第7章 图

主要内容

7.1 图的定义和术语

- 图(Graph)由表示数据元素的集合V和表示数据之间关系的集合E组成，记为G = <V，E>
- 在图中，数据元素通常称作顶点(vertex)，V就是顶点的有穷非空集合
- 顶点的序偶，称之为边(edge)，E是边的集合
- 有向图、带权图、稀疏图、稠密图、完全图、连通图

7.1 图的定义和术语

- 若代表一条边的顶点序偶是无序的(即该边无方向)，则称此图为无向图。
- 若代表一条边的顶点序偶是有序的(即边有方向)，则称此图为有向图。

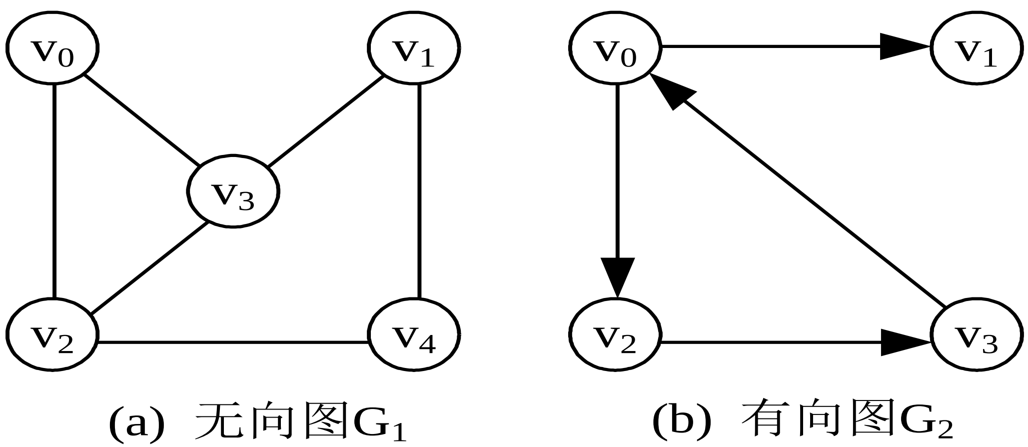
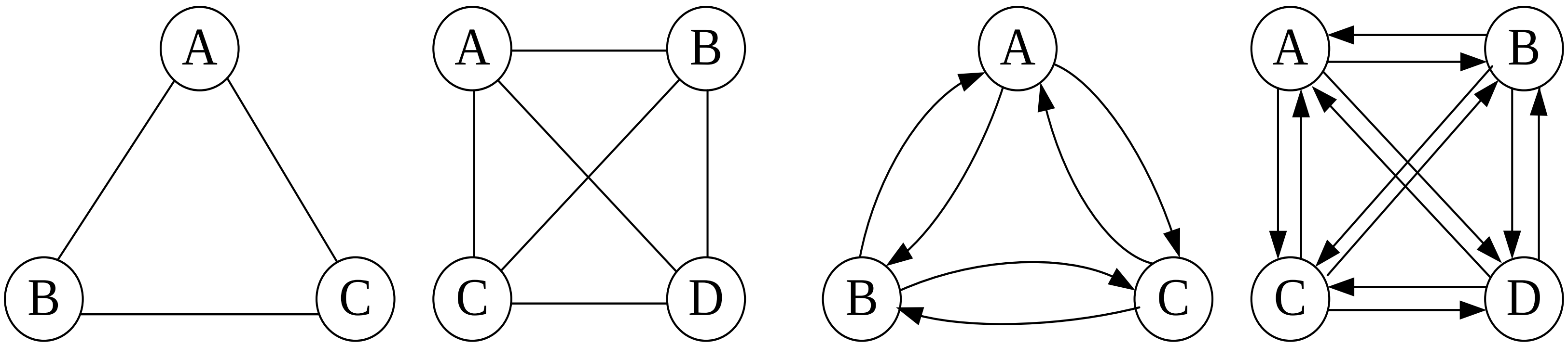


图7.2 图的示例

7.1 图的定义和术语

- 通常用**n**表示图中顶点的数目，用**e**表示边或弧的数目。无向图中**e**的取值范围是从**0**到**n(n - 1)/2**，有向图中**e**的取值范围是从**0**到**n(n - 1)**
- 边数相对较少的图称为稀疏图(sparse graph)
- 边数相对较多的图称为稠密图(dense graph)
- 任何两顶点间都有边相关联的图称为完全图(complete graph)，完全图显然具有最大的边数

7.1 图的定义和术语

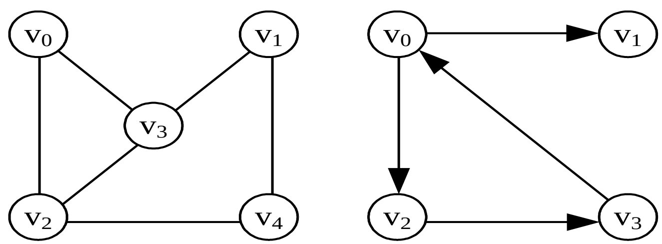


（a）无向完全图　　（b）有向完全图

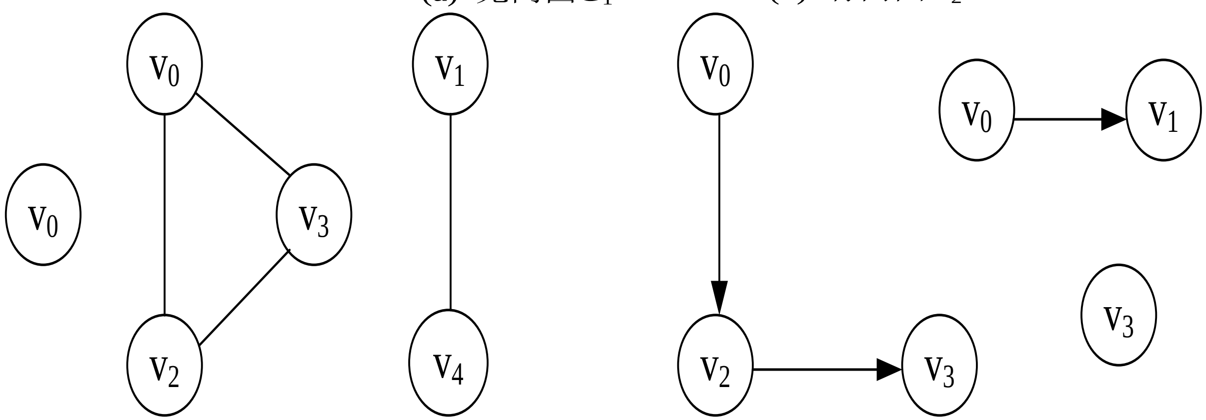
图7.3 完全图

7.1 图的定义和术语

- **设G = <V,E>是一个图，若E′是E的子集，V′是V的子集，且E′中的边仅与V′中顶点相关联，则图G′= (V′,E′)称为图G的**子图(subgraph)。



(a) 无向图G_1　　(b) 有向图G_2



(a) 无向图 G_1的若干子图　　(b) 有向图 G_2的若干子图

图7.4 图7.2的若干子图

7.1 图的定义和术语

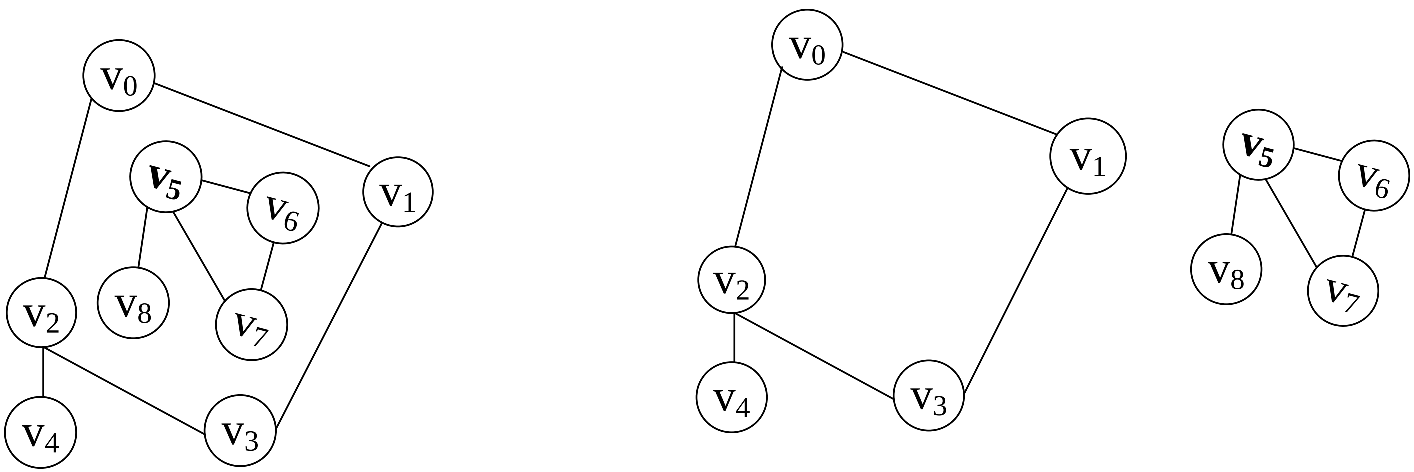
- 无向图G = <V,E>中从顶点vp到顶点vq的路径(path)是一个顶点序列($v_p = v_{i0}$，v_{i1}，v_{i2}，…，$v_{im} = v_q$)，其中(v_{ij-1}，v_{ij})∈E，1≤j≤m。若G是有向图，则路径也是有向的，顶点序列应满足$<v_{ij-1}, v_{ij}>$∈E，1≤j≤m
- 路径长度(length)定义为路径上的边（或弧）的数目
- 第一个顶点和最后一个顶点相同的路径称为回路或环(cycle)
- 序列中顶点不重复出现的路径称为简单路径(simple path)

7.1 图的定义和术语

- 除了第一个顶点和最后一个顶点外，其余顶点不重复的回路，称为简单回路(simple cycle)
- 不带回路的图称为无环图(acyclic graph)
- 不带回路的有向图称为有向无环图(directed acyclic graph，简记为DAG)
- 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他所有顶点，则称此有向图为有根的图，v_0称作图的根

7.1 图的定义和术语

- 在无向图中，如果从顶点v_i到顶点v_j有路径，则称v_i和v_j是连通的(connected)。
- 如果对于图中的任意两个顶点$v_i, v_j \in V$，v_i和v_j都是连通的，则称无向图G 为连通图。
- 连通分量(connected component)定义为无向图中的极大连通子图。



(a) 非连通的无向图G3　　（b） 非连通无向图G3的连通分量

图7.5 非连通无向图的连通分量示意图

7.1 图的定义和术语

- 对于有向图G = <V，E>，若G中任意两个顶点v_i和$v_j(v_i \neq v_j)$，都有一条从v_i到v_j的有向路径，同时还有一条从v_j到v_i的有向路径，则称有向图G是强连通图。有向图强连通的极大子图称为该有向图的强连通分支或者强连通分量。

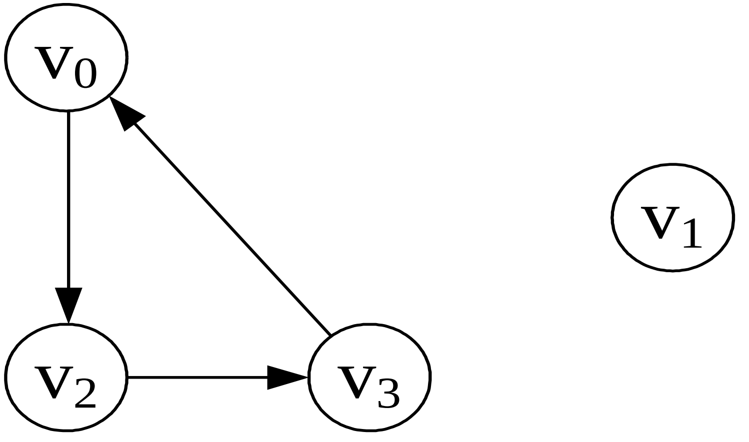
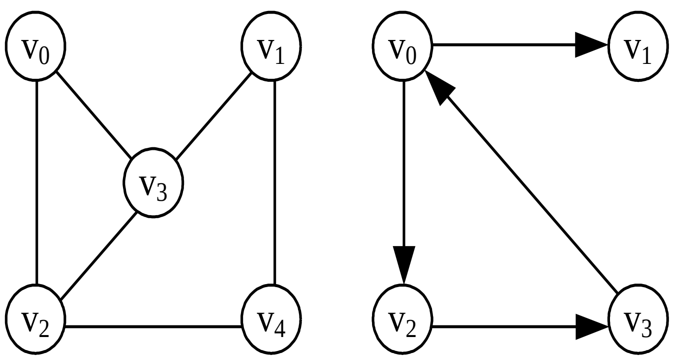


