数据结构

董洪伟

http://hwdong.com

主要内容

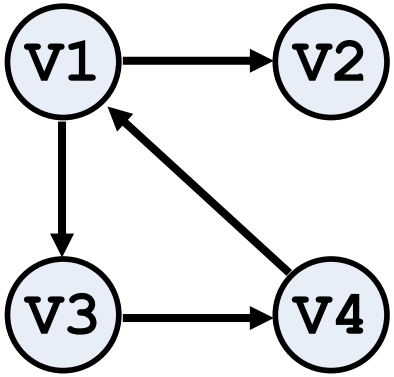
- 图的定义和术语
- 图的存储结构
  - 邻接矩阵、邻接表、十字链表、多重邻接表
- 图的遍历
  - 深度优先、广度优先
- 图的连通性问题
  - 连通分量、生成树、最小生成树
- 有向无环图的应用
  - 拓扑排序、关键路径
- 最短路径问题
  - **Dijkstra**算法、**Floyd**算法

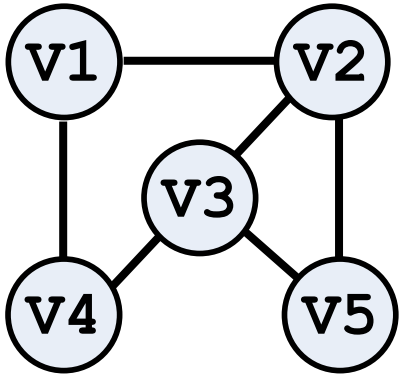
图的定义和术语

- 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

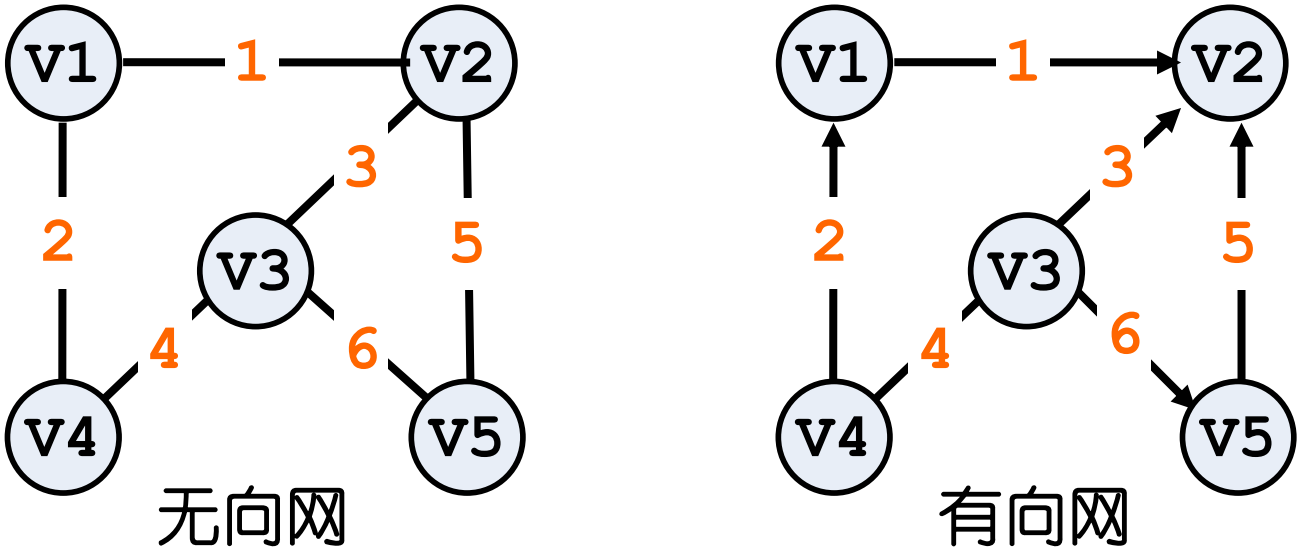
- 若<v,w>∈VR，则<v,w>表示从v到w的一条弧，且称v为弧尾或初始点，w为弧头或终端点，此时的图称之为有向图
- 若<v,w>∈VR，必有<w,v>∈VR，即VR是对称的，则以无序对(v,w)代替这两个有序对，表示v和w之间的一条边，此时的图称为无向图





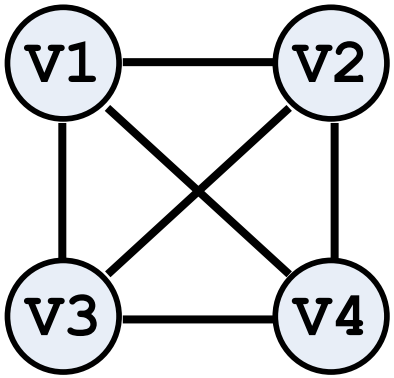
图的定义和术语

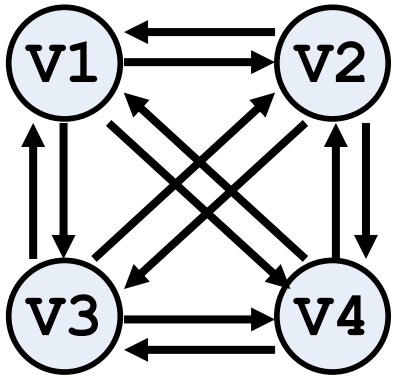
- 权
  - 与图的边（弧）相关的数值
- 网络
  - 带有权的图



图的定义和术语

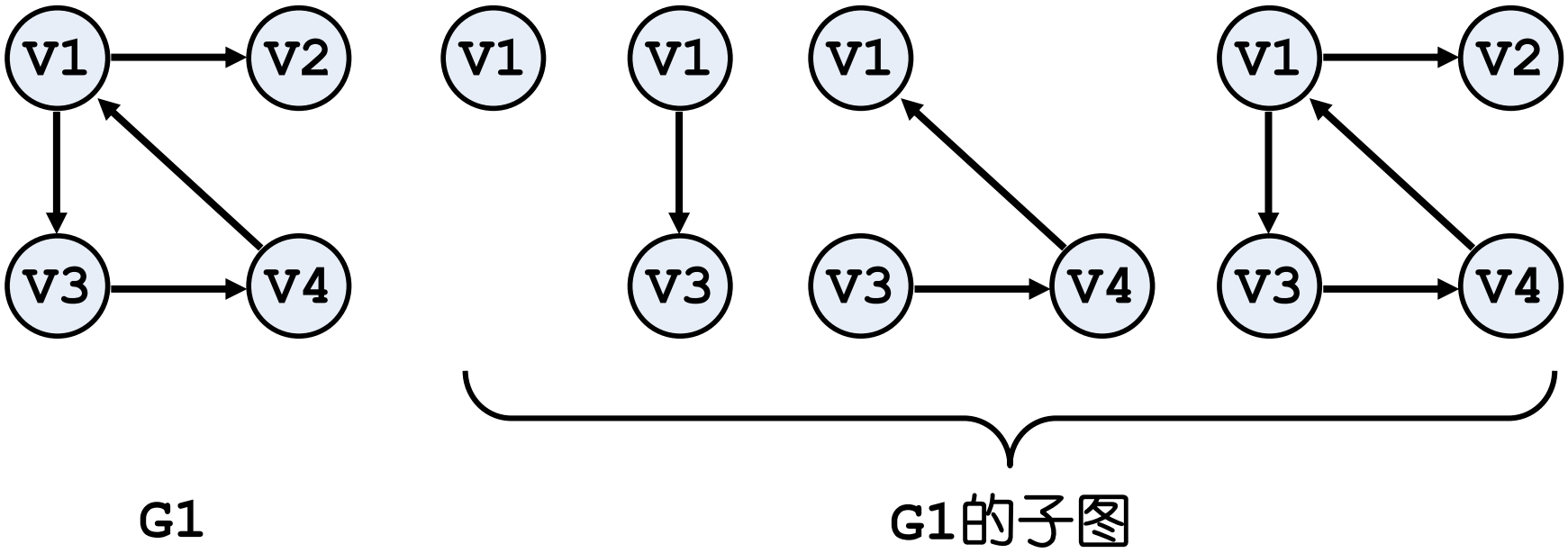
- 完全图：
  - 若有n个顶点的无向图有 $C_n^2 = n(n-1)/2$ 条边，则此图为完全无向图
  - 有n个顶点的有向图有 $n(n-1)$ 条弧，则此图为完全有向图
  - 完全图其实就是边/弧的数量达到最大值

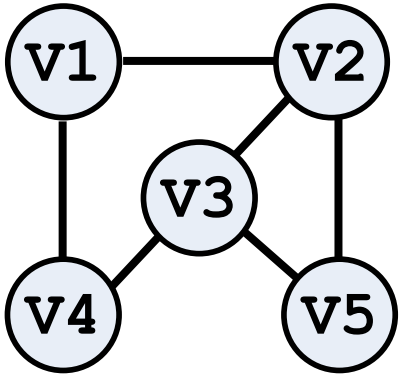




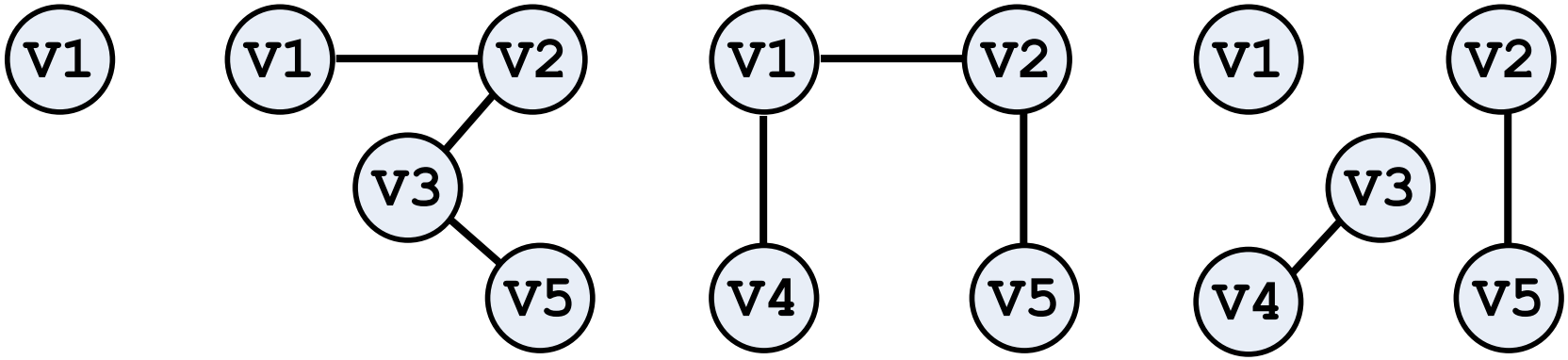
图的定义和术语

- 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





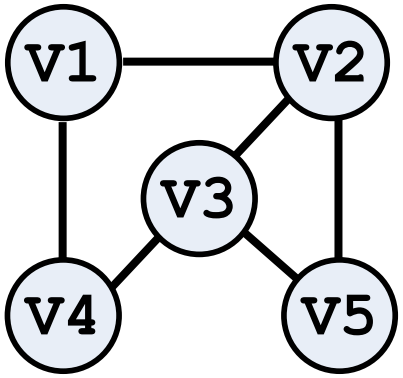
G2



G2的子图

图的定义和术语

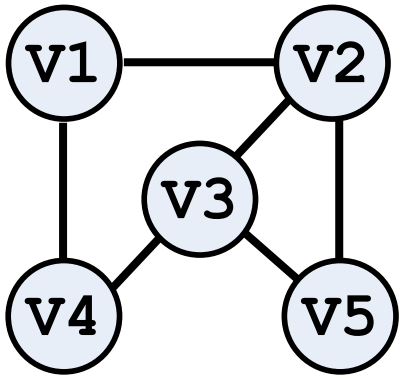
- 邻接点
  - 对于无向图G={V,{E}}，若边(v,v′)∈E，则称v和v′互为邻接点
  - 称边(v,v′)依附于顶点v和v′
  - 或者说边(v,v′)和顶点v，v′相关联



- 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

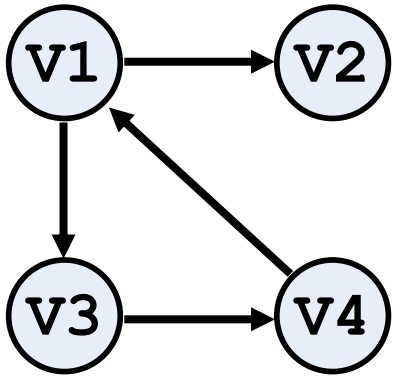
- 度
  - 顶点v的度TD(V)=和v相关联的边的数目



- 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



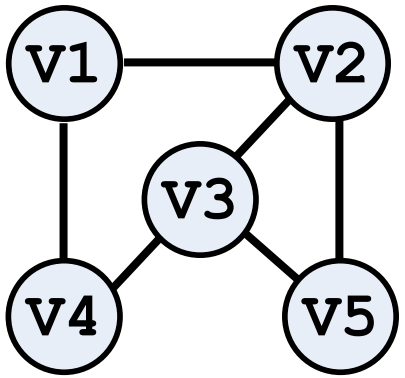
- ID(V1)=1
- OD(V1)=2
- TD(V1)=ID(v1)+OD(v1)=3

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

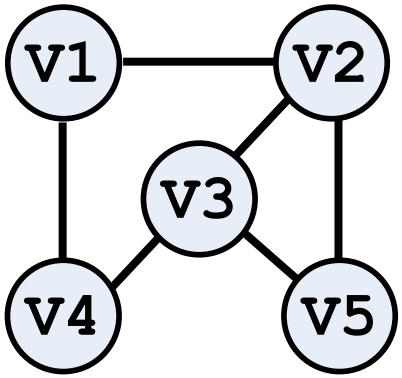


- TD(V1)=2
- TD(V2)=3
- TD(V3)=3
- TD(V4)=2
- TD(V5)=2

- 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)

...

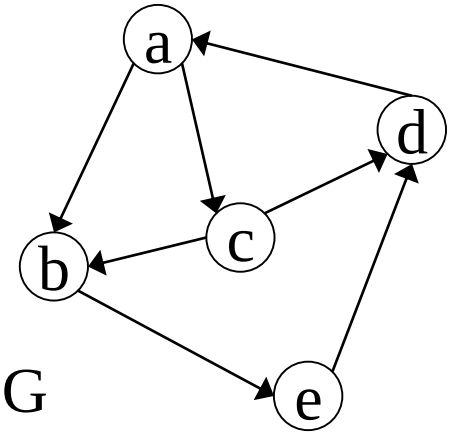
图的定义和术语

- 路径的长度
  - 路径上的边或弧的数目
- 回路（环）
  - 起点与终点相同的路径
- 简单路径
  - 没有重复顶点的路径，即不含回路
- 简单回路
  - 除起点和终点相同外，别无重复顶点的路径

图的定义和术语：例

【例】设有如下有向图 G，则下列顶点序列中哪些是路径，哪些不是？路径中哪些是简单路径，哪些是回路，哪些是简单回路？

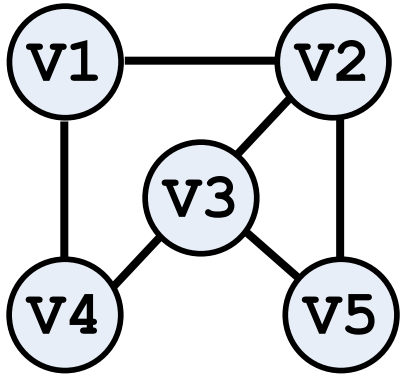
(1)aca　(2)acda　(3)cbedacb　(4)ac



图G

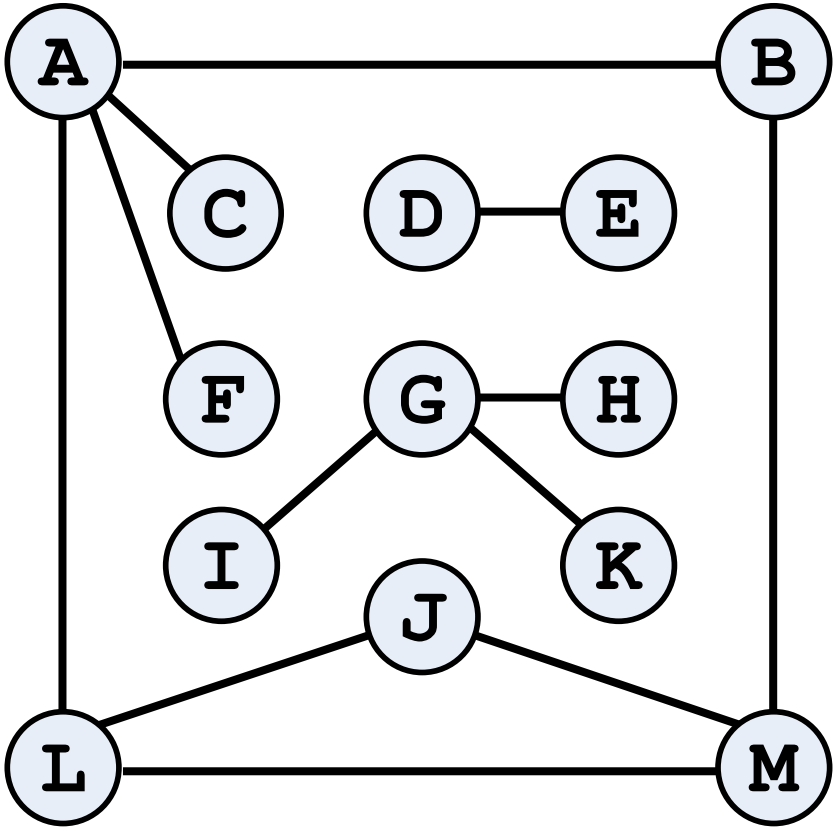
图的定义和术语

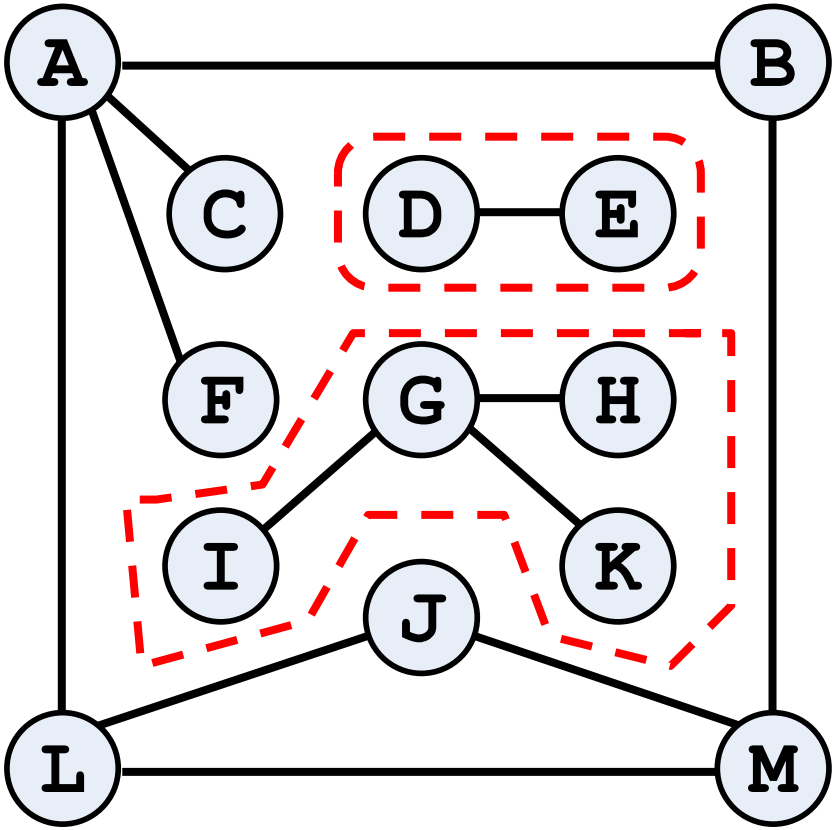
- 连通图
  - 在无向图G中，如果从顶点v到顶点v′有路径，则称v和v′是连通的
  - 如果对于图中的任意两个顶点vi和vj都是连通的，则称G是连通图
  - 是否连通是对无向图来说的

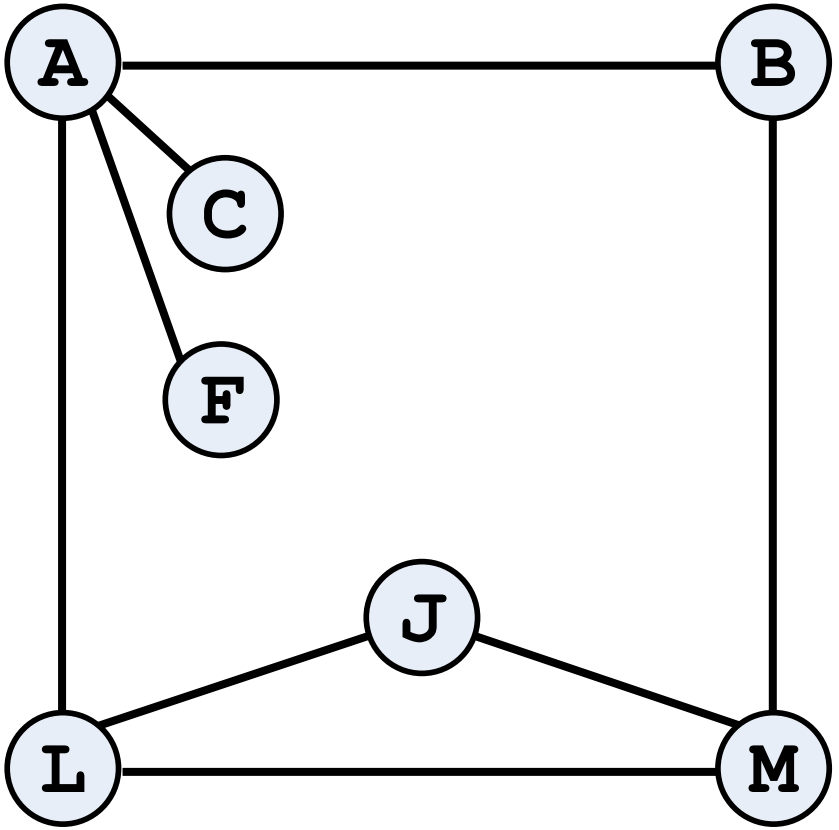


图的定义和术语

- 连通分量
  - 无向图中的极大连通子图

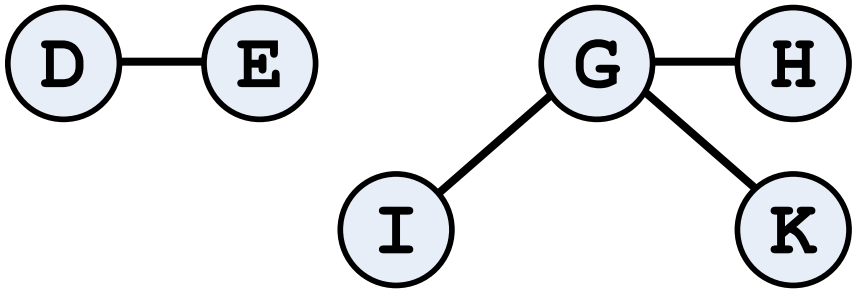






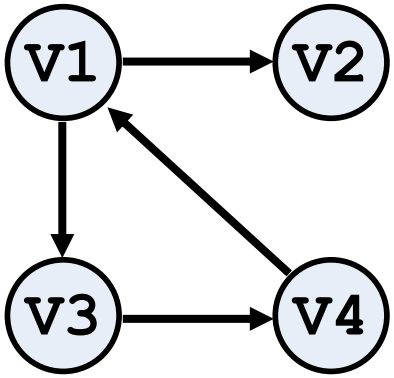
何为极大?

