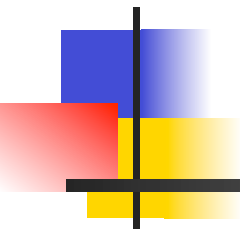


# 第六章 图

---





# 主要内容

---

- **6.1** 图的基本概念
- **6.2** 图的抽象数据类型
- **6.3** 图的存储结构
- **6.4** 图的周游（深度、广度、拓扑）
- **6.5** 最短路径问题
- **6.6** 最小支撑树



## 6.1 图的基本概念

---

- 习惯上，常用 **$G = (V, E)$**  代表一个图 。
- 顶点 (**vertex**)
- 边 (**edge**)
  - 边的始点
  - 边的终点
- 稀疏图 (**sparse graph**)
- 密集图 (**dense graph**)
- 完全图 (**complete graph**)

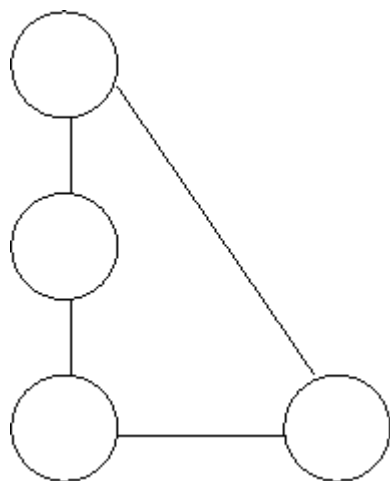


## 6.1 图的基本概念（续）

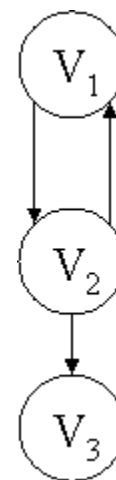
---

- 无向图（**undirected graph**）
  - 图中代表一条边的顶点的偶对无方向性，也即**无序**
- 有向图（**directed graph**或**digraph**）
  - 图中代表一条边的顶点的偶对是**有序**的

## 6.1 图的基本概念（续）



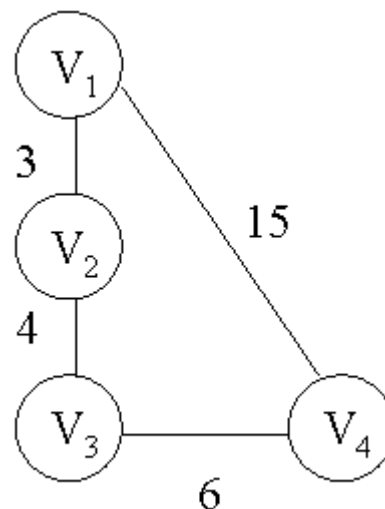
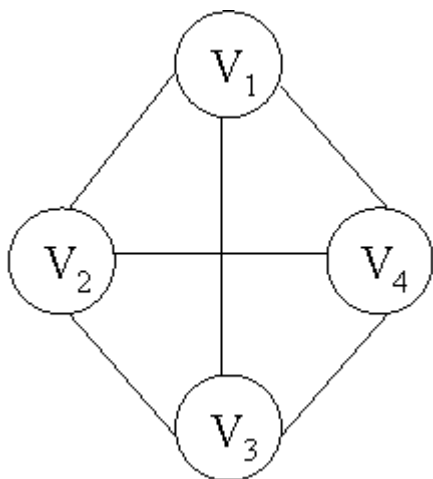
无向图示例



有向图示例

## 6.1 图的基本概念（续）

- 标号图（**labeled graph**）
- 带权图（**weighted graph**）





## 6.1 图的基本概念（续）

---

- 顶点的度（**degree**）
  - 与该顶点相关联的边的数目。
  - 入度(**in degree**)
  - 出度(**out degree**)
- 子图（**subgraph**）
  - 图 $G=(V, E)$ ,  $G'=(V', E')$ 中, 若 $V' \leq V$ ,  $E' \leq E$ , 并且 $E'$ 中的边所关联的顶点都在 $V'$ 中, 则称图 $G'$ 是图 $G$ 的子图



## 6.1 图的基本概念（续）

### ■ 路径（**path**）

- 在图 $G=(V, E)$ 中，如果存在顶点序列 $V_p, V_{i1}, V_{i2}, \dots, V_{in}, V_q$ ，使得 $(V_p, V_{i1}), (V_{i1}, V_{i2}), \dots, (V_{in}, V_q)$ （若对有向图，则使得 $\langle V_p, V_{i1} \rangle, \langle V_{i1}, V_{i2} \rangle, \dots, \langle V_{in}, V_q \rangle$ ）都在 $E$ 中，则称从顶点 $V_p$ 到顶点 $V_q$ 存在一条路径。

### ■ 简单路径（**simple path**）

### ■ 路径长度（**length**）





## 6.1 图的基本概念（续）

---

- 回路（**cycle**，也称为环）
  - 简单回路（**simple cycle**）
- 无环图（**acyclic graph**）
  - 有向无环图（**directed acyclic graph**，简写为**DAG**）



## 6.1 图的基本概念（续）

### ■ 有根的图

- 一个有向图中，若存在一个顶点 $V_0$ ，从此顶点有路径可以到达图中其它所有顶点，则称此有向图为有根的图， $V_0$ 称作图的根。

### ■ 连通的（**connected**）

- 对无向图 $G = (V, E)$ 而言，如果从 $V_1$ 到 $V_2$ 有一条路径（从 $V_2$ 到 $V_1$ 也一定有一条路径），则称 $V_1$ 和 $V_2$ 是连通的（**connected**）。
- 强连通



## 6.1 图的基本概念（续）

---

- 连通分支或者连通分量（**connected component**）
  - 无向图的最大连通子图。
  - 强连通分支（强连通分量）。
- 网络
  - 带权的连通图。
- 自由树（**free tree**）
  - 不带有简单回路的无向图，它是连通的，并且具有  $|V|-1$  条边。



## 6.2 图的抽象数据类型

---

```
class Graph{                                // 图的ADT  
public:  
    int VerticesNum(); //返回图的顶点个数  
    int EdgesNum();   //返回图的边数  
  
    // 返回与顶点oneVertex相关联的第一条边  
    Edge FirstEdge(int oneVertex);  
  
    // 返回与边PreEdge有相同关联顶点oneVertex的  
    // 下一条边  
    Edge NextEdge(Edge preEdge);
```



## 6.2 图的抽象数据类型（续）

---

//添加一条边

```
bool setEdge(int fromVertex,int  
toVertex,int weight);
```

//删一条边

```
bool delEdge(int fromVertex,int  
toVertex);
```

//如果**oneEdge**是边则返回**TRUE**，否则返回  
**FALSE**

```
bool IsEdge(Edge oneEdge);
```



## 6.2 图的抽象数据类型（续）

---

// 返回边**oneEdge**的始点

**int FromVertex(Edge oneEdge);**

// 返回边**oneEdge**的终点

**int ToVertex(Edge oneEdge);**

// 返回边**oneEdge**的权

**int Weight(Edge oneEdge);**

**};**



## 6.3 图的存储结构

---

- **6.3.1 图的相邻矩阵**  
(**adjacency matrix**) 表示法
- **6.3.2 图的邻接表** (**adjacency list**) 表示法
  - **邻接多重表**(**adjacency multilist**)表示法

## 6.3.1 图的相邻矩阵 (adjacency matrix) 表示法

### ■ 相邻矩阵

- 表示顶点间相邻关系的矩阵。
- 若 **G** 是一个具有 **n** 个顶点的图，则 **G** 的相邻矩阵是如下定义的 **n×n** 矩阵：

**$A[i,j]=1$** ，若  $(v_i, v_j)$  (或  $\langle v_i, v_j \rangle$ ) 是图 **G** 的边；

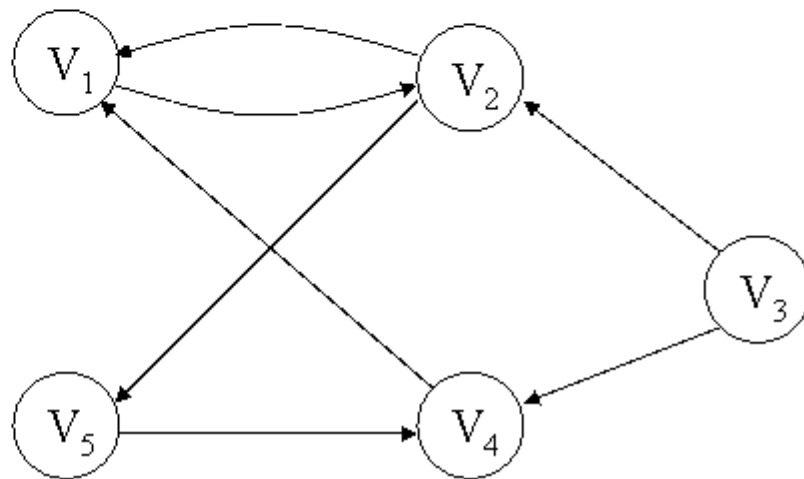
**$A[i,j]=0$** ，若  $(v_i, v_j)$  (或  $\langle v_i, v_j \rangle$ ) 不是图 **G** 的边。

