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

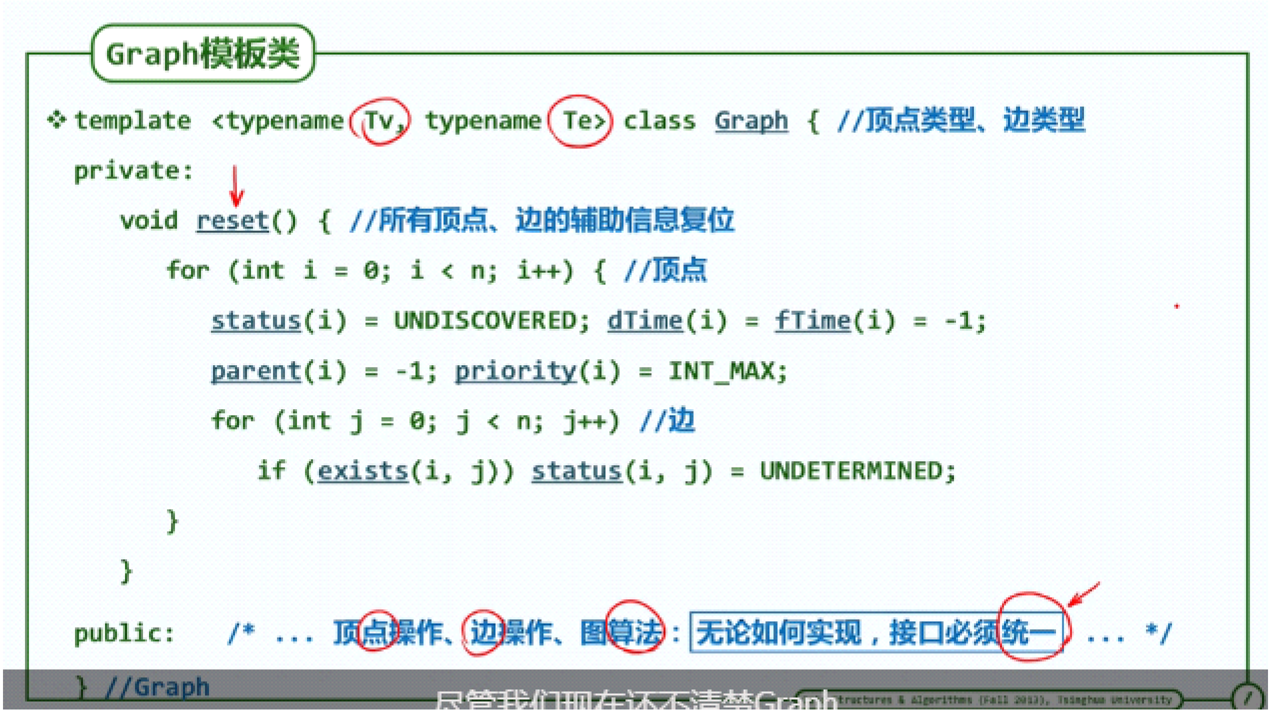
8.1.3 路径 + 环路

简单路径
简单环路
欧拉环路、哈密尔顿环路

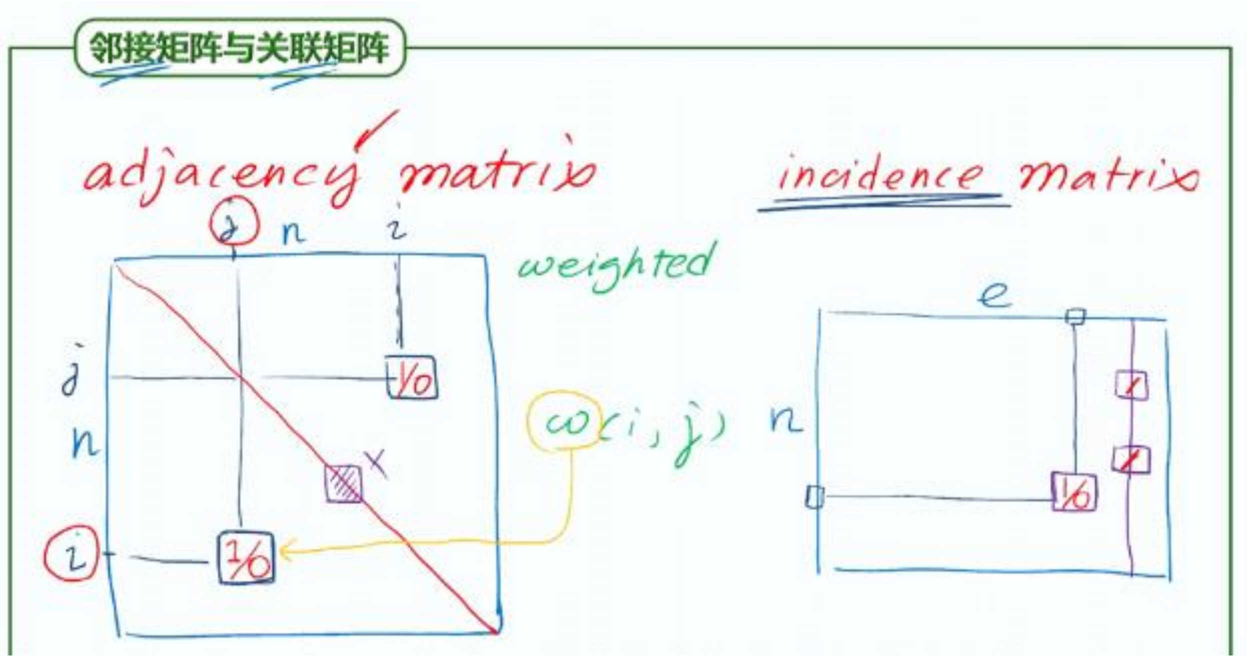
第二节 邻接矩阵

2016年9月1日　　16:43

8.2.1 接口



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

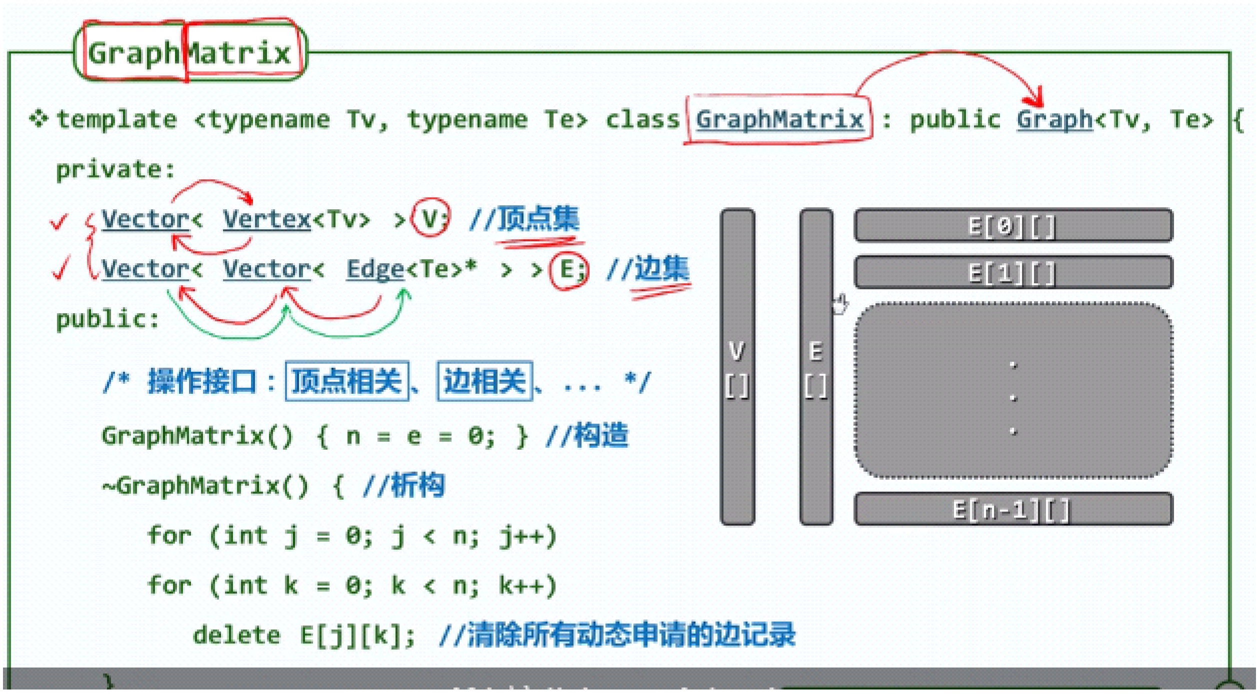
顶点类：

```
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：

