

# 第一节 概述

2016年9月1日 16:35

## 8.1.1 邻接 + 关联

$v$ 与 $v$  邻接  
 $v$ 与 $e$  关联  
不讨论自环

## 8.1.2 无向 + 有向

有向图、无向图、混合图

## 8.1.3 路径 + 环路

简单路径  
简单环路  
欧拉环路、哈密尔顿环路

## 第二节 邻接矩阵

2016年9月1日 16:43

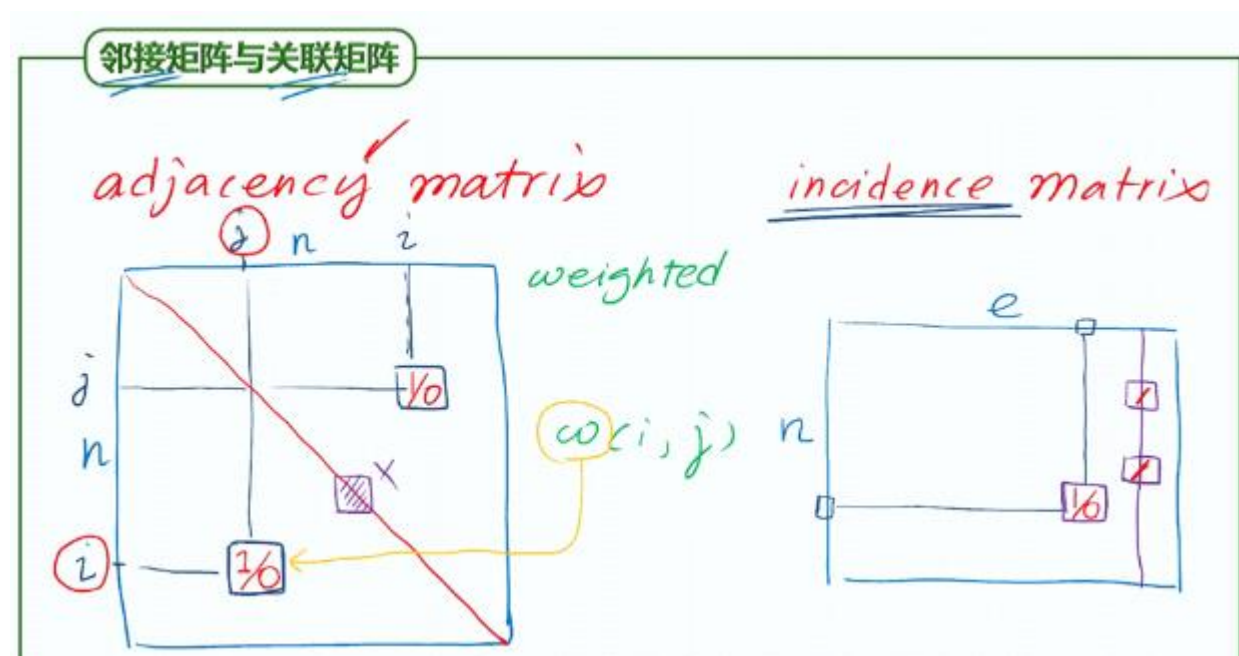
### 8.2.1 接口

**Graph模板类**

```
template <typename Tv, typename Te> class Graph { //顶点类型、边类型
private:
    void reset() { //所有顶点、边的辅助信息复位
        for (int i = 0; i < n; i++) { //顶点
            status(i) = UNDISCOVERED; dTime(i) = fTime(i) = -1;
            parent(i) = -1; priority(i) = INT_MAX;
            for (int j = 0; j < n; j++) //边
                if (exists(i, j)) status(i, j) = UNDETERMINED;
        }
    }
public:
    /* ... 顶点操作、边操作、图算法: 无论如何实现, 接口必须统一 ... */
} //Graph
```

尽管我们现在还不清楚Graph

### 8.2.2 邻接矩阵 + 关联矩阵



邻接矩阵的带权图中可令, T/F换为边的权重。

### 8.2.3 实例

### 8.2.4 顶点和边

顶点类:

**Vertex**

```
❖ typedef enum { UNDISCOVERED, DISCOVERED, VISITED } VStatus;
❖ template <typename Tv> struct Vertex { //顶点对象 (并未严格封装)
    Tv data; int inDegree, outDegree; //数据、出入度数
    VStatus status; //(如上三种) 状态
    int dTime, fTime; //时间标签 ←
    int parent; //在遍历树中的父节点
    int priority; //在遍历树中的优先级 (最短通路、极短跨边等)
    Vertex( Tv const & d ) : //构造新顶点
        data(d), inDegree(0), outDegree(0), status(UNDISCOVERED),
        dTime(-1), fTime(-1), parent(-1),
        priority(INT_MAX) {}
};
```

边类:

**Edge**

```
❖ typedef
    enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD }
    EStatus;
❖ template <typename Te> struct Edge { //边对象 (并未严格封装)
    ✓ Te data; //数据
    ✓ int weight; //权重
    EStatus status; //类型
    Edge( Te const & d, int w ) : //构造新边
        data(d), weight(w), status(UNDETERMINED) {}
};
```

### 8.2.5 邻接矩阵

## GraphMatrix

```

❖ template <typename Tv, typename Te> class GraphMatrix : public Graph<Tv, Te> {
private:
    ✓ Vector< Vertex<Tv> > V; //顶点集
    ✓ Vector< Vector< Edge<Te>* > > E; //边集
public:
    /* 操作接口：顶点相关、边相关、... */
    GraphMatrix() { n = e = 0; } //构造
    ~GraphMatrix() { //析构
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                delete E[j][k]; //清除所有动态申请的边记录
    }
};

```

边集：为向量的向量（二维向量），即为邻接矩阵

### 8.2.6 顶点静态操作

#### 顶点操作

❖ 对于任意顶点  $i$ ，如何枚举其所有的邻接顶点 neighbor？

❖ int nextNbr(int  $i$ , int  $j$ ) { //若已枚举至邻居  $j$ ，则转向下一邻居

while ( (-1 <  $j$ ) && !exists( $i$ , -- $j$ ) ); //逆向顺序查找  $O(n)$

return  $j$ ;

} //改用邻接表可提高至  $O(1 + \text{outDegree}(i))$

❖ int firstNbr(int  $i$ ) {

return nextNbr( $i$ ,  $n$ );

} //首个邻居

### 8.2.7 边操作

边的插入

## 边插入

```
❖ void insert(Te const& edge, int w, int i, int j) { //插入(i, j, w)

    ✓ if ( exists(i, j) ) return; //忽略已有的边

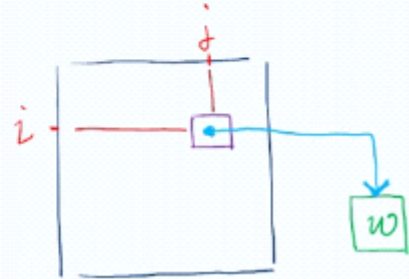
    E[i][j] = new Edge<Te>(edge, w); //创建新边

    e++; //更新边计数

    V[i].outDegree++; //更新关联顶点i的出度

    V[j].inDegree++; //更新关联顶点j的入度

}
```



边的删除

## 边删除

```
❖ Te remove(int i, int j) { //删除顶点i和j之间的联边 ( exists(i, j) )

    Te eBak = edge(i, j); //备份边(i, j)的信息

    delete E[i][j]; E[i][j] = NULL; //删除边(i, j)

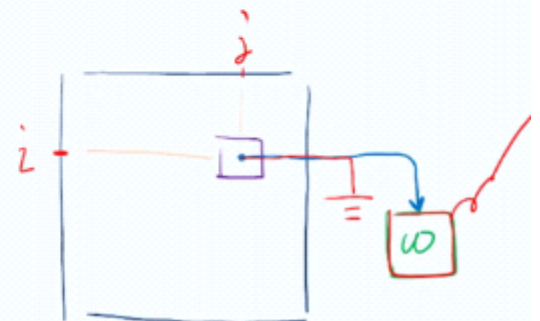
    ✓ e--; //更新边计数

    V[i].outDegree--; //更新关联顶点i的出度

    V[j].inDegree--; //更新关联顶点j的入度

    return eBak; //返回被删除边的信息

}
```