图7.6　有向图G_2的两个强连通分量

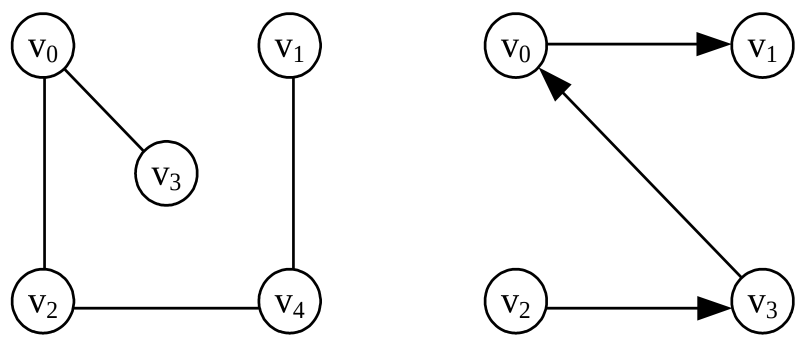
7.1 图的定义和术语

- 一个连通图的生成树是含有该连通图全部顶点的一个极小连通子图。
- 若连通图G的顶点个数为n，则G的生成树的边数为n-1。但是有n-1条边的图不一定是生成树。如果无向图G的一个生成树G′上添加一条边，则G′中一定有环，因为依附于这条边的两个顶点有另一条路径。相反，如果G′的边数小于n-1，则G′一定不连通。



(a) 无向图G_1　　(b) 有向图G_2

图7.2 图的示例

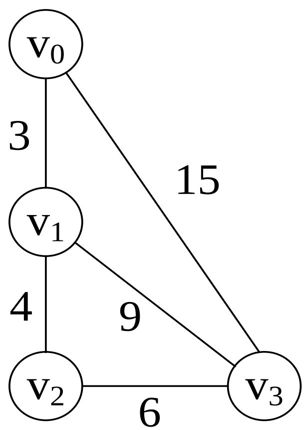


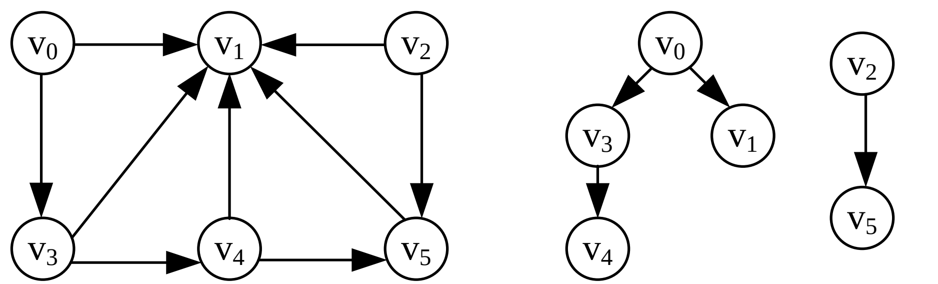
(a) 无向图G_1的生成树　　(b) 有向图G_2的生成树

图7.7　图7.2中无向图和有向图的生成树示例

7.2 图的抽象数据类型

- 自由树(free tree)是不带简单回路的无向图，它是连通的，并且具有**n - 1**条边。网络(network)是带权的连通图，图**7.8**中的**G4**是一个网络。
- 如果一个有向图只有一个顶点的入度为**0**，其余顶点的入度均为**1**，则称为有向树。一个有向图的生成森林由若干棵有向树组成，这些树的并集包含了原图所有顶点，各有向树的弧不相交。图**7.9**就是有向图生成森林的示例。





（a）有向图　（b）有向图的生成森林

图7.8 网络实例G4　图7.9 有向图及其生成森林

7.2 图的抽象数据类型

【代码7.1】 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);
```

7.2 图的抽象数据类型

```
//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);
```

7.2 图的抽象数据类型

```
//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);
};
```

7.3 图的存储结构

- 7.3.1 相邻矩阵
- 7.3.2 邻接表
- 7.3.3 十字链表

7.3.1 相邻矩阵

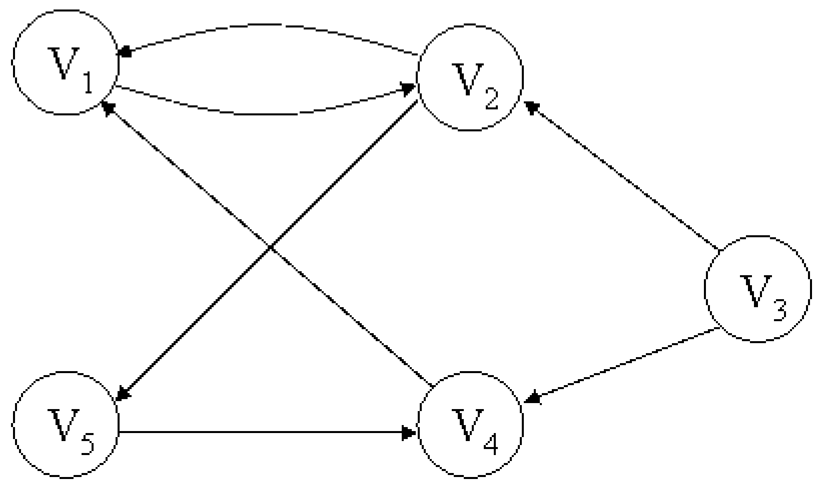
图的相邻矩阵(adjacency matrix，**或**邻接矩阵)**表示顶点之间的邻接关系，即表示顶点之间有边或没有边的情况。**

设G = <V，E>是一个有n个顶点的图，则图的相邻矩阵是一个二维数组A[n，n]，定义如下：

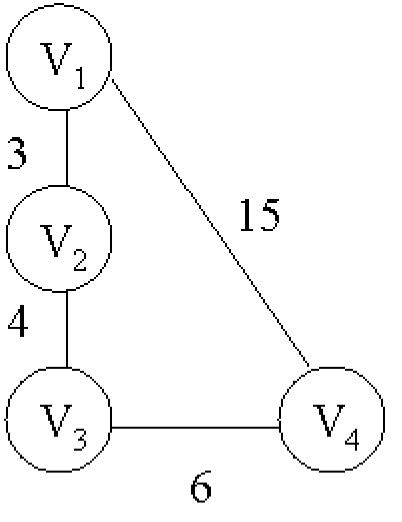
$$A[i, j]=\begin{cases}1, & 若 (V_i, V_j) \in E 或 \langle V_i, V_j\rangle \in E \\ 0, & 若 (V_i, V_j) \notin E 或 \langle V_i, V_j\rangle \notin E\end{cases}$$

- 对于n个顶点的图，相邻矩阵的空间代价都为O(n2)，与边数无关。

$$A7=\begin{bmatrix}0&1&0&0&0\\1&0&0&0&1\\0&1&0&1&0\\1&0&0&0&0\\0&0&0&1&0\end{bmatrix}$$



$$A4=\begin{bmatrix} 0 & 3 & 0 & 15 \\ 3 & 0 & 4 & 0 \\ 0 & 4 & 0 & 6 \\ 15 & 0 & 6 & 0 \end{bmatrix}$$



7.3.1 相邻矩阵

【代码7.2】 图的基类

```
class Edge  {                          // 边类
public:
    int from, to, weight;       // from是边的始点,to是终点,weight是边的权
    Edge()  {                          // 缺省构造函数
        from = -1;  to = -1;  weight = 0;
    }
    Edge(int f,int t,int w)  {                                  // 给定参数的构造函数
        from = f;   to = t;   weight = w;
    }
};
class Graph  {
public:
    int numVertex;                              // 图中顶点的个数
    int numEdge;                                // 图中边的条数
    int *Mark;                                  // 标记图的顶点是否被访问过
    int *Indegree;                              // 存放图中顶点的入度
    Graph(int numVert)  {                       // 图的构造函数
    numVertex = numVert;
    numEdge = 0;
```

7.3.1 相邻矩阵

```
    Indegree = new int[numVertex];
    Mark = new int[numVertex];
    for (int i = 0; i < numVertex; i++) {
            Mark[i] = UNVISITED;              // 标志位设为UNVISITED
            Indegree[i] = 0;                  // 入度设为0
    }
    }

    ~Graph() {                                // 析构函数
    delete [] Mark;                           // 释放Mark数组
    delete [] Indegree;                       // 释放Indegree数组
    }

   int VerticesNum() {                        // 返回图中顶点的个数
    return numVertex;
    }

   bool IsEdge(Edge oneEdge) {                // oneEdge是否是边
    if (oneEdge.weight > 0 && oneEdge.weight < INFINITY && oneEdge.to >= 0)
    return true;
    else return false;
    }
};
```

[代码7.3] 用相邻矩阵表示图

```
class Graphm: public Graph  {
private:
    int **matrix;  // 指向相邻矩阵的指针
public:
    Graphm(int numVert):Graph(numVert)  {          // 构造函数
        int i,j;        // i, j作为for循环中的计数器
        matrix = (int**)new int*[numVertex];
                // 申请matrix数组行向量数组
        for (i = 0;i < numVertex;i++)
                // 申请matrix数组行的存储空间
                 matrix[i] = new int[numVertex];
        for (i = 0;i < numVertex;i++)
        // 相邻矩阵的所有元素都初始化为0
                for (j = 0;j < numVertex;j++)
                        matrix[i][j] = 0;
    }
```

```
Edge FirstEdge(int oneVertex)  {            // 返回顶点oneVertex的第一条边
    Edge myEdge;
    myEdge.from = oneVertex;                // 将顶点oneVertex作为边的始点
// 寻找第一个使得matrix[oneVertex][i]不为0的i值
for (int i = 0; i < numVertex; i ++) {
        if (matrix[oneVertex][i] != 0) {
                myEdge.to = i;
                myEdge.weight = matrix[oneVertex][i];
                break;  //找到了顶点oneVertex的第一条边
        }
    }
    return myEdge;
}
```