### ■ 相邻矩阵的空间代价为 **$O(n^2)$**



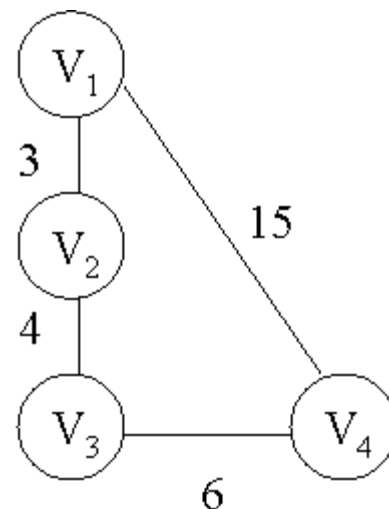
## 6.3.1 图的相邻矩阵 (adjacency matrix) 表示法 (续)

$$A_7 = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

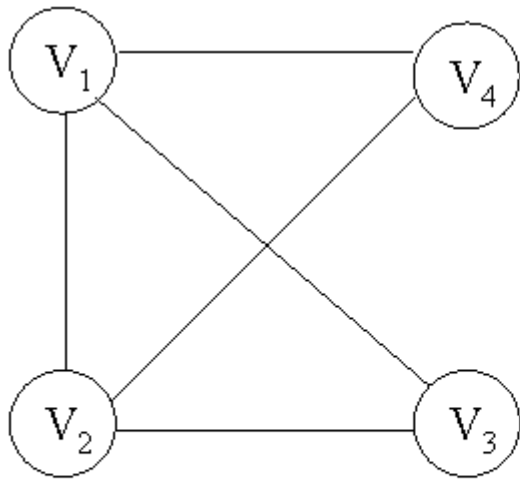


## 6.3.1 图的相邻矩阵 (adjacency matrix) 表示法 (续)

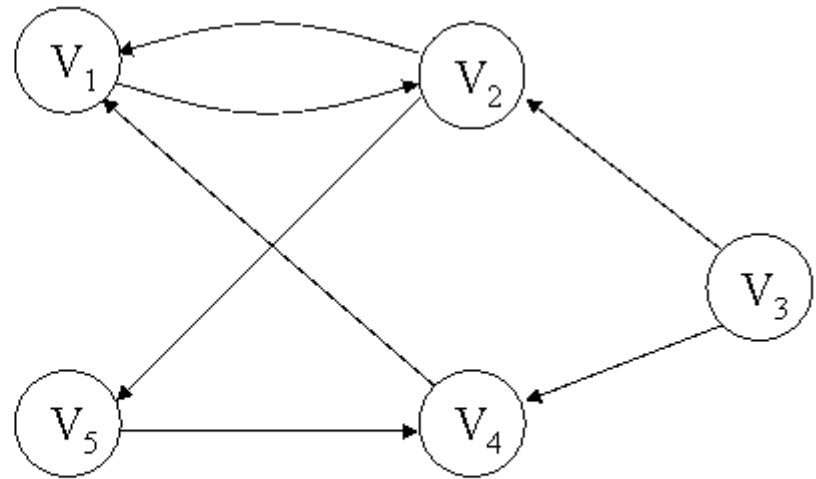
$$\mathbf{A}_4 = \begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$