顶点不能再多了



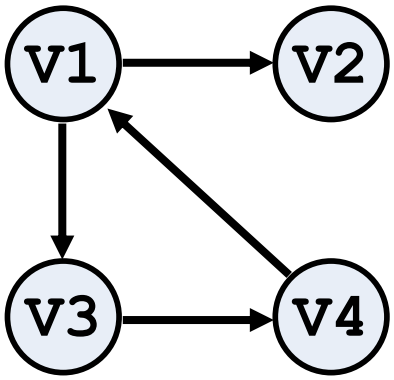
图的定义和术语

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

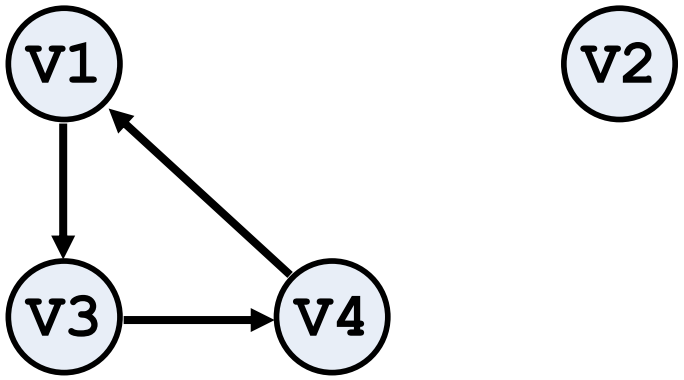


图的定义和术语

- 强连通分量
  - 有向图中的极大强连通子图



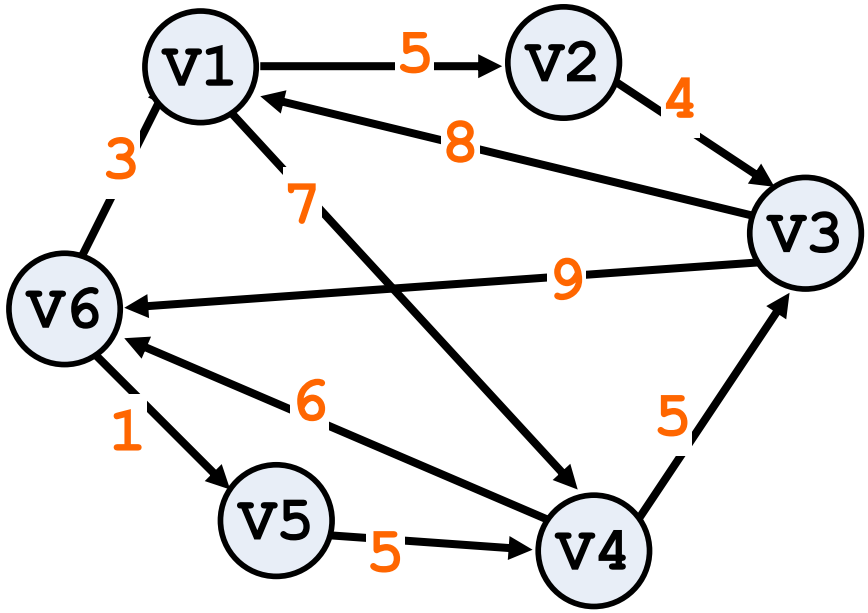
G1不是强连通图



但有两个强连通分量

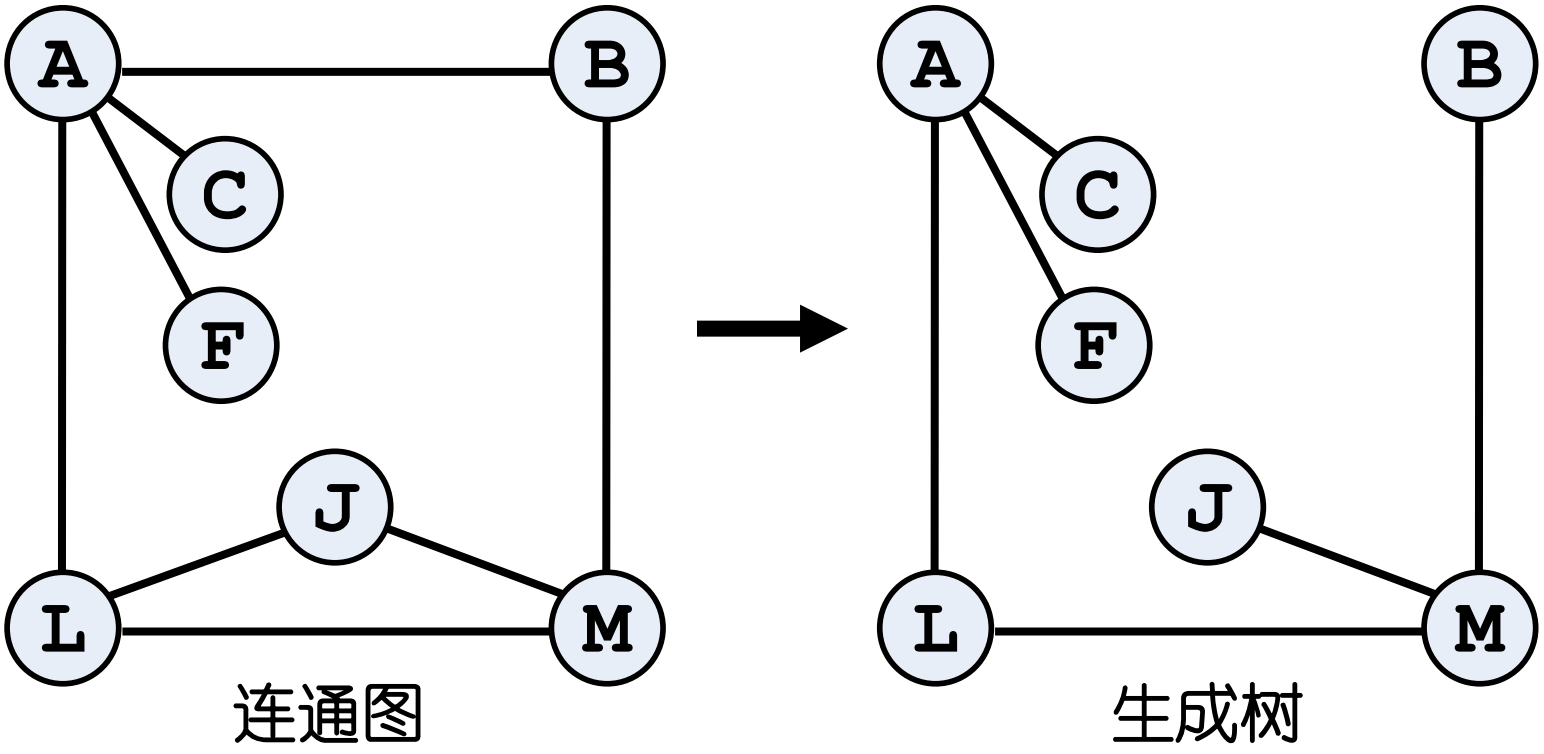
图的定义和术语

- 带权图：边或弧上关联的值
- 带权路径长度：路径上权值之和



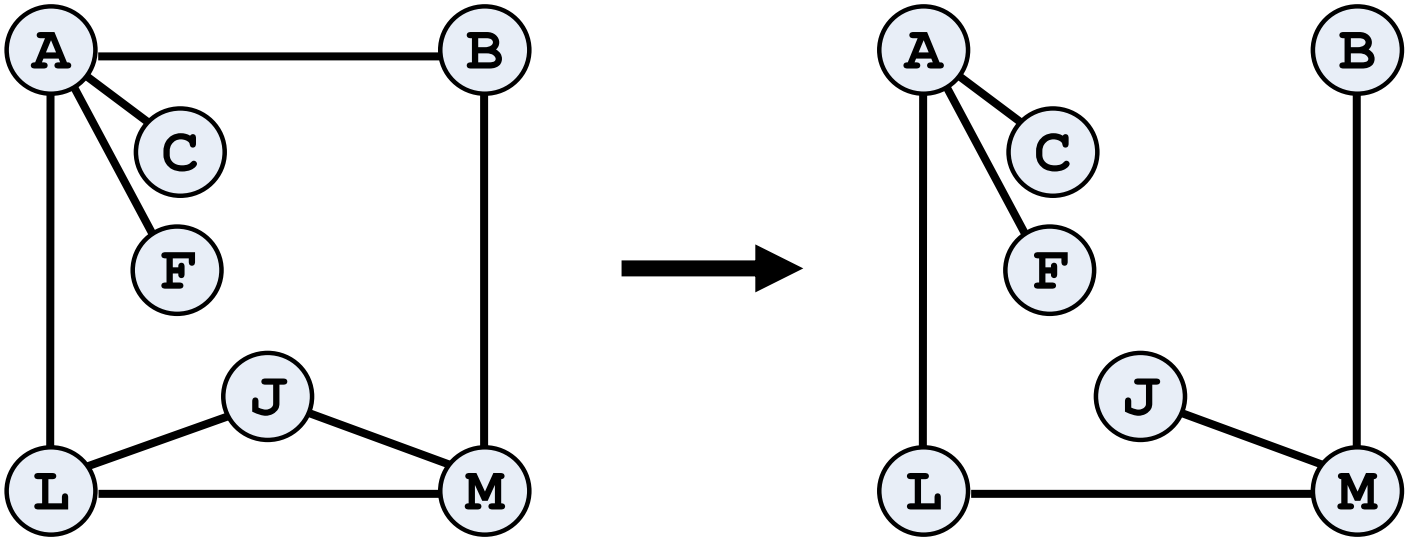
图的定义和术语

- 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

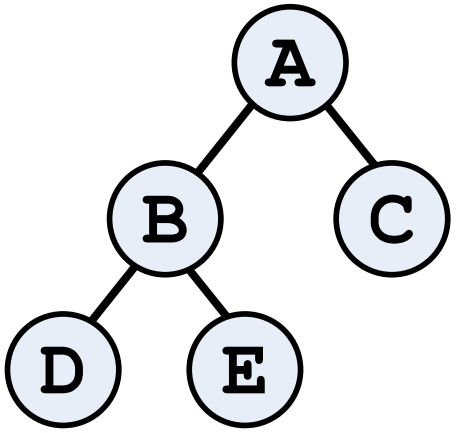
- 理解
  - “包含所有顶点”：顶点一个都不能少
  - “极小”：那只能让边尽量少了
  - “连通”：但是又不能太少

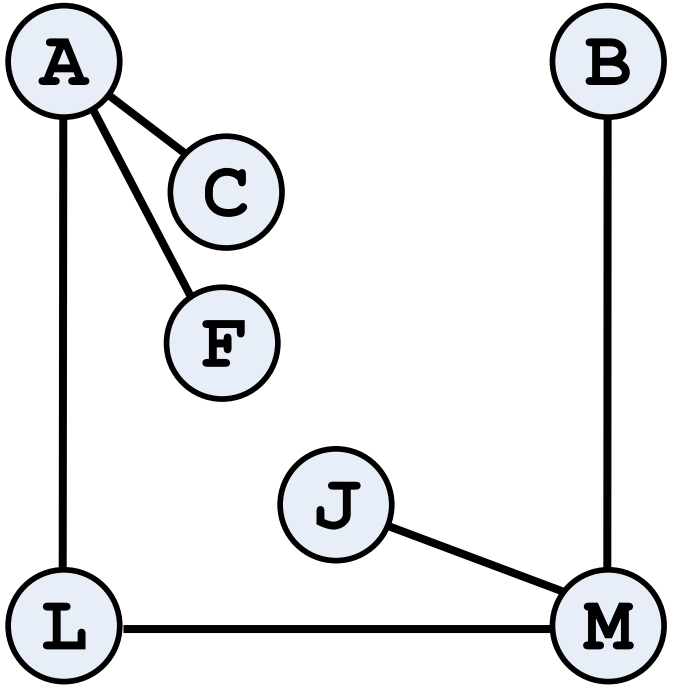


图的定义和术语

- “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

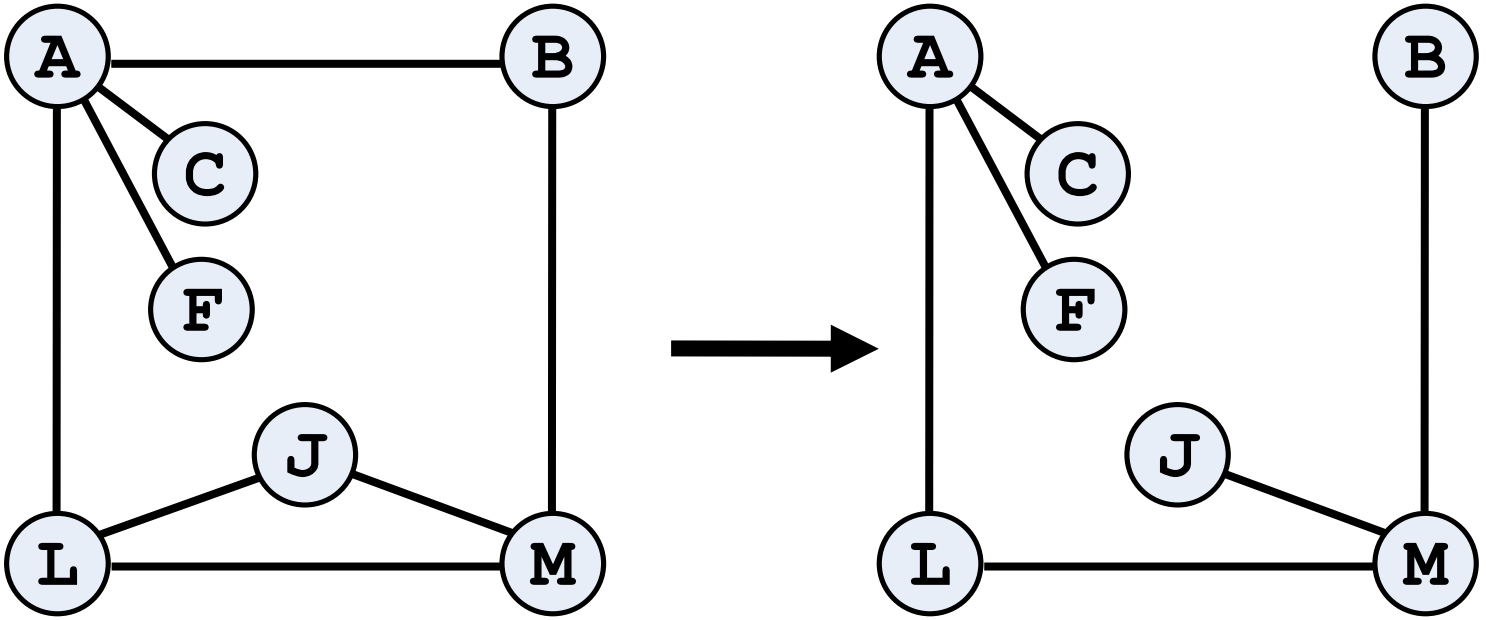
N = B + 1





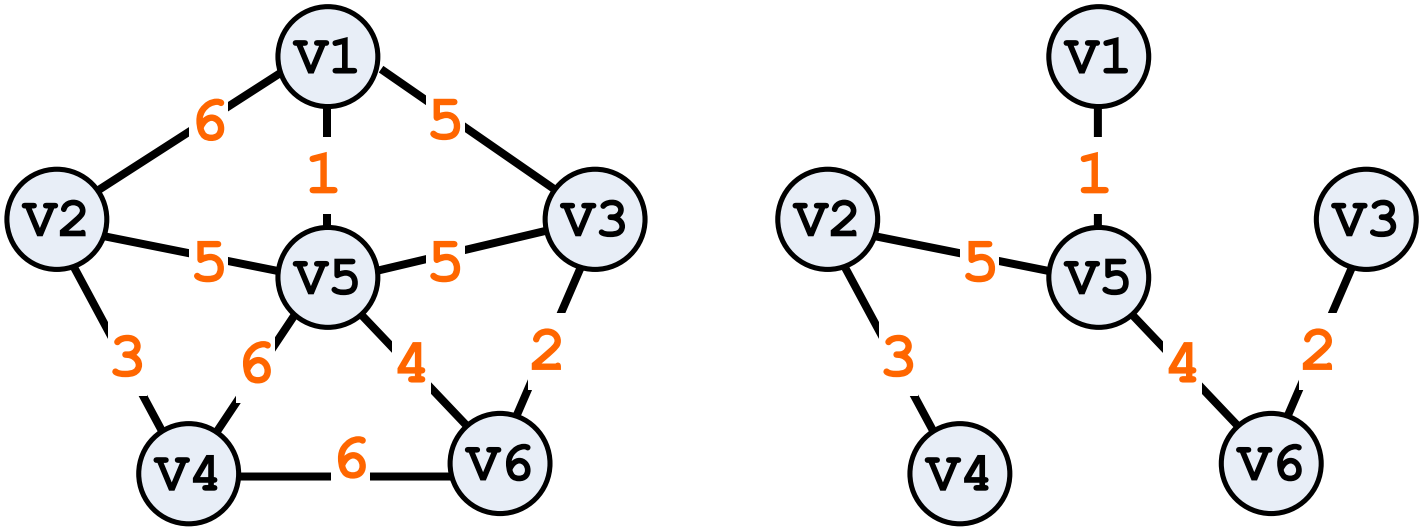
图的定义和术语

- 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

- 带权无向图的最小生成树
  - 各边权值之和最小的生成树



图的定义和术语：类型定义

```
class Graph {
 public:
  Graph ( );
  void InsertVertex ( const Type & vertex );
  void InsertEdge ( const int v1, const int v2, int weight )
  void RemoveVertex ( const int v )
  void RemoveEdge ( const int v1, const int v2 );
  int IsEmpty ( );
  Type GetWeight ( const int v1, const int v2 );
```

图的定义和术语：类型定义

```
  int GetFirstNeighbor ( const int v );
  int GetNextNeighbor ( const int v1, const int v2 );
  void GetVex (int v, Type  &vertex);
  void PutVex (int v, Type  vertex);
  void DFSTraverse (int v, void visit(Type  vertex));
  void BFSTraverse (int v, void visit(Type  vertex));
  int FindVertex ( Type  u );
}
```

图的定义和术语

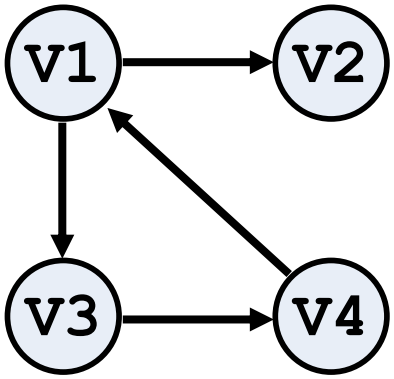
- 本节小结
  - 概念很多要记清楚
  - 了解图有哪些基本操作

图的存储结构

- 数组表示法（邻接矩阵）
- 邻接表
- 十字链表（有向图）
- 多重邻接表

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

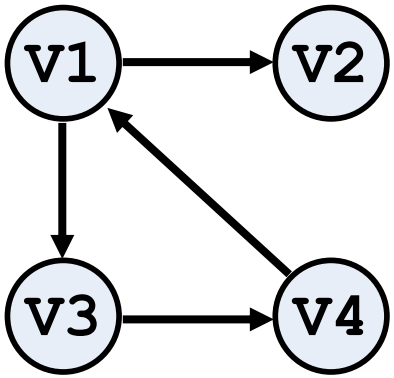


	v1	v2	v3	v4
v1	0	1	1	0
v2	0	0	0	0
v3	0	0	0	1
v4	1	0	0	0

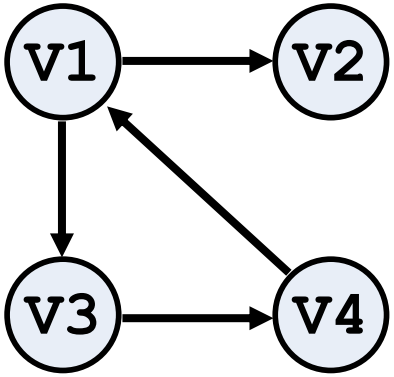
图的存储结构：邻接矩阵

- 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：

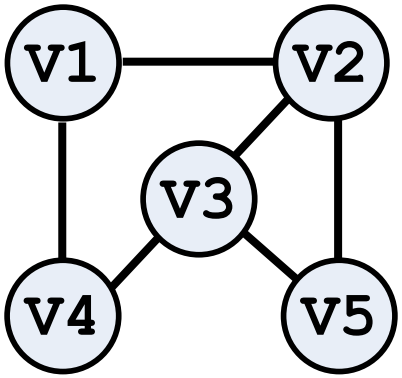
$$G.arcs[i][j] = \begin{cases} 1, & \text{if } <i,j> \in E \text{ or } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$



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



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



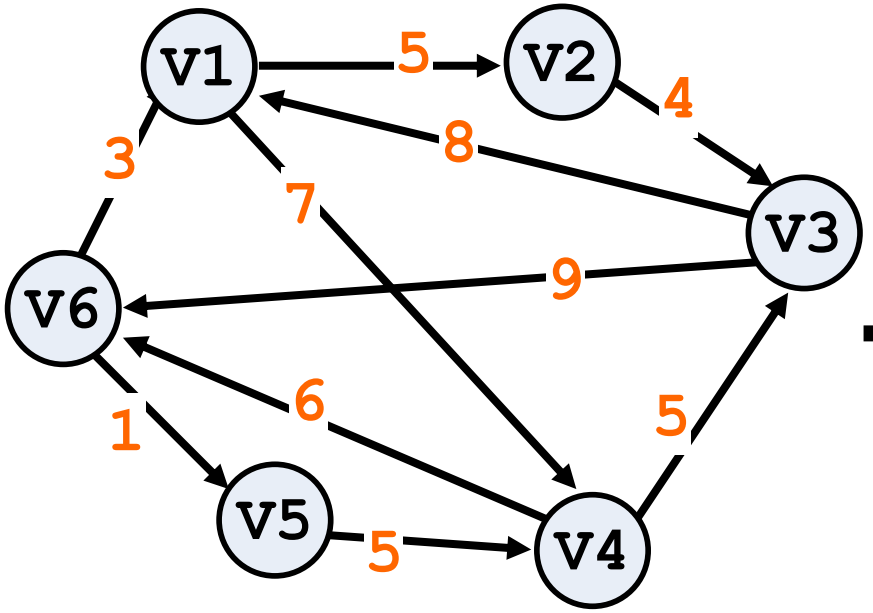
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

- 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

```
//最大顶点数
const int MAX_VERTEX_NUM = 1000 ;
//图的类型：有向图、有向网、无向图、无向网
enum GraphKind{ DG,DN,UDG,UDN};
```

图的存储结构：邻接矩阵

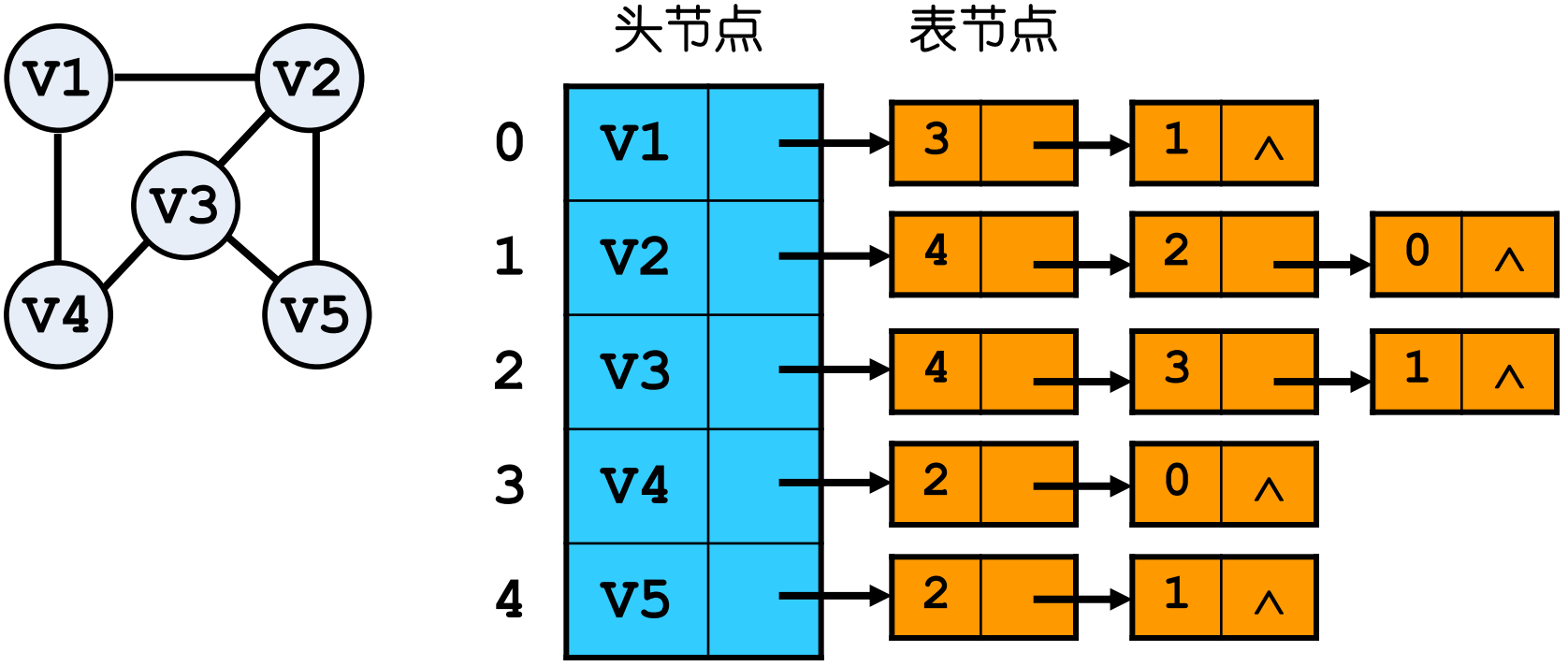
```
//一条边或弧的结构
struct Arc{
   wType adj; //表示是否有边及其权值
   InfoType  *info; //更多相关信息
};
```

图的存储结构：邻接矩阵

```
class MGraph{ //整张图的结构
    //顶点数组
   VertexType vexs [MAX_VERTEX_NUM];
   Arc arcs[MAX_VERTEX_NUM]
           [MAX_VERTEX_NUM];//邻接矩阵
   int    vexnum,          //顶点数
          arcnum           //弧数
   GraphKind  kind;              //图的类型
  //…                      //图的基本操作
};
```