```
Edge NextEdge(Edge preEdge)  {          // 返回与边有相同关联顶点的下一条边
    Edge myEdge;                  // myEdge的初始成员变量to为-1
    myEdge.from = preEdge.from;       // 置边的始点为与上一条边preEdge相同
    if (preEdge.to < numVertex) {
     // 如果preEdge.to+1 >= numVertex,那么就不存在下一条边了
                for (int i = preEdge.to+1;i < numVertex;i++) {
                // 寻找下一个使得matrix[preEdge.from][i]不为0的i值，即下一条边
                      if (matrix[preEdge.from][i] != 0) {
                            myEdge.to=i;
                            myEdge.weight = matrix[preEdge.from][i];
                            break;
                      }
                }
        }
return myEdge;
}
```

7.3.2 邻接表

- 当图中的边数较少时，相邻矩阵就会出现大量的零元素，存储这些零元素将耗费大量的存储空间。对于稀疏图，可以采用邻接表存储法。
- 邻接表(adjacency list)表示法是一种链式存储结构，由一个顺序存储的顶点表和n个链接存储的边表组成。
  - 顶点表目有两个域：顶点数据域和指向此顶点边表指针域
  - 边表把依附于同一个顶点vi的边（即相邻矩阵中同一行的非0元素）组织成一个单链表。边表中的每一个表目都代表一条边，由两个主要的域组成：
    - 与顶点vi邻接的另一顶点的序号
    - 指向边表中下一个边表目的指针

7.3.2 邻接表

- 顶点结点和边（或弧）结点的结构如下所示：

顶点结点

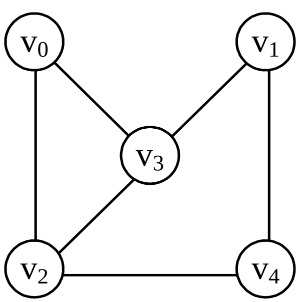
data	firstarc

边（或弧）结点

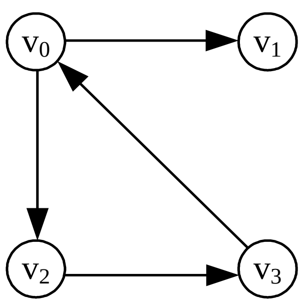
adjvex	nextarc	Info

7.3.2 邻接表

假设（v_i，v_j）是无向图中的一条边，在顶点v_i的边表里存储有边（v_i，v_j）对应的边结点，在顶点v_j的边表里还需存放（v_j，v_i）对应的边结点。因此，对于无向图，同一条边在邻接表中出现两次。



(a) 无向图G_1



(b) 有向图G_2

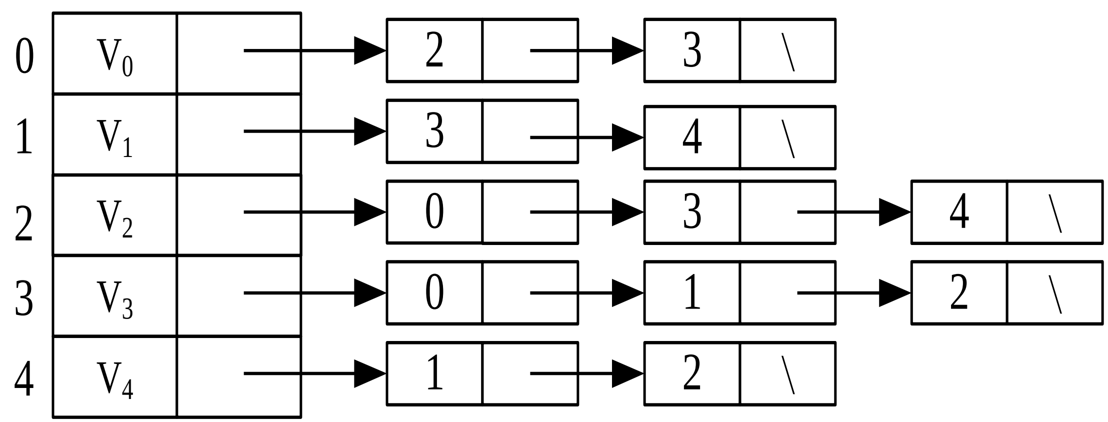
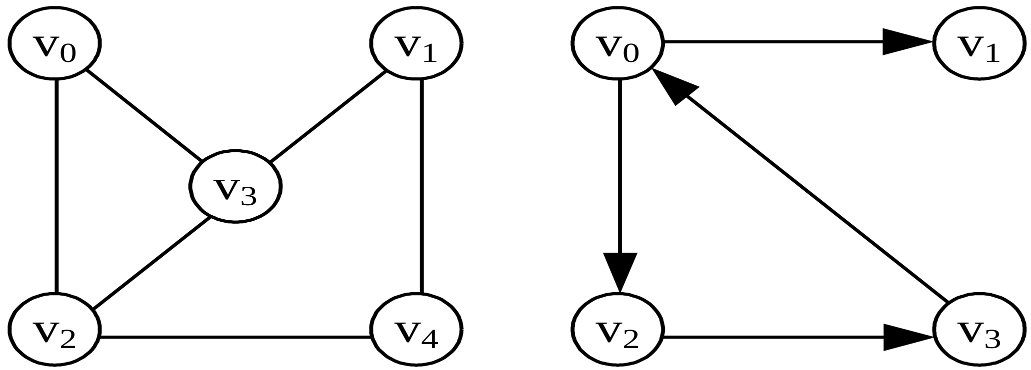
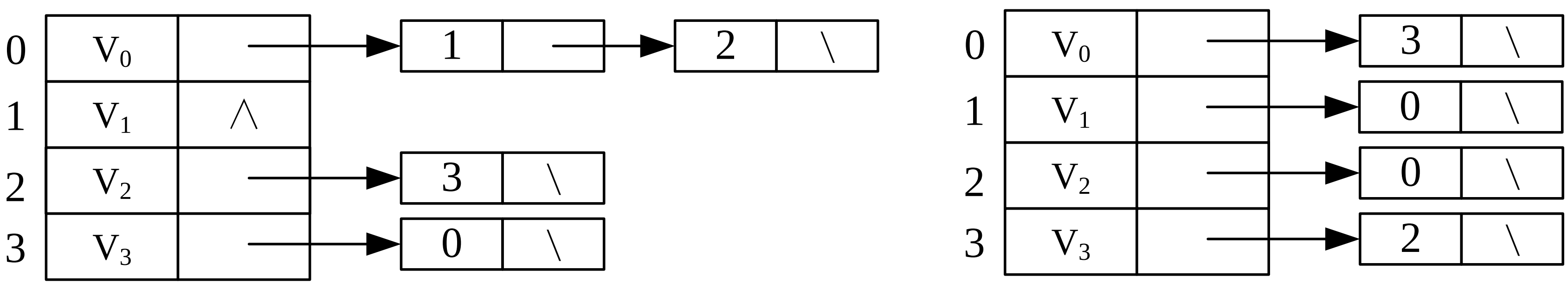


图7.12 无向图G_1的邻接表表示

7.3.2 邻接表



(a) 无向图G_1　　(b) 有向图G_2



（a）有向图G_2的邻接表　　（b）有向图G_2的逆邻接表

图7.13 有向图G2的邻接表和逆邻接表表示

7.3.2 邻接表

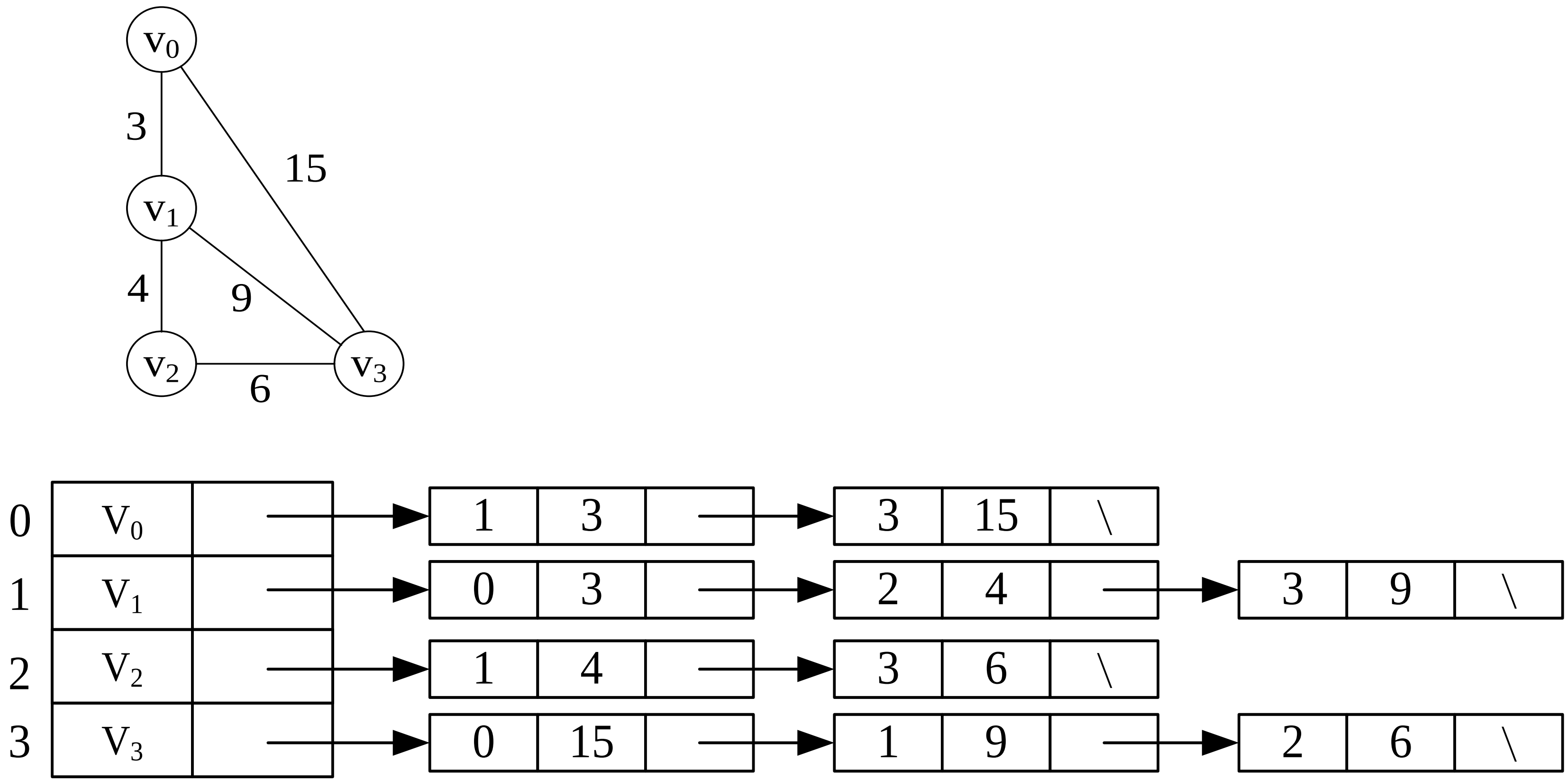


图7.14　带权图G4的邻接表表示

7.3.2 邻接表

- 若图中有n个顶点和e条边，如果该图是无向图，则需要用到n个顶点结点和2e个边结点；
- 若是有向图时，不考虑逆邻接表，只需要n个顶点结点和e个边结点。
- 当边数e很小时，可以节省大量的存储空间。
- 边表中表目顺序往往按照顶点编号从小到大排列。

[代码7.4] 用邻接表存储图

```
struct listUnit  {                  // 邻接表表目中数据部分的结构定义
    int vertex;                     // 边的终点
    int weight;                     // 边的权
};
template<class Elem>
class Link  {                       // 链表元素
public:
    Elem element;                   // 表目的数据
    Link *next;                     // 表目指针，指向下一个表目
    Link(const Elem& elemval,Link *nextval = NULL)  {          // 构造函数
        element = elemval;
        next = nextval;
    }
    Link(Link *nextval = NULL)  {            // 构造函数
    next = nextval;
    }
};
```

```
template<class Elem>
class LList  {                                  // 链表类
public:
    Link<Elem> *head;                           // head保存一个虚的头结点，以方便操作
    LList()  {                                  // 构造函数
        head = new Link<Elem>();
    }
};
class Graphl: public Graph  {
private:
    LList<listUnit> *graList;                             // 保存所有边表的数组
public:
    Graphl(int numVert):Graph(numVert)  {                             // 构造函数
        graList = new LList<listUnit>[numVertex];                     // 为边表graList数组申请空间
    }
    Edge FirstEdge(int oneVertex)  {                      // 返回顶点oneVertex的第一条边
        Edge myEdge;                                      //  myEdge的初始成员变量to为-1
        myEdge.from = oneVertex;                                      // 将顶点oneVertex作为边myEdge的始点
        Link<listUnit> *temp = graList[oneVertex].head;
        if (temp->next != NULL) {                                     // 顶点oneVertex边表非空
            myEdge.to = temp->next->element.vertex;                   // 边表第一个表目的顶点
            myEdge.weight = temp->next->element.weight;
        }
        return myEdge;                                               // 如果没有找到第一条边，myEdge.to=-1
    }
```

```
Edge NextEdge(Edge preEdge) {                          // 返回与边有相同关联顶点的下一条边
    Edge myEdge;                                       // myEdge的初始成员变量to为-1
    myEdge.from = preEdge.from;                        // 将边的始点置为与上一条边的相同
    Link<listUnit> *temp = graList[preEdge.from].head;      // temp指向边表头一个
    while (temp->next != NULL && temp->next->element.vertex <= preEdge.to)
        temp = temp->next;                             // 确定边preEdge的位置
    if (temp->next != NULL) {                               // 边preEdge的下一条边存在
        myEdge.to = temp->next->element.vertex;
        myEdge.weight = temp->next->element.weight;
    }
    return myEdge;
}
```

```
void setEdge(int from,int to,int weight)  {                // 为图设定一条边
    Link<listUnit> *temp = graList[from].head;        // temp指向边表头一个
    while (temp->next != NULL && temp->next->element.vertex < to)
            temp = temp->next;                                     // 确定边(from,to)在边表中的位置
    if (temp->next == NULL) {                                   // 边(from,to)在边表中不存在
            temp->next = new Link<listUnit>;           // 在边表最后加入这条新边
            temp->next->element.vertex = to;
            temp->next->element.weight = weight;
            numEdge++;
            Indegree[to]++;
            return;
    }
    if (temp->next->element.vertex == to) {              // 边(from,to)在边表中已存在
            temp->next->element.weight = weight;                       // 只需要改变边的权值
            return;
    }
    if (temp->next->element.vertex > to) {               // 边(from,to)在边表中不存在
            Link<listUnit> *other = temp->next;
            temp->next = new Link<listUnit>;
            temp->next->element.vertex = to;
            temp->next->element.weight = weight;
            temp->next->next = other;                                  // 连接边表中其后的其他边
            numEdge++;
            Indegree[to]++;
            return;
    }
}
```

