

# 第12章 图的基本概念

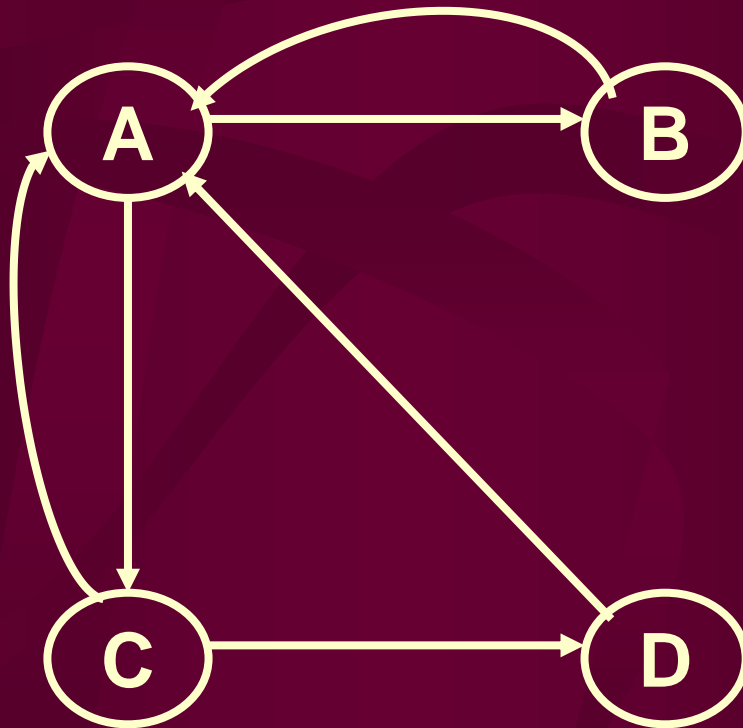
- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 

# 图的定义

- 图可以用 $G=(V, E)$ 表示。其中， $V$ 是顶点的集合， $E$ 是连接顶点的边（弧）的集合。
- 如果边是有方向的，称为有向图。有向图的边用 $\langle \rangle$ 表示。  
 $\langle A, B \rangle$ 表示从 $A$ 出发到 $B$ 的一条边。在有向图中， $\langle A, B \rangle$ 和 $\langle B, A \rangle$ 是不一样的。
- 如果边是无方向的，称为无向图。无向图的边通常用圆括号表示。 $(A, B)$ 表示顶点 $A$ 和 $B$ 之间有一条边。无向图也称为双向图。
- 加权图：边被赋予一个权值的图称为加权图。如果图是有向的，称为加权有向图，如果是无向的，称为加权无向图。

• 如G1:  $V = \{A, B, C, D\}$ ,

$E = \{ \langle A, B \rangle, \langle B, A \rangle, \langle A, C \rangle, \langle C, A \rangle, \langle C, D \rangle, \langle D, A \rangle \}$  表示的图如下所示



$V = \{A, B, C, D, E\},$

$E = \{ (A, B),$

$(A, C),$

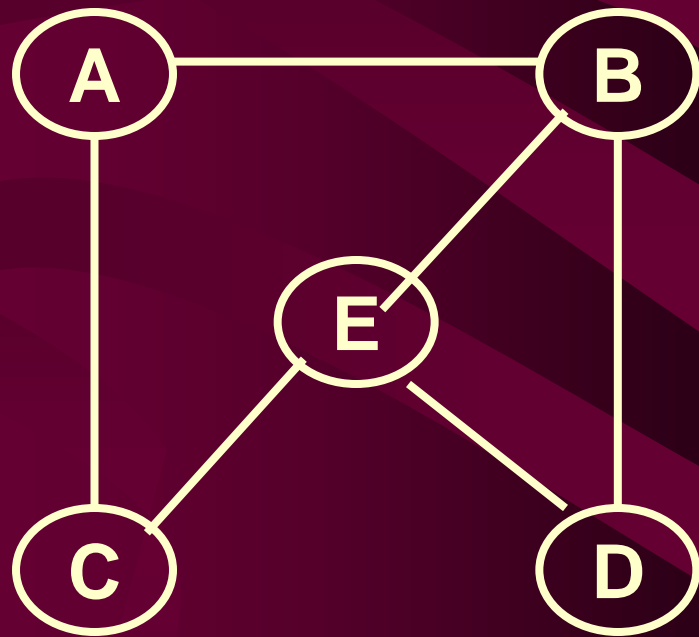
$(B, D),$

$(B, E),$

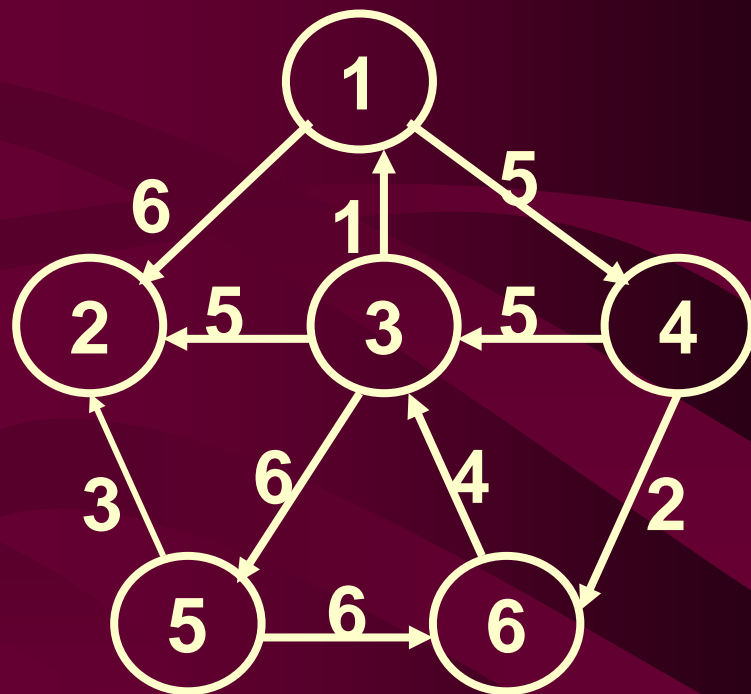
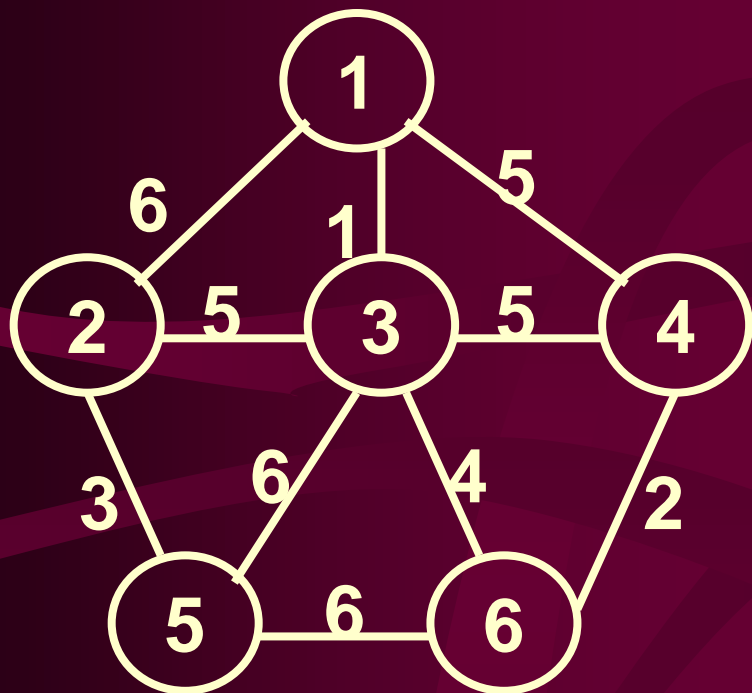
$(D, E),$

$(C, E) \}$

无向图



# 加权图



加权图中边的表示:  $\langle V_i, V_j, W \rangle$  或  $(V_i, V_j, W)$

# 第12章 图的基本概念

- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 

# 图的基本术语

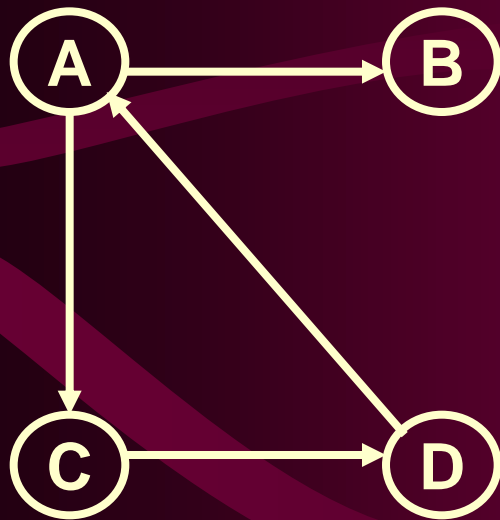
- 邻接：如  $(V_i, V_j)$  是图中的一条边，则称  $V_i$  和  $V_j$  是邻接的。如  $\langle V_i, V_j \rangle$  是图中的一条边，则称  $V_i$  邻接到  $V_j$ ，或  $V_j$  和  $V_i$  邻接。
- 度：无向图中邻接于某一结点的边的总数。
- 入度：有向图中进入某一结点的边数，称为该结点的入度
- 出度：有向图中离开某一结点的边数，称为该结点的出度

# 子图

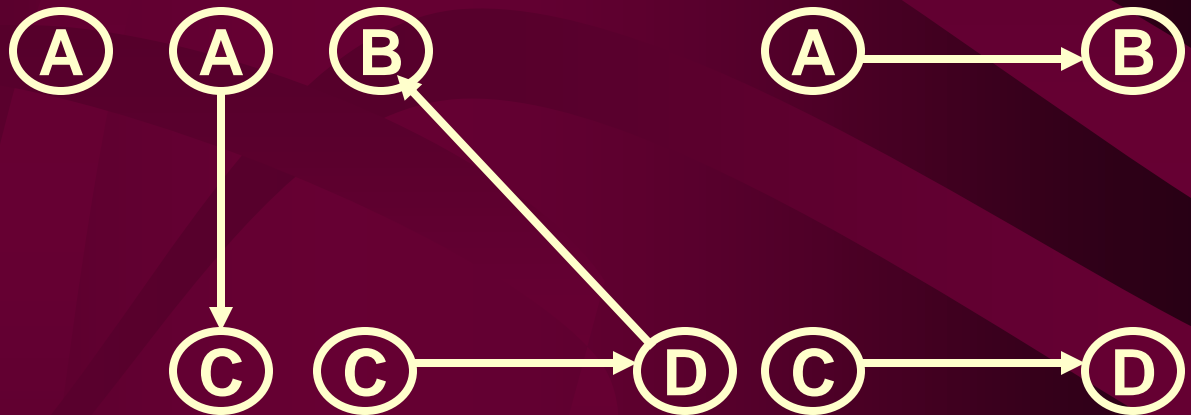
设有两个图 $G = (V, E)$  和  $G' = (V', E')$ ，如果

$V' \subseteq V, E' \subseteq E$  则称 $G'$ 是 $G$ 的子图

有向图 **G1**

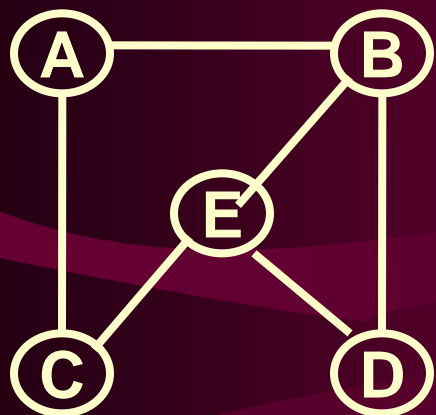


有向图**G1**的子图

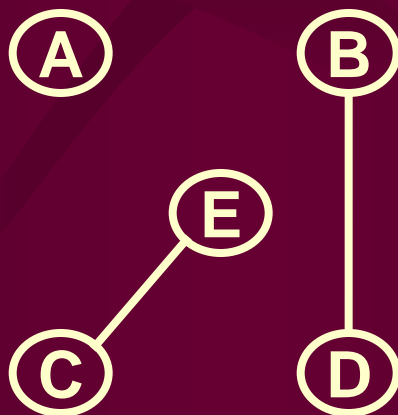
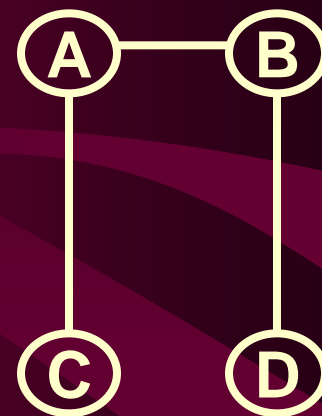
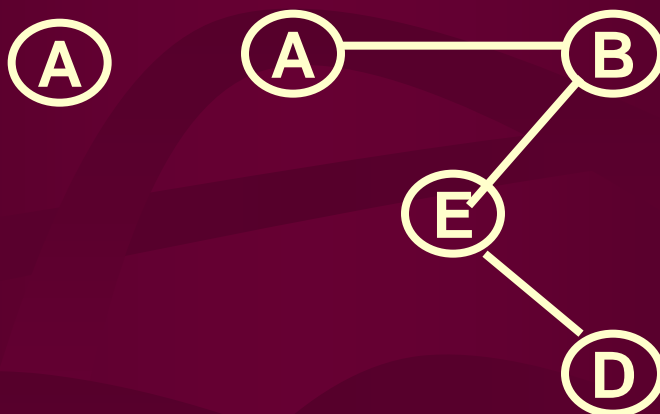




## 无向图 $G_2$



## 无向图 $G_2$ 的子图



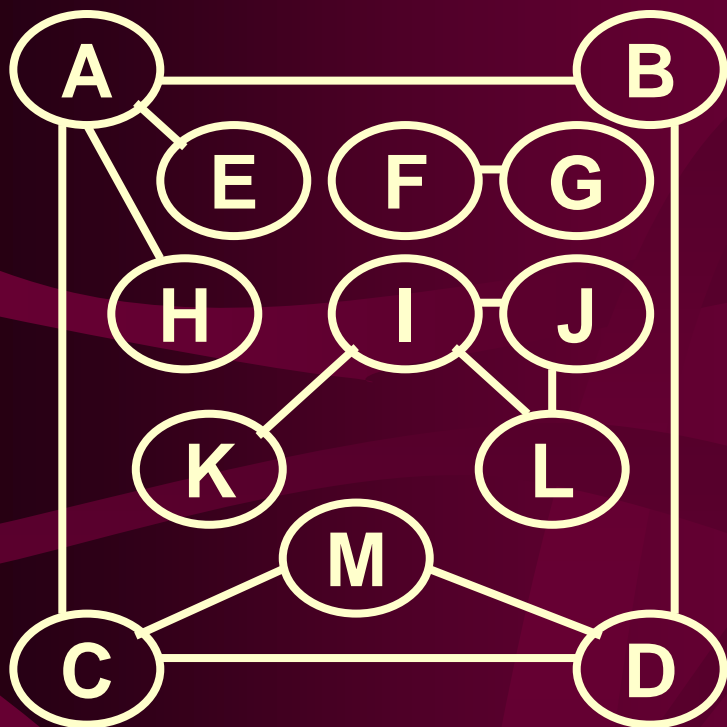
# 路径和路径长度

- 对 $1 < i < N$ ，结点序列 $w_1, w_2, \dots, w_N$  中的结点对  $(w_i, w_{i+1})$  都有  $(w_i, w_{i+1}) \in E$  或  $\langle w_i, w_{i+1} \rangle \in E$ ，那么， $w_1, w_2, \dots, w_N$  是图中的一条路径。
- 非加权的路径长度就是组成路径的边数，对于路径  $w_1, w_2, \dots, w_N$ ，非加权路径长度为 $N-1$ 。
- 加权路径长度是指路径上所有边的权值之和。
- 简单路径和环：如果一条路径上的所有结点，除了起始结点和终止结点可能相同外，其余的结点都不相同，则称其为简单路径。一个回路或环是一条简单路径，其起始结点和终止结点相同，且路径长度至少为1。

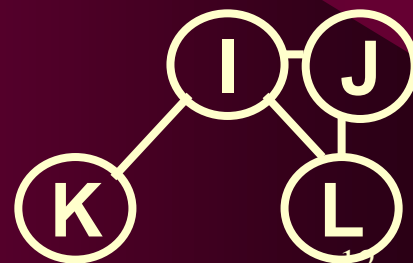
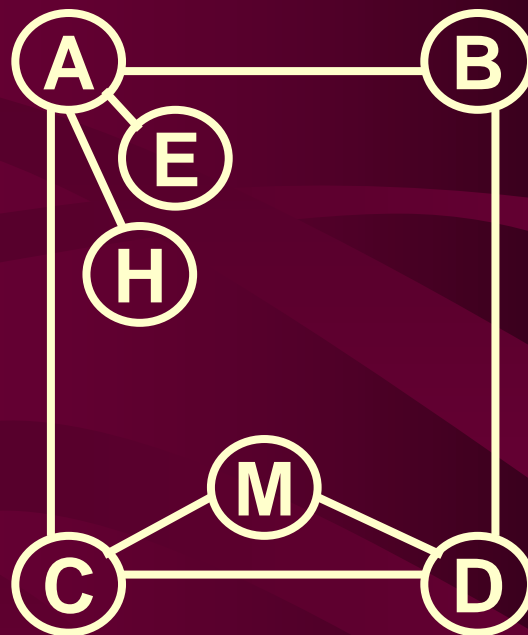
# 无向图的连通性

- 连通：顶点 $v$ 至 $v'$ 之间有路径存在
- 连通图：无向图  $G$  的任意两点之间都是连通的，则称  $G$  是连通图。
- 连通分量：非连通图中的极大连通子图

## 无向图G



## 无向图G的三个连通分量

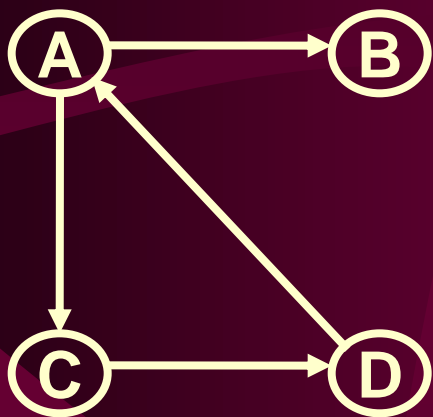


# 有向图的连通性

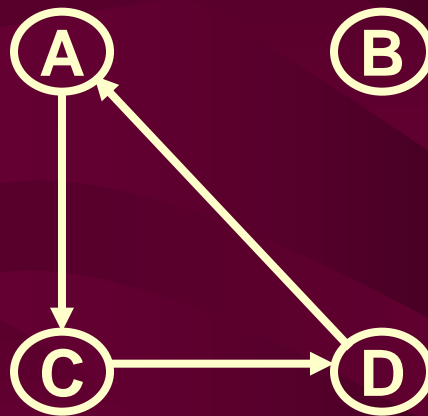
- 强连通图：有向图  $G$  的任意两点之间都是连通的，则称  $G$  是强连通图。
- 强连通分量：极大连通子图
- 弱连通图：如有向图  $G$  不是强连通的，但如果把它看成是无向图时是连通的，则称该图是弱连通的

## 有向图G

不是强连通图。因为B到其他结点都没有路径。但此图是弱连通的。

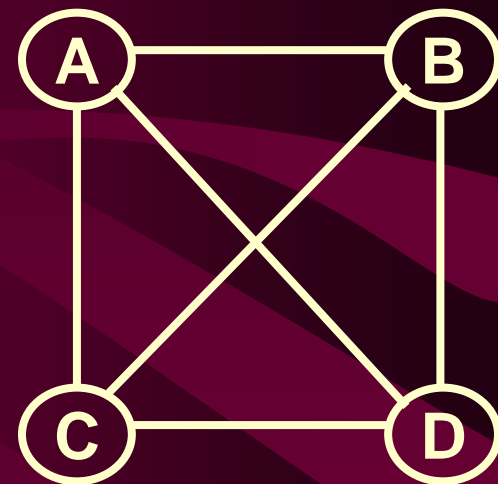


有向图G的两个强连通分量



# 完全图

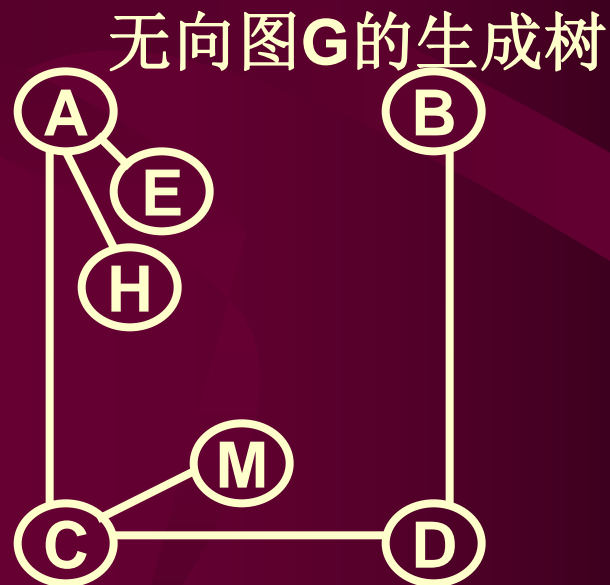
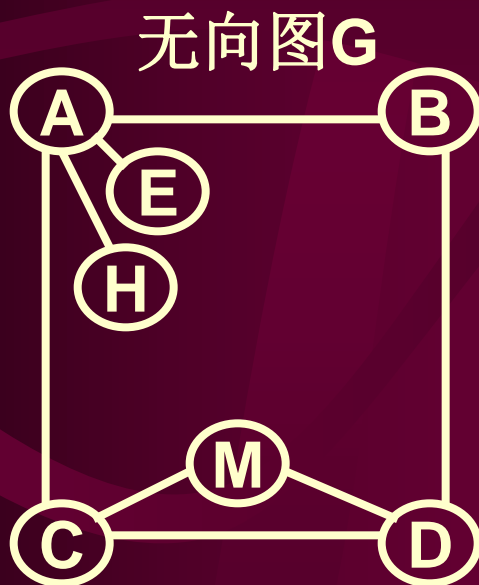
- 完全图：每两个节点之间都有边的无向图称为完全图。完全图有  $n(n-1)/2$  条边的无向图。其中  $n$  是结点数。  
即  $C_n^2$
- 有向完全图：每两个节点之间都有两条弧的有向图称为有向完全图。有向完全图有  $n(n-1)$  条边。其中  $n$  是结点数。即  $2C_n^2$
- 如果一个有向图中没有环，则称为有向无环图，简称为DAG



无向完全图

# 生成树

- 生成树是连通图的极小连通子图。包含图的所有  $n$  个结点，但只含图的  $n-1$  条边。在生成树中添加一条边之后，必定会形成回路或环。





# 第12章 图的基本概念

- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 

# 图的运算

- 常规操作：
  - 构造一个由若干个结点、若干条边组成的图；
  - 判断两个结点之间是否有边存在；
  - 在图中添加或删除一条边；
  - 返回图中的结点数或边数；
  - 按某种规则遍历图中的所有结点。
- 和应用紧密结合的运算：
  - 拓扑排序
  - 找最小生成树
  - 找最短路径等。

# 图的抽象类

```
template <class TypeOfEdge>
class graph {
    public:
        virtual bool insert(int u, int v, TypeOfEdge w) = 0;
        virtual bool remove(int u, int v) = 0;
        virtual bool exist(int u, int v) const = 0;
        virtual numOfVer() const {return Vers;}
        virtual numOfEdge() const {return Edges;}
    protected:
        int Vers, Edges;
};
```

# 第12章 图的基本概念

- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 

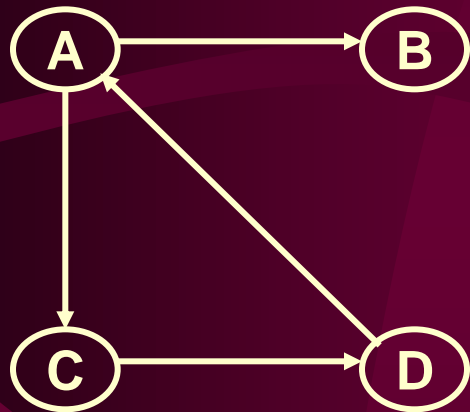
# 图的存储

- 邻接矩阵和加权邻接矩阵
- 邻接表



# 邻接矩阵——有向图

设有向图具有  $n$  个结点, 则用  $n$  行  $n$  列的布尔矩阵  $A$  表示该有向图如果  $i$  至  $j$  有一条有向边,  $A[i, j] = 1$ , 如果  $i$  至  $j$  没有一条有向边,  $A[i, j] = 0$



