数据结构

7 图

董洪伟 陈聪 周世兵
联系电话：13812529213
E-mail: worldguard@163.com

主要内容

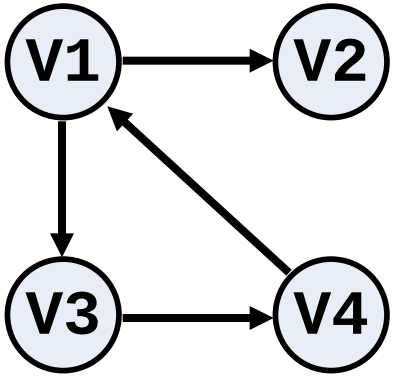
- 图的定义和术语
- 图的存储结构
  - 邻接矩阵、邻接表、十字链表、邻接多重表
- 图的遍历
  - 深度优先、广度优先
- 图的连通性问题
  - 连通分量、生成树、最小生成树
- 有向无环图的应用
  - 拓扑排序、关键路径
- 最短路径问题
  - **Dijkstra**算法、**Floyd**算法

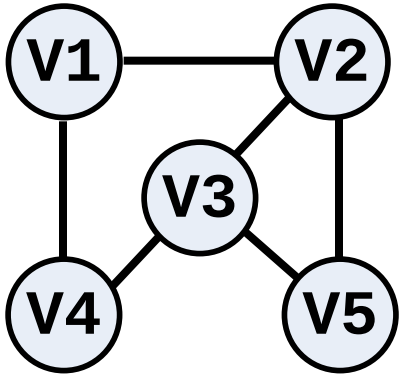
图的定义和术语

- 图定义
  - 图是由顶点集合**V(Vertex)**及顶点间的关系集合**VR**组成的一种数据结构：**Graph = (V, VR)**
  - **V = {x| x∈**某个数据对象**}**，是顶点的有穷非空集合
  - **VR**是顶点之间的关系的集合

图的定义和术语

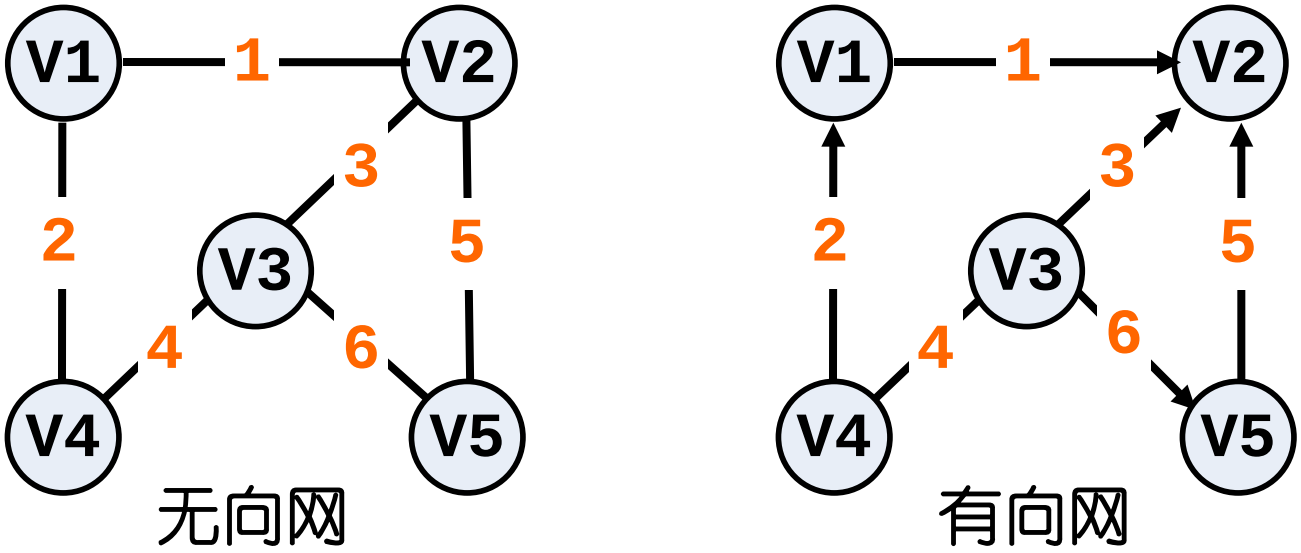
- 若<v,w>∈VR，则<v,w>表示从v到w的一条弧，且称v为弧尾或初始点，w为弧头或终端点，此时的图称之为有向图
- 若<v,w>∈VR，必有<w,v>∈VR，即VR是对称的，则以无序对(v,w)代替这两个有序对，表示v和w之间的一条边，此时的图称为无向图





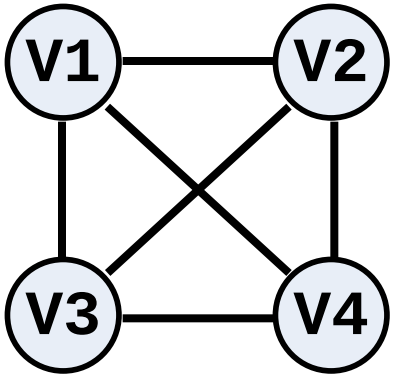
图的定义和术语

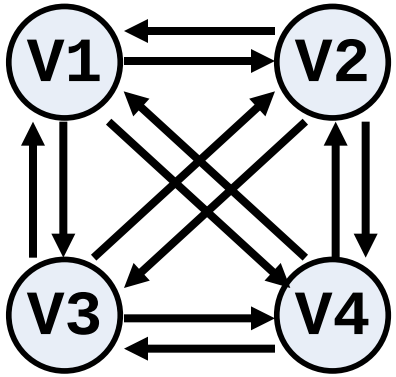
- 权
  - 与图的边（弧）相关的数值
- 网络
  - 带有权的图



图的定义和术语

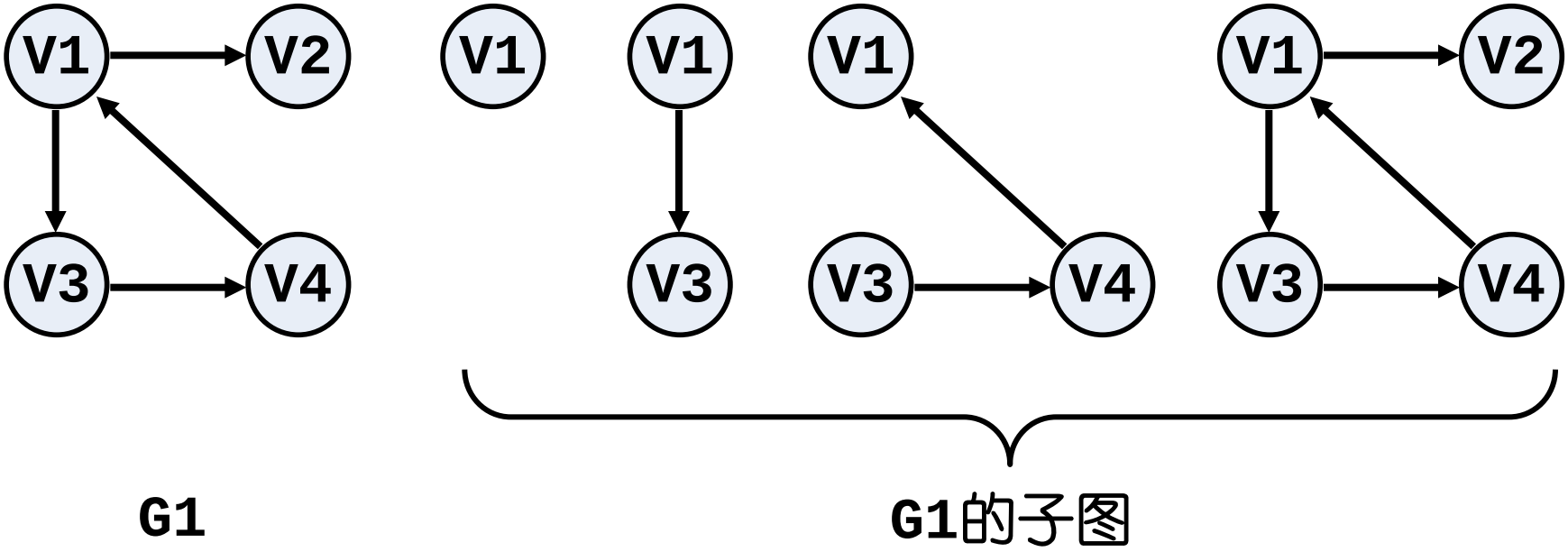
- 完全图：
  - 若有n个顶点的无向图有 $C_n^2 = n(n-1)/2$ 条边，则此图为完全无向图
  - 有n个顶点的有向图有 $n(n-1)$ 条弧，则此图为完全有向图
  - 完全图其实就是边/弧的数量达到最大值

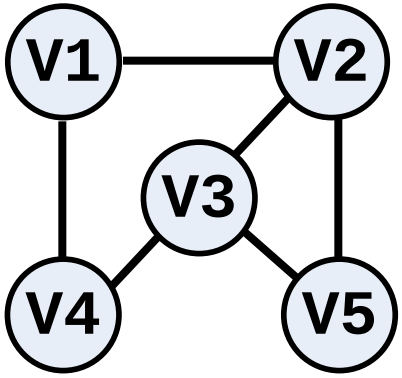




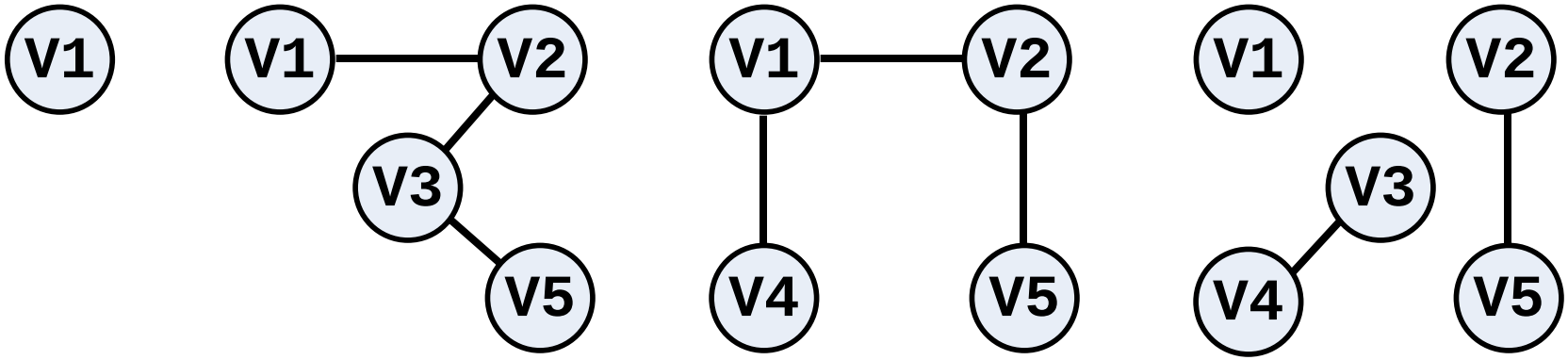
图的定义和术语

- 子图
  - 有两个图G={V,{E}},G′={V′,{E′}}
  - 如果V′⊆V，E′⊆E，则称G′为G的子图





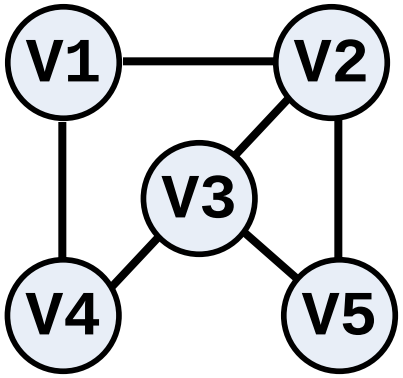
G2



G2的子图

图的定义和术语

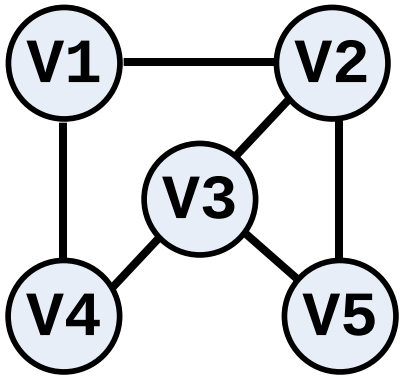
- 邻接点
  - 对于无向图G={V,{E}}，若边(v,v′)∈E，则称v和v′互为邻接点
  - 称边(v,v′)依附于顶点v和v′
  - 或者说边(v,v′)和顶点v，v′相关联



- V1，v2互为邻接点
- 边(v1,v2)依附于顶点v1和v2
- 边(v1,v2)和顶点v1，v2相关联

图的定义和术语

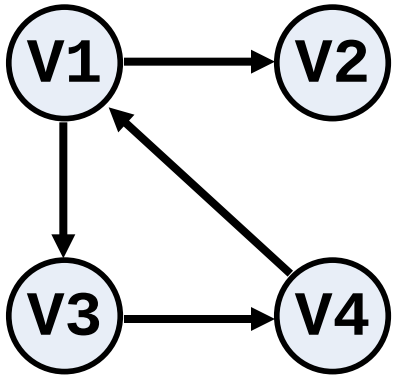
- 度
  - 顶点v的度TD(V)=和v相关联的边的数目



- TD(V1)=2
- TD(V2)=3
- TD(V3)=3
- TD(V4)=2
- TD(V5)=2

图的定义和术语

- 入度和出度
  - 对于有向图G={V,{A}}:
    - v的入度ID(v) = 以顶点v为头的弧的数目
    - V的出度OD(v) = 以顶点v为尾的弧的数目
    - 有向图中，顶点的度 = 入度 + 出度



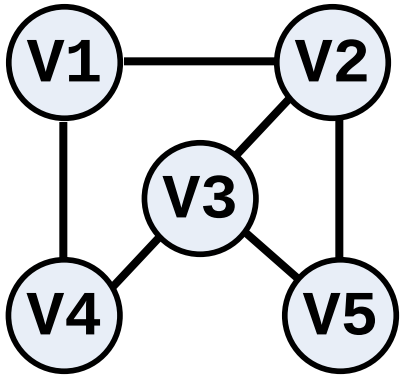
- ID(V1)=1
- OD(V1)=2
- TD(V1)=ID(v1)+OD(v1)=3

图的定义和术语

- 一个有n个顶点，e条边或弧的图满足：

$$e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$$

- 即边（或弧）的总数 = 各个顶点的度的总数的一半

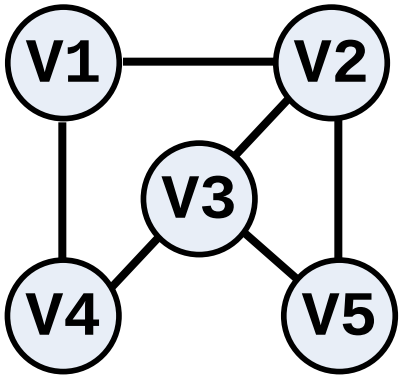


- TD(V1)=2
- TD(V2)=3
- TD(V3)=3
- TD(V4)=2
- TD(V5)=2

- e=6

图的定义和术语

- 路径
  - 在无向图G=(V,E)中，若从顶点v出发，沿一些边经过一些顶点$v_{i,0}$，$v_{i,1}$，...，$v_{i,m}$，到达顶点v′。则称顶点序列(v,$v_{i,0}$,$v_{i,1}$,...,$v_{i,m}$,v′)为从顶点v 到顶点v′的路径
  - 其中($v_{i,j-1}$，$v_{i,j}$)∈E
  - 如果G是有向图，则路径也是有向的



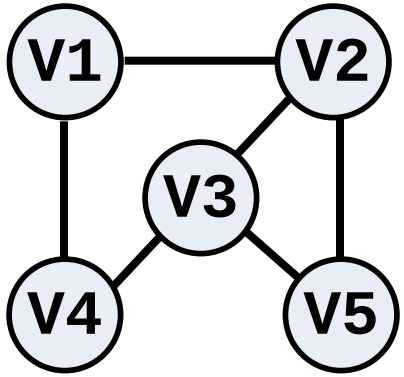
比如v1到v5的路径有：
(v1,v2,v5)
(v1,v2,v3,v5)
...

图的定义和术语

- 路径的长度
  - 路径上的边或弧的数目
- 回路（环）
  - 第一个顶点和最后一个顶点相同的路径
- 简单路径
  - 序列中顶点不重复出现的路径
    - 即不含回路的路径

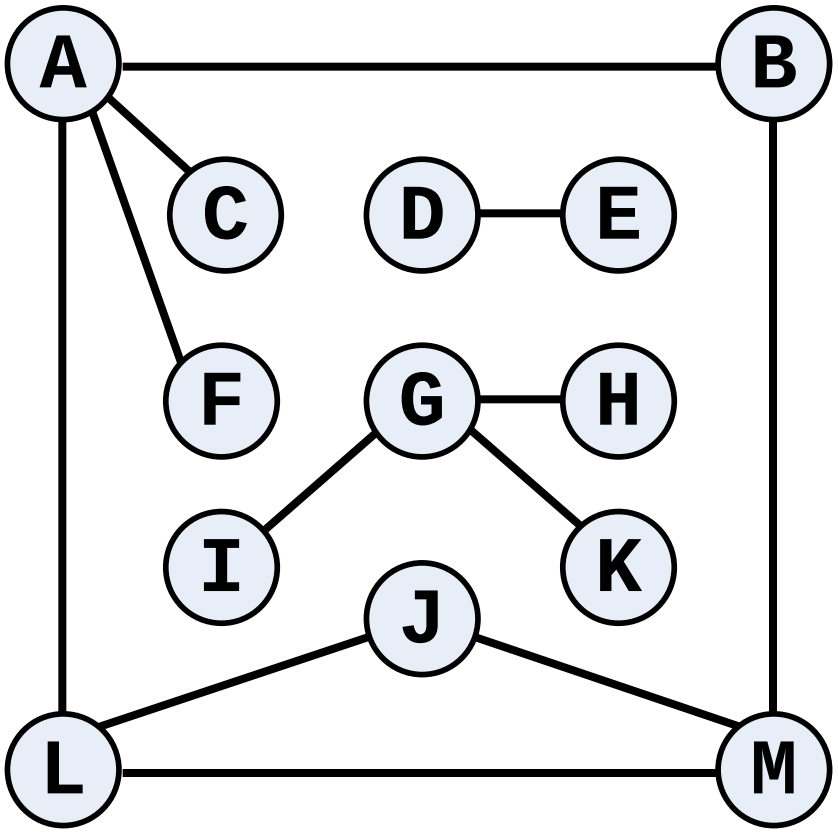
图的定义和术语

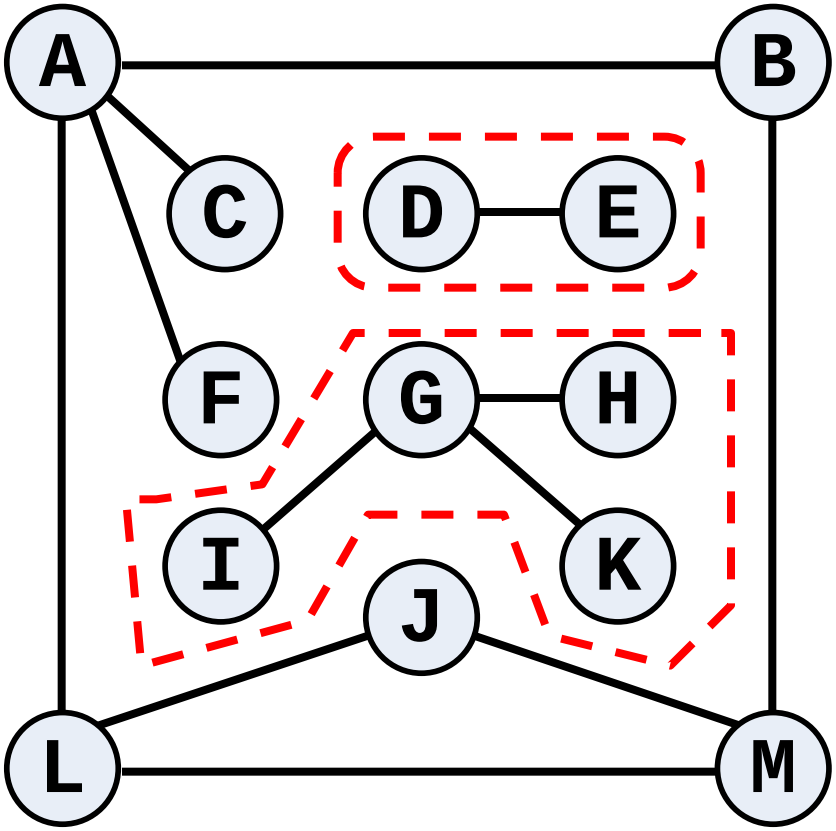
- 连通图
  - 在无向图G中，如果从顶点v到顶点v′有路径，则称v和v′是连通的
  - 如果对于图中的任意两个顶点vi和vj都是连通的，则称G是连通图
  - 是否连通是对无向图来说的

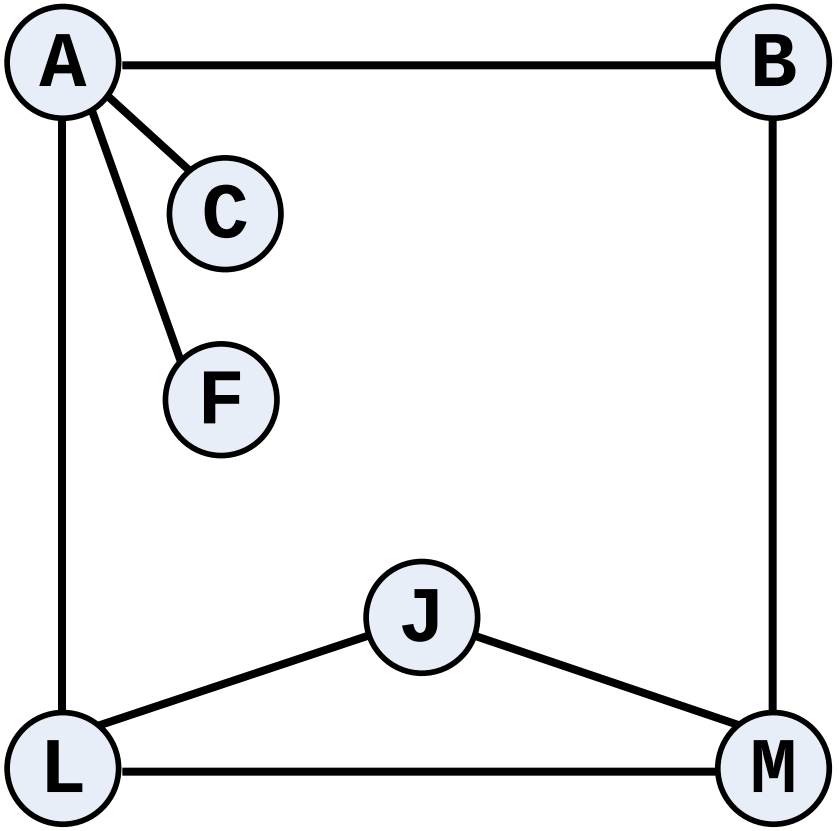


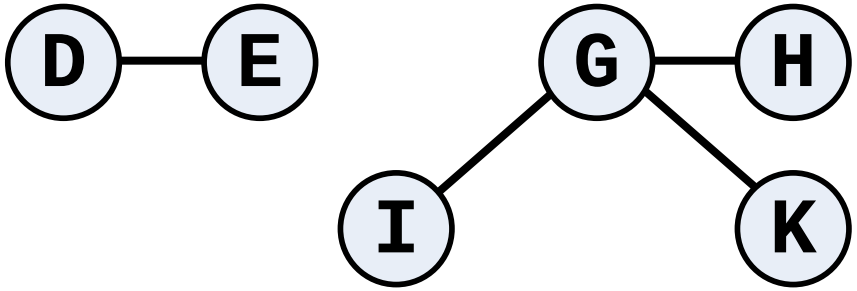
图的定义和术语

- 连通分量
  - 无向图中的极大连通子图



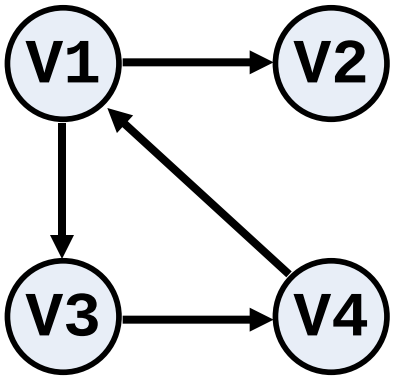

A
B
C
D
E
F
G
H
I
J
K
L
M


A
B
C
F
J
L
M


D
E
G
H
I
K

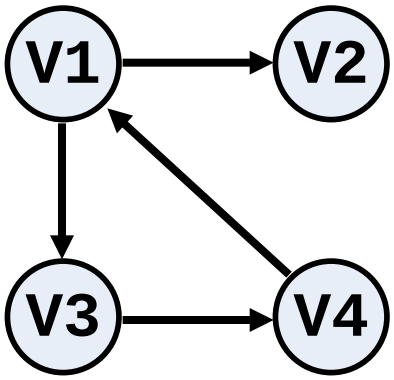
图的定义和术语

- 强连通图
  - 在有向图G中，如果每一对vi,vj，从vi到vj和从vj到vi都存在路径，则称G为强连通图
  - 是否强连通是对有向图来说的

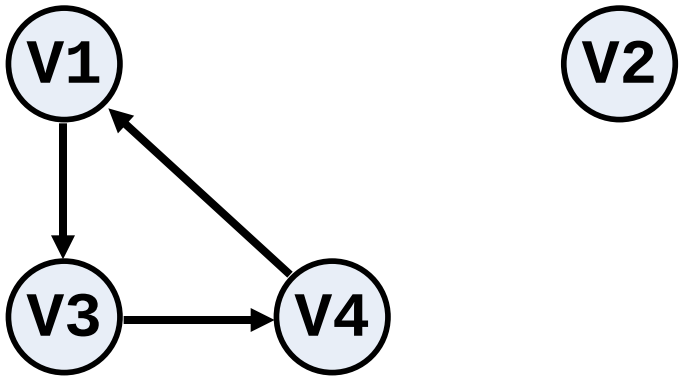


图的定义和术语

- 强连通分量
  - 有向图中的极大强连通子图



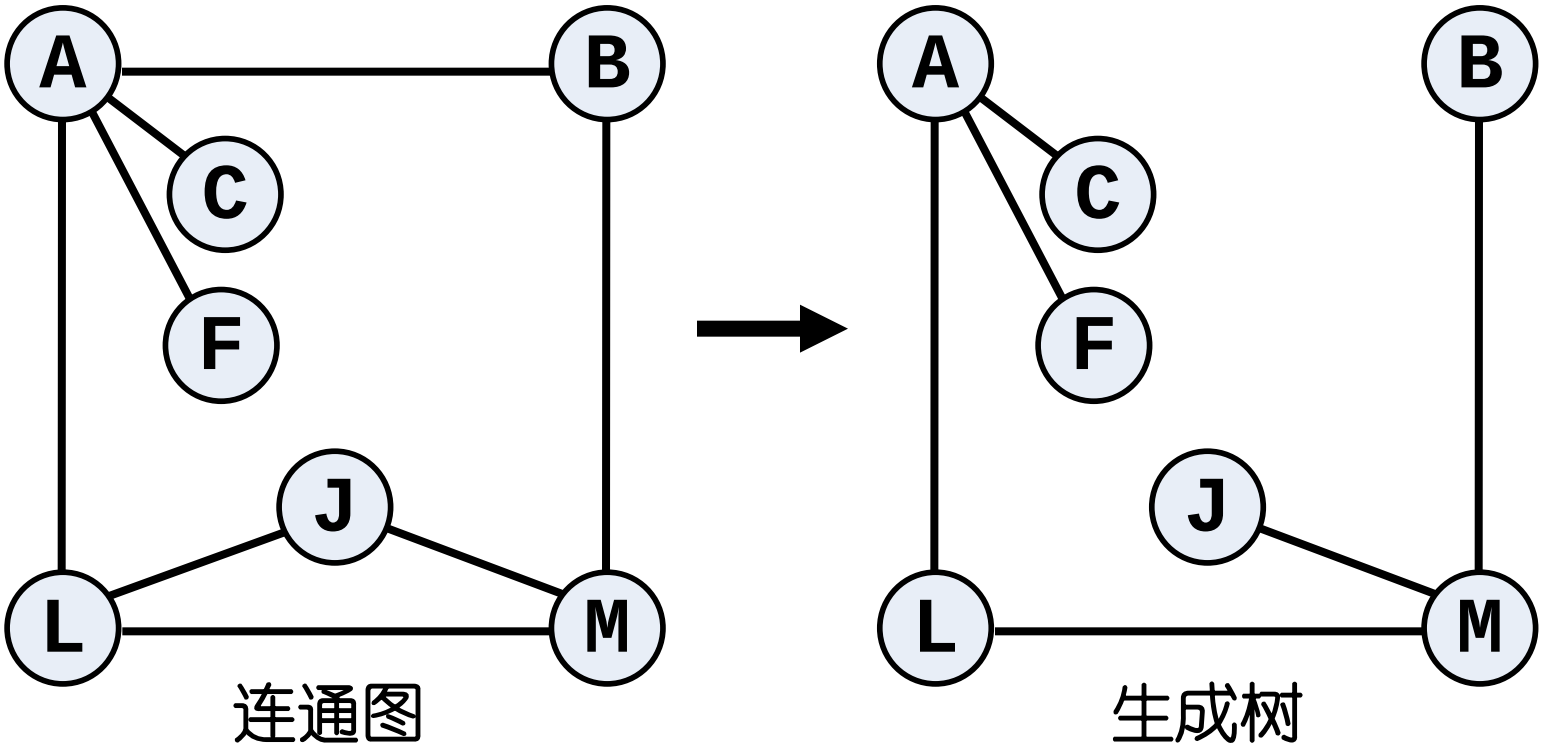
G1不是强连通图



但有两个强连通分量

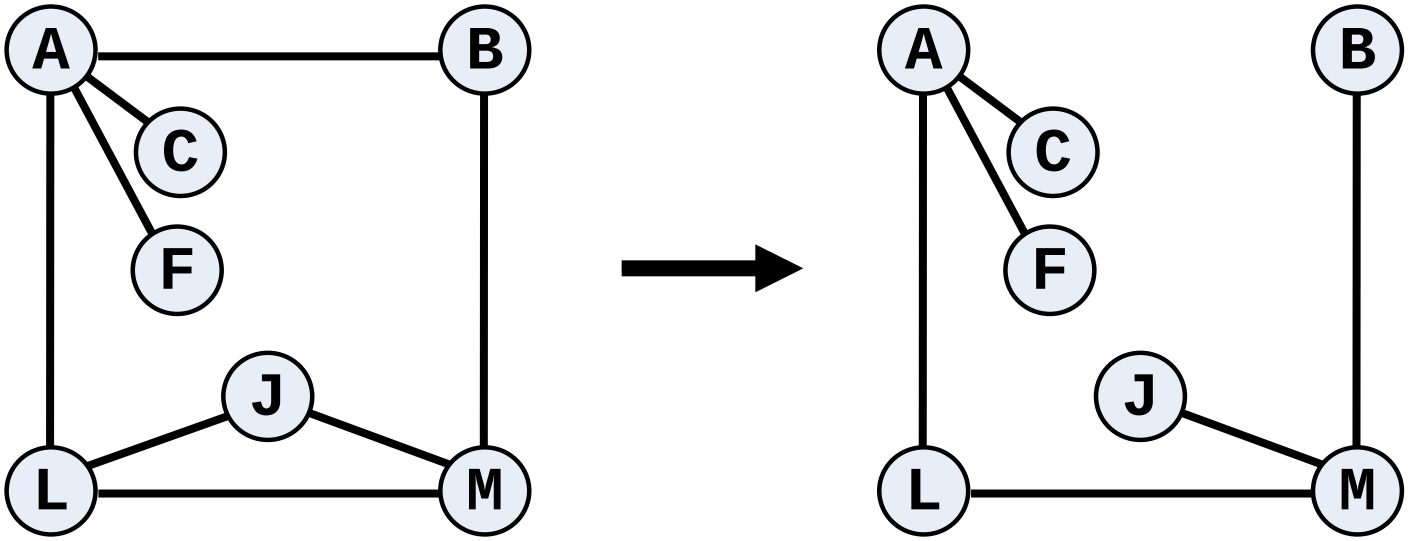
图的定义和术语

- 生成树
  - 一个连通图的包含所有顶点的极小连通子图



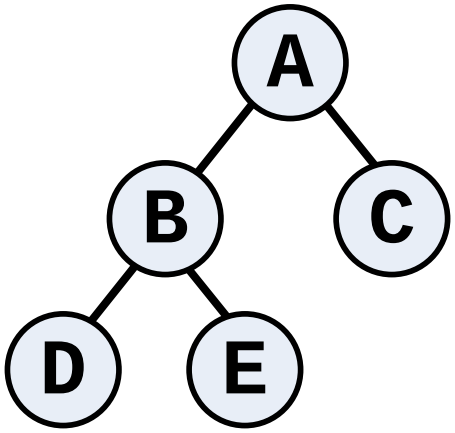
图的定义和术语

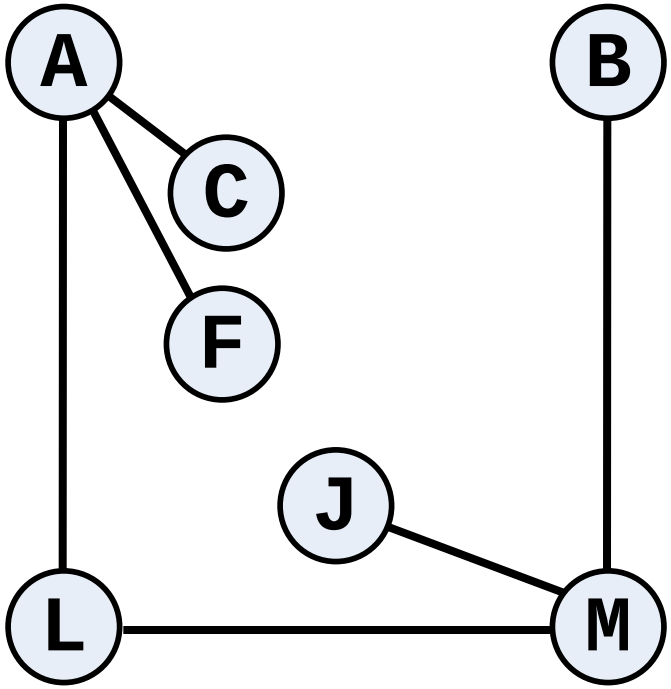
- 理解
  - “包含所有顶点”：顶点一个都不能少
  - “极小”：那只能让边尽量少了
  - “连通”：但是又不能太少