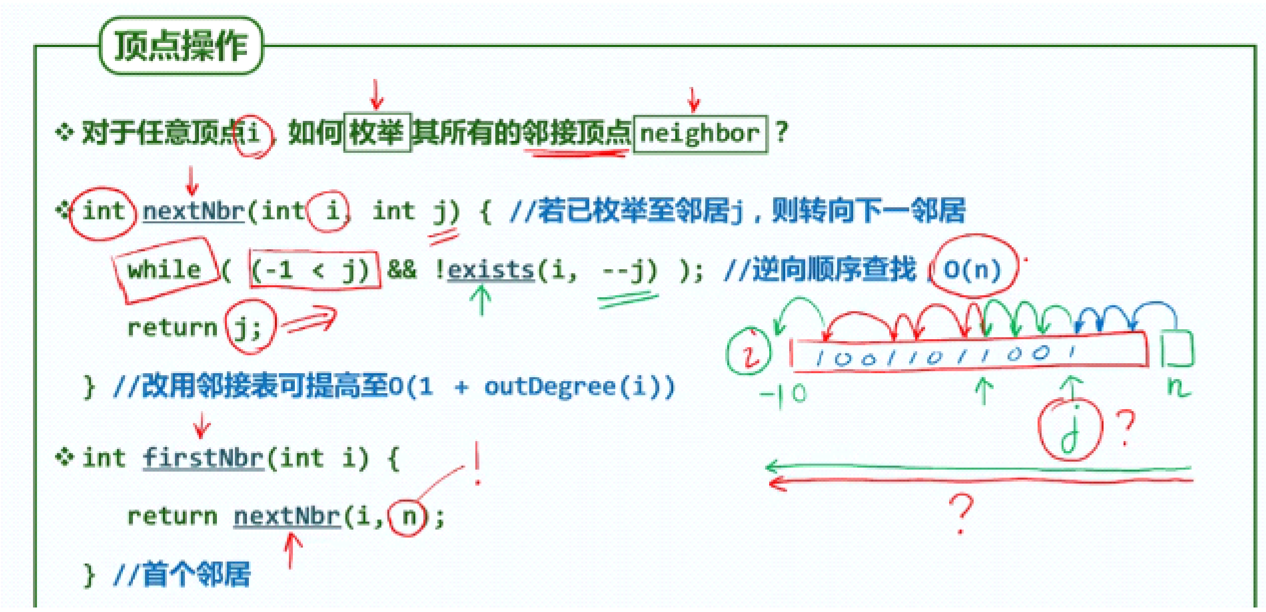
```
Edge
❖ typedef
     enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD }
     EStatus;
❖ template <typename Te> struct Edge { //边对象（并未严格封装）
     Te data; //数据
     int weight; //权重
     EStatus status; //类型
     Edge( Te const & d, int w ) : //构造新边
        data(d), weight(w), status(UNDETERMINED)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