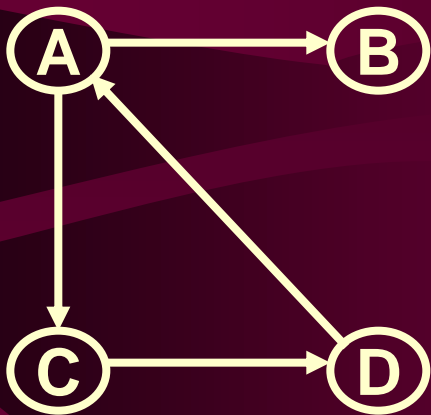
表示成右图矩阵

0	1	1	0
0	0	0	0
0	0	0	1
1	0	0	0

注意: 出度:  $i$ 行之和。入度:  $j$ 列之和。

# 邻接矩阵——有向图

- 在物理实现时的考虑：分别用 0、1、2、3 分别标识结点A、B、C、D。而将真正的数据字段之值放入一个一维数组之中。



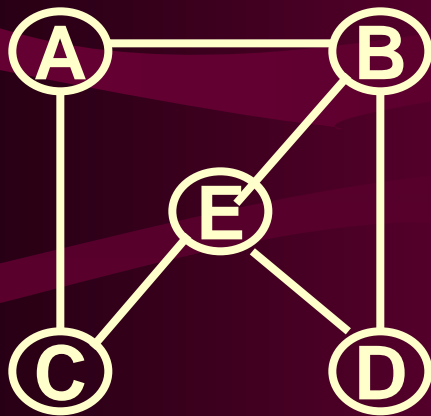
表示成右图矩阵

	0	1	2	3
0	0	1	1	0
1	0	0	0	0
2	0	0	0	1
3	1	0	0	0

0	1	2	3
A	B	C	D

# 邻接矩阵—无向图

设无向图具有  $n$  个结点，则用  $n$  行  $n$  列的布尔矩阵  $A$  表示该无向图；并且  $A[i, j] = 1$ ，如果  $i$  至  $j$  有一条无向边； $A[i, j] = 0$  如果  $i$  至  $j$  没有一条无向边



表示成右图矩阵

0	1	1	0	0
1	0	0	1	1
1	0	0	0	1
0	1	0	0	1
0	1	1	1	0

注意： 无向图的邻接矩阵是一个对称矩阵

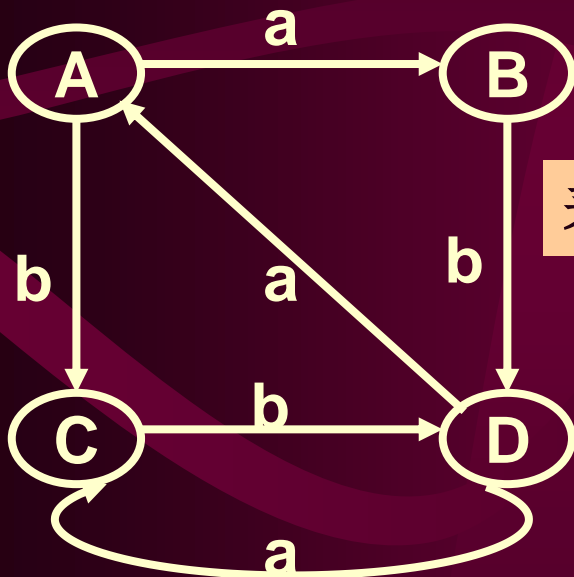
$i$  结点的度： $i$  行或  $i$  列之和。

在物理实现时的考虑，和前一页的无向图类似。



# 加权的邻接矩阵——有向图

设有向图具有  $n$  个结点，则用  $n$  行  $n$  列的矩阵  $A$  表示该有向图； 如果  $i$  至  $j$  有一条有向边且它的权值为  $a$ ，则  $A[i, j] = a$ 。如果  $i$  至  $j$  没有一条有向边。则  $A[i, j] = \text{空 或其它标志}$



表示成右图矩阵

0	a	b	$\infty$
$\infty$	0	$\infty$	b
$\infty$	$\infty$	0	b
a	$\infty$	a	0

# 邻接矩阵的特点

- 优点：判断任意两点之间是否有边方便，仅耗费  $O(1)$  时间。
- 缺点：即使  $\ll n^2$  条边，也需内存  $n^2$  单元，太多；仅读入数据耗费  $O(n^2)$  时间，太长。而大多数的图的边数远远小于  $n^2$ 。

# 邻接矩阵类的定义

```
template <class TypeOfVer, class TypeOfEdge>
class adjMatrixGraph:public graph<TypeOfEdge> {
    public: adjMatrixGraph(int vSize, const TypeOfVer d[],
        TypeOfEdge noEdgeFlag);
        bool insert(int u, int v, TypeOfEdge w);
        bool remove(int u, int v);
        bool exist(int u, int v) const;
        ~adjMatrixGraph() ;
    private:
        TypeOfEdge **edge; //存放邻接矩阵
        TypeOfVer *ver; //存放结点值
        TypeOfEdge noEdge; //邻接矩阵中的 $\infty$ 的表示值
};
```

# 构造函数

```
template <class TypeOfVer, class TypeOfEdge>
adjMatrixGraph<TypeOfVer, TypeOfEdge>::adjMatrixGraph
    (int vSize, const TypeOfVer d[], TypeOfEdge noEdgeFlag)
{ int i, j;
  Vers = vSize;
  Edges = 0;
  noEdge = noEdgeFlag;

  //存放结点的数组的初始化
  ver = new TypeOfVer[vSize];
  for (i=0; i<vSize;++ i) ver[i] = d[i];
```

//邻接矩阵的初始化

```
edge = new TypeOfEdge*[vSize];
```

```
for (i=0; i<vSize; ++ i) {
```

```
    edge[i] = new TypeOfEdge[vSize];
```

```
    for (j=0; j<vSize; ++j) edge[i][j] = noEdge;
```

```
    edge[i][i] = 0;
```

```
}
```

```
}
```

# 析构函数

```
template <class TypeOfVer, class TypeOfEdge>
adjMatrixGraph<TypeOfVer, TypeOfEdge>::
    ~adjMatrixGraph()
{
    delete [] ver;
    for (int i=0; i<Vers; ++i) delete [] edge[i]
    delete [] edge;
}
```

# Insert函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjMatrixGraph<TypeOfVer, TypeOfEdge>::
    insert(int u, int v, TypeOfEdge w)
{ if (u < 0 || u > Vers - 1 || v < 0 || v > Vers - 1)
    return false;
  if (edge[u][v] != noEdge) return false;
  edge[u][v] = w;
  ++Edges;
  return true;
}
```

# Remove函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjMatrixGraph<TypeOfVer, TypeOfEdge>::
    remove(int u, int v)
{ if (u < 0 || u > Vers - 1 || v < 0 || v > Vers - 1)
    return false;
  if (edge[u][v] == noEdge) return false;
  edge[u][v] = noEdge;
  --Edges;
  return true;
}
```



# Exist函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjMatrixGraph<TypeOfVer, TypeOfEdge>::
    exist(int u, int v) const
{ if (u < 0 || u > Vers - 1 || v < 0 || v > Vers - 1)
    return false;
  if (edge[u][v] == noEdge) return false;
  else return true;
}
```

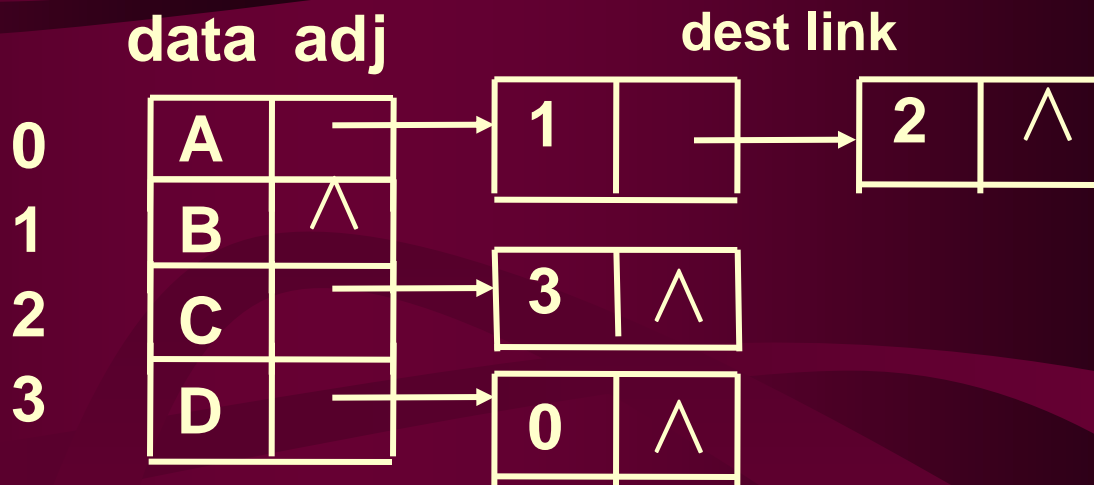
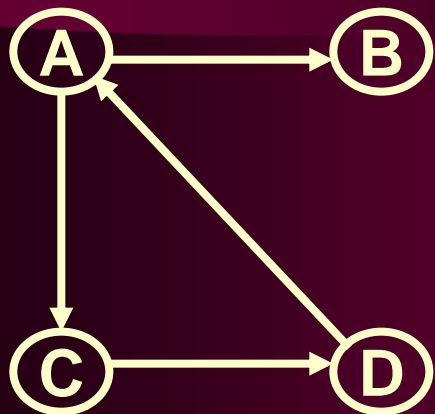
# 图的存储

- 邻接矩阵和加权邻接矩阵
- 邻接表

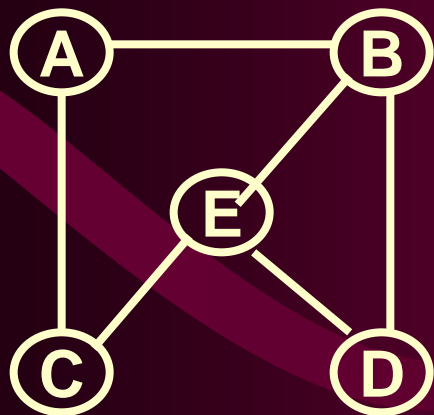
# 邻接表

- 设有向图或无向图具有  $n$  个结点，则用结点表、边表表示该有向图或无向图。
- 结点表：用数组或单链表的形式存放所有的结点值。如果结点数 $n$ 已知，则采用数组形式，否则应采用单链表的形式。
- 边表（边结点表）：每条边用一个结点进行表示。同一个结点出发的所有的边形成它的边结点单链表。

有向图 G1



无向图 G2



# 邻接表的特点

- 邻接表是图的标准存储方式
- 优点：内存 = 结点数 + 边数，处理时间也是结点数 + 边数，即为 $O(|V|+|E|)$ 。
- 当我们谈及图的线性算法时，一般指的是 $O(|V|+|E|)$
- 缺点：
  - 确定  $i \rightarrow j$  是否有边，最坏需耗费  $O(n)$  时间。
  - 无向图同一条边表示两次。边表空间浪费一倍。
  - 有向图中寻找进入某结点的边，非常困难。

# 邻接表类的定义

```
template <class TypeOfVer, class TypeOfEdge>
class adjListGraph:public graph<TypeOfEdge> {
public:
    adjListGraph(int vSize, const TypeOfVer d[]);
    bool insert(int u, int v, TypeOfEdge w);
    bool remove(int u, int v);
    bool exist(int u, int v) const;
    ~adjListGraph() ;
```

**private:**

**struct edgeNode** {**//邻接表中存储边的结点类**

**int end;** **//终点编号**

**TypeOfEdge weight;** **//边的权值**

**edgeNode \*next;**

**edgeNode(int e, TypeOfEdge w, edgeNode \*n = NULL)**

**{ end = e; weight = w; next = n;}**

**};**

**struct verNode**{ **//保存顶点的数据元素类型**

**TypeOfVer ver;** **//顶点值**

**edgeNode \*head;** **//对应的单链表的头指针**

**verNode( edgeNode \*h = NULL) { head = h ;}**

**};**

**verNode \*verList;**

**};**

# 构造函数

```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::
    adjListGraph(int vSize, const TypeOfVer d[])
{ Vers = vSize;
  Edges = 0;

  verList = new verNode[vSize];
  for (int i = 0; i < Vers; ++i) verList[i].ver = d[i];
}
```



# 析构函数

```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::~~adjListGraph()
{ int i;
  edgeNode *p;
  for (i = 0; i < Vers; ++i)
    while ((p = verList[i].head) != NULL) {
      verList[i].head = p->next;
      delete p;
    }
  delete [] verList;
}
```

# Insert函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::
    insert(int u, int v, TypeOfEdge w)
{ verList[u].head =
    new edgeNode(v, w, verList[u].head );
    ++Edges;
    return true;
}
```

# Remove函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::remove(int u, int v)
{edgeNode *p = verList[u].head, *q;
 if (p == NULL) return false; //结点u没有相连的边
 if (p->end == v) //单链表中的第一个结点就是被删除的边
 { verList[u].head = p->next;
  delete p; --Edges;
  return true; }
 while (p->next != NULL && p->next->end != v) p = p->next
 if (p->next == NULL) return false; //没有找到被删除的边
 q = p->next; p->next = q->next; delete q;
 --Edges;
 return true;
}
```

# Exist函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>::
    exist(int u, int v) const
{ edgeNode *p = verList[u].head;
  while (p !=NULL && p->end != v) p = p->next;
  if (p == NULL) return false; else return true;
}
```

# 第12章 图的基本概念

- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 

# 图的遍历

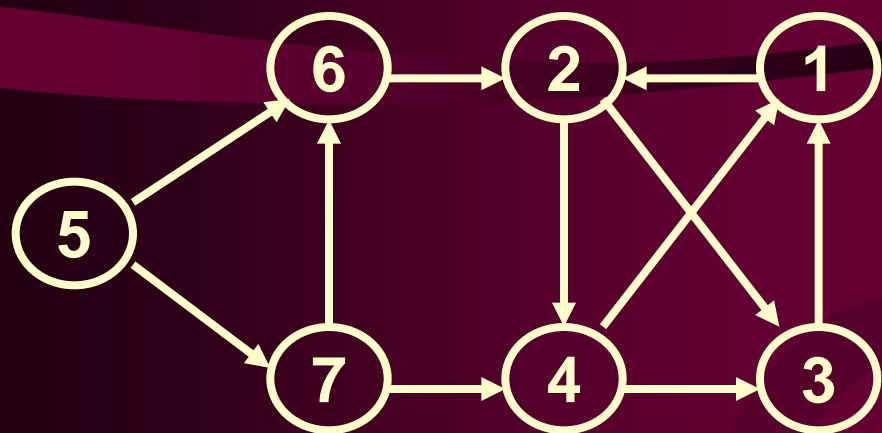
对有向图和无向图进行遍历是按照某种次序系统地访问图中的所有顶点，并且使得每个顶点只能被访问一次。在图中某个顶点可能和图中的多个顶点邻接并且存在回路，因此在图中访问一个顶点 $u$ 之后，在以后的访问过程中，又可能再次返回到顶点 $u$ ，所以图的遍历要比树的遍历更复杂

- 深度优先搜索
- 广度优先搜索



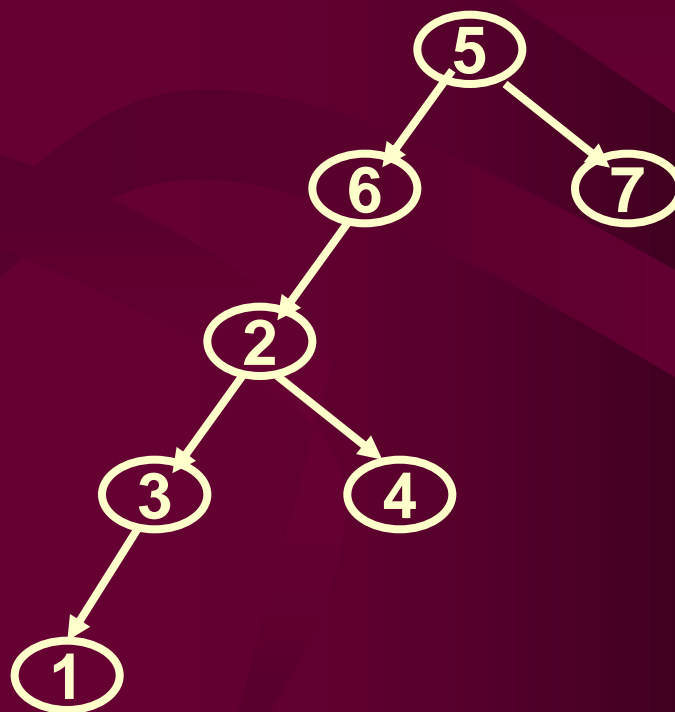
# 深度优先搜索

- 1、选中第一个被访问的顶点；
- 2、对顶点作已访问过的标志；
- 3、依次从顶点的未被访问过的第一个、第二个、第三个…… 邻接顶点出发，进行深度优先搜索；
- 4、如果还有顶点未被访问，则选中一个起始顶点，转向2；
- 5、所有的顶点都被访问到，则结束。



从结点5开始进行深度优先的搜索，则遍历序列可以为：5, 7, 6, 2, 4, 3, 1, 也可以为：5, 6, 2, 3, 1, 4, 7。

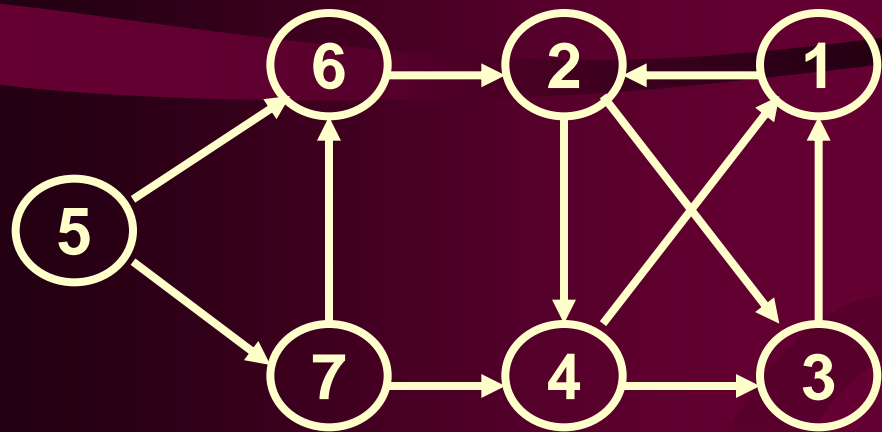
深度优先生成树



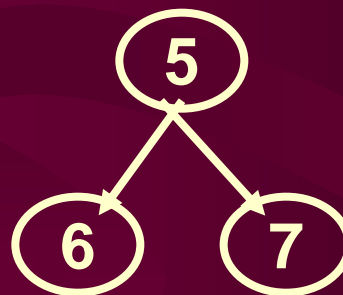
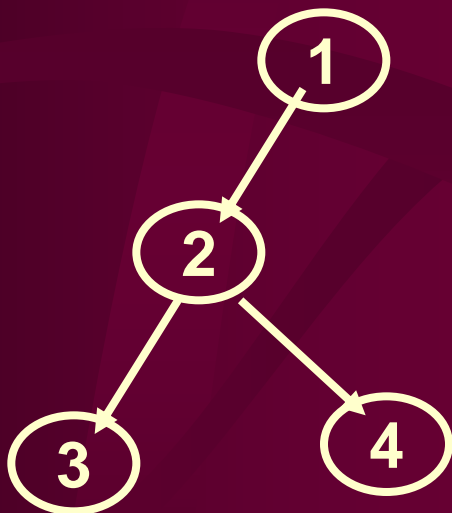


# 深度优先生成森林

- 在进行深度优先搜索 DFS 时，有时并不一定能够保证从某一个结点出发能访问到所有的顶点
- 在这种情况下，必须再选中一个未访问过的顶点，继续进行深度优先搜索。直至所有的顶点都被访问到为止。
- 这时，得到的是一组树而不是一棵树，这一组树被称为深度优先生成森林。



从结点1开始深度  
优先搜索



# 深度优先搜索的实现

- 深度优先搜索DFS的实现方法和树的前序遍历算法类似，但必须对访问过的顶点加以标记
- dfs函数不需要参数，也没有返回值。它从编号最小的结点出发开始搜索，并将对当前对象的深度优先搜索的序列显示在显示器上。

# 深度优先搜索的实现

- 以邻接表为例
- 设置一个数组visited，记录节点是否被访问过
- 设计一个私有的深度优先搜索的函数，从某一节点出发访问所有可达节点
- 如果是无向非连通图的或有向非强连通，则对图中尚未访问的节点反复调用深度优先搜索，形成深度优先搜索的森林。

# 公有的dfs函数

```
void dfs()  
{  
    对每个节点 v  
        visited [v] = false;  
  
    while (v = 尚未访问的节点)  
        dfs(v, visited);  
}
```

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::dfs() const
{bool *visited = new bool[Vers];

for (int i=0; i < Vers; ++i) visited[i] = false;

cout << "当前图的深度优先遍历序列为: " << endl;
for (i = 0; i < Vers; ++i) {
    if (visited[i] == true) continue;
    dfs(i, visited);
    cout << endl;
}
}
```

# 私有的dfs

访问从结点v出发可以访问到的所有结点

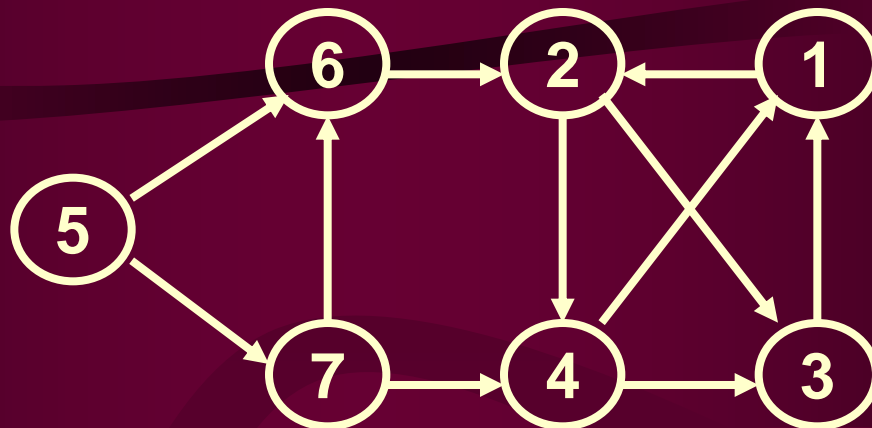
```
void dfs( v, visited )  
{  
    visited[v] = true;  
    for 每个 v的邻接点w  
        if( ! Visited[w] )  
            dfs( w,visited );  
}
```

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::dfs
    (int start, bool visited[]) const
{ edgeNode *p = verList[start].head;

    cout << verList[start].ver << '\t';
    visited[start] = true;
    while (p != NULL){
        if (visited[p->end] == false) dfs(p->end, visited);
        p = p->next;
    }
}
```