图的定义和术语

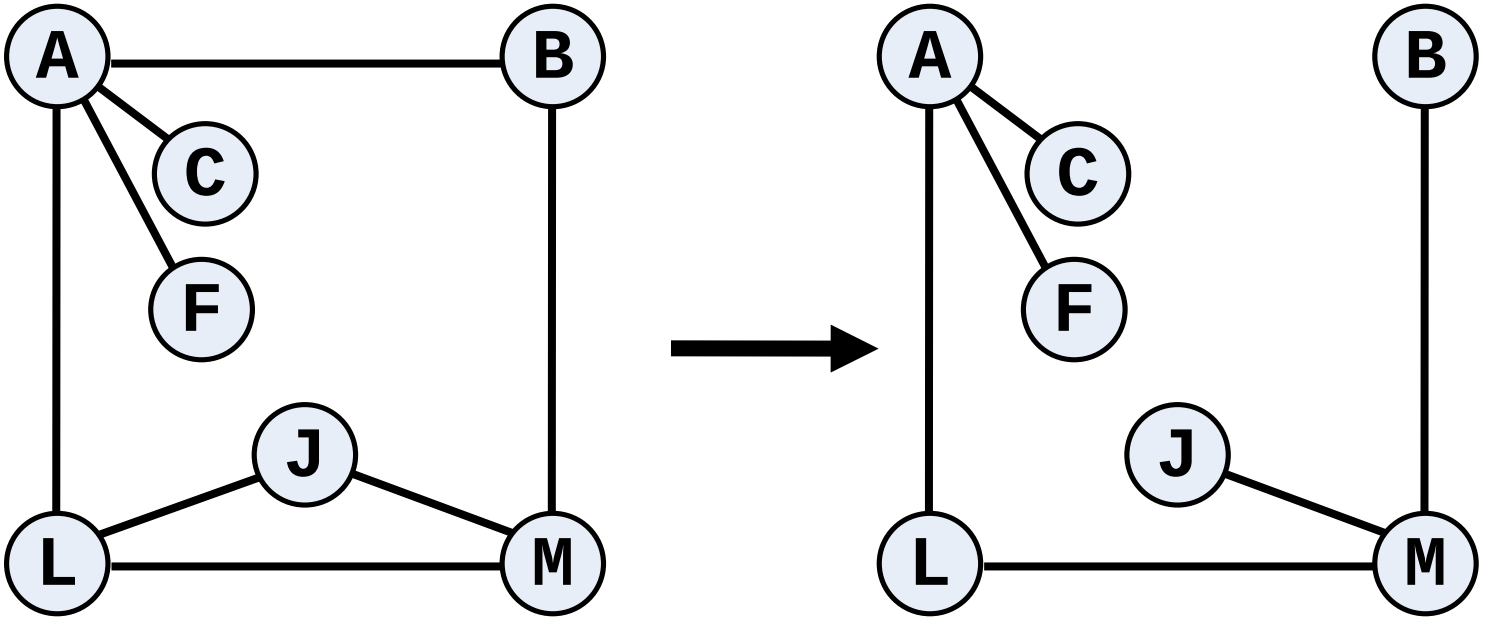
- “生成树若有n个顶点，则定有n-1条边”
  - 凡是树皆如此
  - 多一边则必定形成环
  - 少一边则必定不连通





图的定义和术语

- 思考
  - n-1条边就一定是生成树么？
  - 生成树唯一么？

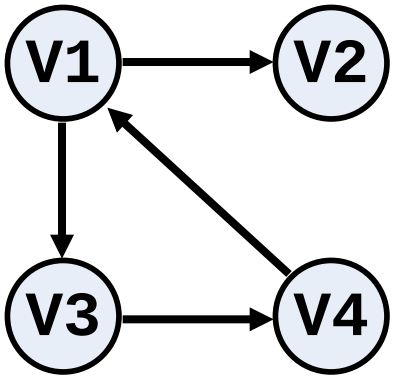


图的定义和术语

- 本节小结
  - 概念很多要记清楚
- 思考题1
  - 习题集7.1

图的存储结构：邻接矩阵

- 数组表示法（邻接矩阵）
  - 顶点表：记录各个顶点的信息
  - 邻接矩阵：表示各个顶点之间的关系

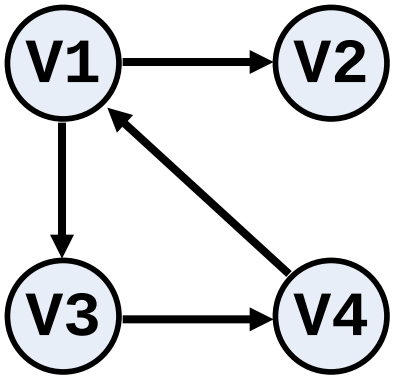


	v1	v2	v3	v4
v1	0	1	1	0
v2	0	0	0	0
v3	0	0	0	1
v4	1	0	0	0

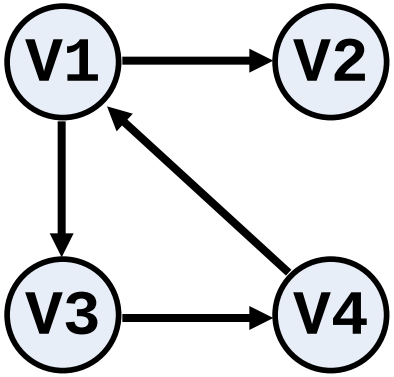
图的存储结构：邻接矩阵

- 邻接矩阵的结构
  - 设图A =(V,E)是一个有n个顶点的图，则图的邻接矩阵是一个二维数组：

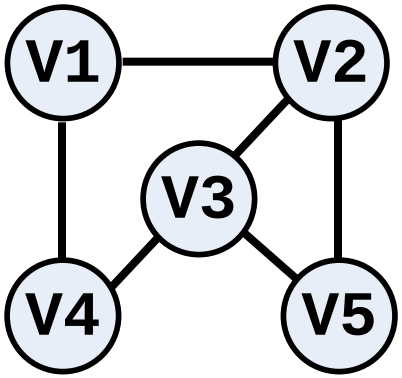
$$G.arcs[i][j]=\begin{cases}1, & \text{if } <i,j>\in E \text{ or } (i,j)\in E\\ 0, & \text{otherwise}\end{cases}$$



$$\begin{array}{c} \\ v1 \\ v2 \\ v3 \\ v4 \end{array}\begin{array}{c} v1\ v2\ v3\ v4 \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$



$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$



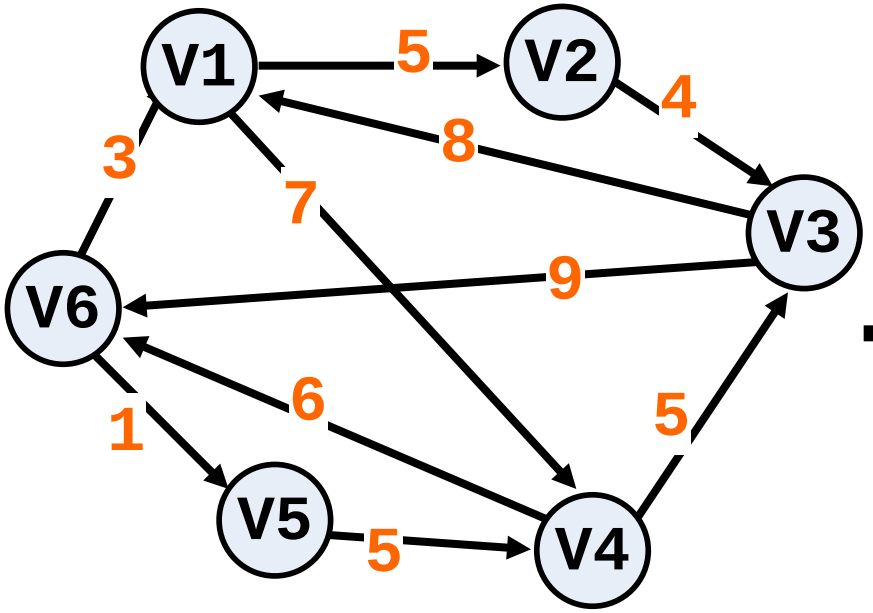
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

- 无向图的邻接矩阵是对称的

图的存储结构：邻接矩阵

- 网络的邻接矩阵

$$G.arcs[i][j] = \begin{cases} w_{i,j}, & \text{if } \langle i,j\rangle \in E \text{ or } (i,j) \in E \\ \infty, & \text{otherwise} \end{cases}$$



$$\begin{pmatrix} \infty & 5 & \infty & 7 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & 5 & \infty & \infty & 6 \\ \infty & \infty & \infty & 5 & \infty & \infty \\ 3 & \infty & \infty & \infty & 1 & \infty \end{pmatrix}$$

图的存储结构：邻接矩阵

- 邻接矩阵的操作
  - 要找出某个顶点i的所有邻接顶点，需要搜索矩阵的第i行（或第i列）
  - 在有向图中，统计第*i*行中“1”的个数可得顶点*i*的出度，统计第*j*列中“1”的个数可得顶点*j*的入度
  - 在无向图中，统计第*i*行(列)“1”的个数可得顶点*i*的度
  - 判断两个顶点(Vi,Vj)是否有边/弧，只需查看矩阵元素G(i,j)是否为“1”

图的存储结构：邻接矩阵

- 邻接矩阵的类型定义

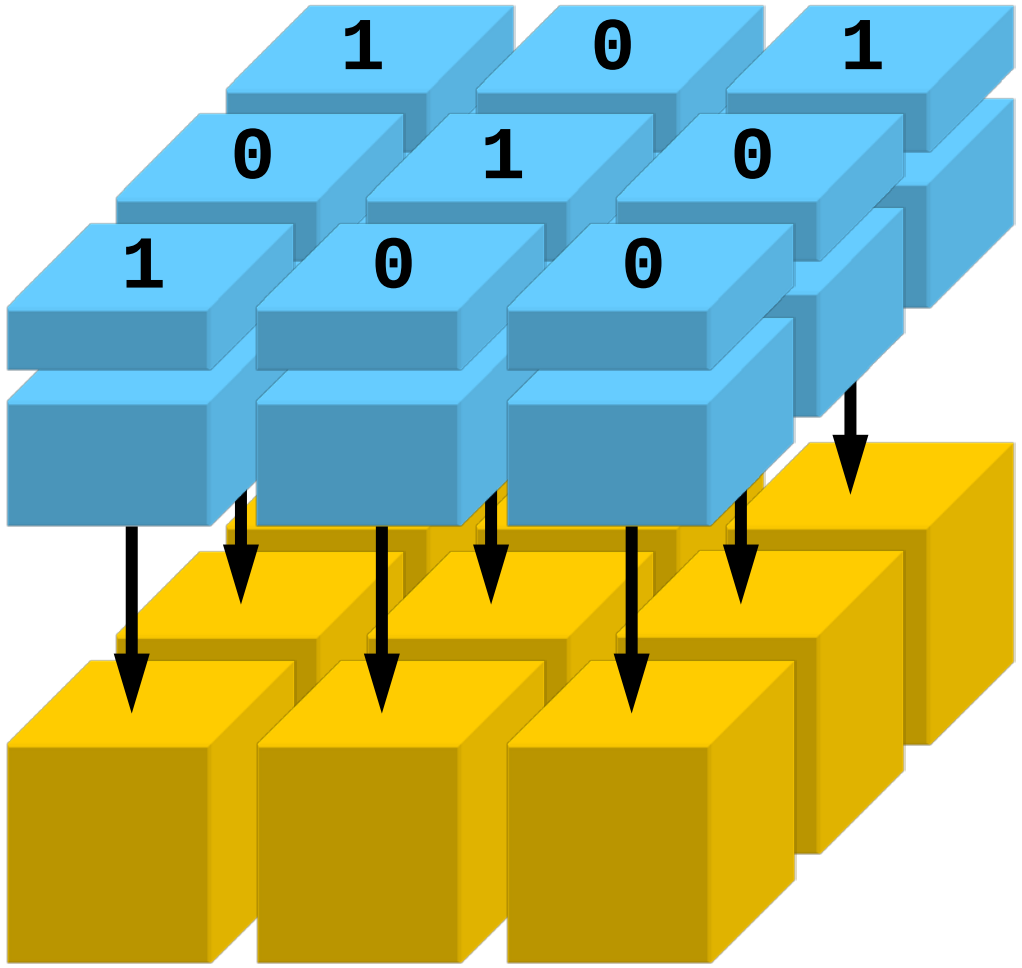
```
//最大的int
#define INFINITY          INT_MAX

//最大顶点数
#define MAX_VERTEX_NUM   20

//图的类型：有向图、有向网、无向图、无向网
typedef enum{DG, DN, UDG, UDN}
                                  GraphKind;
```

图的存储结构：邻接矩阵

```
//一条边的结构
typedef struct ArcCell
{
    //对于无权图，用1或0代表是否相邻
    //对于带权图，表示权值
    VertexType adj;
    //指向该弧相关信息
    InfoType   *info;
}ArcCell,
 AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
```

邻接矩阵AdjMatrix[3][3]

```
Struct ArcCell{
   adj;
   *info;
};
```

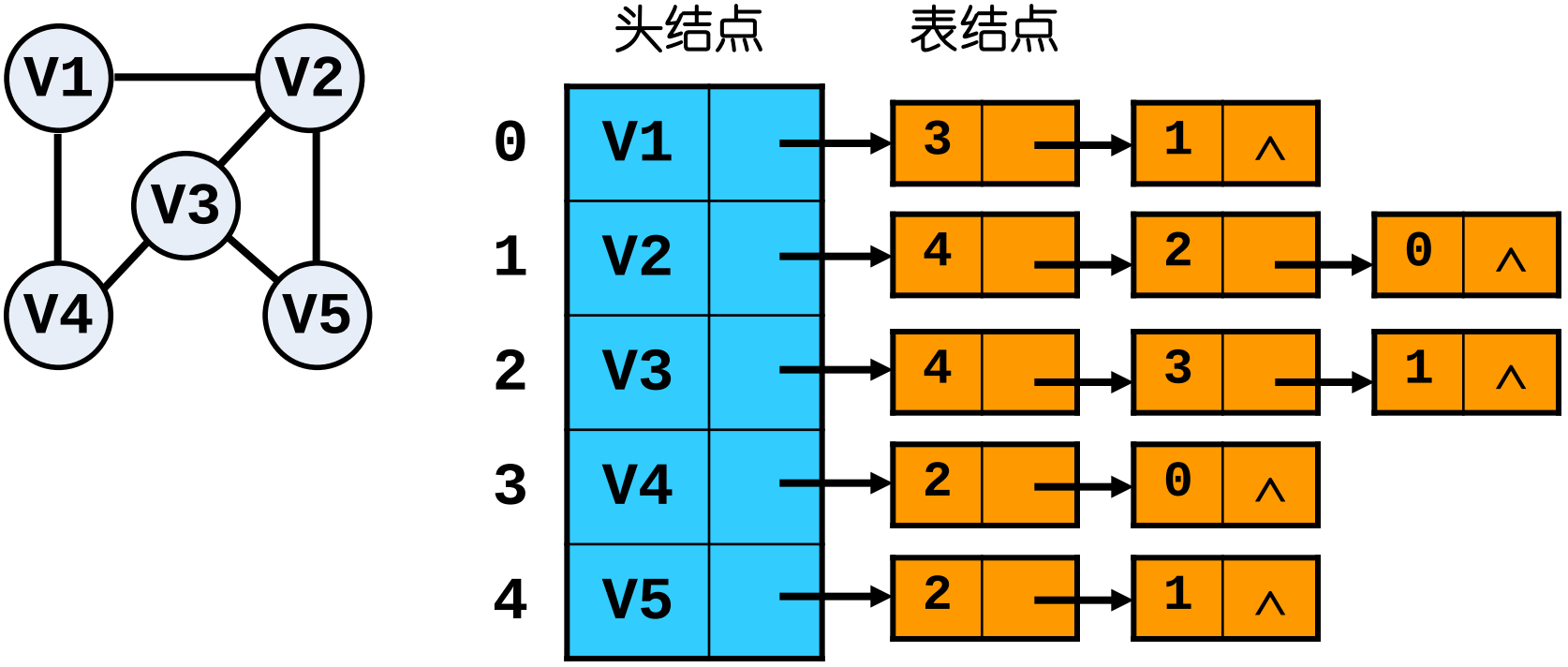


图的存储结构：邻接矩阵

```
typedef struct                          //整张图的结构
{
    //顶点数组
    VertexType vexs[MAX_VERTEX_NUM];
    AdjMatrix  arcs;                    //邻接矩阵
    int        vexnum;                  //顶点数
    int        arcnum;                  //弧数
    GraphKind  kind;                    //图的类型
}MGraph;
```

图的存储结构：邻接表

- 无向图的邻接表



– 表结点：跟当前头结点相连的另一个顶点

图的存储结构：邻接表

- 邻接表的类型定义

```
#define MAX_VERTEX_NUM 20 //最多顶点个数
//表结点结构
typedef struct ArcNode
{
    int       adjvex;   //顶点下标
    ArcNode   *nextarc; //指向下一个表结点
    InfoType *info;     //指向结点信息
}ArcNode;
```

图的存储结构：邻接表

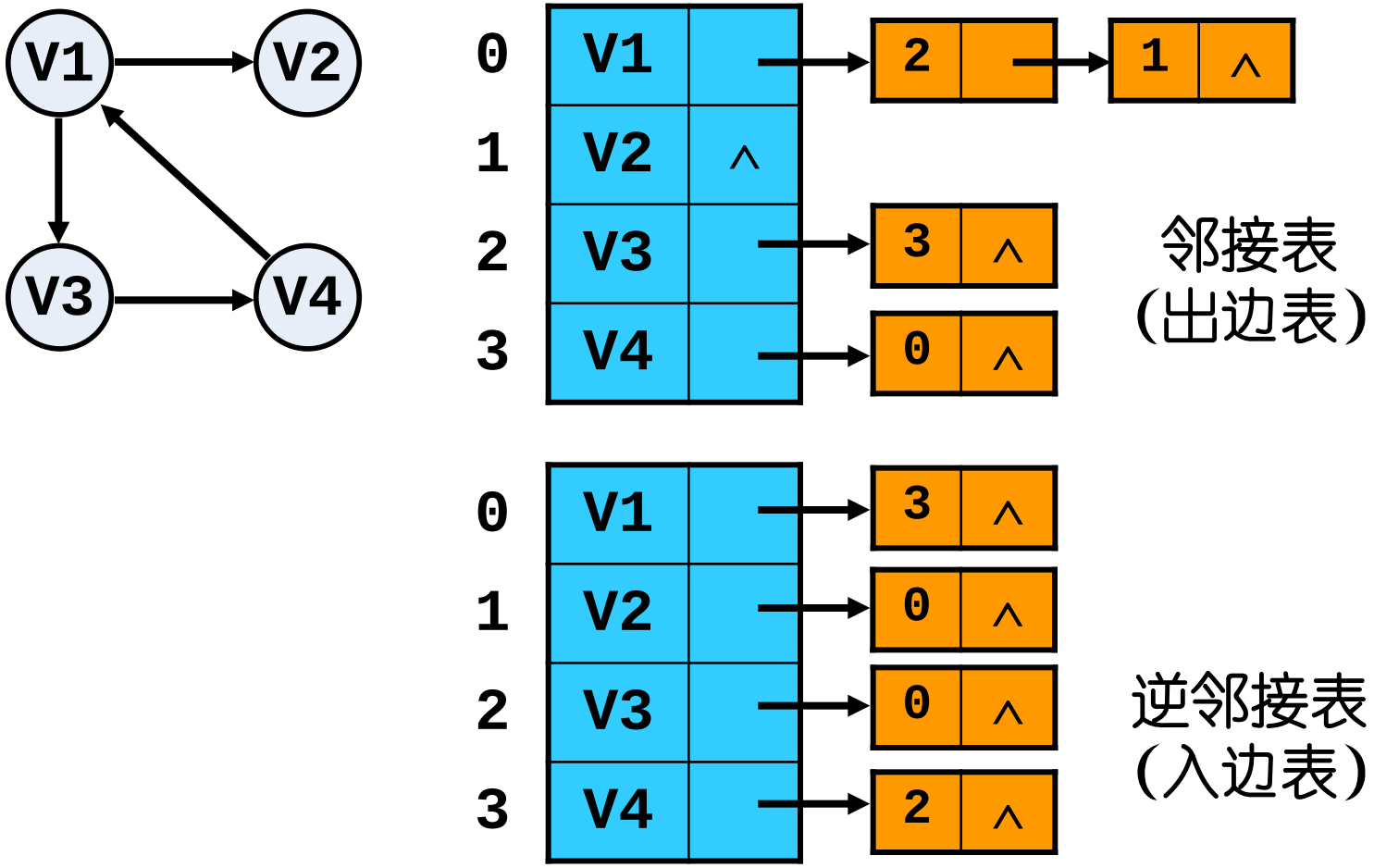
```
//头结点结构
typedef struct VNode
{
    //顶点的数据
    VertexType data;
    //指向第一个表结点
    ArcNode    *firstarc;
}VNode, AdjList[MAX_VERTEX_NUM];
```

图的存储结构：邻接表

```
//邻接表的结构
typedef struct
{
    AdjList vertices; //表结点数组
    int     vexnum;   //顶点个数
    int     arcnum;   //边的条数
    int     kind;     //图的类型
}ALGraph;
```

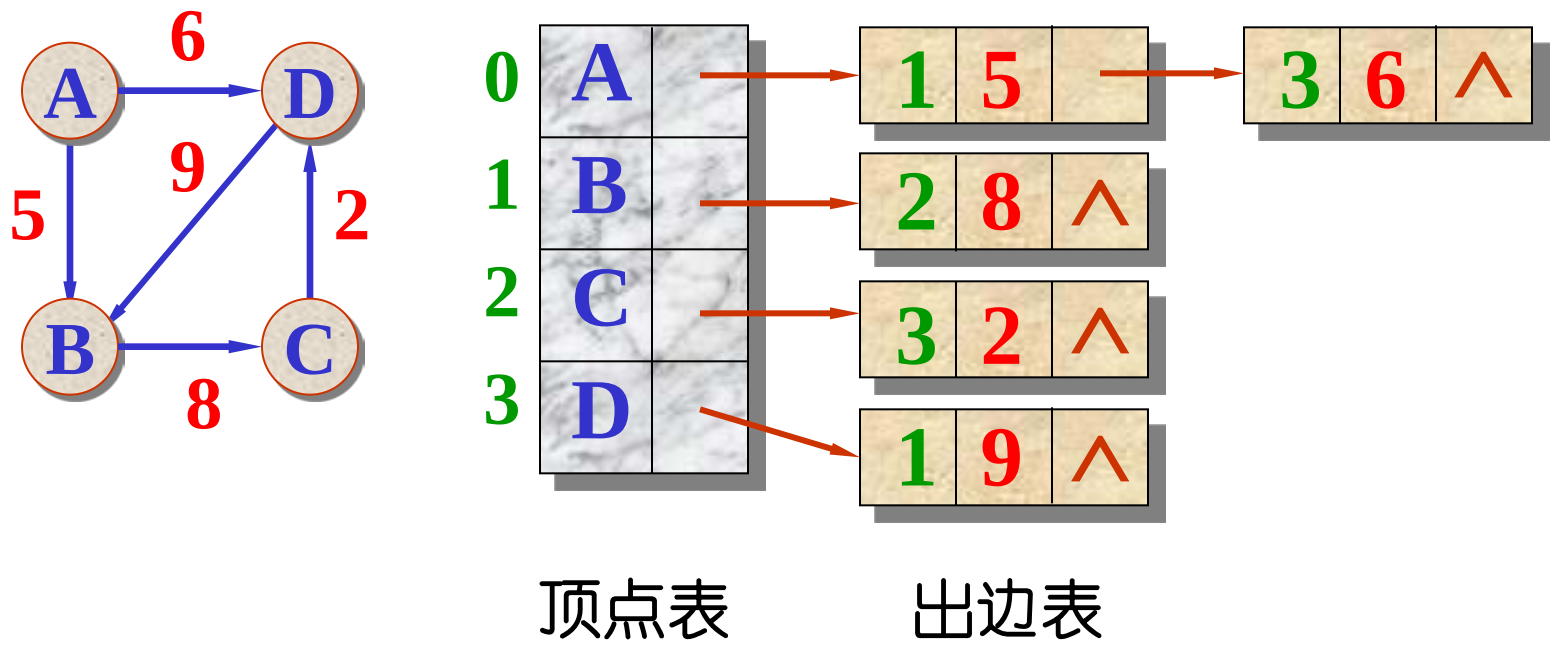
图的存储结构：邻接表

- 有向图的邻接表和逆邻接表



图的存储结构：邻接表

- 网络的邻接表



图的存储结构：邻接表

- 邻接表 V.S. 邻接矩阵
  - 若无向图有n个顶点、e条边，则它的邻接表需要n个头结点和2e个表结点
  - 而邻接矩阵需要n^2个数组单元
  - 所以如果边比较少，邻接表更节省空间

图的存储结构：邻接表

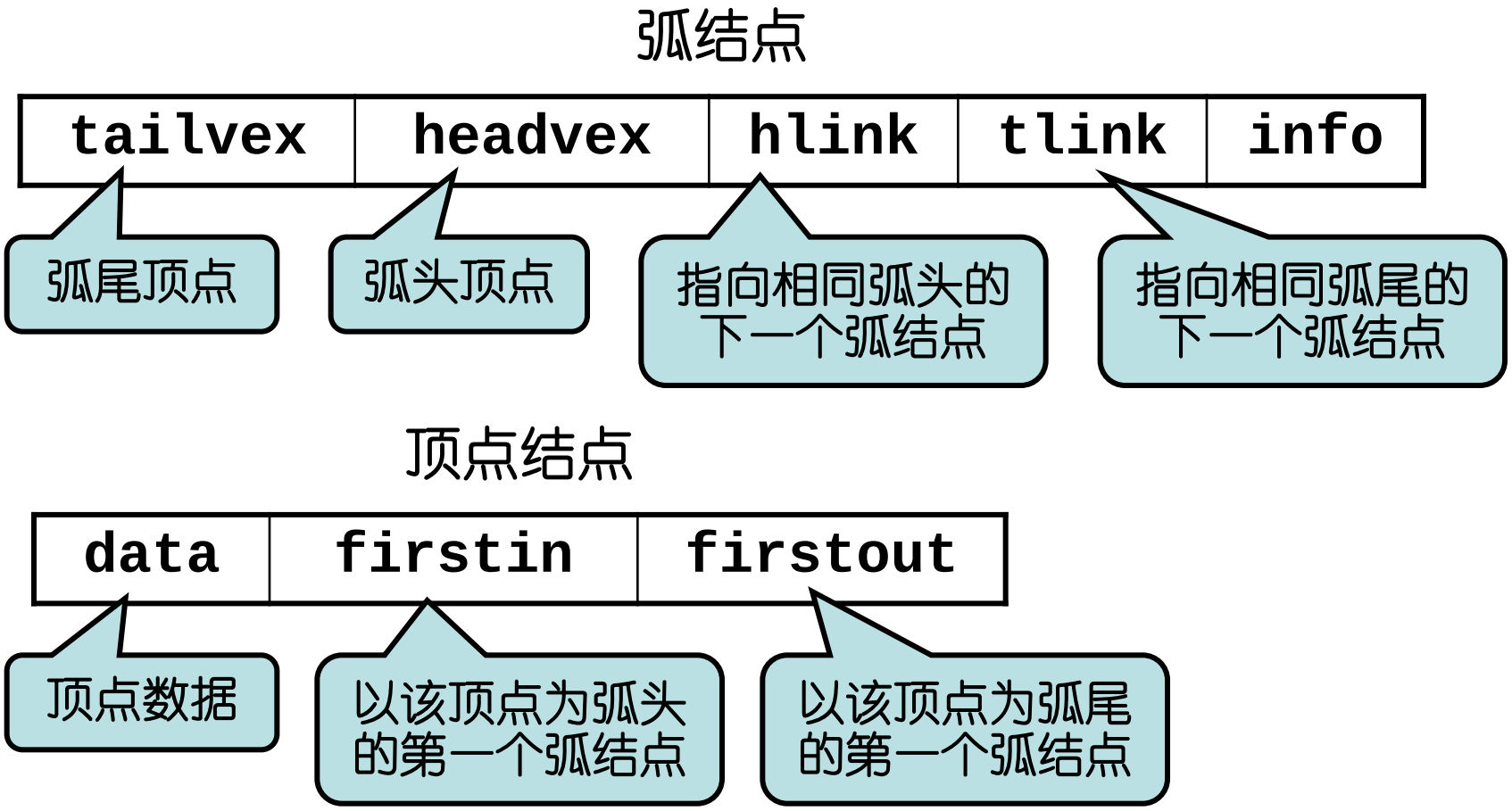
- 邻接表的操作
  - 计算某个顶点的度、出度、入度
    - 无向图中某个头结点后面链接的表结点的个数就是该顶点的度
    - 有向图中：
      - 邻接表的某个头结点后面链接的表结点的个数就是该顶点的出度
      - 逆邻接表的某个头结点后面链接的表结点的个数就是该顶点的入度