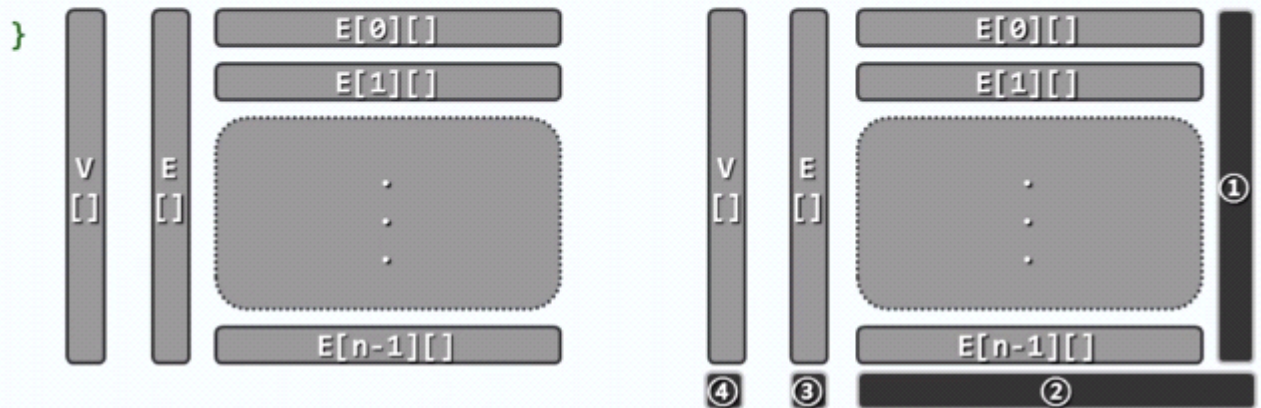


## 8.2.8 顶点动态操作

顶点插入



```
❖ int insert(Tv const & vertex) { //插入顶点, 返回编号
    for (int j = 0; j < n; j++) E[j].insert(NULL); n++; //①
    E.insert( Vector< Edge<Te>* >(n, n, NULL) ); //②③
    return V.insert( Vertex<Tv>(vertex) ); //④
}
```



顶点删除

```
❖ Tv remove(int i) { //删除顶点及其关联边, 返回该顶点信息
    for (int j = 0; j < n; j++)
        if (exists(i, j)) //删除所有出边
            { delete E[i][j]; V[j].inDegree--; }
    E.remove(i); n--; //删除第i行
    for (int j = 0; j < n; j++)
        if (exists(j, i)) //删除所有入边及第i列
            { delete E[j].remove(i); V[j].outDegree--; }
    Tv vBak = vertex(i); //备份顶点i的信息
    V.remove(i); //删除顶点i
    return vBak; //返回被删除顶点的信息
}
```

## 8.2.9 综合评价

优点:  
直观, 易于理解实现  
适用范围广泛

❖ 判断两点之间是否存在联边： $O(1)$

❖ 获取顶点的（出/入）度数： $O(1)$

添加、删除边后更新度数： $O(1)$

❖ 扩展性（scalability）：

得益于Vector良好的空间控制策略

空间溢出等情况可“透明地”予以处理

缺点：空间利用率低下

## 第三节 广度优先搜索BFS

2016年9月1日 17:25

### 8.3.1 化繁为简

遍历算法：非线性结构 -> 半线性结构（支撑树）

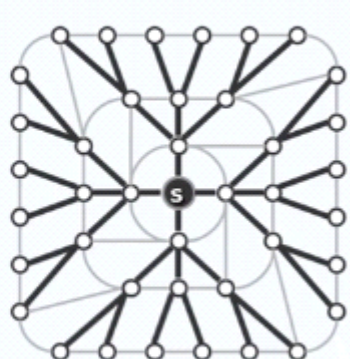
### 8.3.2 策略

**算法**

❖ 始自顶点s的广度优先搜索 **Breadth-First Search**

- ✓ 访问顶点s
- ✓ 依次访问s所有尚未访问的邻接顶点
- ✓ 依次访问它们尚未访问的邻接顶点
- ...
- 如此反复
- ✓ 直至没有尚未访问的邻接顶点