8.2.5 邻接矩阵



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

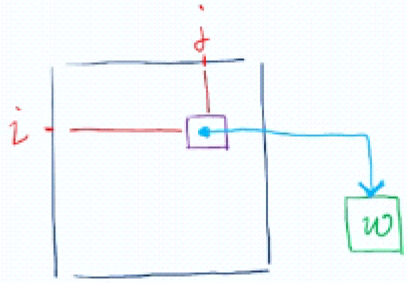


8.2.7 边操作

边的插入

边插入

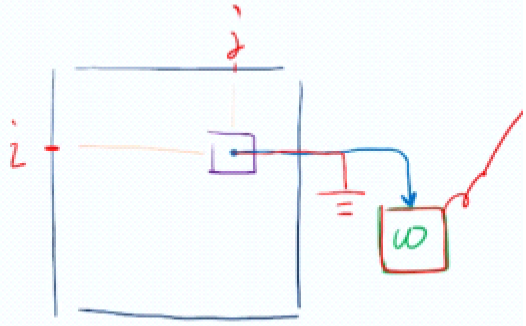
```
❖ void insert(Te const& edge, int w, int i, int j) { //插入(i, j, w)
    if ( exists(i, j) ) return; //忽略已有的边
    E[i][j] = new Edge<Te>(edge, w); //创建新边
    e++; //更新边计数
    V[i].outDegree++; //更新关联顶点i的出度
    V[j].inDegree++; //更新关联顶点j的入度
  }
```



边的删除

边删除

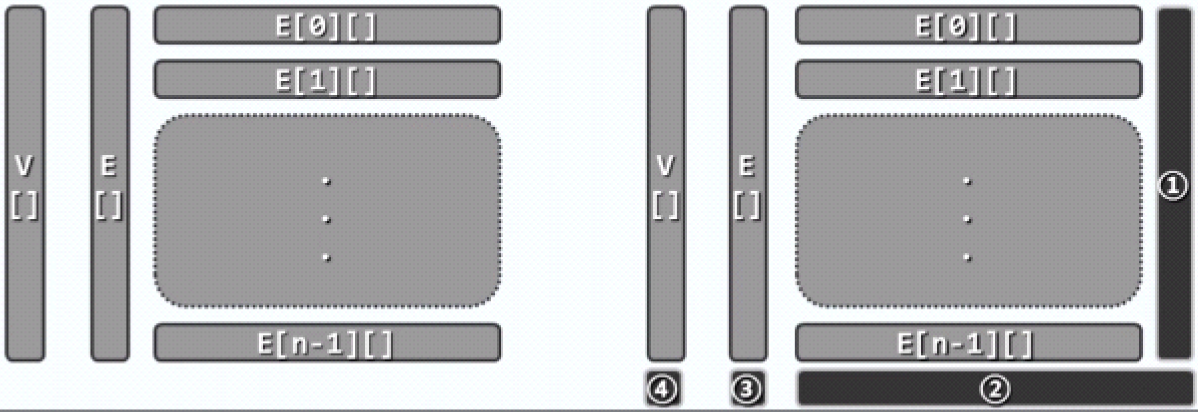
```
❖ Te remove(int i, int j) { //删除顶点i和j之间的联边(exists(i, j))
    Te eBak = edge(i, j); //备份边(i, j)的信息
    delete E[i][j]; E[i][j] = NULL; //删除边(i, j)
    e--; //更新边计数
    V[i].outDegree--; //更新关联顶点i的出度
    V[j].inDegree--; //更新关联顶点j的入度
    return eBak; //返回被删除边的信息
  }
```