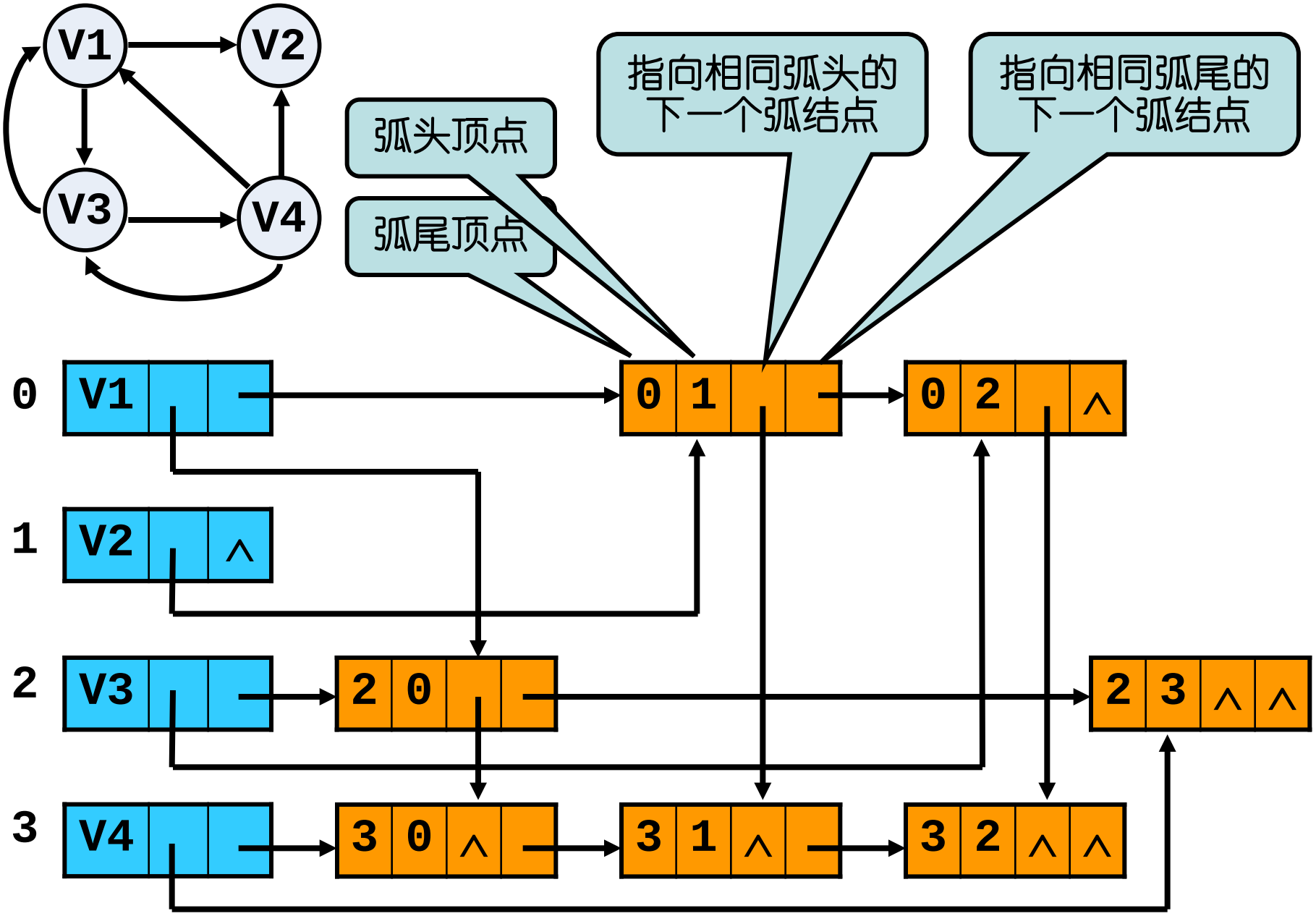
图的存储结构：邻接表

- 判断任意两个顶点(Vi,Vj)是否有边或弧
  - 需要搜索第i个或第j个链表
- 找出某个顶点的所有邻接顶点
  - 只需找到表示这个顶点的头结点

图的存储结构：十字链表

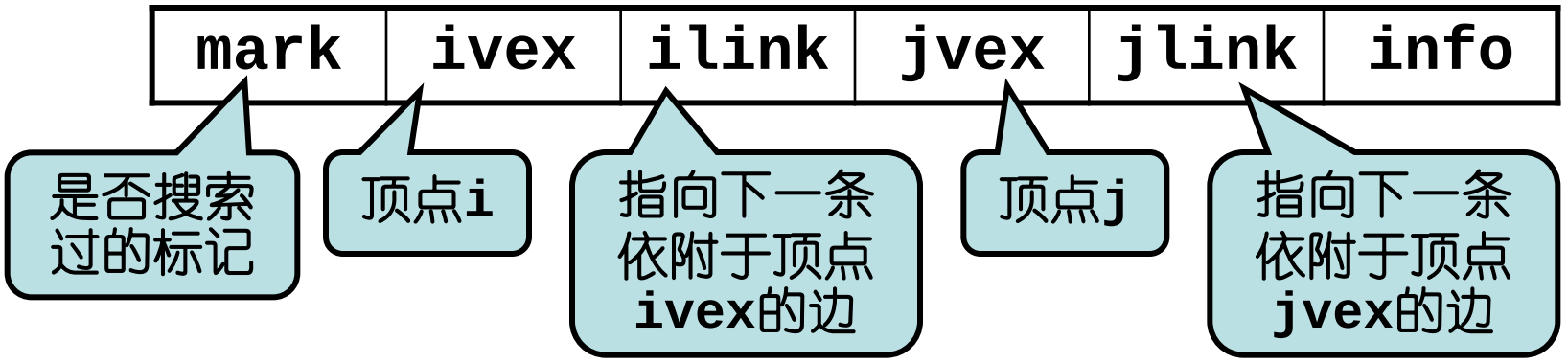
- 十字链表（有向图）

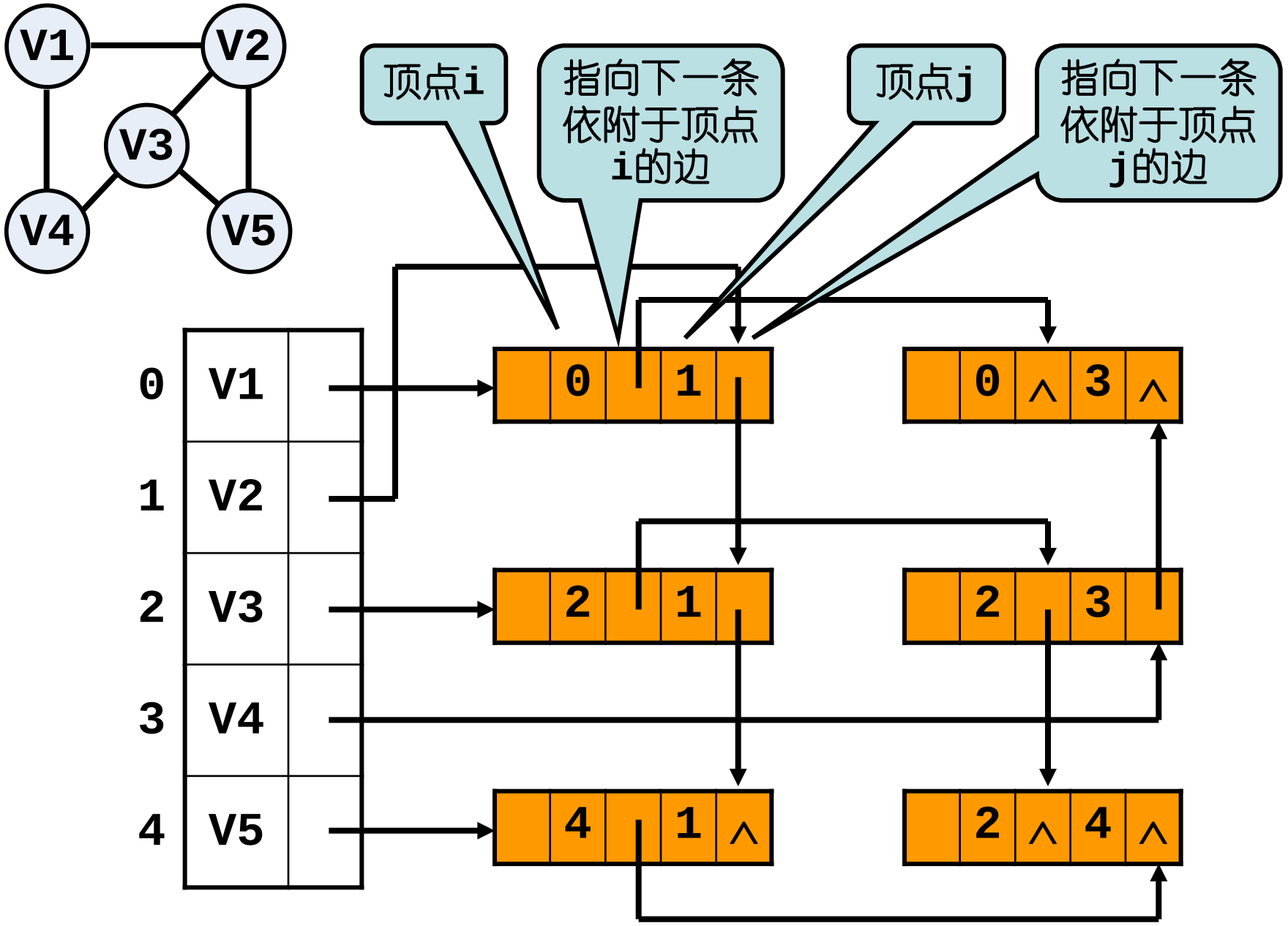



V1
V2
V3
V4
弧头顶点
弧尾顶点
指向相同弧头的
下一个弧结点
指向相同弧尾的
下一个弧结点
0
V1
0 1
0 2 ^
1
V2 ^
2
V3
2 0
2 3 ^ ^
3
V4
3 0 ^
3 1 ^
3 2 ^ ^

图的存储结构：邻接多重表

- 邻接表
  - 每一条边(vi,vj)有两个结点
    - 浪费空间
    - 操作不方便
- 多重邻接表
  - 每一条边只用一个结点表示




V1
V2
V3
V4
V5
顶点i
指向下一条依附于顶点i的边
顶点j
指向下一条依附于顶点j的边
0
V1
1
V2
2
V3
3
V4
4
V5
0
1
0
^
3
^
2
1
2
3
4
1
^
2
^
4
^

图的存储结构

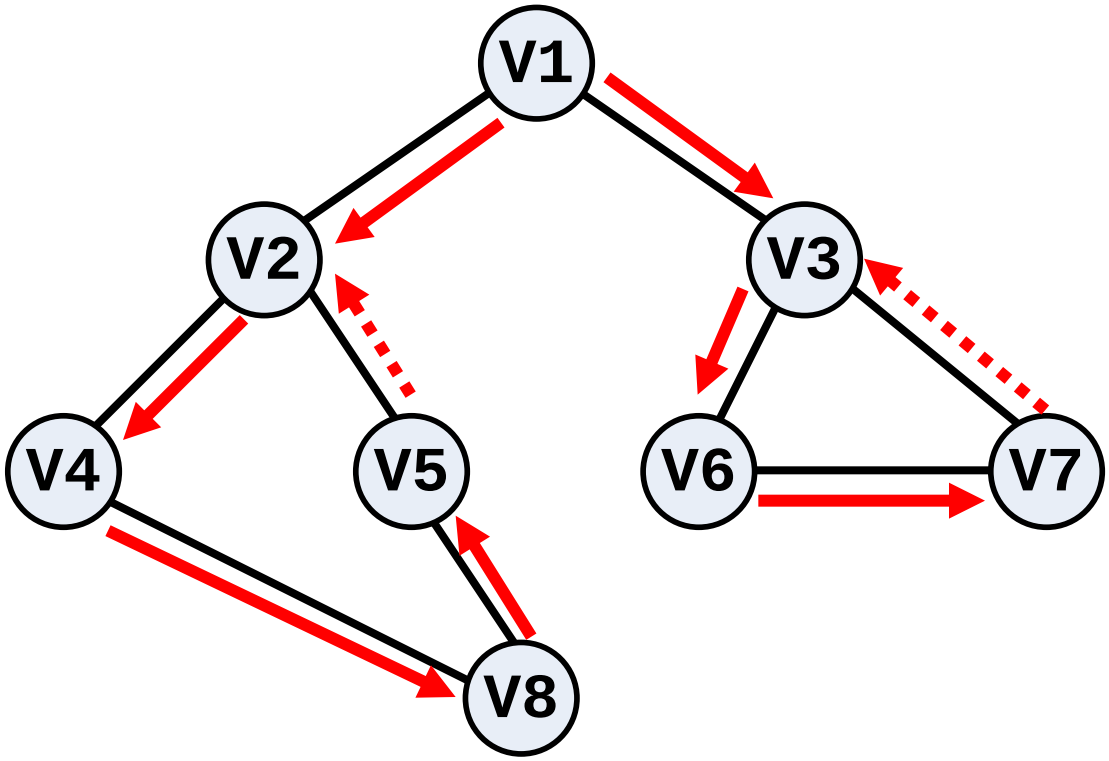
- 本节小结
  - 4种存储结构
  - 能够手工完成：图 <--> 存储结构
  - 重点是邻接矩阵和邻接表
- 思考题2
  - 可以自己随手画一个图，试着画出这个图的4种存储结构

图的遍历

- 图的遍历
  - 从图中某一顶点出发，访问图中所有顶点，并且每个顶点仅被访问一次
  - 为了避免重复，用数组**visited[]**记录每个顶点是否已经被访问过
  - 深度优先遍历
  - 广度优先遍历

图的遍历：深度优先遍历

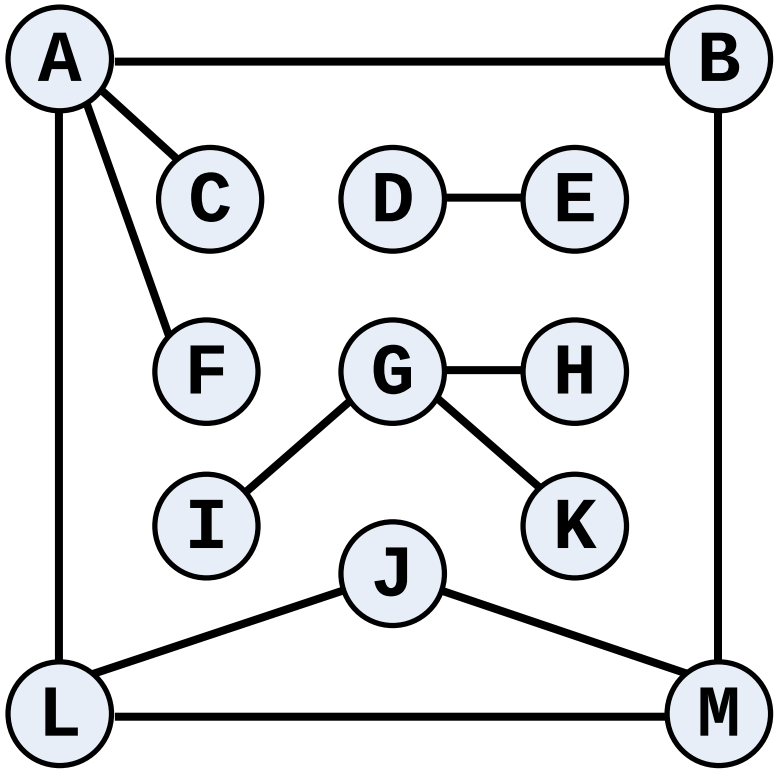
- 深度优先遍历
  - 访问顶点v
  - 依次从v的每一个未被访问的邻接点出发进行深度优先遍历
  - 若此时图中尚有顶点未被访问，则任选其一为起点
  - 重复，直至图中所有顶点都被访问到
  - 又是递归定义！



V1->V2->V4->V8->V5->V3->V6->V7

图的遍历：深度优先遍历

- 练习
  - 写出对下图进行深度优先遍历的结果



- **A B M J L C F**
- **D E**
- **G H I K**

图的遍历：深度优先遍历

- 理解
  - 如果是连通图，大致就是如下过程：
    - 从一个顶点出发，一直往前走
    - 无路可走时，再回头看看有没有没走过的岔路口
  - 如果不是连通图，则分别对每个连通分量进行上述过程
    - 上述过程也告诉我们：从一个顶点出发凡是能遍历到的顶点就形成一个连通分量

```
//图的深度优先遍历
void DFSTranverse(Graph G){
    //初始时将所有顶点都设置为未访问
    for(v = 0; v < G.vexnum; v ++)
        visited[v] = false;
    //从每一个顶点出发
    for(v = 0; v < G.vexnum; v ++)
        //没访问过就开始遍历
        if(!visited[v])
            DFS(G, v);
}
```

```
//递归函数
void DFS(Graph, int v){
    visited[v] = true; //将点v设为已访问
    VisitFunc(v);      //调用访问函数

    //依次对v的每一个邻接点，调用DFS函数
    for(w = FirstAdjVex(G, v); w >= 0;
        w = NextAdjVex(g, v, w))
        if(!visited[w])
            DFS(G, w);
}
```

```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

将v设置为已访问，并访问之

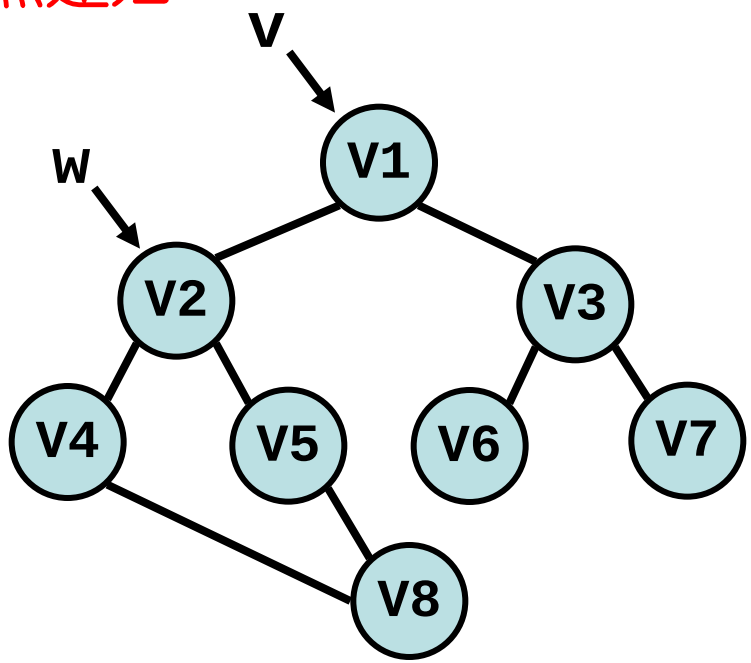
对于v的每个邻接点w

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0

v1



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

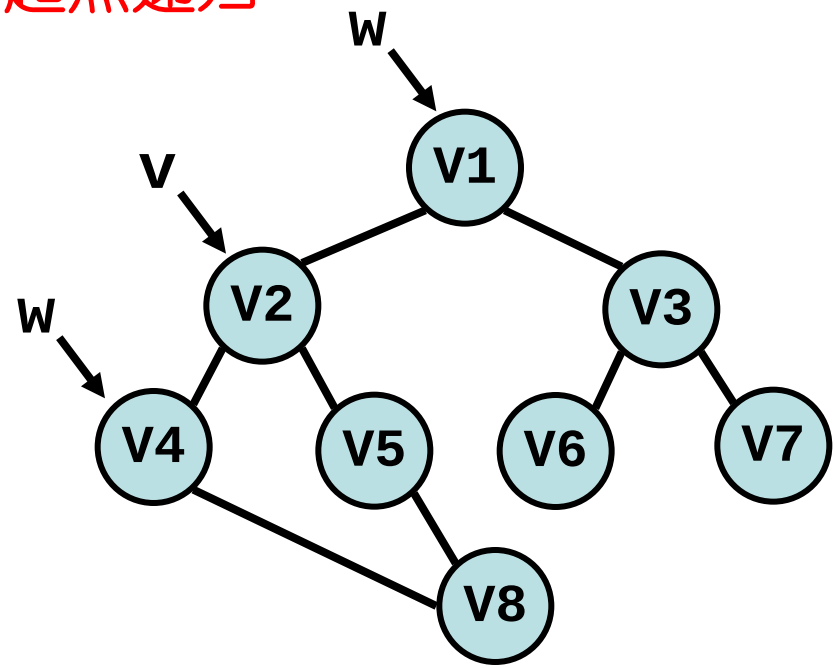
对于v的每个邻接点w

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	0	0	0	0	0	0

v1 v2



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

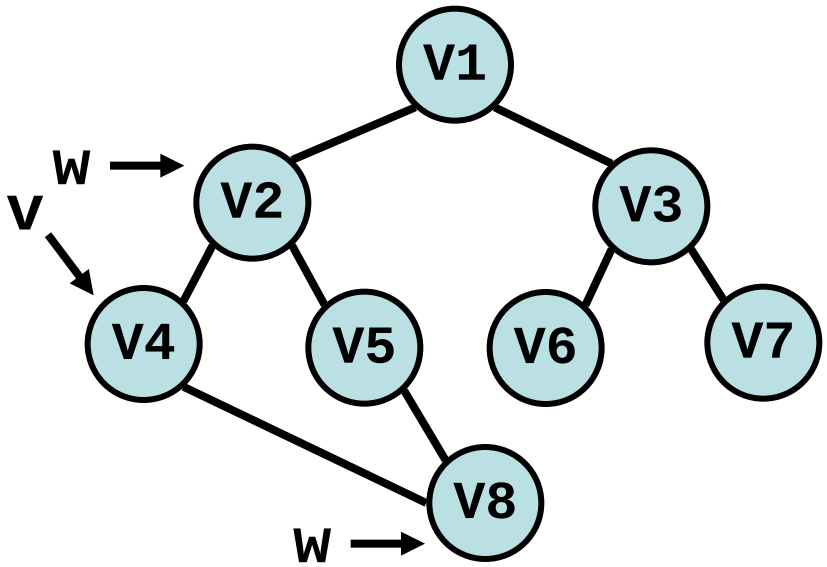
对于v的每个邻接点w

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	0	1	0	0	0	0

v1 v2 v4



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

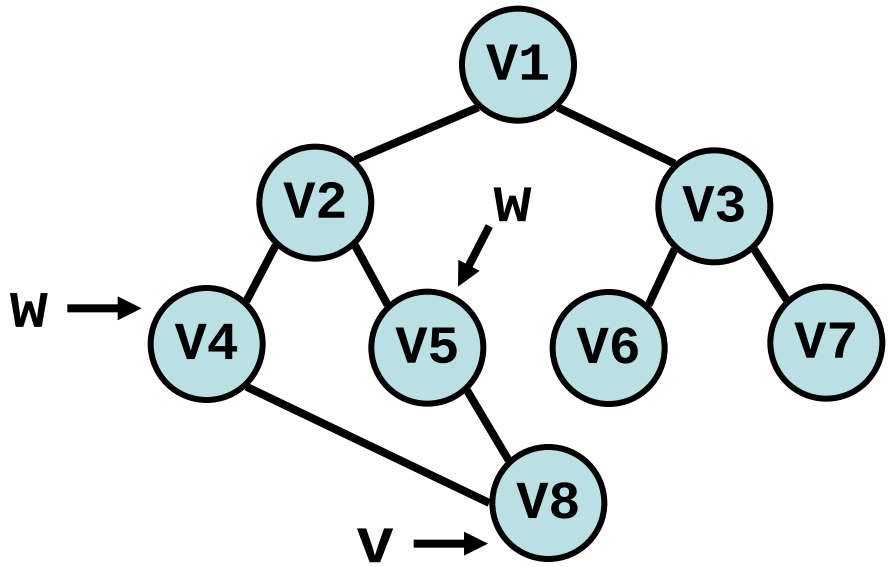
对于v的每个邻接点w

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	0	1	0	0	0	1

v1 v2 v4 v8



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

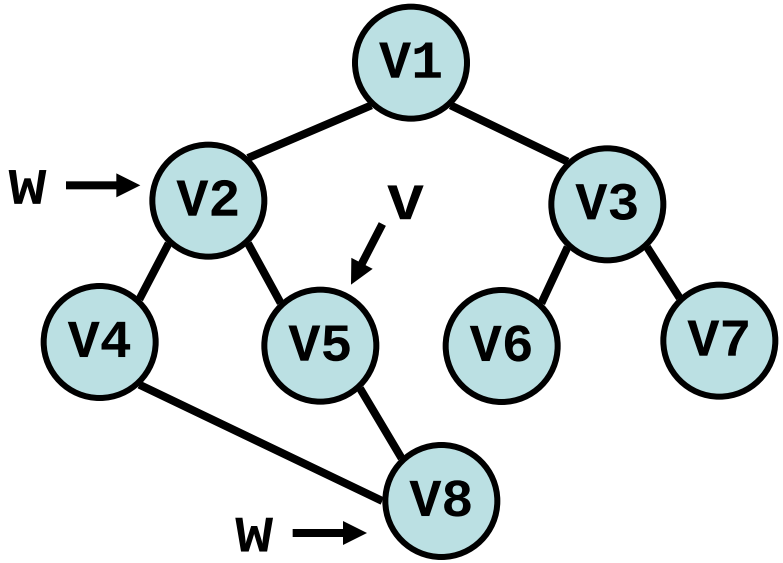
for语句结束，对v的每个邻接点w函数返回

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1

v1 v2 v4 v8 v5



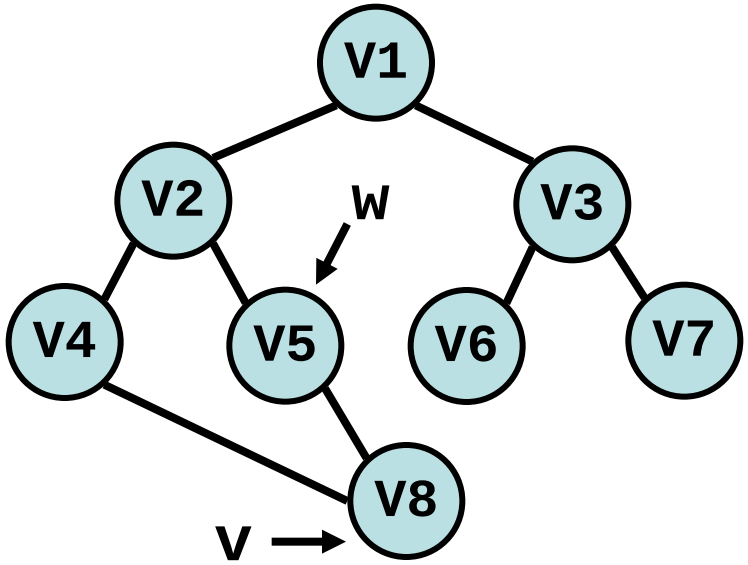
```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

for语句结束，函数返回

visited

1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1

v1 v2 v4 v8 v5



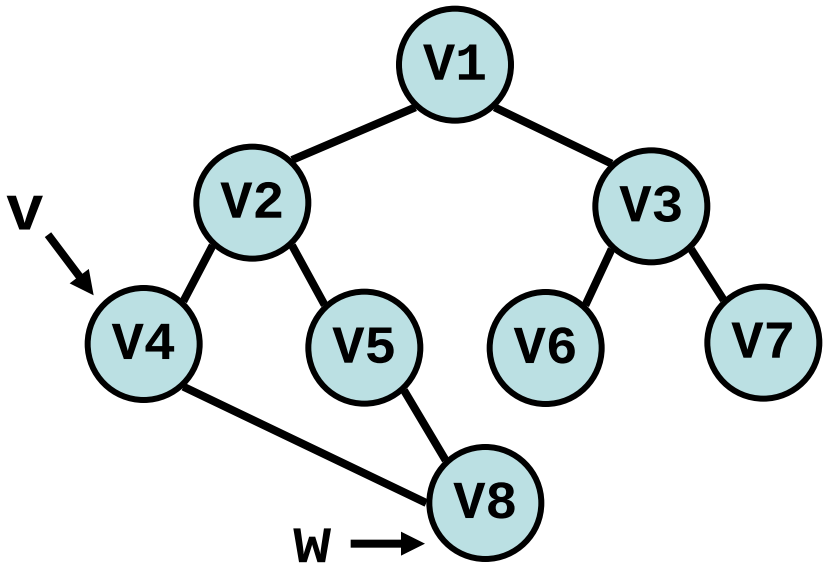
```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

for语句结束，函数返回

visited

1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1

v1 v2 v4 v8 v5



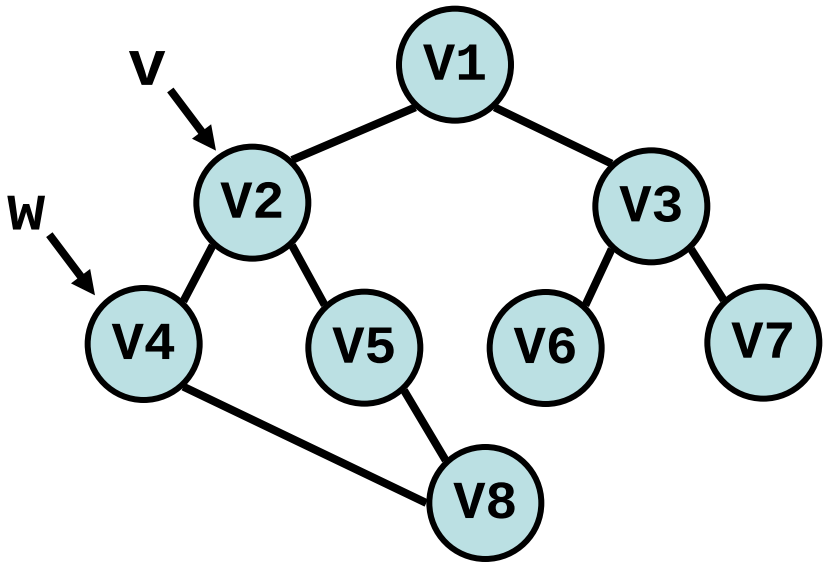
```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

for语句结束，函数返回

visited

1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1

v1 v2 v4 v8 v5



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

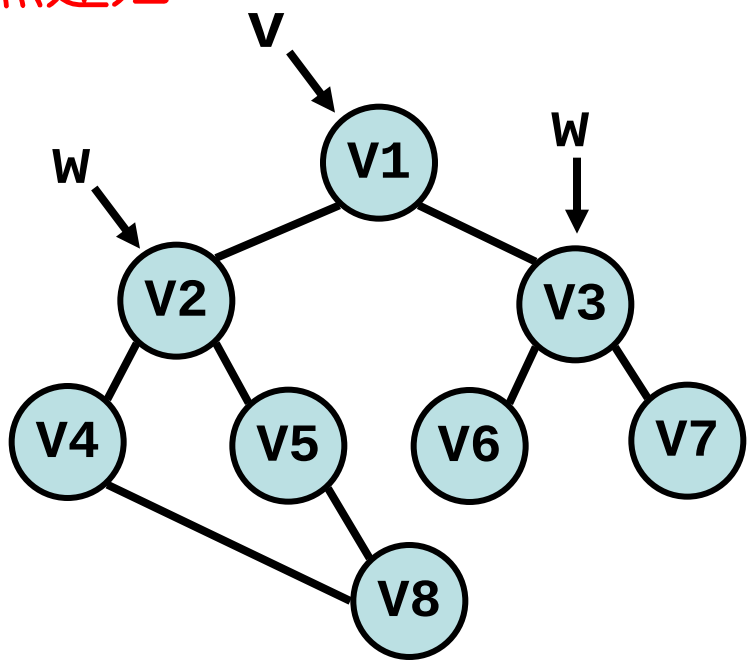
v的下一个邻接点w

若w未访问过，
以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	0	1	1	0	0	1

v1 v2 v4 v8 v5



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

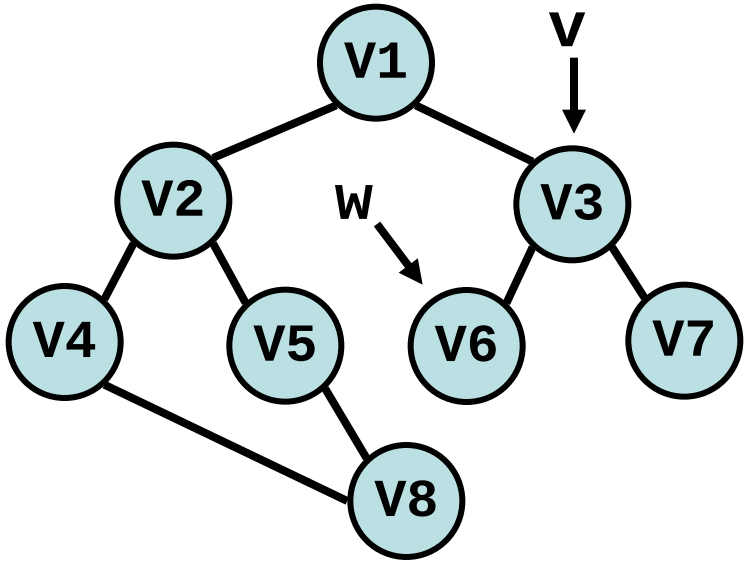
对于v的每个邻接点w

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	0	0	1

v1 v2 v4 v8 v5 v3



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

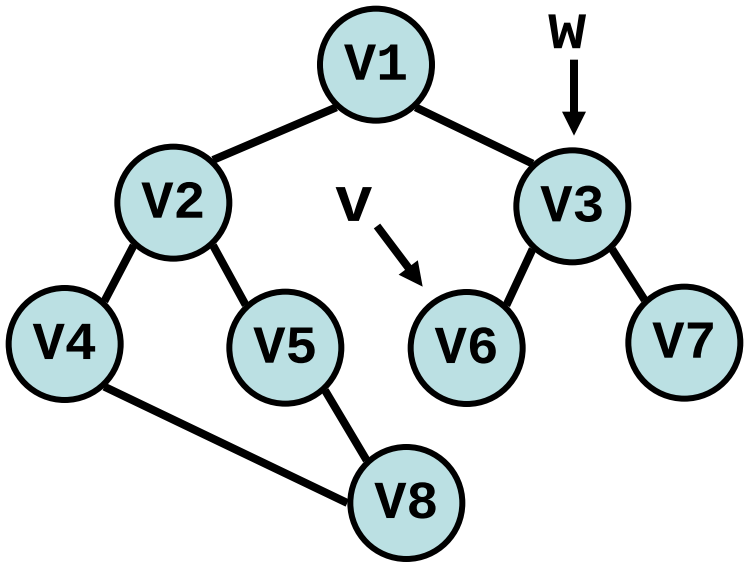
for语句结束对v的每个邻接点w函数返回

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	1	0	1

v1 v2 v4 v8 v5 v3 v6



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

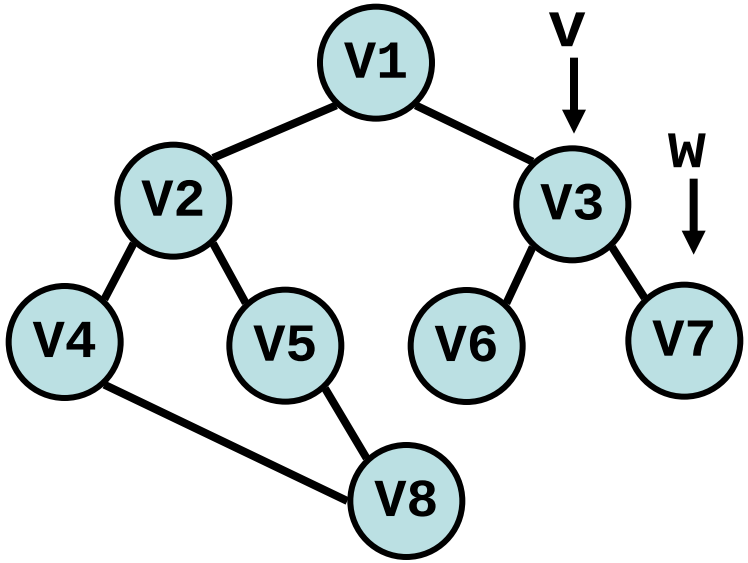
v的下一个邻接点w

若w未访问过，
以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	1	0	1

v1 v2 v4 v8 v5 v3



```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

此时的v就是原来的w

将v设置为已访问，并访问之

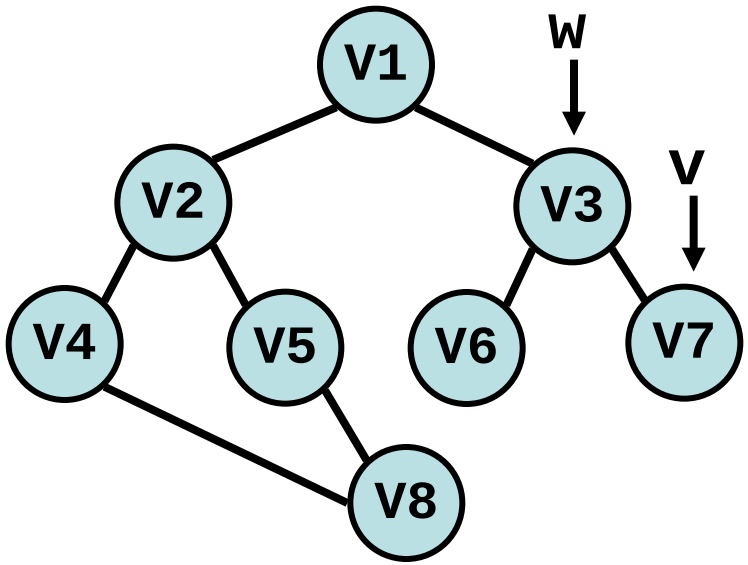
for语句结束，对v的每个邻接点w函数返回

若w未访问过，以w为起点递归

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

v1 v2 v4 v8 v5 v3 v6 v7



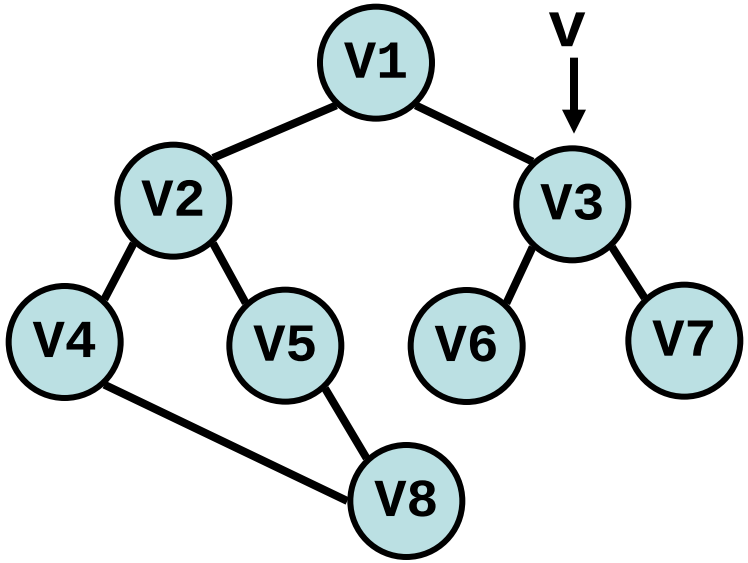
```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

for语句结束，函数返回

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

v1 v2 v4 v8 v5 v3 v6 v7



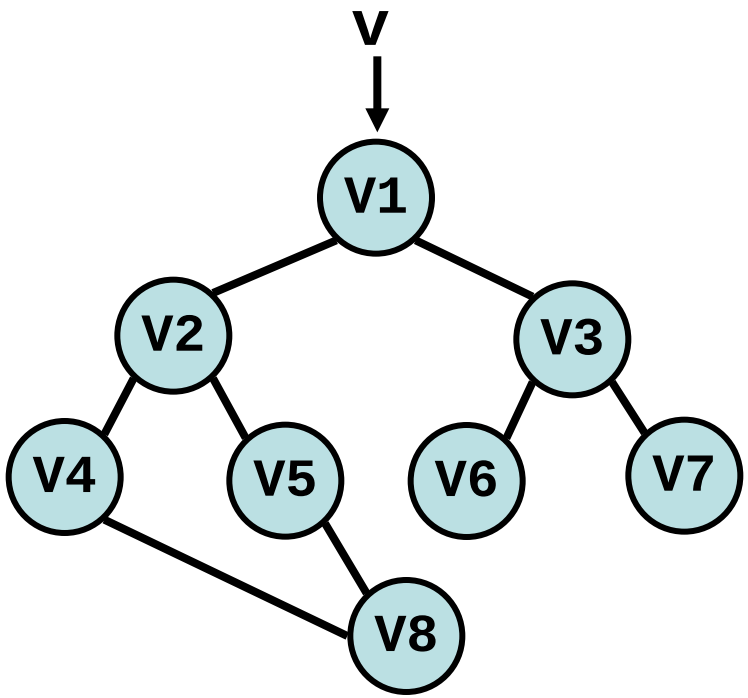
```
void DFS(Graph G, int v)
{
   visited[v] = true;   VisitFunc(v);
   for(w = FirstAdjVex(G, v); w >= 0;
       w = NextAdjVex(G, v, w))
      if(!visited[w])
         DFS(G, w);
}
```

for语句结束，函数返回

visited

1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1

v1 v2 v4 v8 v5 v3 v6 v7



```
void DFS(Graph G, int v)
{
    visited[v] = true;
    VisitFunc(v);
    for(w = FirstAdjVex(G, v);
        w >= 0;
        w = NextAdjVex(G, v, w))
       if(!visited[w])
          DFS(G, w);
}
```

