

## 6. 图

### (b1) 邻接矩阵

邓俊辉

deng@tsinghua.edu.cn

## 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) ) type(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
```

## 邻接矩阵与关联矩阵

❖ **adjacency matrix** : 用二维矩阵记录**顶点**之间的**邻接**关系

——对应：矩阵元素  $\Leftrightarrow$  图中可能存在的边

$$\begin{aligned} A(i, j) &= 1 && \text{若顶点 } i \text{ 与 } j \text{ 之间存在一条边} \\ &= 0 && \text{否则} \end{aligned}$$

既然只考察简单图，对角线统一设置为0

空间复杂度为 $\Theta(n^2)$ ，与图中实际的边数无关

❖ **incidence matrix** : 用二维矩阵记录**顶点与边**之间的**关联**关系

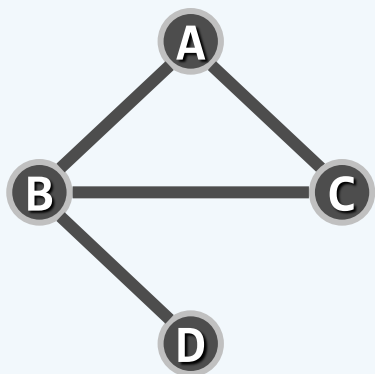
空间复杂度为 $\Theta(n \cdot e) = O(n^3)$

空间利用率  $< 2/n$

解决某些问题时十分有效

# 实例

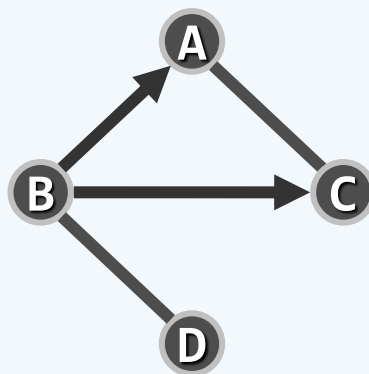
(a) undigraph



$\emptyset$	A	B	C	D
A		1	1	
B	1		1	1
C	1	1		
D		1		

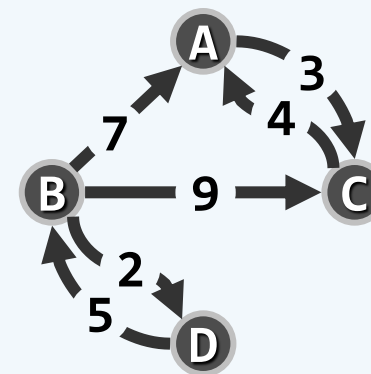
← redundancy

(b) digraph



$\emptyset$	A	B	C	D
A			1	
B	1		1	1
C	1			
D		1		

(c) network



$\infty$	A	B	C	D
A			3	
B	7		9	2
C	4			
D		5		

## 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 ) {}
};
```

## Edge

❖ typedef

```
enum { UNDETERMINED, TREE, CROSS, FORWARD, BACKWARD }  
EType;
```

❖ template <typename Te> struct Edge { //边对象 (并未严格封装)

```
    Te data; //数据
```

```
    int weight; //权重
```

```
    EType type; //在遍历树中所属的类型
```

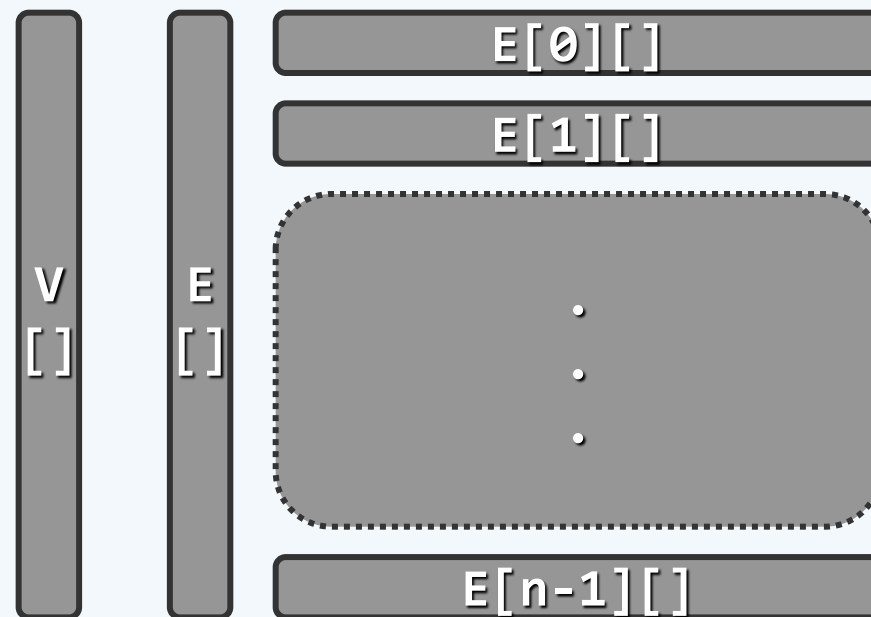
```
    Edge( Te const & d, int w ) : //构造新边
```

```
        data(d), weight(w), type(UNDETERMINED) {}
```

```
};
```