## 6.3.2 图的邻接表 (adjacency list) 表示法

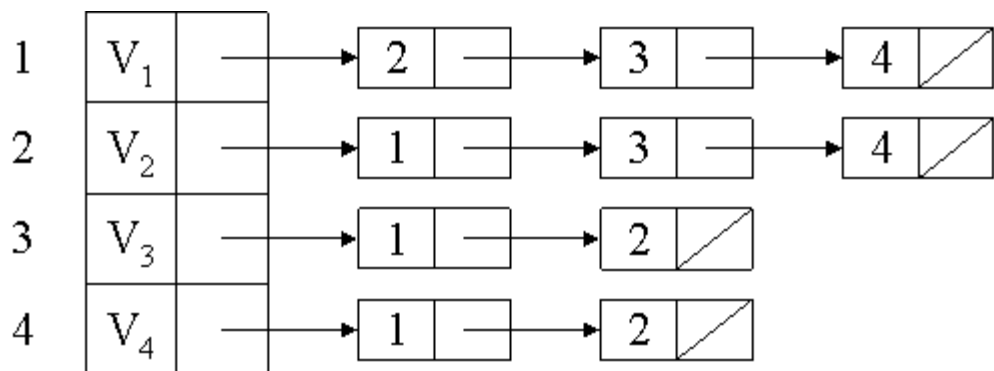


$G_6$



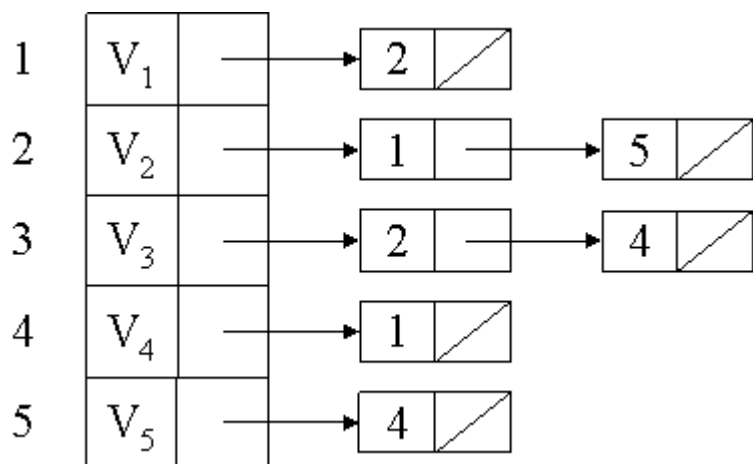
$G_7$

## 6.3.2 图的邻接表 (adjacency list) 表示法 (续)

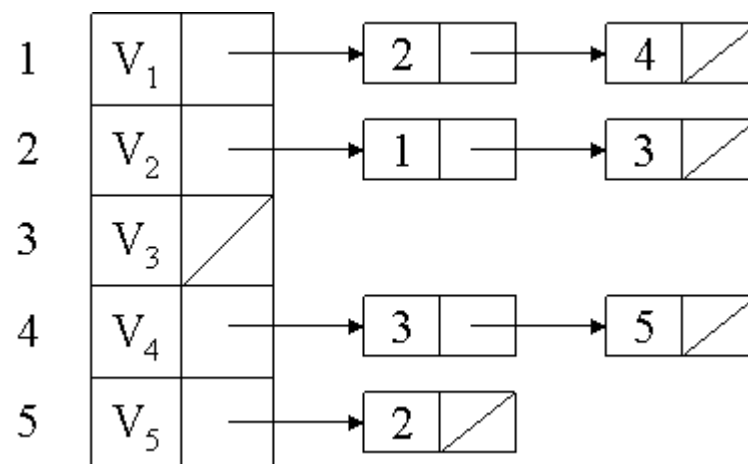


$G_6$ 邻接表表示

## 6.3.2 图的邻接表 (adjacency list) 表示法 (续)



$G_7$  的出边表

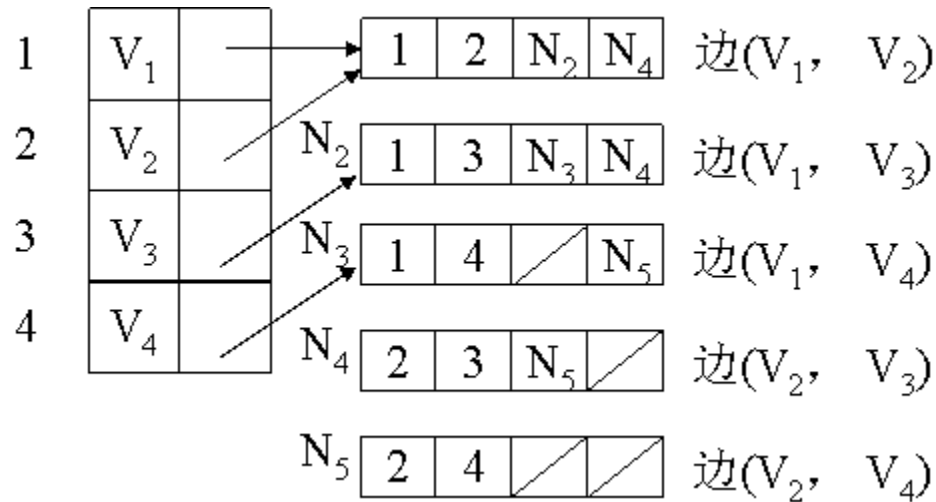


$G_7$  的入边表

# 邻接多重表(adjacency multilist)

- 把邻接表表示中代表同一条边的两个表目合为一个表目
- 图的每条边只用一个多重表表目表示
  - 包括此边的两个顶点的序号
  - 两个指针（一个指针指向与第一个顶点相关联的下一条边，另一个指针指向与第二个顶点相关联的下一条边）
- 在以处理图的边为主，要求每条边处理一次的的实际应用中特别有用。

# 邻接多重表(adjacency multilist)



$G_6$ 的邻接多重表表示

# 有向图邻接多重表(adjacency multilist)

- 在顶点表中设计两个指针
  - 第一个指向以此顶点为始点的第一条边
  - 第二个指向以此顶点为终点的第一条边
- 边表
  - 第一个指针指向始点与本边始点相同的下一条边
  - 第二个指针指向终点与本边终点相同的下一条边
- 故仅用表中第一个链便得到有向图的出边表，仅用第二个链便得到有向图的入边表



# 邻接多重表(adjacency multilist) (续)

1	$V_1$	$N_1$	$N_2$
2	$V_2$	$N_2$	$N_1$
3	$V_3$	$N_6$	
4	$V_4$	$N_5$	$N_4$
5	$V_5$	$N_4$	$N_3$

$N_1$ 

1	2	/	$N_6$
---	---	---	-------

 边 $\langle V_1, V_2 \rangle$

$N_2$ 

2	1	$N_3$	$N_5$
---	---	-------	-------

 边 $\langle V_2, V_1 \rangle$

$N_3$ 

2	5	/	/
---	---	---	---

 边 $\langle V_2, V_5 \rangle$

$N_4$ 

5	4	/	$N_7$
---	---	---	-------

 边 $\langle V_5, V_4 \rangle$

$N_5$ 

4	1	/	/
---	---	---	---

 边 $\langle V_4, V_1 \rangle$

$N_6$ 

3	2	$N_7$	/
---	---	-------	---

 边 $\langle V_3, V_2 \rangle$

$N_7$ 

3	4	/	/
---	---	---	---

 边 $\langle V_3, V_4 \rangle$

## $G_7$ 的邻接多重表表示



## 6.3.2 图的邻接表 (**adjacency list**) 表示法 (续)

---

- **$n$ 个顶点 $m$ 条边的无向图**
  - 需用( **$n+2m$** )个存储单元
- **$n$ 个顶点 $m$ 条边的有向图**
  - 需用( **$n+m$** )个存储单元



## 6.4 图的周游

---

- 图的周游（**graph traversal**）
  - 给出一个图**G**和其中任意一个顶点 **$V_0$** ，从 **$V_0$** 出发系统地访问**G**中所有的顶点，每个顶点访问一次，这叫做**图的周游**。



## 6.4 图的周游（续）

- 图的周游的典型算法
  - 从一个顶点出发，试探性访问其余顶点，同时必须考虑到下列情况：
    - 从一顶点出发，可能不能到达所有其它的顶点，如**非连通图**；
    - 也有可能陷入死循环，如**存在回路的图**。
  - 解决办法
    - 为图的每个顶点保留一个**标志位**（**mark bit**）；
    - 算法开始时，所有顶点的标志位置零；
    - 在周游的过程中，当某个顶点被访问时，其标志位就被标记为已访问。



## 6.4 图的周游（续）

```
//图的周游算法的实现
void graph_traverse(Graph& G){
    //对图所有顶点的标志位进行初始化
    for(int i=0;i<G.VerticesNum();i++)
        G.Mark[i]=UNVISITED;

    //检查图的所有顶点是否被标记过，如果未被标记，
    //则从该未被标记的顶点开始继续周游
    //do_traverse函数用深度优先或者广度优先
    for(int i=0;i<G.VerticesNum();i++)
        if(G.Mark[i]== UNVISITED)
            do_traverse(G, i);
```



## 6.4 图的周游（续）

---

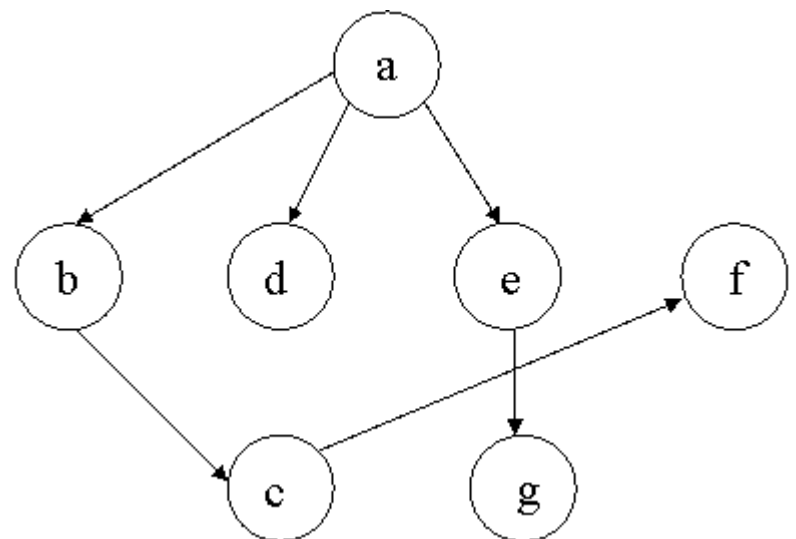
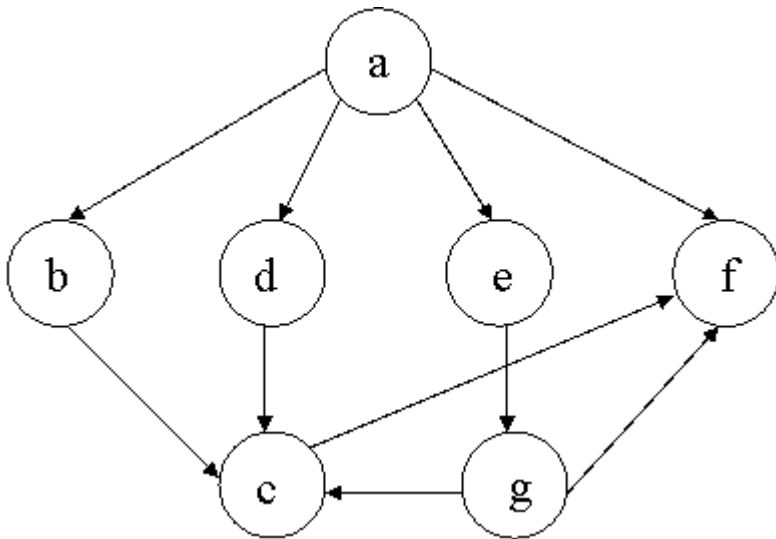
- 图的生成树
  - 图的所有顶点加上周游过程中经过的边所构成的子图称作**图的生成树**。
- 图的生成森林。



## 6.4.1 深度优先搜索

- 深度优先搜索（**depth-first search**，简称**DFS**）基本思想
  - 访问一个顶点**V**，然后访问该顶点邻接到的未被访问过的顶点**V'**，
  - 再从**V'**出发递归地按照深度优先的方式周游，
  - 当遇到一个所有邻接于它的顶点都被访问过了的顶点**U**时，则回到已访问顶点序列中最后一个拥有未被访问的相邻顶点的顶点**W**，
  - 再从**W**出发递归地按照深度优先的方式周游，
  - 最后，当任何已被访问过的顶点都没有未被访问的相邻顶点时，则周游结束。
- 深度优先搜索树（**depth-first search tree**）