层次



Spanning Tree

### 8.3.3 实现

**Graph::BFS()**

```
❖ template <typename Tv, typename Te> //顶点类型、边类型
void Graph<Tv, Te>::BFS( int v, int & clock ) {
    Queue<int> Q; status(v) = DISCOVERED; Q.enqueue(v); //初始化
    while ( !Q.empty() ) { //反复地
        int v = Q.dequeue();
        dTime(v) = ++clock; //取出队首顶点v, 并
        for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //考察v的每一邻居u
            /* ... 视u的状态, 分别处理 ... */
        status(v) = VISITED; //至此, 当前顶点访问完毕
    }
}
```

### 8.3.4 可能情况



## Graph::BFS()

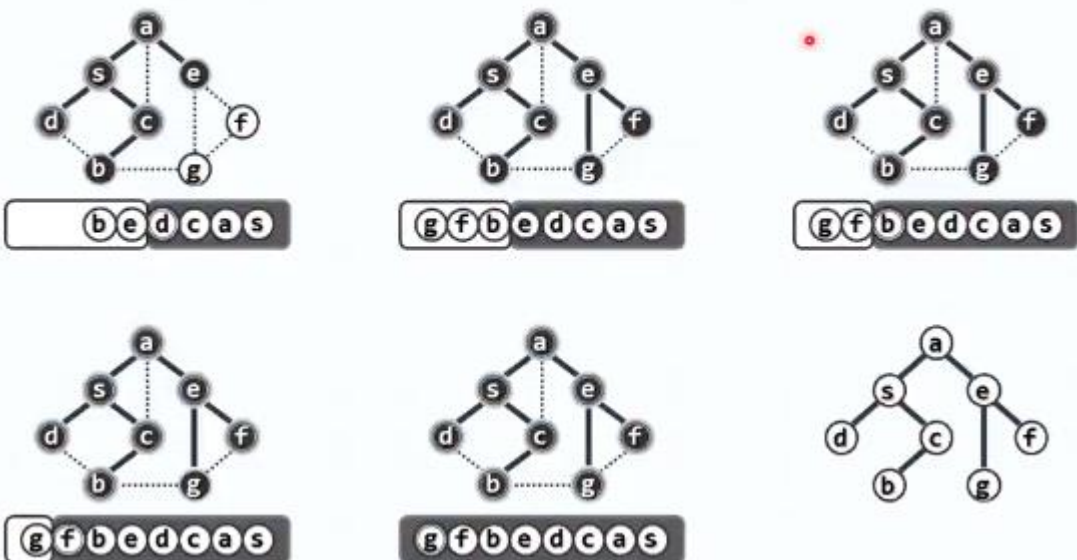
```
❖ while ( !Q.empty() ) { //反复地
    int v = Q.dequeue(); dTime(v) = ++clock; //取出队首顶点v, 并
    for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //考察v的每一邻居u
        if ( UNDISCOVERED == status(u) ) { //若u尚未被发现, 则
            status(u) = DISCOVERED; Q.enqueue(u); //发现该顶点
            status(v, u) = TREE; parent(u) = v; //引入树边
        } else //若u已被发现 (正在队列中), 或者甚至已访问完毕 (已出队列), 则
            status(v, u) = CROSS; //将(v, u)归类于跨边
    status(v) = VISITED; //至此, 当前顶点访问完毕
}
```