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
}
```

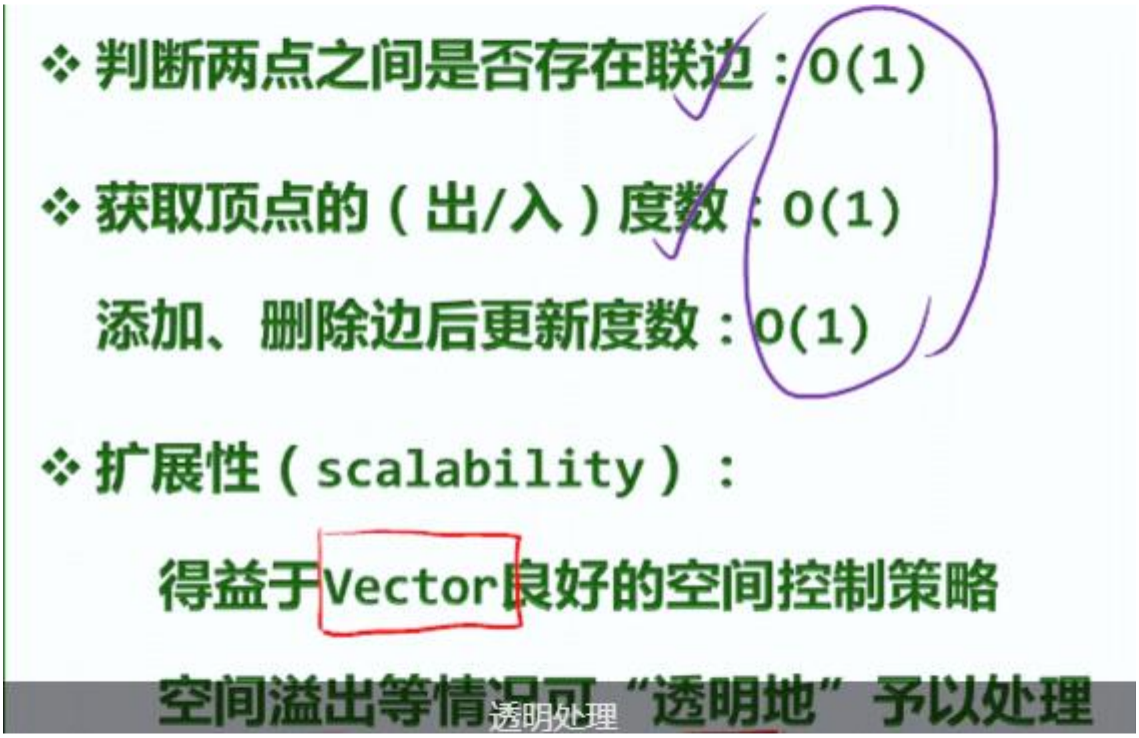


顶点删除