## 6.4.1 深度优先搜索（续）



深度优先搜索的顺序是a, b, c, f, d, e, g





## 6.4.1 深度优先搜索（续）

```
void DFS(Graph& G, int V){    //深度优先搜索算法实现
    G.Mark[V]= VISITED; //访问顶点V，并标记其标志位
    PreVisit(G, V);        //访问V
    for(Edge e=G. FirstEdge(V); G.IsEdge(e);
        e=G. NextEdge(e))
        //访问V邻接到的未被访问过的顶点，并递归地按照
        //深度优先的方式进行周游
        if(G.Mark[G. ToVertices(e)]== UNVISITED)
            DFS(G, G. ToVertices(e));
    PostVisit(G, V);        //访问V
}
```



## 6.4.1 深度优先搜索（续）

- 深度优先搜索算法的时间复杂度

**DFS**对每一条边处理一次（无向图的每条边从两个方向处理），每个顶点访问一次。

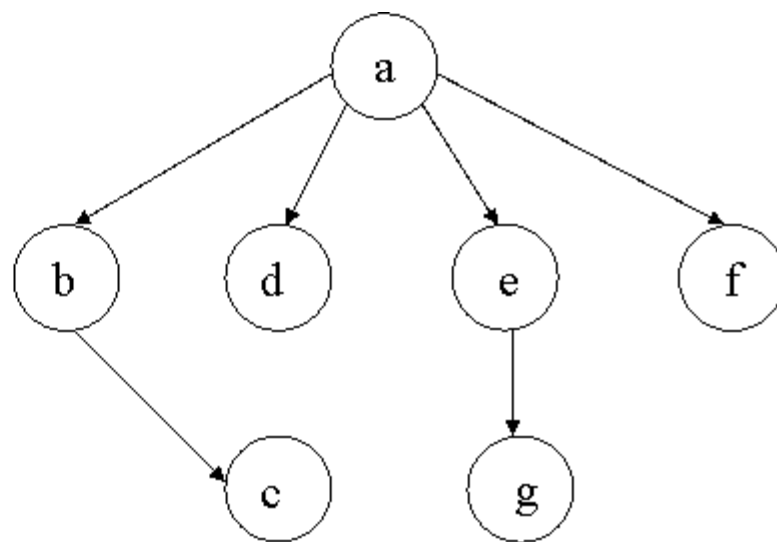
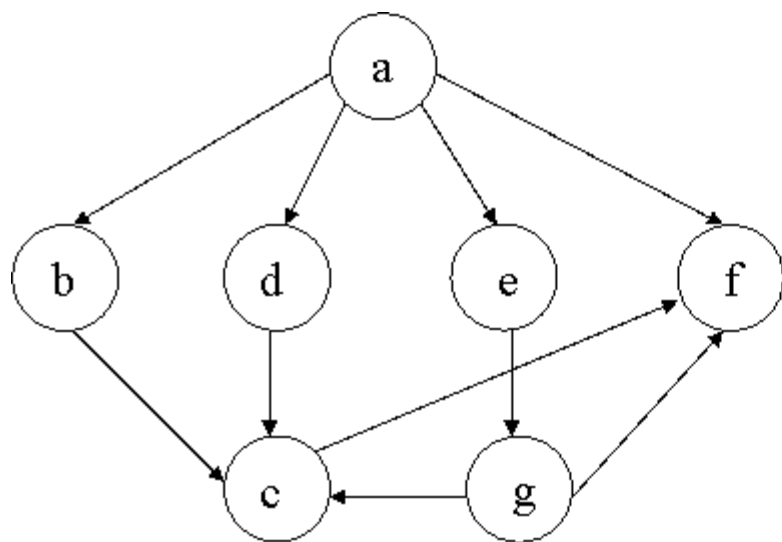
  - 采用邻接表表示时，有向图总代价为 $\Theta(|V| + |E|)$ ，无向图为 $\Theta(|V| + 2|E|)$
  - 采用相邻矩阵表示时，处理所有的边需要 $\Theta(|V|^2)$ 的时间，所以总代价为 $\Theta(|V| + |V|^2) = \Theta(|V|^2)$ 。



## 6.4.2 广度优先搜索

- 广度优先搜索（**breadth-first search**, 简称**BFS**）
  - 它的基本思想是访问顶点 $V_0$ ,
  - 然后访问 $V_0$ 邻接到的所有未被访问过的顶点 $V_{01}, V_{02}, \dots, V_{0i}$ ,
  - 再依次访问 $V_{01}, V_{02}, \dots, V_{0i}$ 邻接到的所有未被访问的顶点,
  - 如此进行下去, 直到访问遍所有的顶点。
- 广度优先搜索树（**breadth-first search tree**）

## 6.4.2 广度优先搜索（续）



广度优先搜索的顺序是a, b, d, e, f, c, g



## 6.4.2 广度优先搜索（续）

---

//广度优先搜索算法的实现

**void BFS(Graph& G, int V){**

    //初始化广度优先周游要用到的队列

**using std::queue;**

**queue<int> Q;**

    //访问顶点**V**，并标记其标志位， **V**入队

**G.Mark[V]= VISITED;**

**Visit(G, V);**

**Q.push(V);**

**while(!Q.empty())**   //如果队列仍然有元素



## 6.4.2 广度优先搜索（续）

---

```
{  
    int V=Q.front(); //顶部元素  
    Q.pop();        //出队  
  
    //将与该点相邻的每一个未访问点都入队  
    for(Edge e=G.FirstEdge(V);  
        G.IsEdge(e);e=G.NextEdge(e))  
    {  
        if(G.Mark[G.ToVertex(e)]==  
            UNVISITED)
```



## 6.4.2 广度优先搜索（续）

---

```
{  
    G.Mark[G.ToVertex(e)]=VISITED;  
    Visit(G, G.ToVertex(e));  
  
    //入队  
    Q.push(G.ToVertex(e));  
}  
}  
}  
}
```



## 6.4.2 广度优先搜索（续）

---

- 广度优先搜索算法的时间复杂度与深度优先搜索算法的时间复杂度相同





## 6.4.3 拓扑排序

---

- 先决条件问题。
- 拓扑排序 (**topological sort**)
  - 将一个有向无环图中所有顶点在不违反**先决条件关系**的前提下排成线性序列的过程称为**拓扑排序**



## 6.4.3 拓扑排序（续）

### ■ 拓扑序列

- 对于有向无环图 $G = (V, E)$ ， $V$ 里顶点的线性序列称作一个**拓扑序列**，如果该顶点序列满足：
  - 若在有向无环图 $G$ 中从顶点 $V_i$ 到 $V_j$ 有一条路径，则在序列中顶点 $V_i$ 必在顶点 $V_j$ 之前。

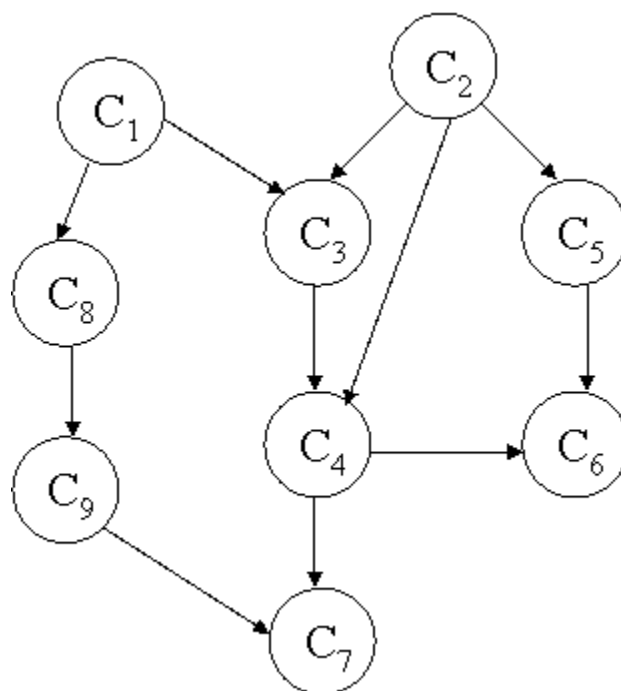


## 6.4.3 拓扑排序（续）

---

课程代号	课程名称	先修课程
<b>C1</b>	高等数学	
<b>C2</b>	程序设计	
<b>C3</b>	离散数学	<b>C1, C2</b>
<b>C4</b>	数据结构	<b>C2, C3</b>
<b>C5</b>	算法语言	<b>C2</b>
<b>C6</b>	编译技术	<b>C4, C5</b>
<b>C7</b>	操作系统	<b>C4, C9</b>
<b>C8</b>	普通物理	<b>C1</b>
<b>C9</b>	计算机原理	<b>C8</b>