```
void delEdge(int from,int to)  {                                  // 删掉图的一条边
    Link<listUnit> *temp = graList[from].head;        // temp指向边表头一个
    while (temp->next != NULL && temp->next->element.vertex < to)
            temp = temp->next;                                    // 确定边(from,to)在边表中的位置
    if (temp->next == NULL)
            return;                                               // 边(from,to)在边表中不存在，直接返回
    if (temp->next->element.vertex > to)
            return;                                               // 边(from,to)在边表中不存在，直接返回
    if (temp->next->element.vertex == to) {           // 边(from,to)在边表中存在
            Link<listUnit> *other = temp->next->next;
            delete temp->next;                                    // 从边表中将其删掉
            temp->next = other;                                   // 其他表目挂接
            numEdge--;                                            // 边数减1
            Indegree[to]--;                                       // 终点的入度减1
            return;
    }
}
};
```

7.3.3 十字链表

- 十字链表（Orthogonal List）是有向图的另一种链式存储结构，可以看成是邻接表和逆邻接表的结合
- 表中对应于有向图的每一条弧有一个表目，共有5个域：头(headvex)和尾(tailvex)分别表示弧头（终点）和弧尾（始点）顶点序号；tailnextarc链接指针指向下一条顶点以tailvex为弧尾的弧；headnextarc指针指向下一条以顶点headvex为弧头的弧；此外还有一个表示弧权值等信息的info域
- 顶点表目由3个域组成：data域存放顶点的相关信息；firstinarc链接指针指向第一条以该顶点为终点的弧；firstoutarc链接指针指向第一条以该顶点为始点的弧。所有的顶点也可以放在顺序存储结构中

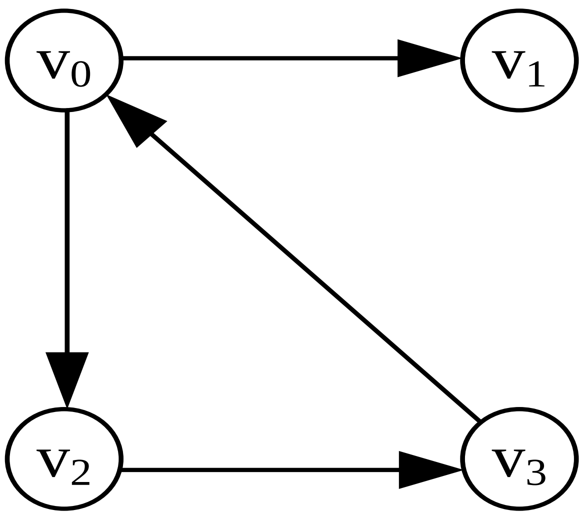
data	firstinarc
	firstoutarc

顶点结点

tailvex	tailnextarc	headvex	headnextarc	info

弧(有向边)结点

7.3.3 十字链表



(b) 有向图G_2

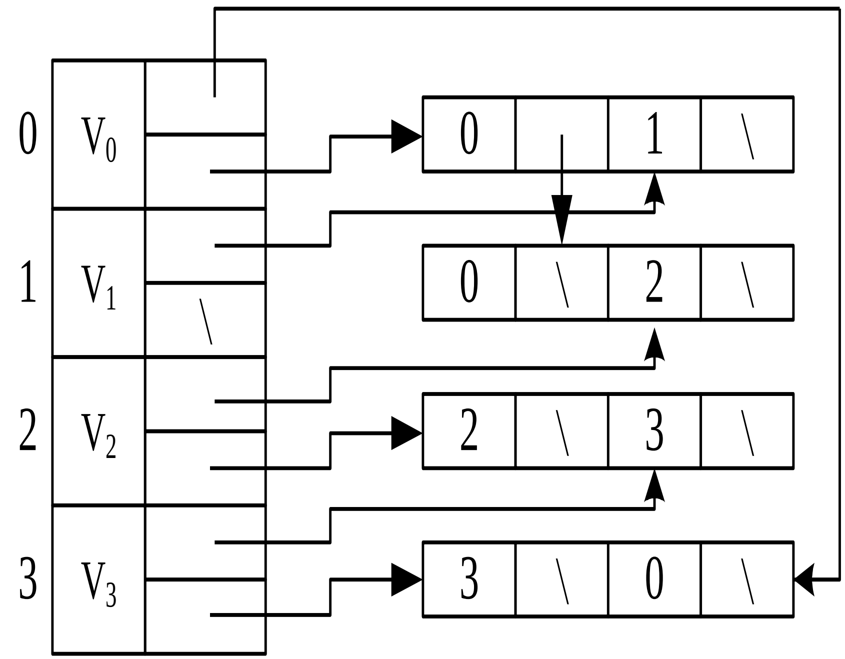


图7.15 有向图G2的十字链表示例

7.3.3 十字链表

- 在十字链表中，很容易找到以vi为始点和终点的弧。
- 从顶点结点vi的firstoutarc出发，由tailnextarc域链接起来的链表，正好是原来的邻接表结构。统计这个链表中的表目个数，可以得到顶点vi的出度。
- 如果从顶点结点vi的firstinarc出发，由headnextarc域链接起来的链表，恰好是原来的逆邻接表结构。统计这个链表中的表目个数，可以求出顶点vi的入度。

<table>
<tr><td rowspan="2">data</td><td>firstinarc</td></tr>
<tr><td>firstoutarc</td></tr>
</table>

顶点结点

tailvex	tailnextarc	headvex	headnextarc	info

弧(有向边)结点

7.4 图的周游

- 图的周游（graph traversal）
  - 图的周游是指从图中的某一个顶点出发，按照一定的策略访问图中的每一个顶点，使得每一个顶点访问且只被访问一次。图的周游算法是求解图的连通性问题、拓扑排序和关键路径等问题的基础。
- 深度优先周游和广度优先周游是两种基本的周游方法，对有向图和无向图都适用

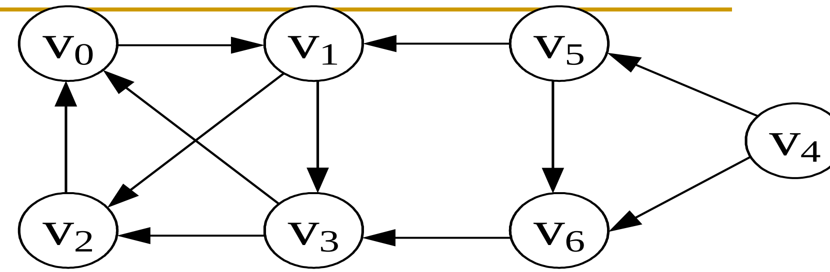
7.4 图的周游

- **7.4.1 深度优先周游**
- **7.4.2 广度优先周游**
- **7.4.3 拓扑排序**

7.4.1 深度优先周游

- 深度优先搜索（depth-first search，简称DFS）类似于树的先根次序周游。深度优先周游方法的特点是尽可能先对纵深方向进行搜索。
- 基本思想
  - 首先选取一个顶点开始搜索。设$v_0 \in V$是源点，访问顶点v_0并标记为VISITED，然后访问v_0邻接到的未被访问过的顶点v_1
  - 再从v_1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顶点都被访问过了的顶点w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u
  - 再从u出发递归地按照深度优先的方式周游
  - 重复上述过程直至从v_0出发的所有边都已检测过为止，此时，图中所有由源点有路径可达的顶点都已被访问过

7.4.1 深度优先周游



- 从顶点v0出发进行搜索，将顶点v0的标记位置为VISITED，然后从v0的邻接表中取出仅有的邻接点v1。因为v1从未曾访问，将顶点v1的标志位记为VISITED，然后从顶点v1的邻接表中选取一个邻接点
- 假设在顶点v1的邻接表中v2排在v3之前，于是先访问v2，将顶点v2的标志位记为VISITED
- 因为v2的邻接表中除了已被访问的顶点v0外，没有其他的邻接点，所以搜索过程回退到顶点v1，找到下一个邻接顶点v3，将顶点v3的标志位记为VISITED
- 查v3的邻接表时，两个邻接点v0和v2都已经被访问，于是回退到顶点v1，此时也出现了相同的情况，最后回退到顶点v0
- 要完成图的周游，还有其他顶点没有被访问过，因此还得继续找一个UNVISITED顶点继续深度优先搜索。比如此时首先访问v4，那么继续深度搜索，依次访问顶点v5和v6

7.4.1 深度优先周游

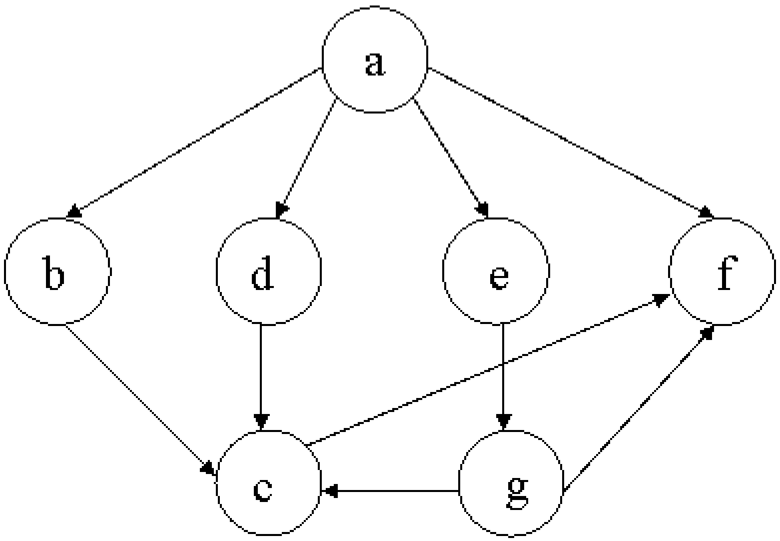
【算法7.5】 图的深度优先周游(DFS)算法

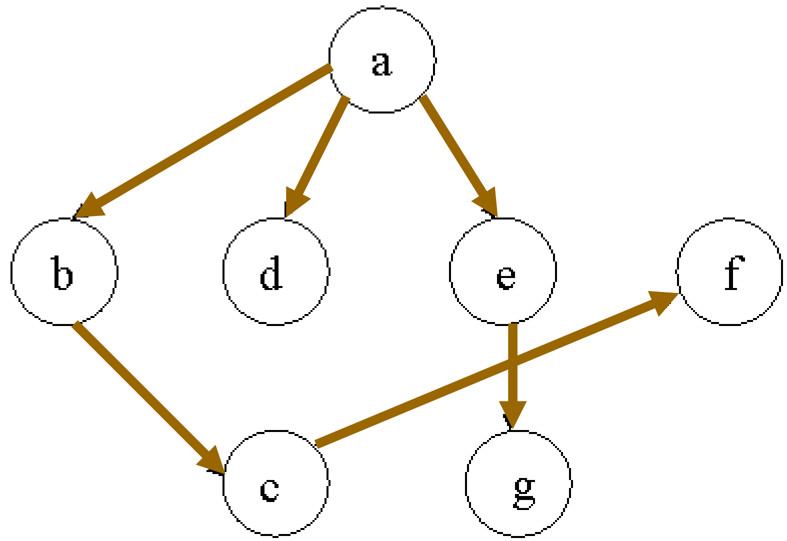
```
void DFS(Graph& G, int v)  {                    // 深度优先搜索的递规实现
    G.Mark[v] = VISITED;                        // 将标记位设置为VISITED
    Visit(G,v);                                 // 访问顶点v
    for (Edge e = G.FirstEdge(v);G.IsEdge(e);e = G.NextEdge(e))
      if (G.Mark[G.ToVertex(e)] == UNVISITED)
          DFS(G, G.ToVertex(e));
}
```

7.4.1 深度优先周游

- 周游图的过程实质上是搜索每个顶点的邻接点的过程，时间主要耗费在从该顶点出发搜索它的所有邻接点上
- 分析上述算法，对于具有**n**个顶点和**e**条边的无向图或有向图，深度优先周游算法对图中每顶点至多调用一次**DFS**函数
  - 用相邻矩阵表示图时，共需检查n^2个矩阵元素，所需时间为$O(n^2)$
  - 用邻接表表示图时，找邻接点需将邻接表中所有边结点检查一遍，需要时间**O(e)**，对应的深度优先搜索算法的时间复杂度为**O(n+e)**

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

7.4.2 广度优先周游

- 广度优先搜索(breadth-first search，简称BFS)。其周游的过程是：
  - 从图中的某个顶点v出发，访问并标记了顶点v之后
  - 横向搜索v的所有邻接点u1，u2，…，ut
  - 在依次访问v的各个未被访问的邻接点之后，再从这些邻接点出发，依次访问与它们邻接的所有未曾访问过的顶点
  - 重复上述过程直至图中所有与源点v有路径相通的顶点都被访问为止

7.4.2 广度优先周游

【算法7.6】图的广度优先周游(BFS)算法