```
❖ 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
透明处理

缺点：空间利用率低下

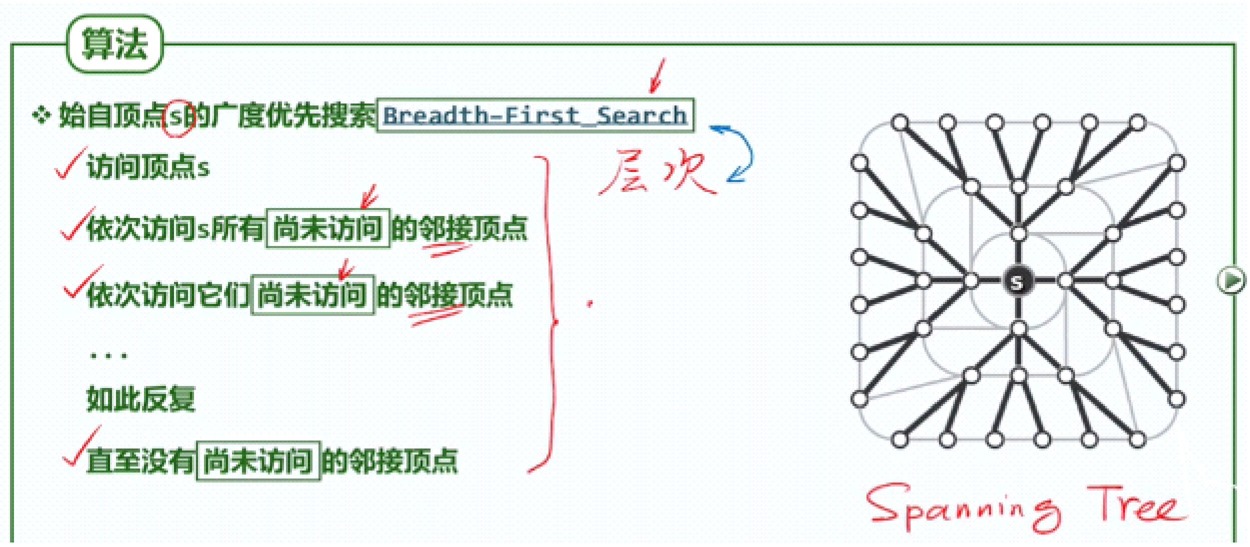
第三节 广度优先搜索BFS

2016年9月1日　　17:25

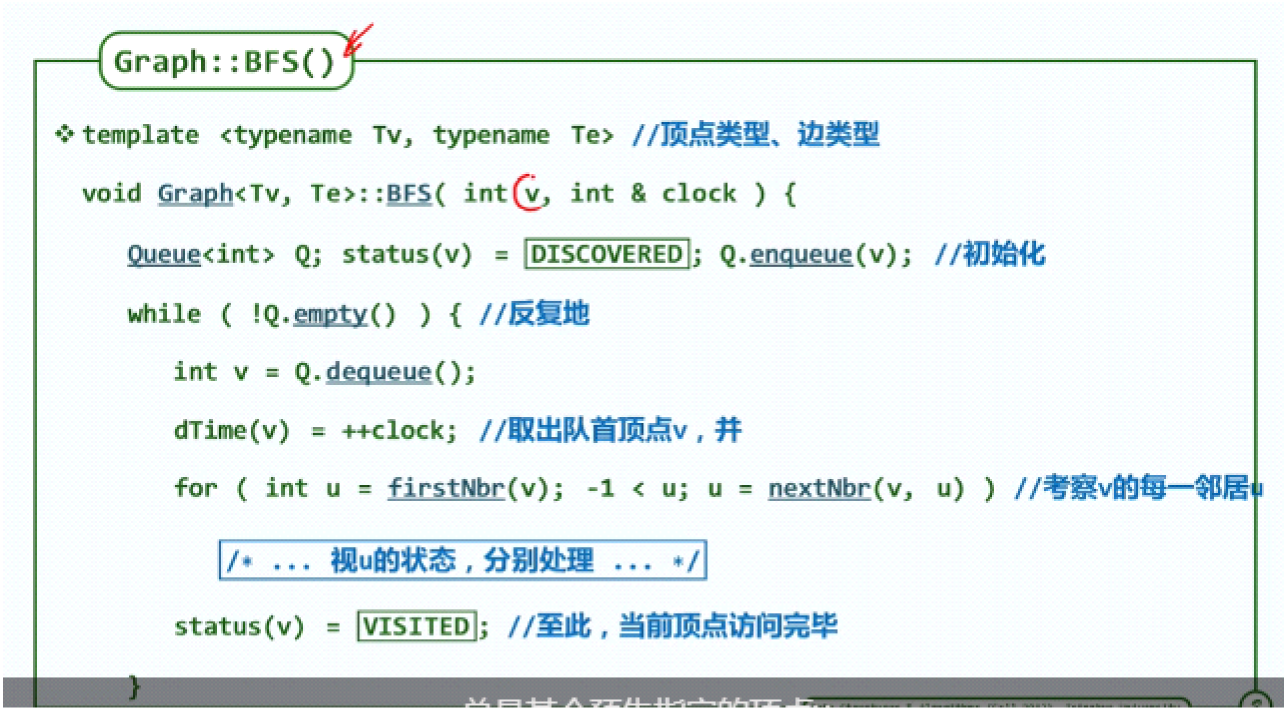