首先访问当前节点

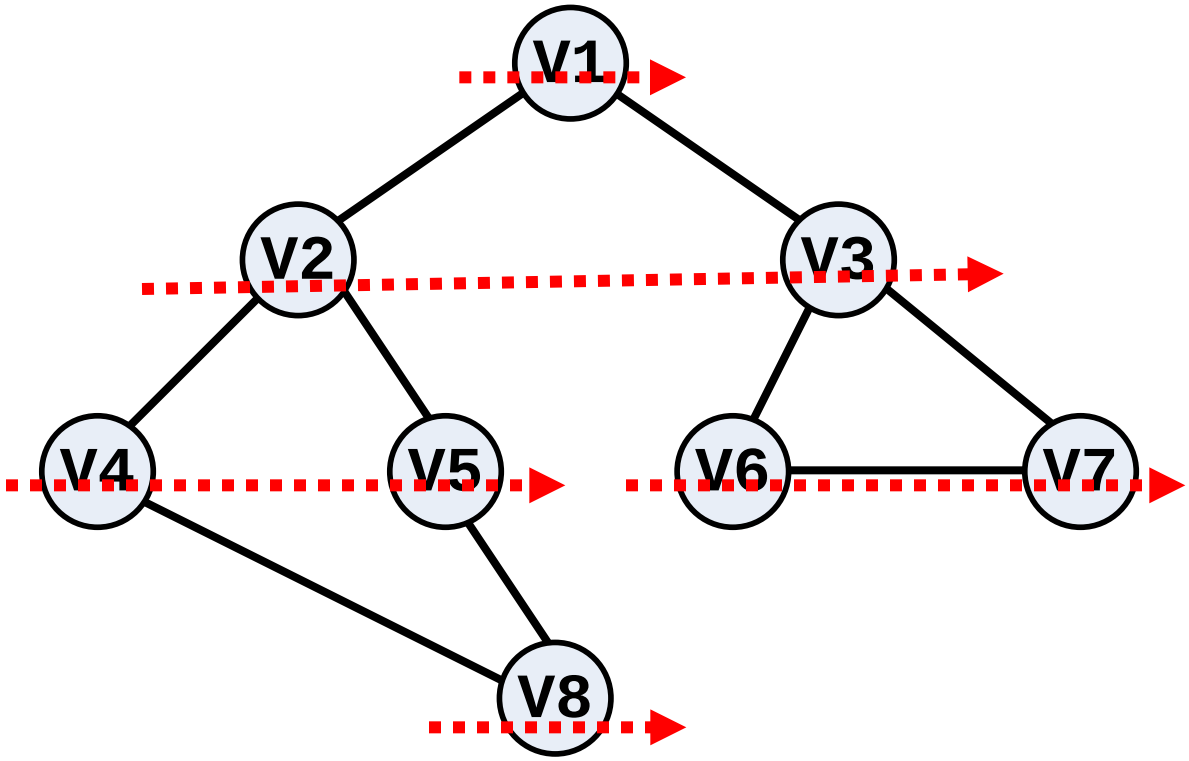
如果w走到底了，换下个邻接点w
尝试如下尝试：
直到所有邻接点都尝试过

只要w未访问过，
“向前走”
即以w为起点递归调用

其实“走迷宫”的问题用的也是同样的思想

图的遍历：广度优先遍历

- 广度优先遍历
  - 访问顶点v
  - 依次访问顶点v的各个未被访问的邻接点
  - 分别从这些邻接点出发，依次访问它们的邻接点，并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问
  - 若此时图中尚有顶点未被访问，则任选其一为起点，重复，直至图中所有顶点都被访问到
  - 注意对比树的层序遍历

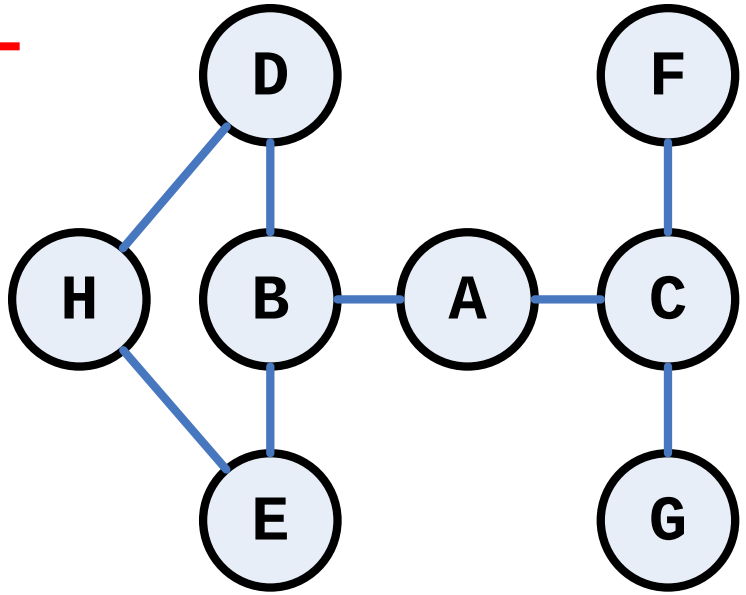


V1->V2->V3->V4->V5->V6->V7->V8

图的遍历：广度优先遍历

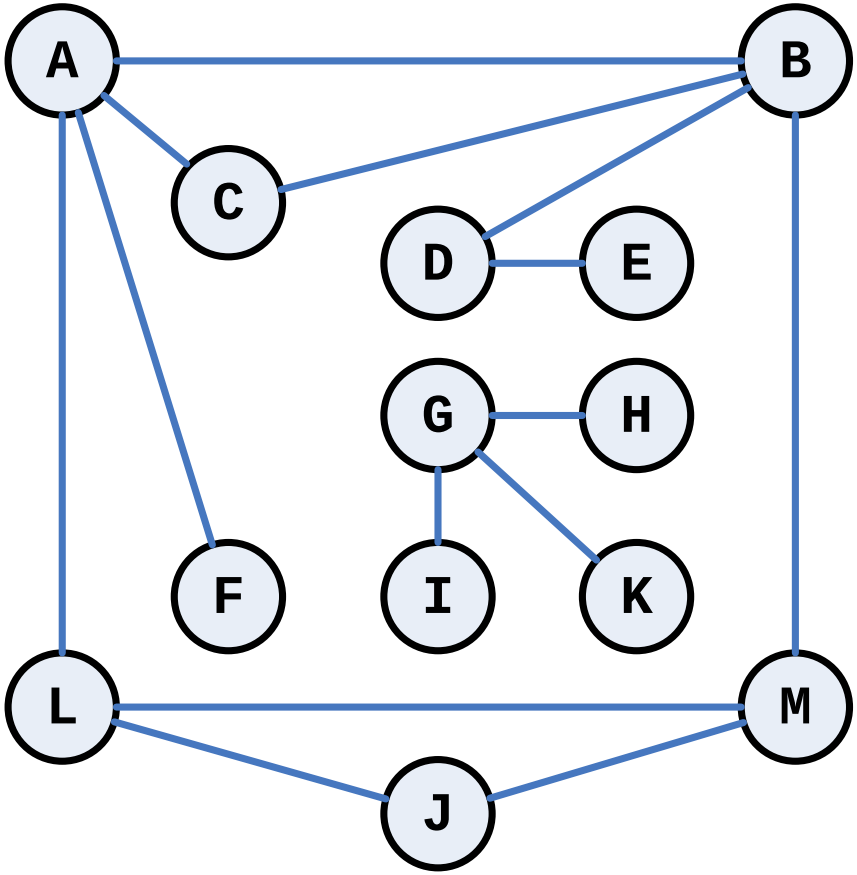
- 练习
  - 写出对下图进行广度优先遍历的结果

A B C D E F G H



图的遍历：广度优先遍历

- 练习
  - 写出对下图进行深度优先遍历和广度优先遍历的结果



图的遍历：广度优先遍历

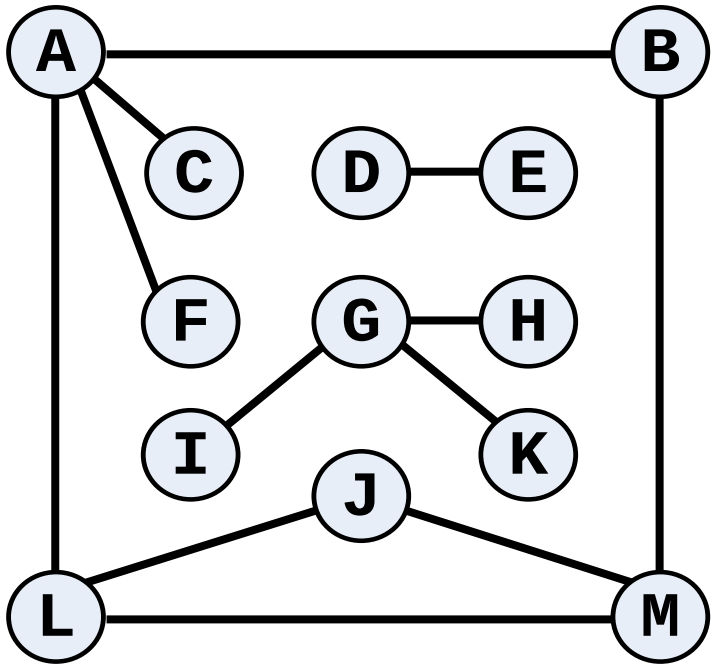
- 算法
  - 对比二叉树的层序遍历算法

图的遍历

- 借助遍历可以很容易做到：
  - 如果是无向图，写出所有连通分量
  - 判断两个顶点是否连通（即是否有路径）
- 作业
  - 在遍历算法的基础上，写出解决以上两个问题的算法
  - 提示：
    - 如果是无向图，所谓两个顶点连通，就是从其中一个出发进行遍历，能够经过另一个
    - 如果是有向图呢？

图的连通性问题：连通分量

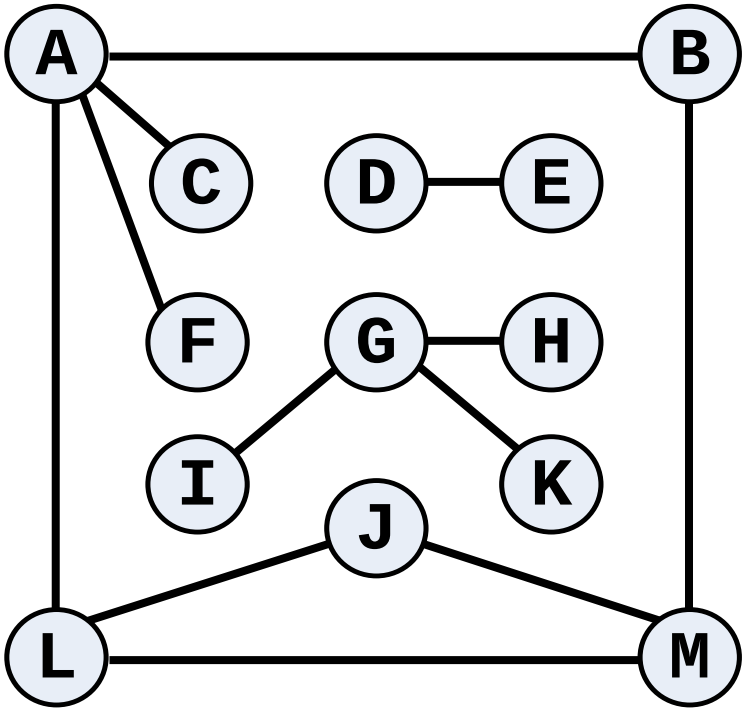
- 无向图的连通分量
  - 连通分量：无向图的极大连通子图
  - 连通图只有一个连通分量，就是它本身
  - 非连通图有多个连通分量



  - 3个连通分量：
    - ALMJBFC
    - DE
    - GKHI

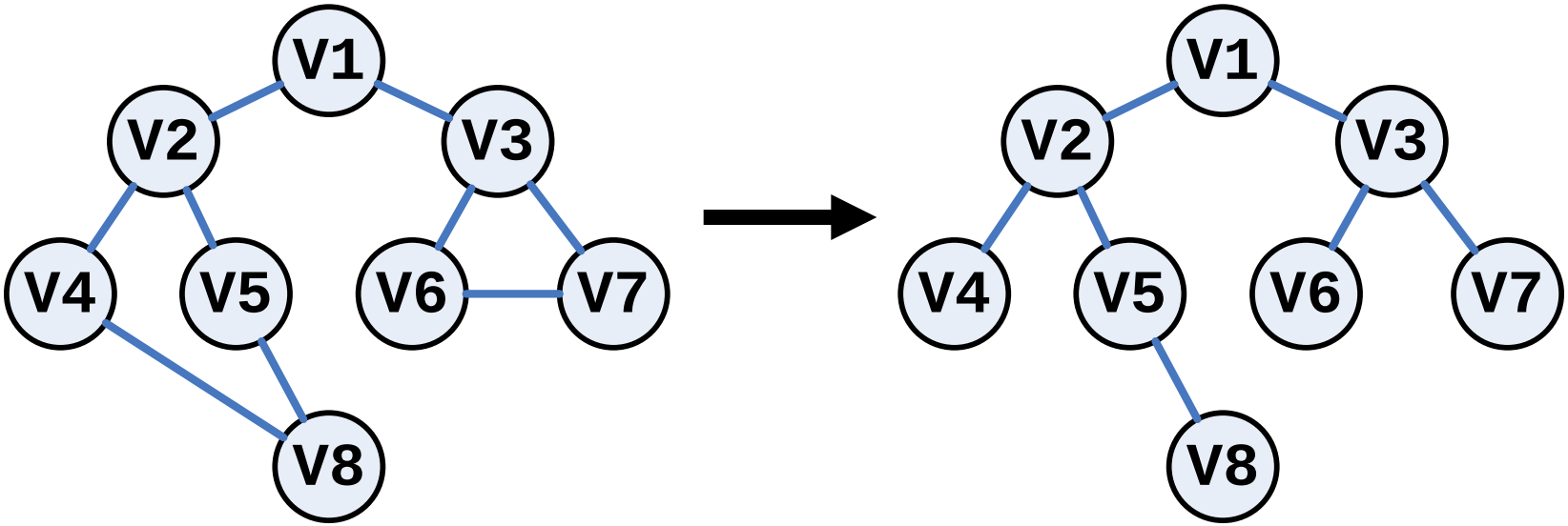
图的连通性问题：连通分量

- 求无向图的连通分量
  - 从每个顶点出发进行遍历（深度或广度优先）
  - 每一次从一个新的起点出发进行遍历得到的顶点序列就是一个连通分量的顶点集合



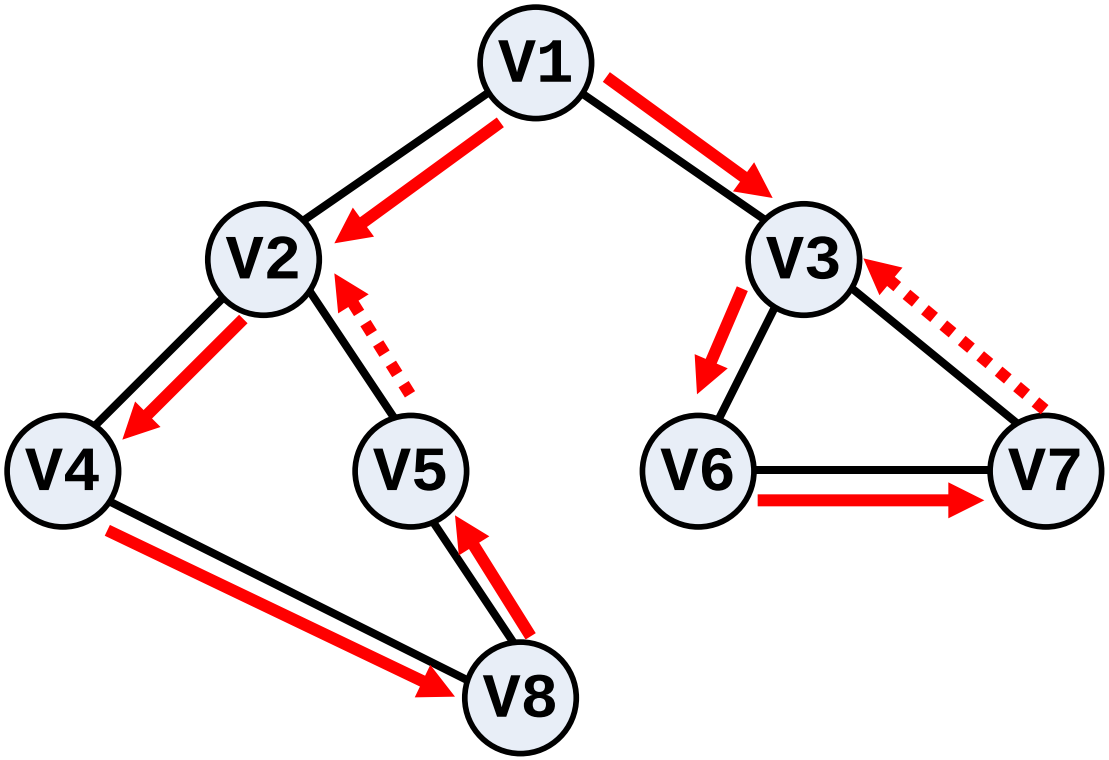
图的连通性问题：生成树

- 生成树（Spanning Tree）
  - 包含原连通图的所有n个顶点
  - 包含原有的其中n-1条边
  - 且保证连通



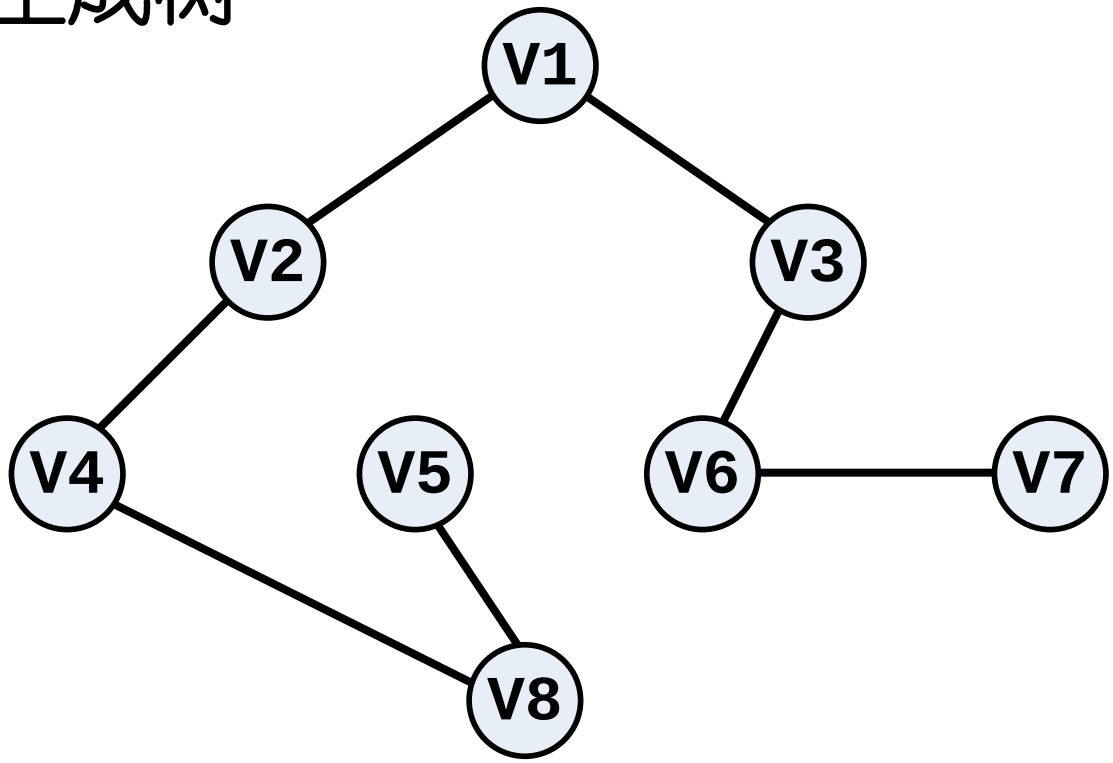
图的连通性问题：生成树

- 求无向图的生成树
  - 对无向图进行遍历（深度或者广度优先）
  - 所经过的边的集合 + 所有顶点 = 生成树



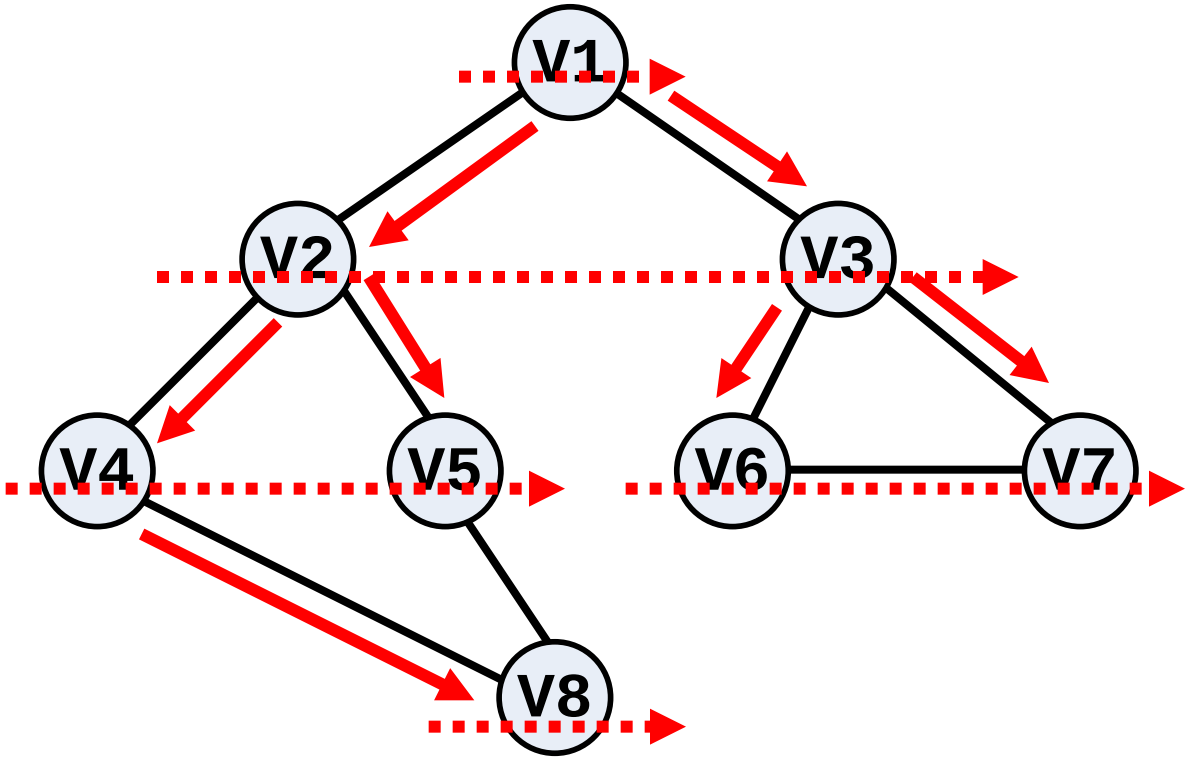
图的连通性问题：生成树

- 深度优先生成树：深度优先遍历得到的生成树



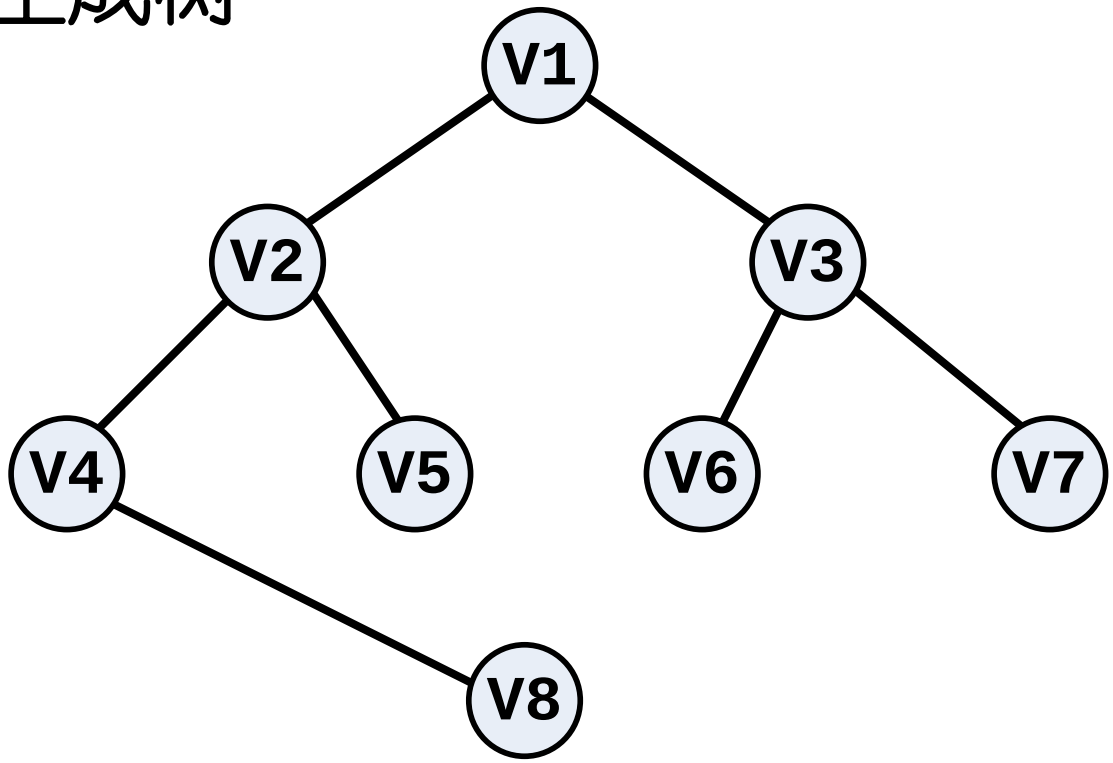
图的连通性问题：生成树

- 或者进行广度优先遍历：



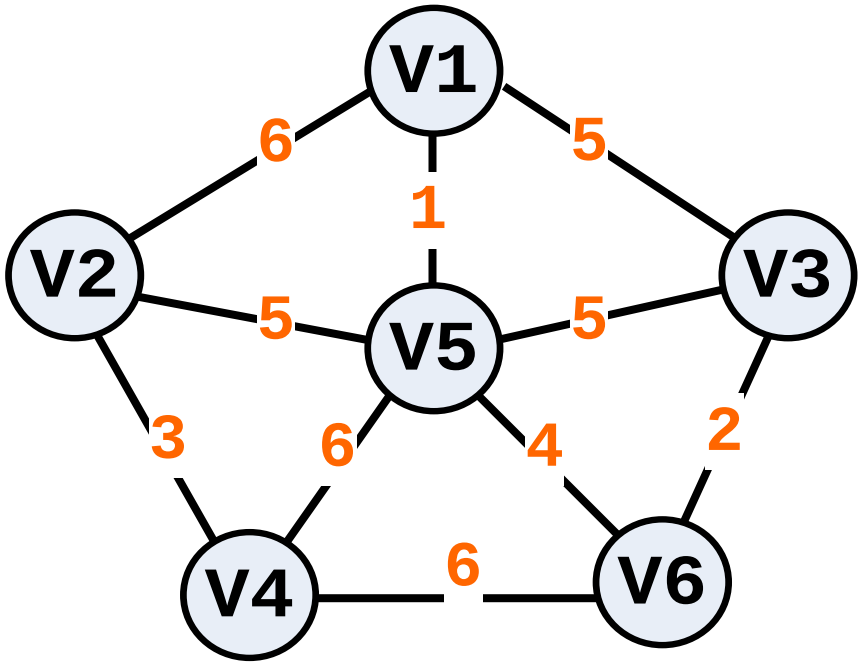
图的连通性问题：生成树

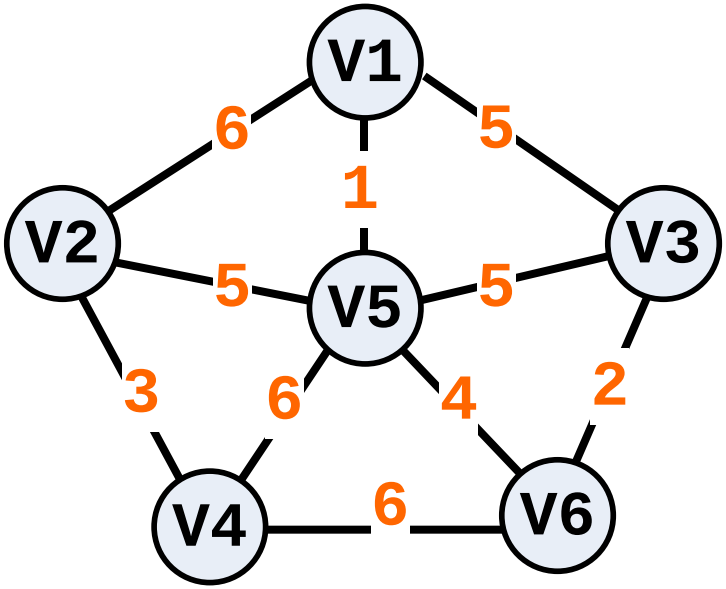
- 广度优先生成树：广度优先遍历得到的生成树

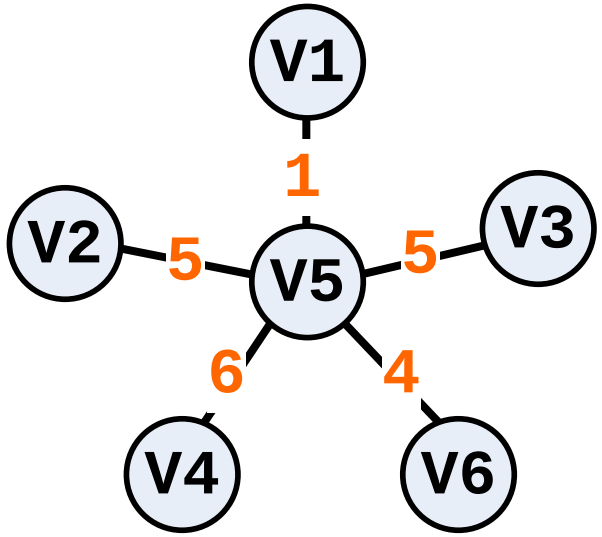


图的连通性问题：最小生成树

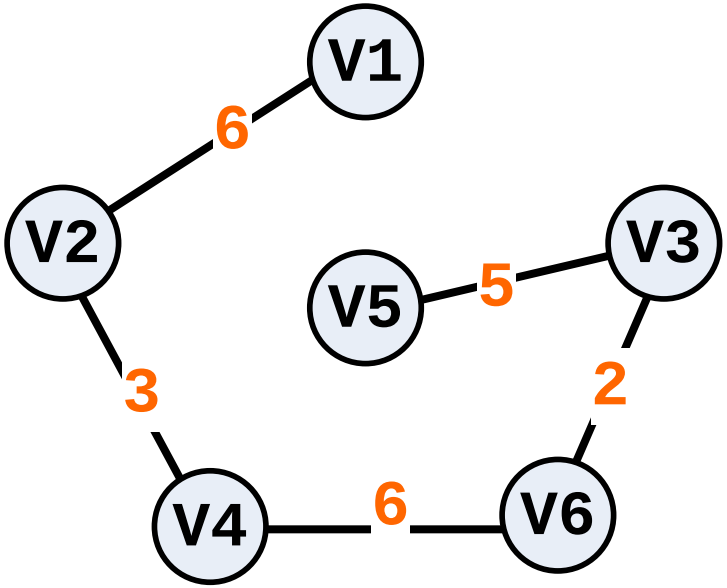
- 应用：连通n个城市的需要多少条道路？
  - 最多n(n-1)/2：任何两个城市之间都修一条路
  - 最少n-1，问题是哪n-1条能使总代价最小呢？