图的存储结构：邻接表

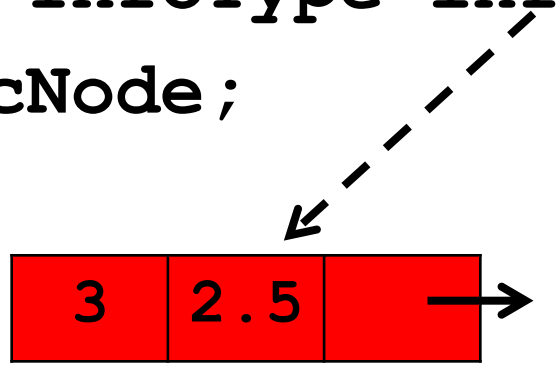
- 无向图的邻接表



– 表结点：跟当前头结点相连的另一个顶点

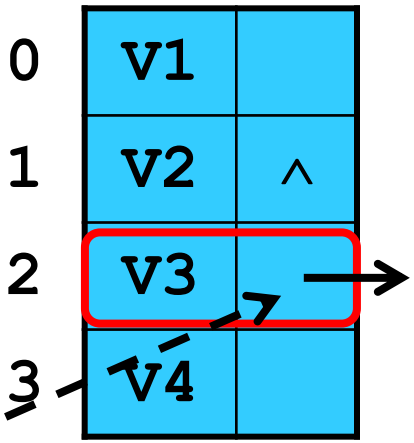
图的存储结构：邻接表

```
//表结点结构
typedef struct _ArcNode{
    int        adjvex;    //顶点下标
    _ArcNode   *next;     //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

```
//顶点结点结构
typedef struct{
      //顶点的数据
      VType data;
      //指向第一个表结点
      ArcNode     *firstarc;
}VNode;
```

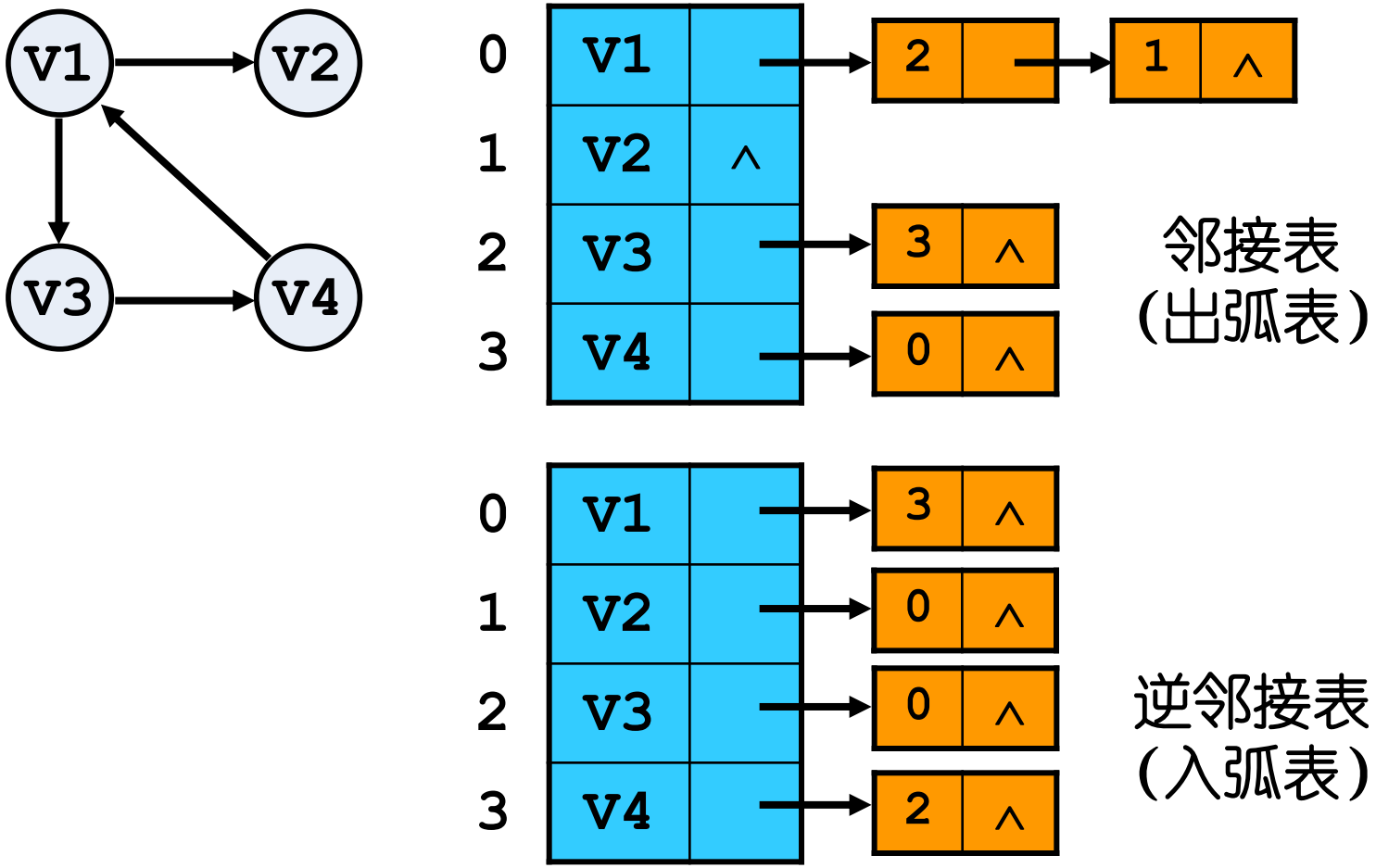


图的存储结构：邻接表

```
//邻接表的结构
#define VNMAX 1000 //最多顶点个数
typedef struct{
    VNode vertices[VNMAX]; //表结点数组
    int      vexnum;    //顶点个数
    int      kind;      //图的类型
}AGraph;
```

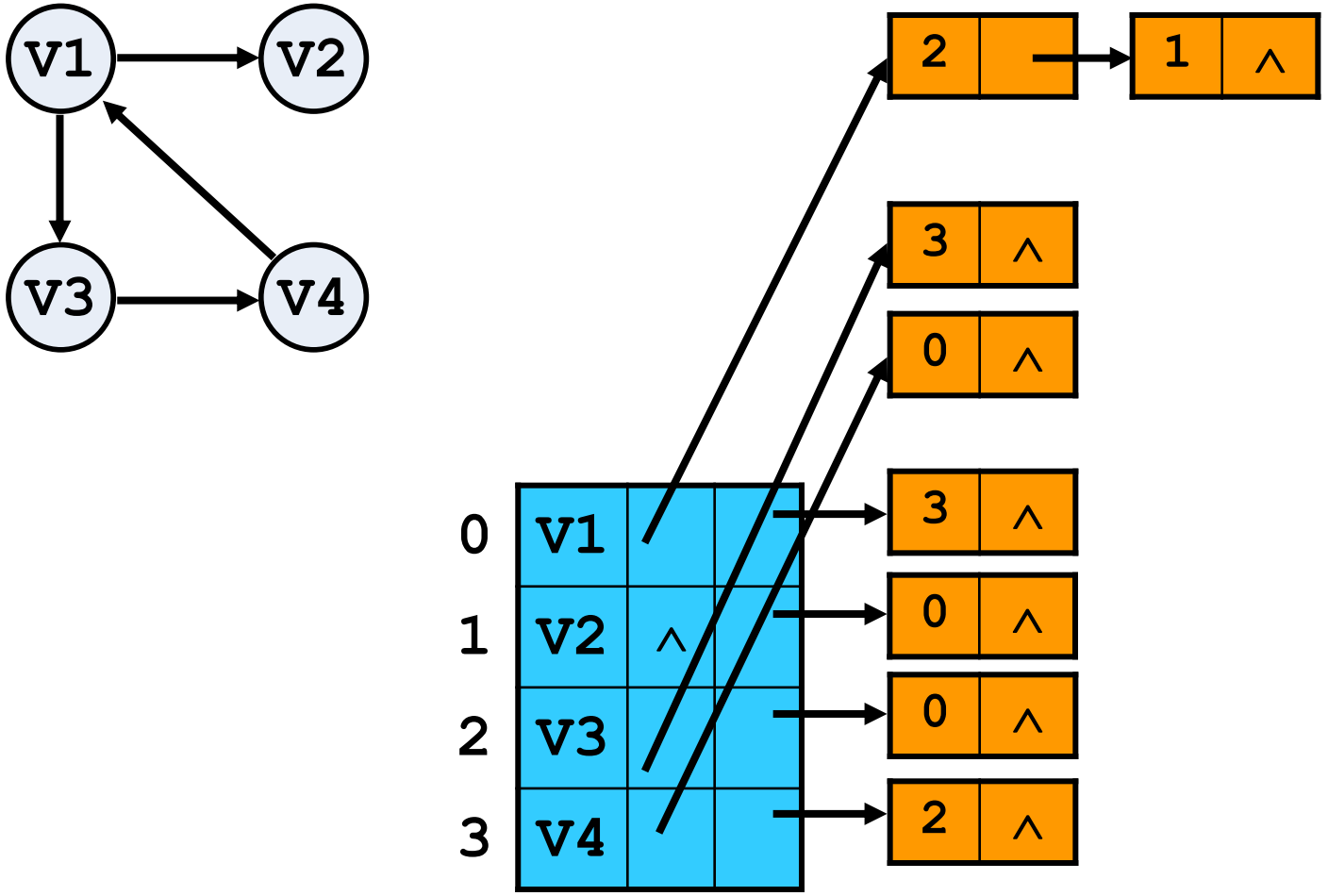
图的存储结构：邻接表

- 有向图的邻接表和逆邻接表



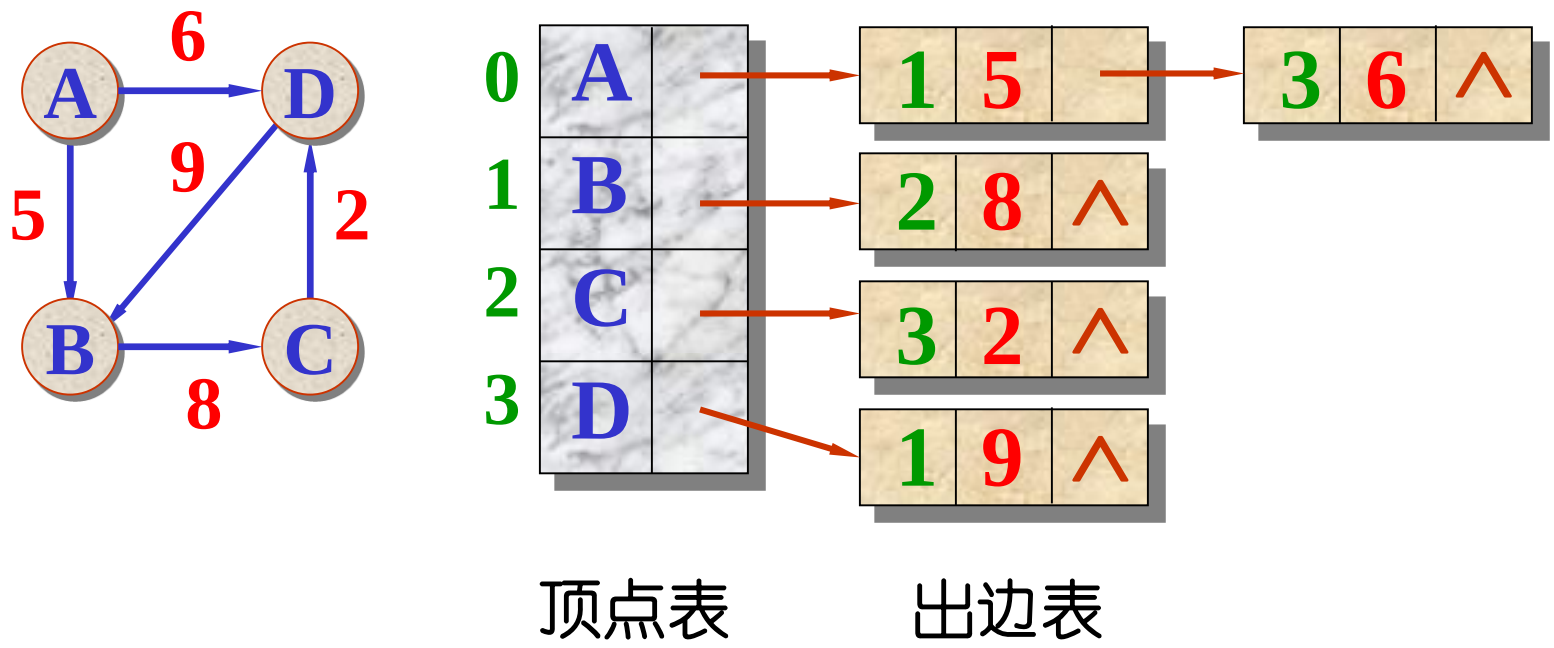
图的存储结构：邻接表

- 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个头节点和2e个表节点
  - 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

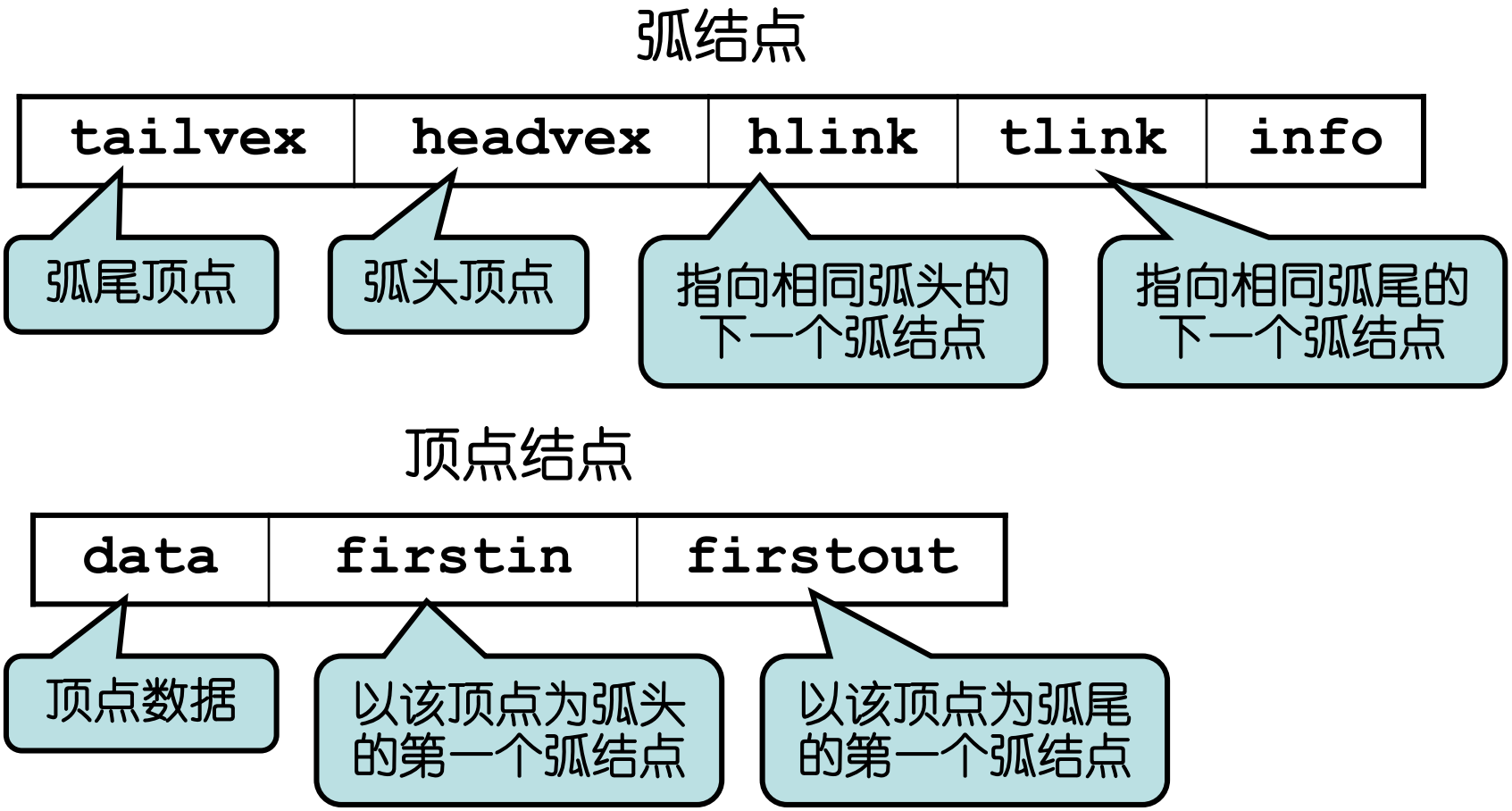
- 邻接表的操作
  - 计算某个顶点的度、出度、入度
    - 无向图中某个头结点后面链接的表结点的个数就是该顶点的度
    - 有向图中：
      - 邻接表的某个头结点后面链接的表结点的个数就是该顶点的出度
      - 逆邻接表的某个头结点后面链接的表结点的个数就是该顶点的入度

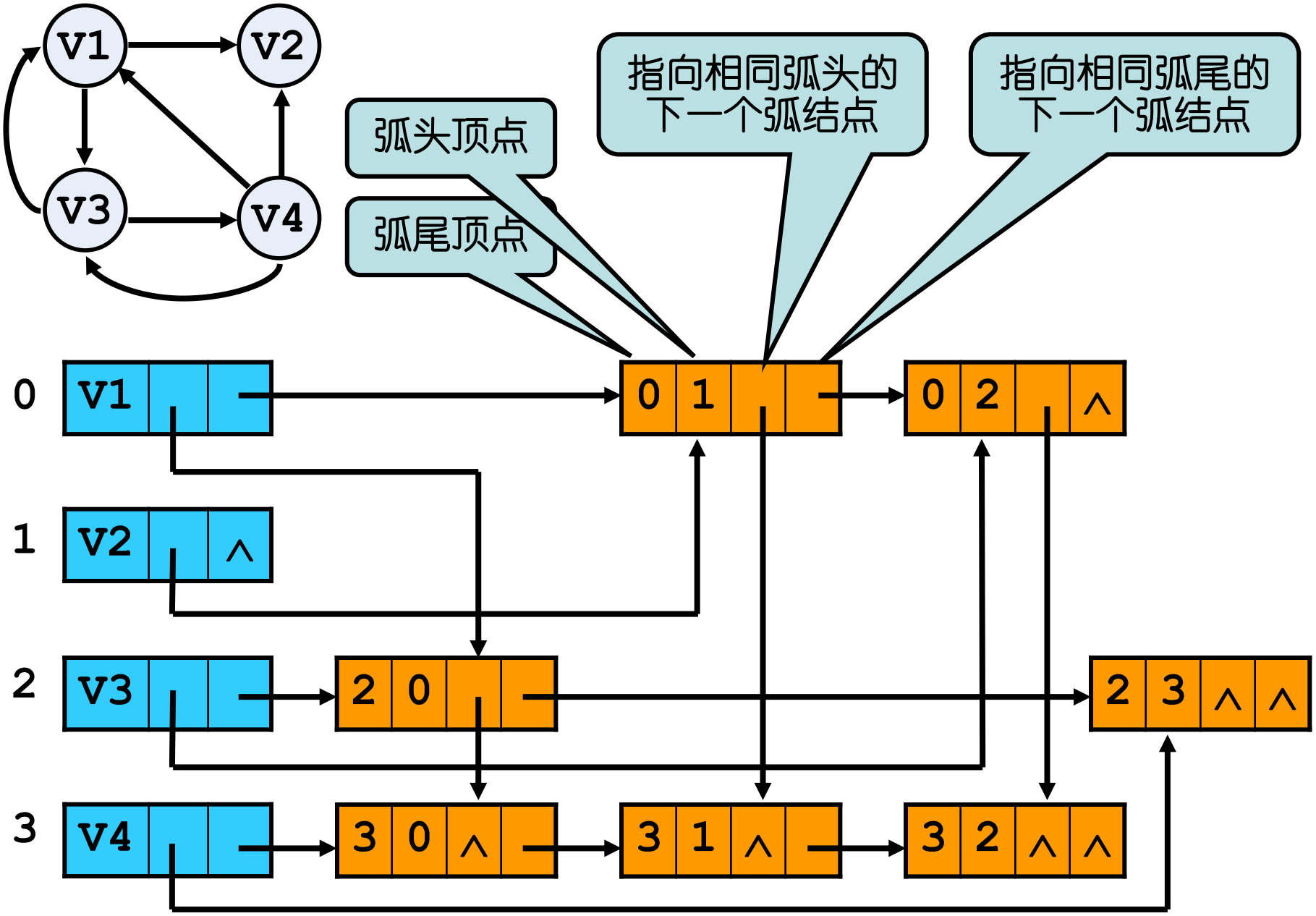
图的存储结构：邻接表

- 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

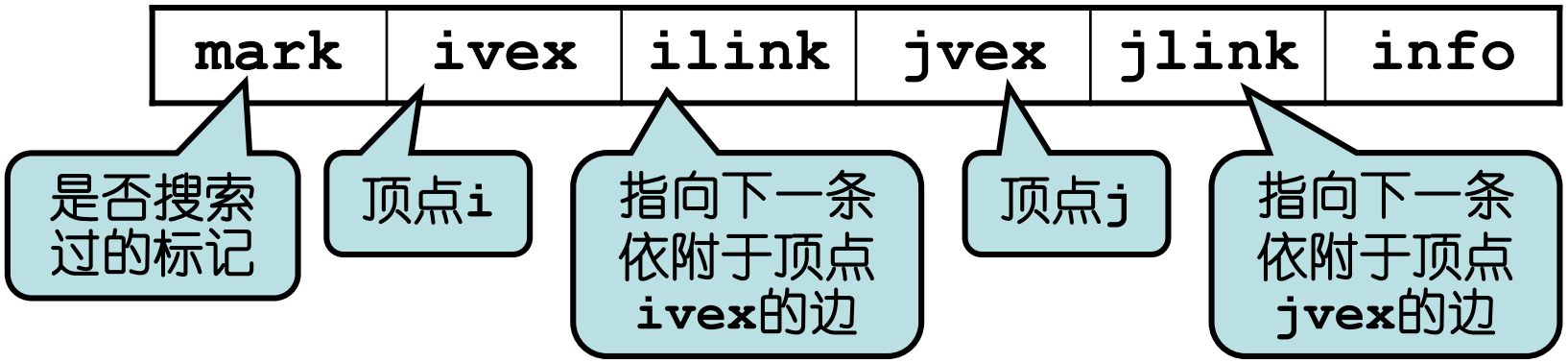
- 十字链表（有向图）

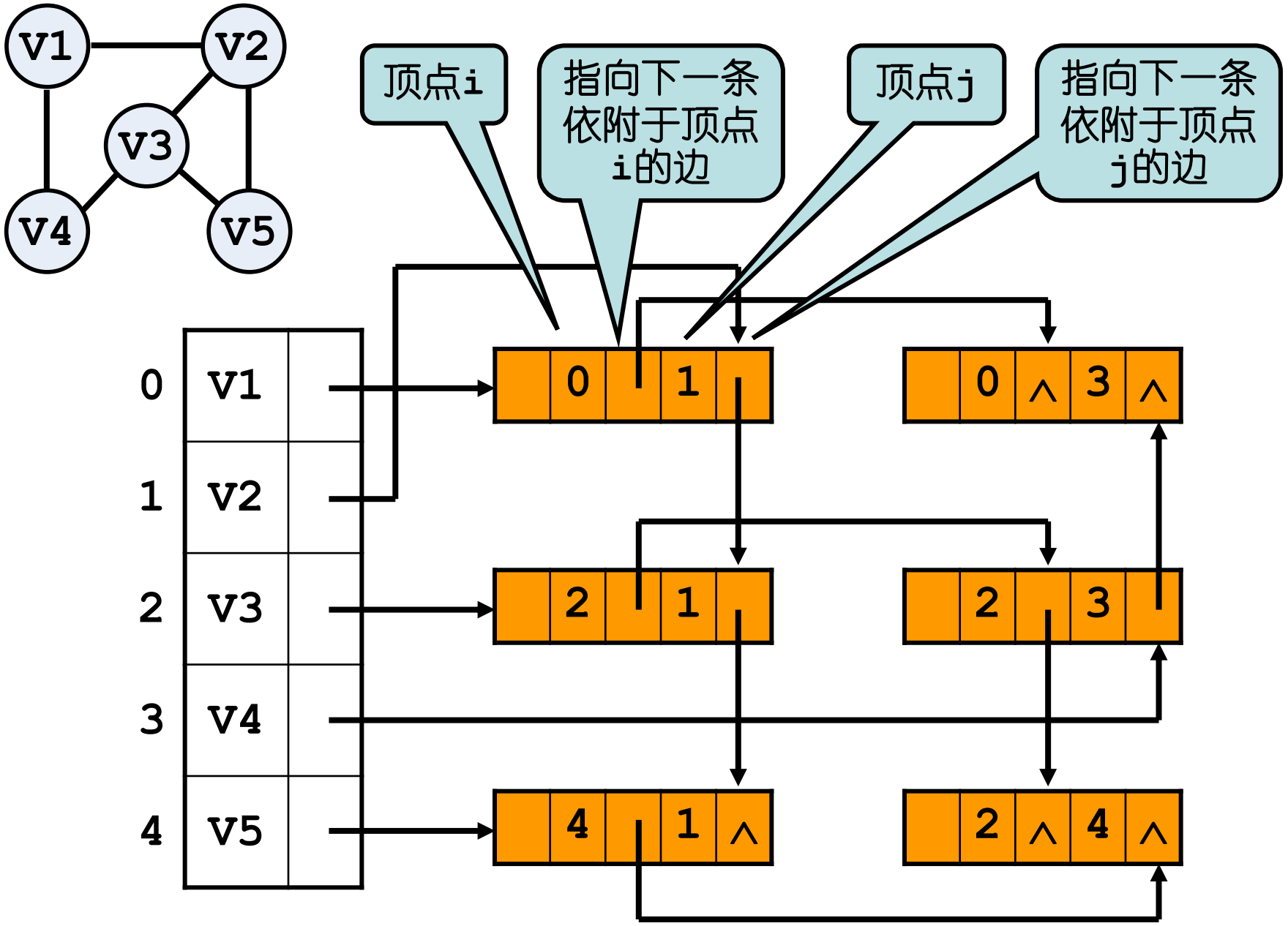



V1
V2
V3
V4
弧头顶点
弧尾顶点
指向相同弧头的
下一个弧结点
指向相同弧尾的
下一个弧结点
0
1
2
3
V1
V2
V3
V4
0 1
0 2
2 0
2 3
3 0
3 1
3 2

图的存储结构：多重邻接表

- 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(无向图)
  - 每一条边只用一个结点表示



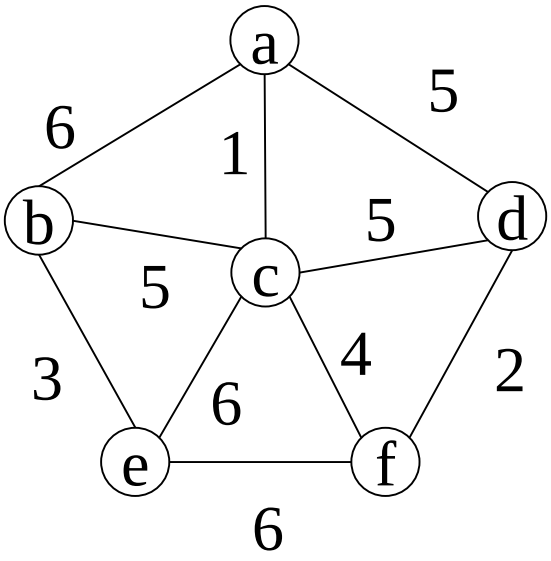

V1
V2
V3
V4
V5
顶点i
指向下一条
依附于顶点
i的边
顶点j
指向下一条
依附于顶点
j的边
0
1
2
3
4
V1
V2
V3
V4
V5
0
1
0
^
3
^
2
1
2
3
4
1
^
2
^
4
^

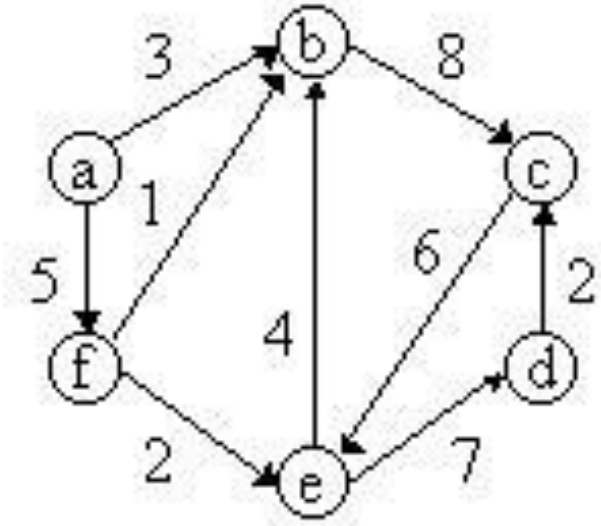
图的存储结构

- 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

图的存储结构：练习

画出下列图的各 3 种存储结构

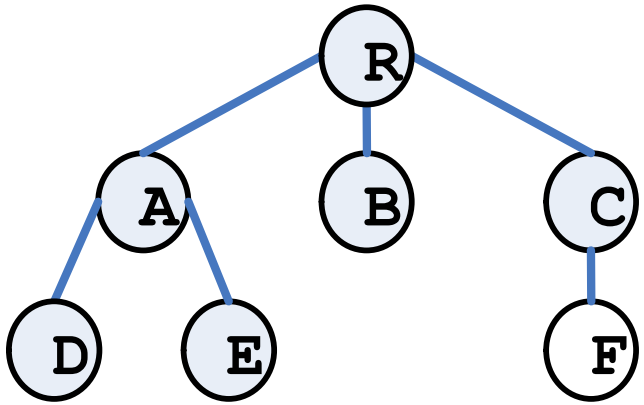




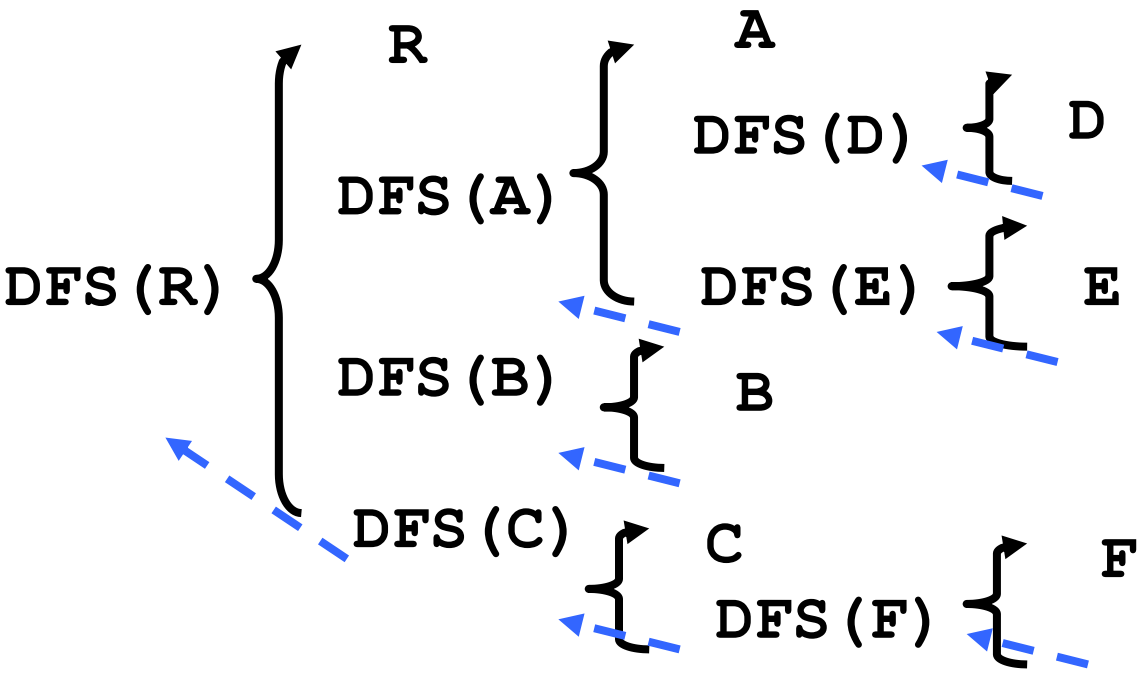
图的遍历

- 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
  - 类似于树的先根遍历和层次遍历，有图的深度优先遍历　和　广度优先遍历

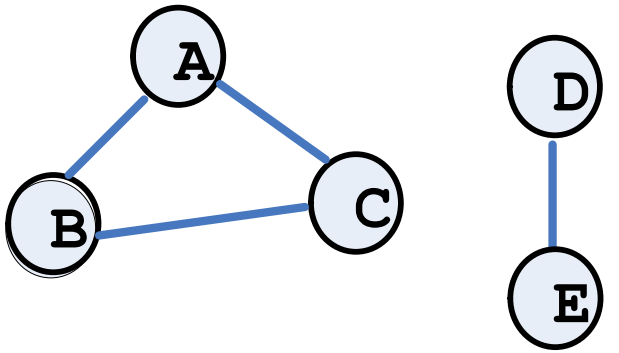
回顾树的先根遍历




