
        visited[k] = 1;
        for(j=0; j < G->vexnum; j++){       //更新lowcost数组
            if(visited[j]==0 && G->edge[k][j] < lowcost[j]){
                lowcost[j] = G->edge[k][j];
            }
        }
    }
    printf("当前最小生成树的权值之和为%d", sum);
```

```
}
```

(2) kruskal算法

思路:

1. 并查集的思路，首先创建并查集，构建寻找根节点的函数，以及自定义比较函数
2. 先对并查集中的边权值按照从小到大的方式进行排序，排好序后根据边的个数进行遍历
3. 每次选取边权值最小的节点加入连通分量中，如果起点和终点的连通分量一致，说明加入这条边存在环，不应该加入，所以直接跳过，如果起点和终点的连通分量不一致，即两个点的根节点不同，则说明加入当前较短边不存在环，可以加入

代码:

```
//最小生成树  kruskal
typedef struct node{    //定义一条边，起点、终点和权值
    int from;
    int to;
    int weight;
}node;

typedef struct EdgeGraph{
    node edge[maxsize];     //图
    int vnum, anum;         //顶点数和边数
}EdgeGraph;

int find(int parent[], int x){  //查找根节点
    int  t = x;
    while(parent[t]!=-1){
        t = parent[t];
    }
    return t;
}

int cmp(const void *a, const void *b){      //qsort自定义排序，这里是递增排序
    return ((node*)a)->weight - ((node*)b)->weight;
}

int kruskal(EdgeGraph *G, int &sum){    //算法实现
    int parent[maxsize];
    int i,a,b;
    for(i=0; i < maxsize; i++){
        parent[i] = -1;
    }

    qsort(G->edge, G->anum, sizeof(node),cmp); //对边进行递增排序
    for(i=0; i < G->anum; i++){
        a = find(parent,G->edge[i].from);   //找到所在生成树的根节点
        b = find(parent,G->edge[i].to);
        if(a!=b){                           //两个连通分量
            parent[a] = b;
            sum+=G->edge[i].weight;         //合并，并计算最小连通分量
        }
    }
    printf("最小生成树的权值为：%d",sum);
```

```
}
```

###