V1
V2
V3
V4
V5
V6
6
5
1
5
5
3
6
4
2
6


V1
V2
V3
V4
V5
V6
1
5
5
6
4

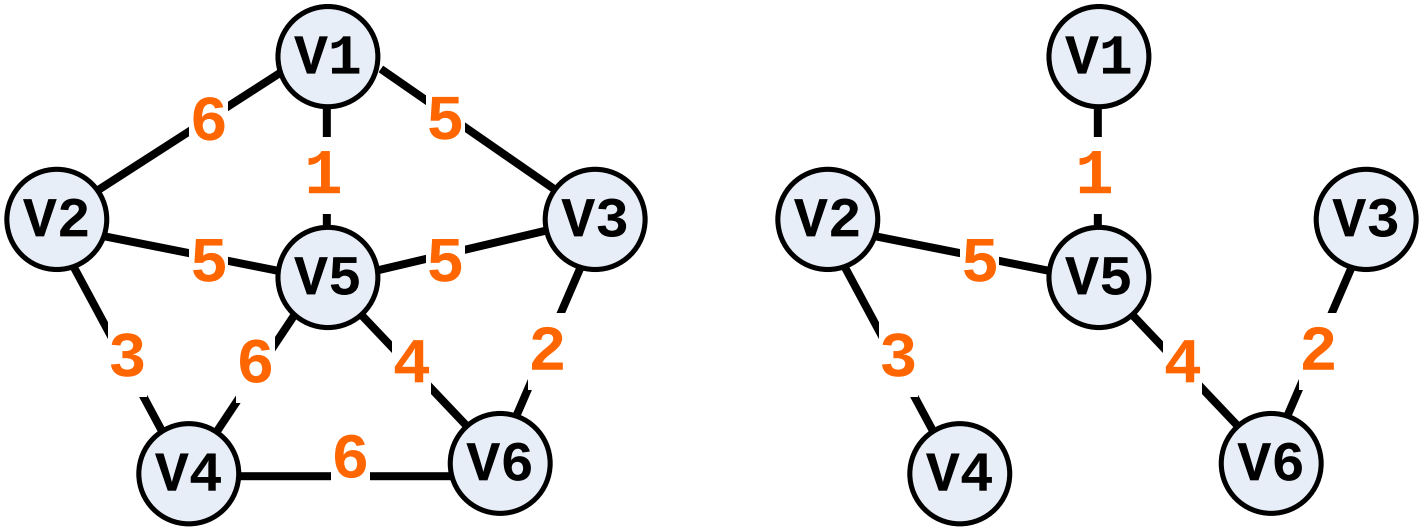
总长21


V1
V2
V3
V4
V5
V6
6
5
3
2
6

总长22

图的连通性问题：最小生成树

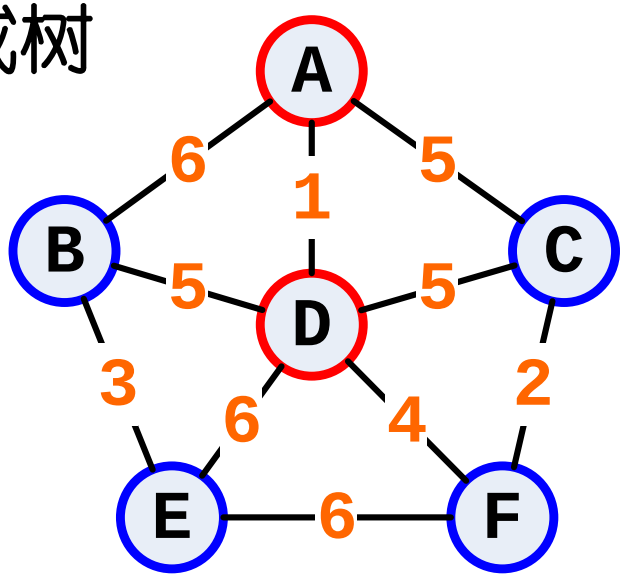
- 最小生成树
  - **Minimum Cost Spanning Tree**
  - 各边权值之和最小的生成树



- 问题：最小生成树唯一么？

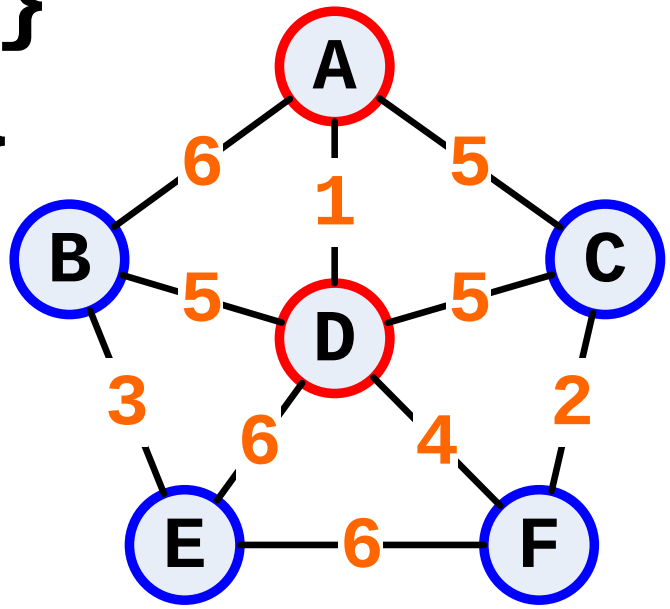
图的连通性问题：最小生成树

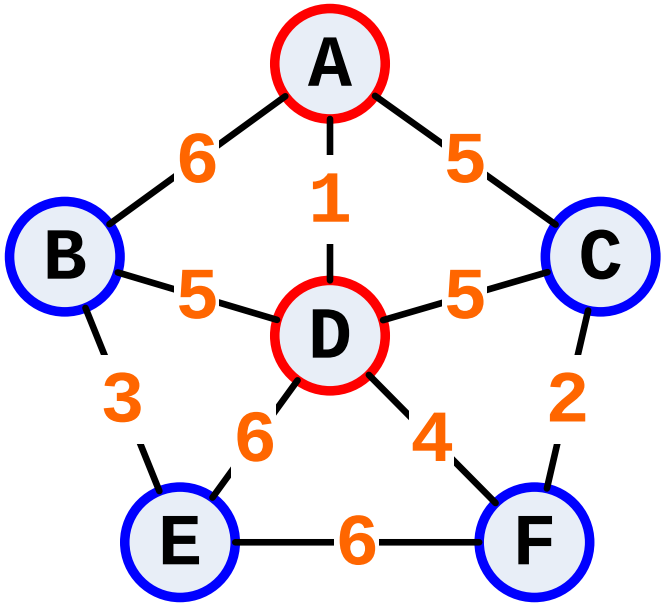
- 最小生成树的性质（MST性质）：
  - 设连通网 N=(V,{E})
  - U为V的非空子集
  - F={(v1,v2)|(v1,v2)∈E,v1∈U,v2∈V-U}
  - 设(u,v)是F中权值最小的边，则必存在一棵包含(u,v)的最小生成树



图的连通性问题：最小生成树

- 原图N = {V, {E}}
  - V = {A,B,C,D,E,F}
  - E = {(A,B),(A,C),(A,D)...}
- U⊂V，比如U = {A, D}
- V-U = {B, C, E, F}
- F = {(A,B),(A,C),
  (D,B),(D,C),
  (D,E),(D,F)}



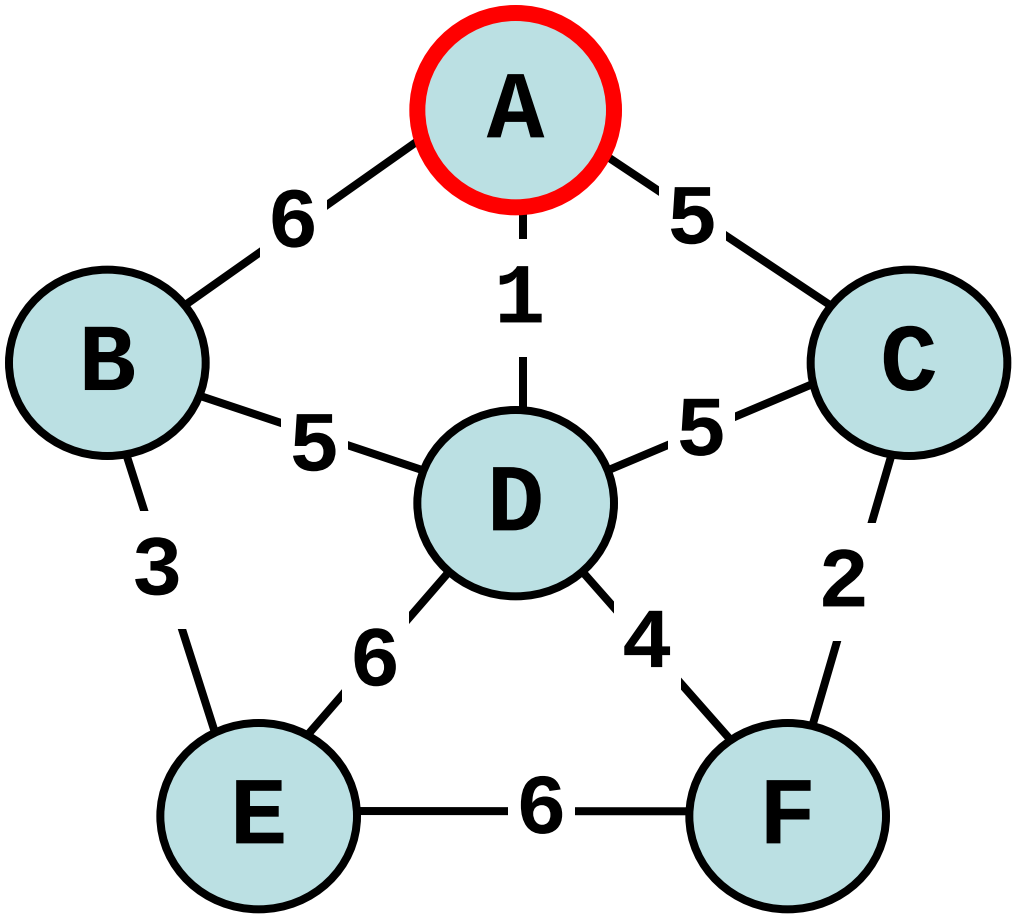


- F = {(A,B),(A,C),
  (D,B),(D,C),
  (D,E),(D,F)}
- (u,v)是F中权值最小的一条边
  = (D,F)
- 结论是：N的最小生成树中一定有一棵包含了(D,F)

最小生成树

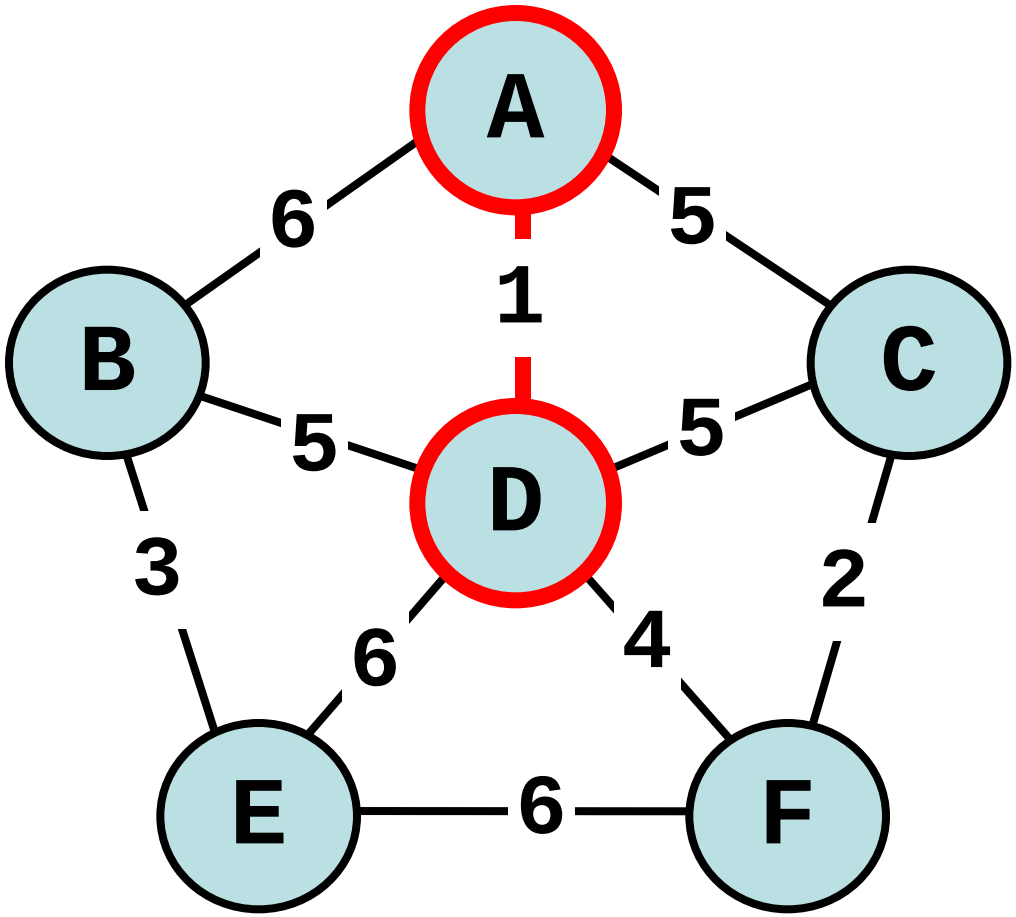
- **Prim算法**
  - U为最小生成树中顶点的集合
    - 初始时，$U=\{u_0\}$
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中

- **Prim算法**
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



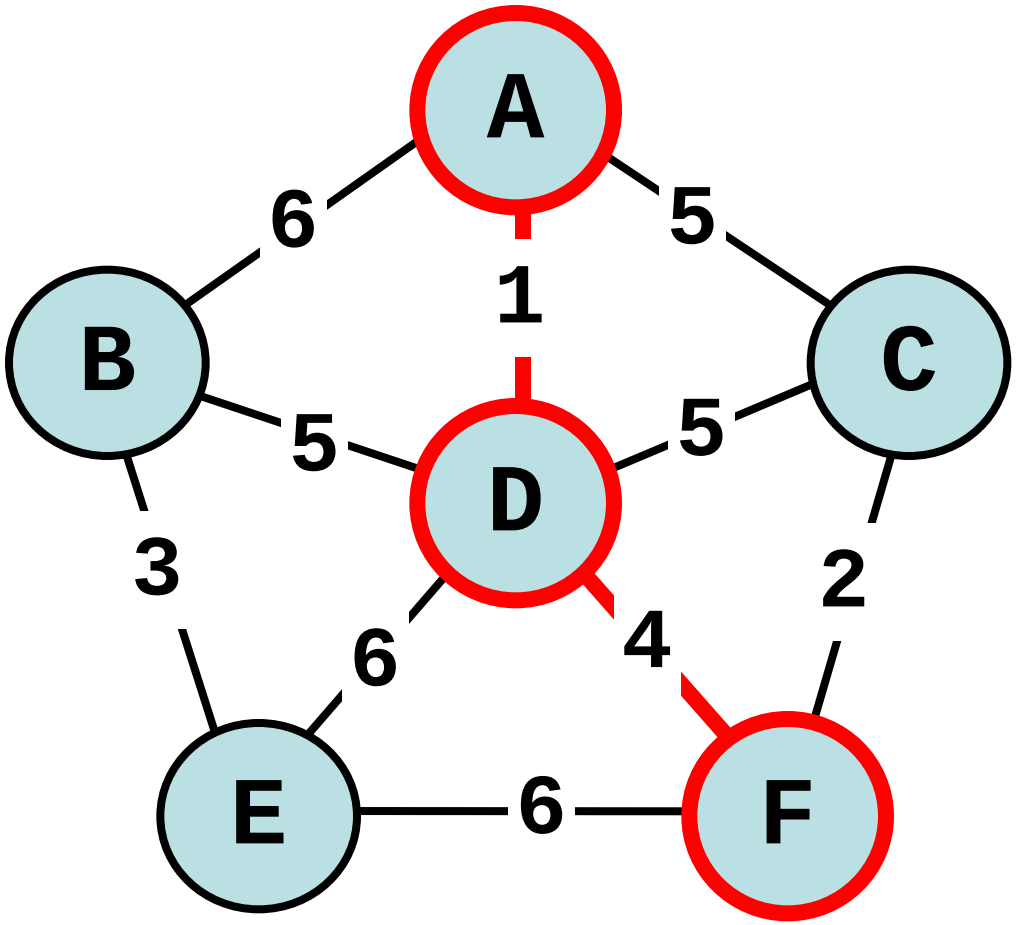
U = {A}

- **Prim算法**
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



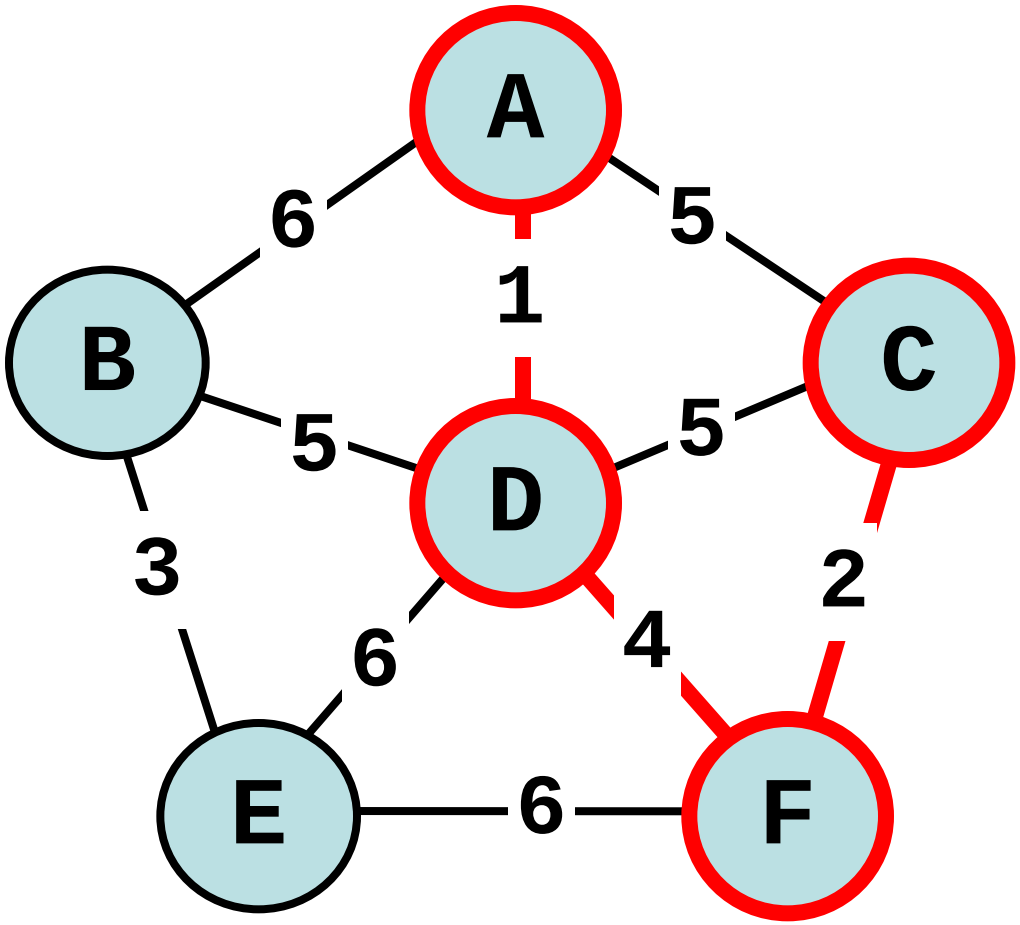
U = {A,D}

- **Prim算法**
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



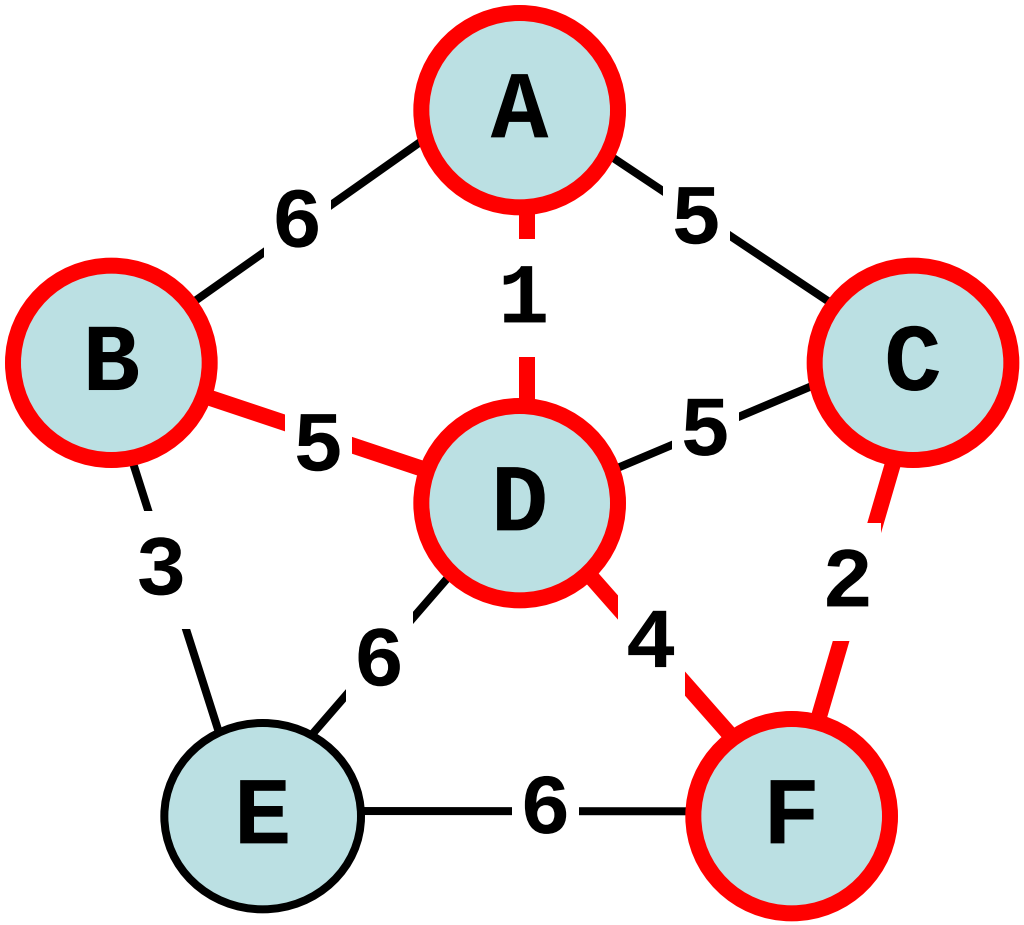
U = {A,D,F}

- **Prim算法**
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



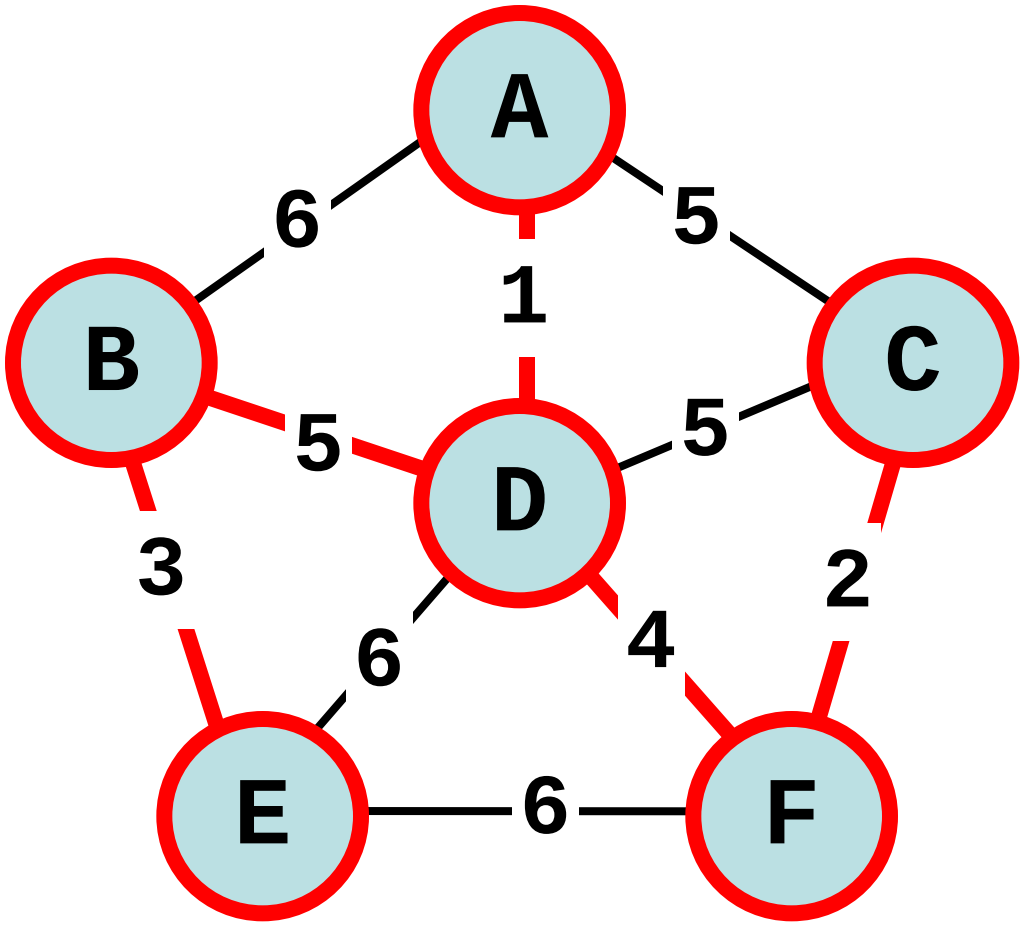
U = {A,D,F,C}

- **Prim算法**
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



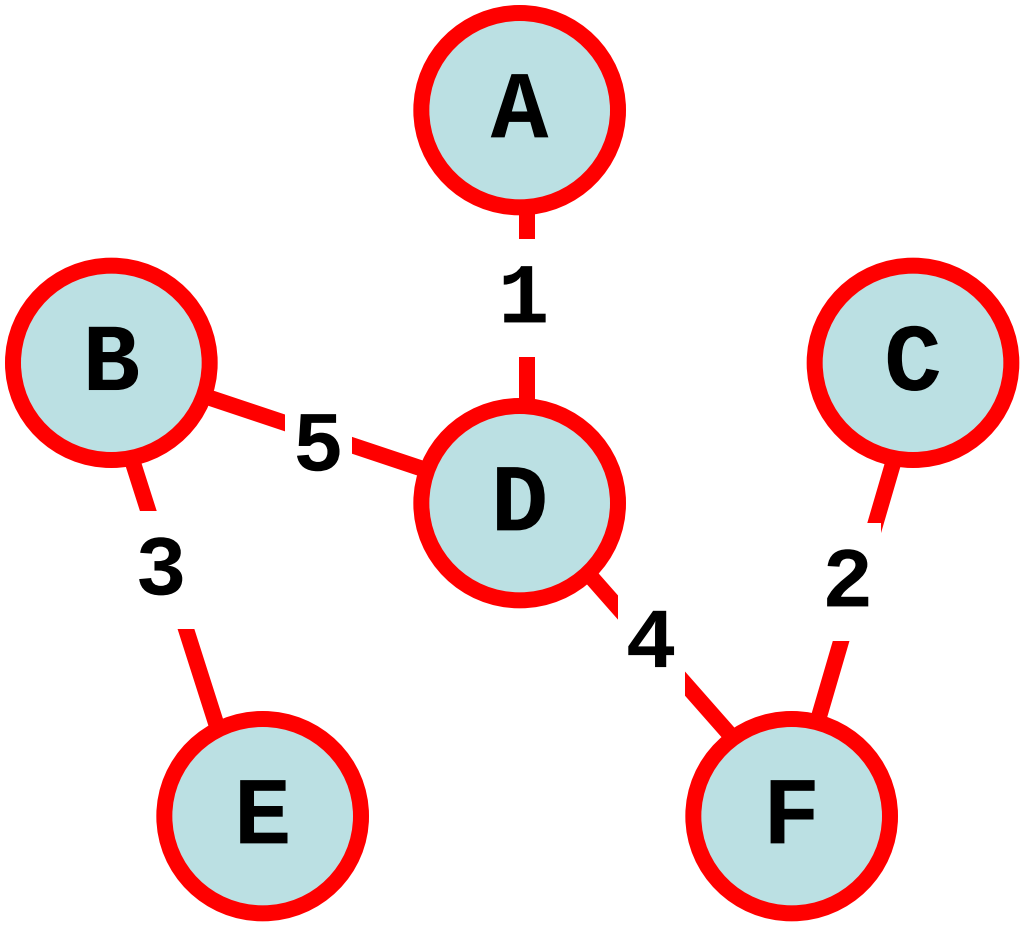
U = {A,D,F,C,B}

- Prim算法
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中



U = {A,D,F,C,B,E}

- Prim算法
  - U为最小生成树中顶点的集合
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中