```
DFS(V){
    直接访问V;
    for(V的每个孩子W)
        DFS(W);
}
```



树的遍历方法是否不修改就适用于图的遍历呢?




```
DFS(V){
    直接访问V;
    for(V的每个邻接点W)
        DFS(W);
}
```

问题 1：由于图存在环路，所以会导致无限循环

解决方法：设置顶点访问标志

问题 2：由于图不一定连通，从一个顶点出发的遍历只能访问其所在的连通分量中的所有顶点

解决方法：重复调用从一个顶点出发的**DFS**或**BFS**

图的遍历：深度优先遍历

深度优先遍历

```
DFS(V){
    直接访问V;
    flag(v) = 1;
    for(V的每个邻接点W)
        if( flag(w)!=1)
            DFS(W);
}
```

从一个顶点出发的深度优先遍历

```
DFS(G){
   for(每个v)
      flag[v]= 0;
   for(每个v)
      if(flag(v)==0)
         DFS(v);
}
```

整个图的深度优先遍历

图的遍历：深度优先遍历

```
DFS(G){
   for(每个v)
      flag = 0;
   for(每个v)
      if(flag(v)==0)
         DFS(v);
}
```

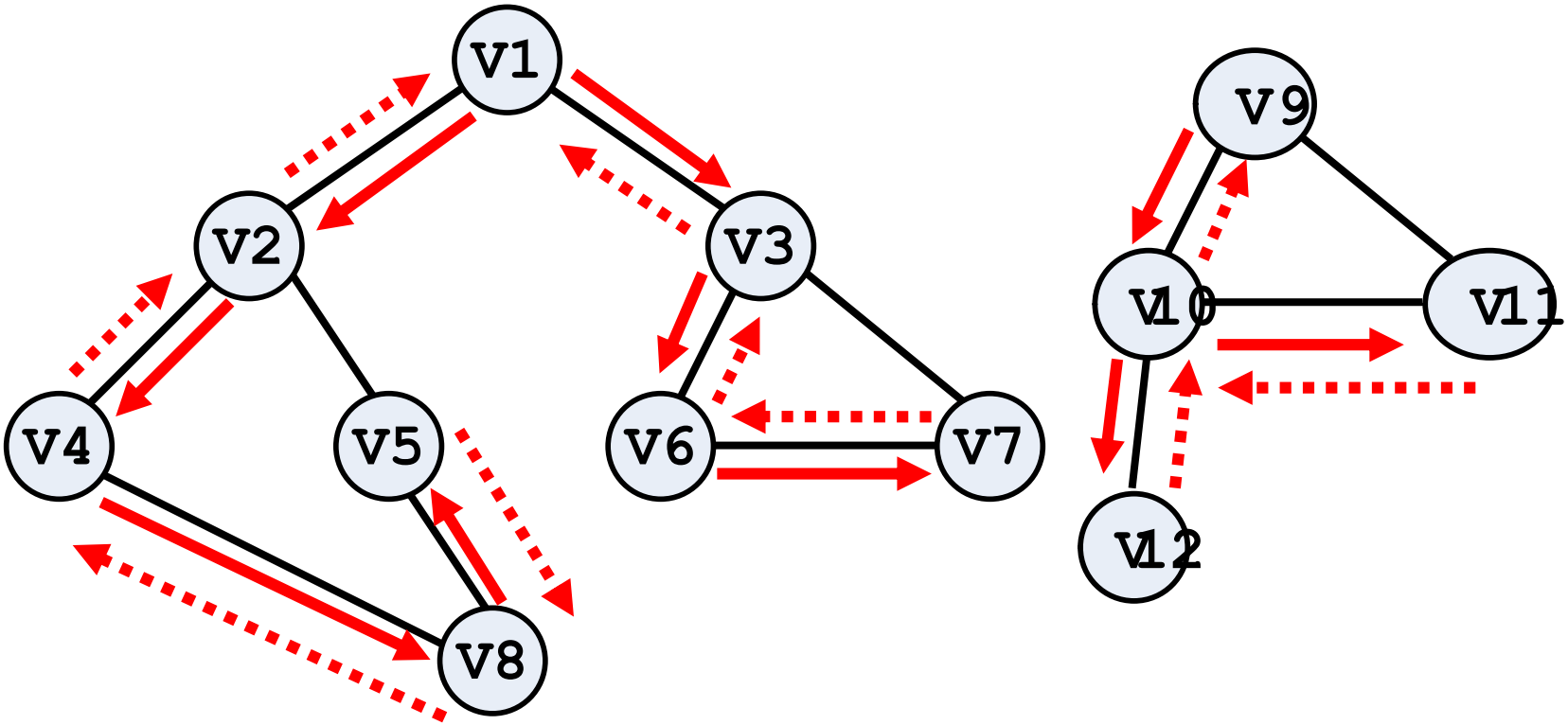
时间复杂度：

1）初始化访问标志 O(n)

2)每个顶点的DFS,对每个顶点，要访问其邻接点：

对于邻接表，次数为顶点的度，而所有顶点的度之和为2e；

对于邻接矩阵，次数为n*n

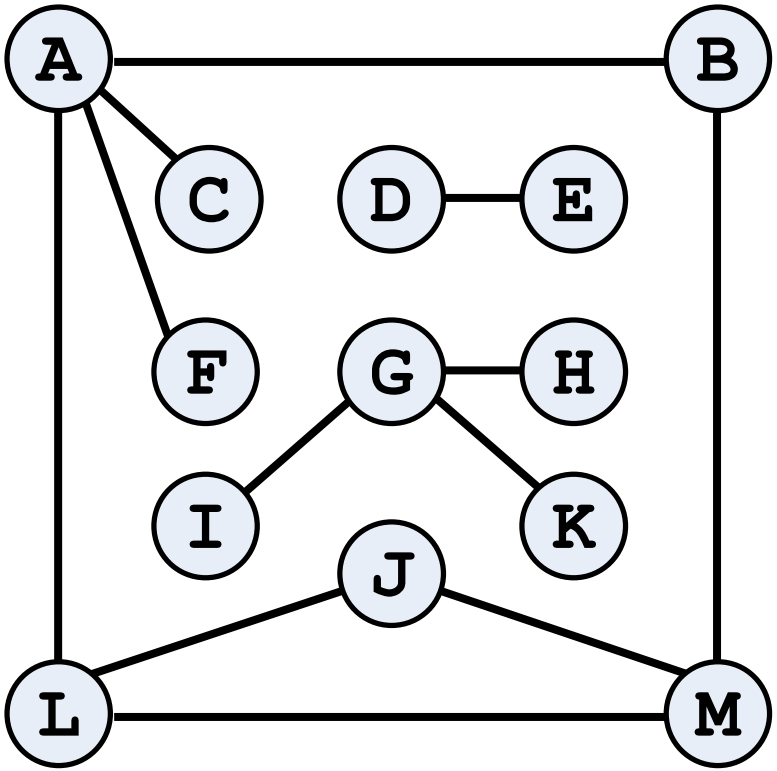


V1->V2->V4->V8->V5->V3->V6->V7

V9->V10->V12->V11

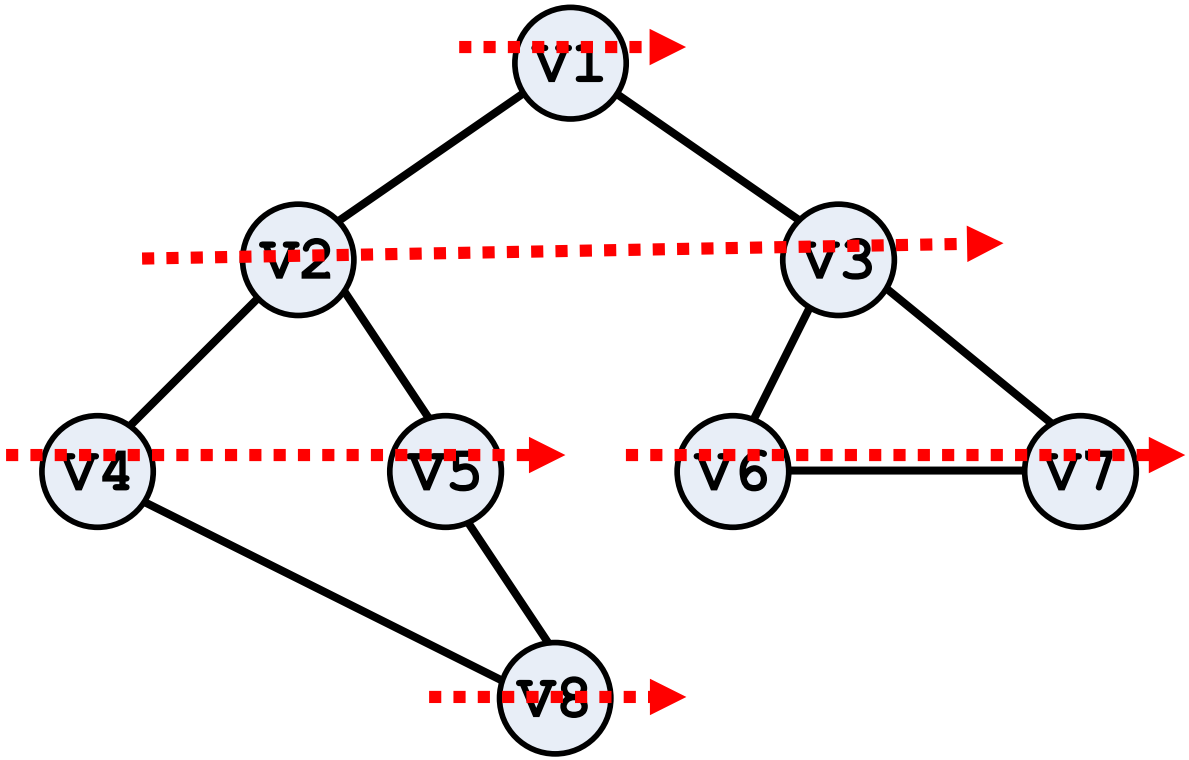
图的遍历：深度优先遍历

- 练习
  - 写出对下图进行深度优先遍历的结果



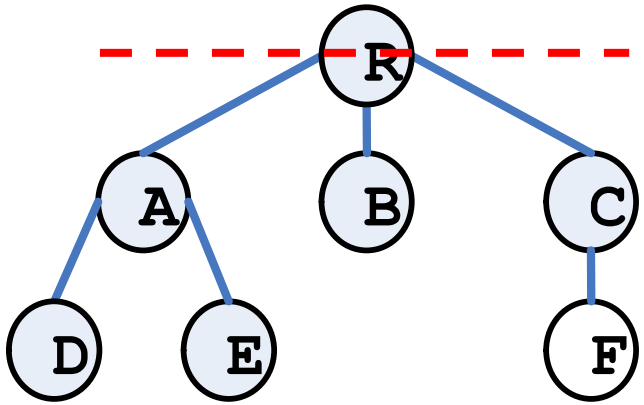
  - **A B M J L C F**
  - **D E**
  - **G H I K**

图的遍历：广度优先遍历



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

回顾树的层次遍历

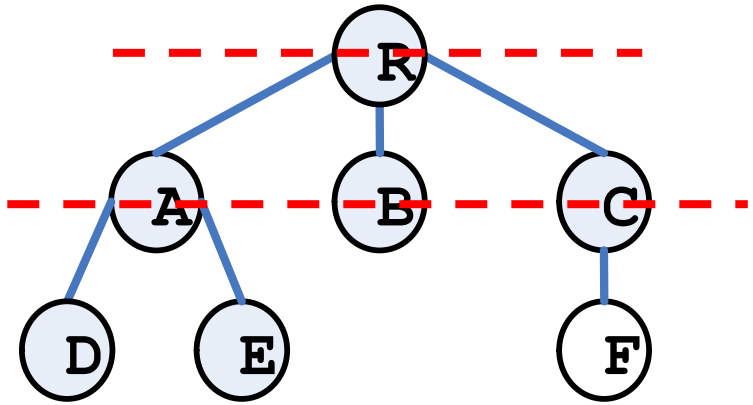



```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

先进先出：

R

回顾树的层次遍历



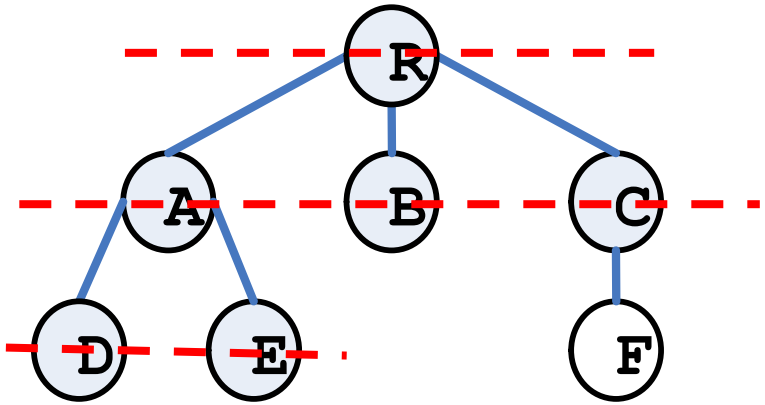

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

先进先出：

R

A B C

回顾树的层次遍历



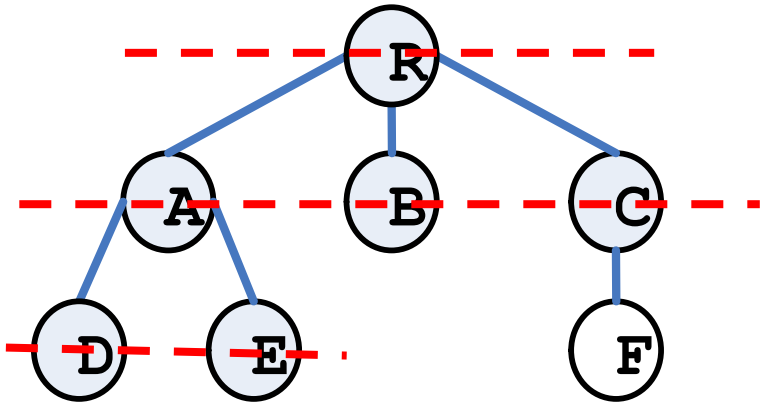
先进先出：

R A

B C D E

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历

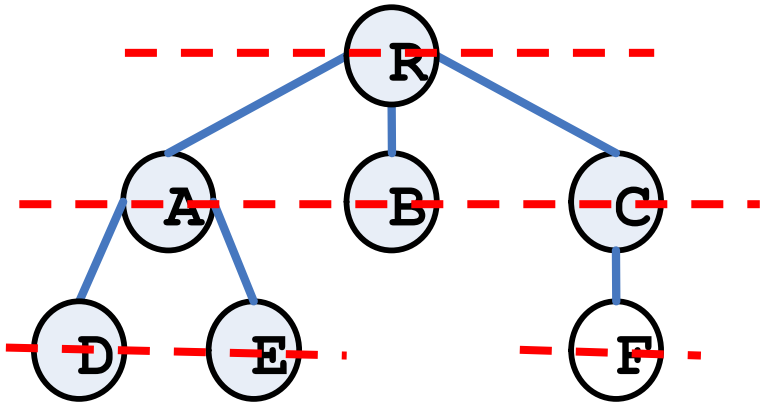


先进先出：

R A B C D E

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历

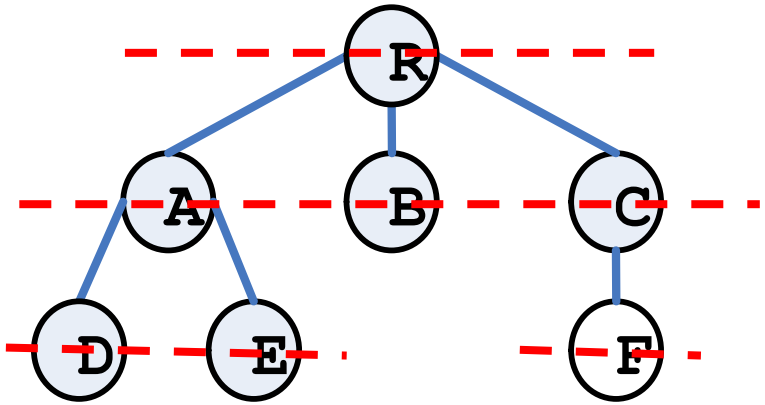


先进先出：

R A B C D E F

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历

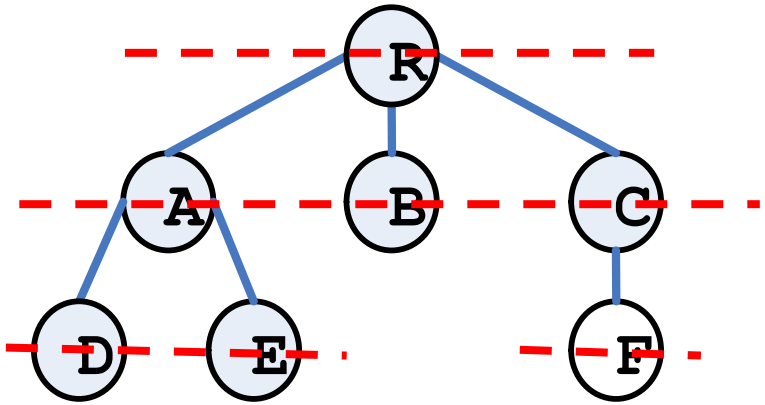


先进先出：

R A B C D E F

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历

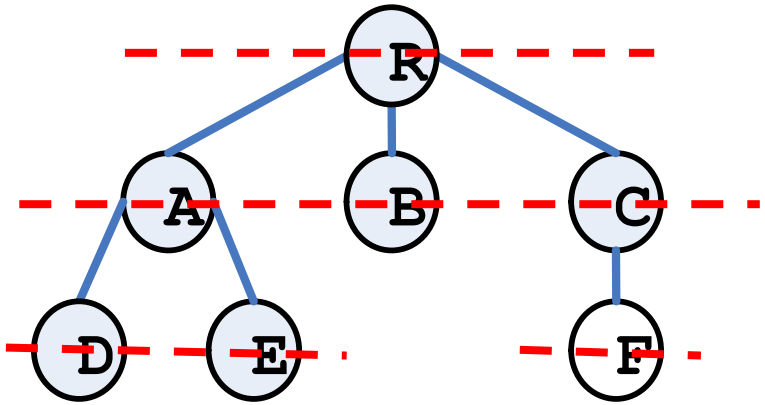


先进先出：

R A B C D E F

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历

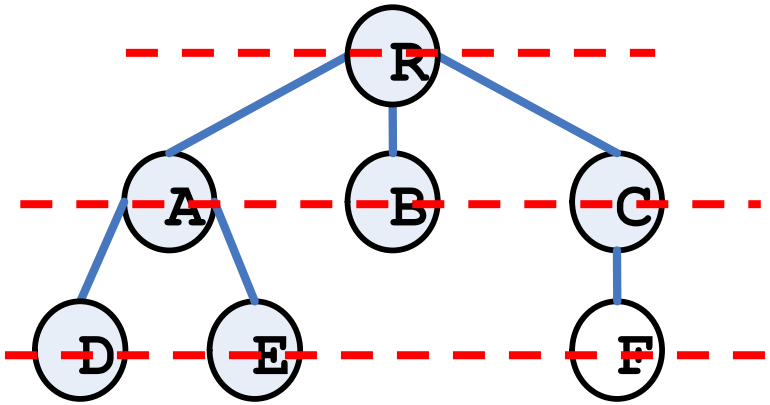


先进先出：

R A B C D E F

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

回顾树的层次遍历



先进先出：

R A B C D E F

```
BFS(R){
    R入队;
    while(队列不空){
        队头v出队并访问;
        v的孩子依次入队;
    }
}
```