8.3.1 化繁为简

遍历算法：非线性结构 -> 半线性结构（支撑树）

8.3.2 策略

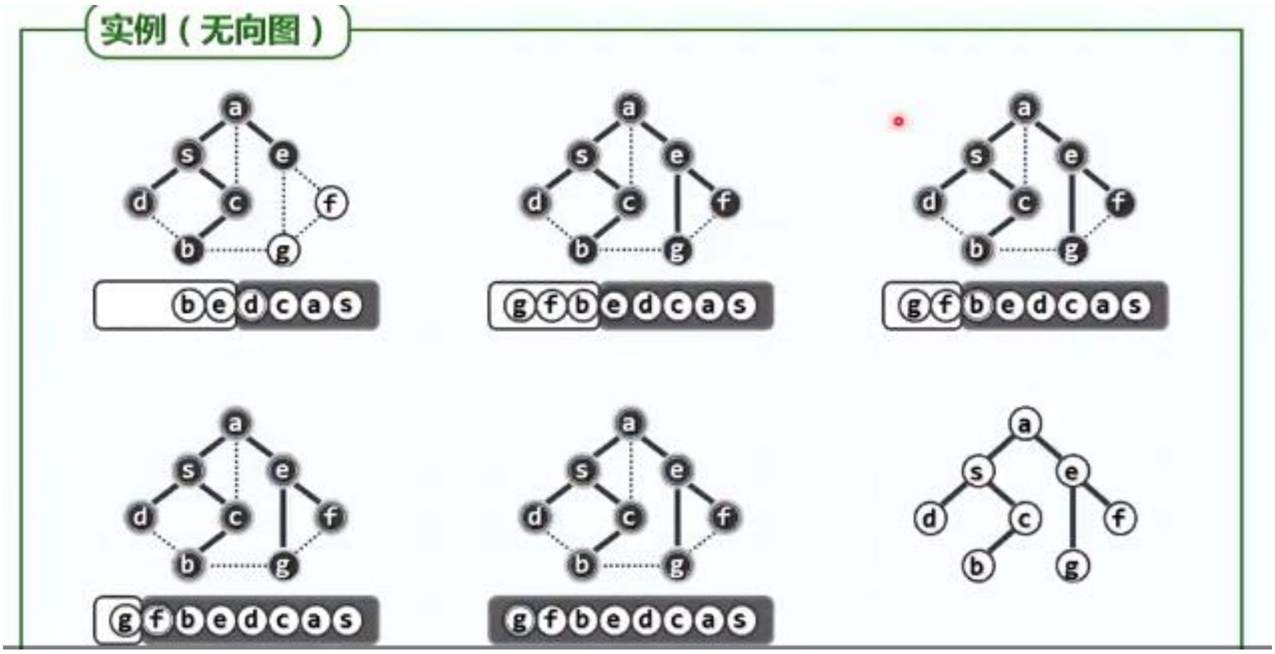


8.3.3 实现

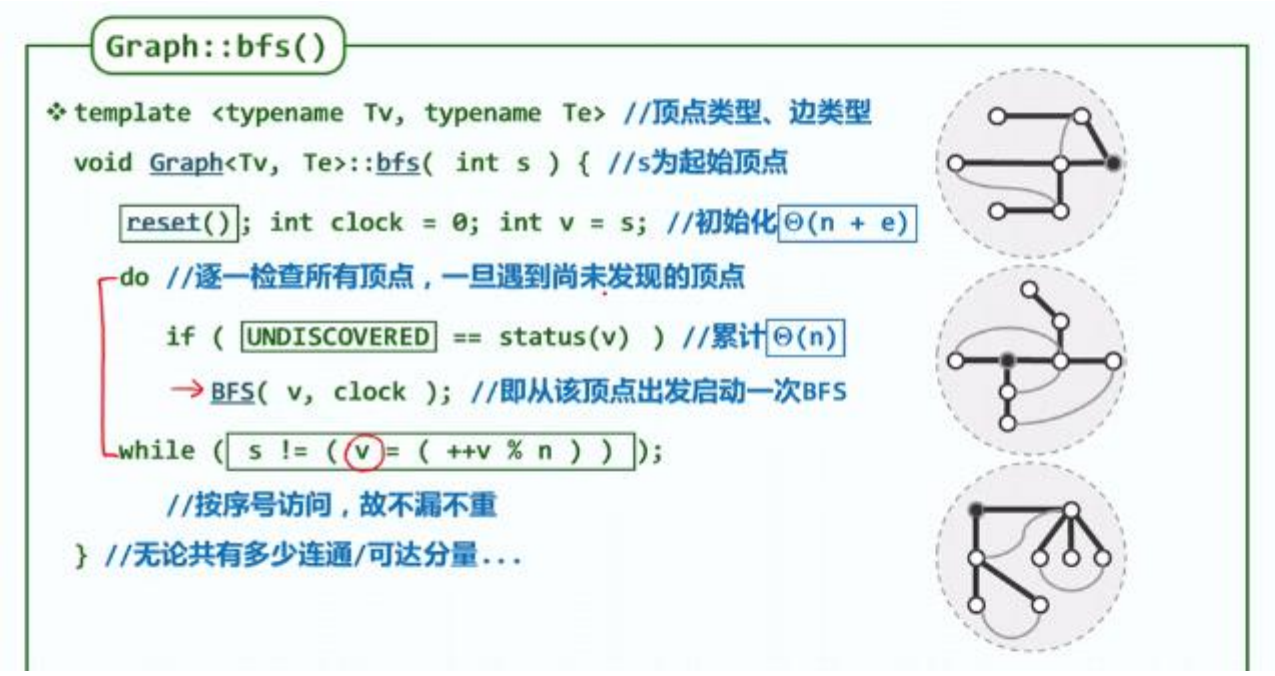


8.3.4 可能情况