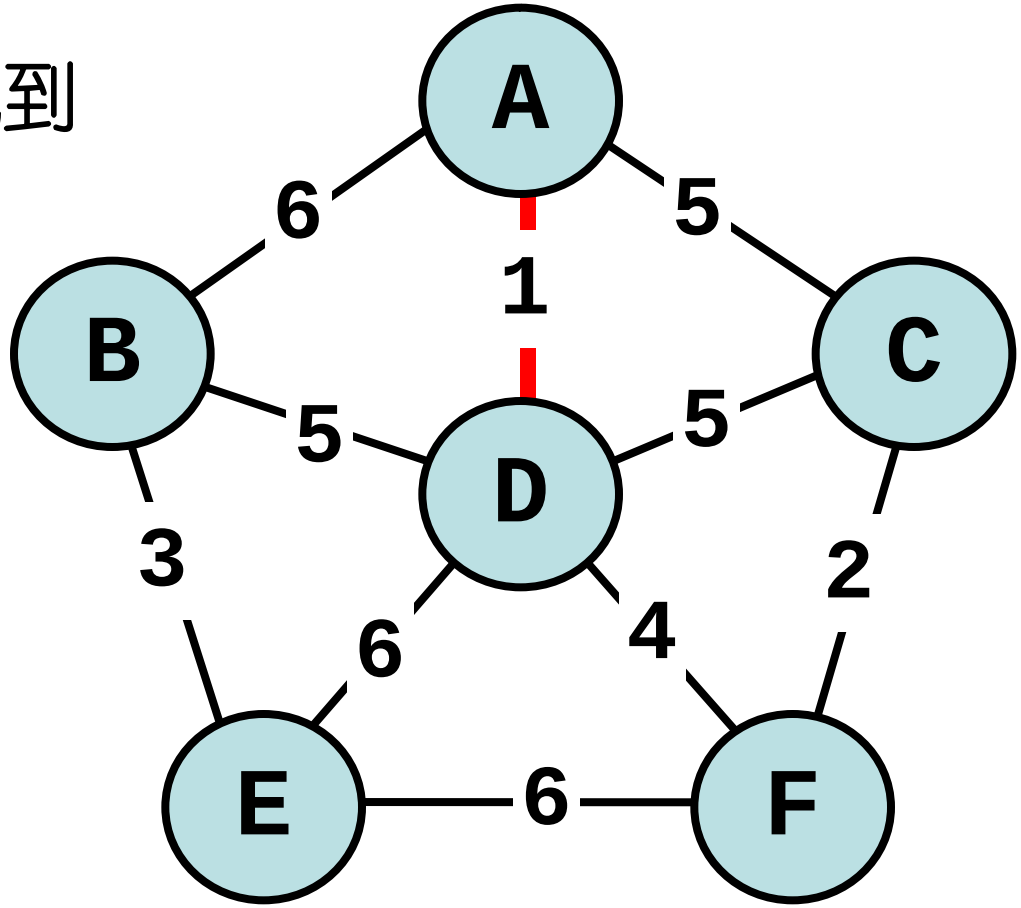


U = {A,D,F,C,B,E}

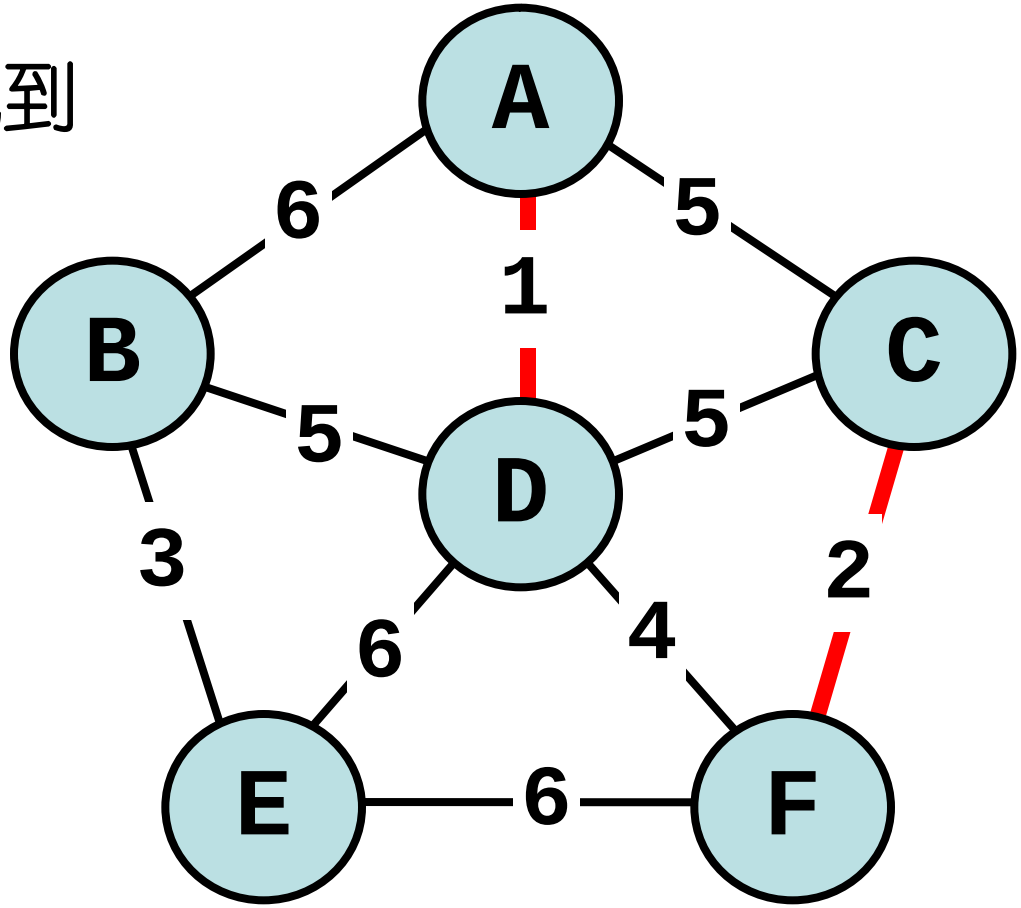
最小生成树

- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 不过要求不能产生回路

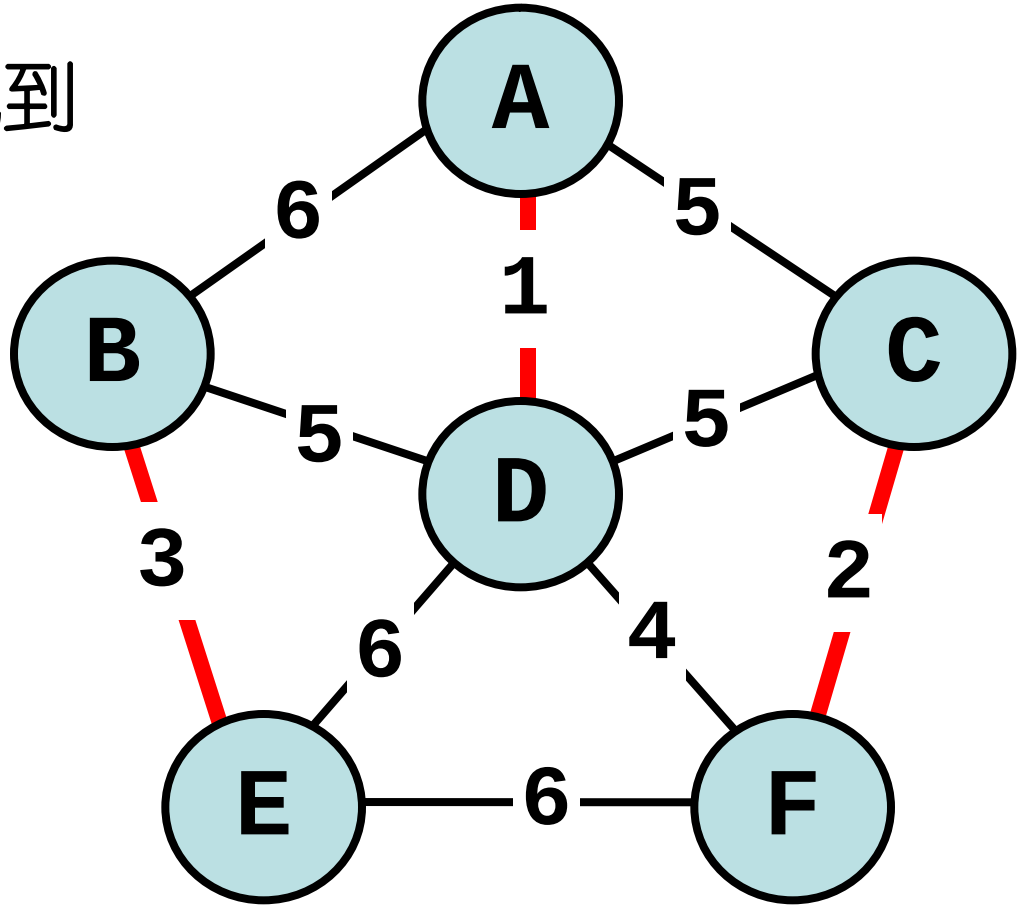
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



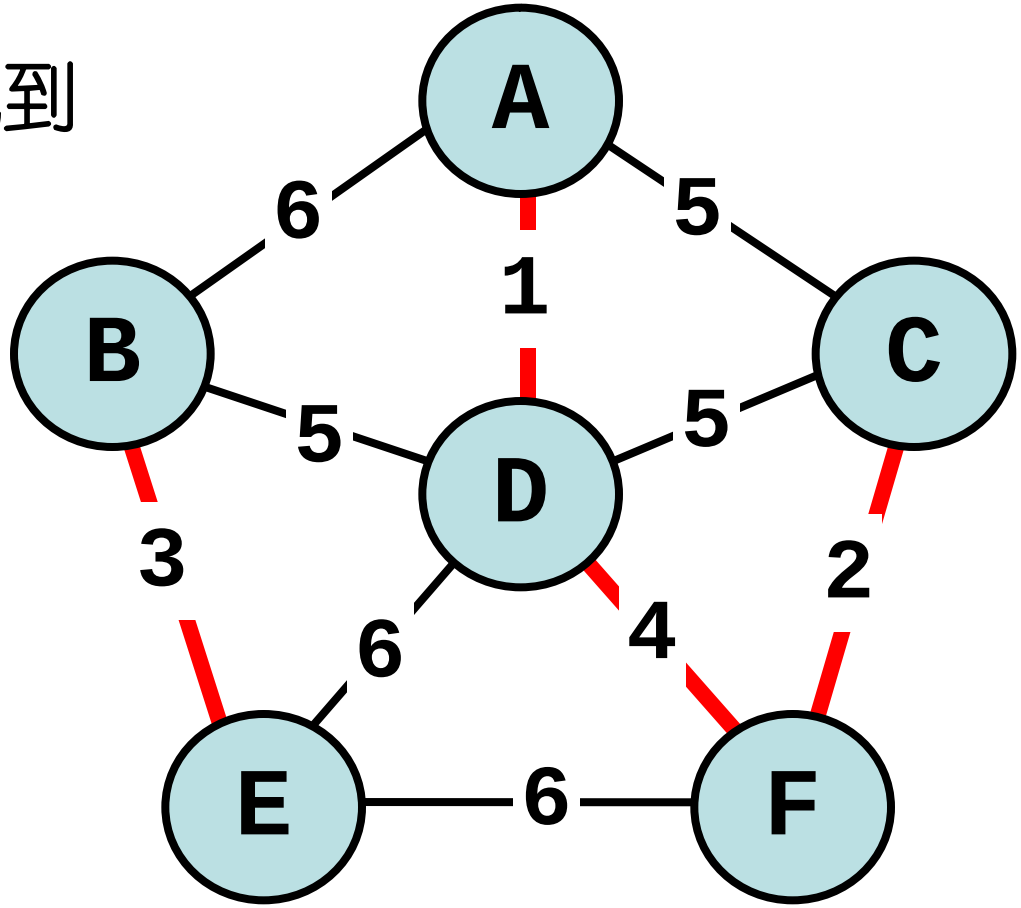
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



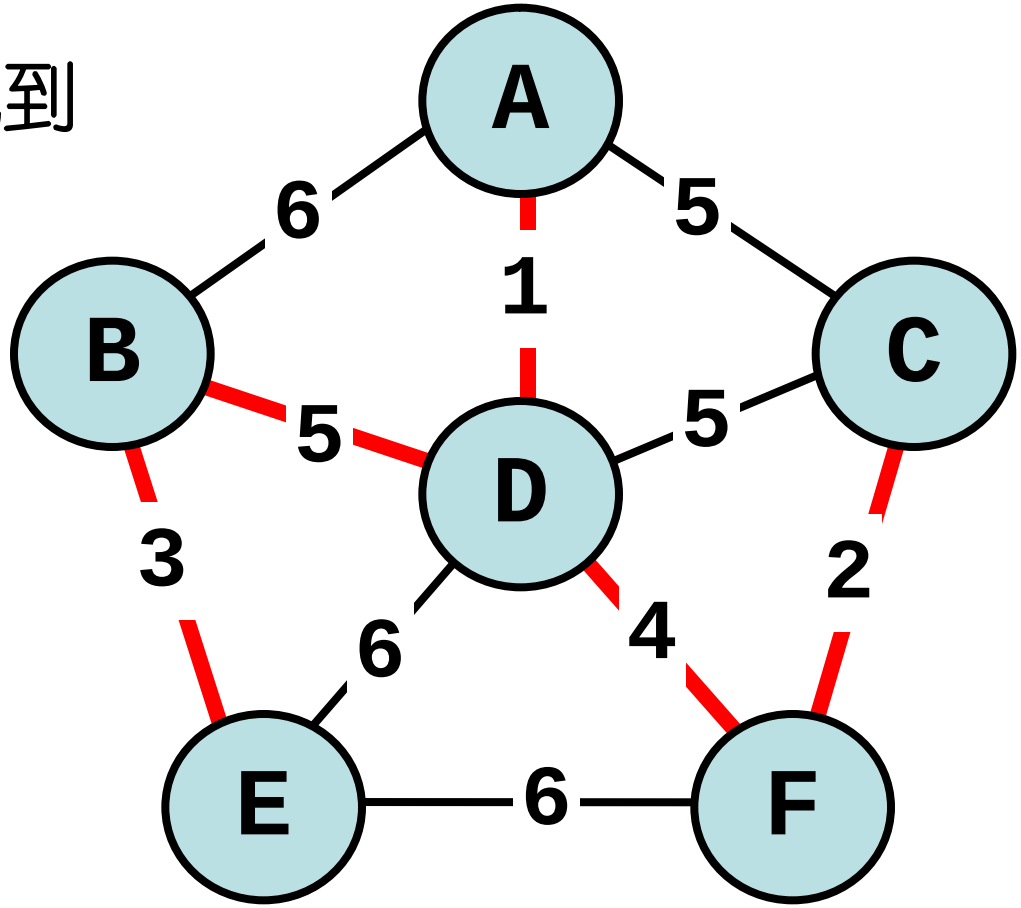
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



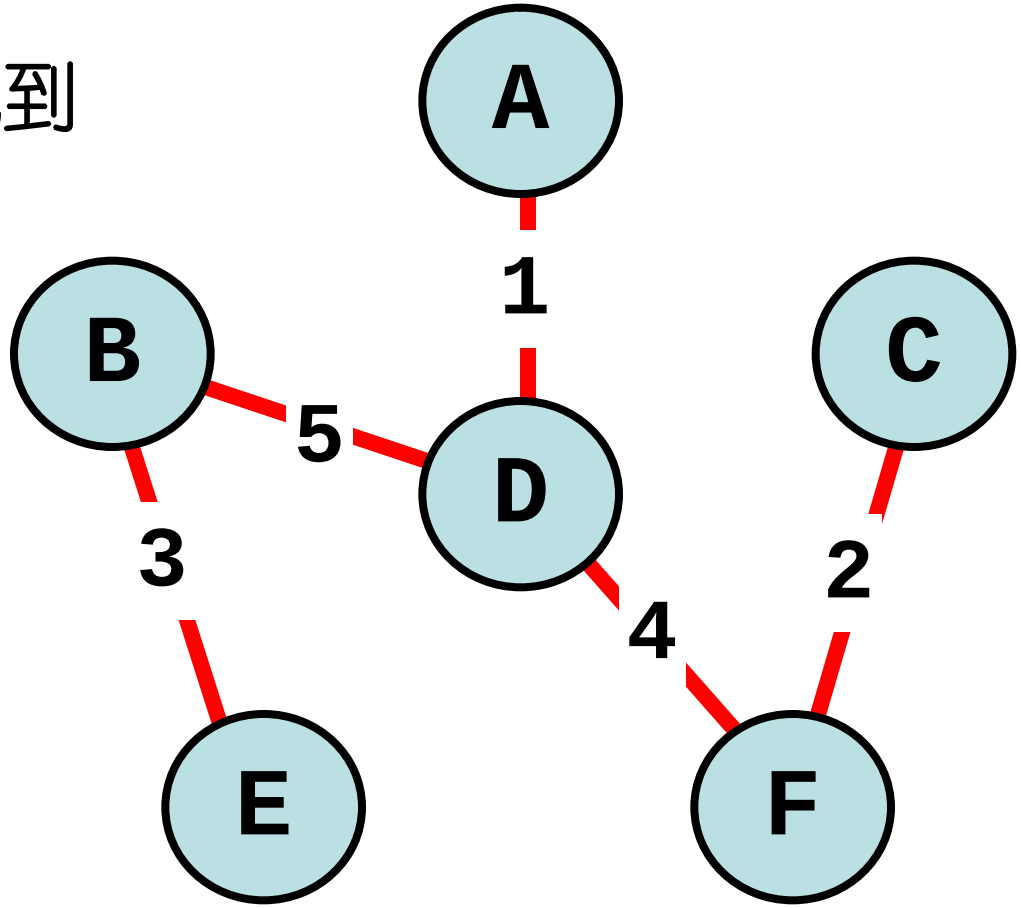
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - ...
  - 但不能产生回路



图的连通性问题

- 本节小结
  - 连通分量：借助遍历来求，从一个顶点出发，凡是能遍历到的就是一个连通分量
  - 生成树：不一定唯一，可以借助遍历来求
  - 最小生成树：
    - 也不一定唯一
    - 算法：**Prim、Kruskal**
- 思考题4
  - 习题集**7.7**

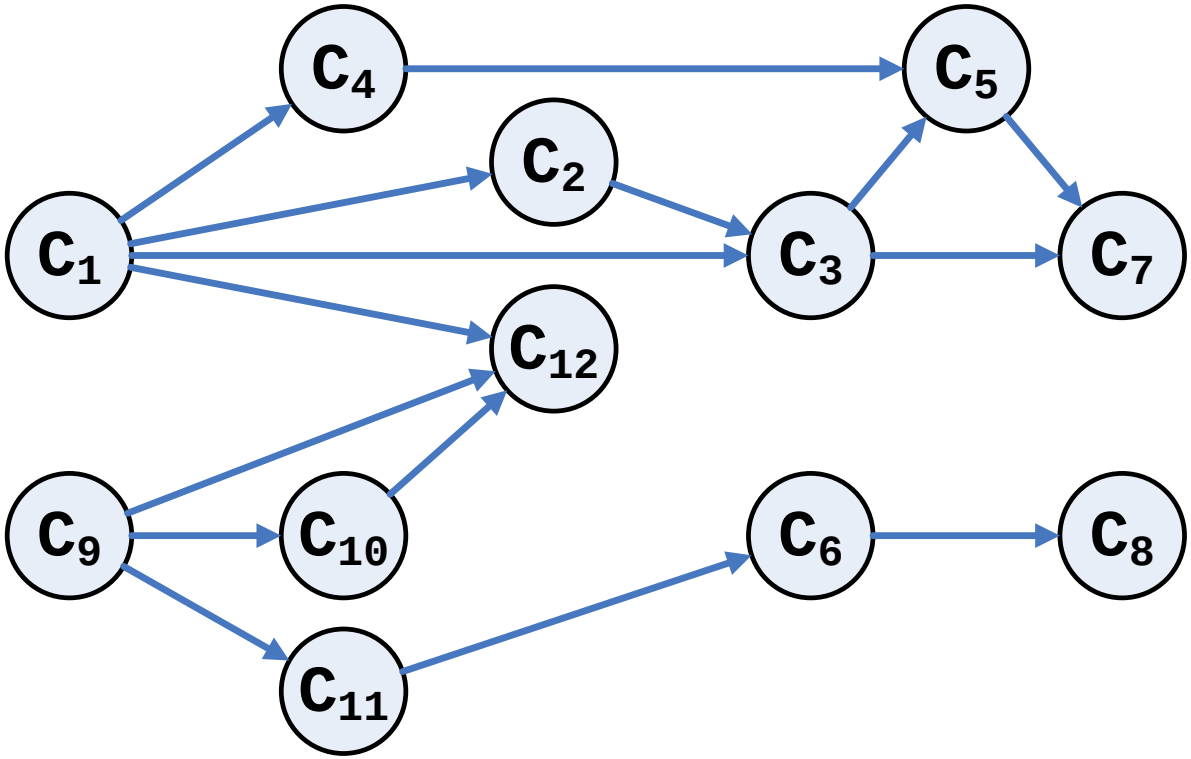
有向无环图的应用：拓扑排序

- 应用：很多课程之间有一定的先后关系限制

课程编号	课程名称	先决条件
C1	程序设计	无
C2	离散数学	C1
C3	数据结构	C1，C2
C4	汇编语言	C1
C5	语言设计分析	C3，C4
C6	计算机原理	C11
C7	编译原理	C5，C3
C8	操作系统	C3，C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C9，C10，C1

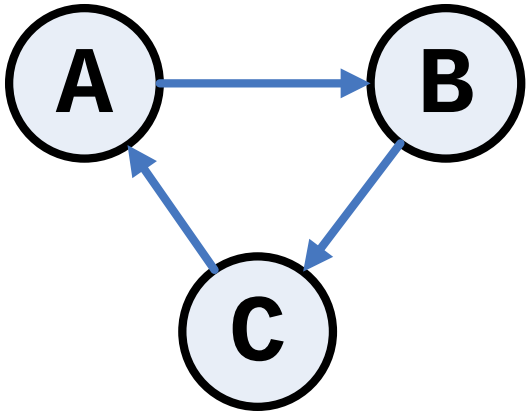
有向无环图的应用：拓扑排序

- 上述关系可以用一个有向无环图描述
  - 顶点表示活动，弧表示先后关系
  - **AOV = Activity On Vertex**



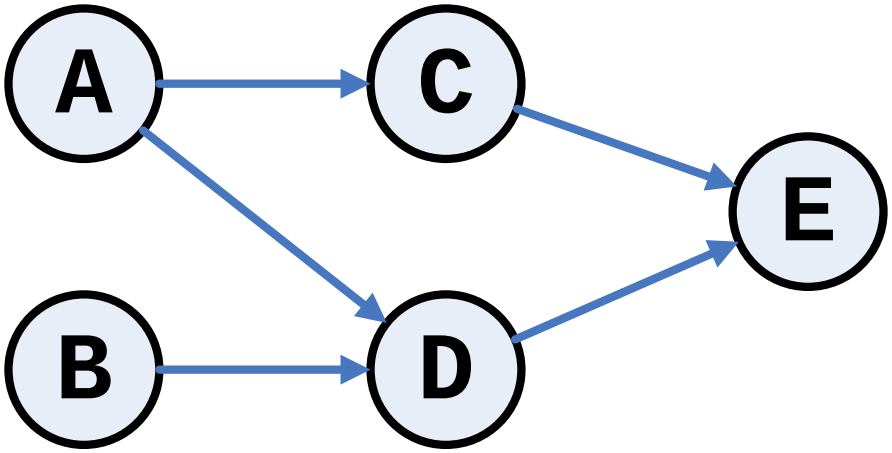
有向无环图的应用：拓扑排序

- 问题：（拓扑排序）
  - 假设学生每学期只上一门课，应该如何安排课程？
  - 即把上述AOV图按照课程的先后关系转换成一个线性的序列
- 注意：
  - 此图必须是有向无环图



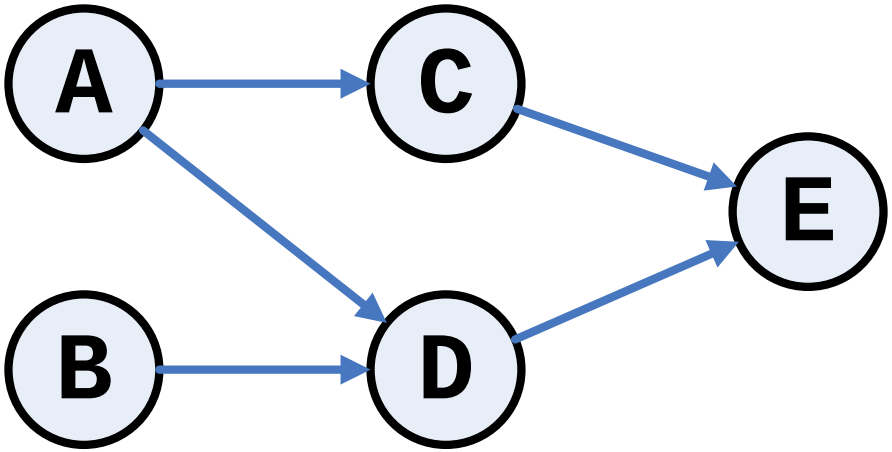
有向无环图的应用：拓扑排序

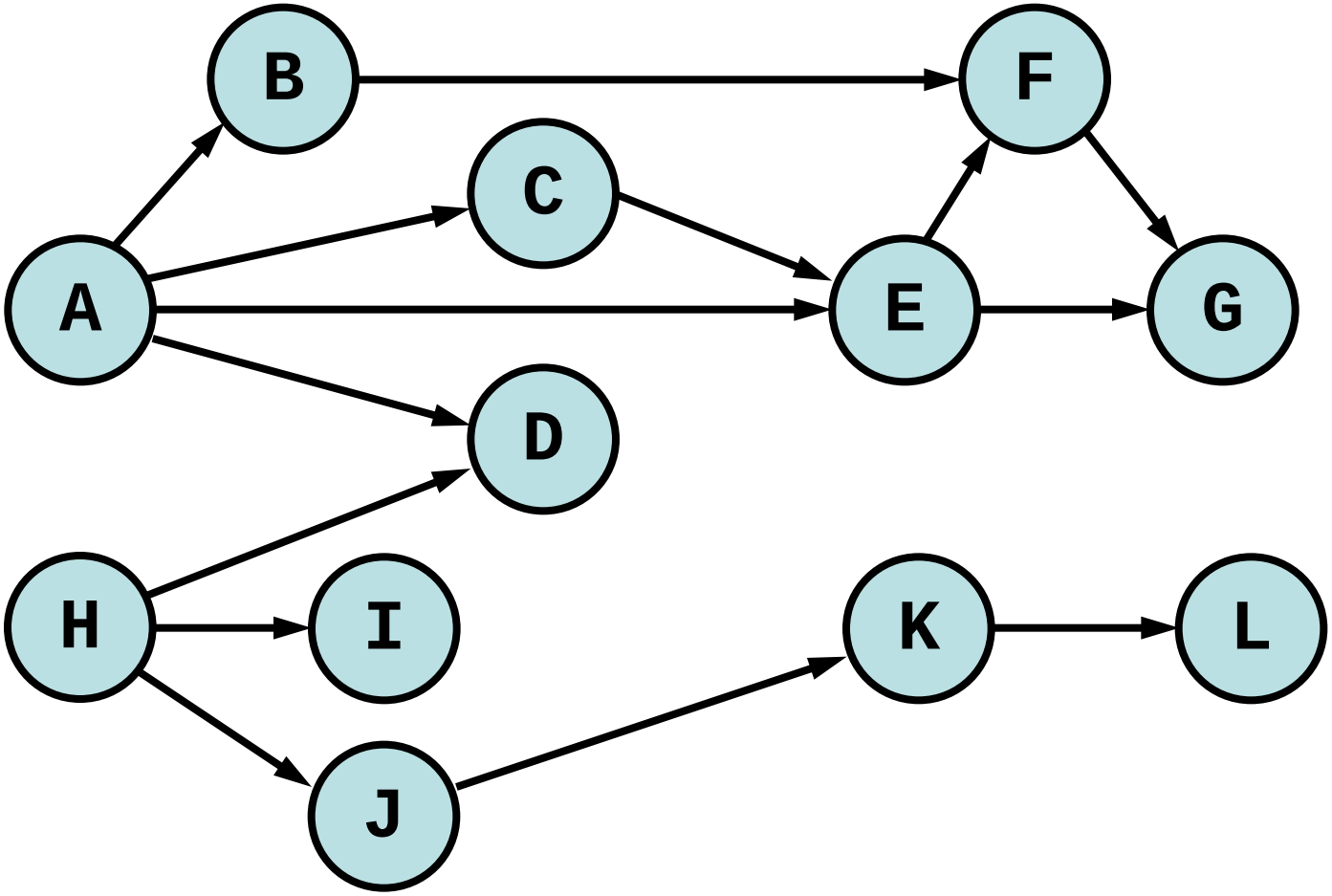
- 思考
  - 弧尾顶点应该在弧头顶点之前输出
    - A在C，D之前、B在D之前，C，D在E之前
  - 没有弧线直接相连的两个顶点顺序不限
    - A和B，C和D先后不限