对图调用dfs



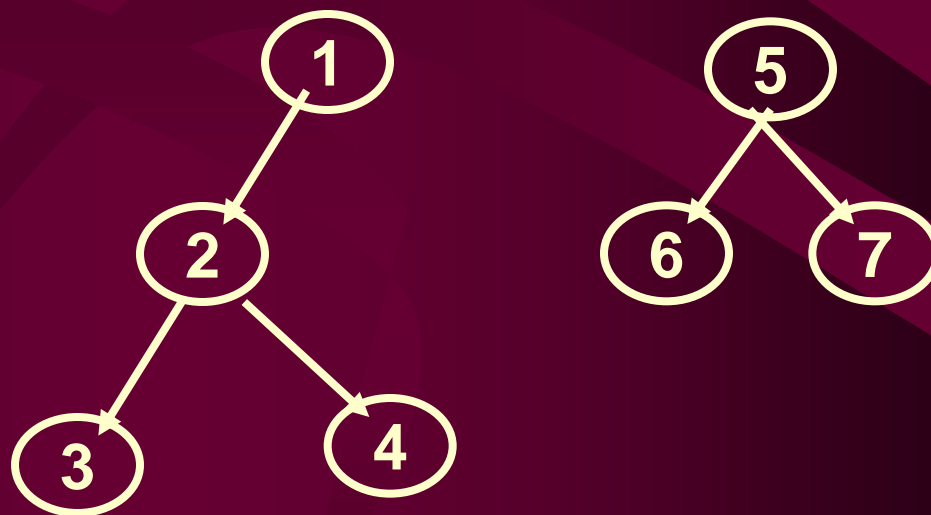
结果为:

当前图的深度优先搜索序列为:

1 2 4 3

5 6 7

即对应于左图的  
深度优先生成森林



# 时间性能分析

- dfs函数将对所有的顶点和边进行访问，因此它的时间代价和顶点数  $|V|$  及边数  $|E|$  是相关的，即是 $O(|V|+|E|)$ 。
- 如果图是用邻接矩阵来表示，则所需要的时间是 $O(|V|^2)$ 。

# 图的遍历

对有向图和无向图进行遍历是按照某种次序系统地访问图中的所有顶点，并且使得每个顶点只能被访问一次。在图中某个顶点可能和图中的多个顶点邻接并且存在回路，因此在图中访问一个顶点 $u$ 之后，在以后的访问过程中，又可能再次返回到顶点 $u$ ，所以图的遍历要比树的遍历更复杂

- 深度优先搜索
- 广度优先搜索

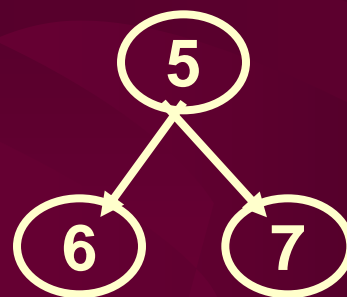
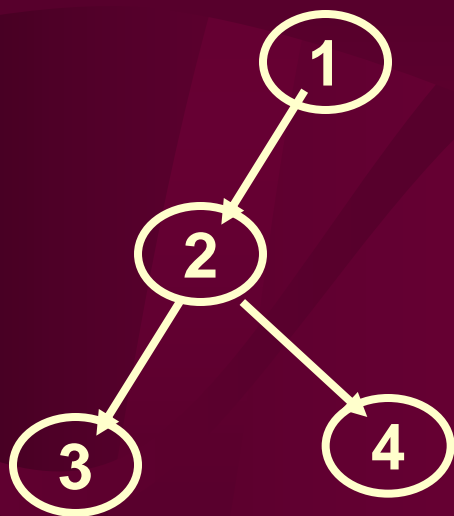
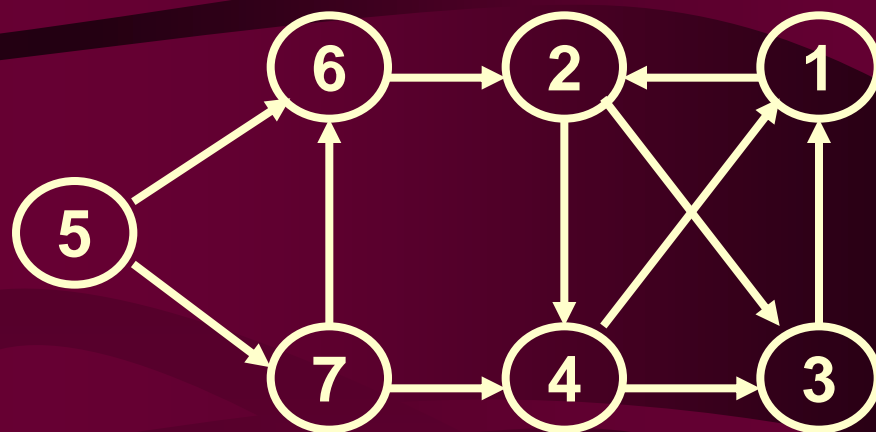


# 广度优先搜索 BFS

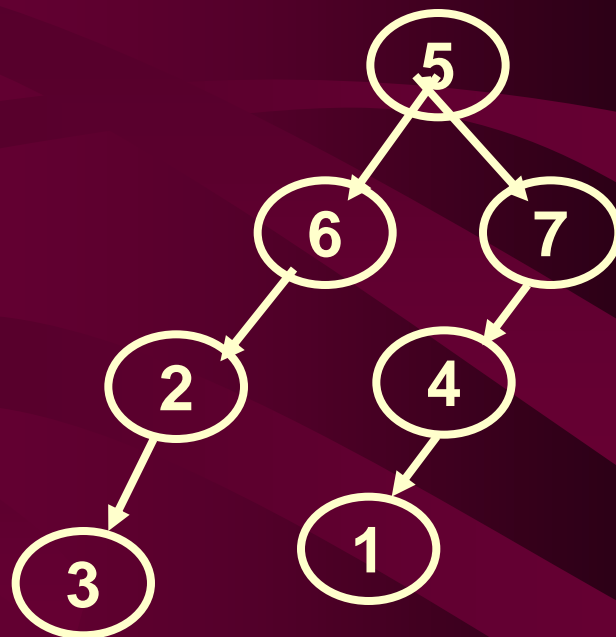
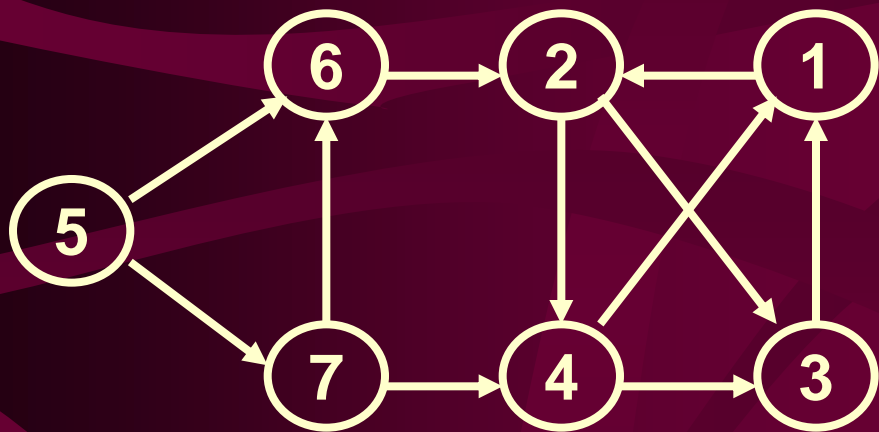
广度优先搜索类似于树的从树根出发的按照层次的遍历。它的访问方式如下：

- 1、选中第一个被访问的顶点；
- 2、对顶点作已访问过的标志；
- 3、依次访问已访问顶点的未被访问过的第一个、第二个、第三个.....第  $m$  个邻接顶点  $W_1$  、  $W_2$ 、 $W_3$ .....  $W_m$  ， 进行访问且进行标记，转向3；
- 4、如果还有顶点未被访问， 则选中一个起始顶点，转向2；
- 5、所有的顶点都被访问到， 则结束。

按照顶点序号小的先访问，序号大的后访问的原则，则它的广度优先访问序列为：1，2，4，3，5，6，7。  
对应的广度优先生成森林为



从不同的结点开始可以得到不同的搜索序列。例如，从5开始广度优先搜索这个图，得到的遍历序列为：5，6，7，2，4，3，1。



# 广度优先搜索的实现

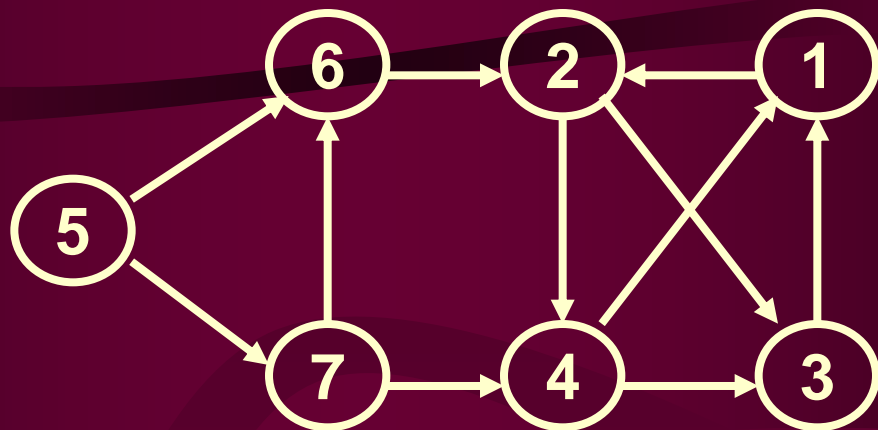
- 需要记录每个结点是否已被访问
- 需要记住每个已被访问的结点的后继结点，然后依次访问这些后继结点。这可以用一个队列来实现
- 过程：
  - 将序号最小的顶点放入队列
  - 重复取队列的队头元素进行处理，直到队列为空。对出队的每个元素，首先检查该元素是否已被访问。如果没有被访问过，则访问该元素，并将它的所有的没有被访问过的后继入队
  - 检查是否还有结点未被访问。如果有，重复上述两个步骤

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::bfs()
{
    const
    { bool *visited = new bool[Vers];
      int currentNode;
      linkQueue<int> q;
      edgeNode *p;
      for (int i=0; i < Vers; ++i) visited[i] = false;
      cout << "当前图的广度优先遍历序列为: "
            << endl;
```



```
for (i = 0; i < Vers; ++i) {  
    if (visited[i] == true) continue;  
    q.enqueue(i);  
    while (!q.isEmpty()) {  
        currentNode = q.dequeue();  
        if (visited[currentNode] == true) continue;  
        cout << verList[currentNode].ver << '\t';  
        visited[currentNode] = true;  
        p = verList[currentNode].head;  
        while (p != NULL){  
            if (visited[p->end] == false) q.enqueue(p->end);  
            p = p->next;  
        }  
        cout << endl;  
    }  
}
```

对图调用bfs



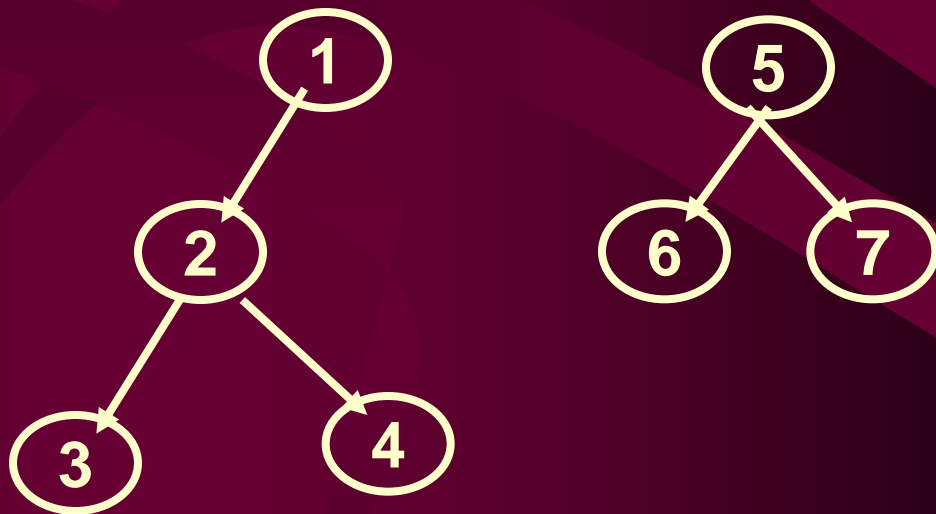
结果为:

当前图的广度优先搜索序列为:

1 2 4 3

5 6 7

即对应于左图的  
深度优先生成森林







# 时间性能分析

- 在广度优先搜索中，每个顶点都入队一次。
- 对于每个出队的结点，需要查看它的所有邻接结点。
- 假如图是用邻接表存储，查看所有的邻接结点需要 $O(|E|)$ 的时间，每个结点入队一次需要 $O(|V|)$ 的时间，因此广度优先遍历的时间复杂度为 $O(|E|+|V|)$ 。
- 如果图是用邻接矩阵存储，查看某个结点所有的边需要 $O(|V|)$ 的时间，因此广度优先搜索的时间复杂度为 $O(|V|^2)$ 。

# 第12章 图的基本概念

- 图的定义 
- 图的术语 
- 图的运算 
- 图的存储 
- 图的遍历 
- 图遍历的应用 





# 图遍历的应用

- 无向图的连通性 
- 欧拉回路 
- 有向图的连通性 
- 拓扑排序 

# 无向图的连通性

- 深度优先搜索和广度优先搜索都可以用来测试无向图的连通性。
- 如果无向图是连通的，则从无向图中的任意结点出发进行深度优先搜索或广度优先搜索都可以访问到每一个结点。访问的次序是一棵深度/广度优先生成树。
- 如果图是非连通的，深度/广度优先搜索可以找到一片深度/广度优先生成森林。每棵树就是一个连通分量。对无向图来说，深度/广度优先搜索可以找到了它的所有连通分量。
- 前面介绍的讨论的深度优先和广度优先遍历中，都已实现了这个功能。在这两个函数的输出中，每一行代表一个连通分量。

# 图遍历的应用

- 无向图的连通性 
- 欧拉回路 
- 有向图的连通性 
- 拓扑排序 

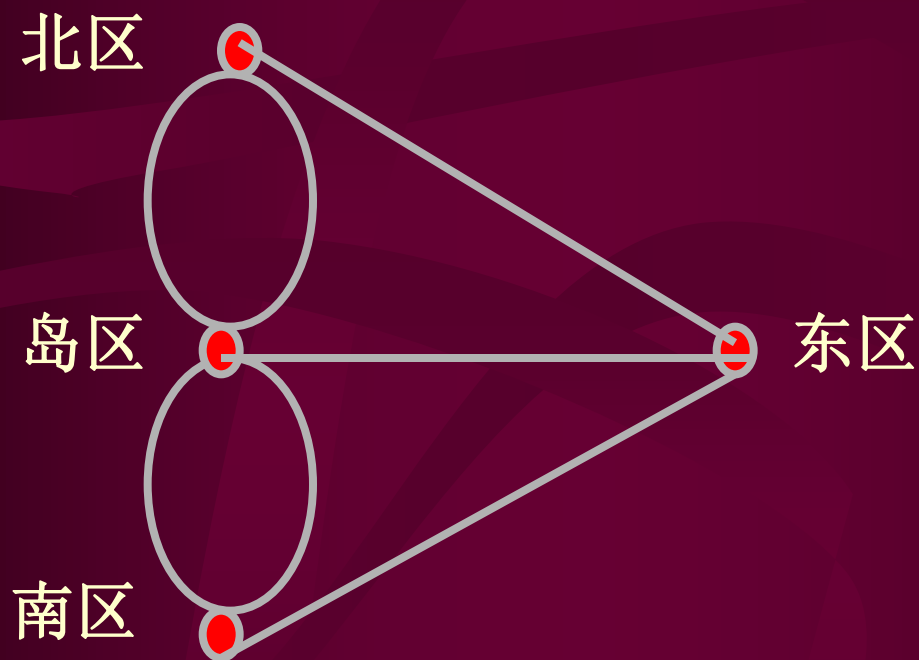
# 欧拉回路

- 哥尼斯堡七桥问题就是：能否找到一条走遍这七座桥，而且每座桥只经过一次，最后又回到原出发点的路径。





# 七桥问题的抽象



# 欧拉的证明

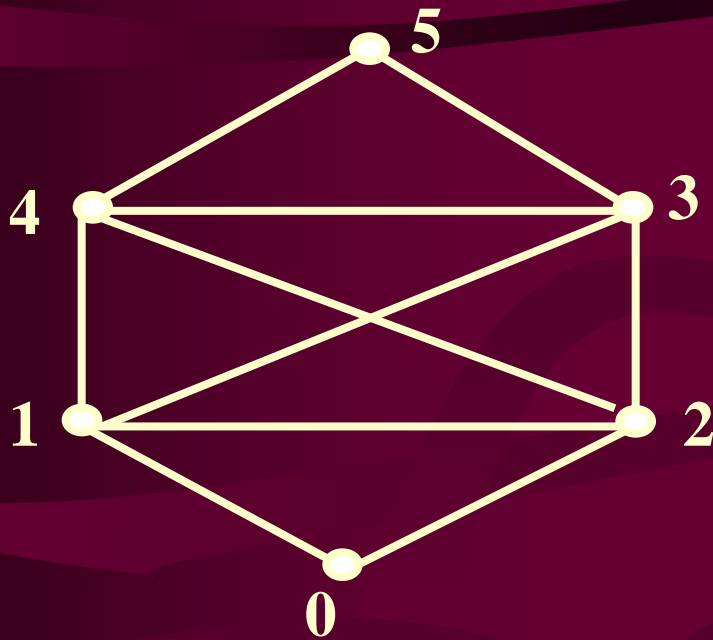
- 如果有奇数桥的地方不止两个，满足要求的路径是找不到的。
- 如果只有两个地方有奇数桥，可以从这两个地方之一出发，经过所有的桥一次，再回到另一个地方。
- 如果都是偶数桥，从任意地方出发都能回到原点。

# 欧拉回路和欧拉路径

- 如果能够在图中找到一条路径，使得该路径对图的每一条边正好经过一次，这条路径被称为**欧拉路径**。
- 如果再增加一个附加条件，即起点和终点是相同的，这条路径被称为**欧拉回路**。

# 基本想法

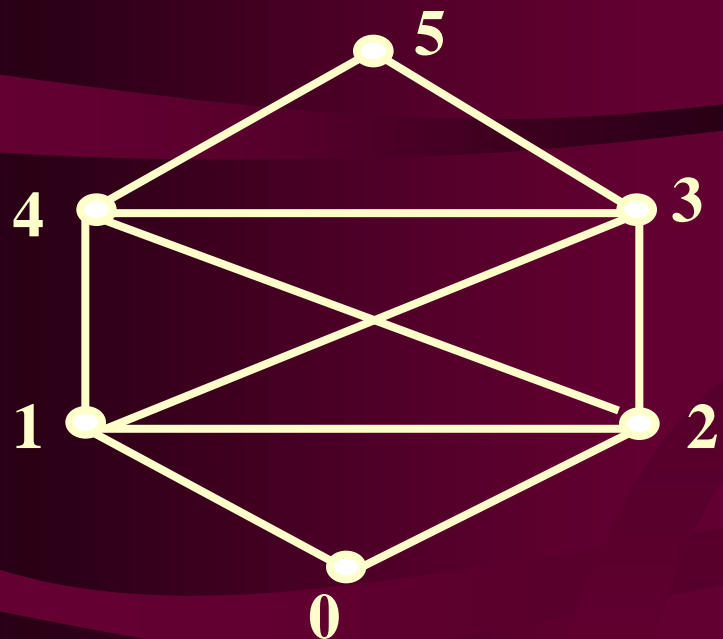
- 执行一次深度优先的搜索。从起始结点开始，沿着这条路一直往下走，直到无路可走。而且在此过程中不允许回溯。因此欧拉回路问题也被称为**一笔画问题**。
- 但是有很多的搜索方案是行不通的。



对于上图，它的所有结点的度均为偶数，应该存在欧拉回路。但如果从结点5出发开始深度优先的访问，选择的路径为5->4->3->5，则此时，就无法访问其他结点了，因为5没有其他的尚未被访问的边了。

# 解决方法

- 找出路径上的另外一个尚有未访问的边的顶点，开始另一次深度优先的搜索，将得到的遍历序列拼接到原来的序列中，直到所有的边都已被访问。



先找到  $5 \rightarrow 4 \rightarrow 3 \rightarrow 5$

在路径上找一个尚有边未被访问的结点，如：4，开始另一次深度优先遍历。得到路径  $4 \rightarrow 2 \rightarrow 1 \rightarrow 4$

将第二条路径拼接到第一条路径上，得到：

$5 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5$

3号结点还有未访问的边，从3号结点再开始一次深度优先遍历，得到路径  $3 \rightarrow 1 \rightarrow 0 \rightarrow 2 \rightarrow 3$

将第三条路径拼接到第一条路径上，得到：

$5 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 0 \rightarrow 2 \rightarrow 3 \rightarrow 5$

# 寻找欧拉回路

- 检查存在性
- 找出回路：
  - 执行一次深度优先的搜索。从起始结点开始，沿着这条路一直往下走，直到无路可走。而且在此过程中不允许回溯。
  - 路径上是否有一个尚有未访问的边的顶点。如果有，开始另一次深度优先的搜索，将得到的遍历序列拼接原来的序列中，直到所有的边都被访问。



# 欧拉回路的实现

- 欧拉回路是由一段一段的路径拼接起来的。为此，设计了一个私有的成员函数EulerCircuit来获得一段路径。公有的EulerCircuit函数调用私有的EulerCircuit函数获得一段段的路径，并将它们拼接起来，形成一条完整的欧拉回路。
- 为了拼接方便起见，找到的欧拉回路被保存在一个单链表中，单链表的结点类型为EulerNode。
- EulerNode保存两个内容：结点的编号和下一结点的指针。它被定义为邻接表类的私有的内嵌类。

## 欧拉回路的实现 续

- 欧拉回路中，一条边不能走两遍。为此，当一条边被访问以后，就将这条边删除
- Clone函数创建一份邻接表的拷贝，以便在找完路径后能恢复这个图的邻接表

# 公有的EulerCircuit函数

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::EulerCircuit
    (TypeOfVer start)
{ EulerNode *beg, *end, *p, *q, *tb, *te;
  int numOfDegree;
  edgeNode *r;
  verNode *tmp;

  //检查是否存在欧拉回路
  for (int i=0; i<Vers; ++i) {
      numOfDegree = 0; r = verList[i].head;
      while (r != 0) {++numOfDegree; r= r->next;}
      if (numOfDegree ==0 || numOfDegree % 2)
          {cout << "不存在欧拉回路" << endl; return;}
  }
```

```
//寻找起始结点的编号
for (i = 0; i<Vers; ++i)
    if (verList[i].ver == start) break;
if (i == Vers)
{ cout << "起始结点不存在" << endl;
  return;
}
```

```
//创建一份邻接表的拷贝
tmp = clone();
```

```
//寻找从i出发的路径,  
//路径的起点和终点地址分别是beg和end  
beg = EulerCircuit(i, end);  
while (true) {  
    p = beg;  
    while (p->next != NULL)  
        if (verList[p->next->NodeNum].head != NULL) break;  
        else p = p->next;  
    if (p->next == NULL) break;  
    q = p->next;  
    tb = EulerCircuit(q->NodeNum, te);  
    te->next = q->next;  
    p->next = tb;  
    delete q;  
}
```

**//恢复原图**

```
delete [] verList;  
verList = tmp;
```

**//显示得到的欧拉回路**

```
cout << "欧拉回路是: " << endl;  
while (beg !=NULL) {  
    cout << verList[beg->NodeNum].ver << '\t';  
    p = beg; beg = beg->next;  
    delete p;  
}  
cout << endl;  
}
```

# 欧拉路径中的结点类

```
struct EulerNode{  
    int NodeNum;  
    EulerNode *next;  
    EulerNode(int ver)  
        {NodeNum = ver; next =NULL;}  
};
```

# clone函数的实现

```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::verNode *
    adjListGraph<TypeOfVer, TypeOfEdge>::clone( ) const
{ verNode *tmp = new verNode[Vers];
  edgeNode *p;

  for (int i = 0; i < Vers; ++i) {
    tmp[i].ver = verList[i].ver;
    p = verList[i].head;
    while (p != NULL) {
      tmp[i].head =
        new edgeNode(p->end, p->weight, tmp[i].head);
      p = p->next;
    }
  }
  return tmp; }
```







# 私有的EulerCircuit函数

```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::EulerNode *adjListGraph
    <TypeOfVer, TypeOfEdge>::EulerCircuit(int start, EulerNode *&end)
{EulerNode *beg;
 int nextNode;
 beg = end = new EulerNode(start);
 while(verList[start].head != NULL) {
     nextNode = verList[start].head->end;
     remove( start, nextNode);
     remove(nextNode, start);
     start = nextNode;
     end->next = new EulerNode(start);
     end = end->next;
 }
 return beg;
}
```