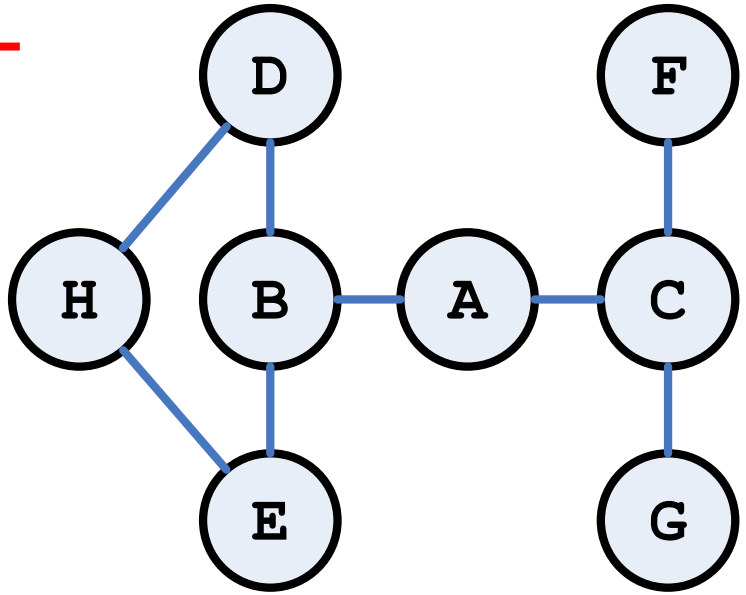
BFS (G)

```
For(每个v) flag[v] = 0;
InitQueue (Q);  //初始化队列
while(有未访问的顶点v){
  EnQueue(Q, v); //顶点作为起点入队;
  while( !IsEmpty(Q) )  //队列不空
   1)DeQueue(Q,u);visit(v);flag[v] = 1;
   2)for(u的未被访问邻接点w) EnQueue(Q, w);
  }
}
```

图的遍历：广度优先遍历

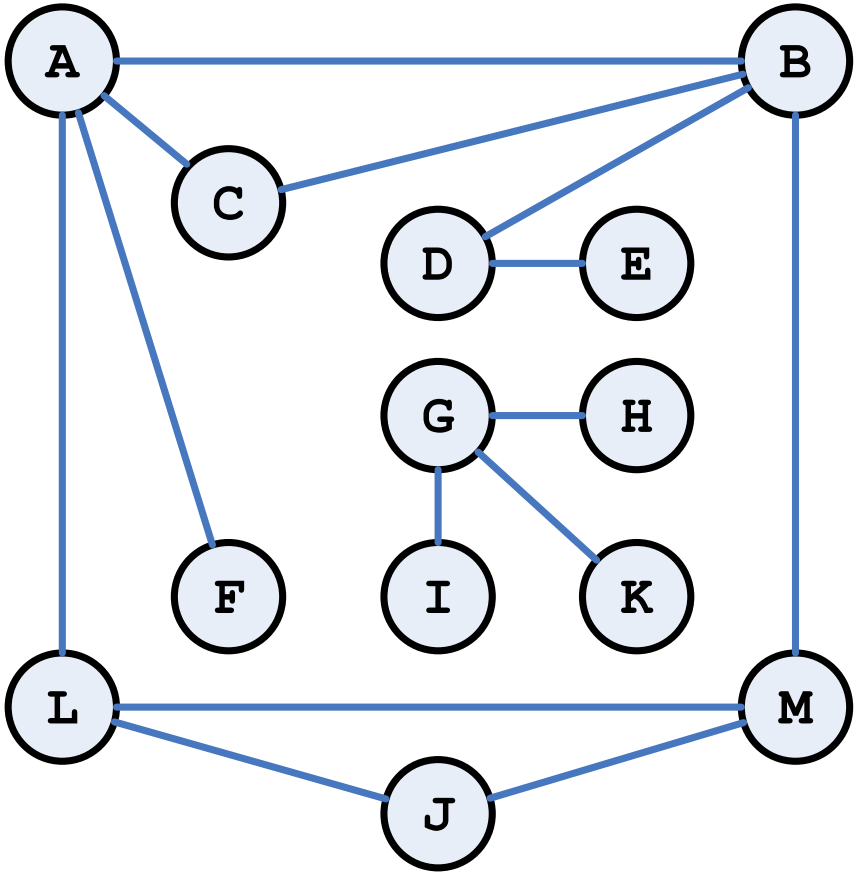
- 练习
  - 写出对下图进行广度优先遍历的结果

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

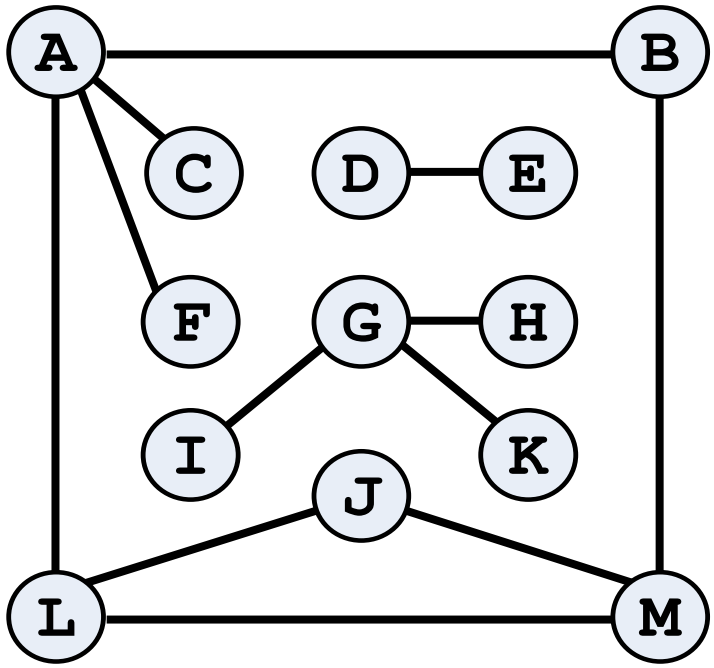
- 问题：

  图的存储结构中都有顶点表，为什么还要这么麻烦的遍历呢？直接访问顶点表不是更简单快捷吗？

- 回答：
  - 遍历算法除访问顶点外，还经过边，因此，可以解决连通性相关问题，如求连通分量、生成树、判断两个顶点是否连通等（即是否有路径）

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

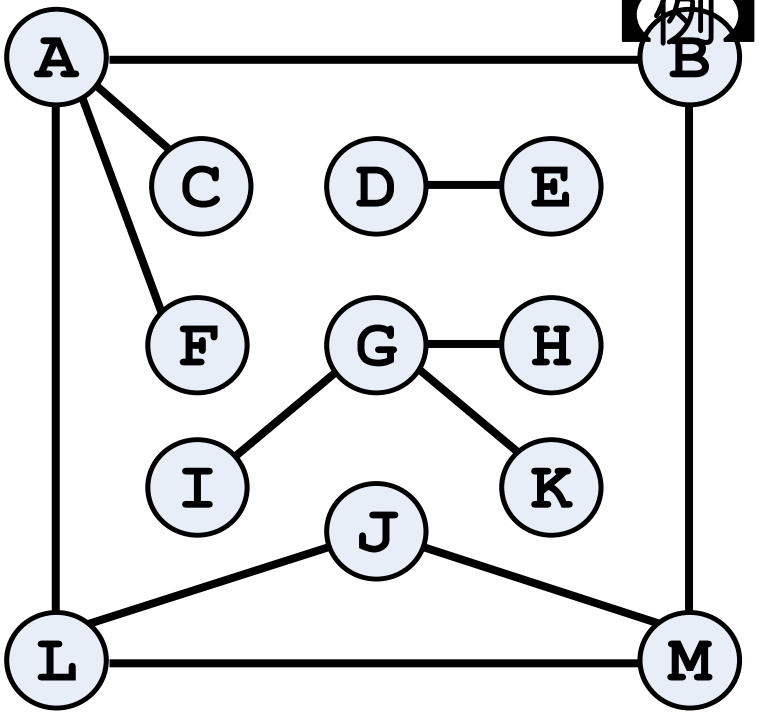


  - 3个连通分量：
    - **ALMJBFC**
    - **DE**
    - **GKHI**

图的连通性问题：连通分量

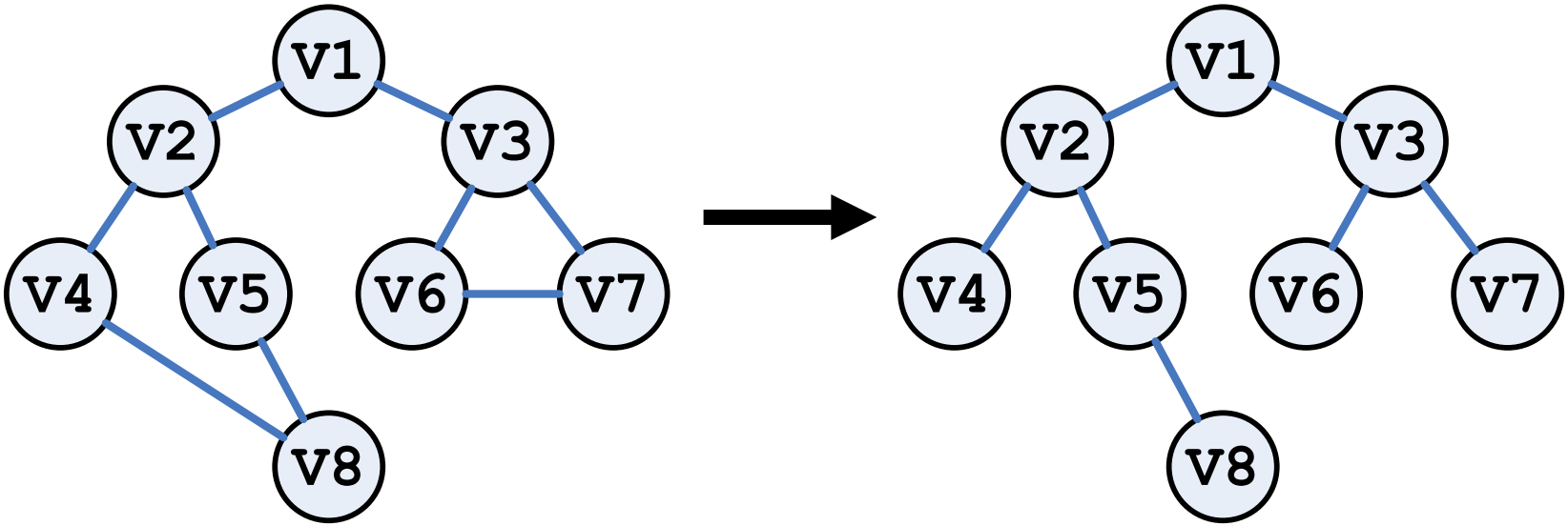
- 求无向图的连通分量
  - 从每个顶点出发进行遍历（深度或广度优先）
  - 每一次从一个新的起点出发进行遍历得到的顶点序列就是一个连通分量的顶点集合