## 6.4.3 拓扑排序（续）



学生课程的安排图



## 6.4.3 拓扑排序（续）

---

- 任何无环有向图，其顶点都可以排在一个拓扑序列里，其拓扑排序的方法是：
  - （**1**）从图中选择一个入度为**0**的顶点且输出之。
  - （**2**）从图中删掉此顶点及其所有的出边。
  - （**3**）回到第（**1**）步继续执行。



## 6.4.3 拓扑排序（续）

---

//队列方式实现的拓扑排序

```
void TopsortbyQueue(Graph& G) {  
    for(int i=0;i<G.VerticesNum();i++)  
        G.Mark[i]=UNVISITED;    //初始化标记数组  
using std::queue;  
queue<int> Q;                    //初始化队列  
for(i=0; i<G.VerticesNum(); i++) {  
    if(G.Indegree[i]==0)  
        Q.push(i);                //图中入度为0的顶点入队  
}  
  
while(!Q.empty()){                //如果队列中还有图的顶点
```



## 6.4.3 拓扑排序（续）

---

```
int V=Q.front();           //顶部元素
Q.pop();                   //一个顶点出队
Visit(G, V);              //访问该顶点
G.Mark[V]=VISITED;

//边e的终点的入度值减1
for(Edge e= G.FirstEdge(V); G.IsEdge(e);e=G.NextEdge(e))
{
    G.Indegree[G.ToVertex(e)]--;
    if(G.Indegree[G.ToVertex(e)]==0)
        Q.push(G.ToVertex(e));    //入度为0的顶点入队
}
}
```



## 6.4.3 拓扑排序（续）

---

```
for(i=0; i<G.VerticesNum(); i++) {  
    if(G.Mark[i]==UNVISITED)  
    {  
        Print(“图有环”);    //图有环  
        break;  
    }  
}  
}
```





## 6.4.3 拓扑排序（续）

---

//深度优先搜索实现的拓扑排序

```
void Do_topsort(Graph& G, int V,int  
    *result,int& tag)
```

```
{
```

```
    G.Mark[V]= VISITED;
```

```
    for(Edge e= G.FirstEdge(V);
```

```
        G.IsEdge(e);e=G.NextEdge(e)) {
```

```
        //访问V邻接到的所有未被访问过的顶点
```

```
        if(G.Mark[G.ToVertex(e)]== UNVISITED)
```



## 6.4.3 拓扑排序（续）

---

```
//递归调用
Do_topsort(G,
G.ToVertex(e),result,tag);
}
result[tag++] = V;
}
```



## 6.4.3 拓扑排序（续）

---

//深度优先搜索方式实现的拓扑排序,结果是颠倒的

**void TopsortbyDFS(Graph& G)**

**{**

    //对图所有顶点的标志位进行初始化

**for(int i=0; i<G.VerticesNum(); i++)**

**G.Mark[i]=UNVISITED;**

**int \*result=new int[G.VerticesNum()];**

**int tag=0;**

    //对图的所有顶点进行处理

**for(i=0; i<G.VerticesNum(); i++)**

**if(G.Mark[i]== UNVISITED)**



## 6.4.3 拓扑排序（续）

---

```
Do_topsort(G,i,result,tag);    //调用递归函数
for(i=G.VerticesNum()-1;i>=0;i--) //逆序输出
{
    Visit(G, result[i]);
}
}
```

注：深度优先搜索方式实现的拓扑排序无法判断图是否存在环。



## 6.4.3 拓扑排序（续）

- 拓扑排序的时间复杂度
  - 图的每条边处理一次
  - 图的每个顶点访问一次

所以，深度优先搜索方式实现的拓扑排序的时间复杂度同图的深度优先搜索方式周游。

队列方式的拓扑排序：在有环的情况下会提前退出，从而可能没处理完所有的边和顶点。



## 6.4.3 拓扑排序（续）

- 队列方式拓扑排序的时间复杂度
  - 关键是，算法开始时找出所有入度为**0**的顶点（同时可知道其它顶点的入度）
    - 当采用相邻矩阵时，算法开始时找所有入度为**0**的顶点需要  $\Theta(|V|^2)$  的时间，加上处理边、顶点的时间，总代价为  $\Theta(|V|^2 + |E| + |V|) = \Theta(|V|^2)$
    - 当采用邻接表时，因为在顶点表的每个顶点中可以有一个字段来存储入度，所以只需  $\Theta(|V|)$  的时间，加上处理边、顶点的时间，总代价为  $\Theta(2|V| + |E|)$



## 6.5 最短路径问题

---

- **6.5.1 单源最短路径 (single-source shortest paths)**
  - 指的是对已知图 $G = (V, E)$ ，给定源顶点 $s \in V$ ，找出 $s$ 到图中其它各顶点的最短路径。
- **6.5.2 每对顶点间的最短路径 (all-pairs shortest paths)**
  - 指的是对已知图 $G = (V, E)$ ，任意的顶点 $V_i, V_j \in V$ ，找出从 $V_i$ 到 $V_j$ 的最短路径。



## 6.5.1 单源最短路径

---

- **Dijkstra**算法基本思想：
  - 把图中所有顶点分成两组
    - 第一组包括已确定最短路径的顶点
    - 第二组包括尚未确定最短路径的顶点；
  - 按最短路径长度递增的顺序逐个把第二组的顶点加到第一组中去
    - 直至从 $s$ 出发可以到达的所有顶点都包括进第一组中。





## 6.5.1 单源最短路径（续）

- 在这个过程中，总保持从**s**到第一组各顶点的最短路径长度都不大于从**s**到第二组的任何顶点的最短路径长度，而且，每个顶点都对应一个距离值：
  - 第一组的顶点对应的距离值就是从**s**到该顶点的最短路径长度
  - 第二组的顶点对应的距离值是从**s**到该顶点的只包括第一组的顶点为中间顶点的最短路径长度



## 6.5.1 单源最短路径（续）

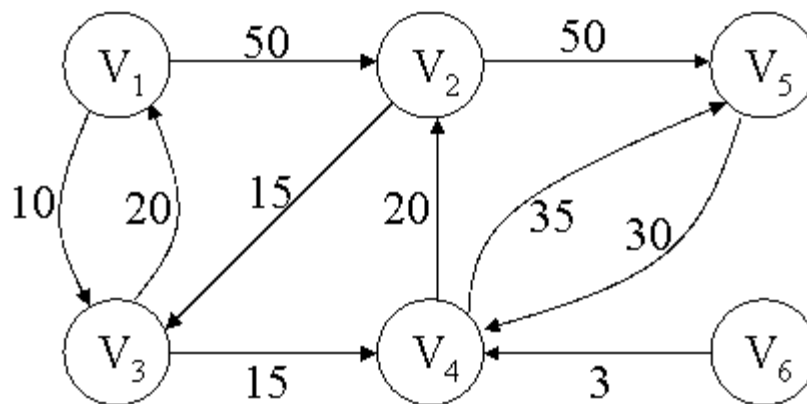
- **Dijkstra**算法的具体做法：
  - 一开始第一组只包括顶点 $s$ ，第二组包括其它所有顶点；
  - $s$ 对应的距离值为0，而第二组的顶点对应的距离值这样确定：
    - 若图中有边 $\langle s, V_i \rangle$ 或者 $(s, V_i)$ ，则 $V_i$ 的距离值为此边所带的权，否则 $V_i$ 的距离值为 $\infty$ 。
  - 然后，每次从第二组的顶点中选一个其距离值为最小的顶点 $V_m$ 加入到第一组中；