# 哈密尔顿回路问题

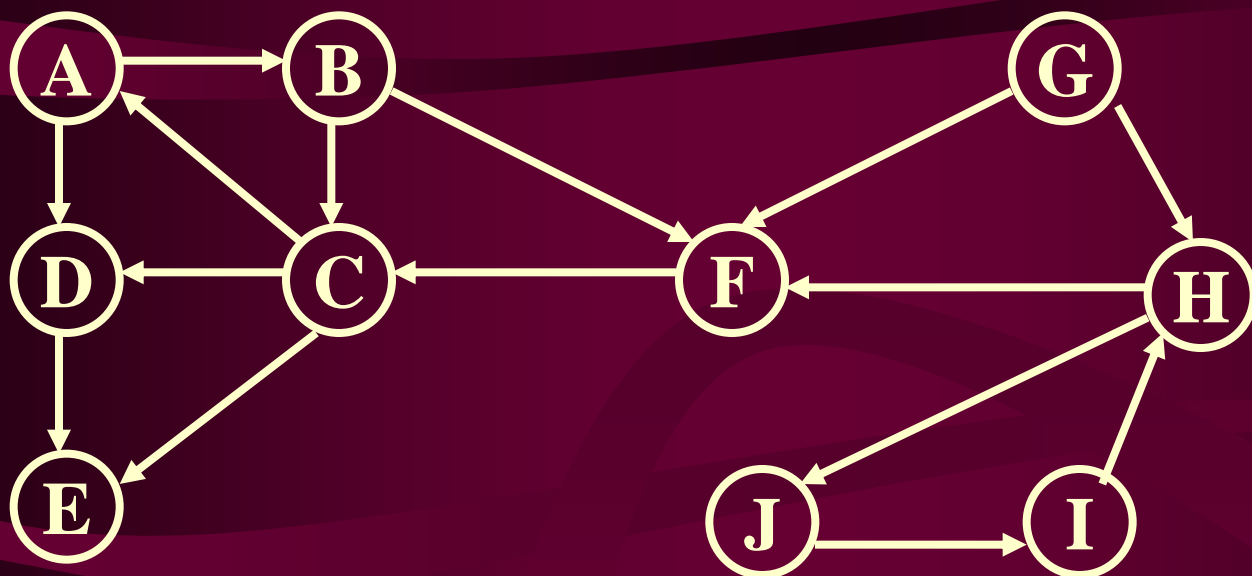
- 该回路通过图的每一个结点一次，且仅通过一次。
- 一个图是否存在哈密尔顿回路至今仍未找到满足该问题的充要条件。

# 图遍历的应用

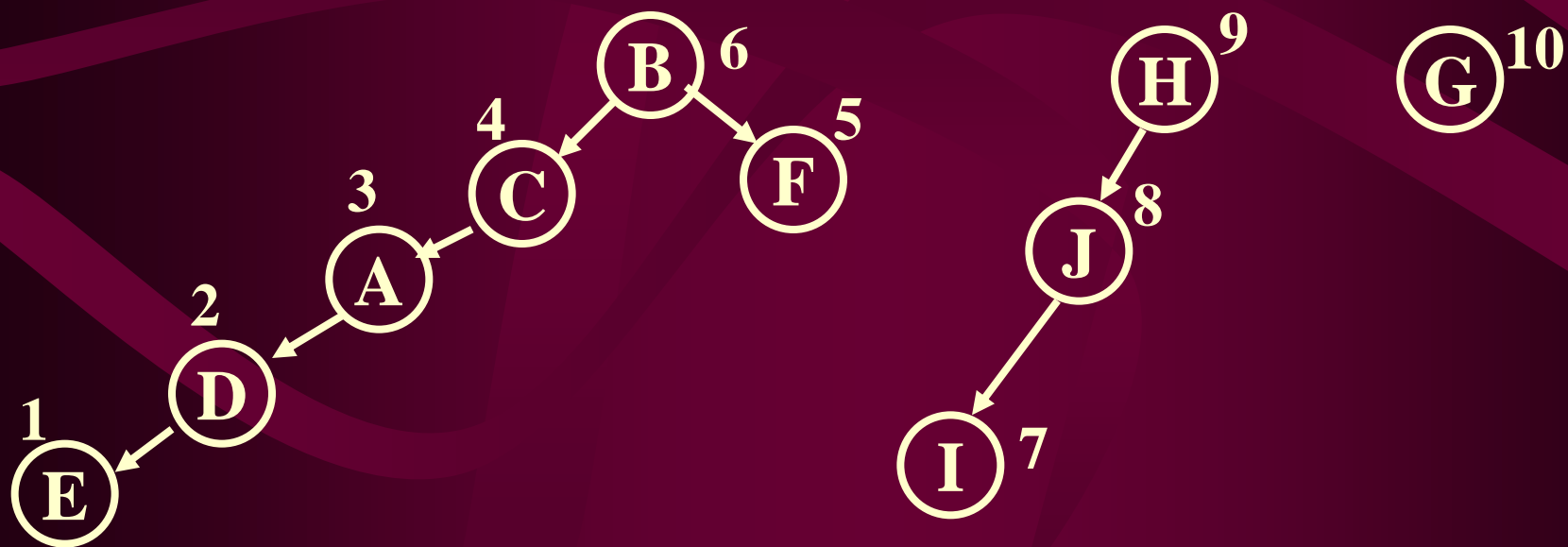
- 无向图的连通性 
- 欧拉回路 
- 有向图的连通性 
- 拓扑排序 

# 有向图的连通性

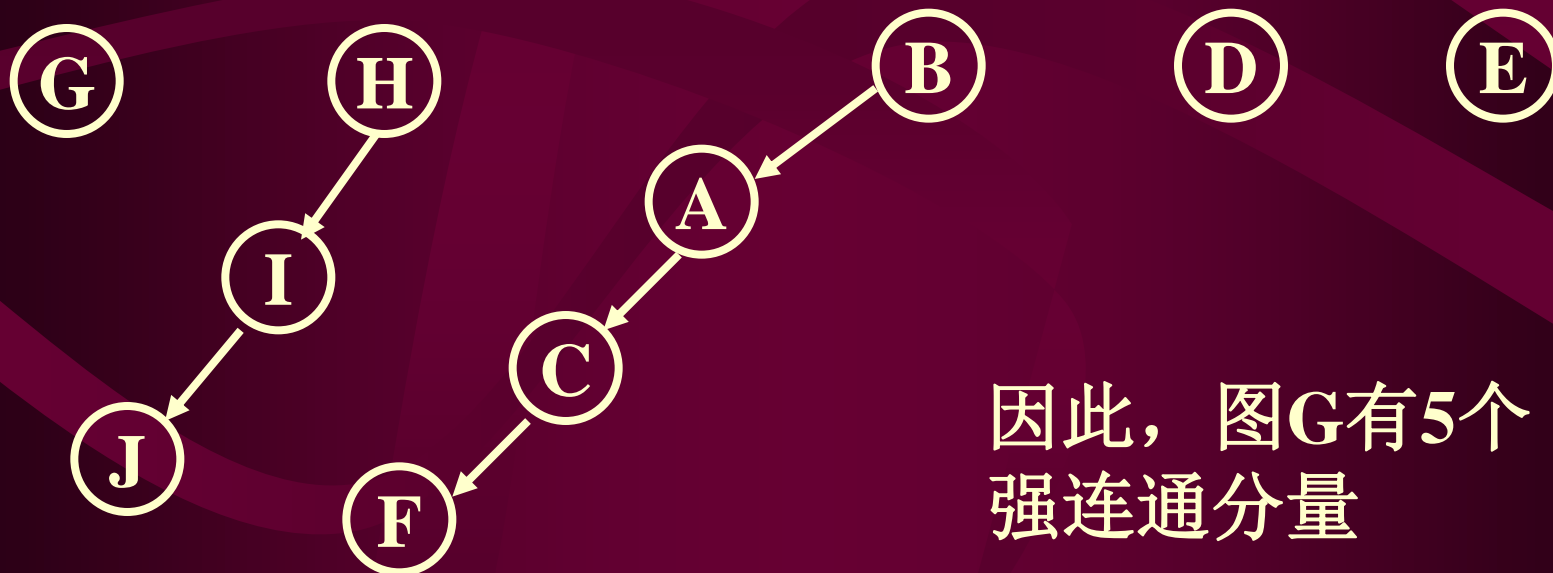
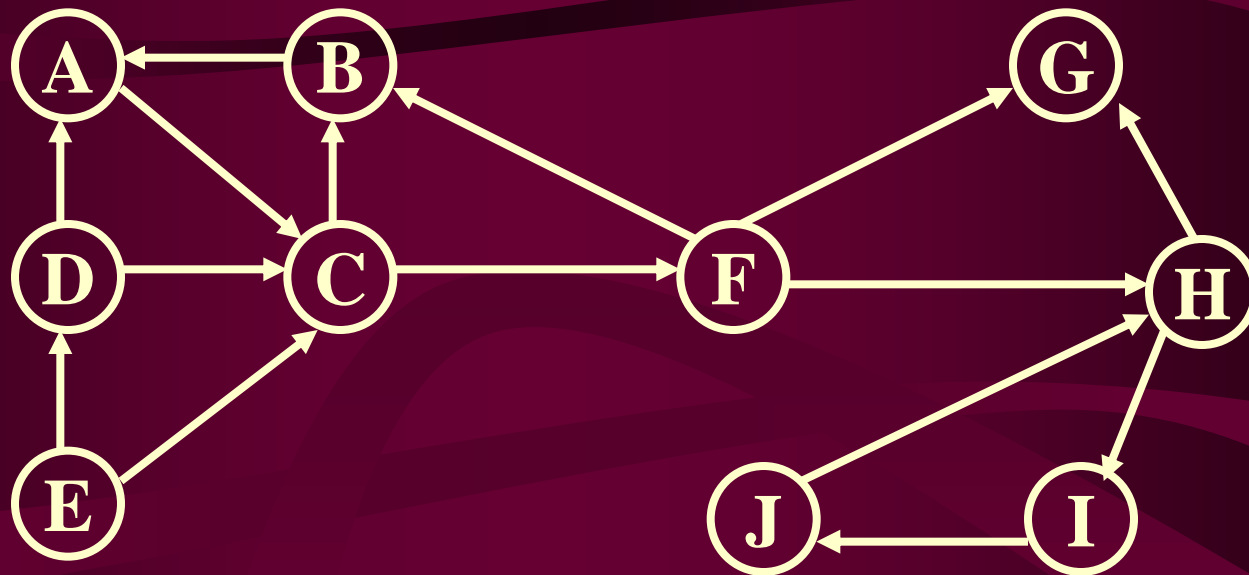
- 对有向图，深度优先搜索可以测试是否强连通，并找出所有强连通分量
- 找强连通分量的方法
  - 从任意节点开始深度优先遍历 $G$ 。对森林中的每棵树进行深度优先遍历，并按遍历的顺序给每个节点编号
  - 将 $G$ 的每条边逆向，形成 $G_r$ 。从编号最大的节点开始深度优先遍历 $G_r$ 。得到的深度优先遍历森林的每棵树就是 $G$ 的强连通分量。



从B开始深度优先搜索







Gr



因此，图G有5个  
强连通分量

# 图遍历的应用

- 无向图的连通性 
- 欧拉回路 
- 有向图的连通性 
- 拓扑排序 

# 拓扑排序

设 $G = (V, E)$ 是一个具有 $n$ 个顶点的有向无环图。 $V$ 中的顶点序列 $V_1, V_2, \dots, V_n$ 称为一个拓扑序列，当且仅当该序列满足下列条件：若在 $G$ 中，从 $V_i$ 到 $V_j$ 有一条路径，则序列中 $V_i$ 必须排在 $V_j$ 的前面。

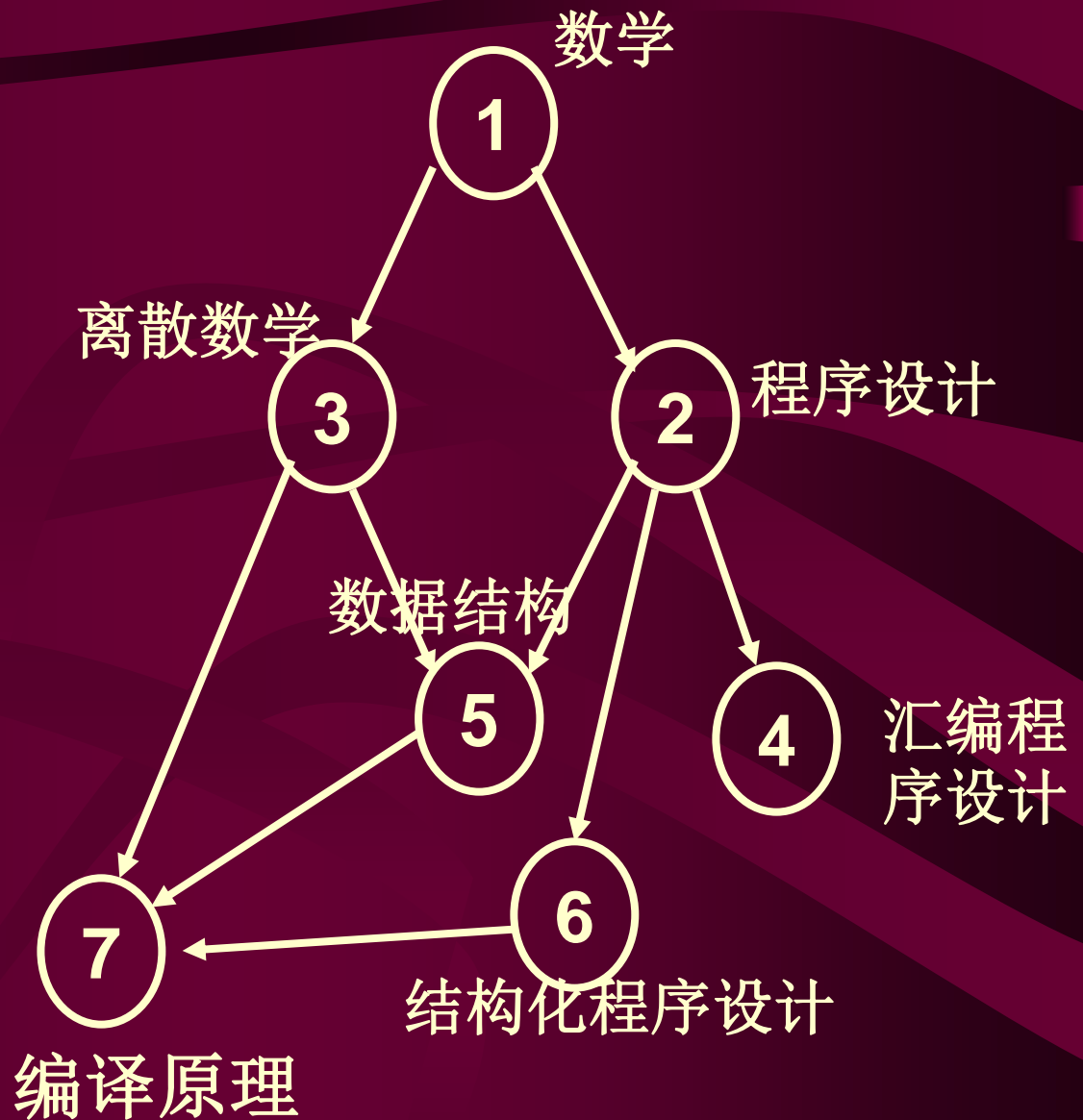


下述集合  $M$  代表课程的集合

- 1 代表数学,
- 2 代表程序设计,
- 3 代表离散数学,
- 4 代表汇编程序设计,
- 5 代表数据结构,
- 6 代表结构化程序设计,
- 7 代表编译原理

关系 $R$ 表示课程学习的先后关系, 如数学必须在离散数学之前学习。要求排一张学习的先后次序表。<sup>97</sup>

用有向图表示关系  
R。节点集为课程  
集合。如果课程i  
和j有关系R，则有  
一条边。



数学

1

离散数学

3

程序设计

2

数据结构

5

4

汇编程序  
设计

7

6

结构化程序设计

编译原理

可行的排课:

方案1:

1, 2, 3, 4, 5, 6, 7

方案2:

1, 2, 3, 5, 6, 4, 7

方案3:

1, 2, 3, 5, 6, 7, 4

。 。 。

# 找出拓扑排序的过程

- 第一个输出的结点（序列中的第一个元素）：  
必须无前驱，即入度为0
- 后驱：必须等到它的前驱输出之后才输出。
- 无前驱及后件的结点：任何时候都可输出。
- 逻辑删除法：当某个节点被输出后，就作为该节点被删除。所有以该节点作为前驱的所有节点的入度减1。



数学	0					
离散数学	1	0				
程序设计	1	0	0			
编译原理	3	3	2	2	1	0
数据结构	2	2	1	0		
结构化程序设计	1	1	1	0	0	
汇编程序设计	1	1	1	0	0	0

输出：

数学， 离散数学， 程序设计， 数据结构， 结构化程序设计，  
编译原理， 汇编程序设计

# 拓扑排序的实现

- 计算每个结点的入度，保存在数组inDegree中；
- 检查inDegree中的每个元素，将入度为0的结点入队；
- 不断从队列中将入度为0的结点出队，输出此结点，并将该结点的后继结点的入度减1；如果某个邻接点的入度为0，则将其入队。

```

template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::topSort( ) const
{ linkQueue<int> q;
  edgeNode *p;
  int current, *inDegree = new int[Vers];
  for (int i = 0; i < Vers; ++i)  inDegree[i] = 0;
  for ( i = 0; i < Vers; ++i)
      for (p = verList[i].head; p != NULL; p = p->next)
          ++inDegree[p->end];
  }
  for (i = 0; i < Vers; ++i)  if (inDegree[i] == 0) q.enqueue(i);
  cout << "拓扑排序为: " << endl;
  while( !q.isEmpty( ) ){
      current = q.dequeue( );
      cout << verList[current].ver << '\t';
      for (p = verList[current].head; p != NULL; p = p->next)
          if( --inDegree[p->end] == 0 )  q.enqueue( p->end );
      }
  cout << endl;
}

```



# 时间复杂度

- 如果图以邻接表表示
- 计算入度需要 $O(|V|+|E|)$ 的时间，搜索入度为0的结点需要 $O(|V|)$ 的时间。每个结点入一次队、出一次队。每出一次队，需要检查它的所有后继结点，因此也需要 $O(|V|+|E|)$ 的时间。所以总的执行时间也是 $O(|V|+|E|)$

# 总结

- 图是一种最一般的数据结构，有着广泛的用途。
- 图可以用邻接矩阵、邻接表和其他方法来存储
- 图的遍历：深度优先搜索和广度优先搜索，并给出了它们在邻接表的存储方式下的实现。
- 图的应用：
  - 检测无向图的连通性
  - 寻找无向图的欧拉回路
  - 寻找有向图的强连通分量
  - 拓扑排序

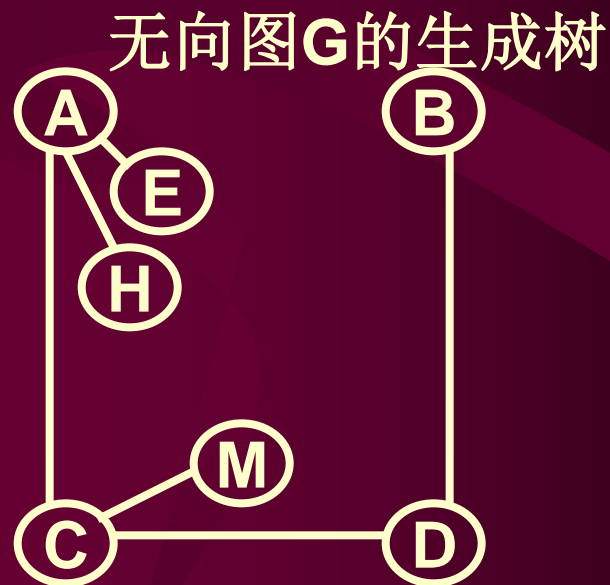
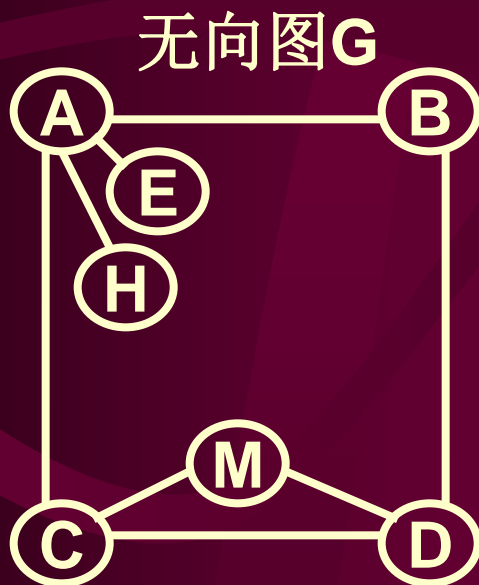
# 第13章 最小生成树

- 生成树与最小生成树
- Kruskal算法
- Prim算法
- 算法的正确性



# 生成树

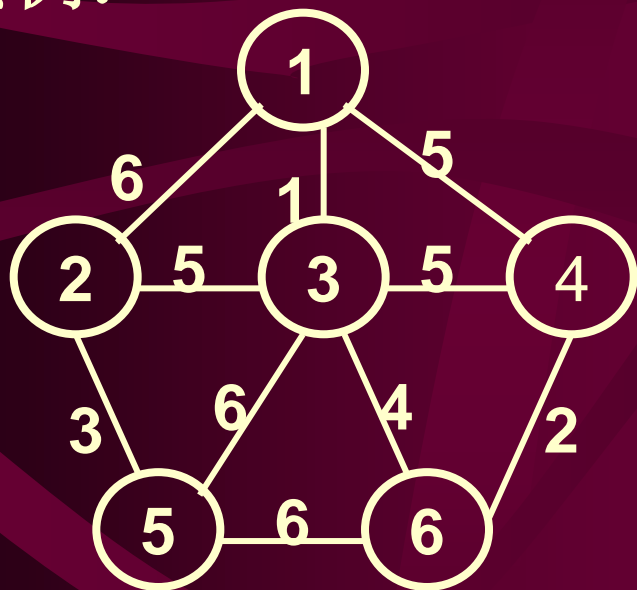
- 生成树是无向连通图的极小连通子图。包含图的所有  $n$  个结点，但只含图的  $n-1$  条边。在生成树中添加一条边之后，必定会形成回路或环。