我们讲过每次迭代都会首先取出

5

### 8.3.5 实例

#### 实例 (无向图)



### 8.3.6 多连通

### Graph::bfs()

❖ template <typename Tv, typename Te> //顶点类型、边类型

void Graph<Tv, Te>::bfs( int s ) { //s为起始顶点

reset(); int clock = 0; int v = s; //初始化  $\Theta(n + e)$

do //逐一检查所有顶点，一旦遇到尚未发现的顶点

if ( UNDISCOVERED == status(v) ) //累计  $\Theta(n)$

→ BFS( v, clock ); //即从该顶点出发启动一次BFS

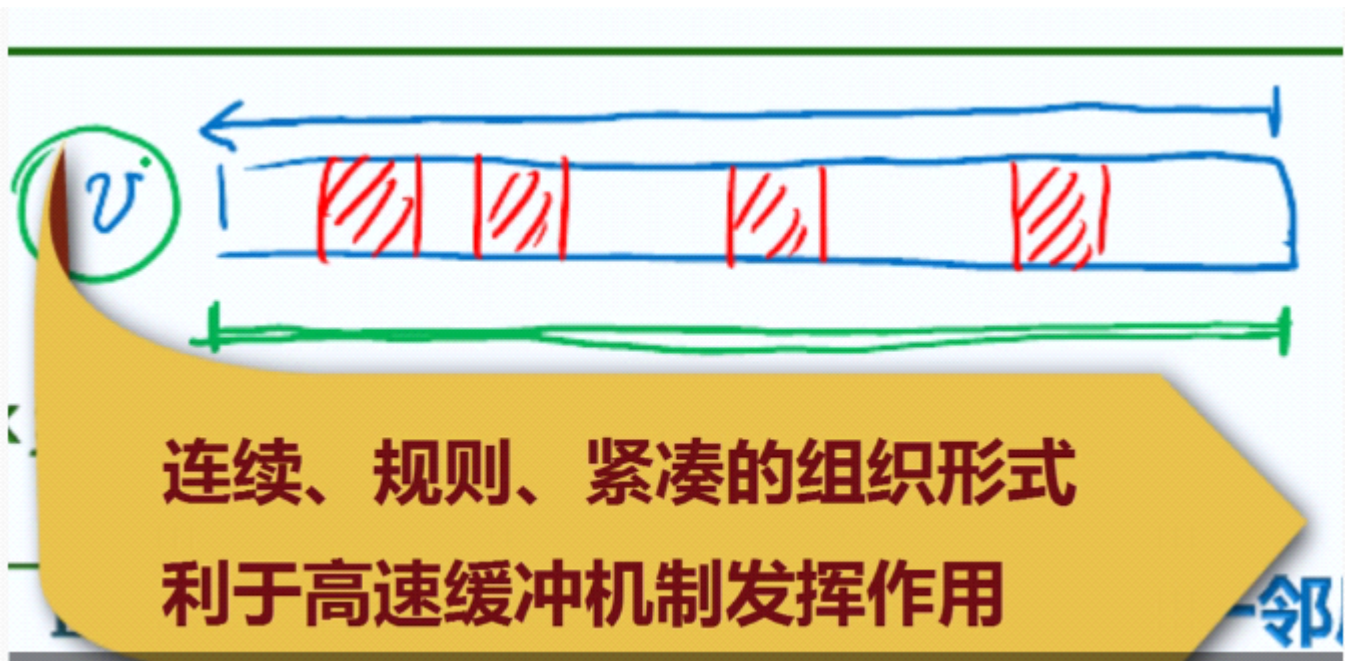
while ( s != ( v = ( ++v % n ) ) );

//按序号访问，故不漏不重

} //无论共有多少连通/可达分量...



### 8.3.7 复杂度



复杂度为:  $O(n^2 + e)$ , 但是在内层循环中的  $n$  极小, 所以可以记为  $O(n + e)$

### 8.3.8 最短路径

BFS所得即为图中两点间的最短通路。

## 第四节 深度优先搜索DFS

2016年9月1日 18:04

### 8.4.1 算法

一条通路走到底，走不通时，逐级回溯，直至完成一棵支撑树

### 8.4.2 框架

```
Graph::DFS()

❖ template <typename Tv, typename Te> //顶点类型、边类型

void Graph<Tv, Te>::DFS( int v, int & clock ) {

    dTime(v) = ++clock; status(v) = DISCOVERED; //发现当前顶点v

    for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v的每一邻居u

        /* ... 视u的状态，分别处理 ... */

        /* ... 与BFS不同，含有递归 ... */

    status(v) = VISITED; fTime(v) = ++clock; //至此，当前顶点v方告访问完毕
```