```
void BFS(Graph& G, int v)  {
    using std::queue;                   // 使用STL中的队列
    queue<int> Q;
    Visit(G,v);                         // 访问顶点v
    G.Mark[v] = VISITED;                // 将标记位设置为VISITED
    Q.push(v);                          // 顶点v入队列
    while (!Q.empty()) {                // 如果队列非空
        int u = Q.front ();             // 获得队列顶部元素
```

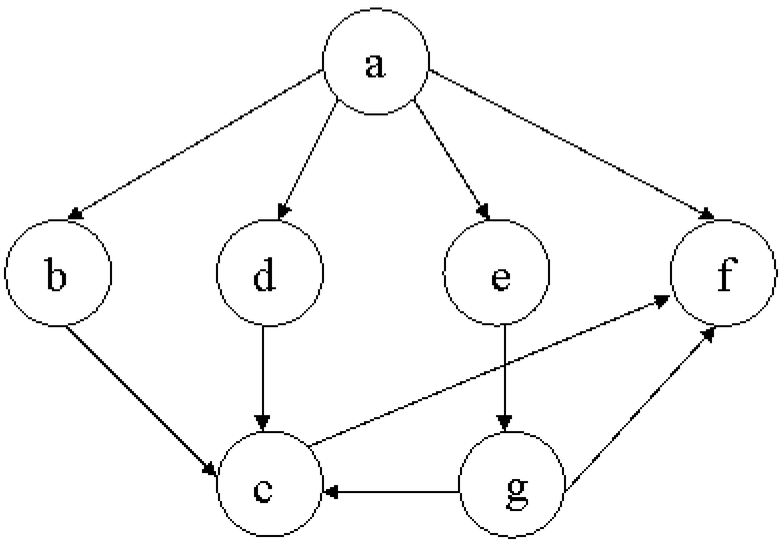
7.4.2 广度优先周游

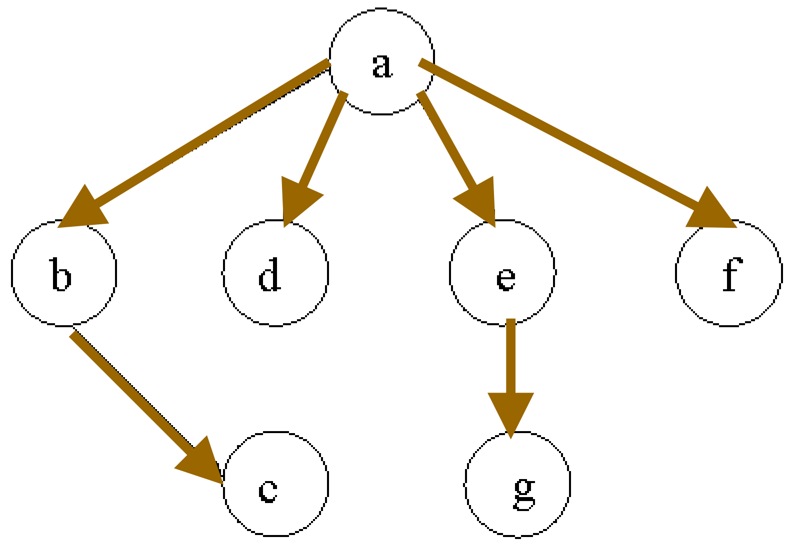
```
        Q.pop();                        // 队列顶部元素出队
        // 与该顶点邻接的所有未访问顶点入队
        For (Edge e = G.FirstEdge(u);G.IsEdge(e);e = G.NextEdge(e))
                if (G.Mark[G.ToVertex(e)] == UNVISITED) {
                        Visit(G, G.ToVertex(e));
                        G.Mark[G.ToVertex(e)] = VISITED;
                        Q.push(G.ToVertex(e));
                }
    }
}
```

7.4.2 广度优先周游

- 广度优先搜索实质上与深度优先相同，只是访问顺序不同而已。二者时间复杂度也相同。

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

7.4.3 拓扑排序

- 一个无环的有向图称为有向无环图(**directed acyclic graph**，简称**DAG**)
- 根据有向图建立拓扑序列的过程称为拓扑排序(**topological sorting**)
- 拓扑排序可以解决先决条件问题，即以某种线性顺序来组织多项任务，以便能够在满足先决条件的情况下逐个完成各项任务

7.4.3 拓扑排序

- 拓扑序列

  有向无环图常用来描述一个过程或一个系统的进行过程。对于有向无环图G = <V，E>，如果顶点序列满足：

  - **如果有向无环图G中存在顶点v_i到v_j的一条路径，那么在序列中顶点v_i必在顶点v_j之前，顶点集合V的这种线性序列称作一个**拓扑序列(topological order)

7.4.3 拓扑排序

课程代号	课程名称	先修课程
C0	高等数学	无
C1	程序设计	无
C2	离散数学	**C0，C1**
C3	数据结构	**C1，C2**
C4	算法语言	**C1**
C5	编译技术	**C3，C4**
C6	操作系统	**C3，C8**
C7	普通物理	**C0**
C8	计算机原理	**C7**

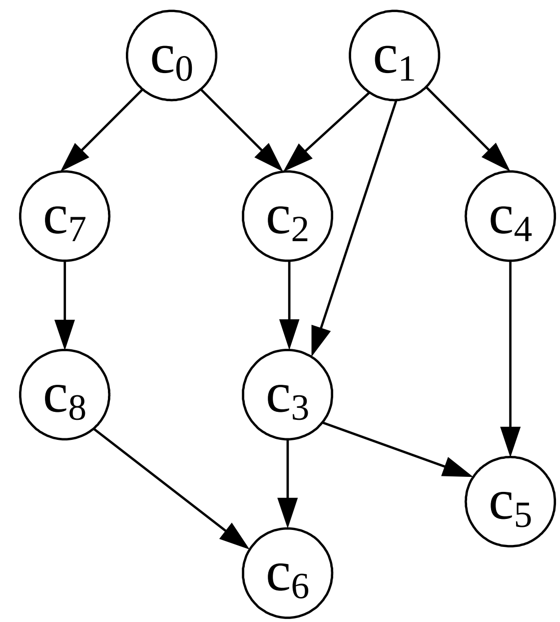


图7. 17 表示课程优先关系的有向无环图

7.4.3 拓扑排序

- 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：
  - 从有向图中选出一个没有前驱（入度为0）的顶点并输出它
  - 删除图中该顶点和所有以它为起点的弧
  - 回到第（1）步继续执行

7.4.3 拓扑排序

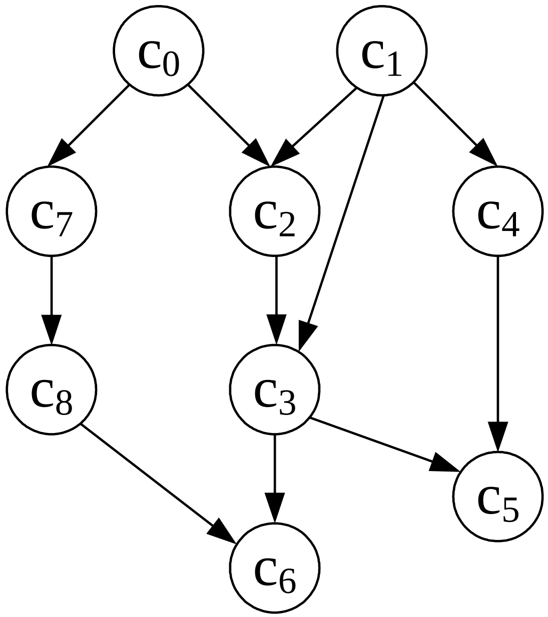


图7. 17 表示课程优先关系的
有向无环图

- 图7.17所示的有向无环图进行拓扑排序可以得到拓扑序列：
  - {c0，c1，c2，c3，c4，c5，c7，c8，c6}，
- 也可以得到：
  - {c0，c7，c8，c1，c4，c2，c3，c6，c5}等
  - 一个有向无环图顶点的拓扑序列可能不是惟一的

7.4.3 拓扑排序

- 不断重复上述两个步骤，会出现两种情形：
  - 要么有向图中顶点全部被输出，要么当前图中不存在没有前驱的顶点。当图中的顶点全部输出时，就完成了有向无环图的拓扑排序；
  - 当图中还有顶点没有输出时，说明有向图中含有环。可见拓扑排序可以检查有向图是否存在环。

7.4.3 拓扑排序

【算法7.7】用队列实现的图拓扑排序

```
void TopsortbyQueue(Graph& G)  {
    for (int i = 0;i < G.VerticesNum();i++) // 初始化Mark数组
        G.Mark[i] = UNVISITED;
    using std::queue;                          // 使用STL中的队列
    queue<int> Q;
    for (i = 0; i < G.VerticesNum(); i++)   // 入度为0的顶点入队
        if (G.Indegree[i] == 0)
            Q.push(i);
```

7.4.3 拓扑排序

```
while (!Q.empty()) {                          // 如果队列非空
        int v = Q.front();                    // 获得队列顶部元素
        Q.pop();                              // 队列顶部元素出队
        Visit(G,v);                           // 问顶点v
        G.Mark[v] = VISITED;                   // 将标记位设置为VISITED
        for (Edge e = G.FirstEdge(v);G.IsEdge(e);e = G.NextEdge(e)) {
                G.Indegree[G.ToVertex(e)]--;              // 与该顶点相邻的顶点入度减
  1
                if (G.Indegree[G.ToVertex(e)] == 0)   // 如果顶点入度减为0则入队
                Q.push(G.ToVertex(e));
        }
  }
```

7.4.3 拓扑排序

```
for (i = 0; i < G.VerticesNum(); i++) // 利用标记位可以判断图中是否有环
        if (G.Mark[i] == UNVISITED) {
                cout<<" 此图有环！ ";
                break;
        }
}
```

7.4.3 拓扑排序

- 对于有n个顶点和e条边的图，若其存储结构用邻接表表示，实现拓扑排序开始时建立入度为0的顶点队列需要检查所有顶点一次，需要时间O(n)，然后排序中每个顶点输出一次，更新顶点的入度需要检查每条边总计e次，故执行时间为O(n+e)。

7.5 最短路径

- 7.5.1 单源最短路径
- 7.5.2 每对顶点之间的最短路径

7.5.1 单源最短路径

- 给定一个带权图G = <V，E>，其中每条边（v_i，v_j）上的权W[v_i，v_j]是一个非负实数。另外，给定V中的一个顶点s充当源点。
- 现在要计算从源点s到所有其他各顶点的最短路径，这个问题通常称为单源最短路径(single-source shortest paths)问题。

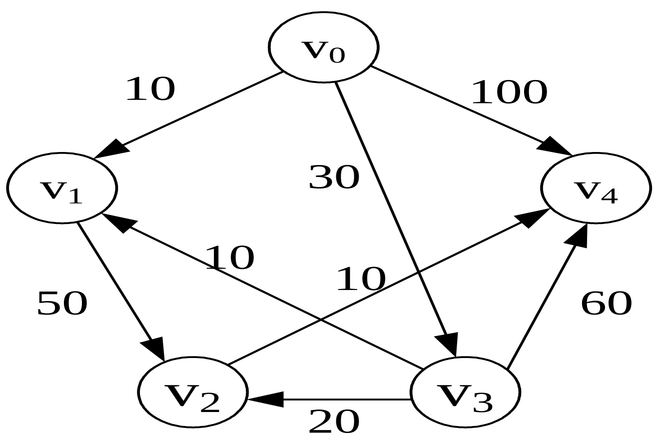


图7.19 单源最短路径的示例

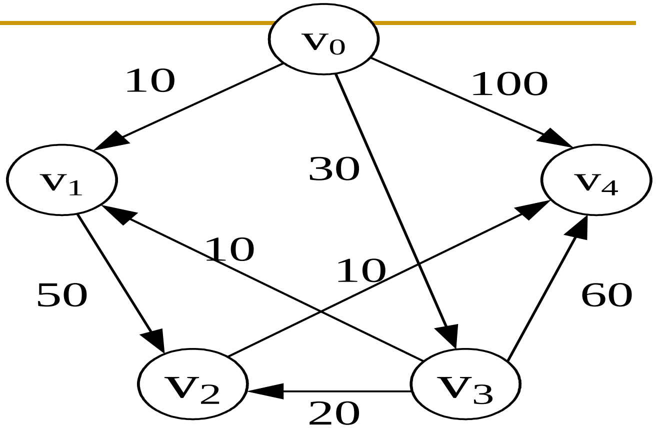
7.5.1 单源最短路径

- 解决单源最短路径问题的一个常用算法是**Dijkstra**算法，它是由**E.W.Dijkstra**提出的一种按路径长度递增的次序产生到各顶点最短路径的贪心算法。
- **E.W.Dijkstra**：
  - **1930**年**5**月**11**日出身于**the Netherlands Rotterdam.**
  - **1972**年获得图灵奖（对开发**ALGOL**做出了重要贡献）
  - 去世于**2002**年**8**月**6**日于**Nuenen, the Netherlands.**

7.5.1 单源最短路径

- **Dijkstra算法基本思想：**
  - 把图的顶点集合化分成两个集合S和V-S。第一个集合S表示最短距离已经确定的顶点集，即一个顶点如果属于集合S当且仅当从源点s到该顶点的最短路径已知
  - 其余的顶点放在另一个集合V-S中。初始时，集合S只包含源点，即S = {s}，这时只有源点到自己的最短距离是已知的。设v是V中的某个顶点，把从源点s到顶点v且中间只经过集合S中顶点的路径称为从源点到v的特殊路径，并用数组D来记录当前所找到的从源点s到每个顶点的最短特殊路径长度
  - 从尚未确定最短路径长度的集合V-S中取出一个最短特殊路径长度最小的顶点u，将u加入集合S，同时修改数组D中由s可达的最短路径长度

7.5.1 单源最短路径



迭代步数	S	v_0	v_1	v_2	v_3	v_4
初始状态	$\{v_0\}$	Length:0 pre:0	length:<u>10</u> pre:0	length:∞ pre:0	length:30 pre:0	length:100 pre:0
1	$\{v_0, v_1\}$	Length:0 pre:0	length:10 pre:0	length:60 pre:1	length:<u>30</u> pre:0	length:100 pre:0
2	$\{v_0, v_1, v_3\}$	Length:0 pre:0	length: 10 pre:0	length:<u>50</u> pre:3	length:30 pre:0	length:90 pre:3
3	$\{v_0, v_1, v_3, v_2\}$	length:0 pre:0	length: 10 pre:0	length:50 pre:3	length:30 pre:0	length:<u>60</u> pre:2
4	$\{v_0, v_1, v_3, v_2, v_4\}$	length:0 pre:0	length: 10 pre:0	length:50 pre:3	length:30 pre:0	length:60 pre:2

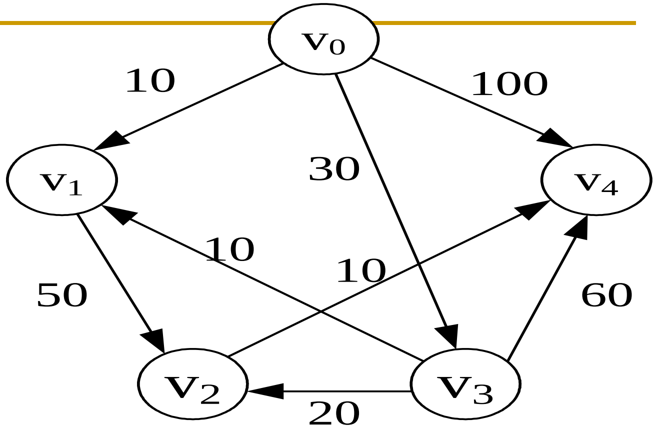
7.5.1 单源最短路径

- **Dijkstra算法基本步骤：**
  - D的初始状态为：如果从源点s到顶点v有弧则D[v]记为弧的权值；否则将D[v]置为无穷大。
  - 每次从尚未确定最短路径长度的集合V-S中取出一个最短特殊路径长度最小的顶点u，将u加入集合S，同时修改数组D中由s可达的最短路径长度：若加进u做中间顶点，使得vi的最短特殊路径长度变短，则修改vi的距离值（即当D[u] + W[u, vi] < D[vi]时，令D[vi] = D[u] + W[u, vi]）。
  - 然后重复上述操作，一旦S包含了所有V中的顶点，D中各顶点的距离值就记录了从源点s到该顶点的最短路径长度。

7.5.1 单源最短路径

- 推导最短路径：
  - 距离数组中还可以设立一个域来记录从源点到顶点v的最短路径上v前面经过的一个顶点，这样就可以推导出最短路径。
  - 初始时，对所有的v ≠ s，均设置其前一个顶点为s。在Dijkstra算法更新最短路径长度时，只要D[u] + W[u,v] < D[v]，就设置v的前一个顶点为u，否则不做修改。
  - 当Dijkstra算法终止时就可以找到源点s到顶点v的最短路径上每个顶点的前一个顶点，从而找到源点s到顶点v的最短路径。

7.5.1 单源最短路径



迭代步数	S	v_0	v_1	v_2	v_3	v_4
初始状态	$\{v_0\}$	Length:0 pre:0	length:10 pre:0	length:∞ pre:0	length:30 pre:0	length:100 pre:0
1	$\{v_0, v_1\}$	Length:0 pre:0	length:10 pre:0	length:60 pre:1	length:30 pre:0	length:100 pre:0
2	$\{v_0, v_1, v_3\}$	Length:0 pre:0	length: 10 pre:0	length:50 pre:3	length:30 pre:0	length:90 pre:3
3	$\{v_0, v_1, v_3, v_2\}$	length:0 pre:0	length: 10 pre:0	length:50 pre:3	length:30 pre:0	length:60 pre:2
4	$\{v_0, v_1, v_3, v_2, v_4\}$	length:0 pre:0	length: 10 pre:0	length:50 pre:3	length:30 pre:0	length:60 pre:2

7.5.1 单源最短路径

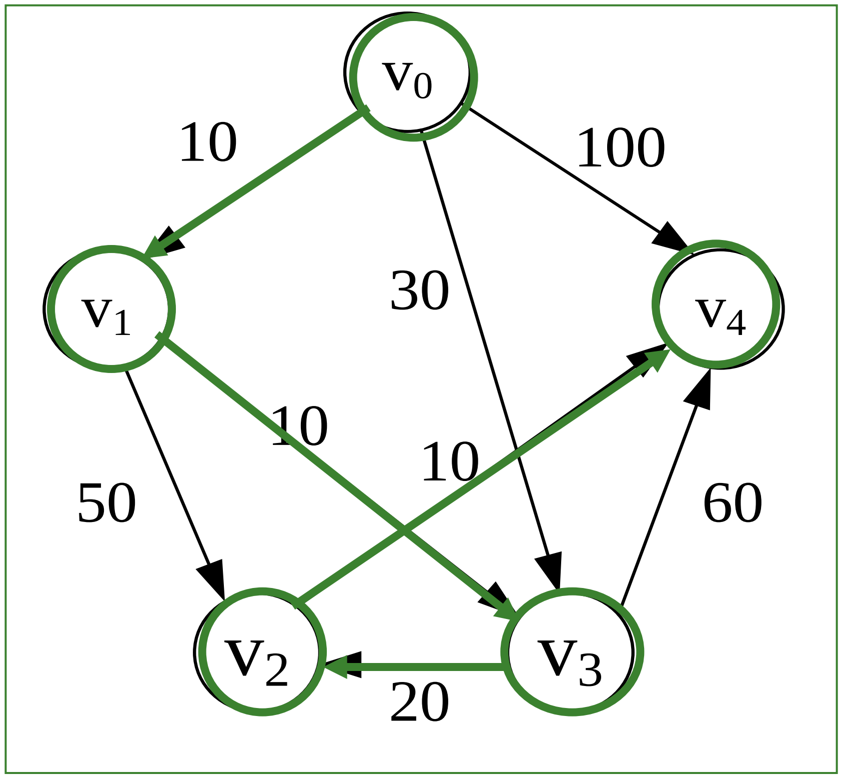


图7.20

图7.19用Dijkstra算法的处理过程，源顶点为v_0

7.5.1 单源最短路径

【代码7.8】 **Dijkstra算法**