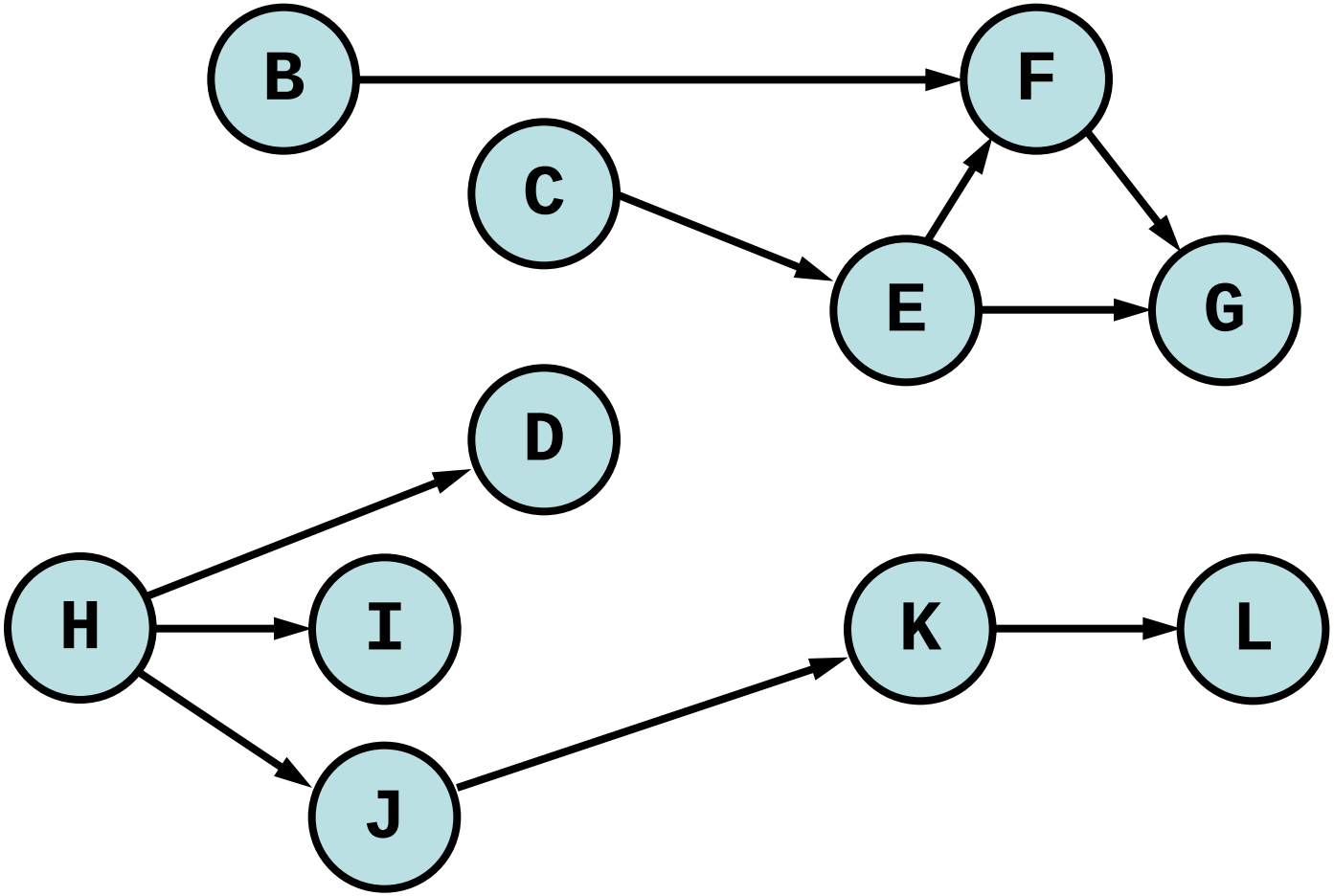
有向无环图的应用：拓扑排序

- 算法
  - 在图中任选一个没有前驱的顶点输出
  - 删除该顶点及所有以它为弧尾的弧
  - 重复，直至全部顶点输出或图中不存在无前驱的顶点为止

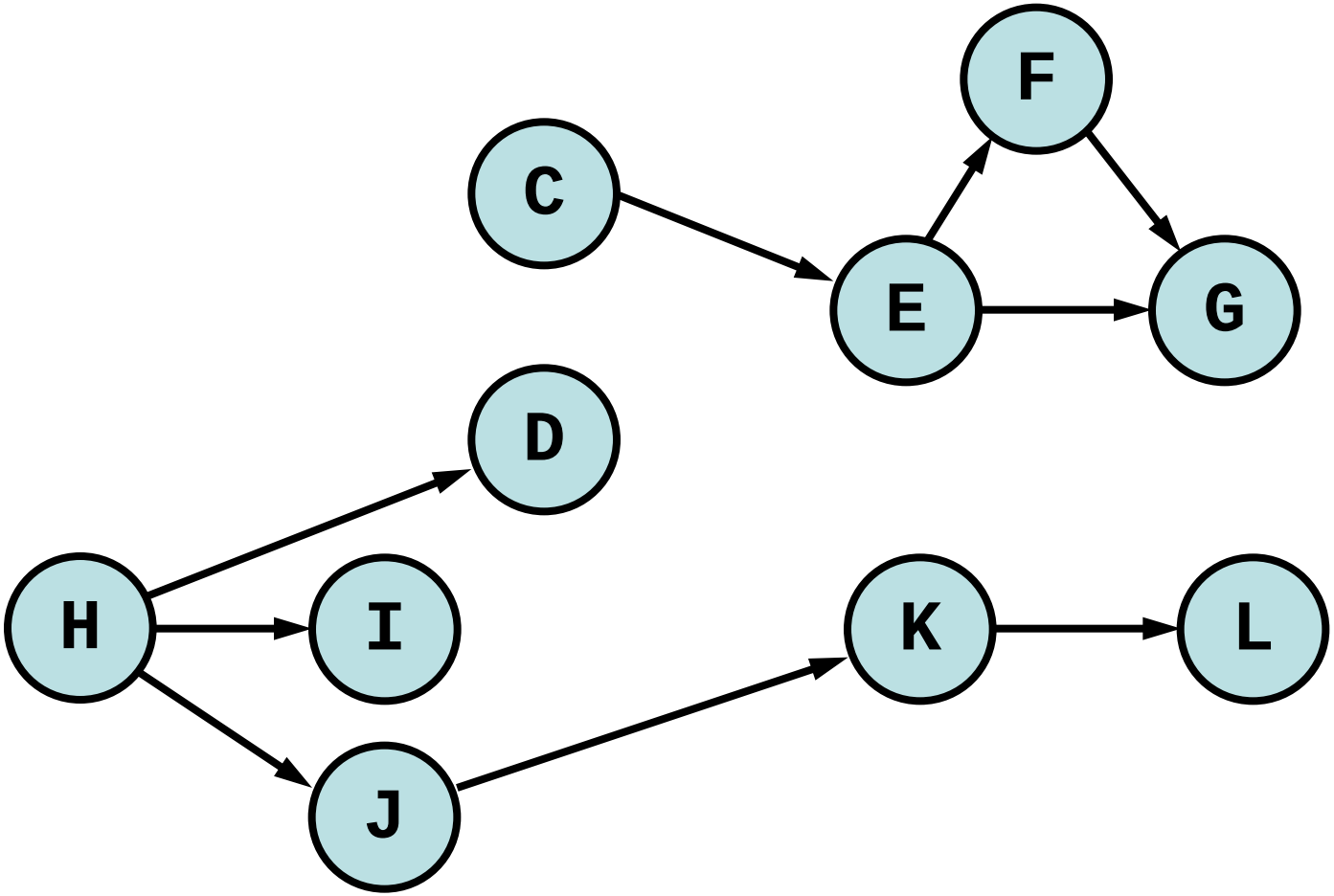




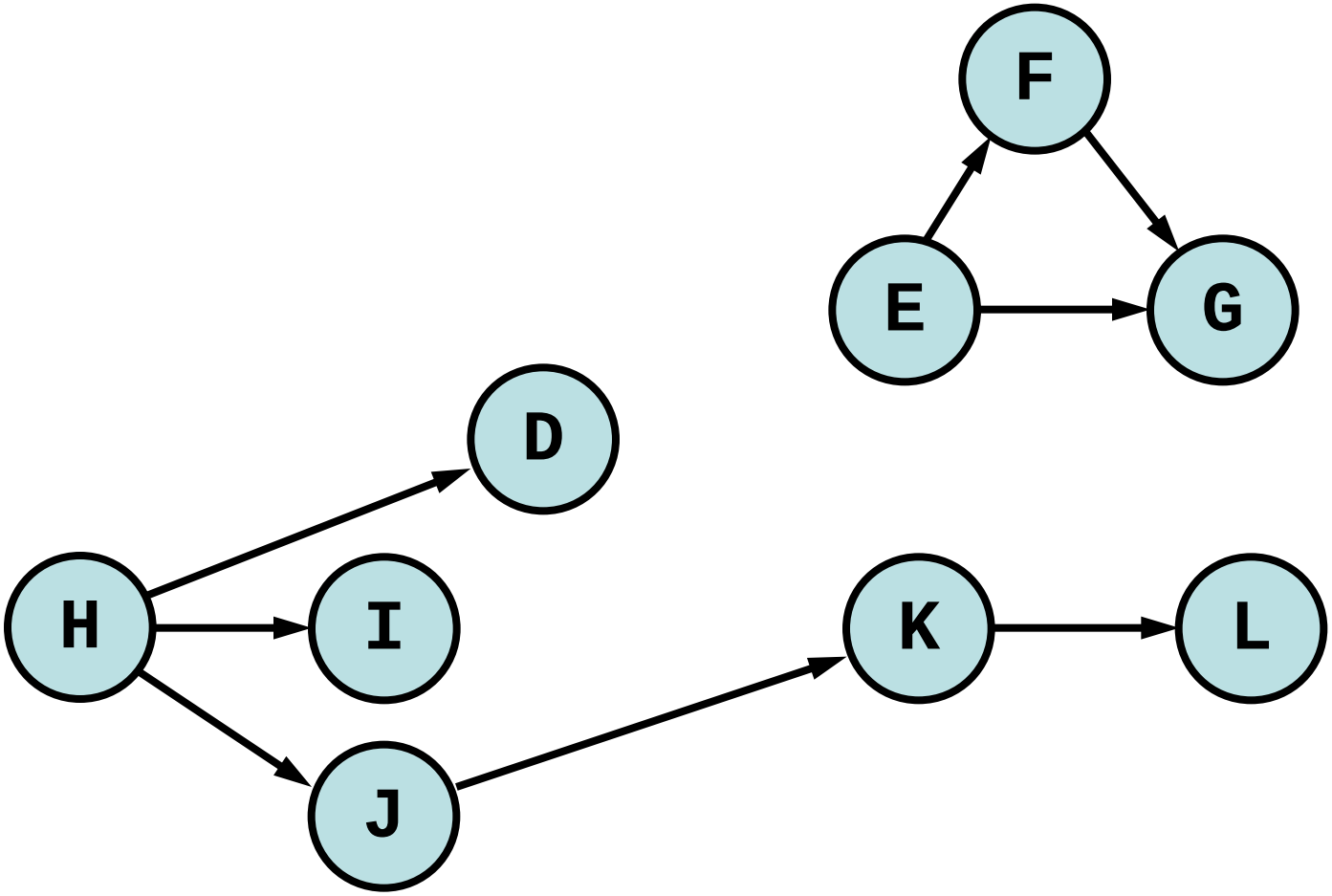
- 无前驱的顶点有：A，H
- 输出：A
- 删除A和以A为弧尾的弧



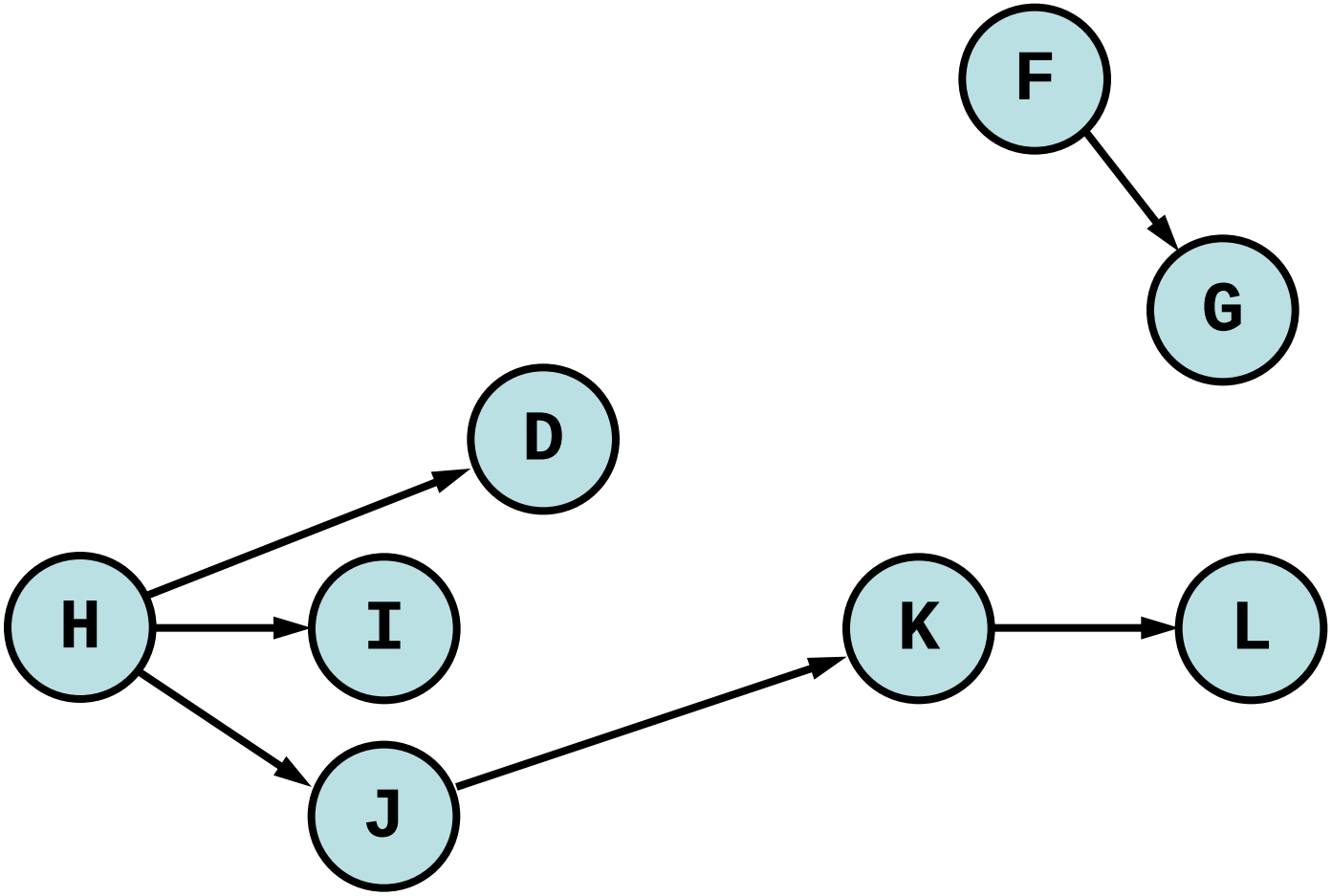
- 无前驱的顶点有：B， H
- 输出：A,B
- 删除B和以B为弧尾的弧



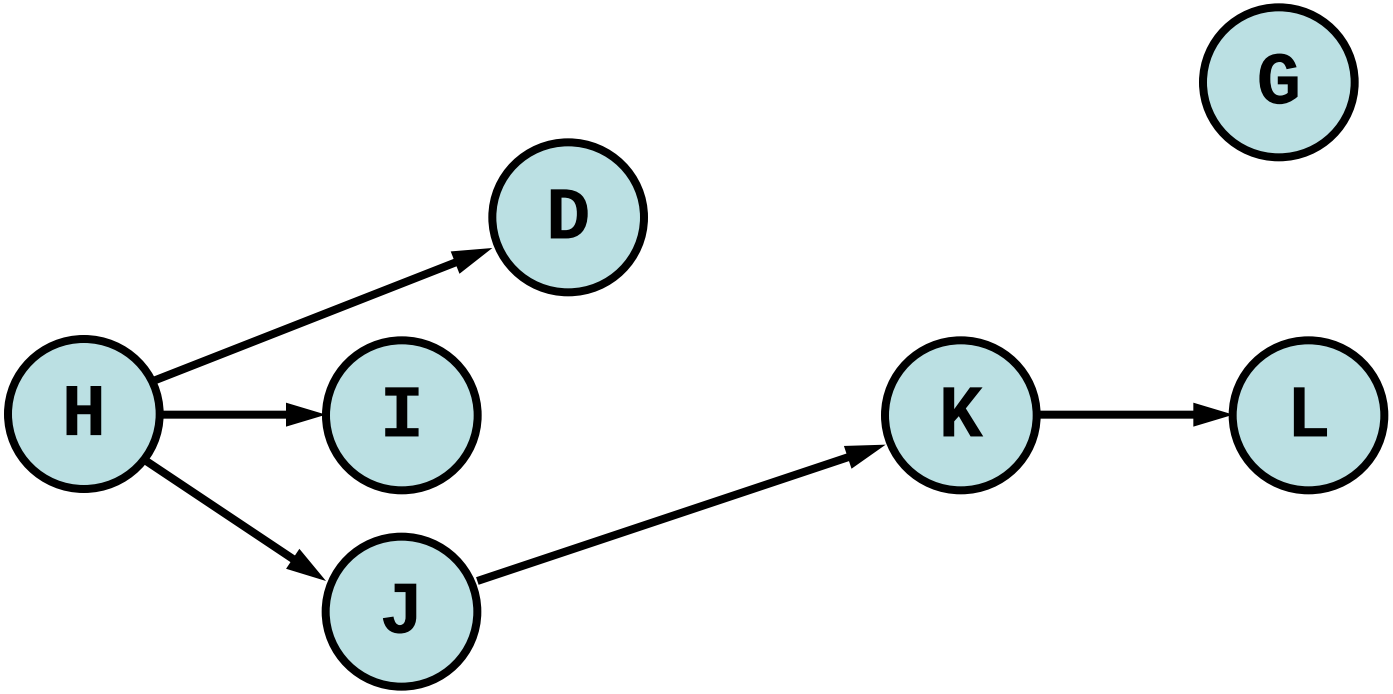
- 无前驱的顶点有：C， H
- 输出：A,B,C
- 删除C和以C为弧尾的弧



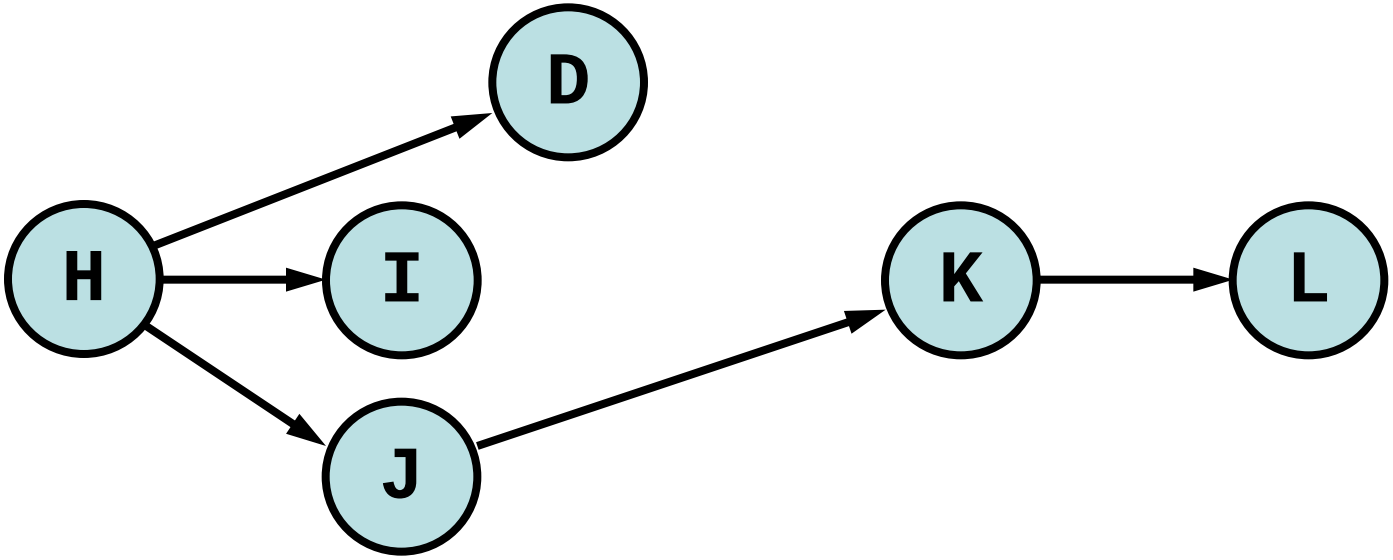
- 无前驱的顶点有：**E， H**
- 输出：**A,B,C,E**
- 删除**E**和以**E**为弧尾的弧



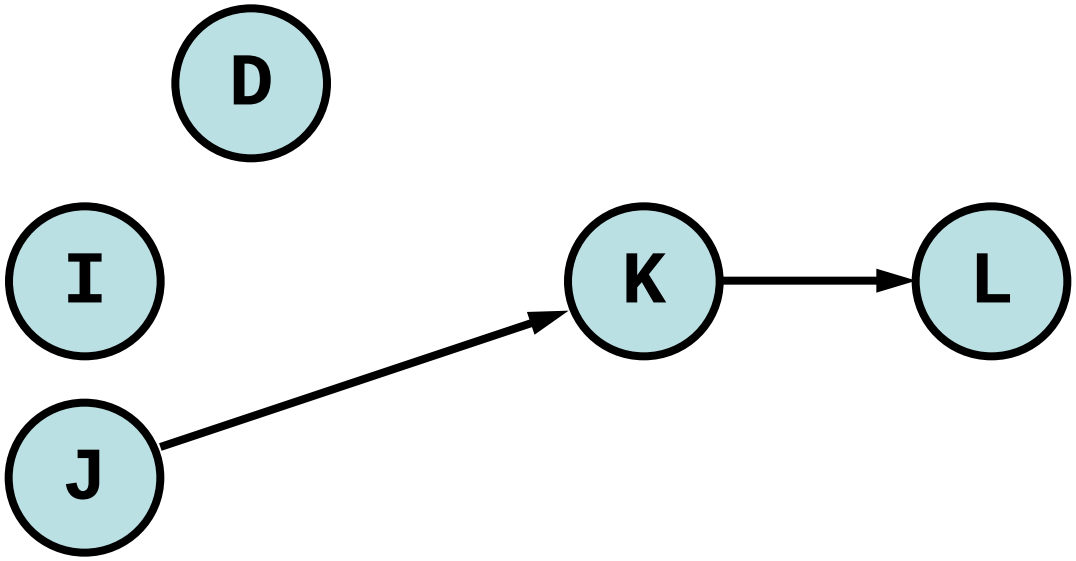
- 无前驱的顶点有：F， H
- 输出：A,B,C,E,F
- 删除F和以F为弧尾的弧



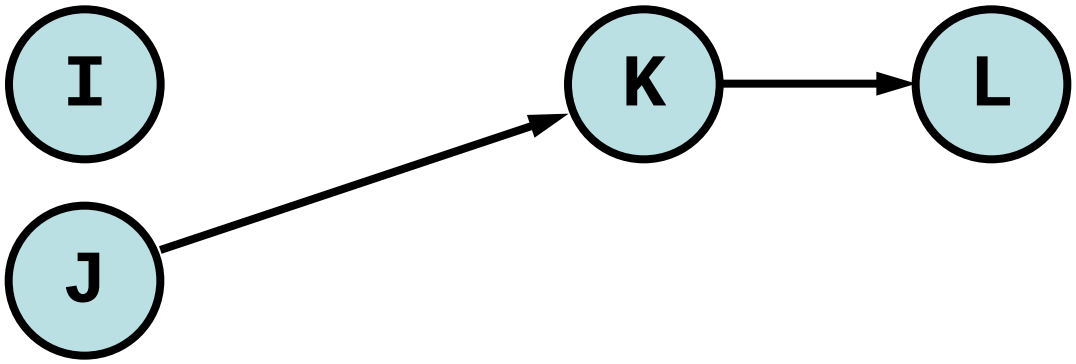
- 无前驱的顶点有：G
- 输出：A,B,C,E,F,G
- 删除G和以G为弧尾的弧



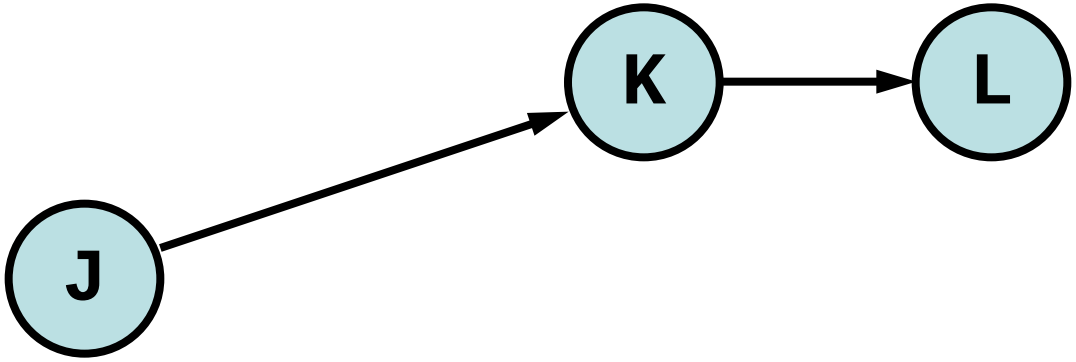
- 无前驱的顶点有：H
- 输出：A,B,C,E,F,G,H
- 删除H和以H为弧尾的弧



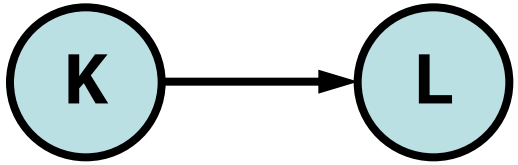
- 无前驱的顶点有：D, I, J
- 输出：A,B,C,E,F,G,H,D
- 删除D和以D为弧尾的弧



- 无前驱的顶点有：I, J
- 输出：A,B,C,E,F,G,H,D,I
- 删除I和以I为弧尾的弧



- 无前驱的顶点有：J
- 输出：A,B,C,E,F,G,H,D,I,J
- 删除J和以J为弧尾的弧

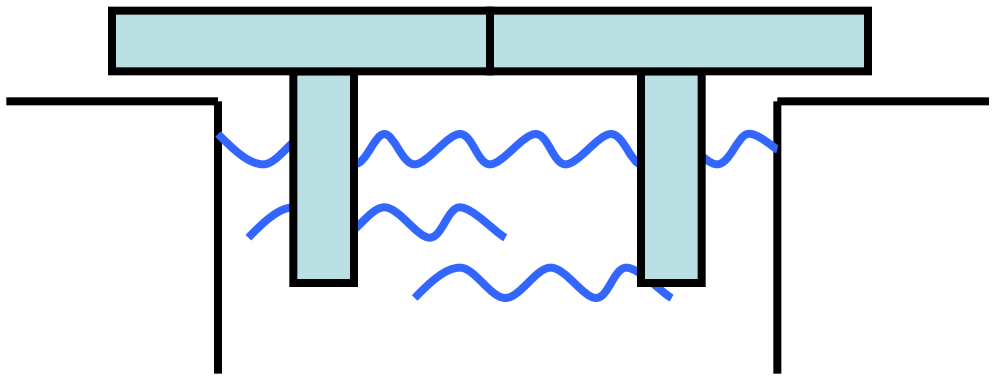


- 无前驱的顶点有：K
- 输出：A,B,C,E,F,G,H,D,I,J,K
- 删除K和以K为弧尾的弧

- 无前驱的顶点有：L
- 输出：A,B,C,E,F,G,H,D,I,J,K,L
- 删除L和以L为弧尾的弧

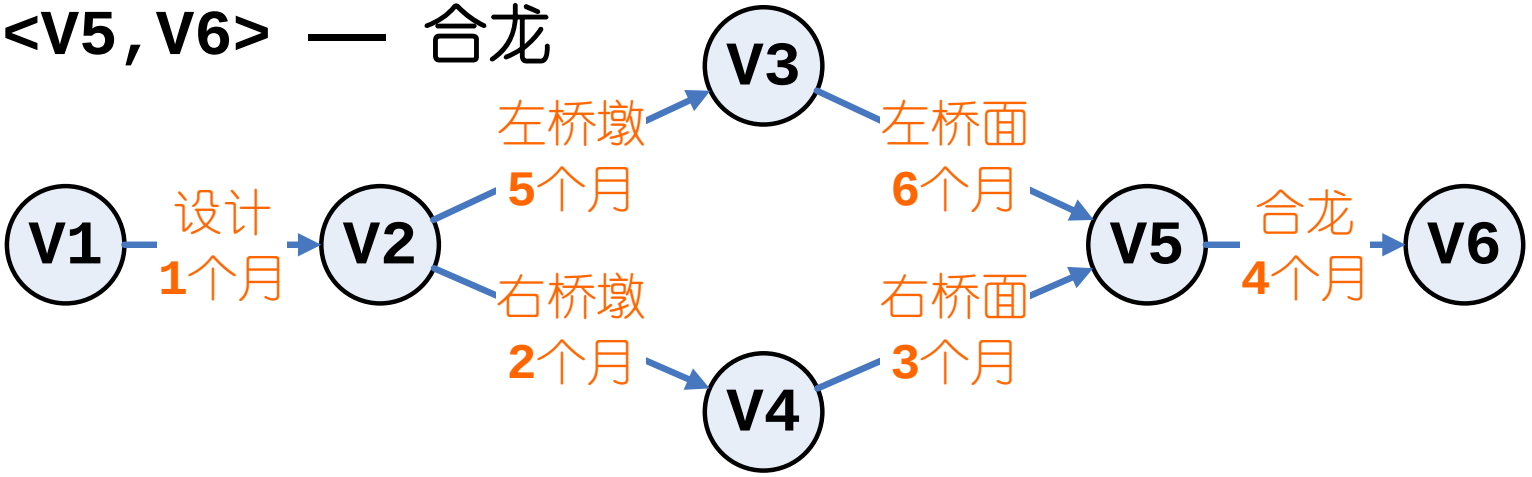
有向无环图的应用：关键路径

- 应用
  - 假设有一桥梁建设工程，先进行设计，然后左右两岸同时开工（独立），分别需要先架设桥墩，再架设桥面，最后再将左右桥面合龙



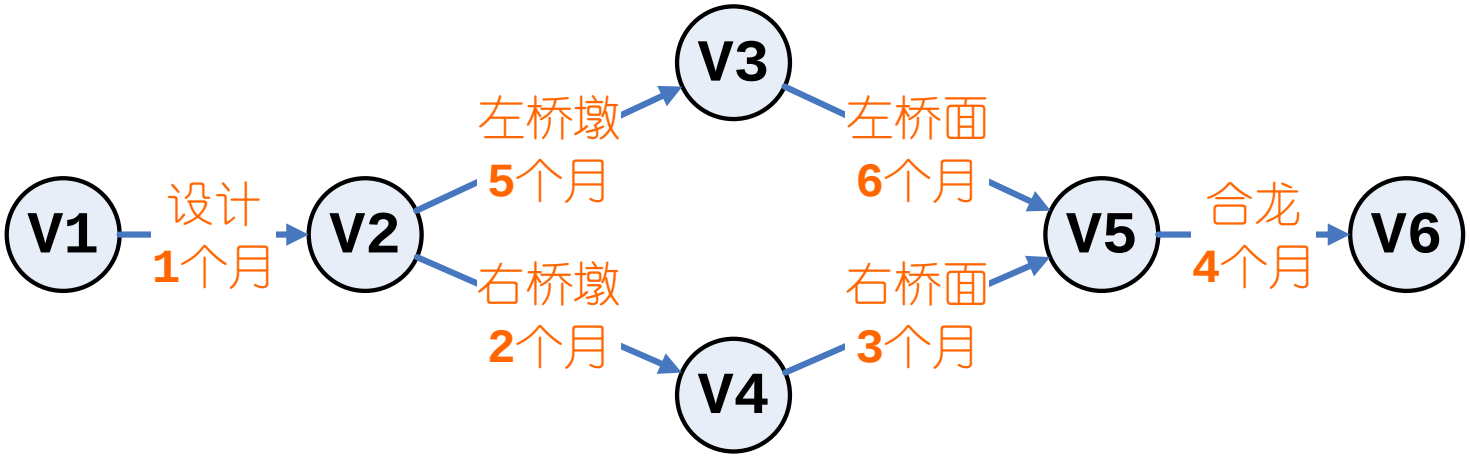
有向无环图的应用：关键路径

- 我们可以用一个图来表示上述过程：
  - 弧代表活动（费时）：
    - <V1,V2> —— 设计
    - <V2,V3> —— 左岸桥墩的架设
    - <V3,V5> —— 左岸桥面的架设
    - <V2,V4> —— 右岸桥墩的架设
    - <V4,V5> —— 右岸桥面的架设
    - <V5,V6> —— 合龙



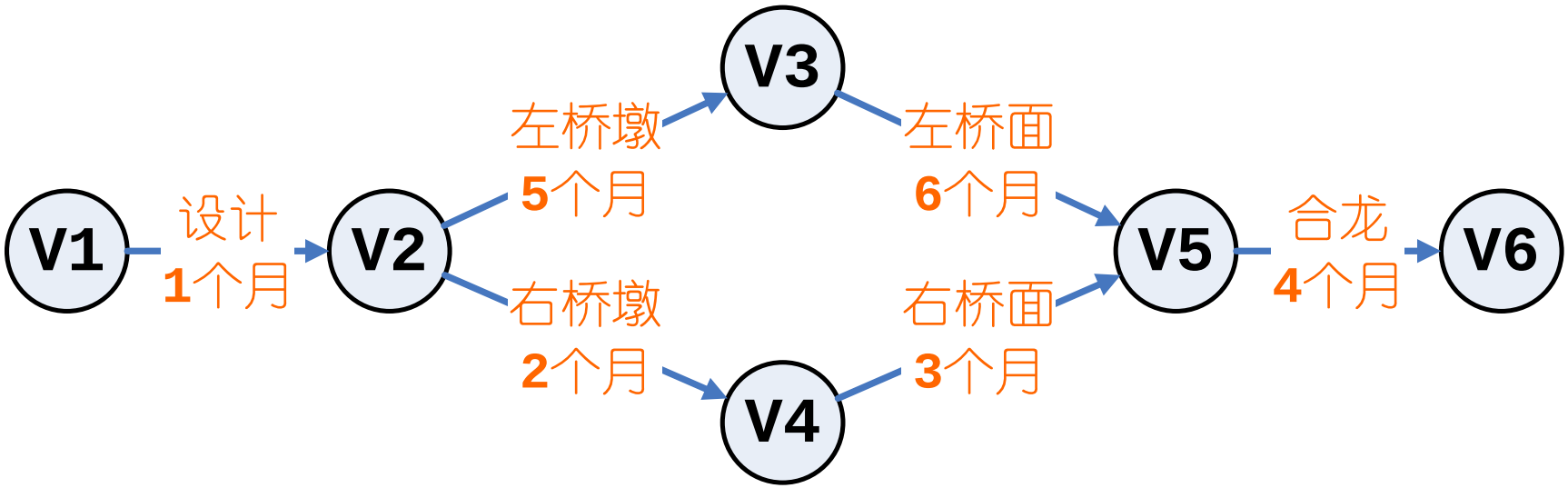
有向无环图的应用：关键路径

- 顶点代表事件（瞬间）：
  - **V1** —— 开始设计
  - **V2** —— 设计完成，开始动工
  - **V3** —— 左岸桥墩完工，桥面开工
  - **V4** —— 右岸桥墩完工，桥面开工
  - **V5** —— 左右桥面完工，开始合龙
  - **V6** —— 合龙完成，整个工程竣工



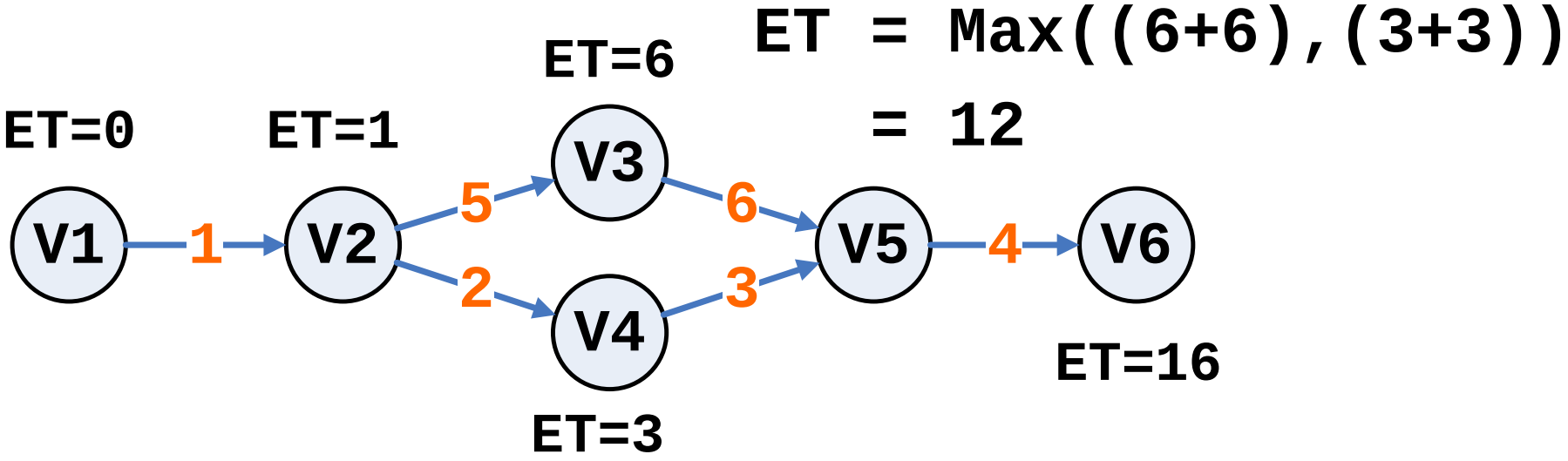
有向无环图的应用：关键路径

- 权值代表活动的所需的时间**duration of time**
- 这样的图称作AOE (Activity On Edge)