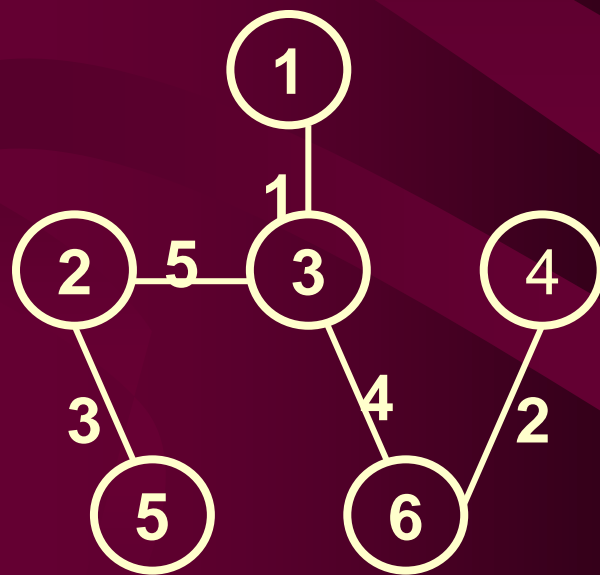
# 最小生成树

- 定义：加权无向图的所有生成树中边的权值（代价）之和最小的树。

- 实例：



- 左图的最小代价生成树



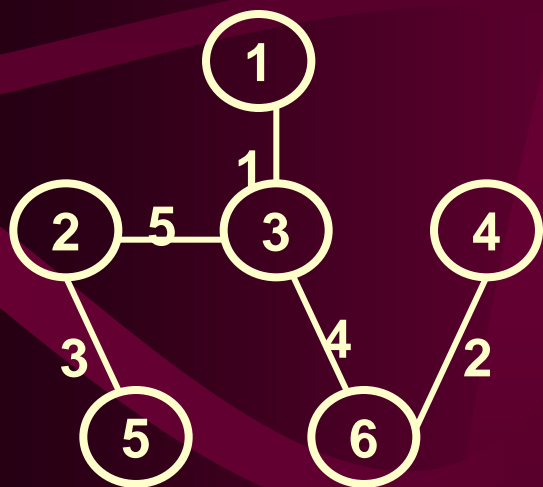
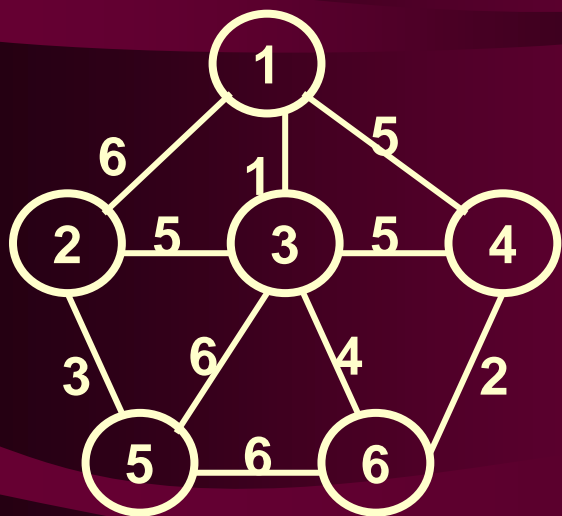
# 第13章 最小生成树

- 生成树与最小生成树
- Kruskal算法
- Prim算法
- 算法的正确性



# Kruscal 算法

- 基本思想：考虑图中权值最小的边。如果加入这条边不会导致回路，则加入；否则考虑下一条边，直到包含了所有的顶点
- 实现：
  - 初始时，设置生成树为  $(V, \Phi)$ ，如果  $V$  有  $n$  个顶点，则初始的生成树为具有  $n$  个连通分量的树。
  - 按权值的大小逐个考虑所有的边，如果改变的加入能连接两个连通分量，则加入。当生成树只有一个连通分量时，算法结束。



最小代价生成树

1、初始连通分量:  $\{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}$

2、反复执行添加、放弃动作。

边      动作

连通分量

$(1,3)$     添加       $\{1,3\}, \{4\}, \{5\}, \{6\}, \{2\}$

$(4,6)$     添加       $\{1,3\}, \{4,6\}, \{2\}, \{5\}$

$(2,5)$     添加       $\{1,3\}, \{4,6\}, \{2,5\}$

$(3,6)$     添加       $\{1,3,4,6\}, \{2,5\}$

$(1,4)$     放弃      因构成回路

$(3,4)$     放弃      因构成回路

$(2,3)$     添加       $\{1,3,4,5,6,2\}$



# 算法难点及解决方案

- 如何从所有边中选择代价最小的边：用一个优先级队列来实现。将所有的边放入一个优先级队列，边的优先级就是它的权值。权值越小，优先级越高。
- 如何判断加入一条边后会不会形成回路：用并查集来实现。将一个连通分量表示为并查集中的一个子集，检查一条边加入后会不会形成回路可以通过对边的两个端点分别执行Find操作。如果两个Find的结果相同，则表示两个端点已连通，加入这条边会形成回路，否则将这条边加入生成树。添加边的操作就是一个Union操作，将两个端点所属的子集归并起来，表示其中的所有顶点都已连通。

# 定义优先级队列中的元素类型

```
struct edge {  
    int beg, end;  
    TypeOfEdge w;  
    bool operator<(const edge &rp) const  
        {return w < rp.w;}  
};
```

# kruskal算法的实现

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::kruskal( ) const
{ int edgesAccepted = 0,  u, v;
  edgeNode *p;
  edge e;
  DisjointSet ds( Vers );
  priorityQueue<edge> pq;

  //生成优先级队列
  for (int i = 0; i< Vers; ++i) {
    for (p = verList[i].head; p != NULL; p = p->next)
      if (i < p->end) { e.beg = i;
                        e.end = p->end;
                        e.w = p->weight;
                        pq.enqueue(e); }
  }
```

//开始归并

```
while( edgesAccepted < Vers - 1 )
```

```
{ e = pq.dequeue();
```

```
  u = ds.Find(e.beg);
```

```
  v = ds.Find(e.end);
```

```
  if( u != v )
```

```
    { edgesAccepted++;
```

```
      ds.Union( u, v );
```

```
      cout << '(' << verList[e.beg].ver << ','
```

```
        << verList[e.end].ver << ")\t";
```

```
    }
```

```
  }
```

```
}
```

# 时间复杂度

- 生成优先级队列的for循环将所有的边入队。需要的时间是 $O(|E|\log|E|)$
- 在最坏的情况下，归并的循环可能需要检查所有的边。对于每条边，最多需要执行两次Find操作和一次Union操作。因此，归并循环的最坏情况的时间复杂度是 $O(|E|\log|V|)$ 。
- 在一个连通图中，一般边数总比结点数大，所以，Kruskal算法的时间复杂度是 $O(|E|\log|E|)$

# 第13章 最小生成树

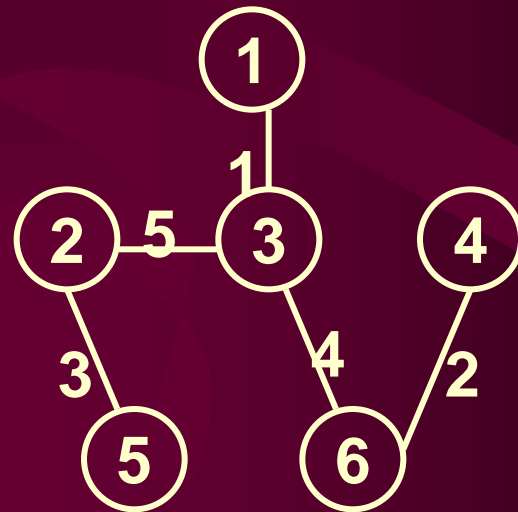
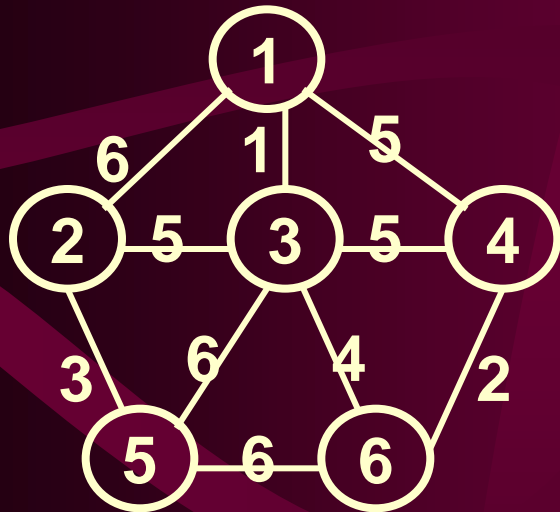
- 生成树与最小生成树
- Kruskal算法
- Prim算法
- 算法的正确性



# Prim算法

- 从顶点的角度出发。初始时，顶点集 $U$ 为空，然后逐个加入顶点，直到包含所有顶点。
- 过程：首先选择一个顶点，加入顶点集。然后重复下列工作，直到 $U = V$ 
  - 选择连接  $U$  和  $V-U$  中代价最小的边  $(u, v)$
  - 把  $(u, v)$  加入生成树的边集， $v$ 加入到 $U$

	可供选择的边	选择的边	U	V-U
1	初始时		{1}	{2,3,4,5,6}
2	(1,2,6),(1,3,1),(1,4,5)	(1,3)	{1,3}	{2,4,5,6}
3	(3,2,5),(3,4,5),(3,5,6),(3,6,4)	(3,6)	{1,3,6}	{2,4,5}
4	(3,2,5), (6,4,2),(6,5,6)	(6,4)	{1,3,4,6}	{2,5}
5	(3,2,5), (6,5,6),	(3,2)	{1,2,3,4,6}	{5}
6	(2,5,3)	(2,5)	{1,2,3,4,5,6}	





# Prim算法的实现

- 用一个布尔型的一维数组flag来保存哪些结点在U中，哪些结点不在U中的信息。
- 用两个一维数组lowCost和startNode来记录U中的结点到V-U中结点的权值最小的边。
- lowCost[i]表示U中的结点到结点i的边的最小权值。
- startNode[i]表示从U中的哪一个结点出发到结点i的权值是lowCost[i]。

# Prim算法的伪代码

```
Void prim ()
```

```
{ 初始化：将flag的元素全部置成false； 将lowCost的元素全部置成无穷大。
```

```
  设start = 0，表示第一个放入U的结点是0号结点；
```

```
  重复n-1次下列操作：
```

```
    { 对于start的每一条边 (start, v)
```

```
      如果v不在生成树中，并且边的权值w小于lowCost[v]
```

```
        { lowCost[v] = w; startNode[v] = start; }
```

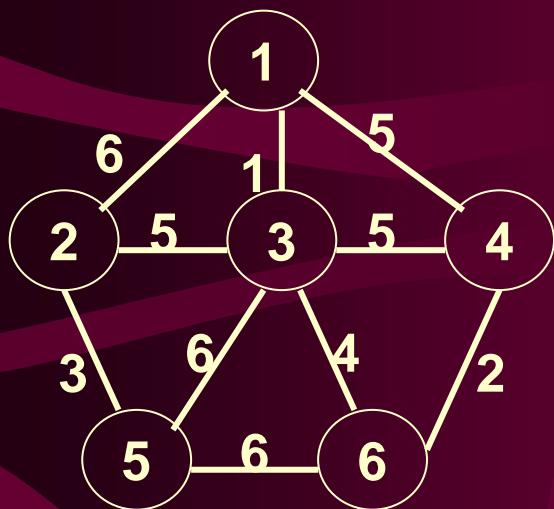
```
    flag[start] = true;
```

```
    从lowCost中寻找最小的元素，将下标存入start
```

```
  }
```

```
}
```

# prim算法运行过程中startNode和lowCost数组的变化



编号	startNode	lowCost
0	随机值	$\infty$
1	随机值	5
2	随机值	1
3	随机值	5
4	随机值	6
5	随机值	4

编号	visited
0	F
1	F
2	F
3	F
4	F
5	F

# 邻接表类中prim算法的实现

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::prim
    (TypeOfEdge noEdge) const
{ bool *flag = new bool[Vers];
  TypeOfEdge *lowCost = new TypeOfEdge[Vers];
  int *startNode = new int[Vers];
  edgeNode *p;
  TypeOfEdge min;
  int start, i, j;

  for (i = 0; i < Vers; ++i) {
      flag[i] = false;
      lowCost[i] = noEdge;  }
```

```

start = 0;
for ( i= 1; i < Vers; ++i) {
    for (p = verList[start].head; p != NULL; p = p->next)
        if (!flag[p->end] && lowCost[p->end] > p->weight) {
            lowCost[p->end] = p->weight;
            startNode[p->end] = start; }
    flag[start] = true;
    min = noEdge;
    for (j = 0; j < Vers; ++j)
        if (lowCost[j] < min) {min = lowCost[j]; start = j;}
    cout << '(' << verList[startNode[start]].ver << ', '
        << verList[start].ver << ")\t";
    lowCost[start] = noEdge;
}
delete [] flag;
delete [] startNode;
delete [] lowCost;
}

```

# 时间复杂度

- 函数的主体是一个嵌套循环，外层循环执行 $|V|$ 次，内层循环也执行 $|V|$ 次。所以，Prim算法的时间复杂度是 $O(|V|^2)$ 。

# 第13章 最小生成树

- 生成树与最小生成树
- Kruskal算法
- Prim算法
- 算法的正确性



# 算法的正确性

- 定理13.1: 假设  $G = \{V, E\}$  是一个连通图,  $U$  是顶点集合  $V$  的一个非空子集。若  $(u, v)$  是一条代价最小的边, 且  $u \in U$ ,  $v \in V - U$ , 则必存在一棵包括边  $(u, v)$  在内的最小生成树。



# 定理的证明

- 用反证法证明。
- 假定在图  $G = \{V, E\}$  中，存在一棵不包括代价最小的边  $(u, v)$  在内的最小生成树，设其为  $T$ 。将边  $(u, v)$  添加到树  $T$ ，由于顶点  $u, v$  本来就是连通的，现在又增加了一条新的通路，所以便形成了一条包含边  $(u, v)$  的回路。因此，必定存在另一条边  $(u', v')$ ，且  $u' \in U, v' \in V - U$ 。为了消除上述的回路，可以将边  $(u', v')$  删除。记为  $T' = T + (u, v) - (u', v')$ 。  $T'$  仍然包含  $V$  的所有的顶点，而且这些顶点之间仍然是连通的，而且代价比  $T$  小。所以新的生成树  $T'$  将是代价最小的树。和原假设矛盾，原命题得证。

# 总结

- 最小生成树是加权无向连通图的权值和最小的极小连通子图，它有很重要的应用价值。
- 寻找最小生成树的两个经典算法：Kruskal算法和Prim算法，给出了它们在邻接表类中的实现。
- 最后证明了两个算法的正确性。





# 第14章 最短路径问题

- 单源最短路径
- 所有顶点对间的最短路径



# 单源最短路径

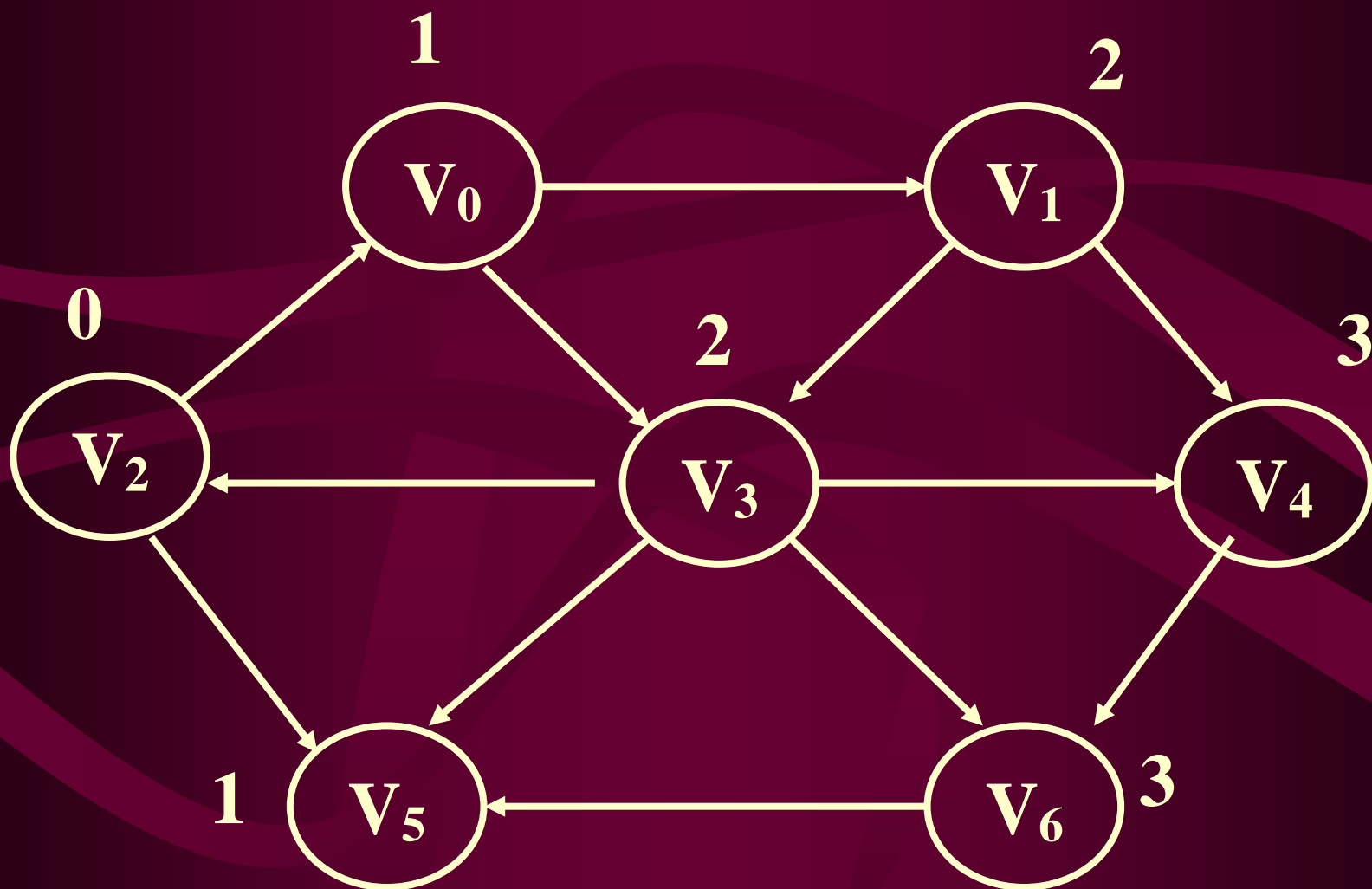
给出一个加权图和图上的一个节点 $s$ ，找出 $s$ 到图中每一节点的最短路径

- 非加权图的最短路径 
- 加权图的最短路径 
- 带有负权值的图 
- 无环图 

# 非加权的最短路径

- 采用广度优先搜索，它按层处理一层的所有结点：离起始结点最近的结点最先处理，距离最远的最晚处理。
- 具体过程：
  - 从s到s的最短路径为0
  - 通过搜索S的所有邻接结点就能找到离S距离为1的所有结点
  - 搜索离S距离为1的所有结点的邻接结点就能找到距离为2的节点
  - 重复上述过程，直到所有节点都访问到为止

找 $v_2$ 到其他节点的最短距离



# 存储设计

- 数组distance: 记录从源点到达每个结点的最短距离。
- 数组prev: 记录要到达此结点, 必须到达的前一结点。
- 例如, 对于上图中的结点v4, prev[v4]记录的是v1。也就是说, 从源点到达v4必须先到v1。而prev[v1]记录的是v0, prev[v0]记录的是v2。从prev数组, 我们可以追溯到这条路径。例如, 对于v4, 我们可以追溯到这条路径为v2-v0-v1-v4。

# 过程抽象

```
unweightedShortDistance(start)
{ for (每个结点v) distance[v] = 无穷大;
  distance[start] = 0;
  prev[start] = 0;
  for (int curDistance = 0; curDistance < 结点数; ++curDistance)
    { for (每个结点u)
      if (distance[u] == curDistance){
        for (u的每一个邻接点v)
          if (distance[v]== 无穷大) {
            distance[v]= curDistance+1;
            prev[v] = u;    }
          }
    }
}
```



# 分析

- 算法的时间复杂度为 $O(|V||E|)$ 。
- 算法效率之所以低的原因之一在于：为保证所有结点都找到了最短路径，算法假设最长的路径是经过所有的结点。
- 该算法效率之所以低的第二个原因在于：内层循环的设计。在处理距离为 $d$ 的结点时，我们找到了所有距离为 $d+1$ 的结点。但算法并没有利用这个成果，而在下一个循环周期中又用顺序查找的方法检查了所有结点，从中挑出距离为 $d+1$ 的结点。这浪费了大量的时间。

# 算法的改进

- 外层循环没必要执行 $|V|$ 次，只要所有的结点都已找到了最短距离就可以了。
- 第二层循环也没必要执行 $|V|$ 次，只要检查新找到最短路径的结点。用一个队列保存新找到路径的结点

# 改进算法的伪代码

```
unweightedShortDistance(start)
{ for (每个结点v) distance(v) = 无穷大;
  distance[start] = 0;
  start入队;
  while (队列非空) {
    取出队头元素存入u;
    for (u的每一个邻接点v)
      if (distance[v]== 无穷大) {
        distance[v]= distance[u] + 1;
        prev[v] = u;
        v入队;  }
  }
}
```

# 邻接表中的实现

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>
    ::unweightedShortDistance
    (TypeOfVer start, TypeOfEdge noEdge) const
{ linkQueue<int> q;
  TypeOfEdge *distance = new TypeOfEdge[Vers];
  int *prev = new int [Vers];
  int u, sNo;
  edgeNode *p;

  for (int i = 0; i< Vers; ++i) distance[i] = noEdge;
```

//寻找起始结点的编号

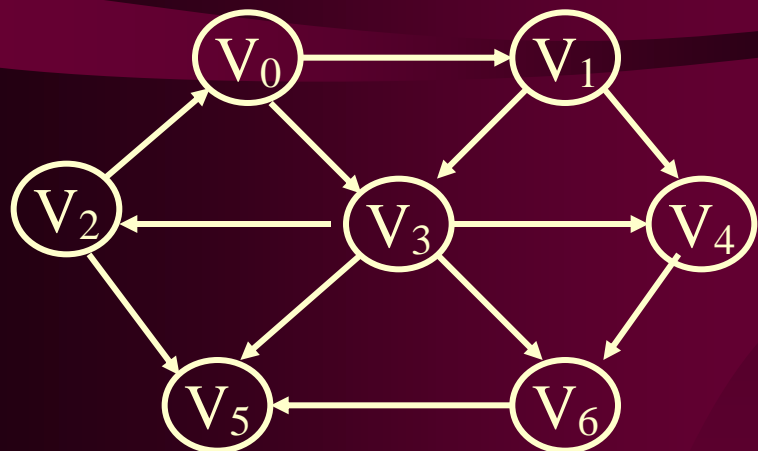
```
for (sNo = 0; sNo < Vers; ++ sNo) if (verList[sNo].ver == start) break;
if (sNo == Vers){
    cout << "起始结点不存在" << endl;
    return; }
}
```

//寻找最短路径

```
distance[sNo] = 0; prev[sNo] = sNo; q.enqueue(sNo);
while (!q.isEmpty()) {
    u = q.dequeue();
    for (p = verList[u].head; p != NULL; p = p->next)
        if (distance[p->end] == noEdge) {
            distance[p->end] = distance[u] + 1;
            prev[p->end] = u;
            q.enqueue(p->end); }
}
```

//输出最短路径

```
for (i = 0; i < Vers; ++i) {
    cout << "从" << start << "到" << verList[i].ver << "的路径为:" << endl;
    printPath(sNo, i, prev); cout << endl; }
}
```



队列的变化:

2

$d[2]=0$

0 5

$d[0]=1, d[5]=1$

5 1 3

$d[1]=2, d[3]=2$

1 3

3 4

$d[4]=3$

4 6

$d[6]=3$

6

空

算法结束

# printPath函数的实现

- 路径的存储不是正向的。即可以从第一个结点找到第二个结点，从第二个结点找到第三个结点，……。而是逆向存储的，即最后一个结点记住倒数第二个结点，倒数第二个结点记住倒数第三个结点，……。
- 适合用递归处理。于是我们定义了一个私有的递归成员函数printPath来输出这条路径。

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>
    ::printPath(int start, int end, int prev[]) const
{ if (start == end) {
    cout << verList[start].ver ;
    return;
}
printPath(start, prev[end], prev);
cout << " - " << verList[end].ver ;
}
```



# 函数的输出

从v2到v0的路径为:

$v2 - v0$

从v2到v1的路径为:

$v2 - v0 - v1$

从v2到v2的路径为:

$v2$

从v2到v3的路径为:

$v2 - v0 - v3$

从v2到v4的路径为:

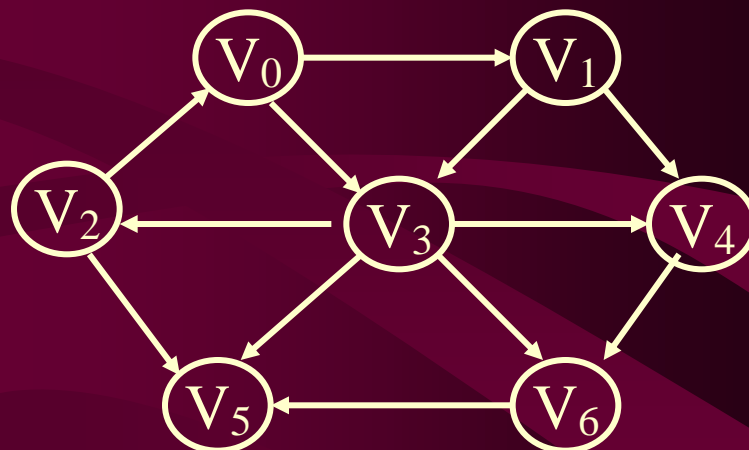
$v2 - v0 - v3 - v4$

从v2到v5的路径为:

$v2 - v5$

从v2到v6的路径为:

$v2 - v0 - v3 - v6$







# 时间复杂度

- 算法的主体是while循环，该循环一直执行到队列为空。而图中的每个结点都必须而且仅入队一次，因此该循环必须执行 $|V|$ 个循环周期。
- 每个循环周期检查出队结点的所有边，整个while循环检查了图中所有的边。因此，while循环的运行时间为 $O(|E|)$ 。前面的一些辅助工作，如初始化、寻找起始结点的编号等所需要的时间是 $O(|V|)$ 。所以算法的总运行时间为 $O(|V| + |E|)$ 。

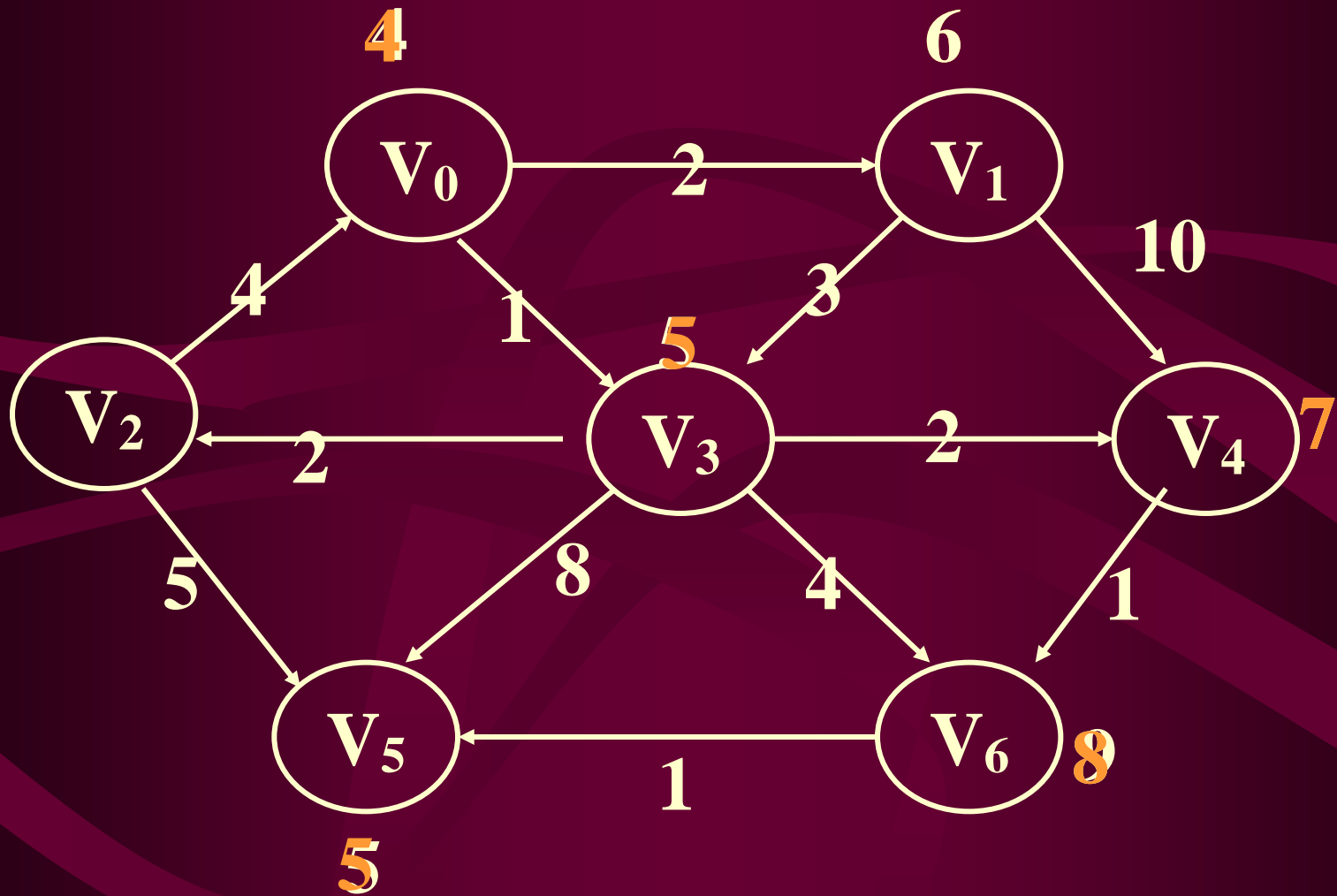
# 单源最短路径

给出一个加权图和图上的一个节点 $s$ ，找出 $s$ 到图中每一节点的最短路径

- 非加权图的最短路径 
- 加权图的最短路径 
- 带有负权值的图 
- 无环图 

# Dijkstra算法

- 保存了一个顶点集 $S$ 。 $S$ 中的顶点是已经找到了最短路径的顶点。
- 开始时，顶点集合 $S$ 只包含源点一个顶点。
- 反复执行以下循环，直至顶点集 $S$  包含所有的顶点为止：  
对于在顶点集 $V - S$ 中的每个顶点，考察新加入顶点集 $S$ 中的结点是否有边到达自己。如果存在，则检查这条路径是否比原来已知的路径要短。如果是，则更新源点到此结点的距离和路径；然后，从 $V - S$ 中寻找一个路径最短的结点，从源点到这个结点已经不可能有更好的路径了，把它加入顶点集 $S$



# 存储设计

- 和非加权图的实现中一样，需要保存的距离和路径信息
- 还需要保存哪些结点在S中，哪些结点不在S中的信息。
- 设计三个数组：**distance**、**prev**和**known**,

# 伪代码

```
void dijkstra( start )
{ for ( 图中的每个顶点v ) {
    distance[v] = INFINITY;
    known[v] = false;
}
distance[start] = 0;
for (i = 1; i < Vers; ++i) {
    v = 所有known标记为false的结点中的路径最短者;
    known[v] = true;
    for (v的每一个邻接点w)
        if (known[w] 等于false &&
            distance[v] + (v,w)的权值 < distance[w] )
            { distance[w] = distance[v] + (v,w)的权值
              Prev[w] = v;    }
    }
}
```

# 邻接表类中实现的Dijkstra算法

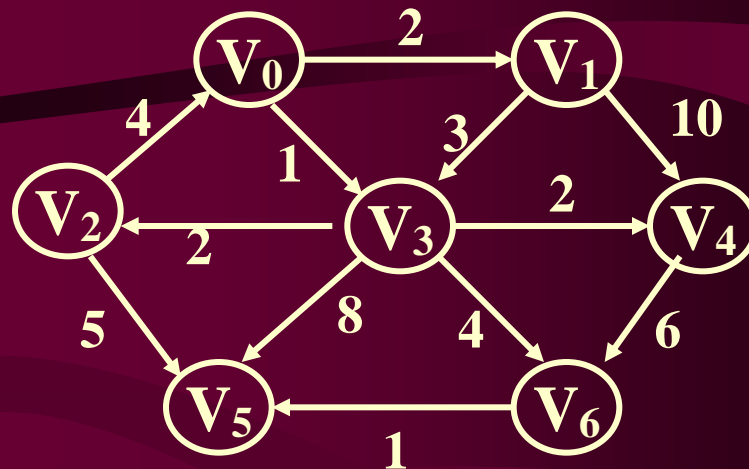
```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>
    ::dijkstra(TypeOfVer start, TypeOfEdge noEdge) const
{
    TypeOfEdge *distance = new TypeOfEdge[Vers];
    int *prev = new int [Vers];
    bool *known = new bool[Vers];
    int u, sNo, i, j;
    edgeNode *p;
    TypeOfEdge min;
    for (i = 0; i < Vers; ++i) { //初始化
        known[i] = false;
        distance[i] = noEdge;
    }
}
```



```
for (sNo = 0; sNo < Vers; ++sNo)  if (verList[sNo].ver == start) break;
    if (sNo == Vers){ cout << "起始结点不存在" << endl; return; }
    distance[sNo] = 0;  prev[sNo] = sNo;
```

```
for (i = 1; i < Vers; ++i) {
    min = noEdge;
    for (j = 0; j < Vers; ++j)
        if (!known[j] && distance[j] < min) { min = distance[j]; u = j; }
    known[u] = true; //将u放入S
    for (p = verList[u].head; p != NULL; p = p->next)
        if (!known[p->end] && distance[p->end] > min + p->weight) {
            distance[p->end] = min + p->weight;
            prev[p->end] = u;  }
    }
for (i = 0; i < Vers; ++i) { //输出所有的路径信息
    cout << "从" << start << "到" << verList[i].ver << "的路径为:" << endl;
    printPath(sNo, i, prev);
    cout << "\t长度为: " << distance[i] << endl;
}
}
```

# 执行dijkstra函数的过程：源点为v1



u	dist [0] prev[0] known[0]	dist [1] prev[1] known[1]	dist [2] prev[2] known[2]	dist[3] prev[3] known[3]	dist [4] prev[4] known[4]	dist [5] prev[5] known[5]	dist [6] prev[6] known[6]
	$\infty$ , -, F	0, 1, F	$\infty$ , -, F	$\infty$ , -, F	$\infty$ , -, F	$\infty$ , -, F	$\infty$ , -, F
1	$\infty$ , -, F	0, 1, T	$\infty$ , -, F	3, 1, F	10, 1, F	$\infty$ , -, F	$\infty$ , -, F
3	$\infty$ , -, F	0, 1, T	5, 3, F	3, 1, T	5, 3, F	11, 3, F	7, 3, F
2	9, 2, F	0, 1, T	5, 3, T	3, 1, T	5, 3, F	10, 2, F	7, 3, F
4	9, 2, F	0, 1, T	5, 3, T	3, 1, T	5, 3, T	10, 2, F	7, 3, F
6	9, 2, F	0, 1, T	5, 3, T	3, 1, T	5, 3, T	8, 6, F	7, 3, T
5	9, 2, F	0, 1, T	5, 3, T	3, 1, T	5, 3, T	8, 6, T	7, 3, T

从v1到v0的路径为:

$v1 - v3 - v2 - v0$  长度为: 9

从v1到v1的路径为:

$v1$  长度为: 0

从v1到v2的路径为:

$v1 - v3 - v2$  长度为: 5

从v1到v3的路径为:

$v1 - v3$  长度为: 3

从v1到v4的路径为:

$v1 - v3 - v4$  长度为: 5

从v1到v5的路径为:

$v1 - v3 - v6 - v5$  长度为: 8

从v1到v6的路径为:

$v1 - v3 - v6$  长度为: 7





## 输出结果

# 时间复杂度

- Dijkstra 算法运行时间主要由两部分组成：查找路径最短的尚未在S中的结点u，以及扫描u的邻接点，更新他们的距离。
- 每次找出距离最短的结点所需的时间是 $O(|V|)$ ，在整个算法的执行过程中，寻找距离最短的结点将花去 $O(|V|^2)$ 的时间。
- 更新V - S中的结点的距离所需的时间是 $O(|E|)$
- 总的时间复杂度是 $O(|E| + |V|^2) = O(|V|^2)$

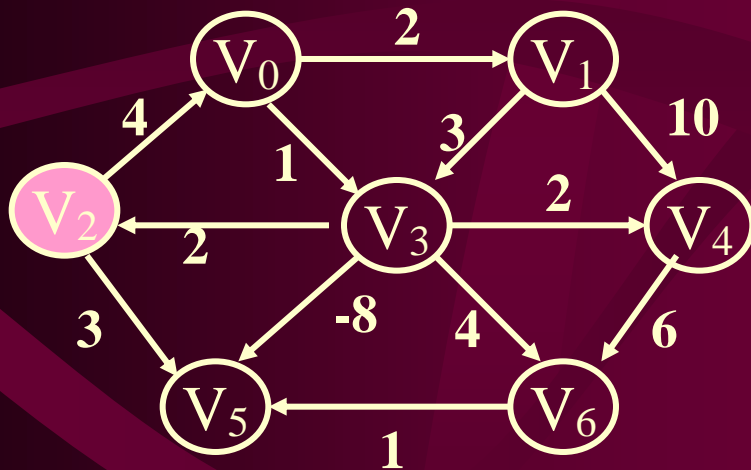
# 单源最短路径

给出一个加权图和图上的一个节点 $s$ ，找出 $s$ 到图中每一节点的最短路径

- 非加权图的最短路径 
- 加权图的最短路径 
- 带有负权值的图 
- 无环图 

# 带有负权值的图

- 如果一个图带有负权值的边，Dijkstra算法无法正常工作



如按Dijkstra算法，从结点2出发首先会认为结点5是已知节点，但事实上，2-0-3-5是一条更短的路径。

# 一个直观的解决方案

- 对每条边的权值都增加一个常量，使之消除负边，然后再使用Dijkstra算法。
- 问题：经过边数多的路径，权值增加得也多，并不等价于原问题

# 一个可行的解决方案

- 将加权图和非加权图算法组合起来
- 思想：放弃known的概念，穷举所有的路径，选择最短的一条。
- 实现：利用一个队列
  - 开始时，将源点s放入队列
  - 重复以下过程，直到队列为空：
    - 出队一个节点v
    - 对v的所有邻接点w，如果经过v到w的距离比已知的s到w的距离短，则更新w的距离，并将w入队



# 算法分析

- 适用于无负环的图
- 时间效益：每个节点至多出队 $v$ 次，运行时间是 $O(|E| \cdot |V|)$

# 单源最短路径

给出一个加权图和图上的一个节点 $s$ ，找出 $s$ 到图中每一节点的最短路径

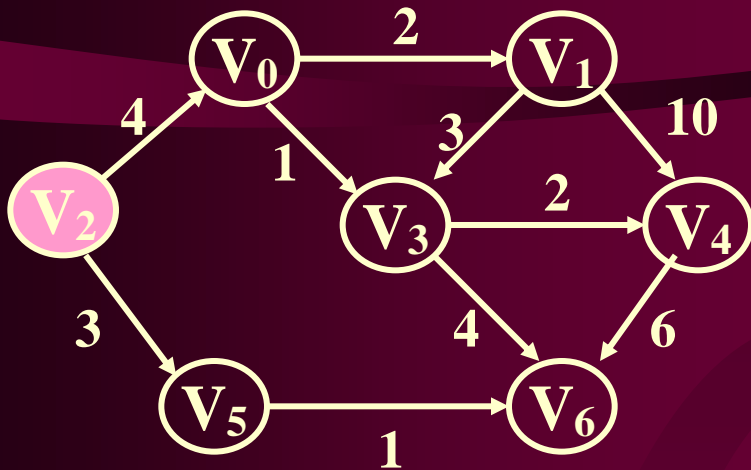
- 非加权图的最短路径
- 加权图的最短路径
- 带有负权值的图
- 无环图

# 无环图的最短路径

- 当图中无环时，可以对Dijkstra算法进行改进，使之达到 $O(|V|+|E|)$ 的时间效益
- 思想：按照拓扑排序的次序选择结点

拓扑排序:

2, 0, 5, 1, 3, 4, 6



	d	p	d	p	d	p	d	p	d	p	d	p
2	0											
0	$\infty$		4	2								
5	$\infty$		3	2								
1	$\infty$				6	0						
3	$\infty$				5	0						
4	$\infty$								16	1	7	3
6	$\infty$						4	5				

# 第14章 最短路径问题

- 单源最短路径
- 所有顶点对间的最短路径

# 所有节点对的最短路径问题

- 方法一，对每一结点运行Dijkstra最短路径算法；
- 方法二，用Floyd算法。时间复杂性 $O(N^3)$ 。具体思想是一次将每个节点作为中间节点，看看从起始点到中间结点，再从中间节点到终止点的距离是不是比已知的距离短。如是，则替代。

# Floyd 算法

- 使用n行n列的矩阵A用于计算最短路径。

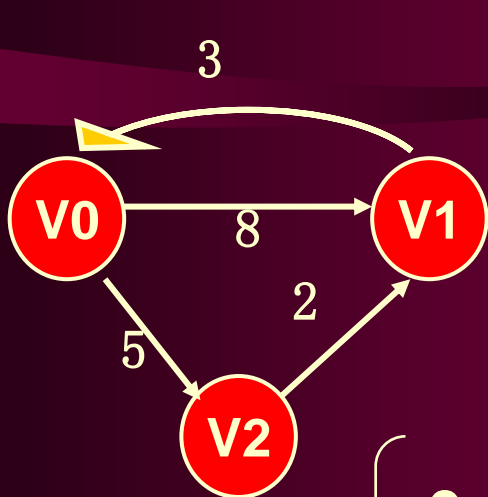
初始时,  $A[i, j] = c[i, j]$

- 进行n次迭代

在进行第 k 次迭代时, 我们将使用如下的公式

$$A_k[i, j] = \text{MIN} \begin{cases} A_{k-1}[i, j] \\ A_{k-1}[i, k] + A_{k-1}[k, j] \end{cases}$$

注意: 第k次迭代时, 针对结点k进行。原 $A_{k-1}$  矩阵的第k行, 第k列保持不变。左上至右下的对角线元素也不变。



	0	1	2
0	0	8	5
1	3	0	$\infty$
2	$\infty$	2	0

0	8	5
3	0	$\infty$
$\infty$	2	0

0	8	5
3	0	8
$\infty$	2	0

A初值

A0

0	8	5
3	0	8
5	2	0

0	7	5
3	0	8
5	2	0

A1

A2



# 存储设计

- Floyd算法除了给出任意两个结点之间的最短路径之外，还需要给出路径的组成。
- 每条路径信息也只保存这条路径上的前一结点的信息。
- 路径信息的保存需要一个 $|V| \times |V|$ 的二维数组prev。prev[i][j]的值表示从i到j的最短路径上的前一结点的序号。

# 邻接矩阵类中 Floyd算法的实现

```
template <class TypeOfVer, class TypeOfEdge>
void adjMatrixGraph<TypeOfVer, TypeOfEdge>::floyd() const
{ TypeOfEdge **d = new TypeOfEdge*[Vers];
  int **prev = new int*[Vers];
  int i, j, k;
```

//初始化

```
for (i = 0; i < Vers; ++i) {
    d[i] = new TypeOfEdge[Vers];
    prev[i] = new int[Vers];
    for (j = 0; j < Vers; ++j) {
        d[i][j] = edge[i][j];
        prev[i][j] = (edge[i][j] != noEdge) ? i : -1;
    }
}
```

//迭代过程

```
for (k = 0; k < Vers; ++k)
    for (i = 0; i < Vers; ++i)
        for (j = 0; j < Vers; ++j)
            if (d[i][k] + d[k][j] < d[i][j]) {
                d[i][j] = d[i][k] + d[k][j];
                prev[i][j] = prev[k][j]; }
}
```

//输出过程

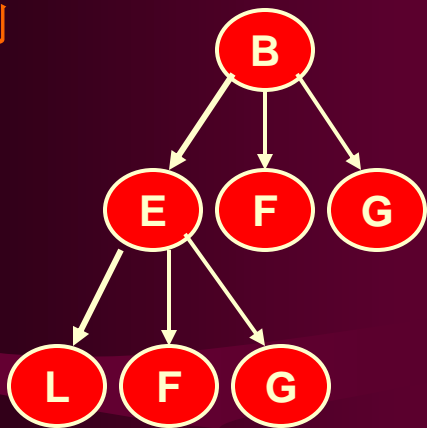
```
cout << "最短路径长度: " << endl;
for (i = 0; i < Vers; ++i) {
    cout << endl;
    for (j = 0; j < Vers; ++j) cout << d[i][j] << '\t'; }

cout << "最短路径: " << endl;
for (i = 0; i < Vers; ++i) {
    cout << endl;
    for (j = 0; j < Vers; ++j) cout << prev[i][j] << '\t'; }
}
```

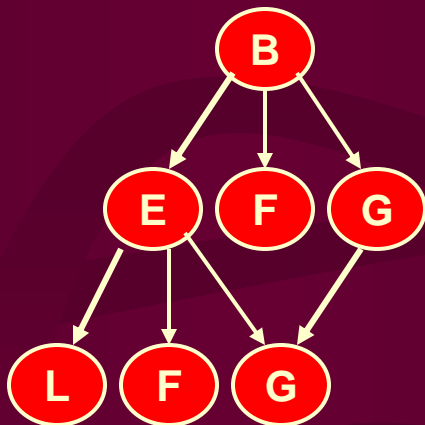
# 活动网络

## 1、何为有向无环图（DAG 图）

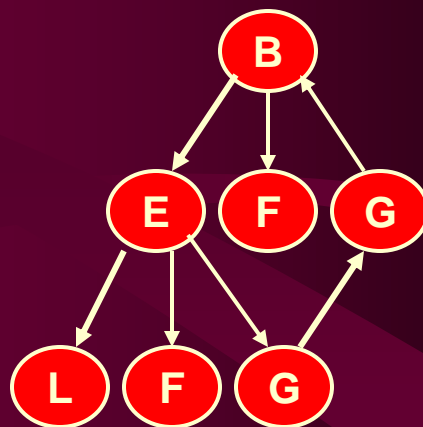
### • 实例



有向树



DAG图



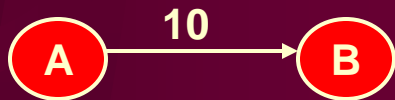
有向图（含环）

### • 用途：描述工程项目或系统进行的工具

**AOV 网络：**定义结点为活动，有向边的指向表示活动执行的次序。



**AOE网络：**定义结点为事件，有向边的指向表示事件的执行次序。单位是时间（时刻）。有向边定义为活动，它的权值定义为活动进行所需要的时间。



# 活动网络

## 2、拓扑排序 (Topological Sort) AOV 网络的应用。

- 偏序：若集合  $X$  上的关系  $R$  是传递的、自反的、反对称的，则称  $R$  是集合  $X$  上的偏序关系。
- 全序：若关系  $R$  是集合  $X$  上的偏序关系，如果对于每个  $x, y$  属于  $X$ ，必有  $x R y$  或  $y R x$ ，则称  $R$  是集合  $X$  上的全序关系。
- 拓扑排序：如果有一个序列  $A = a_1, a_2, a_3, \dots, a_n$  是集合  $X$  上的元素的一个序列，且当  $i < j$  时， $(a_i, a_j)$  属于  $R$ ，则称  $A$  是相对于  $R$  拓扑排序的。

注意：且当表示  $(a_i, a_j)$  属于  $R$ ， $i < j$  同时存在  
 $i$  和  $j$  是元素  $a_i$  和  $a_j$  在序列  $A$  中的序号。

用  表示  $(a_i, a_j)$  属于  $R$ ， $i < j$

为了表示的方便，令  为前件，而  为后件。前件的序号一定要小于后件。否则将不满足拓扑排序的定义。

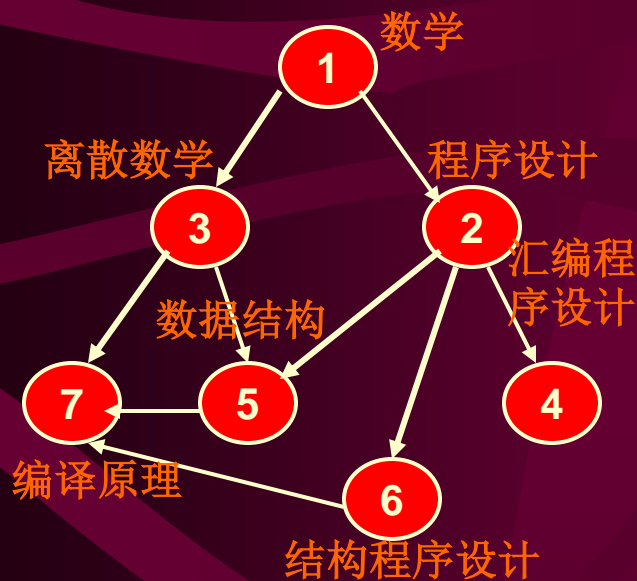
- 用途：描述工程项目或系统进行的次序

**AOV 网络**：定义结点为活动，有向边的指向表示活动执行的次序。

# 活动网络

## 2、拓扑排序 (Topological Sort)

- 实例：下述集合  $M$  代表课程的集合，其中，1代表数学，2代表程序设计，3代表离散数学，4代表汇编程序设计，5代表数据结构，6代表结构程序设计，7代表编译原理。关系  $R$  课程学习的先后关系，如数学必须在离散数学之前学习。要求排一张学习的先后次序表。