```
class Dist {          // Dist类，Dijkstra和Floyd算法用于保存最短路径信息
public:
    int index;        // 顶点的索引值，仅Dijkstra算法用到
    int length;       // 当前最短路径长度
    int pre;          // 路径最后经过的顶点
};
void Dijkstra(Graph& G, int s, Dist* &D) { // s是源点
    D = new Dist[G. VerticesNum()];   // 数组D记录当前找到的最短特殊路径长度
    for (int i = 0; i < G.VerticesNum(); i++) {// 初始化Mark数组、D数组
        G.Mark[i] = UNVISITED;
    D[i].index = i;
```

7.5.1 单源最短路径

```
        D[i].length = INFINITE;
        D[i].pre = s;
}
D[s].length = 0;                          // 源点到自身的路径长度置为0
MinHeap<Dist> H(G. EdgesNum());   // 最小值堆用于找出最短路径
H.Insert(D[s]);
for (i = 0; i < G.VerticesNum(); i++) {
    bool FOUND = false;
    Dist d;
    while (!H.isEmpty())  {
        d = H.RemoveMin();                  // 获得到s路径长度最小的顶点
        if (G.Mark[d.index] == UNVISITED) {   // 如果未访问过则跳出循环
                FOUND = true;
                break;
        }
}
```

7.5.1 单源最短路径

```
if (!FOUND)                      // 若没有符合条件的最短路径则跳出本次循环
        break;
int v = d.index;
G.Mark[v] = VISITED;      // 将该顶点的标记位设置为VISITED
// 加入v以后需要刷新D中v的邻接点的最短路径长度
for (Edge e = G.FirstEdge(v); G.IsEdge(e);e = G.NextEdge(e))
    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;
          H.Insert(D[G.ToVertex(e)]);
    }
}
}
```

7.5.1 单源最短路径

- 对于n个顶点e条边的图，图中的任何一条边都可能在最短路径中出现，因此最短路径算法对每条边至少都要检查一次
- 代码7.8采用最小堆来选择权值最小的边，那么每次改变最短特殊路径长度时需要对堆进行一次重排，此时的时间复杂度为O((n + e)loge)，适合于稀疏图
- 如果像算法7.10介绍的Prim算法那样，通过直接比较D数组元素，确定代价最小的边就需要总时间$O(n^2)$；取出最短特殊路径长度最小的顶点后，修改最短特殊路径长度共需要时间O(e)，因此共需要花费$O(n^2)$的时间，这种方法适合于稠密图

7.5.1 单源最短路径

- 这种算法为什么得到的就是最短路径呢?
- 事实上，如果存在一条从源点到u且长度比D[u]更短的路径，假设这条路径经过第二个集合的某些顶点（不妨设为x）才到达u。在这条路径上分别用d(s，x)，d(x，u)和d(s，u)表示源点s到顶点x、顶点x到顶点u和源点s到顶点u的距离，那么有d(s，u) = d(s，x) + d(x，u) < D[u]，且D[x] ≤ d(s，x)。由边的权值的非负性，推得D[x] < D[u]。这与是第二个集合中距离值最小的顶点矛盾，所以每次加入第一个集合的u的距离值就是从s到u的最短路径长度。

7.5.2 每对顶点之间的最短路径

- 找出从任意顶点vi到顶点v j的最短路径，这个问题通常称为带权图的所有顶点对之间的最短路径(**all-pairs shortest paths**)问题。
- 解决这一问题可以每次以一个顶点为源点，重复执行**Dijkstra**算法**n**次，这样就可以求得所有的顶点对之间的最短路径及其最短路径长度，其时间复杂度为**O(n3)**。
- **Floyd**算法是典型的动态规划法，自底向上分别求解子问题的解，然后由这些子问题的解得到原问题的解。这个算法的时间复杂度也是**O(n3)**，但形式上比较简单。
- **Robert W.Floyd:**
  - **1936**年**6**月**8**日生于纽约，“自学成才”
  - 斯坦福大学计算机科学系教授
  - **1978**年图灵奖获得者

7.5.2 每对顶点之间的最短路径

- **Floyd算法用相邻矩阵adj来表示带权有向图，该算法的基本思想是：**
  - 初始化$adj^{(0)}$为相邻矩阵adj
  - 在矩阵$adj^{(0)}$上做n次迭代，产生一个矩阵序列$adj^{(1)}$，…，$adj^{(k)}$，…，$adj^{(n)}$
  - 其中经过第k次迭代，$adj^{(k)}[i，j]$的值等于从顶点v_i到顶点v_j路径上所经过的顶点序号不大于k的最短路径长度

7.5.2 每对顶点之间的最短路径

- 由于进行第k次迭代时已求得矩阵$adj^{(k-1)}$，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，那么此时就有$adj^{(k)}[i，j] = adj^{(k-1)}[i，j]$
  - 另一种是中间经过顶点v_k，此时$adj^{(k)}[i，j] < adj^{(k-1)}[i，j]$，这条由顶点$v_i$经过$v_k$到顶点$v_j$的中间顶点序号不大于k的最短路径由两段组成：
    - 一段是从顶点v_i到顶点v_k的中间顶点序号不大于k-1的最短路径
    - 另一段是从顶点v_k到顶点v_j的中间顶点序号不大于k-1的最短路径
    - 路径长度应为这两段长度之和，用下面的公式计算：$adj^{(k)}[i，j] = adj^{(k-1)}[i，k] + adj^{(k-1)}[k，j]$

7.5.2 每对顶点之间的最短路径

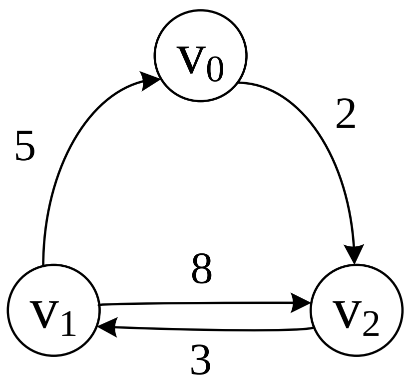


图7.21 每对顶点间的最短路径的示例

$$\mathrm{adj}^{(0)}=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 8\\ \infty & 3 & 0\end{bmatrix}\qquad \mathrm{path}=\begin{bmatrix}0 & -1 & 0\\ 1 & 1 & 1\\ -1 & 2 & 2\end{bmatrix}$$

$$\mathrm{adj}^{(1)}=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 7\\ \infty & 3 & 0\end{bmatrix}\qquad \mathrm{path}=\begin{bmatrix}0 & -1 & 0\\ 1 & 1 & 0\\ -1 & 2 & 2\end{bmatrix}$$

$$\mathrm{adj}^{(2)}=\begin{bmatrix}0 & \infty & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{bmatrix}\qquad \mathrm{path}=\begin{bmatrix}0 & -1 & 0\\ 1 & 1 & 0\\ 1 & 2 & 2\end{bmatrix}$$

$$\mathrm{adj}^{(3)}=\begin{bmatrix}0 & 5 & 2\\ 5 & 0 & 7\\ 8 & 3 & 0\end{bmatrix}\qquad \mathrm{path}=\begin{bmatrix}0 & 2 & 0\\ 1 & 1 & 0\\ 1 & 2 & 2\end{bmatrix}$$

图7.22 每对顶点间的最短路径的**Floyd**算法迭代过程

7.5.2 每对顶点之间的最短路径

【算法7.9】 Floyd算法

```
void Floyd(Graph& G, Dist** &D)  {
    int i,j,v;
    D = new Dist*[G.VerticesNum()];     // 为数组D申请空间
    for (i = 0; i < G.VerticesNum();i++)
            D[i] = new Dist[G.VerticesNum()];
    for (i = 0;i < G.VerticesNum();i++)   // 初始化数组D
            for (j = 0;j < G.VerticesNum();j++) {
                    if (i == j) {
                            D[i][j].length = 0;
                            D[i][j].pre = i;
                    }
```

7.5.2 每对顶点之间的最短路径

```
            else {
                  D[i][j].length = INFINITE;
                  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;
      }
```

7.5.2 每对顶点之间的最短路径