【例】



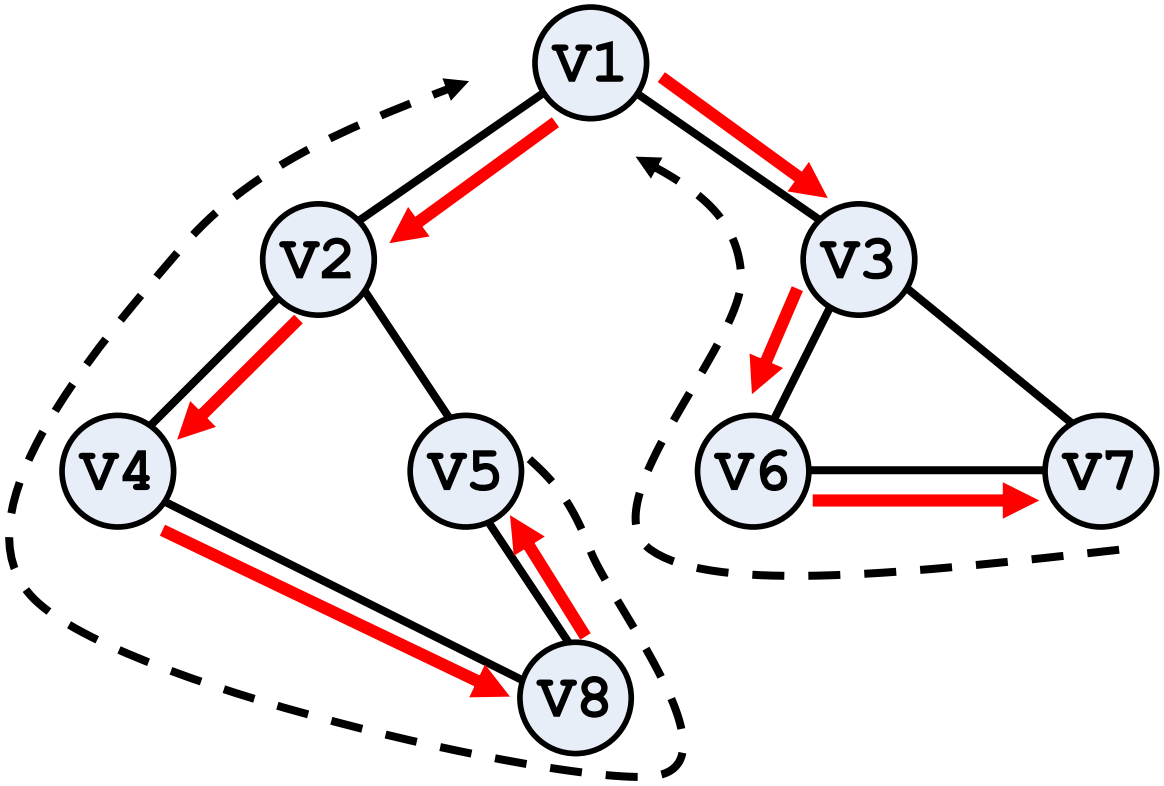
图的连通性问题：生成树

- 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



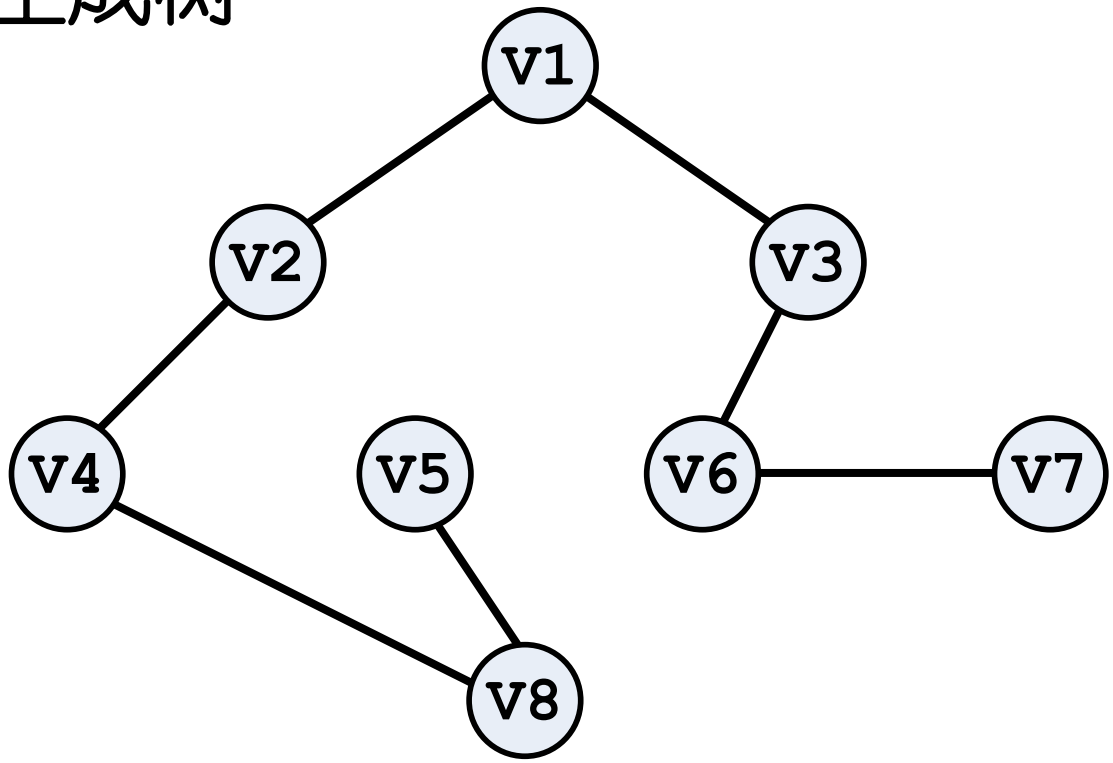
图的连通性问题：生成树

- 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



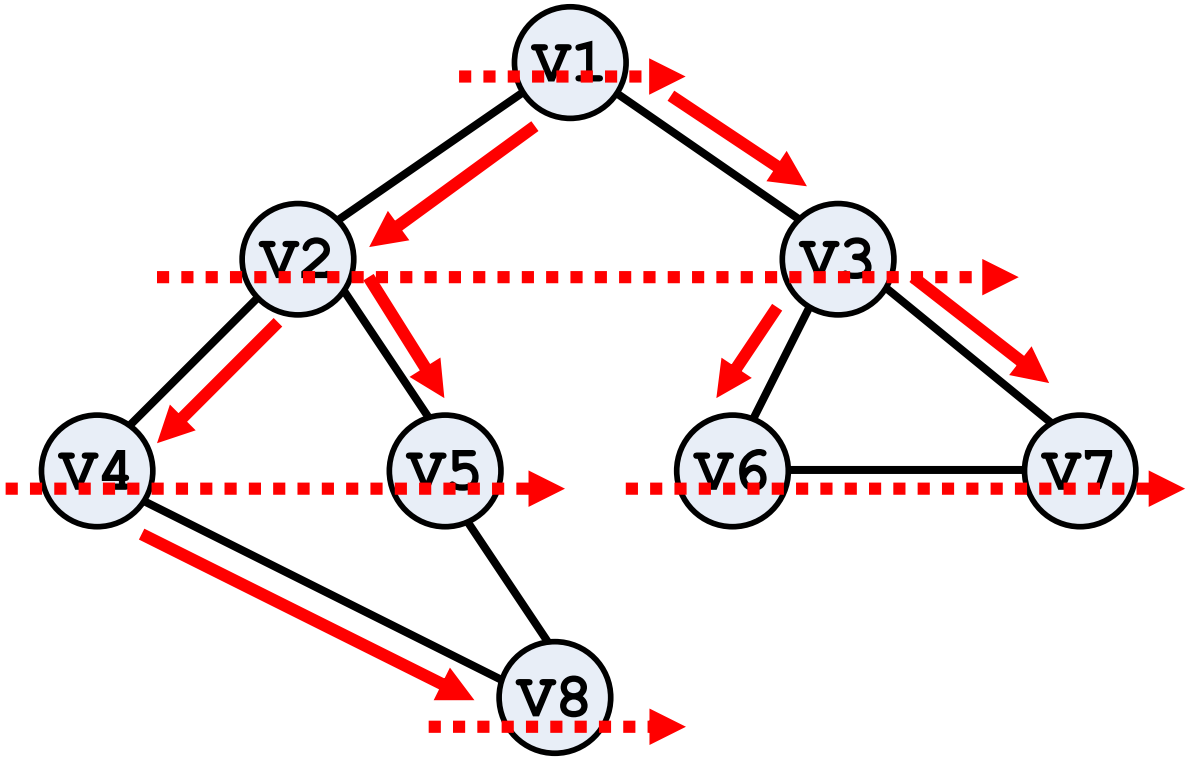
图的连通性问题：生成树

- 深度优先生成树：深度优先遍历得到的生成树



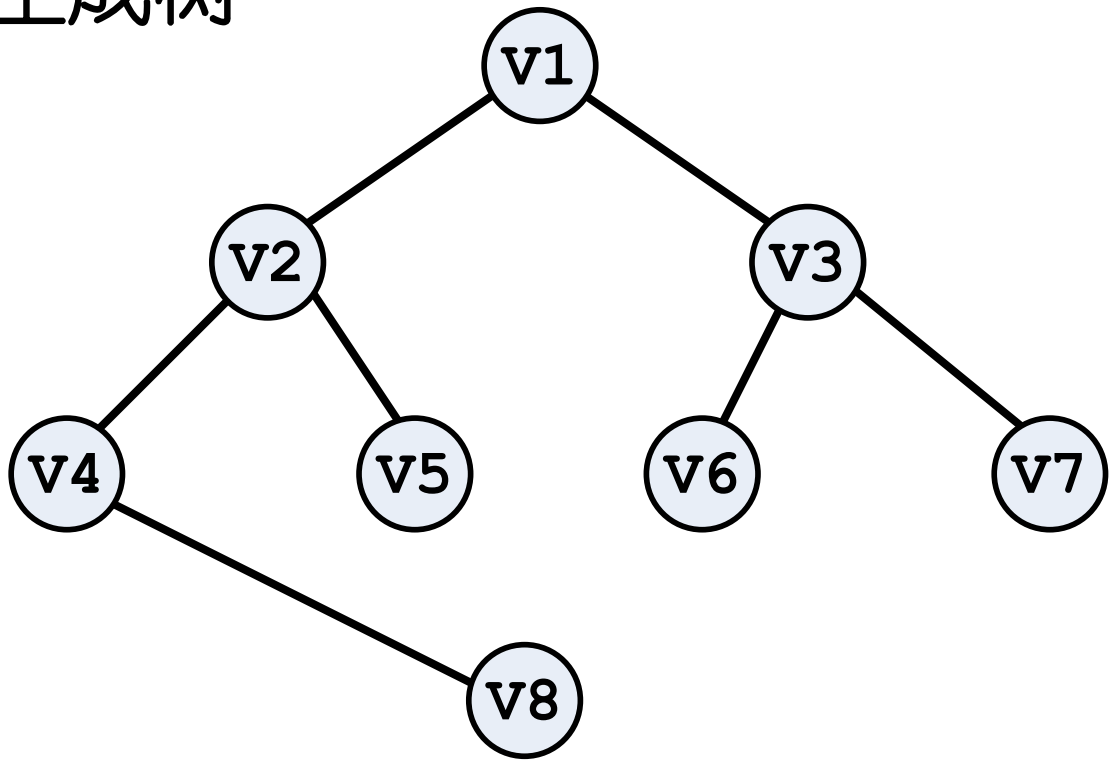
图的连通性问题：生成树

- 或者进行广度优先遍历：



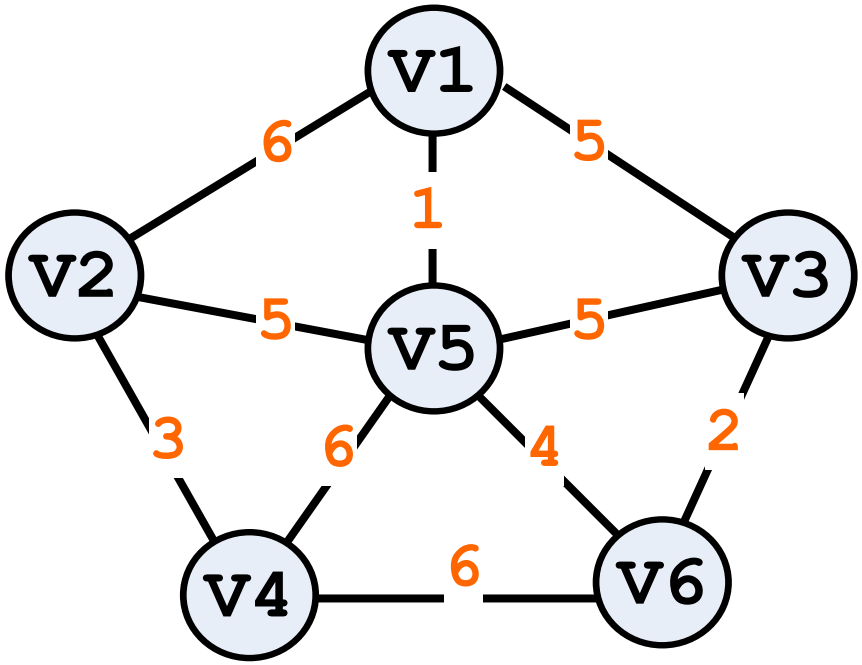
图的连通性问题：生成树

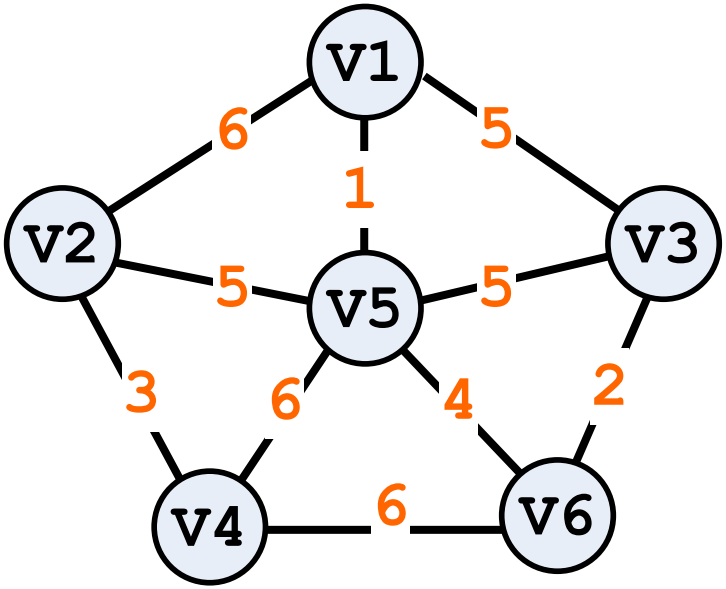
- 广度优先生成树：广度优先遍历得到的生成树

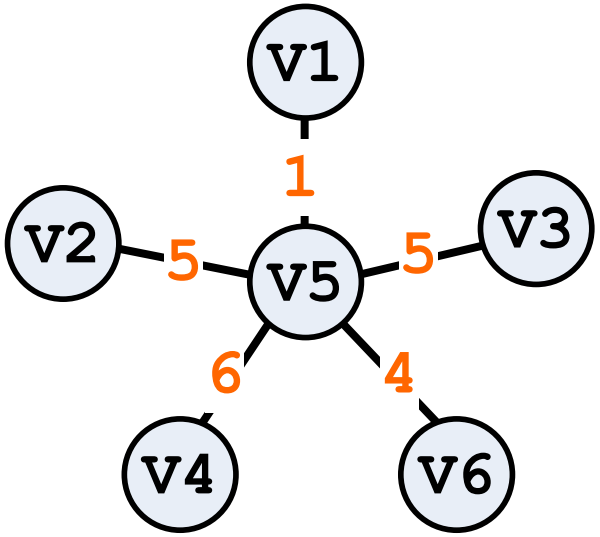


图的连通性问题：最小生成树

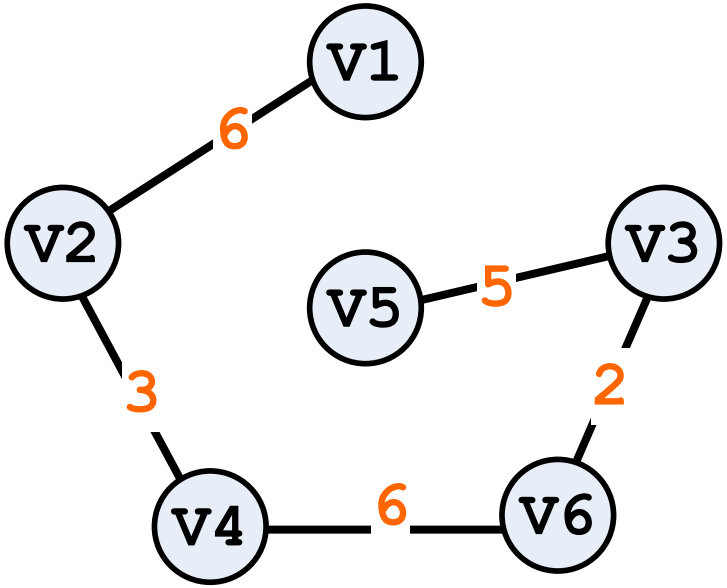
- 应用：连通n个城市的需要多少条道路？
  - 最多n(n-1)/2：任何两个城市之间都修一条路
  - 最少n-1，问题是哪n-1条能使总代价最小呢？




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


V1
V2
V3
V5
V4
V6
1
5
5
6
4

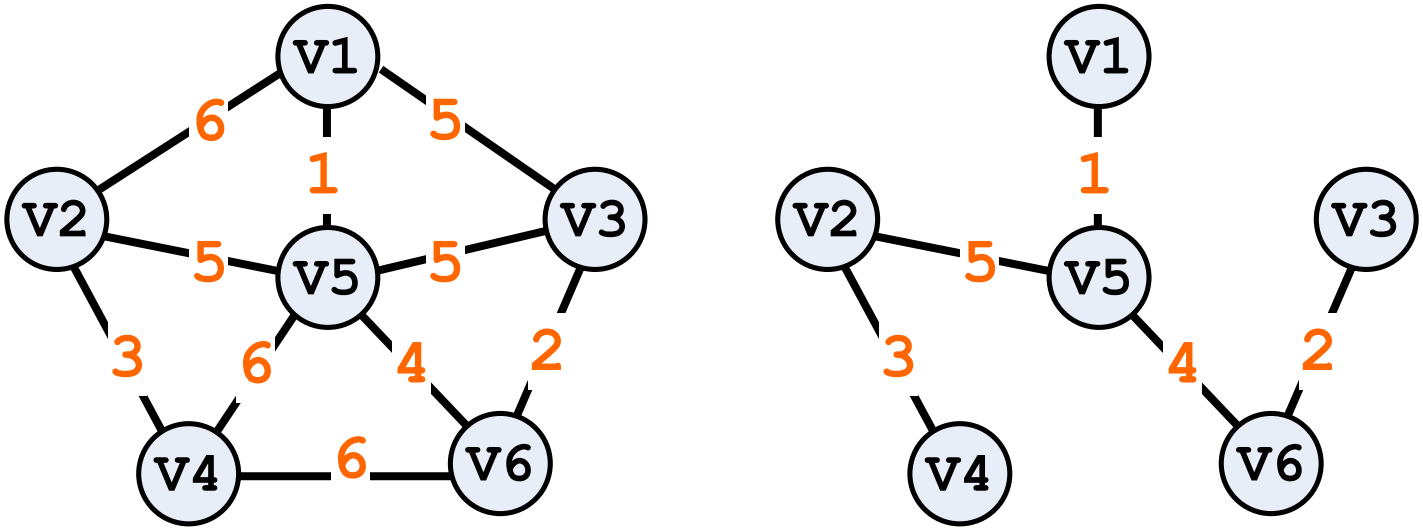
总长21


V1
V2
V3
V5
V4
V6
6
5
3
2
6

总长22

图的连通性问题：最小生成树

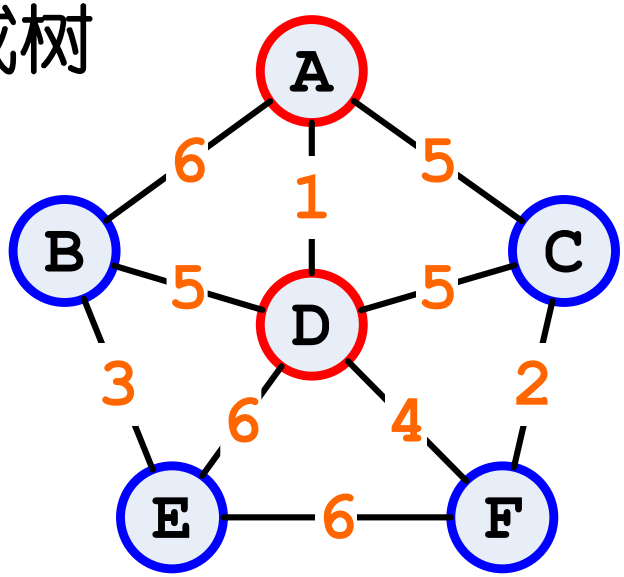
- 最小生成树
  - **Minimum Cost Spanning Tree**
  - 各边权值之和最小的生成树



- 问题：最小生成树唯一么？

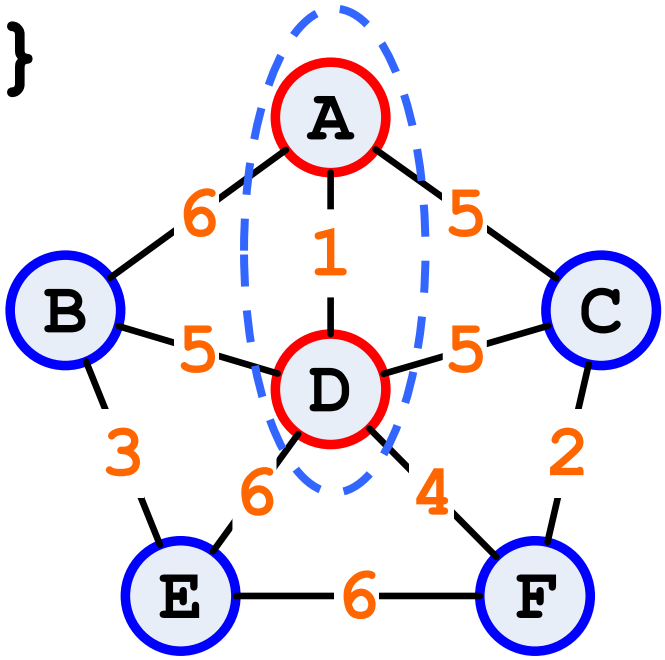
图的连通性问题：最小生成树

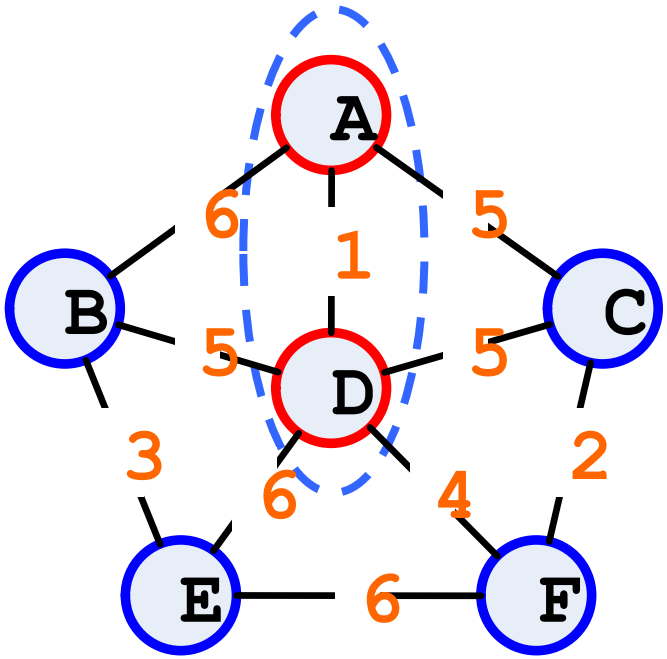
- 最小生成树的性质（MST性质）：
  - 设连通网 N=(V,{E})
  - U为V的非空子集
  - F={(v1,v2)|(v1,v2)∈E,v1∈U,v2∈V-U}
  - 设(u,v)是F中权值最小的边，则必存在一棵包含(u,v)的最小生成树



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)}



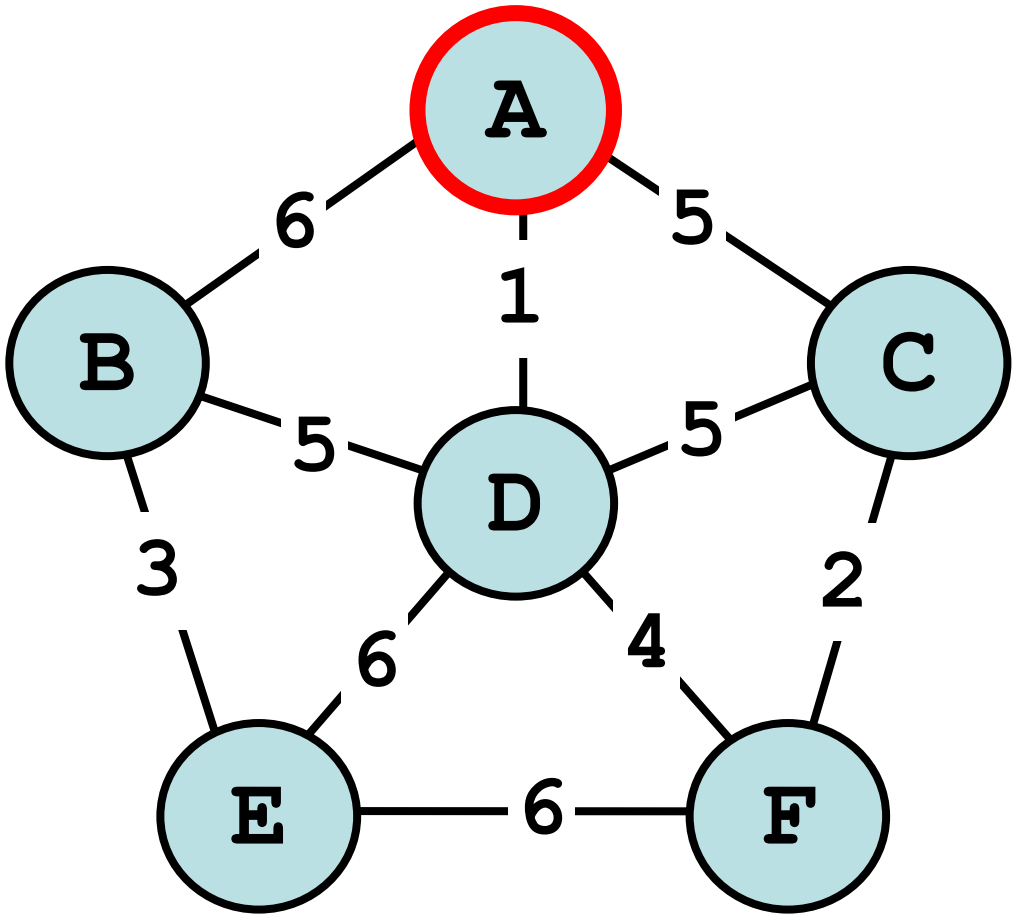


- F = {(A,B),(A,C),
  (D,B),(D,C),
  (D,E),(D,F)}
- (u,v)是F中权值最小的一条边
  = (D,F)
- 结论是：N的最小生成树中一定有一棵包含了(D,F)

最小生成树

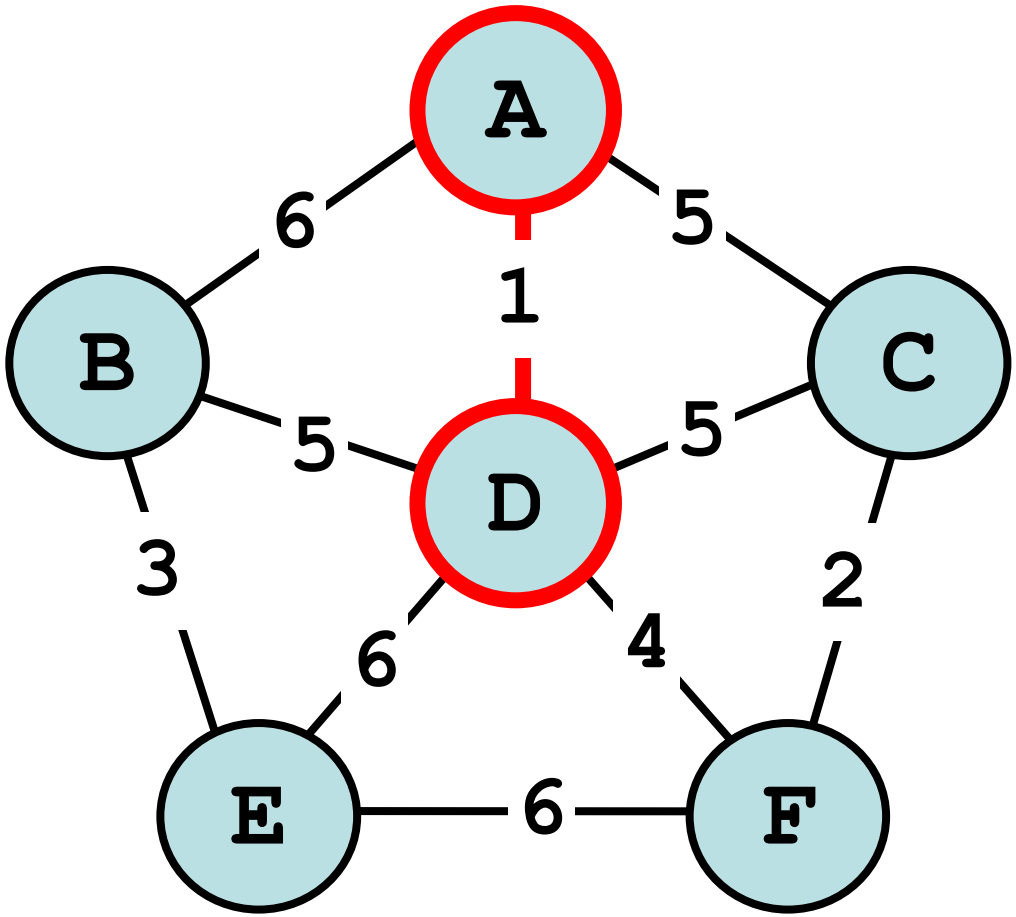
- **Prim算法**
  - U为最小生成树中顶点的集合
    - 初始时，$U=\{u_0\}$
  - 从剩下的顶点中找到一个离U最近的直接相连的顶点v，把它加入U
  - 重复，直到所有的顶点都加入到U中