## 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  
        }  
};
```



## 顶点操作

❖ Tv & vertex(int i) { return V[i].data; } //数据

int inDegree(int i) { return V[i].inDegree; } //入度

int outDegree(int i) { return V[i].outDegree; } //出度

VStatus & status(int i) { return V[i].status; } //状态

int & dTime(int i) { return V[i].dTime; } //时间标签dTime

int & fTime(int i) { return V[i].fTime; } //时间标签fTime

int & parent(int i) { return V[i].parent; } //在遍历树中的父亲

int & priority(int i) { return V[i].priority; } //优先级数



## 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 + outDegree(i))  
  
❖ int firstNbr( int i ) {  
    return nextNbr( i, n );  
} //首个邻居
```

## 边操作

- ❖ `bool exists( int i, int j ) { //判断边(i, j)是否存在 ( 短路求值 )`  
    `return (0 <= i) && (i < n) && (0 <= j) && (j < n) && E[i][j] != NULL;`  
    `} //以下假定exists(i, j)...`
- ❖ `Te & edge( int i, int j ) //边(i, j)的数据`  
    `{ return E[i][j]->data; } //O(1)`
- ❖ `EType & type( int i, int j ) //边(i, j)的类型`  
    `{ return E[i][j]->type; } //O(1)`
- ❖ `int & weight( int i, int j ) //边(i, j)的权重`  
    `{ return E[i][j]->weight; } //O(1)`

## 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

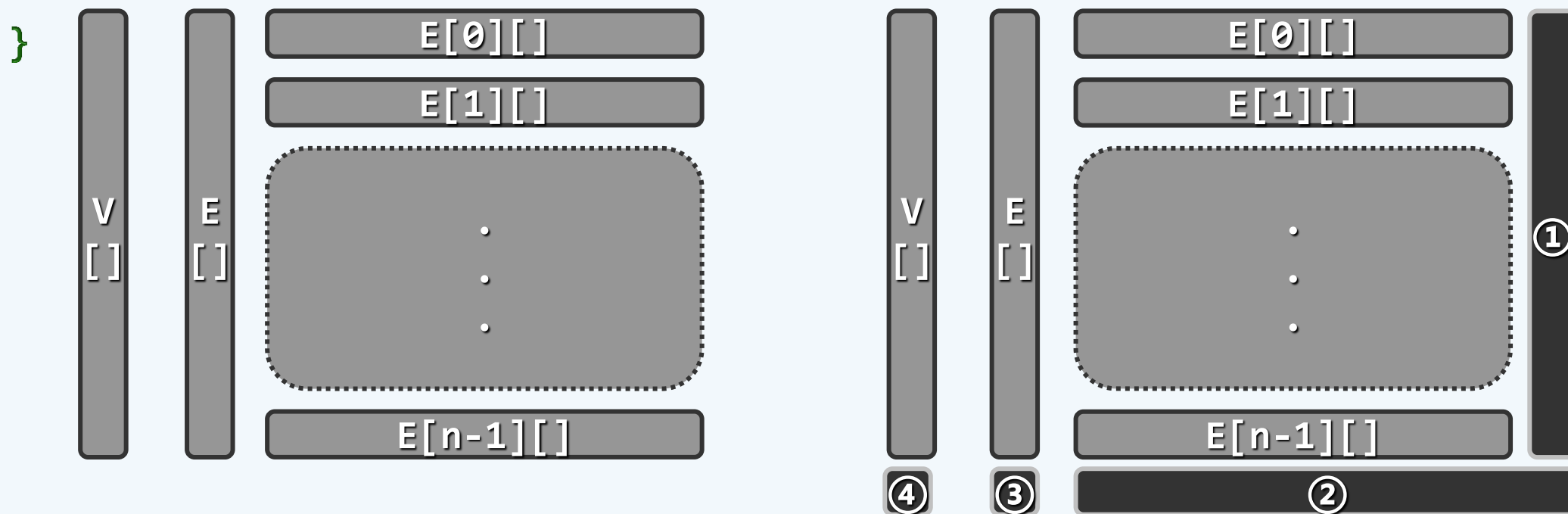
}
```

## 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  
  
}
```

## 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++ ) //删除所有出边

        if ( exists( i, j ) ) { delete E[i][j]; V[j].inDegree--; }

    E.remove(i); n--; //删除第i行

    Tv vBak = vertex( i ); V.remove( i ); //备份之后，删除顶点i

    for ( int j = 0; j < n; j++ ) //删除所有入边及第i列

        if ( Edge<Te> * e = E[j].remove( i ) ) { delete e; V[j].outDegree--; }

    return vBak; //返回被删除顶点的信息

}
```

## 优点

- ❖ 直观，易于理解和实现
- ❖ 适用范围广泛：digraph / network / cyclic / ...  
尤其适用于稠密图 (dense graph)
- ❖ 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

- ❖  $\Theta(n^2)$ 空间，与边数无关！
- ❖ 真会有这么多条边吗？不妨考察一类特定的图...

❖ 平面图 (planar graph) : 可嵌入于平面的图

❖ Euler's formula (1750):

$$v - e + f - c = 1, \text{ for any PG}$$

❖ 平面图 :  $e \leq 3 \times n - 6 = O(n) \ll n^2$

此时，空间利用率  $\approx 1/n$

❖ 稀疏图 sparse graph

空间利用率同样很低，可采用压缩存储技术