```
// 顶点i到顶点j的路径经过顶点v如果变短，则更新路径长度
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;
                        }
}
```

7.6 最小生成树

- 最小生成树的概念
  - 连通图的
  - 最小生成树生成树
  - 应用举例
- 最小生成树的构造算法
  - Prim算法
  - Kruskal算法

最小生成树的概念（1）

- 连通图的生成树
  - 复习：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（生成树）
  - 一般说，一个连通无向图的生成树不一定是唯一的（除非这个图本身就是树）
  - 对于**n**个结点的图，其生成树中必定有**n-1**条边

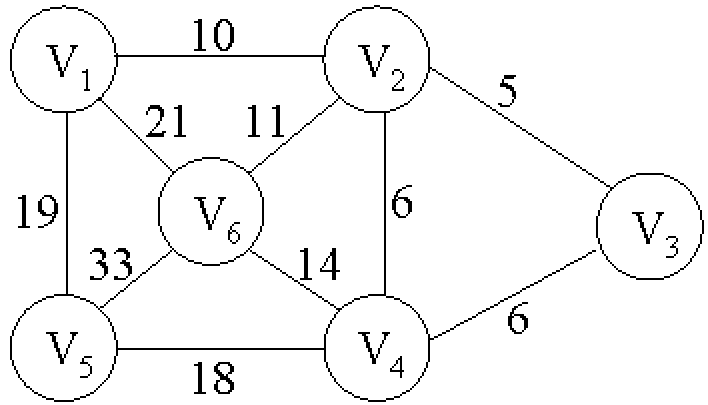
最小生成树的概念（2）

- 最小生成树
  - 在一个带权连通无向图中可能找到许多生成树
  - 我们可以提出进一步的要求，希望确定一棵总权值最小的生成树，这就是最小生成树的问题
  - 由于生成树只有有限个，其中必然有权值之和最小的。

最小生成树的概念（3）

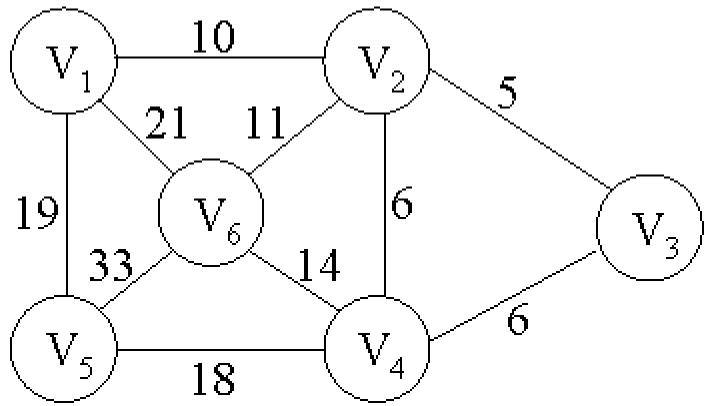
- 图G的生成树是一棵包含G的所有顶点的树，树上所有权值总和表示代价，那么在G的所有的生成树中，代价最小的生成树称为图G的最小生成树(minimum-cost spanning tree，简称MST)。

最小生成树的概念（4）

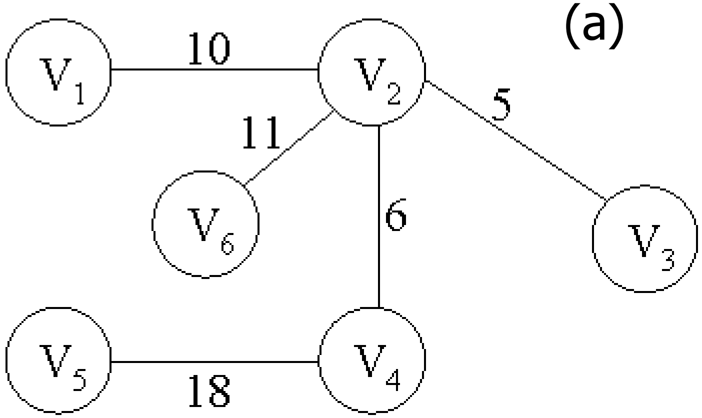


- 应用举例
  - 上图代表6个城市间的交通网，边上的权表示公路的造价
  - 现在要用公路把6个城市连接起来（这至少要修5条公路）
  - 如何设计使得这5条公路的总造价最少呢？

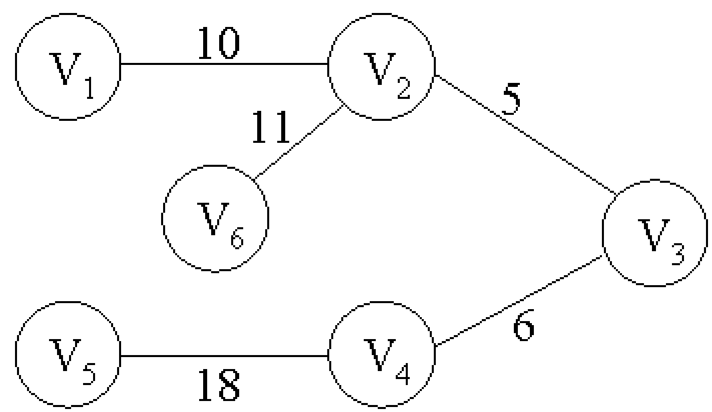
最小生成树的概念（5）



(a)



(a)的MST

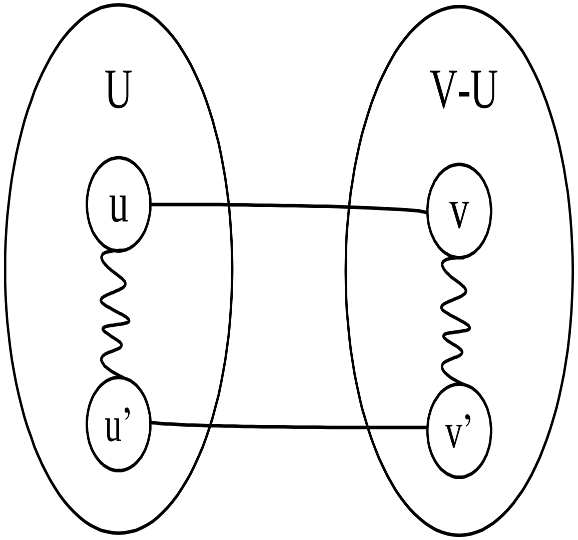


(a)的MST

最小生成树的概念（6）

- 设G = <V，E>是一个连通的带权图，其中每条边（v_i，v_j）上带有权W(v_i，v_j)
- 集合U是顶点集V的一个非空真子集。构建生成树时需要一条边连通顶点集合U和V-U
- 如果（u，v）∈E，其中u∈U，v∈V-U，且边（u，v）是符合条件的权值W (u，v)最小的，那么一定存在一棵包含边(u，v)的最小生成树
- 这条性质也称为MST性质

最小生成树的概念（7）



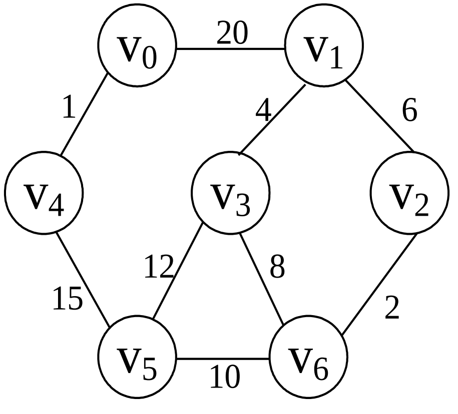
- **MST性质可以用反证法证明如下：**
  - 假设图G的任何一棵最小生成树都不包含边（u，v），显然当把边（u，v）加入到G的一棵最小生成树T中时，由生成树的定义，将产生一个含有边（u，v）的回路
  - 由于T是生成树，T中必存在不同于边（u，v）的另一条边（u'，v'），使得u'∈U，v'∈V-U，且u和u'之间，v和v'之间均有路径相连通
  - 将边（u'，v'）删除，就消除了上述回路，并得到了另一棵生成树T'。由于W(u，v)≤W(u'，v')，所以生成树T'的耗费不大于树T的耗费。于是T'是一棵包含边(u，v)的最小生成树，与假设矛盾

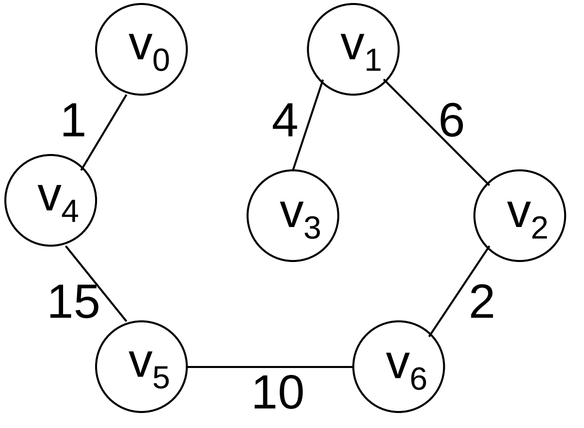
7.6.1 Prim算法

- 设G = <V，E>是一个联连通的带权图，其中V是顶点的集合，E是边的集合，TE为最小生成树的边的集合。则Prim算法通过以下步骤得到最小生成树：
  - 初始状态：U = {u0}，TE = {}。其中u0是顶点集合V中的某一个顶点
  - 在所有u∈U，v∈V-U的边(u,v)∈E中找一条权值最小的边(u0，v0)，将这条边加进集合TE中，同时将此边的另一顶点v0并入U。这一步骤的作用是在边集E里找一条两个顶点分别在集合U和V-U中且权值最小的边，把这条边添加到边集TE中，并把这条边上不在U中的那个顶点加入到U中
  - 如果U = V，则算法结束；否则重复步骤2
  - 算法结束时，TE中包含了G中的n-1条边。经过上述步骤选取到的所有边恰好就构成了图G的一棵最小生成树

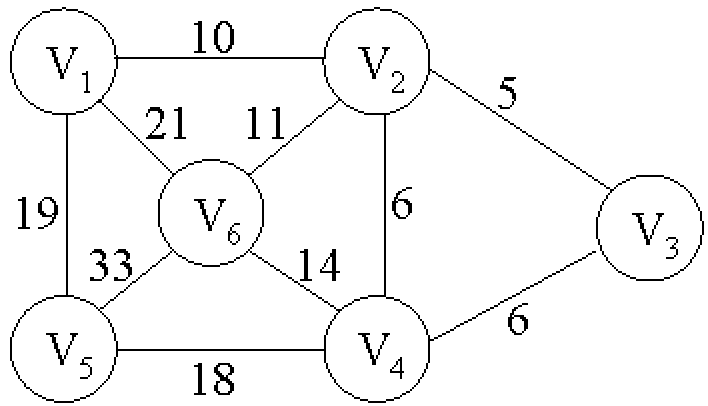
7.6.1 Prim算法

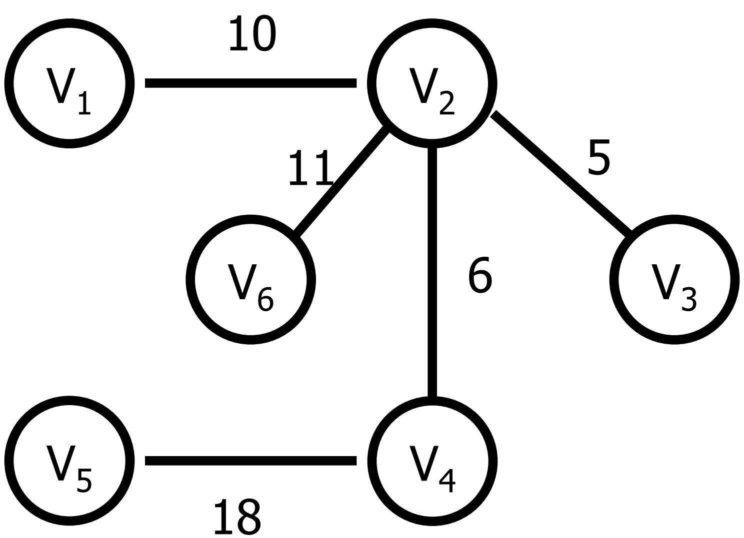
Prim算法构造图7.24中带权图的最小生成树的步骤

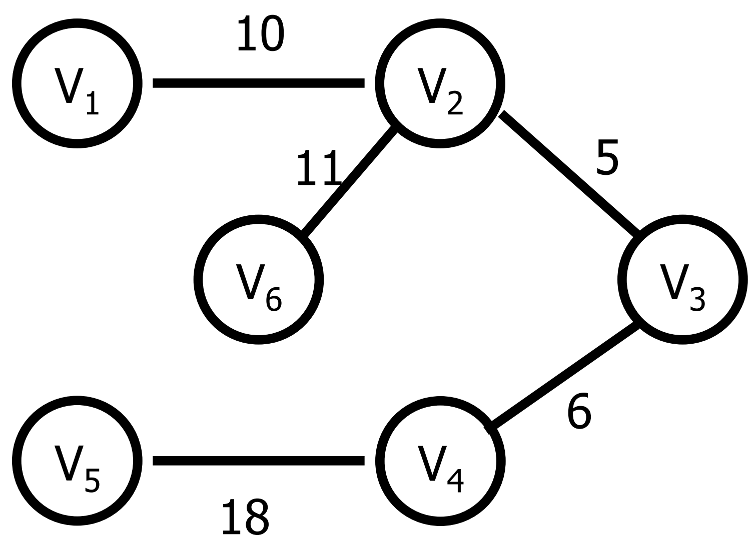




7.6.1 Prim算法







"十一五"国家级规划教材。张铭，王腾蛟，赵海燕，《数据结构与算法》，高教社，2008.6。

7.6.1 Prim算法

【算法7.10】**Prim算法**

```
void Prim(Graph& G, int s, Edge* &MST)  {
// s是开始顶点，数组MST用于保存最小生成树的边
    int MSTtag = 0;                              // 最小生成树的边计数
    MST = new Edge[G.VerticesNum()-1];           // 为数组MST申请空间
    Dist *D;
    D = new Dist[G. VerticesNum()];              // 为数组D申请空间
    for (int i = 0; i < G.VerticesNum(); i++) {  // 初始化Mark数组、D数组
        G.Mark[i] = UNVISITED;
        D[i].index = i;
        D[i].length = INFINITE;
        D[i].pre = s;
    }
```

7.6.1 Prim算法

```
D[s].length = 0;
G.Mark[s]= VISITED;                    // 开始顶点标记为VISITED
int v = s;
for (i = 0; i < G.VerticesNum()-1; i++) {
    if (D[v] == INFINITY) return;    // 非连通，有不可达顶点
    // 因为v的加入，需要刷新与v相邻接的顶点的D值
    for (Edge e = G.FirstEdge(v); G.IsEdge(e);e = G.NextEdge(e))
            if (G.Mark[G.ToVertex(e)] != VISITED &&
                  (D[G.ToVertex(e)].length > e.weight)) {
            D[G.ToVertex(e)].length = e.weight;
            D[G.ToVertex(e)].pre = v;
            }
    v = minVertex(G, D);          // 在D数组中找最小值记为v
    G.Mark[v] = VISITED; // 标记访问过
```

7.6.1 Prim算法