- **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



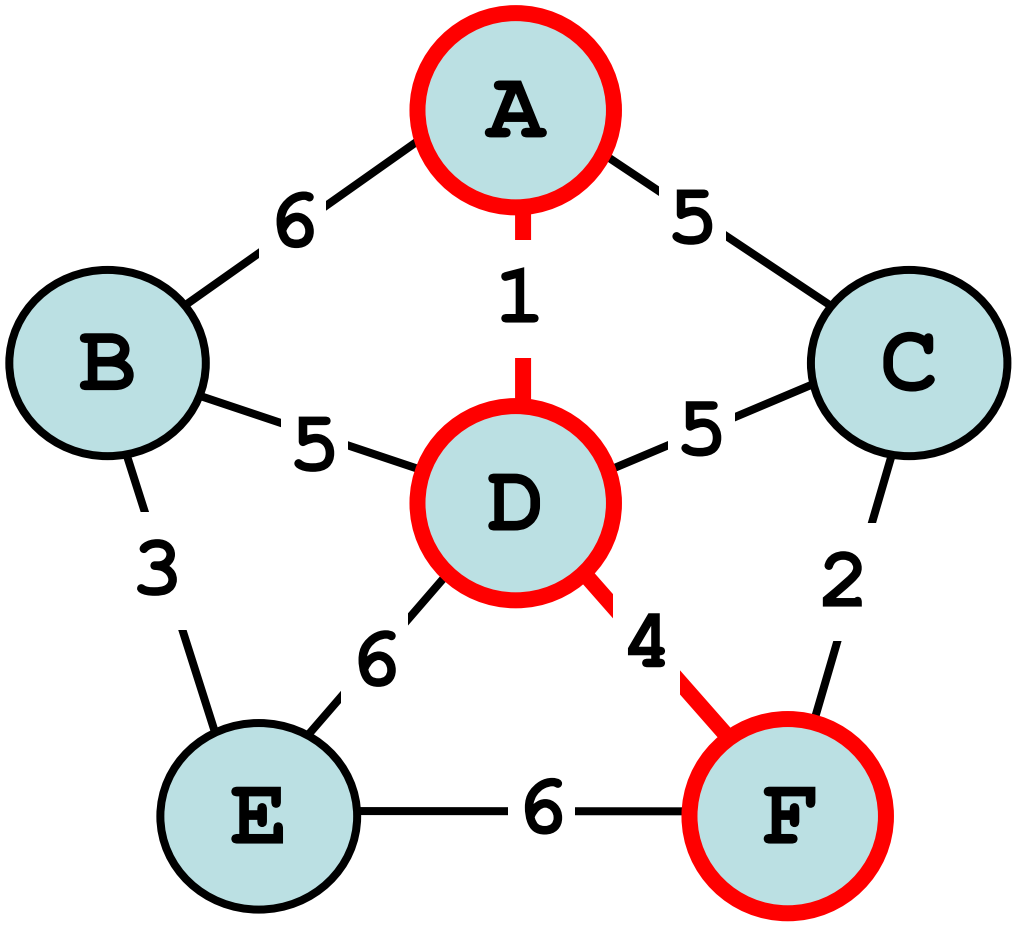
U = {A}

- **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



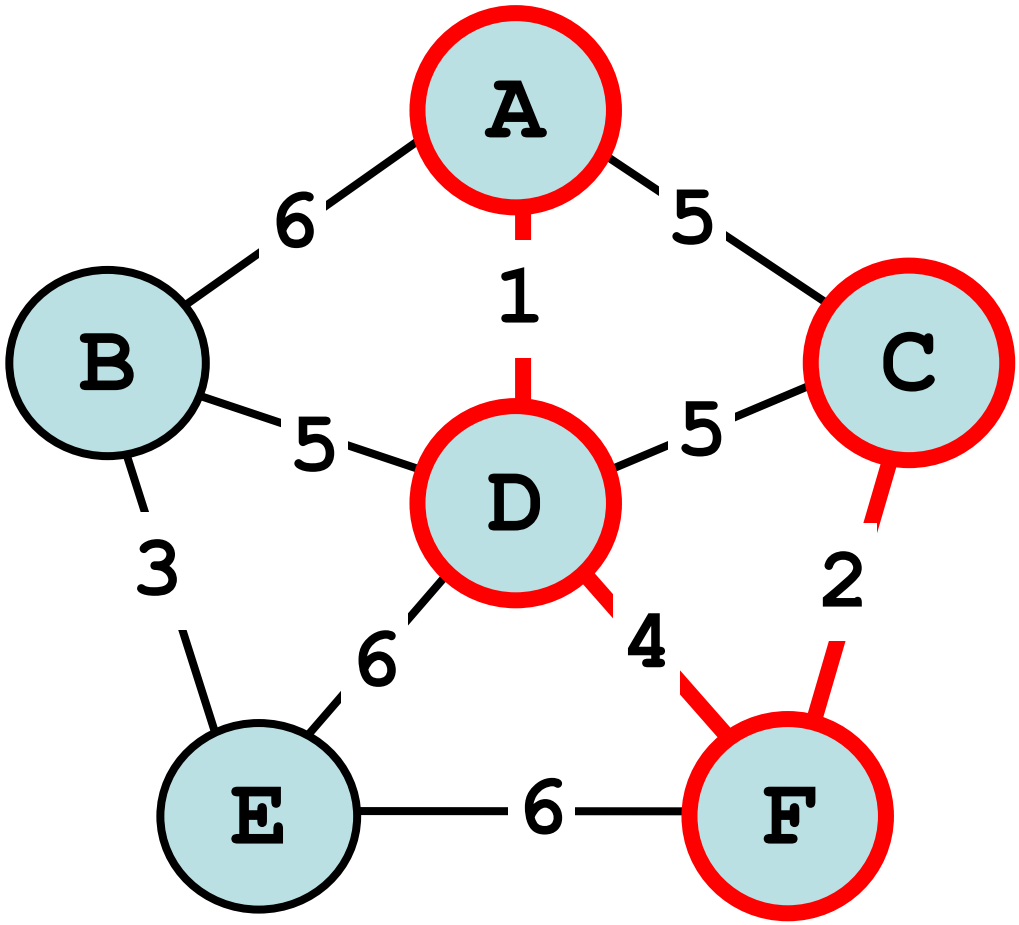
U = {A,D}

- **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



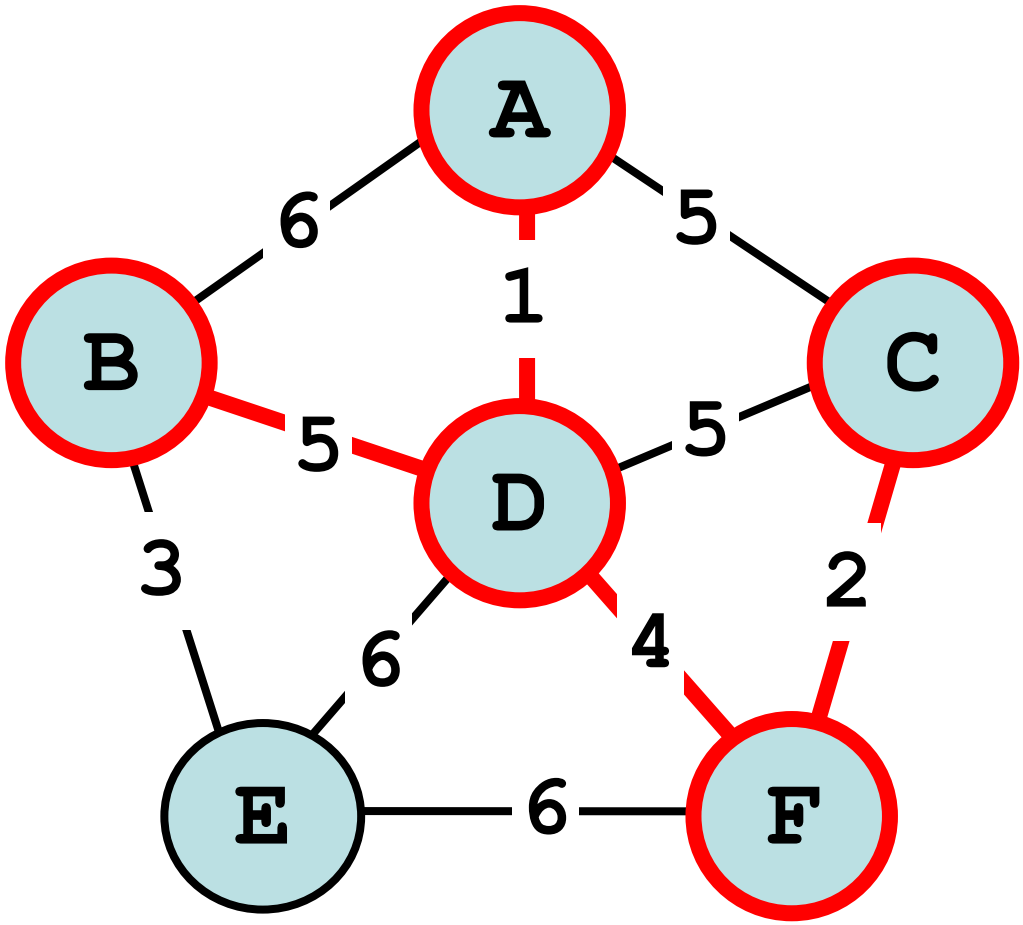
U = {A,D,F}

- **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



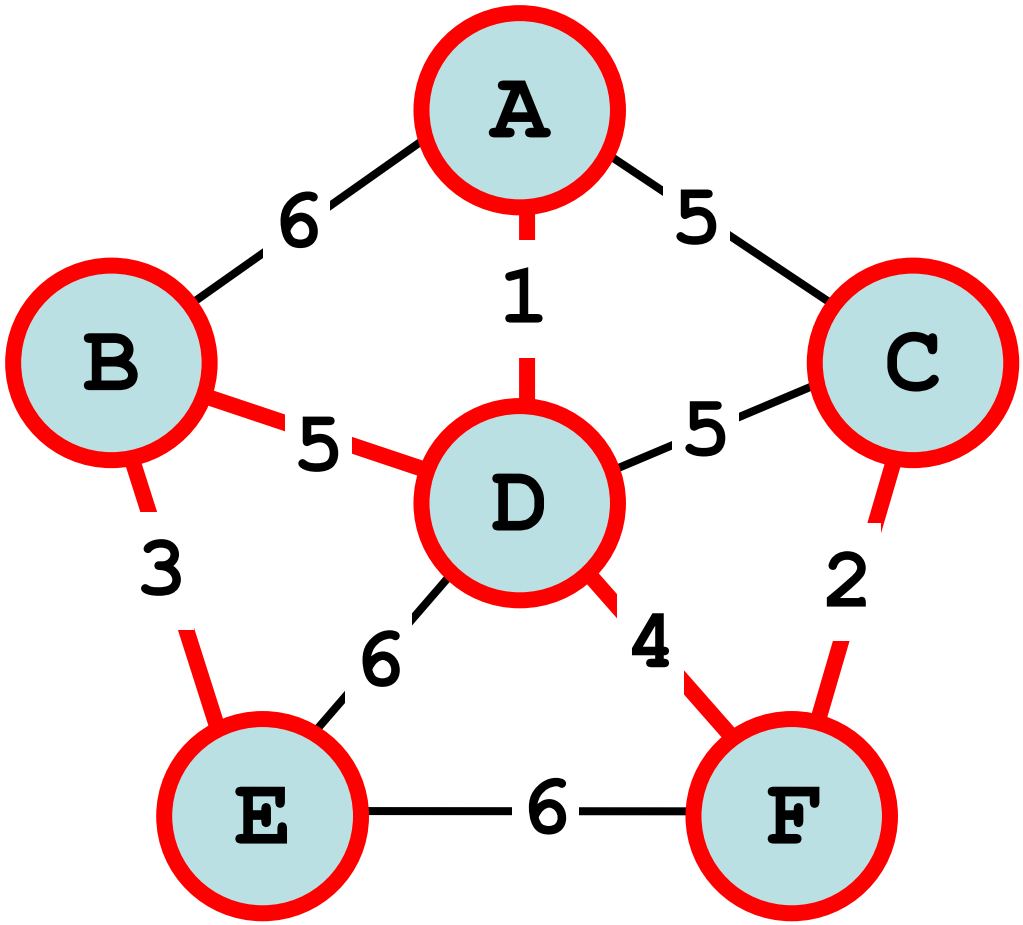
U = {A,D,F,C}

- **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



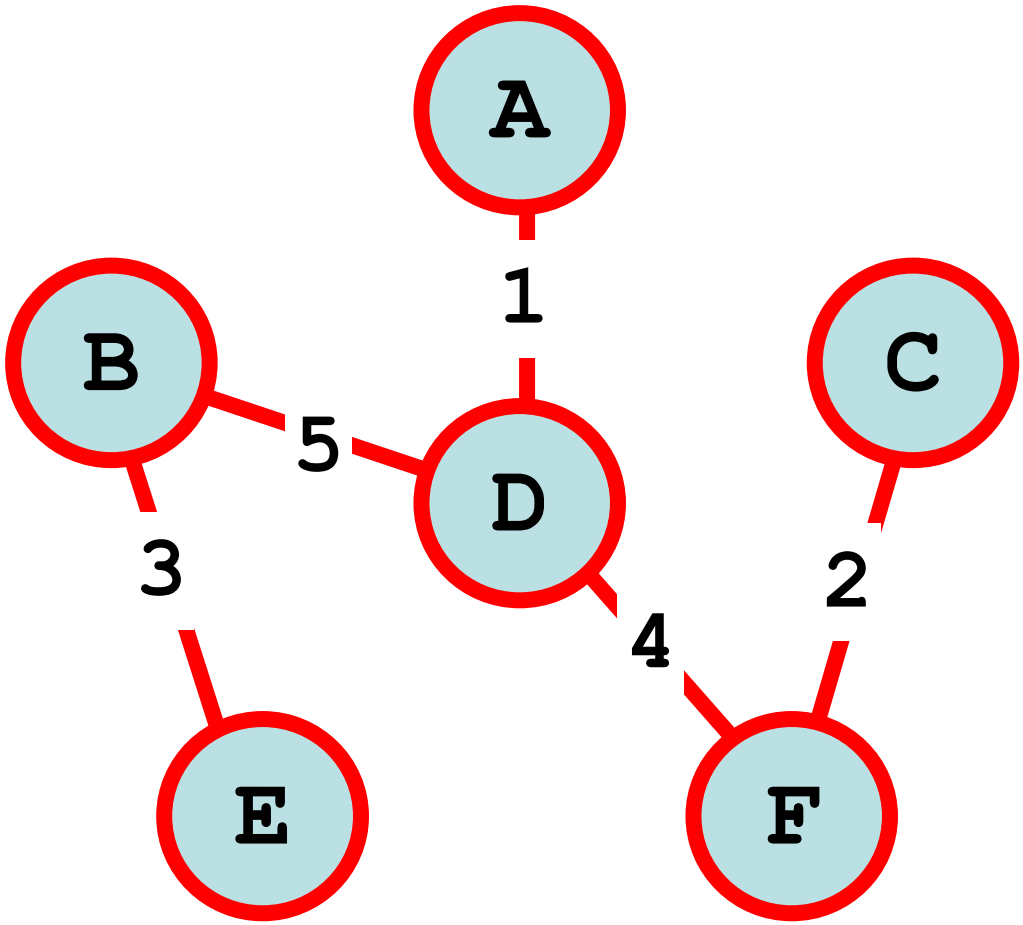
U = {A,D,F,C,B}

- **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}

Prim算法

- 需要将顶点集合分为U和V-U,关键是在连接U和V-U选择一条最小权边(u,v)

- 定义V-U中顶点v到U的距离为

  cost(v)=Min{cost(uj,v)| uj∈U}

22 v_{ik}

Prim算法:最小代价边数组

记录V－U中每个顶点到集U的最小代价边的辅助数组:

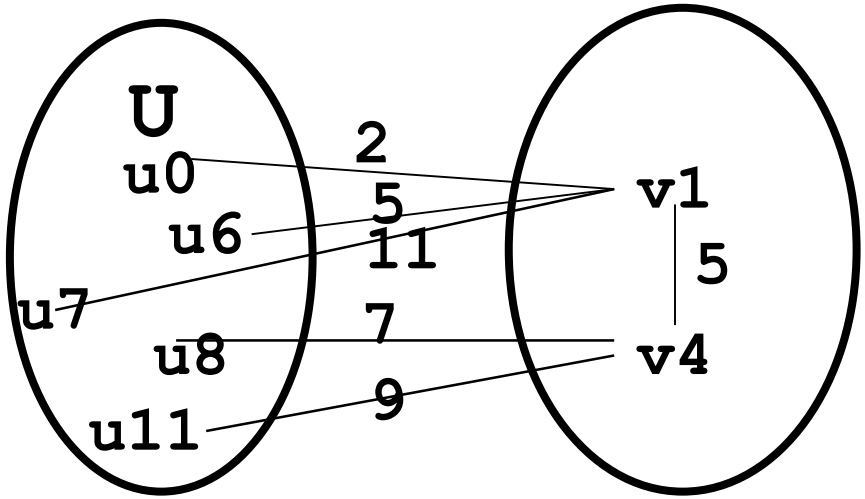
```
struct {
  Vertex adjvex;          //u_k∈U
  float lowcost;    //Min{cost(u_j,v_i)| u_j∈U}
} closedge[VNUM];
```

closedge[i] = Min{cost(u_j, v_i)| u_j∈U, v_i∈V-U}, 表示v_i到U的最小距离，即与v_i构成该最小距离的那个u_j∈U. 一旦某个v_i属于U，则令相应的closedge[i].

lowcost = 0，表示该点已属于U而不是V-U了。

因此closedge既记录了V-U中顶点v到U的最小距离，也区分了顶点集合为U和V-U

Prim算法:最小代价边数组



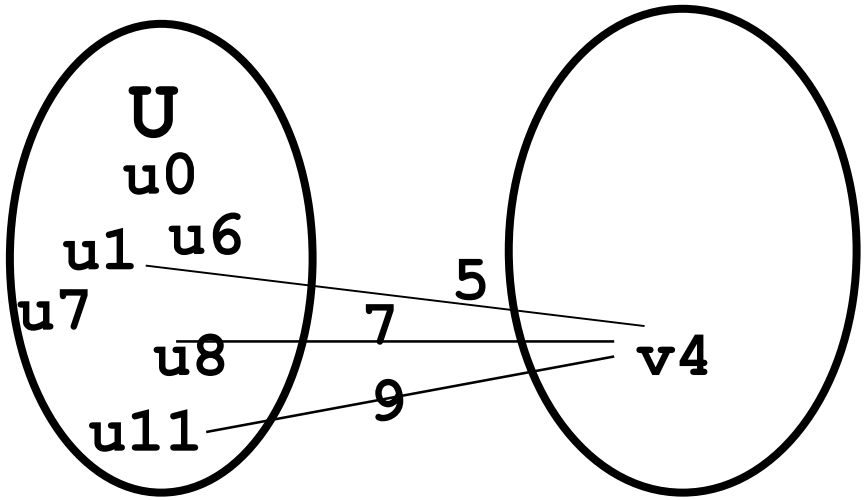
closedge[1] = {u0,2}

closedge[4] = {u8,7}

v1加入U后,V-U中其他顶点Vj到U的最短距离是?

设v1u0是最小权边, v1加入U后,看看V-U中剩余的vj与v1是否有边而且该边的权小于closedge[j].lowCost,如是,则更新closedge[j].lowCost

Prim算法:最小代价边数组




```
closedge[1] = {u0,0}
closedge[4] = {u1,5}
```

vk加入U后,看看V-U中剩余的vj与vk是否有边而且该边的权小于closedge[j].lowCost,如是,则更新closedge[j].lowCost

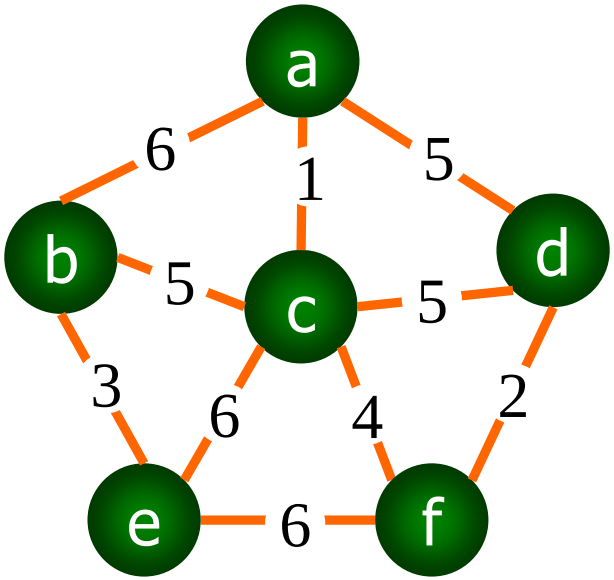
❖ **Prim算法**

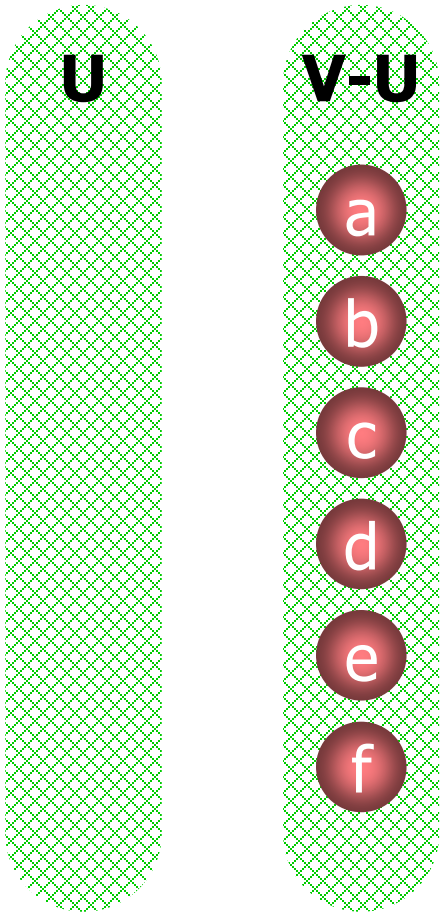
closedge

	adj	cost
0		
1		
2		
3		
4		
5		

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							101

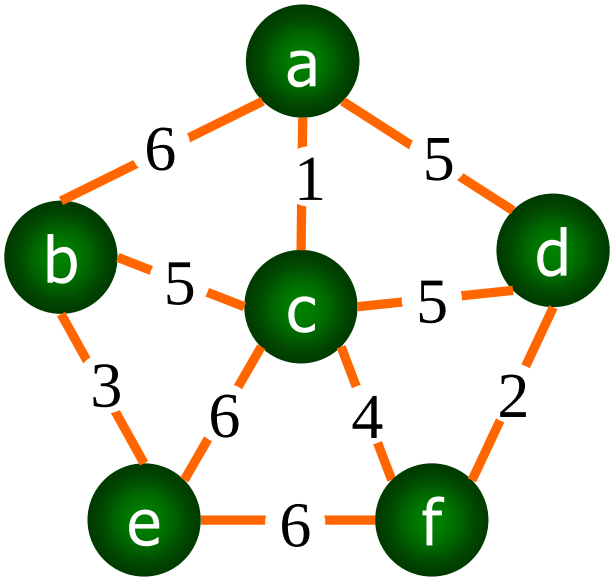
❖ **Prim算法**

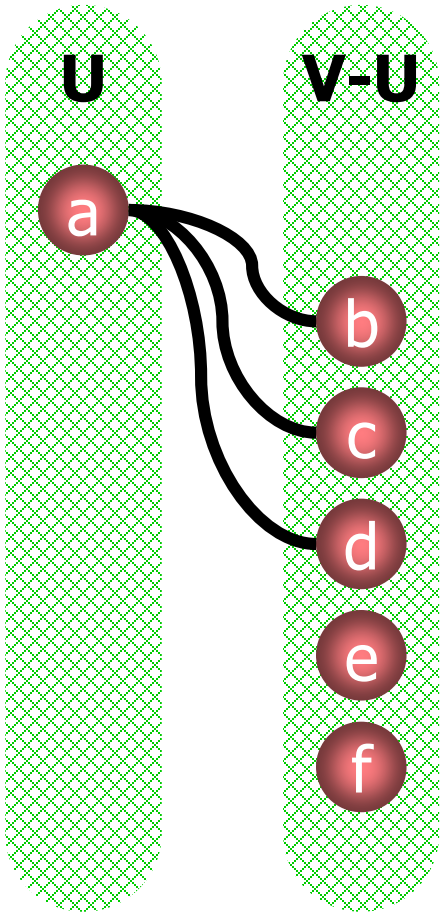
closedge

	adj	cost
0		0
1	0	6
2	0	1
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5
0	∞	6	1	5	∞	∞
1	6	∞	5	∞	3	∞
2	1	5	∞	5	6	4
3	5	∞	5	∞	∞	2
4	∞	3	6	∞	∞	6
5	∞	∞	4	2	6	∞

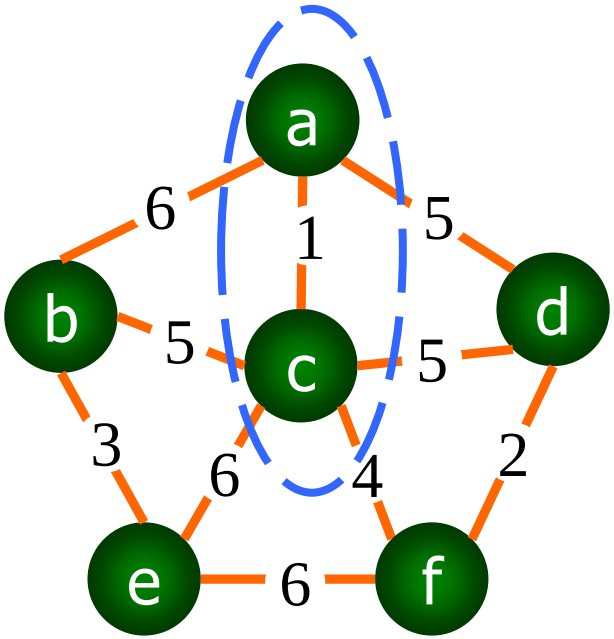
❖ **Prim算法**

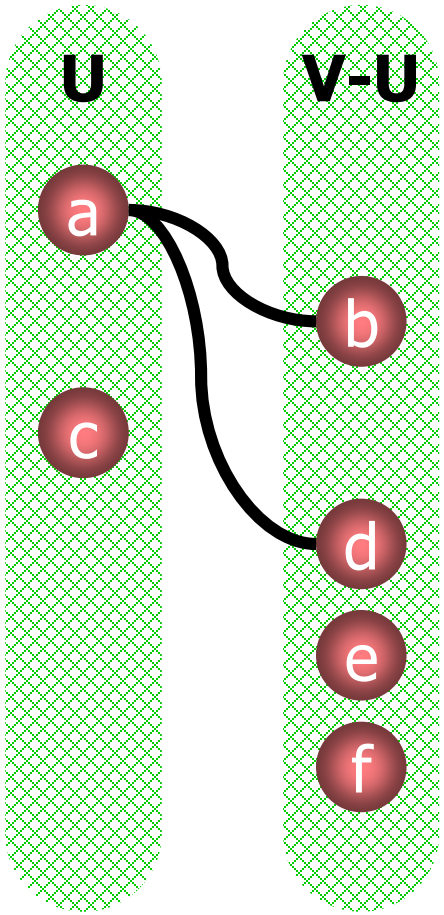
closedge

	adj	cost
0		0
1	0	6
2	0	1
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f

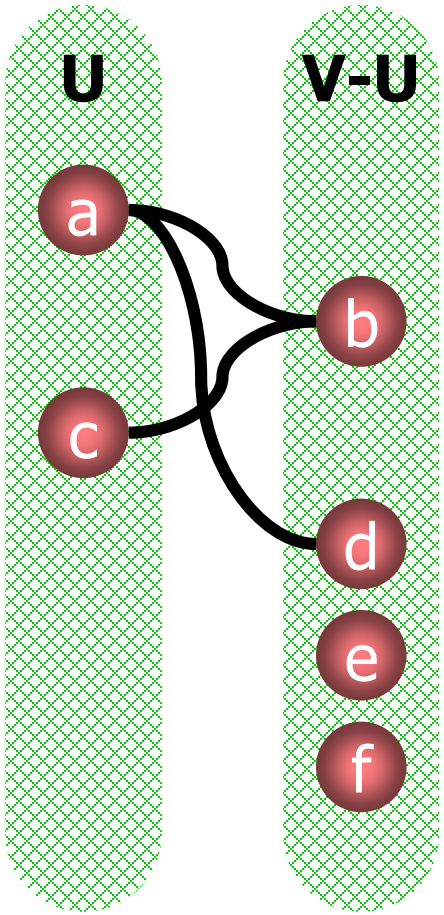




G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							103

❖ **Prim算法**

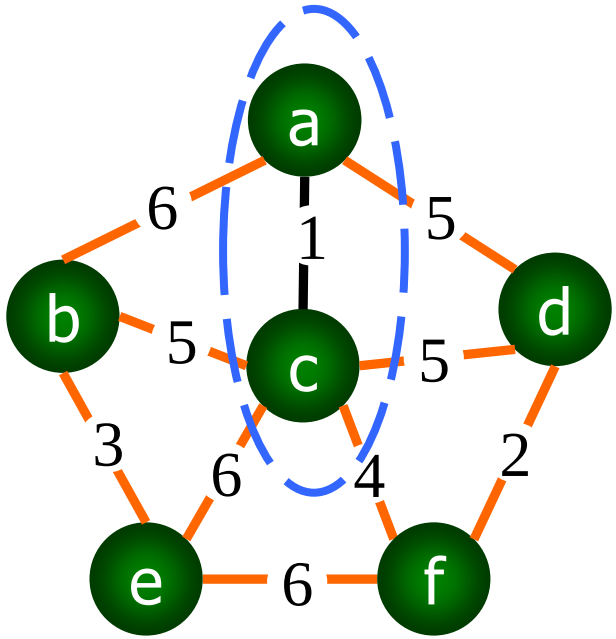


closedge

	adj	cost
0		0
1	0	6
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

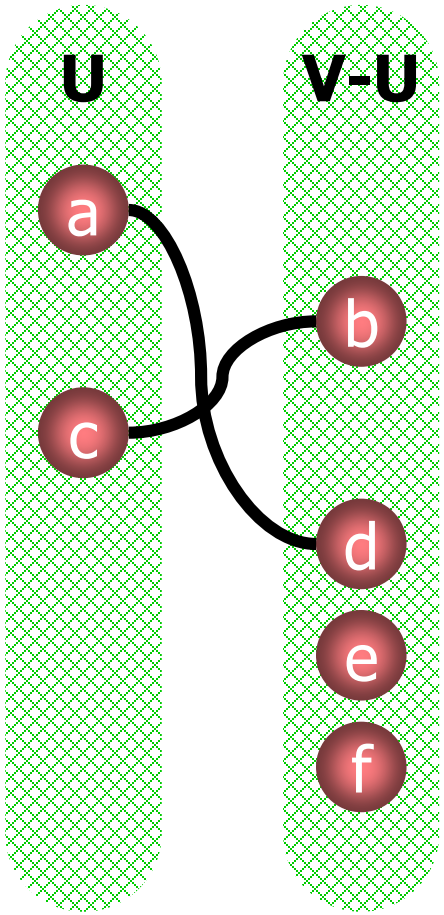
0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5
0	∞	6	1	5	∞	∞
1	6	∞	5	∞	3	∞
2	1	5	∞	5	6	4
3	5	∞	5	∞	∞	2
4	∞	3	6	∞	∞	6
5	∞	∞	4	2	6	∞