第一个输出的结点（序列中的第一个元素）：必须无前件

后件：必须等到它的前件输出之后，因此时序号才会大于前件的序号。

无前件及后件的结点：任何时候都可输出。

序列：1、3、2、4、6、5、7 合乎拓扑排序的要求

序号 1、2、3、4、5、6、7 前件的序号 < 后件的序号

序列：3、1、2、4、6、5、7 不合乎拓扑排序的要求

序号 1、2、3、4、5、6、7

元素 3 的序号 1 (后件) < 元素 1 (前件) 的序号 2

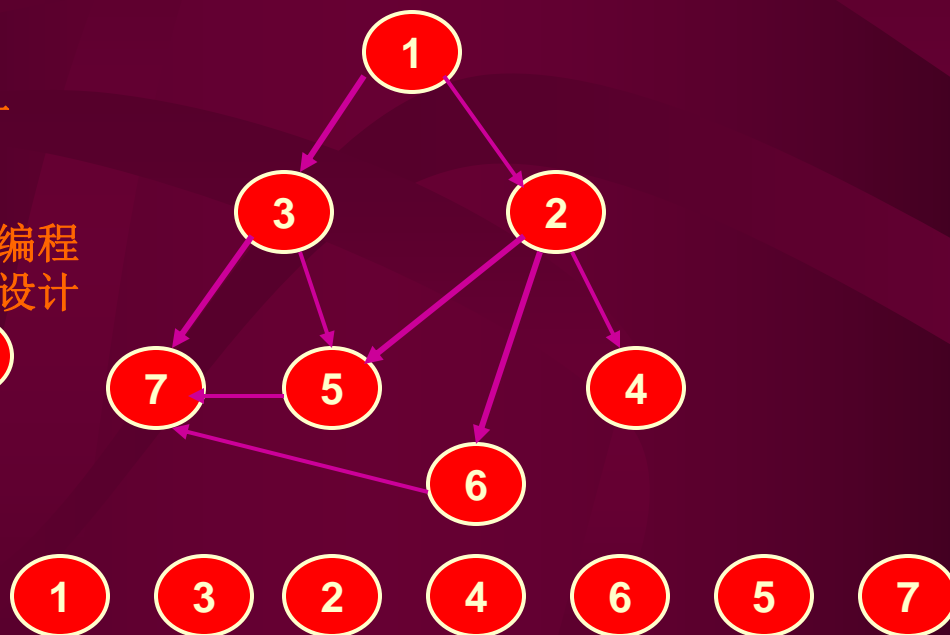
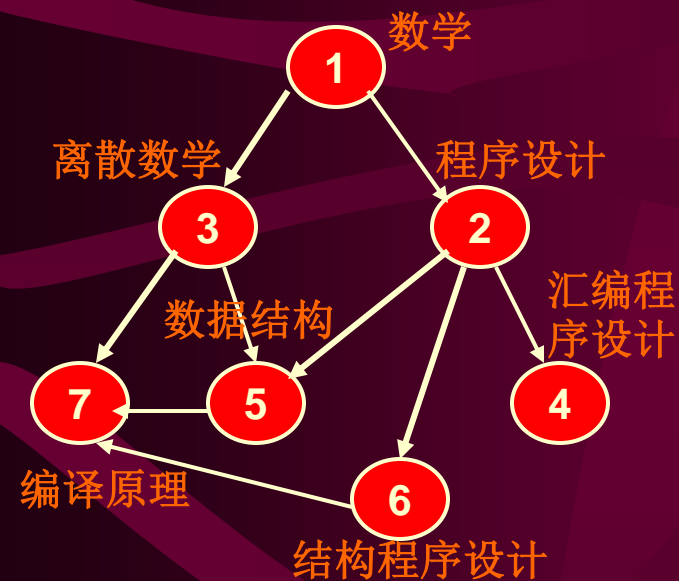
元素 3 的序号 1 (后件) < 元素 2 (前件) 的序号 3



# 活动网络

## 2、拓扑排序 (Topological Sort)

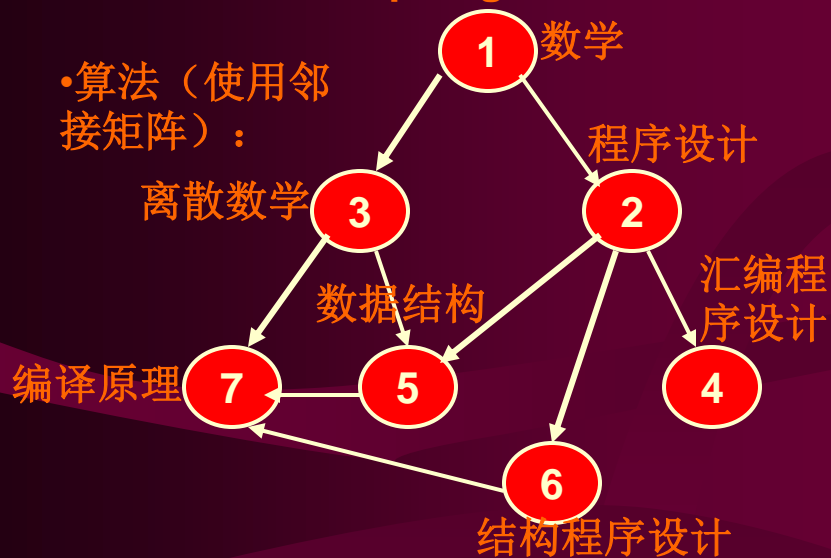
- 实例：下述集合  $M$  代表课程的集合，其中，1代表数学，2代表程序设计，3代表离散数学，4代表汇编程序设计，5代表数据结构，6代表结构程序设计，7代表编译原理。关系  $R$  课程学习的先后关系，如数学必须在离散数学之前学习。要求排一张学习的先后次序表。



# 活动网络

## 2、拓扑排序 (Topological Sort)

• 算法 (使用邻接矩阵):



算法的执行步骤:

- 1、找到全为零的第  $j$  列, 输出  $j$
- 2、将第  $j$  行的全部元素置为零
- 3、找到全为零的第  $k$  列, 输出  $k$
- 4、将第  $k$  行的全部元素置为零

.....

反复执行 3、4; 直至所有元素输出完毕。

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	0	0	0	1	1	1	0
3	0	0	0	0	1	0	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0

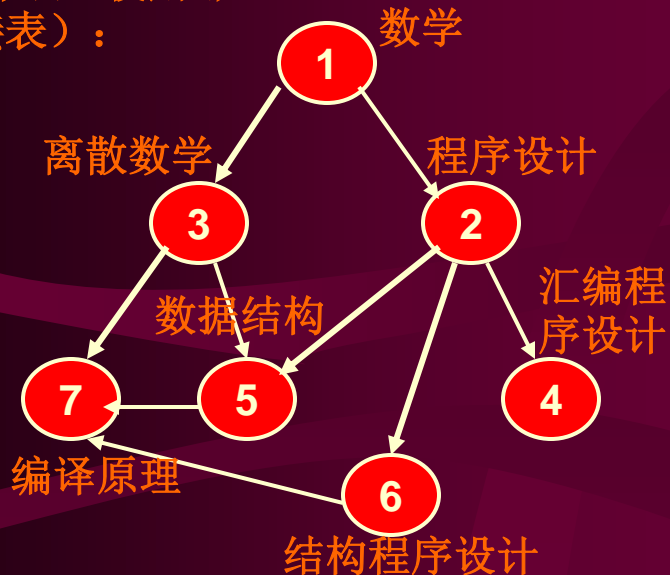
1	0	0	0	0	0	0	0
2	0	0	0	1	1	1	0
3	0	0	0	0	1	0	1
4	0	0	0	0	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	0	1
7	0	0	0	0	0	0	0



# 活动网络

## 2、拓扑排序 (Topological Sort)

• 算法 (使用邻接表):



算法的执行步骤:

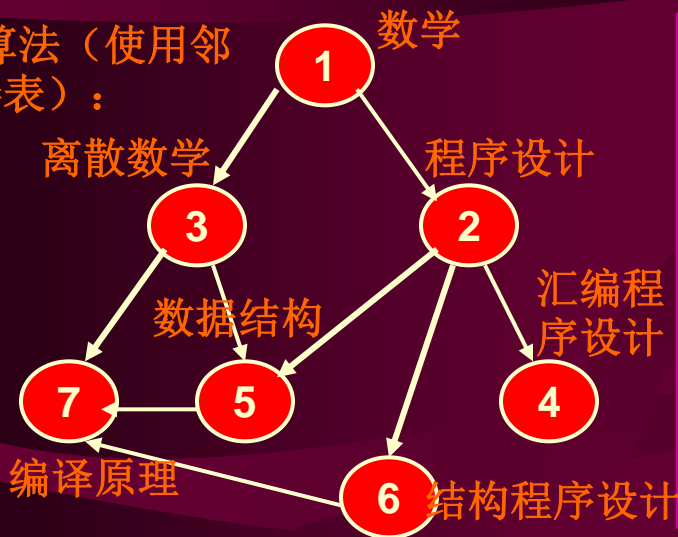
- 1、用一个数组记录每个结点的入度。将入度为零的结点进栈。
- 2、将栈中入度为零的结点V输出。
- 3、根据邻接表找到结点V的所有的邻接结点，并将这些邻接结点的入度减一。如果某一结点的入度变为零，则进栈。
- 4、反复执行 2、3；直至栈空为止。

.....

次序执行结束，如果输出结点数等于图的结点总数，则有向图无环，否则有向图有环。

# 活动网络

•算法（使用邻接表）：



算法的执行步骤：

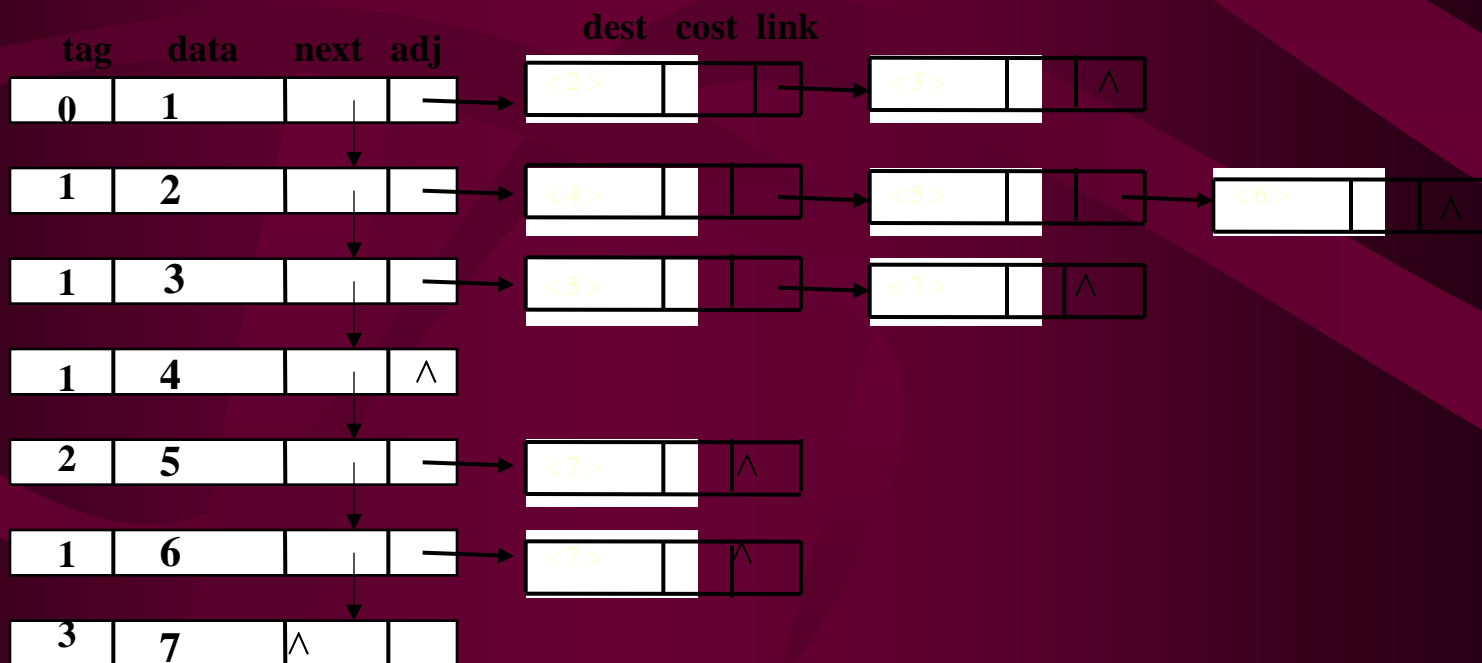
- 1、用**tag**记录结点的入度。将入度为零的结点进栈。
- 2、将栈中入度为零的结点**V**输出。
- 3、根据邻接表找到结点**V**的所有邻接结点，并将这些邻接结点的入度减一。若某结点入度变为零，则进栈
- 4、反复执行 2、3；直至栈空为止。

.....

程序执行结束，如果输出结点数等于图的结点总数，则有向图无环，否则有向图有环。

栈

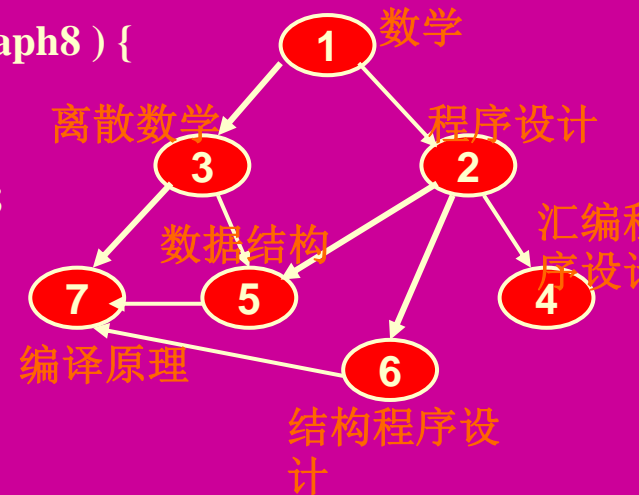
<1>



# 活动网络

2、Topological Sort使用单链表表示的结点表的邻接表) 的程序:

```
template <class VertexType, class EdgeType>
void Topological_Order ( Graph < VertexType , EdgeType > & graph8 ) {
    Vertex< VertexType,EdgeType > *p,*m;
    Stack<Vertex< VertexType,EdgeType > * > s(graph8.NumVertex);
    p = graph8.head;
    while( p ) { p->tag = 0; p = p->next; } // 每个顶点的入度数清 0。
    p = graph8.head;
    while( p ) {
        m = graph8.GetFirstNeighbor(p);
        while( m ) { m->tag++; m = graph8.GetNextNeighbor(p,m); }
        p = p->next;          // 计算每个顶点的入度数。
    }
    p = graph8.head;
    while( p ) { if ( p->tag == 0 ) s.Push(p); p = p->next;} //入度数为 0 的顶点进栈。
```

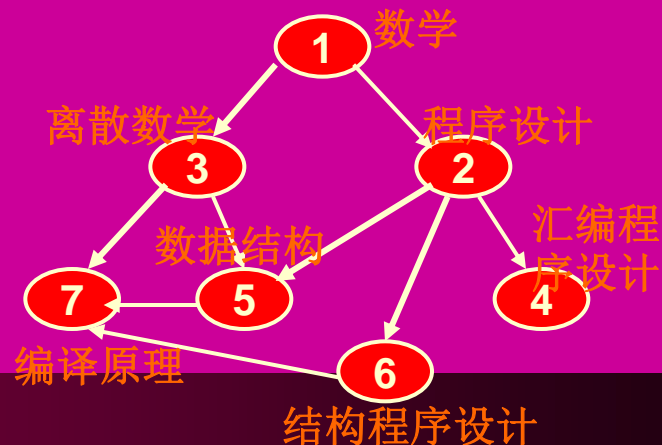
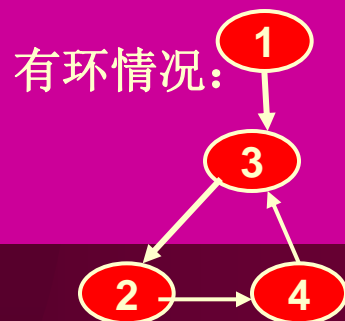


# 活动网络

## 2、Topological Sort使用单链表表示的结点表的邻接表) 的程序:

```
for ( int i = 0; i < graph8.NumVertex; i++ )  
    if ( s.IsEmpty() ) { cout << "There is a cycle in the Graph." << endl; return; }  
    else {  
        p = s.Pop();  
        cout << "Element " << p->data << " is in position" << i+1 << endl;  
        //依次输出拓扑排序序列中的第1、2、..... 个顶点。  
        m = graph8.GetFirstNeighbor(p);  
        // 得到顶点p 的第一个邻接顶点地址。  
        while( m ) {  
            // m 非空，还存在下一个邻接顶点。  
            if ( --m->tag == 0 ) s.Push( m ); // 入度为0 的顶点进栈。  
            m = graph8.GetNextNeighbor(p,m); //得到下一个邻接顶点地址。  
        }  
    }  
}
```

```
return;  
}
```



# 活动网络

## 3、关键路径: AOE 网络的应用

- 用途: 估算工程项目完成时间

**AOE网络**: 定义结点为事件, 有向边的指向表示事件的执行次序。单位是时间(时刻)。有向边定义为活动, 它的权值定义为活动进行所需要的时间。

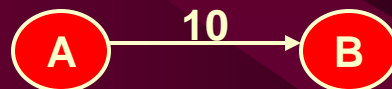
- 术语:

源点: 表示整个工程的开始点, 也称起点。

收点: 表示整个工程的结束点, 也称汇点。

事件结点: 单位时间, 表示的是时刻。

活动(有向边): 它的权值定义为活动进行所需要的时间。方向表示起始结点事件先发生, 而终止结点事件才能发生。

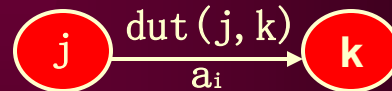


**事件的最早发生时间 ( $Ve(j)$ )**: 从起点到本结点的最长的路径。意味着事件最早能够发生的时刻。

**事件的最迟发生时间 ( $VL(j)$ )**: 不影响工程的如期完工, 本结点事件必须发生的时刻。

活动的最早开始时间:  $e(a_i) = Ve(j)$

活动的最迟开始时间:  $l(a_i) = VL(k) - dut(j, k)$



# 活动网络

## 3、关键路径

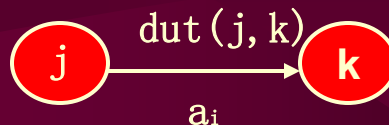
### • 术语:

事件的最早发生时间 ( $Ve(j)$ ): 从起点到本结点的最长的路径。意味着事件最早能够发生的时刻。



事件的最迟发生时间 ( $VI(j)$ ): 不影响工程的如期完工, 本结点事件必须发生的时刻。

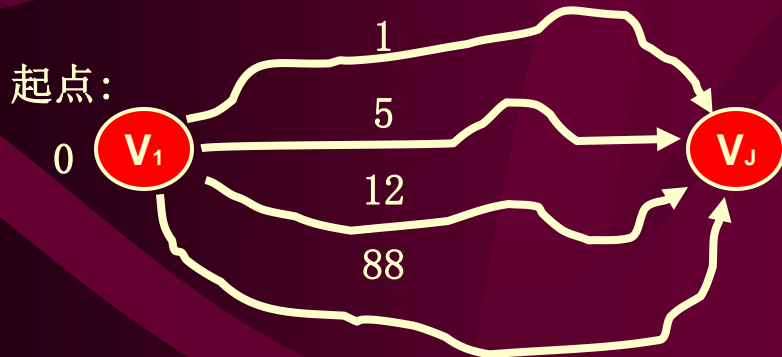
活动的最早开始时间:  $e(a_i) = Ve(j)$



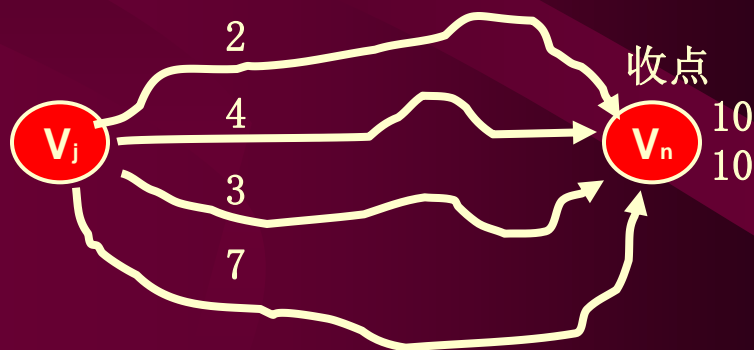
活动的最迟开始时间:  $l(a_i) = VI(k) - dut(j, k)$

关键活动: 最早开始时间 = 最迟开始时间的活动

关键路径: 从源点到收点的最长的一条路径, 或者全部由关键活动构成的路径。



$Ve(V_j) = 88$  取 1、5、12、88 的最大值 88

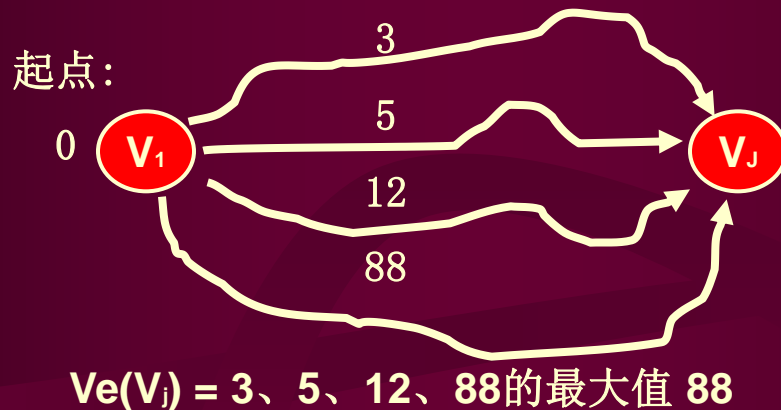


$VI(V_j) =$  取  $10-2$ 、 $10-4$ 、 $10-3$ 、 $10-7$  的最小值 3; 或  $10 -$  最长路径 7

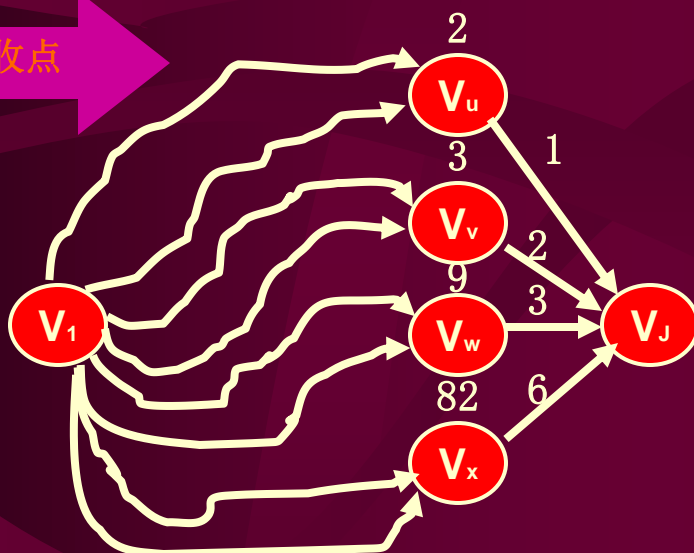
# 活动网络

## 3、关键路径

### • $Ve(j)$ 的求法:



由源点至收点



注意：本图说明由已知的所有的起始结点的  $Ve$ ，可得知终止结点的  $Ve$ 。

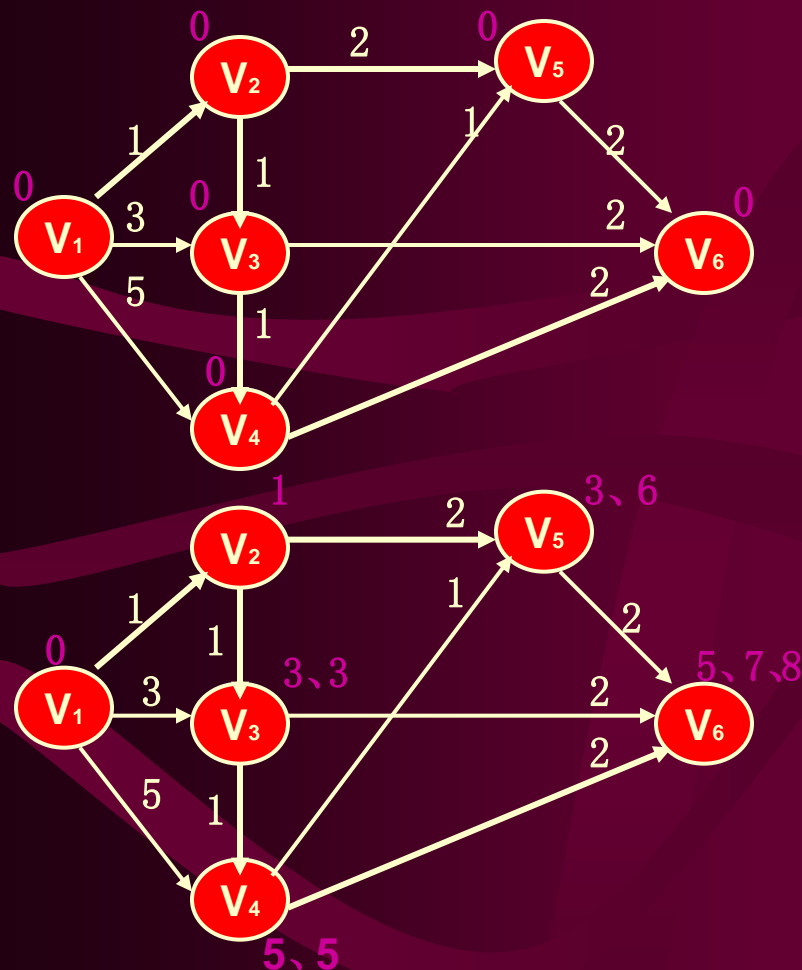
如：从  $Ve(V_u)$ 、 $Ve(V_v)$ 、 $Ve(V_w)$ 、 $Ve(V_x)$  可知  $Ve(V_j)$