## 6.5.1 单源最短路径（续）

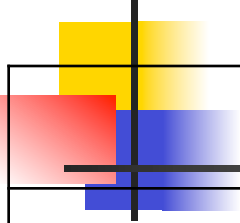
- 每往第一组加入一个顶点 $V_m$ ，就要对第二组的各顶点的距离值进行一次修正：
  - 若加进 $V_m$ 做中间顶点，使从 $s$ 到 $V_i$ 的最短路径比不加 $V_m$ 的为短，则需要修改 $V_i$ 的距离值。
- 修改后再选距离值最小的顶点加入到第一组中，如此进行下去，直到图的所有顶点都包括在第一组中或者再也没有可加入到第一组的顶点存在。

## 6.5.1 单源最短路径（续）



求上图中顶点 $V_1$ 到其它各顶点的最短路径

## 6.5.1 单源最短路径（续）



	$V_1$	$V_2$	$V_3$	$V_4$	$V_5$	$V_6$
初始状态	length:0 pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1
$V_1$ 进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1	length: $\infty$ pre:1
$V_3$ 进入第一组	length:0 pre:1	length:50 pre:1	length:10 pre:1	length:25 pre:3	length: $\infty$ pre:1	length: $\infty$ pre:1
$V_4$ 进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length: $\infty$ pre:1
$V_2$ 进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length: $\infty$ pre:1
$V_5$ 进入第一组	length:0 pre:1	length:45 pre:4	length:10 pre:1	length:25 pre:3	length:60 pre:4	length: $\infty$ pre:1

用Dijkstra算法的处理过程，源顶点为 $V_1$



## 6.5.1 单源最短路径（续）

---

**//Dijkstra算法**

```
void Dijkstra(Graph& G,int s, Dist* &D) {  
    D=new Dist[G.VerticesNum()];
```

```
    //初始化Mark数组、D数组
```

```
    //minVertex函数中会用到Mark数组的信息
```

```
    for(int i=0;i<G.VerticesNum();i++){
```

```
        G.Mark[i]=UNVISITED;
```

```
        D[i].length= INFINITY;
```

```
        D[i].pre=s;
```

```
    }
```

```
    D[s].length=0;
```



## 6.5.1 单源最短路径（续）

---

```
for(i=0;i<G.VerticesNum();i++){
    int v=minVertex(G,D);    //找到距离s最小的顶点
    if(D[v].length==INFINITY)
        break;
    G.Mark[v]=VISITED;        //把该点加入已访问组
    Visit(G,v);                //打印输出
    //刷新D中的值，因为v的加入D的值需要改变，
    //只要刷新与v相邻的点的值
    for(Edge e=G.FirstEdge(v);
        G.IsEdge(e);e=G.NextEdge(e)){
```



## 6.5.1 单源最短路径（续）

---

```
if(D[G.ToVertex(e)].length>(D[v].length  
+G.Weight(e))){
```

```
D[G.ToVertex(e)].length=D[v].length  
+G.Weight(e);
```

```
D[G.ToVertex(e)].pre=v;
```

```
}  
}  
}  
}
```





## 6.5.1 单源最短路径（续）

---

其中的**Dist**类可以如下定义：

```
Class Dist{  
int length;    //与源s的距离  
int pre;       //前面的顶点  
};
```

而**minVertex()**函数可用最小堆（**Minheap**）等方式实现。



## 6.5.1 单源最短路径（续）

### ■ **Dijkstra**算法时间代价分析

- 如果**minVertex()**函数不采用最小堆的方式，而是通过两两比较来扫描**D**数组
  - 因为每次**minVertex()**扫描需要进行 $|V|$ 次，而在更新**D**值处总共扫描 $|E|$ 次
  - 所以本方法的总时间代价为 $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$ ，因为 $|E|$ 在 $O(|V|^2)$ 中



## 6.5.1 单源最短路径（续）

- 如果**minVertex()**函数采用最小堆的方式，
  - 每次改变**D[i].length**，可以通过先删除再重新插入的方法来改变顶点*i*在堆中的位置，
  - 或者仅为某个顶点添加一个新值(更小的)，作为堆中新元素(而不作删除旧值的操作，因为旧值被找到时，该顶点一定被标记为**VISITED**，从而被忽略)。
  - 不作删除旧值的缺点是，在最差情况下，它将使堆中元素数目由 $\Theta(|V|)$ 增加到 $\Theta(|E|)$ ，此时总的时间代价为 $\Theta((|V| + |E|)\log |E|)$ ，因为处理每条边时都必须对堆进行一次重排。



## 6.5.2 每对顶点间的最短路径

### ■ Floyd算法

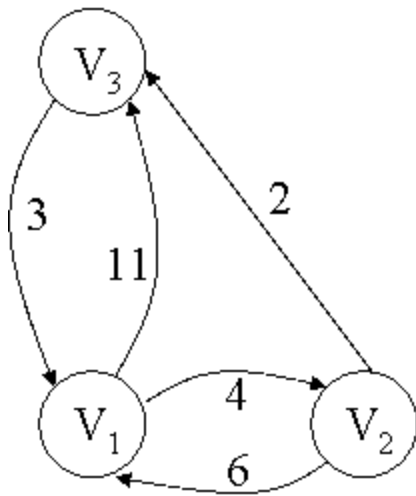
算法思想：

- 假设用相邻矩阵 $\mathbf{adj}$ 表示图
- Floyd算法递归地产生一个矩阵序列 $\mathbf{adj}^{(0)}, \mathbf{adj}^{(1)}, \dots, \mathbf{adj}^{(k)}, \dots, \mathbf{adj}^{(n)}$
- $\mathbf{adj}^{(k)}[i, j]$ 等于从顶点 $V_i$ 到顶点 $V_j$ 中间顶点序号不大于 $k$ 的最短路径长度

## 6.5.2 每对顶点间的最短路径 (续)

- 假设已求得矩阵 $\mathbf{adj}^{(k-1)}$ , 那么从顶点 $V_i$ 到顶点 $V_j$ 中间顶点的序号不大于 $k$ 的最短路径有两种情况:
  - 一种是中间不经过顶点 $V_k$ , 那么就有 $\mathbf{adj}^{(k)}[i, j] = \mathbf{adj}^{(k-1)}[i, j]$
  - 另一种是中间经过顶点 $V_k$ , 那么 $\mathbf{adj}^{(k)}[i, j] < \mathbf{adj}^{(k-1)}[i, j]$ , 且 $\mathbf{adj}^{(k)}[i, j] = \mathbf{adj}^{(k-1)}[i, k] + \mathbf{adj}^{(k-1)}[k, j]$

# 6.5.2 每对顶点间的最短路径 (续)



$$\text{adj}^{(0)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

$$\text{path} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 0 & 3 \end{bmatrix}$$

$$\text{adj}^{(1)} = \begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$\text{path} = \begin{bmatrix} 1 & 1 & 1 \\ 2 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

$$\text{adj}^{(2)} = \begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$\text{path} = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

$$\text{adj}^{(3)} = \begin{bmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

$$\text{path} = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 2 & 2 \\ 3 & 1 & 3 \end{bmatrix}$$

©版权所有，转载或翻印必究

## 6.5.2 每对顶点间的最短路径 (续)

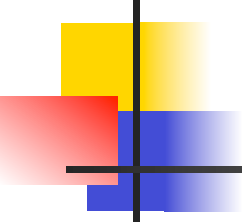
//Floyd算法

```
void Floyd(Graph& G, Dist** &D){  
    int i,j,v;           //i,j,v作为计数器
```

```
    D=new    Dist*[G.VerticesNum()];  
    for(i=0; ;i<G.VerticesNum();i++) {  
        D[i]=new Dist[G.VerticesNum()];  
    }
```

```
    //初始化D数组  
    for(i=0;i<G.VerticesNum();i++)  
        for(j=0;j<G.VerticesNum();j++)
```

## 6.5.2 每对顶点间的最短路径 (续)



```
if(i==j){  
    D[i][j].length=0;  
    D[i][j].pre=i;  
}else {  
    D[i][j]=INFINITY;  
    D[i][j].pre=-1;  
}  
  
for(v=0;v<G.VerticesNum();v++)  
    for(Edge e=G.FirstEdge(v);  
        G.IsEdge(e);e=G.NextEdge(e)){  
  
        D[v]  
        [G.ToVertex(e)].length=G.Weight(e);  
        D[v][G.ToVertex(e)].pre=v;  
    }  
}
```