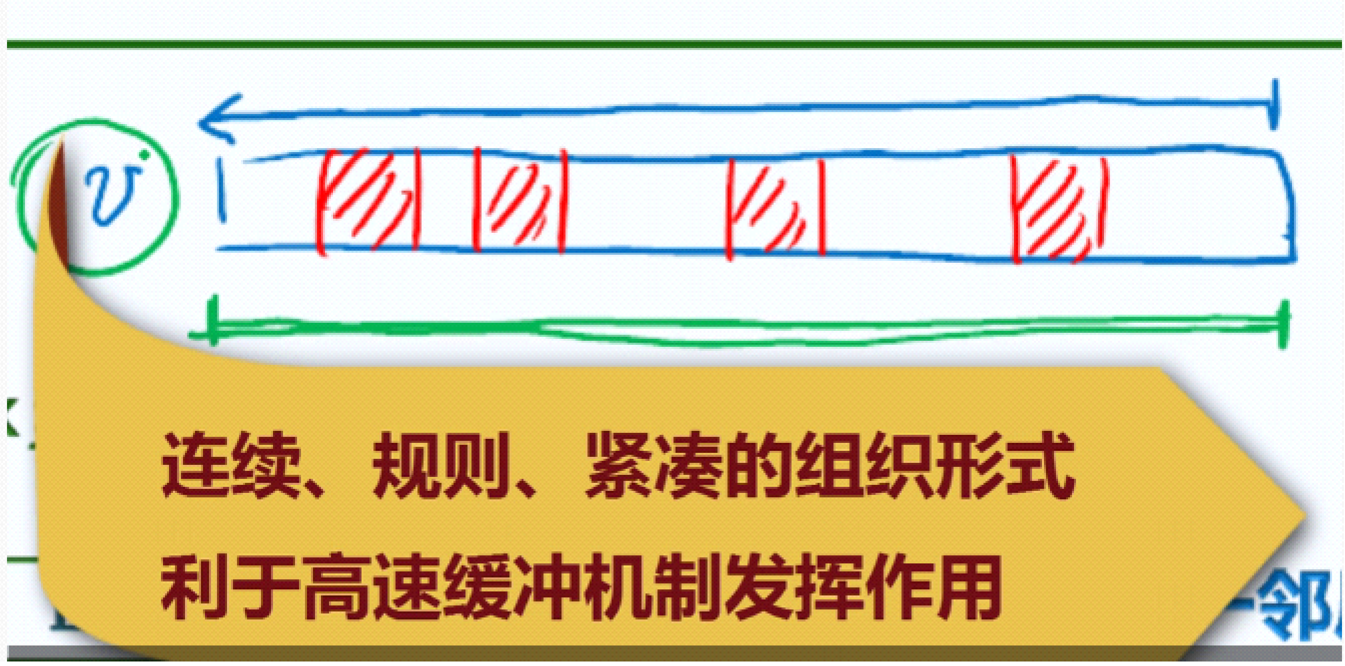
8.3.5 实例



8.3.6 多连通



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。

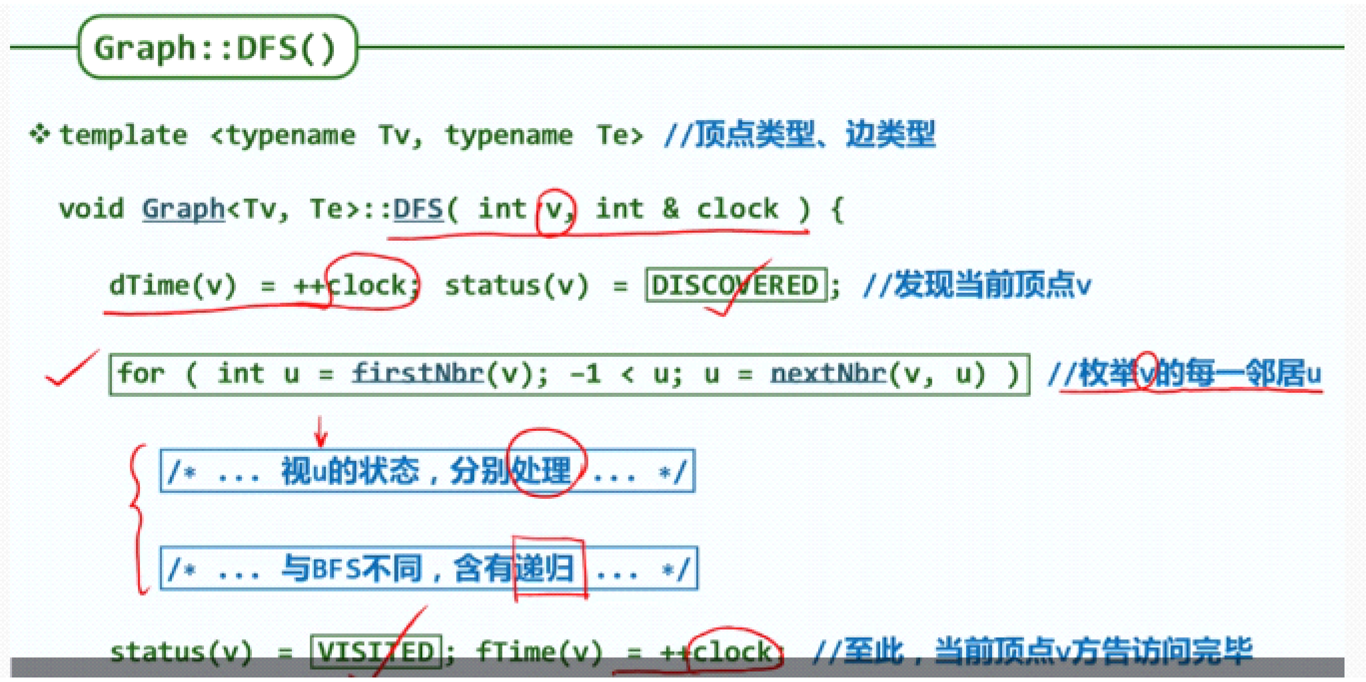
第四节 深度优先搜索DFS