### 8.4.3 细节

```
Graph::DFS()

❖ for ( int u = firstNbr(v); -1 < u; u = nextNbr(v, u) ) //枚举v所有邻居u

    switch ( status(u) ) { //并视其状态分别处理

        case UNDISCOVERED: //u尚未发现，意味着支撑树可在此拓展

            status(v, u) = TREE; parent(u) = v; DFS(u, clock); break; //递归

        case DISCOVERED: //u已被发现但尚未访问完毕，应属被后代指向的祖先

            status(v, u) = BACKWARD; break;

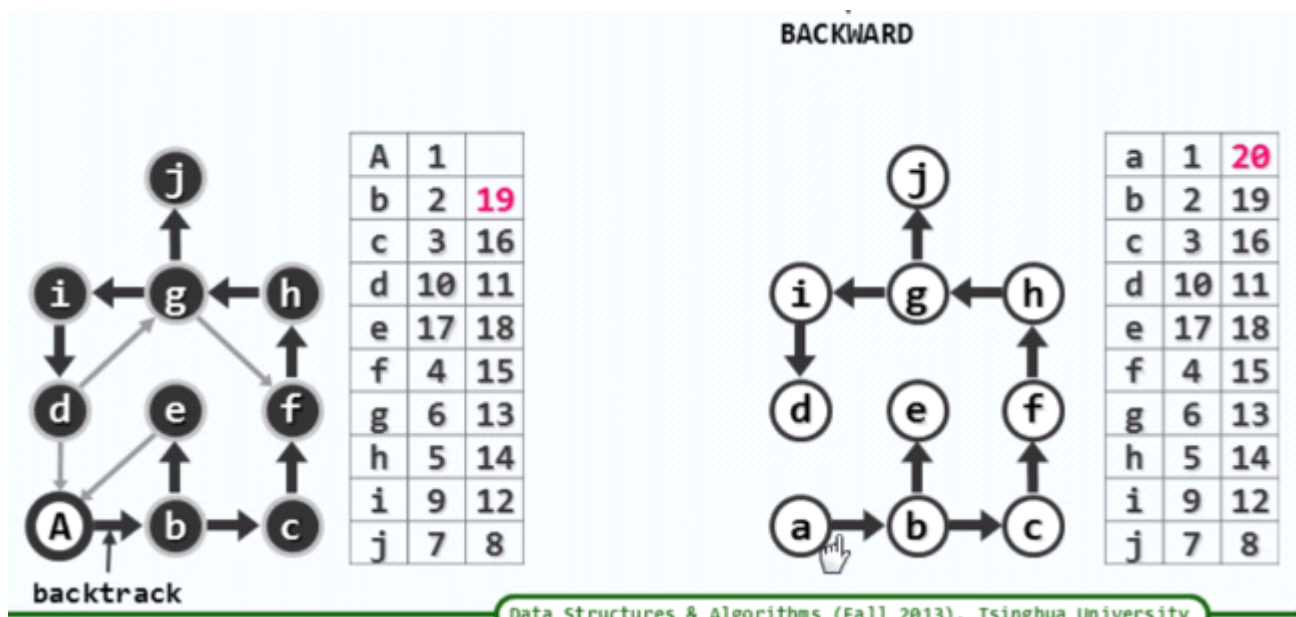
        default: //u已访问完毕 ( VISITED, 有向图 )，则视承袭关系分为前向边或跨边

            status(v, u) = dTime(v) < dTime(u) ? FORWARD : CROSS; break;

    } //switch
```



#### 8.4.4 无向图



#### 8.4.5 有向图

backward: 后代指向祖先

forward: 祖先指向后代

cross: 无直接关系

#### 8.4.6 多可达域

参照BFS算法，在外层套上一个while循环

#### 8.4.7 嵌套引理

dTime与fTime的巨大作用