## 6.5.2 每对顶点间的最短路径 (续)

//如果两个顶点间的最短路径经过顶点 $v$ , 则有  
// $D[i][j].length > (D[i][v].length + D[v][j].length)$

```
for(v=0;v<G.VerticesNum();v++)
  for(i=0;i<G.VerticesNum();i++)
    for(j=0;j<G.VerticesNum();j++)
      if(D[i][j].length > (D[i][v].length
        + D[v][j].length)){
        D[i][j].length = D[i][v].length
          + D[v][j].length;
        D[i][j].pre = D[v][j].pre;
      }
```

## 6.5.2 每对顶点间的最短路径 (续)



---

注：其中**Dist**类型与**Dijkstra**算法用到的**Dist**类型一致。

## 6.5.2 每对顶点间的最短路径 (续)

- 因为**Floyd**算法的时间复杂度主要在于三重**for**循环，所以很明显，其复杂度是 **$O(n \times n \times n)$**

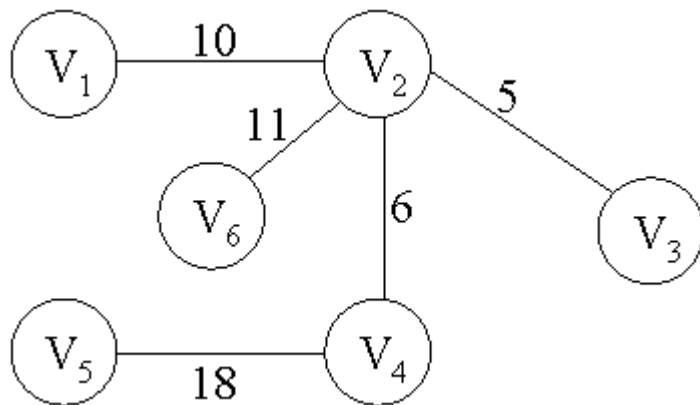
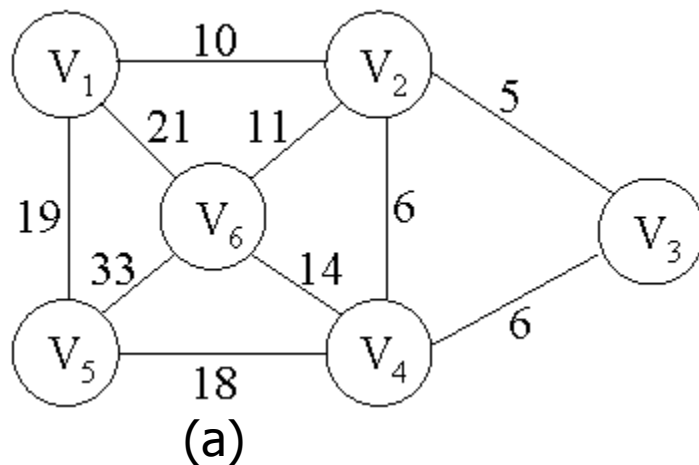


## 6.6 最小支撑树

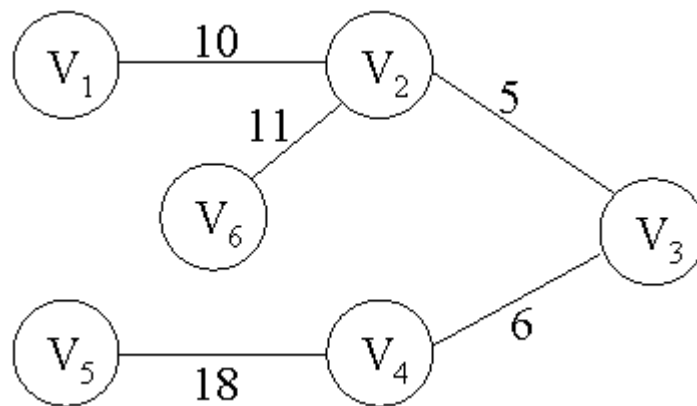
---

- 最小支撑树（**minimum-cost spanning tree**，简称**MST**）
  - 对于带权的连通无向图**G**，其最小支撑树是一个包括**G**的所有顶点和部分边的图，这部分的边满足下列条件：
    - （1）这部分的边能够保证图是连通的；
    - （2）这部分的边，其权的总和最小。

## 6.6 最小支撑树（续）



(a)的MST



(a)的MST

©版权所有，转载或翻印必究



## 6.6.1 Prim算法

- **Prim算法的具体操作是：**
  - 从图中任意一个顶点开始，首先把这个顶点包括在**MST**里，
  - 然后在那些其一个端点已在**MST**里，另一个端点还未在**MST**里的边中，找权最小的一条边，并把这条边和其不在**MST**里的那个端点包括进**MST**里。
  - 如此进行下去，每次往**MST**里加一个顶点和一条权最小的边，直到把所有的顶点都包括进**MST**里。



## 6.6.1 Prim算法（续）

- 证明用**Prim**算法构造的生成树确实是**MST**。

首先证明这样一个结论：

设 $T(V^*, E^*)$ 是连通无向图 $G=(V, E)$ 的一棵正在构造的生成树，又 $E$ 中有边 $e=(V_x, V_y)$ ，其中 $V_x \in V^*$ ， $V_y$ 不属于 $V^*$ ，且 $e$ 的权 $W(e)$ 是所有一个端点在 $V^*$ 里，另一端不在 $V^*$ 里的边的权中最小者，则一定存在 $G$ 的一棵包括 $T$ 的**MST**包括边 $e=(V_x, V_y)$ 。



## 6.6.1 Prim算法（续）

---

### ■ 用反证法

- 设**G**的任何一棵包括**T**的**MST**都不包括 $\mathbf{e}=(\mathbf{V}_x, \mathbf{V}_y)$ , 且设**T'** 是一棵这样的**MST**
- 由于**T'** 是连通的, 因此有从 $\mathbf{V}_x$ 到 $\mathbf{V}_y$ 的路径 $\mathbf{V}_x, \dots, \mathbf{V}_y$



## 6.6.1 Prim算法（续）

- 把边 $e=(v_x, v_y)$ 加进树 $T'$ ，得到一个回路 $v_x, \dots, v_y, v_x$
- 上述路径 $v_x, \dots, v_y$ 中必有边 $e'=(v_p, v_q)$ ，其中 $v_p \in V^*$ ， $v_q$ 不属于 $V^*$ ，由条件知边的权 $W(e') \geq W(e)$ ，从回路中去掉边 $e'$ ，回路打开，成为另一棵生成树 $T''$ ， $T''$ 包括边 $e=(v_x, v_y)$ ，且各边权的总和不大于 $T'$ 各边权的总和

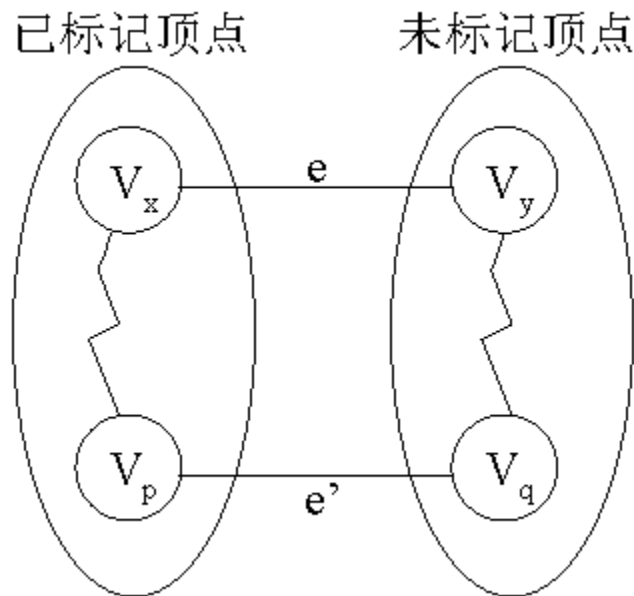


## 6.6.1 Prim算法（续）

---

- 因此**T**”是一棵包括边**e**的**MST**，与假设矛盾，即证明了我们的结论。
- 也证明了**Prim**算法构造**MST**的方法是正确的，因为我们是从**T**包括任意一个顶点和**0**条边开始，每一步加进去的都是**MST**中应包括的边，直至最后得到**MST**。