2016年9月1日 18:04

8.4.1 算法

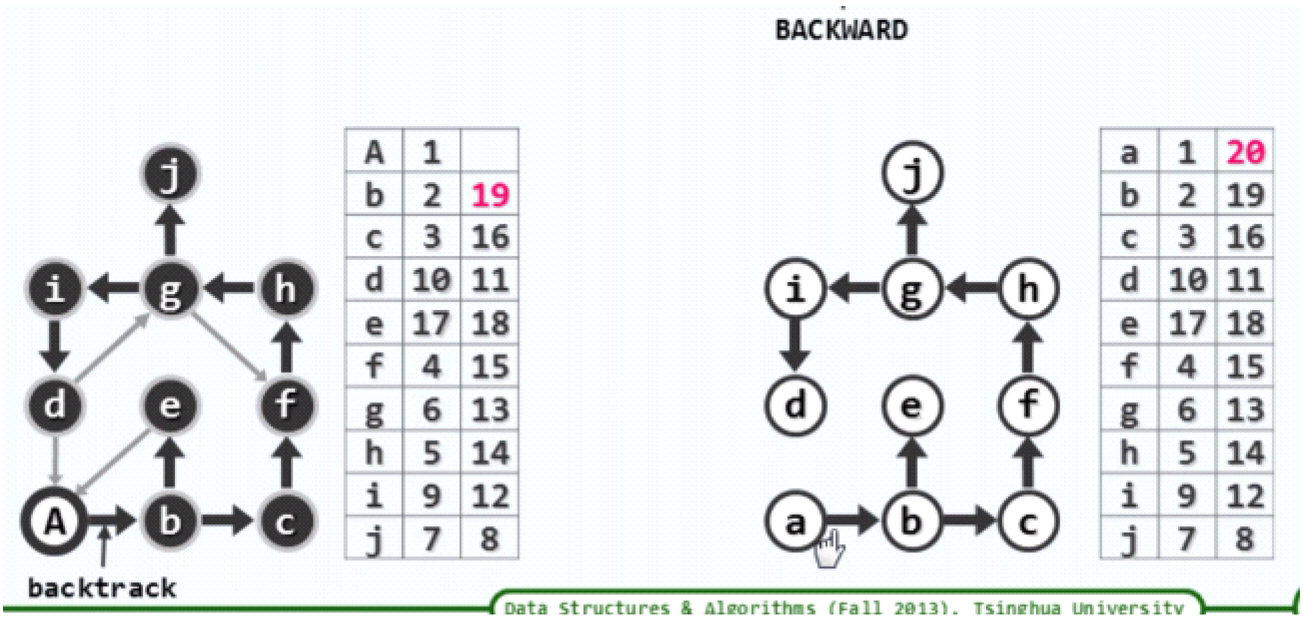
一条通路走到底，走不通时，逐级回溯，直至完成一棵支撑树

8.4.2 框架



8.4.3 细节

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