有向无环图的应用：关键路径

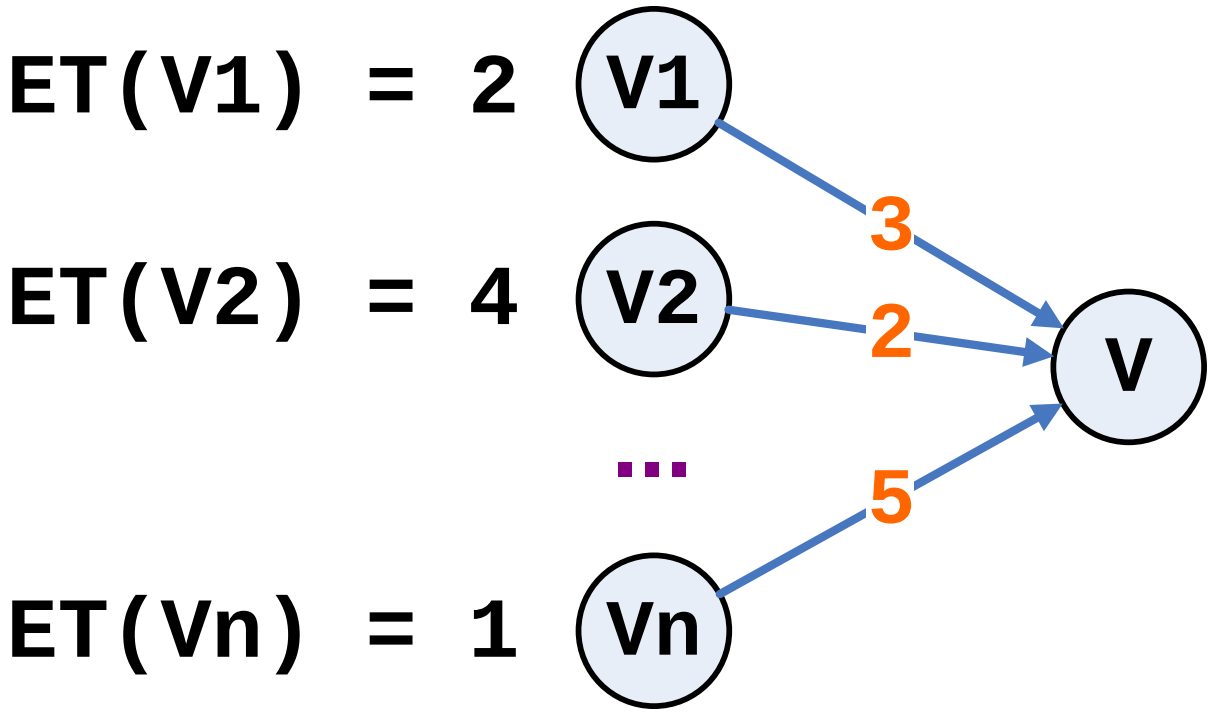
- 事件的最早发生时间(Earliest Time)
  - 假设V1从第0天开始
  - V2、V3、V4最早要第几天才能开始？
  - V5最早要第几天才能开始？
  - V6最早要第几天才能开始？



有向无环图的应用：关键路径

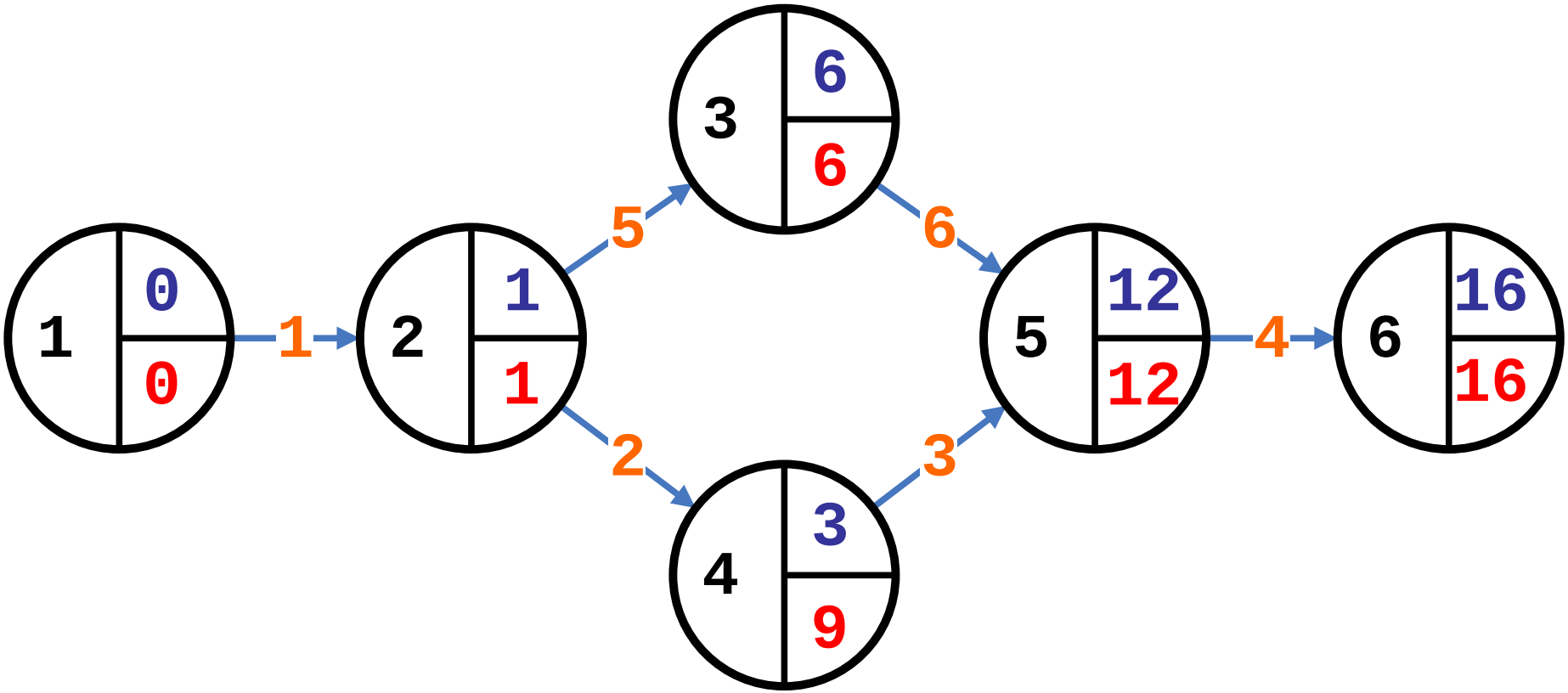
- 事件的最早发生时间(Earliest Time)

$$ET(V) = \max_i(ET(V_i) + dut(i, V))$$



有向无环图的应用：关键路径

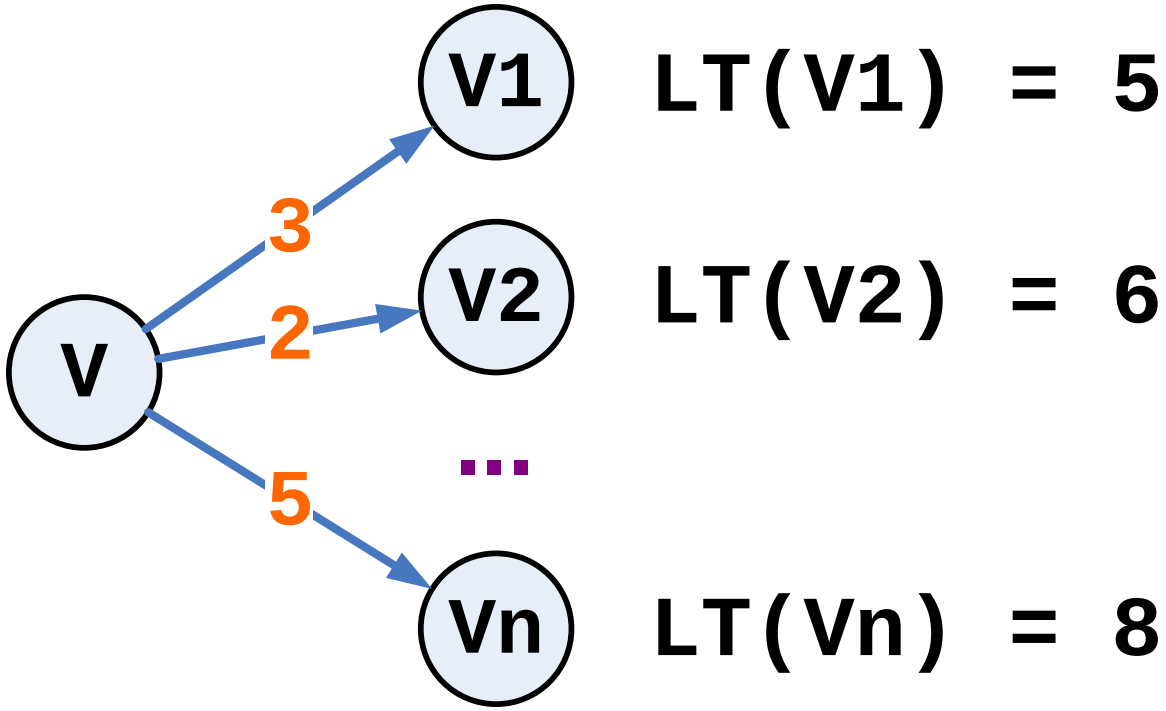
- 事件的最晚发生时间(Latest Time)



有向无环图的应用：关键路径

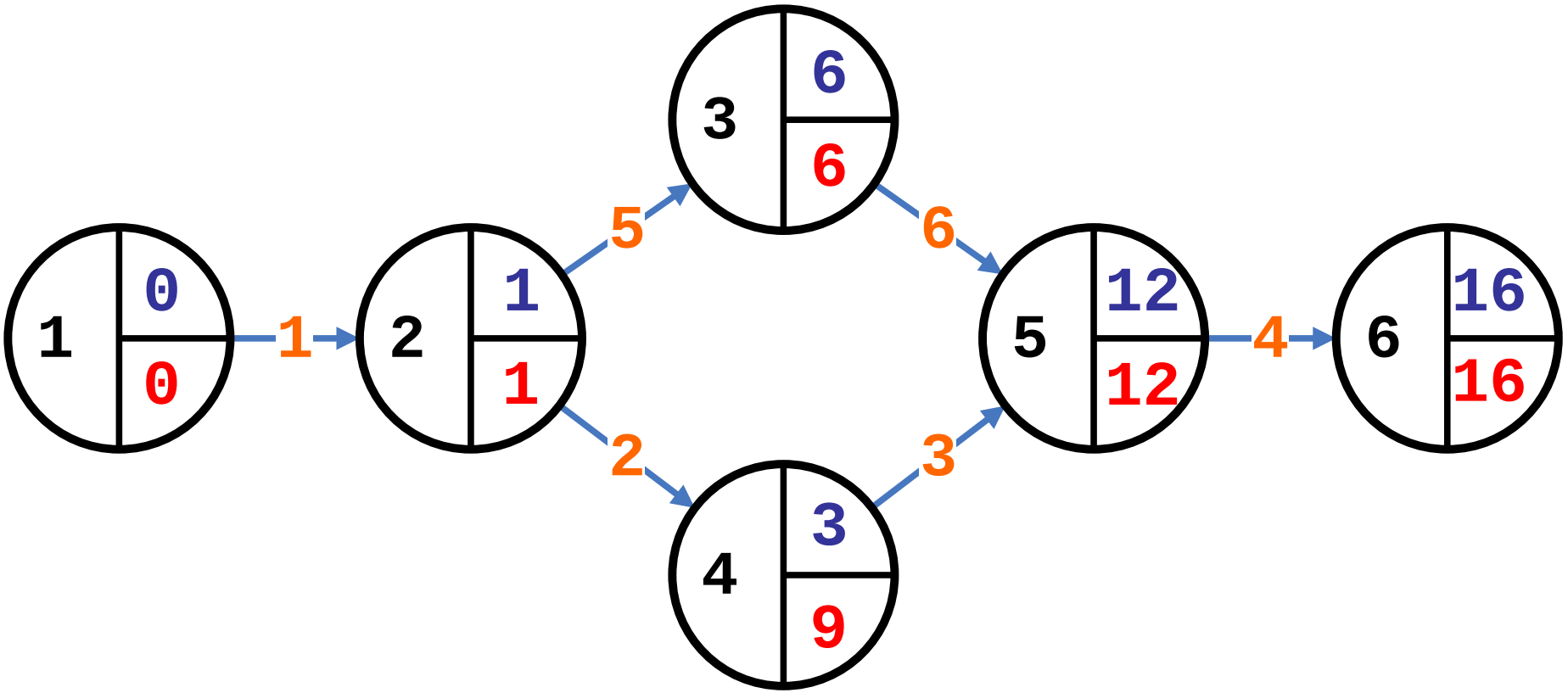
- 事件的最晚发生时间(Latest Time)

$$LT(V) = \min_{i}(LT(i) - dut(V,i))$$



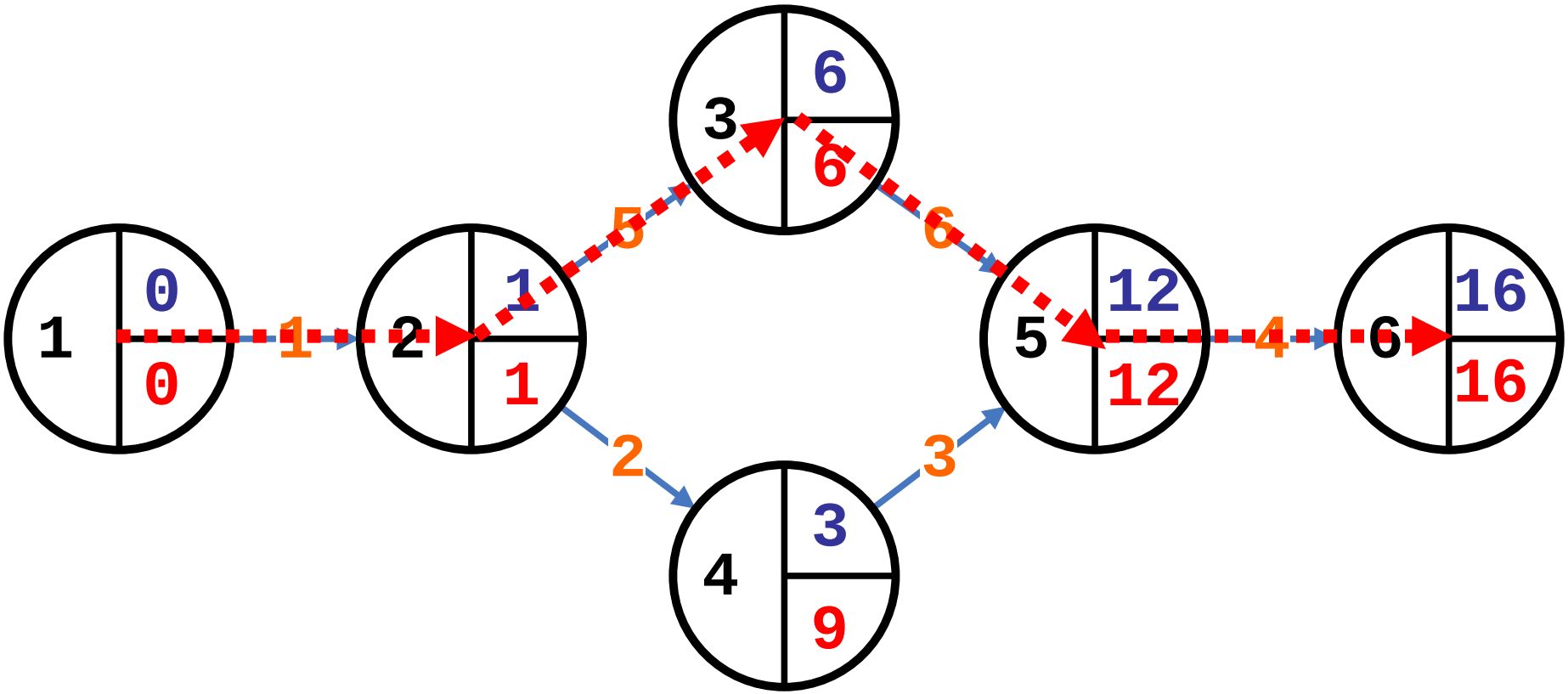
有向无环图的应用：关键路径

- 事件的机动时间(Slack Time)
  - ST(V) = LT(V) – ET(V)



有向无环图的应用：关键路径

- 关键路径
  - 机动时间为0的事件组成

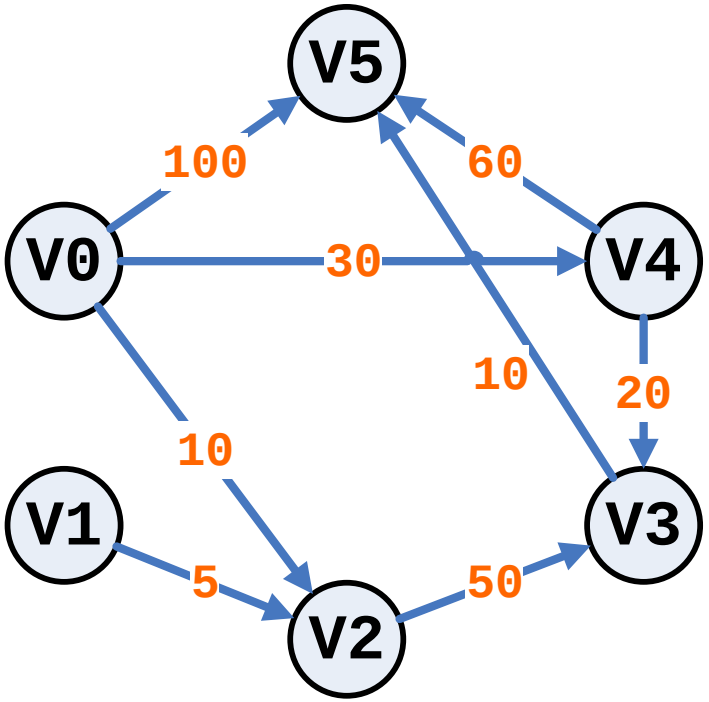


有向无环图的应用

- 本节小结
  - 拓扑排序
    - 比较简单
    - 了解即可
  - 关键路径
    - 比较麻烦
    - 手工掌握
- 思考题5
  - 习题集7.10

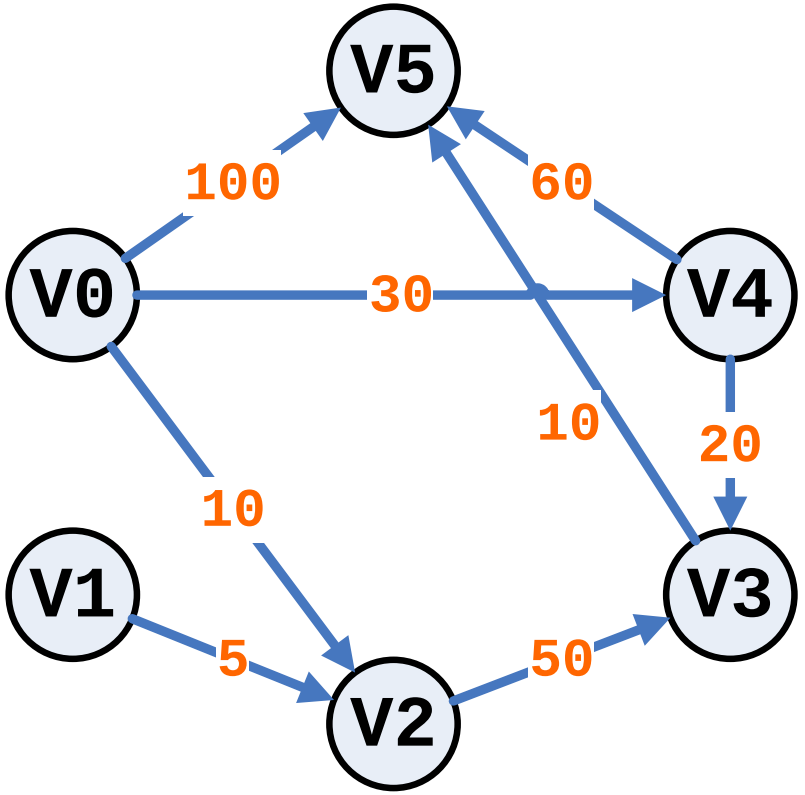
最短路径问题

- 最短路径问题
  - 从一个顶点出发，求出去往其它所有顶点的最短路径及其长度
  - 求出任意两个顶点的最短路径及其长度



最短路径问题：Dijkstra算法

- **Dijkstra算法**
  - 求出从一个顶点出发到其它所有顶点的最短路径及其长度



最短路径问题：Dijkstra算法

- 算法
  - 数据结构
    - **S = Set**：已知最短路径的目的顶点集合
    - **D = Distance**：起点到终点的距离
    - **P = Path**：终点的前一个顶点
  - 初始化
    - **S = {V0}**
    - 对于所有顶点**v**
      - 若存在弧**<V0,V>, D(v) = arcs(V0,V)**
      - 否则 **D(v) = ∞**

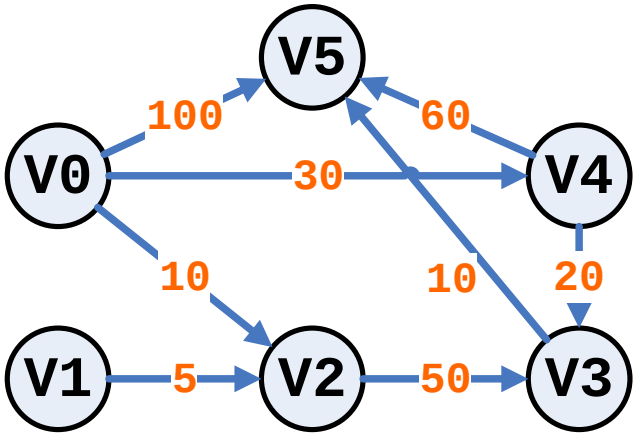
最短路径问题：Dijkstra算法

- 循环
  - 从不在S的顶点中查找D(w)最小的顶点w
  - 把w加入S，即w为已知最短路径
  - 更新w的所有邻接点v的D(v)

```
if( D(w)+c(w,v) < D(v) )
    D(v) = D(w)+arcs(w,v);
```

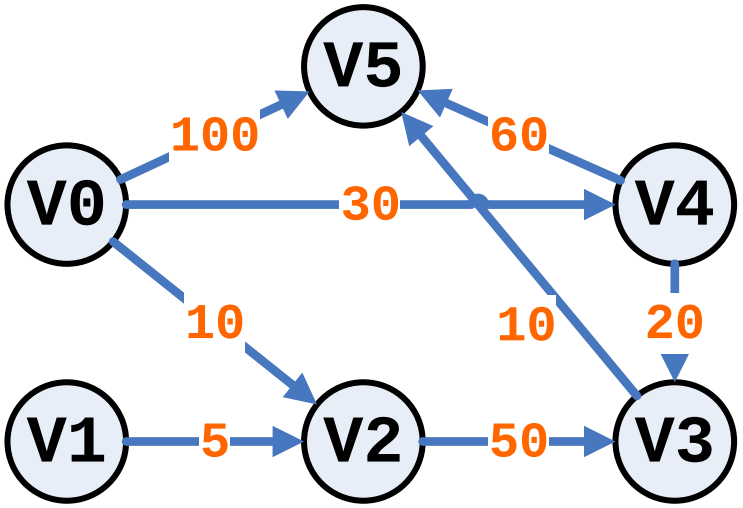
  - 循环，直到所有顶点都加入S

终点	i=1	i=2	i=3	i=4	i=5
V1	∞				
V2	10				
V3	∞				
V4	30				
V5	100				
S	V0				



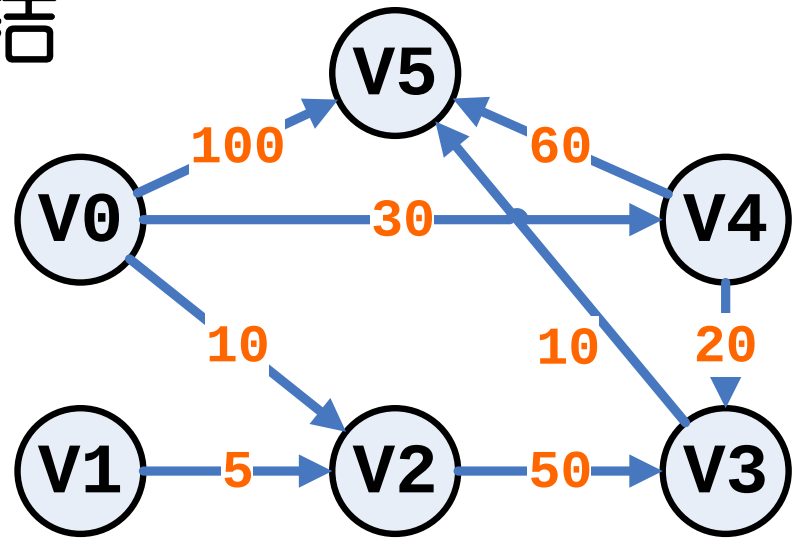
最短路径问题：Dijkstra算法

- **Dijkstra算法复杂度**
  - 时间复杂度 = $O(n^2)$
- 怎么求任意两点之间的最短距离？
  - 方法一：以每一个顶点作为起点调用Dijkstra算法



最短路径问题：Floyd算法

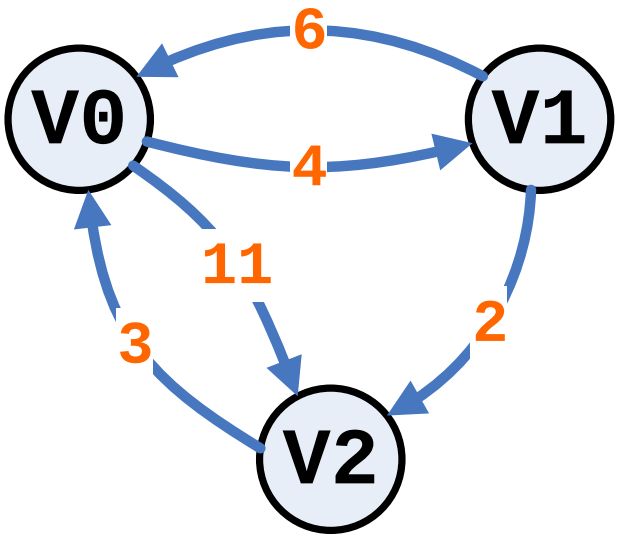
- 怎么求任意两点之间的最点距离？
  - 方法一：以每一个顶点作为起点调用Dijkstra算法，时间复杂度 = $O(n^3)$
  - 方法二：Floyd算法，时间复杂度也是$O(n^3)$，但是更简洁



最短路径问题：Floyd算法

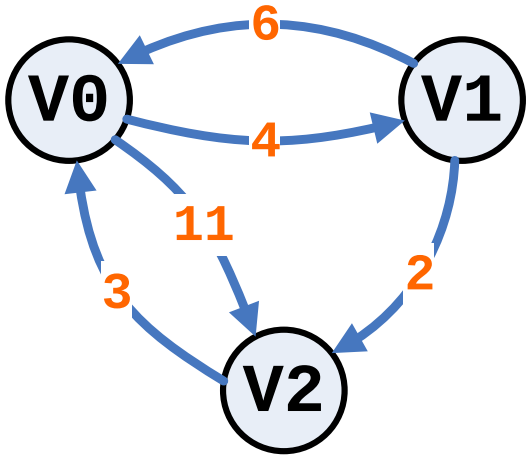
- 图的表示：邻接矩阵
- 初始时：
  - 任意两个顶点的最短路径就是直接连接的弧（有6对）

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0



最短路径问题：Floyd算法

- 如果绕道过去会不会更近呢？
  - 从哪个顶点绕道？
  - 每一个都试一试



	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

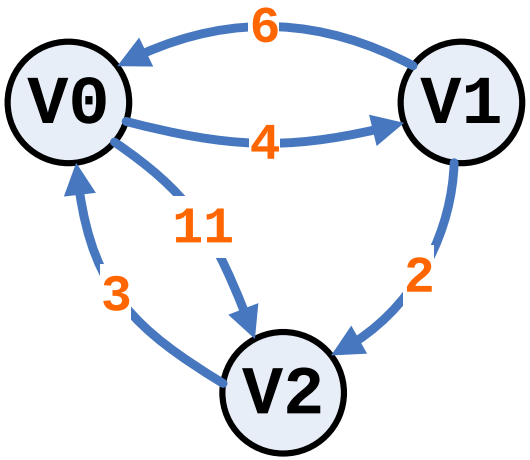
绕道V0 →

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

绕道V0

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0



绕道V1

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

绕道V2

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

最短路径问题：Floyd算法

- 算法分析
  - 时间复杂度 = $O(n^3)$
  - 分别要尝试从n个顶点绕道
  - 每尝试一个顶点，需要扫描整个矩阵

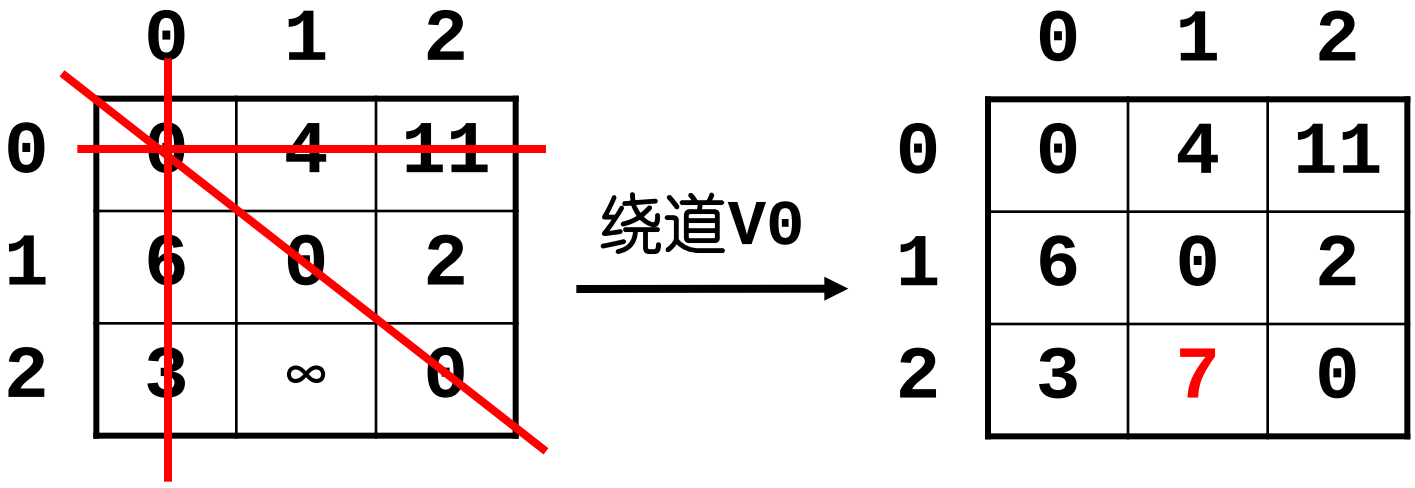
	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

绕道V0 →

	0	1	2
0	0	4	11
1	6	0	2
2	3	7	0

最短路径问题：Floyd算法

- 不过可以改进
  - 尝试绕道Vi，不需要扫描整个矩阵，对角线，第i行，第i列可以省略
  - 这样每次只须扫描$n^2-(3n-2)$个单元



最短路径问题：Floyd算法

- 补充题
  - 有一图的邻接矩阵如下，试给出用弗洛伊德算法求各点间最短距离的矩阵序列$A^{(1)}$、$A^{(2)}$、$A^{(3)}$和$A^{(4)}$。

$$A=\begin{bmatrix} 0 & 5 & \infty & 4 \\ \infty & 0 & 2 & \infty \\ 3 & 5 & 0 & \infty \\ 2 & \infty & 6 & 0 \end{bmatrix}$$

最短路径问题：Floyd算法

- 解答：

$$A^{(0)}=\begin{bmatrix}0 & 5 & \infty & 4\\ \infty & 0 & 2 & \infty\\ 3 & 5 & 0 & \infty\\ 2 & \infty & 6 & 0\end{bmatrix}\quad A^{(1)}=\begin{bmatrix}0 & 5 & \infty & 4\\ \infty & 0 & 2 & \infty\\ 3 & 5 & 0 & 7\\ 2 & 7 & 6 & 0\end{bmatrix}\quad A^{(2)}=\begin{bmatrix}0 & 5 & 7 & 4\\ \infty & 0 & 2 & \infty\\ 3 & 5 & 0 & 7\\ 2 & 7 & 6 & 0\end{bmatrix}$$

$$A^{(3)}=\begin{bmatrix}0 & 5 & 7 & 4\\ 5 & 0 & 2 & 9\\ 3 & 5 & 0 & 7\\ 2 & 7 & 6 & 0\end{bmatrix}\quad A^{(4)}=\begin{bmatrix}0 & 5 & 7 & 4\\ 5 & 0 & 2 & 9\\ 3 & 5 & 0 & 7\\ 2 & 7 & 6 & 0\end{bmatrix}$$

最短路径问题

- 本节小结
  - 手工掌握：要能写出过程
- 思考题6
  - 习题集7.11、7.13

本章小结

- 图的定义和术语
- 图的存储结构
  - 邻接矩阵、邻接表、十字链表、多重邻接表
- 图的遍历
  - 深度优先、广度优先
- 图的连通性问题
  - 连通分量、生成树、最小生成树
- 有向无环图的应用
  - 拓扑排序、关键路径
- 最短路径问题
  - **Dijkstra**算法、**Floyd**算法

作业和思考题

- 作业
  - 改编图的遍历算法，变成以下两个：
    - 对于无向图，写出所有连通分量
    - 判断两个顶点是否有路径（不论有向无向）
- 思考题
  - **1、7、10、11、13**