## 6.6.1 Prim算法（续）



Prim算法证明过程的图示



## 6.6.1 Prim算法（续）

---

//最小支撑树的**Prim**算法

```
void Prim(Graph& G, int s, Edge* &MST ) {  
    //最小支撑树  
    int MSTtag=0; //最小支撑树边的标号  
    Edge *MST=new Edge[G.VerticesNum()-1];  
  
    //最小值堆（minheap）  
    MinHeap<Edge> H(G.EdgesNum());  
  
    //初始化Mark数组、距离数组  
    for(int i=0;i<G.VerticesNum();i++)  
        G.Mark[i]=UNVISITED;
```



## 6.6.1 Prim算法（续）

---

```
int v=s;  
G.Mark[v]=VISITED;           //开始顶点首先被标记  
  
do{  
    //将以v为顶点,  
    //另一顶点未被标记的边插入最小值堆H  
for(Edge e= G. FirstEdge(v);G.IsEdge(e);e=G.  
    NextEdge(e))  
        if(G. Mark[G. ToVertex(e)]==UNVISITED)  
            H.Insert(e);
```



## 6.6.1 Prim算法（续）

---

```
//寻找下一条权最小的边
bool Found=false;
while(!H.empty())
{
    e=H.RemoveMin();
    if(G. Mark[G. ToVertex(e)]==UNVISITED)
    {
        Found=true;
        break;
    }
}
```



## 6.6.1 Prim算法（续）

---

```
if(!Found)
{
    Print("不存在最小支撑树。");
    delete [] MST;           //释放空间
    MST=NULL;                 //MST是空数组
    return;
}
```



## 6.6.1 Prim算法（续）

---

**v= G. ToVertex(e);**

**//在顶点v的标志位上做已访问的标记**

**G.Mark[v]=VISITED;**

**//将边e加到MST中**

**AddEdgetoMST(e,MST,MSTtag++);**

**}while(MSTtag < (G. VerticesNum()-1))**

**}**





## 6.6.1 Prim算法（续）

---

- **Prim算法与Dijkstra算法十分相似，唯一区别是：Prim算法要寻找的是离已加入顶点距离最近的顶点，而不是寻找离固定顶点距离最近的顶点，所以其时间复杂度分析与Dijkstra算法相同。**

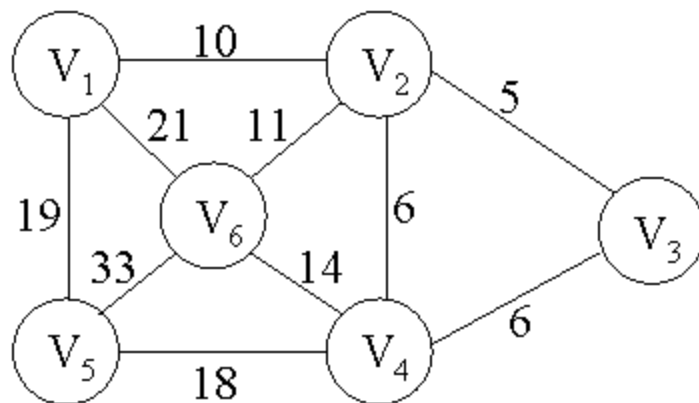


## 6.6.2 Kruskal算法

### ■ Kruskal算法的基本思想是：








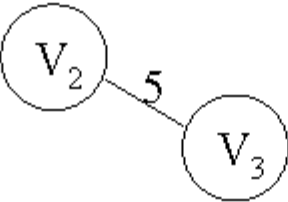




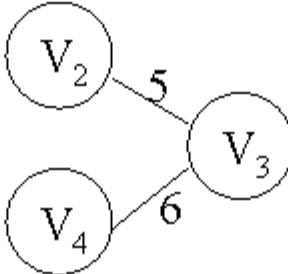


- 对于图 $G = (V, E)$ ，开始时，将顶点集分为 $|V|$ 个等价类，每个等价类包括一个顶点；
- 然后，以权的大小为顺序处理各条边，如果某条边连接两个不同等价类的顶点，则这条边被添加到**MST**，两个等价类被合并为一个；
- 反复执行此过程，直到只剩下一个等价类。

## 6.6.2 Kruskal算法（续）



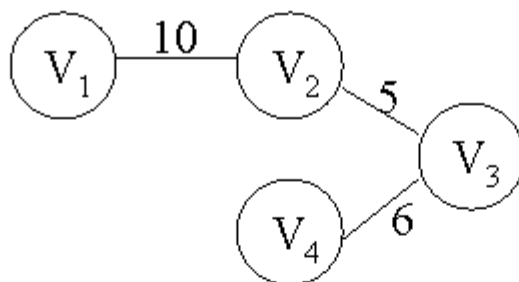
对上图用Kruskal算法，其处理过程见下图

## 6.6.2 Kruskal算法（续）

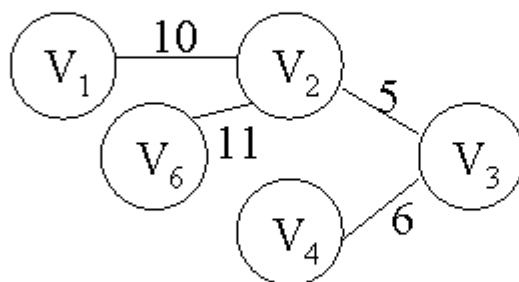
初始状态:						
步骤1: 处理边( $V_2, V_3$ )						
步骤2: 处理边( $V_3, V_4$ )						

## 6.6.2 Kruskal算法（续）

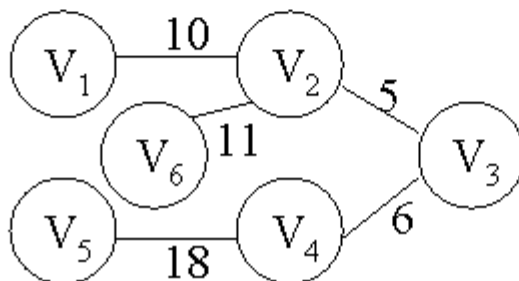
步骤3:  
处理边( $V_1, V_2$ )



步骤4:  
处理边( $V_2, V_6$ )



步骤5:  
处理边( $V_4, V_5$ )





## 6.6.2 Kruskal算法（续）

---

**//最小支撑树的Kruskal算法**

```
void Kruskal(Graph& G, Edge* &MST )  
{
```

```
    Partree A(G.VerticesNum()); //等价类
```

```
        //最小值堆（minheap）
```

```
        MinHeap<Edge> H(G.EdgesNum());
```

```
    //最小支撑树
```

```
    MST=new Edge[G.VerticesNum()-1];
```

```
    int MSTtag=0;
```

```
        //最小支撑树边的标号
```



## 6.6.2 Kruskal算法（续）

---

```
//将图的所有边插入最小值堆H中
for(int i=0; i<G.VerticesNum(); i++)
{
    for(Edge e= G. FirstEdge(i);
        G.IsEdge(e);e=G. NextEdge(e))
    {
        if(G.FromVertex(e)< G.ToVertex(e))
            H.Insert(e);
    }
}
```



## 6.6.2 Kruskal算法（续）

```
int EquNum=G.VerticesNum(); //开始时有|V|个等价类
while(EquNum>1)              //合并等价类
{
    Edge e=H.RemoveMin();    //获得下一条权最小的边
    if(e.weight>=INFINITY)
    {
        Print("不存在最小支撑树.");
        delete [] MST;        //释放空间
        MST=NULL;             //MST是空数组
        return;
    }
}
```





## 6.6.2 Kruskal算法（续）

```
int from=G.FromVertex(e);           //记录该条边的信息
int to= G.ToVertex(e);
if(A.differ(from,to)) //如果边e的两个顶点不在一个等价类
{
    //将边e的两个顶点所在的两个等价类合并为一个
    A.UNION(from,to);

    //将边e加到MST
    AddEdgetoMST(e,MST,MSTtag++);

    //将等价类的个数减1
    EquNum--;
}
}
```



## 6.6.2 Kruskal算法（续）

- 使用了路径压缩，`diff`和`UNION`函数几乎是常数
- 假设可能对几乎所有边都判断过了，则最坏情况下算法时间代价为  $O(|E| \log |E|)$ ，即堆排序的时间
- 通常情况下只找了接近顶点数目那么多边的情况下，MST就已经生成，时间代价接近于  $O(|V| \log |E|)$



# 总结

---

- **6.1** 图的基本概念
- **6.2** 图的抽象数据类型
- **6.3** 图的存储结构
- **6.4** 图的周游
- **6.5** 最短路径问题
- **6.6** 最小支撑树