注意：开始时只有源点的  $Ve = 0$  已知，利用拓扑排序的序列可得到每一个结点的  $Ve$ 。且结点的  $Ve$  确定次序和拓扑排序的序列相同。

# 活动网络

## 3、关键路径

- 求事件的最早发生时间的实例



利用拓扑排序算法求事件结点的最早发生时间的执行步骤:

- 1、设每个结点的最早发生时间为0，将入度为零的结点进栈。
- 2、将栈中入度为零的结点V取出，并压入另一栈，用于形成逆向拓扑排序的序列。
- 3、根据邻接表找到结点V的所有的邻接结点，将结点V的最早发生时间 + 活动的权值得到的和同邻接结点的原最早发生时间进行比较；如果该值大，则用该值取代原最早发生时间。另外，将这些邻接结点的入度减一。如果某一结点的入度变为零，则进栈。
- 4、反复执行 2、3；直至栈空为止。

正向拓扑排序:

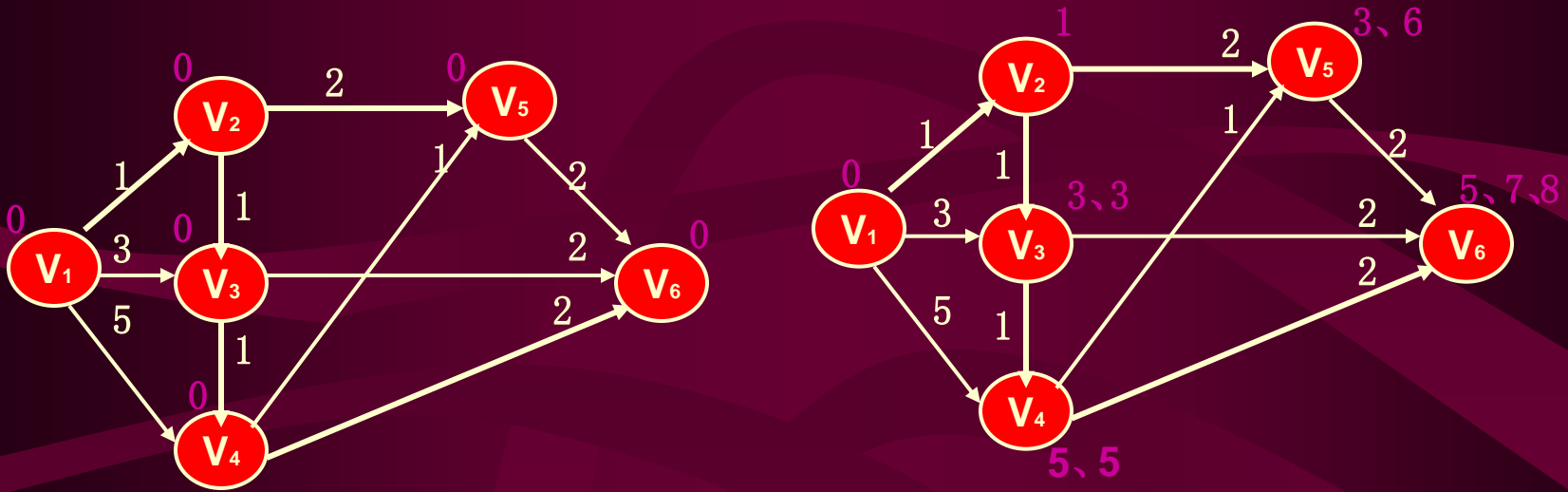




# 活动网络

## 3、关键路径

- 求事件的最早发生时间的程序实现



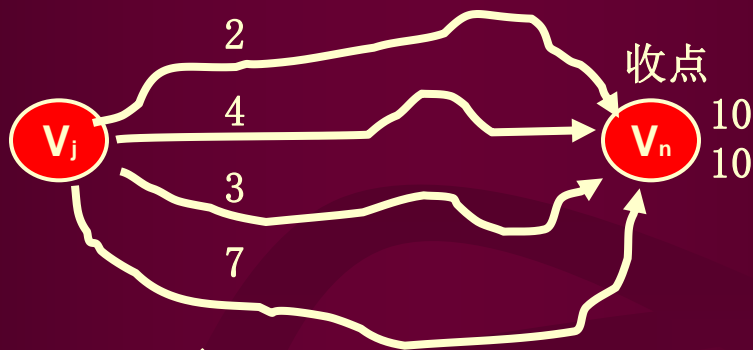
正向拓扑排序:



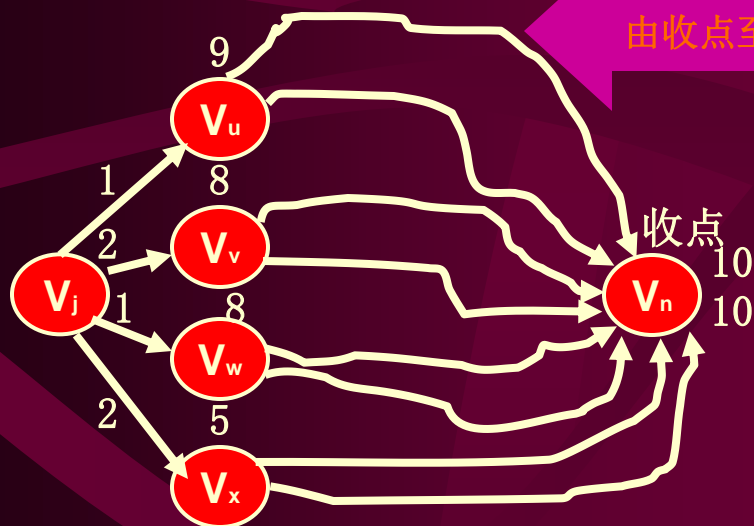
# 活动网络

## 3、关键路径

- $VI(j)$  的求法:



$VI(V_j) = \text{取 } 10-2、10-4、10-3、10-7$   
的最小值 3



$VI(V_j) = \text{取终止结点的最迟发生时间}$   
- 各自的边的权值的差的最小值 3

由收点至源点

注意：本图说明由已知的所有的终止结点的  $VI$ ，可得知起始结点的  $VI$ 。

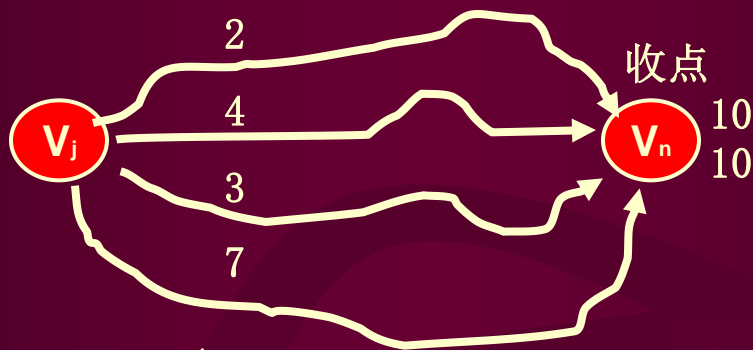
如：从  $VI(V_u)$ 、 $VI(V_v)$ 、 $VI(V_w)$ 、 $VI(V_x)$  可知  $VI(V_j)$

注意：开始时只有收点的  $VI = Ve(V_n)$  已知，利用逆拓扑排序的序列可得到每一个结点的  $VI$ 。且结点的  $VI$  确定次序和逆拓扑排序的序列相同。

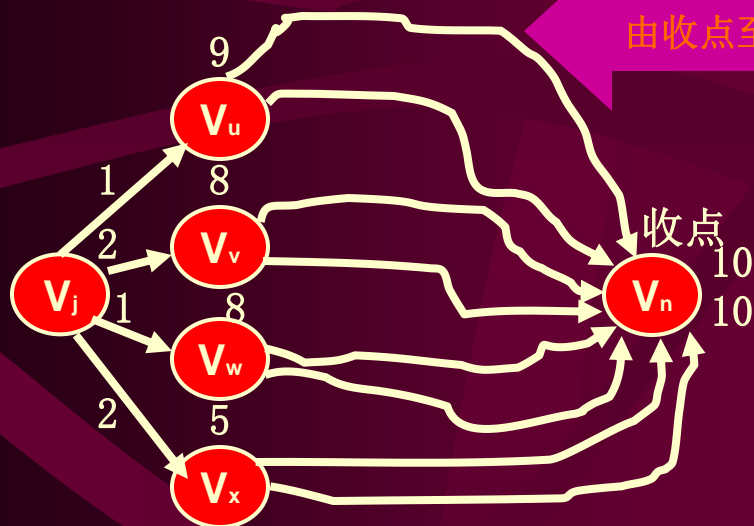
# 活动网络

## 3、关键路径

- $VI(j)$  的求法:



$VI(V_j) = \text{取 } 10-2、10-4、10-3、10-7$   
的最小值 3



$VI(V_j) = \text{取终止结点的最迟发生时间}$   
- 各自的边的权值的差的最小值 3

由收点至源点

注意：本图说明由已知的所有的终止结点的  $VI$ ，可得知起始结点的  $VI$ 。

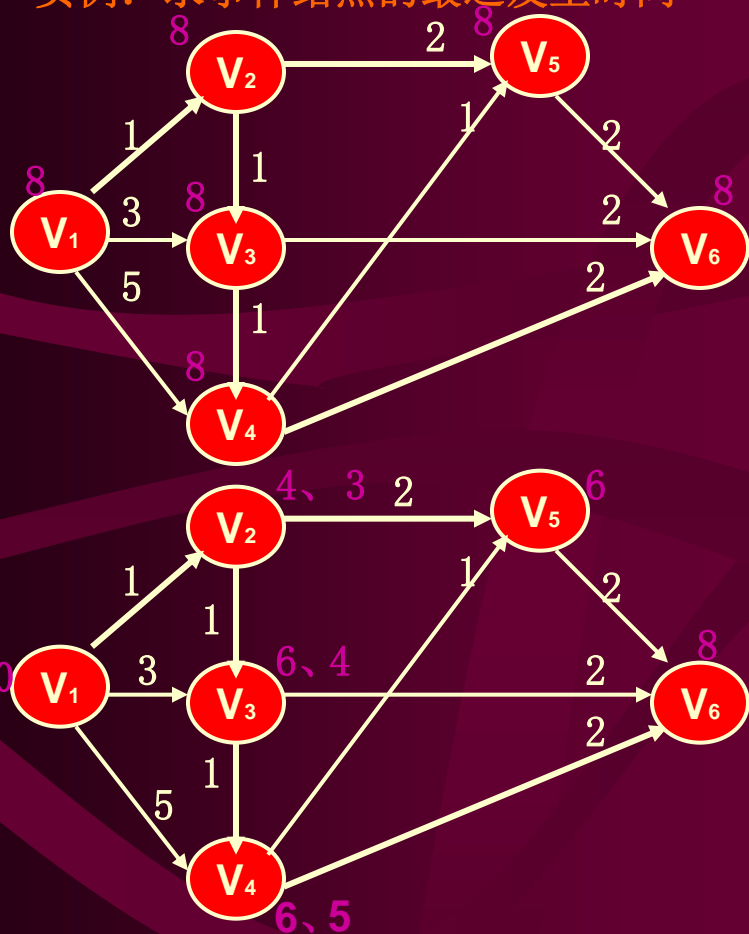
如：从  $VI(V_u)$ 、 $VI(V_v)$ 、 $VI(V_w)$ 、 $VI(V_x)$  可知  $VI(V_j)$

注意：开始时只有收点的  $VI = Ve(V_n)$  已知，利用逆拓扑排序的序列可得到每一个结点的  $VI$ 。且结点的  $VI$  确定次序和逆拓扑排序的序列相同。

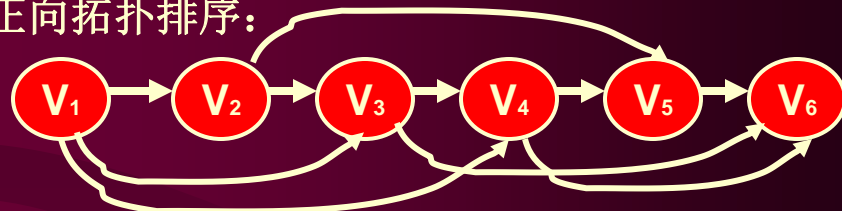
# 活动网络

## 3、关键路径

- 实例：求事件结点的最迟发生时间



正向拓扑排序：



·逆向拓扑排序同每个结点的 VL 发生时间的确定次序是一致的。

证：·最末一个结点的 VL 最先确定。它确定之后，就可推出对相应的起始结点的影响，如上图的 V<sub>5</sub> V<sub>4</sub> V<sub>3</sub> 结点的 VL 当前值，或者可以形象地认为结点 V<sub>5</sub> V<sub>4</sub> V<sub>3</sub> 的出度相应的减少了 1。

·第二个可以确定 VL 的结点是正向拓扑排序中的倒数第二个结点，如 V<sub>5</sub>。理由：此时它已相当于出度为零的结点。即在它之前已确定 VL 的结点，对它发生的影响已全部产生(反证：如果没有全部产生，就意味着 V<sub>5</sub> V<sub>6</sub> 之间还有一个结点 如 V<sub>x</sub> 还对它产生影响，边 V<sub>5</sub> V<sub>x</sub> 存在。但这样一来，它将不会是正向拓扑排序中的倒数第二个结点。证毕。)

·其余依次类推。

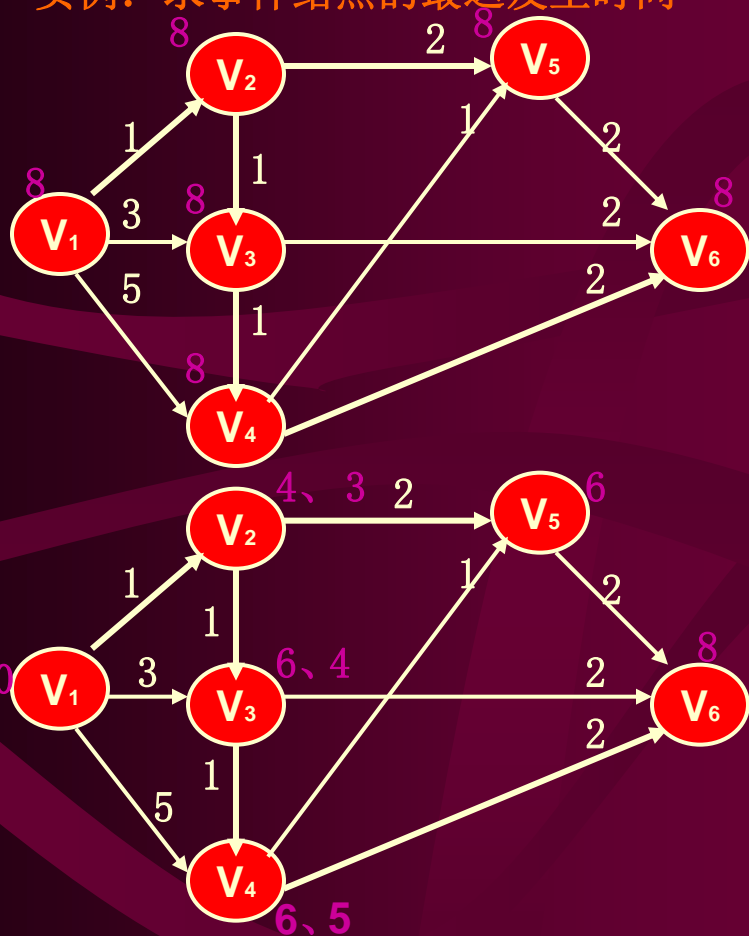
逆向拓扑排序：



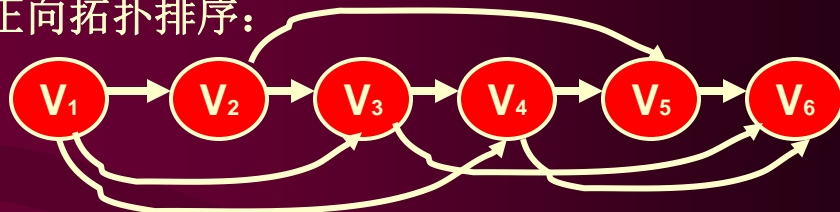
# 活动网络

## 3、关键路径

- 实例：求事件结点的最迟发生时间



正向拓扑排序：



利用逆向拓扑排序算法求事件结点的最迟发生时间的执行步骤：

- 1、设每个结点的最迟发生时间为收点的最早发生时间。
- 2、将栈中的结点V取出。
- 3、根据邻接表找到结点V的所有的到达结点W，将结点W的最迟发生时间 - 活动的权值得到的差同结点V的原最迟发生时间进行比较；如果该值小，则用该值取代结点V的原最迟发生时间。
- 4、反复执行 2、3；直至栈空为止。

逆向拓扑排序：

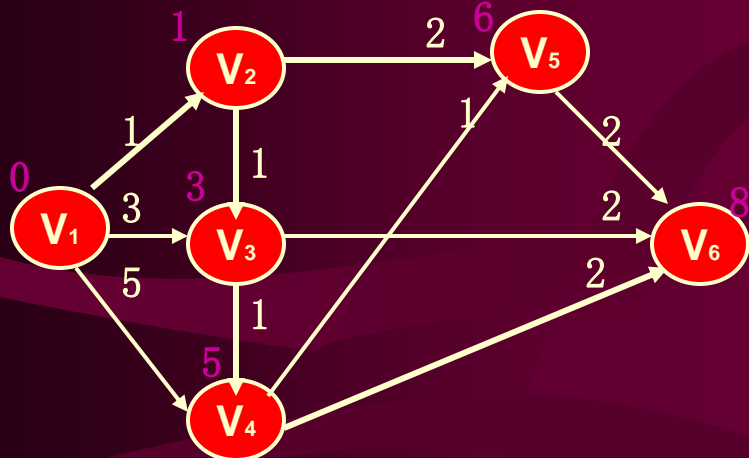


# 活动网络

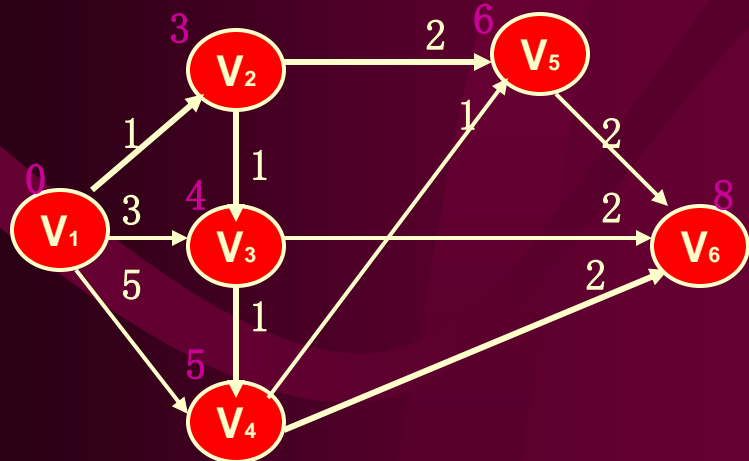
活动 $\langle v,w \rangle$ 最早发生时间为事件 $v$ 的最早发生时间，  
 活动 $\langle v,w \rangle$ 最迟发生时间为事件 $w$ 的最迟发生时间 $\langle v,w \rangle$  权值。  
 活动最早最迟时间一致时，该活动为关键活动。

## 3、关键路径

### 实例的事件结点的最早发生时间



### 实例的事件结点的最迟发生时间

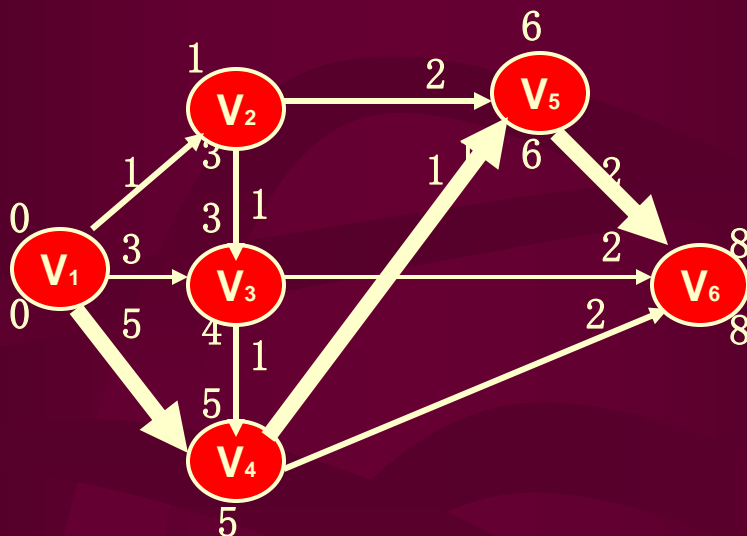


边	最早发生时间	最迟发生时间	
$V_1 \rightarrow V_2$	0	2	
$V_1 \rightarrow V_3$	0	1	
$V_1 \rightarrow V_4$	0	0	关键活动
$V_2 \rightarrow V_3$	1	3	
$V_2 \rightarrow V_5$	1	4	
$V_3 \rightarrow V_4$	3	4	
$V_3 \rightarrow V_6$	3	6	
$V_4 \rightarrow V_5$	5	5	关键活动
$V_4 \rightarrow V_6$	5	6	
$V_5 \rightarrow V_6$	6	6	关键活动

# 活动网络

## 3、关键路径

- 实例的关键路径（粗大的黑色箭头所示）



- 注意：关键路径可有多条  
缩短工期必须缩短关键活动所需的时间
- 时间代价：图中的结点和边都要访问到，故： $O(e+n)$ 。

# 总结

- 求结点之间的最短路径有很重要的应用。
- 介绍了求单源最短路径和所有结点对之间的最短路径的方法。
- 重点介绍了两个常用的算法：求单源最短路径的Dijkstra算法和求所有结点之间的最短路径的Floyd算法。



# 作业