```
            Edge edge(D[v].pre, D[v].index, D[v].length); // 保存边
            AddEdgetoMST(edge,MST,MSTtag++);      // 将边edge加到MST中
    }
}
int minVertex(Graph& G, Dist* & D)  {       // 在Dist数组中找最小值
    int i, v;
    for (i = 0; i < G.VerticesNum(); i++)
            if (G.Mark[i] == UNVISITED) {
                    v = i;                    // 让v为随意一个未访问的顶点
                    break;
            }
    for (i = 0; i < G.VerticesNum(); i++)
            if ((G.Mark[i] == UNVISITED) && (D[i] < D[v]))
                    v = i;                    // 保存当前发现的具有最小距离的顶点
    return v;
}
```

7.6.1 Prim算法

- Prim算法非常类似于Dijkstra算法，但Prim算法中的距离值不需要累积，直接采用离集合最近的边距。Prim算法的时间复杂度也与Dijkstra算法相同。
- 本算法通过直接比较D数组元素，确定代价最小的边就需要总时间O(n2)；取出权最小的顶点后，修改D数组共需要时间O(e)，因此共需要花费O(n2)的时间，算法7.10适合于稠密图。

7.6.2 Kruskal算法

- **Kruskal算法：**
  - 使用的贪心准则，从剩下的边中选择具有最小权值且不会产生环路的边加入到生成树的边集中。

7.6.2 Kruskal算法

- **Kruskal算法的构造思想是：**
  - 首先将G中的n个顶点看成是独立的n个连通分量，这时的状态是有n个顶点而无边的森林，可以记为T = <V，{}>。
  - 然后在E中选择代价最小的边，如果该边依附于两个不同的连通分支，那么将这条边加入到T中，否则舍去这条边而选择下一条代价最小的边。
  - 依此类推，直到T中所有顶点都在同一个连通分量中为止，此时就得到图G的一棵最小生成树。

7.6.2 Kruskal算法

Kruskal算法构造图7.24中带权图的最小生成树的步骤

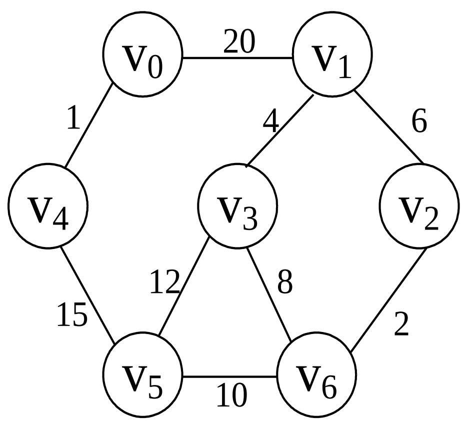
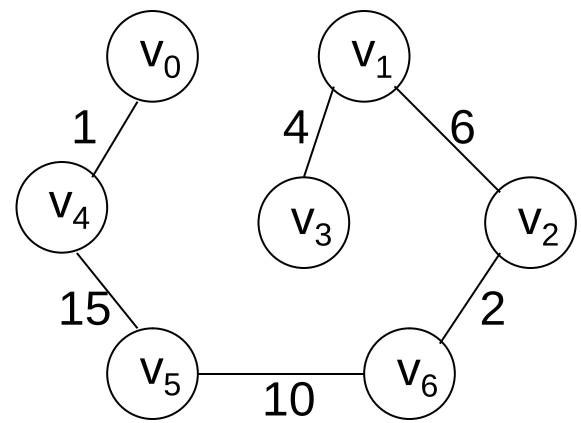
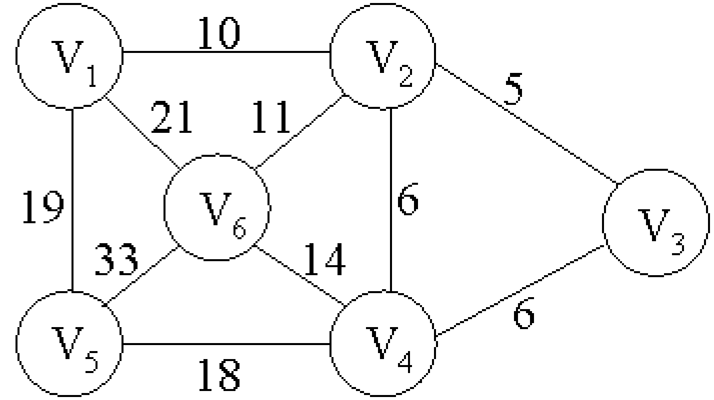


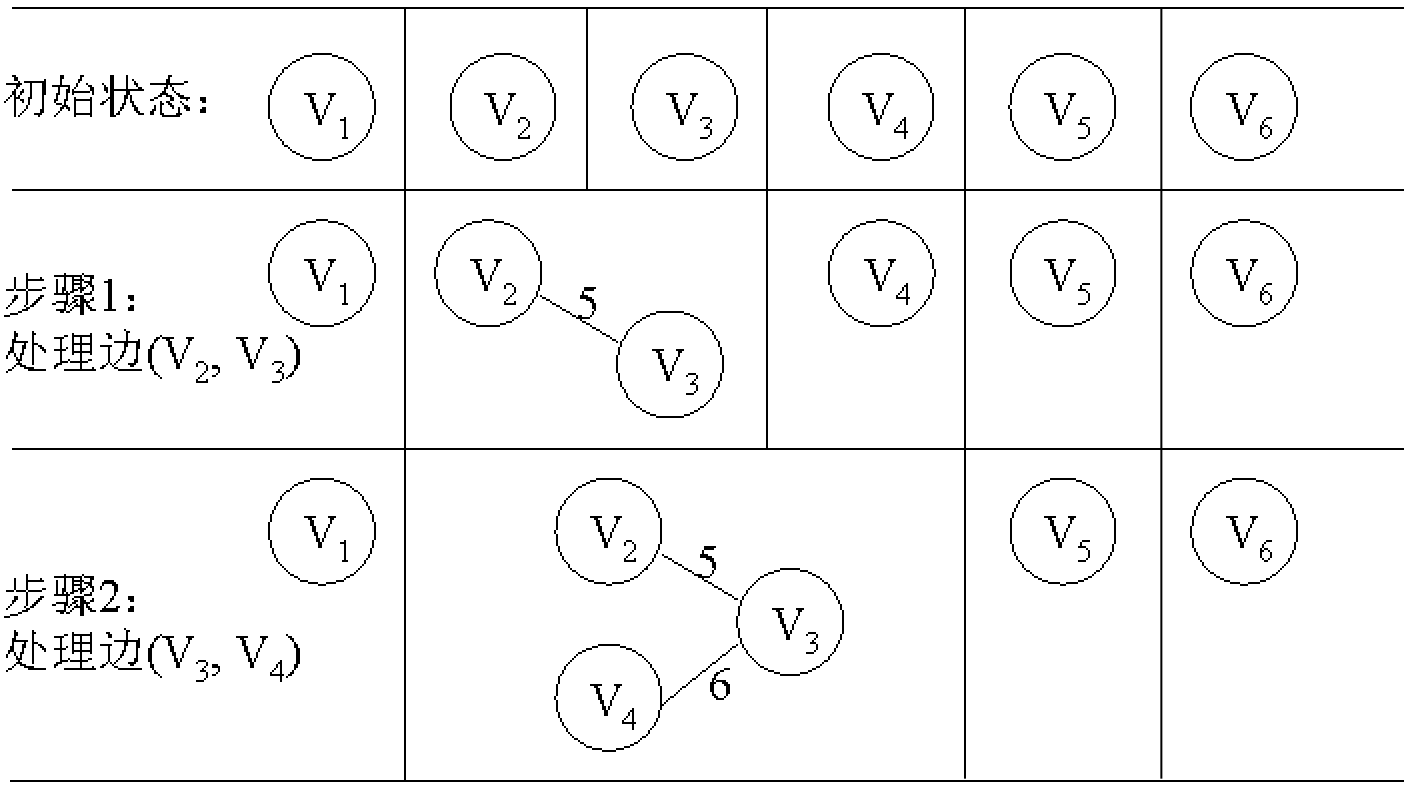
图7.24 带权图



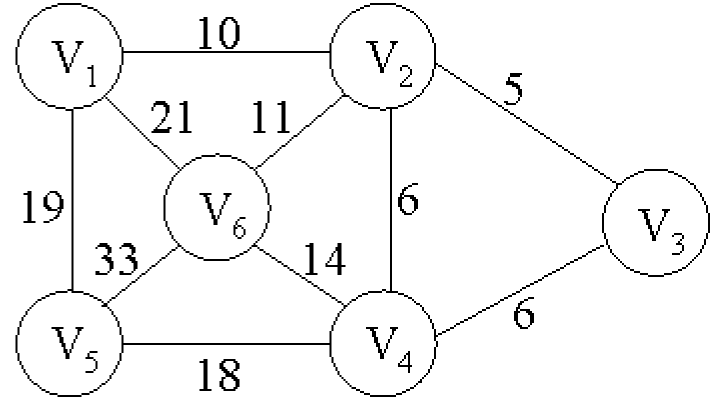
“十一五”国家级规划教材。张铭，王腾蛟，赵海燕，《数据结构与算法》，高教社，2008. 6。

7.6.2 Kruskal算法

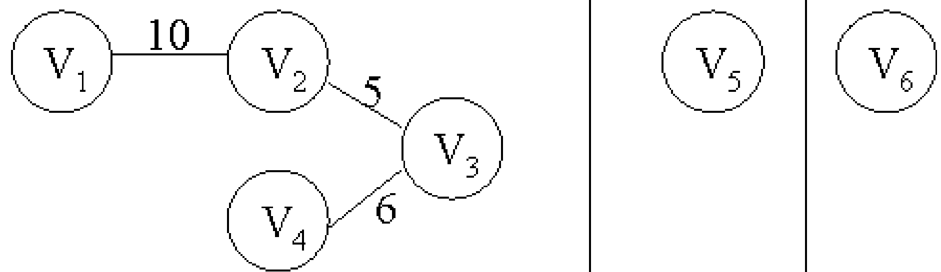




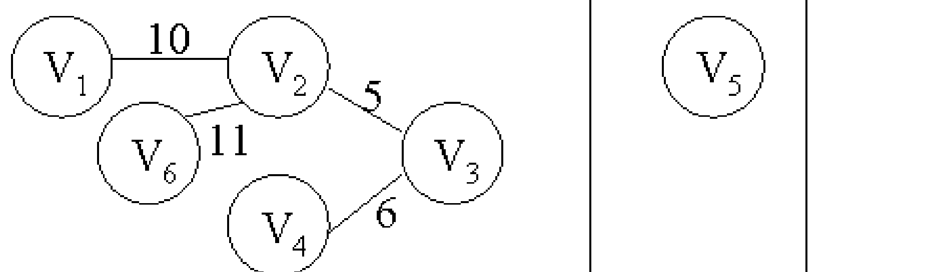
7.6.2 Kruskal算法



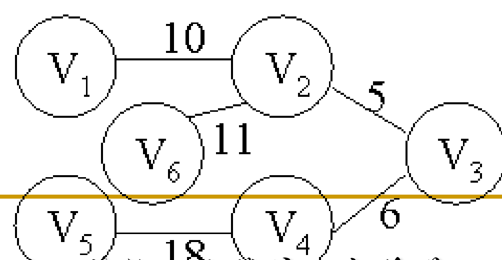
步骤3：
处理边(V_1, V_2)



步骤4：
处理边(V_2, V_6)



步骤5：
处理边(V_4, V_5)



“十一五”国家级规划教材。张铭，王腾蛟，赵海燕，《数据结构与算法》，高教社，2008. 6。

7.6.2 Kruskal算法

【算法7.11】 **Kruskal算法**

```
void Kruskal(Graph& G, Edge* &MST) {
// 数组MST用于保存最小生成树的边
    ParTree<int> A(G.VerticesNum());                    // 等价类
    MinHeap<Edge> H(G.EdgesNum());                      // 最小堆
    MST = new Edge[G.VerticesNum()-1];                  // 为数组MST申请空间
    int MSTtag = 0;                                     // 最小生成树的边计数
    bool heapEmpty;
    for (int i = 0; i < G.VerticesNum(); i++) // 将图的所有边插入最小堆H中
        for (Edge e = G. FirstEdge(i); G.IsEdge(e);e = G. NextEdge(e))
            if (G.FromVertex(e) < G.ToVertex(e))
                // 对于无向图，防止重复插入边
                H.Insert(e);
```

7.6.2 Kruskal算法

```
int EquNum = G.VerticesNum();          // 开始n个顶点分别作为一个等价类
while (EquNum > 1) {                   // 当等价类的个数大于1时合并等价类
        heapEmpty = H.isEmpty();
        if (!heapEmpty)
                Edge e = H.RemoveMin();    // 获得一条权最小的边
        if (heapEmpty || e.weight == INFINITY) {
                cout << "不存在最小生成树." <<endl;
        delete [] MST;                           // 释放空间
        MST = NULL;                              // MST赋为空
                return;
        }
        int from = G.FromVertex(e);              // 记录该条边的信息
        int to = G.ToVertex(e);
```

7.6.2 Kruskal算法

```
    if (A.Different(from,to)) {          // 边e的两个顶点不在一个等价类
        A.Union(from,to); // 将边e的两个顶点所在的等价类合并为一个
        AddEdgetoMST(e,MST,MSTtag++); // 将边e加到MST
        EquNum--;                                  // 等价类的个数减1
        }
    }
}
```

【算法7.11结束】

7.6.2 Kruskal算法

- Kruskal算法的时间复杂度为O(eloge)。这个算法的时间复杂度主要取决于边数，因此Kruskal算法适合于构造稀疏图的最小生成树。

7.7 图知识点总结

- 7.1 图的基本概念
- 7.2 图的抽象数据类型
- 7.3 图的存储结构
- 7.4 图的周游（深度、广度、拓扑）
- 7.5 最短路径问题
- 7.6 最小生成树

The End