❖ **Prim算法**

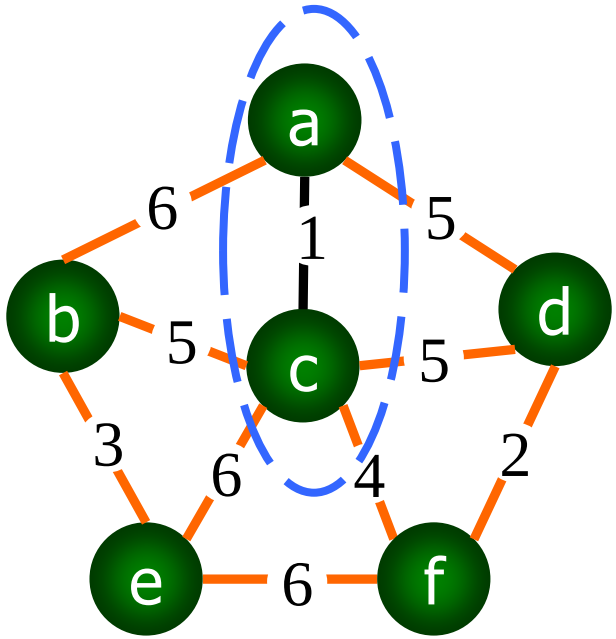


closedge

	adj	cost
0		0
1	0	6
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5
0	∞	6	1	5	∞	∞
1	6	∞	5	∞	3	∞
2	1	5	∞	5	6	4
3	5	∞	5	∞	∞	2
4	∞	3	6	∞	∞	6
5	∞	∞	4	2	6	∞

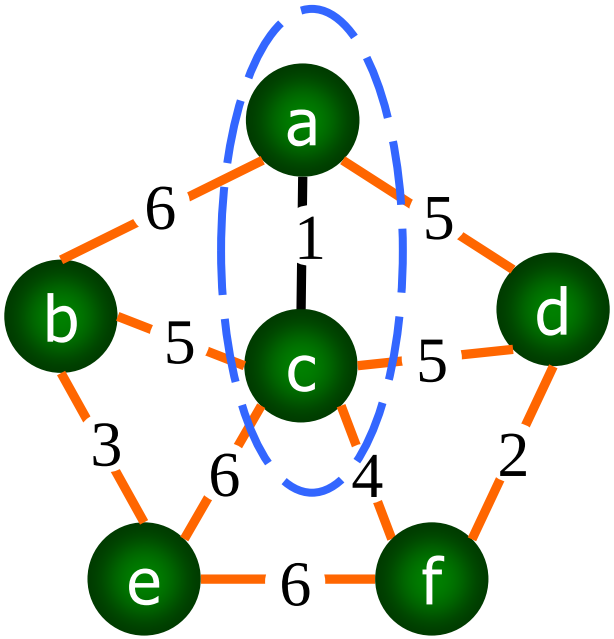
❖ **Prim算法**

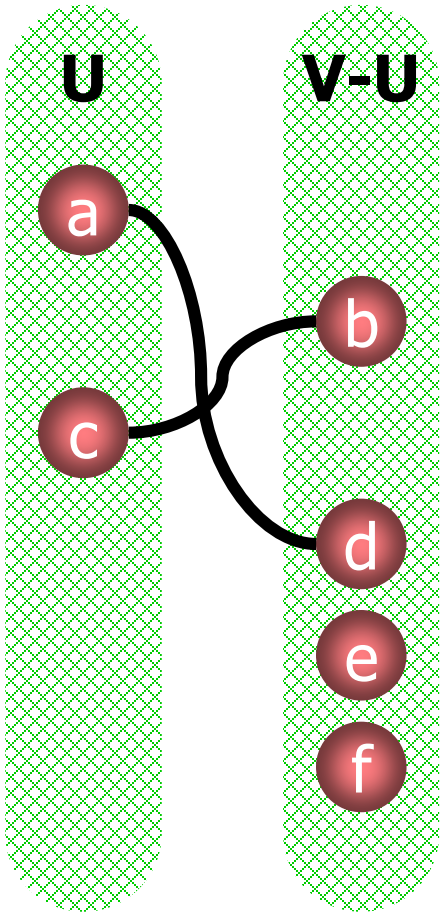
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							106

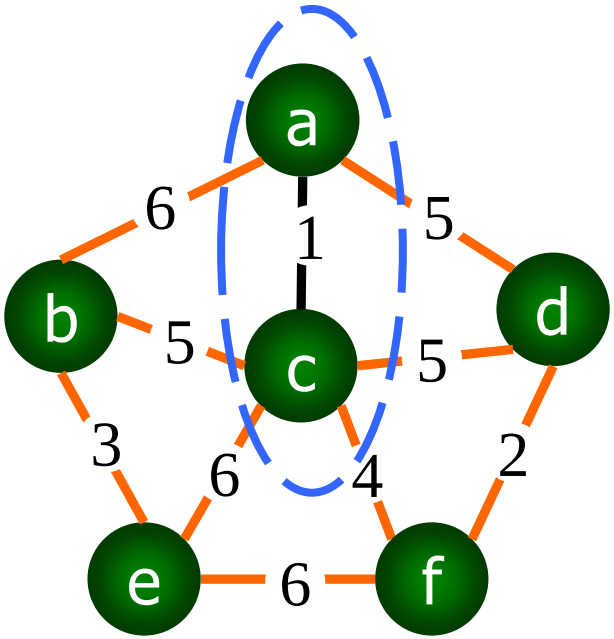
❖ **Prim算法**

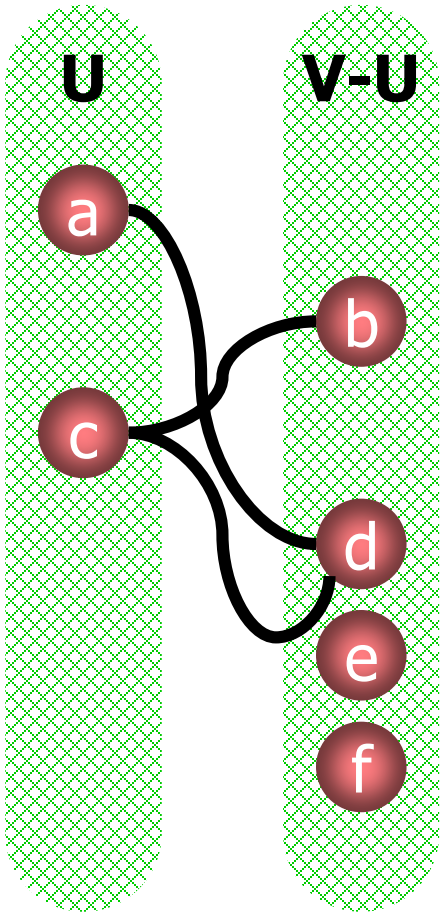
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							107

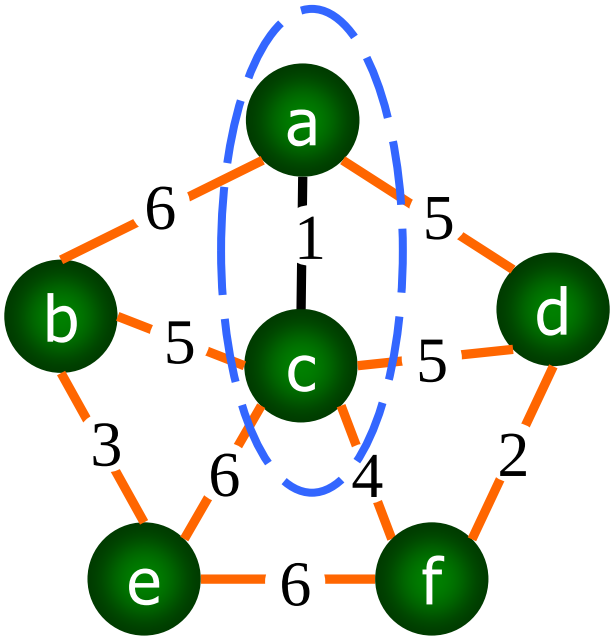
❖ **Prim算法**

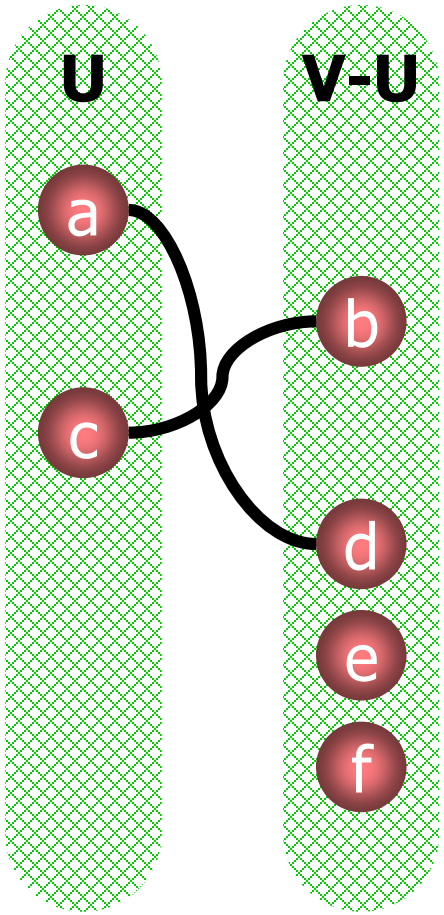
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							108

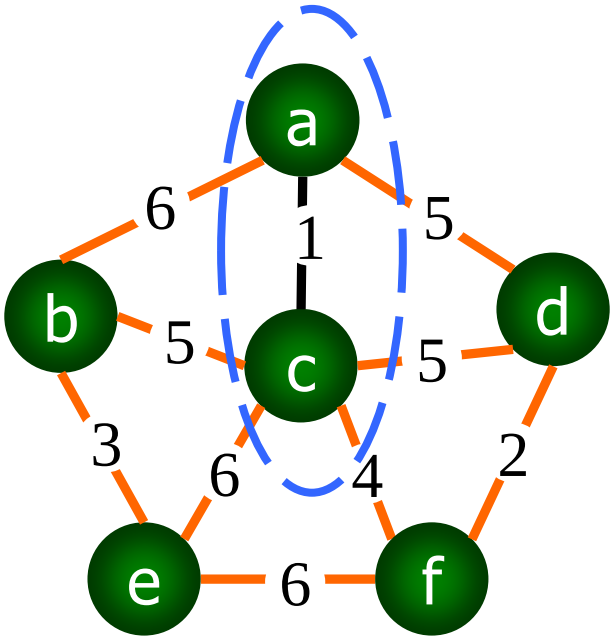
❖ Prim算法

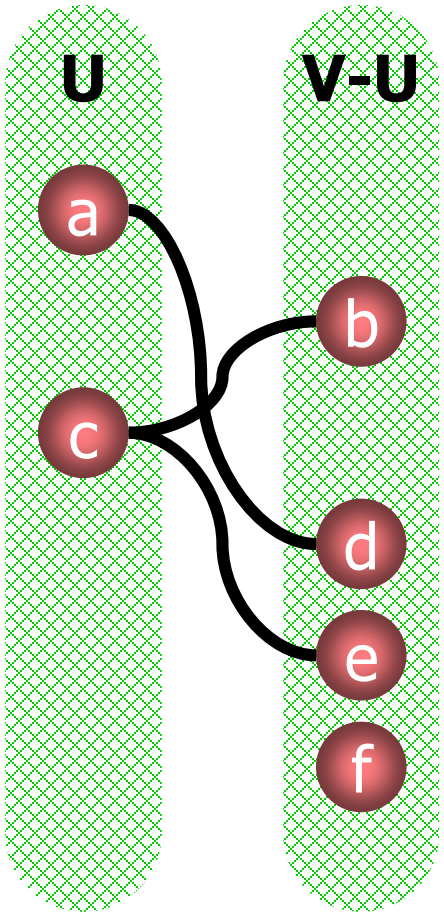
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	0	∞
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							109

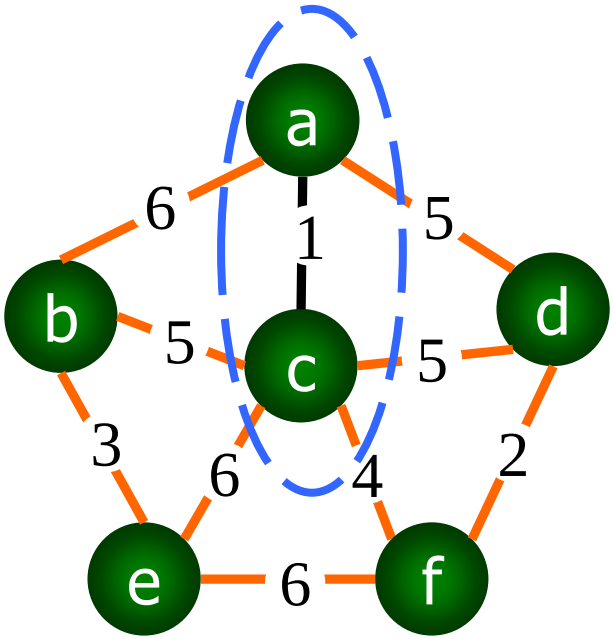
❖ **Prim算法**

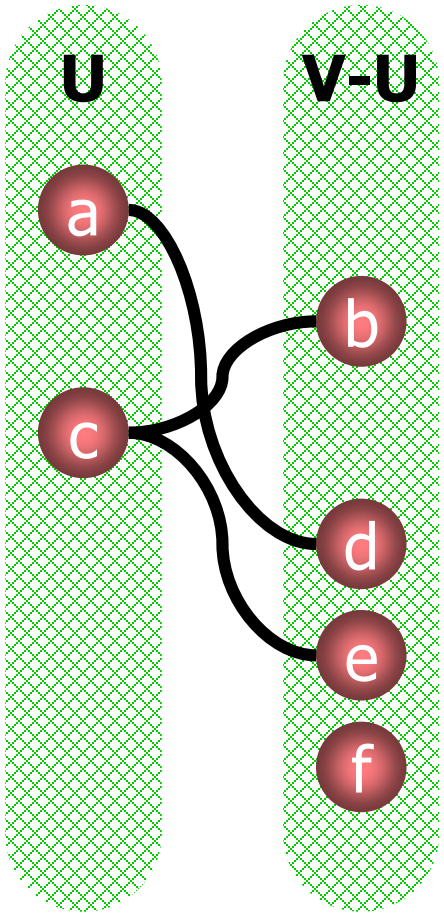
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							110

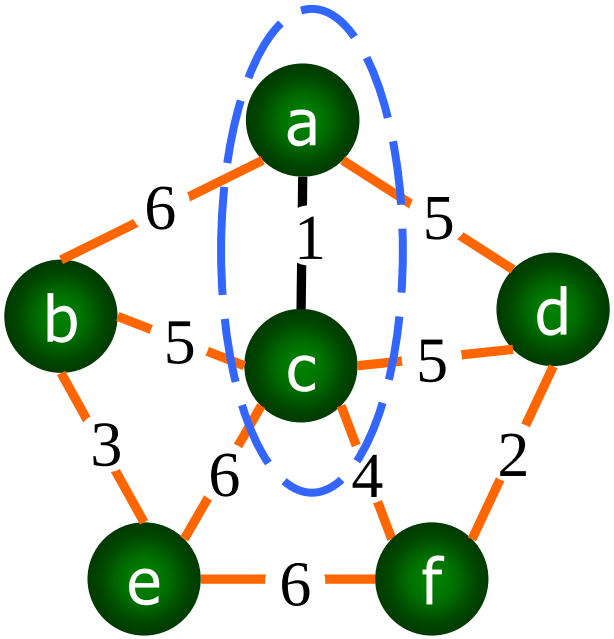
❖ **Prim算法**

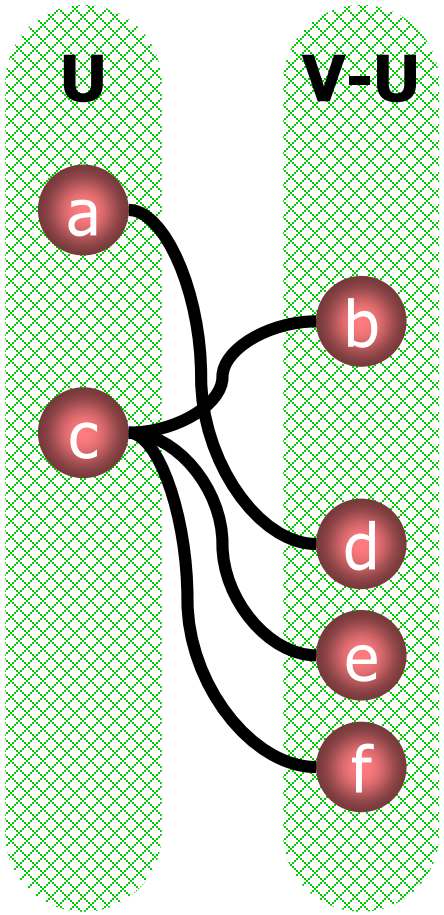
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	0	∞

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							111

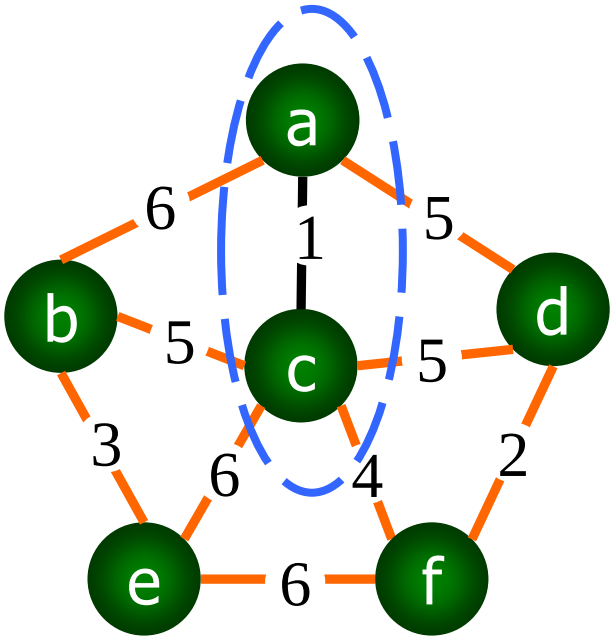
❖ **Prim算法**

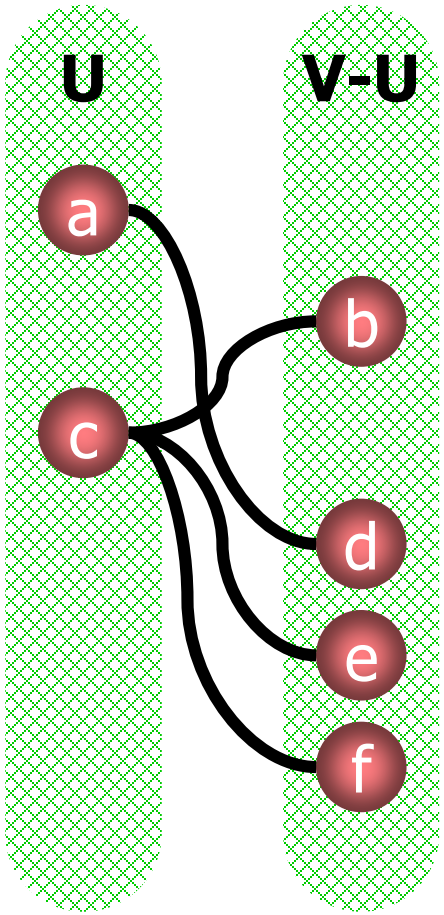
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	2	4

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							112

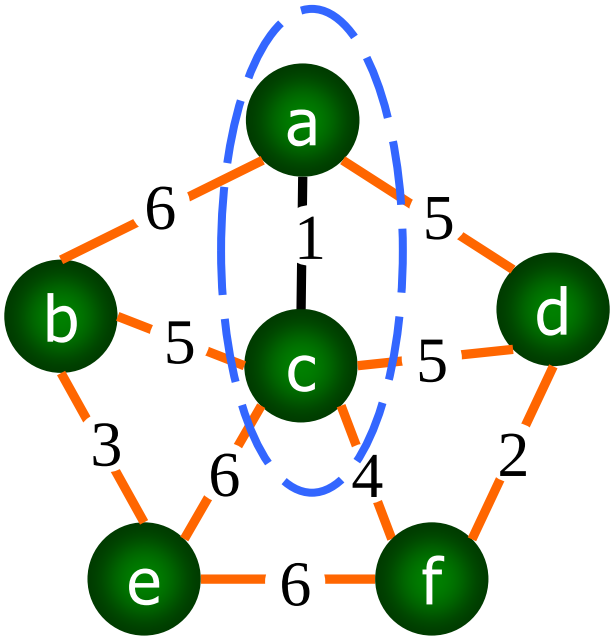
❖ **Prim算法**

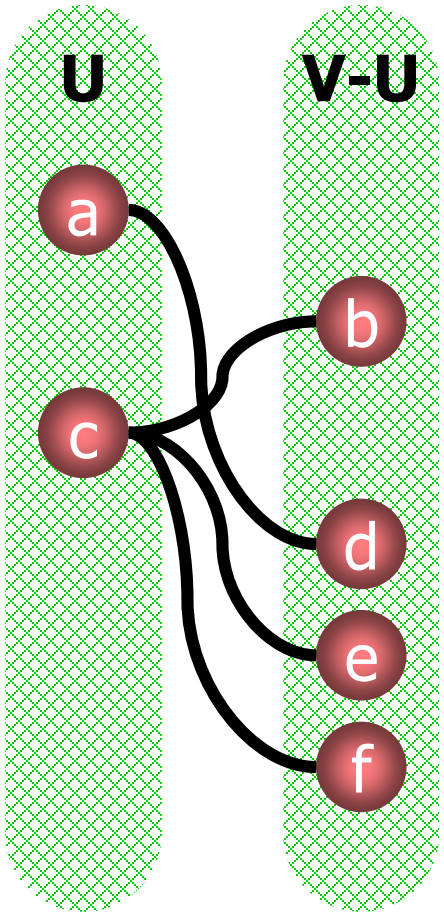
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	2	4

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							113

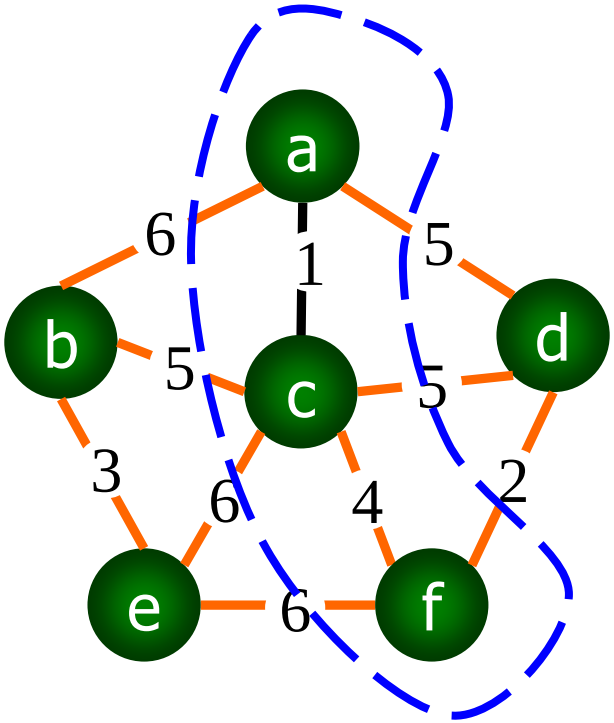
❖ **Prim算法**

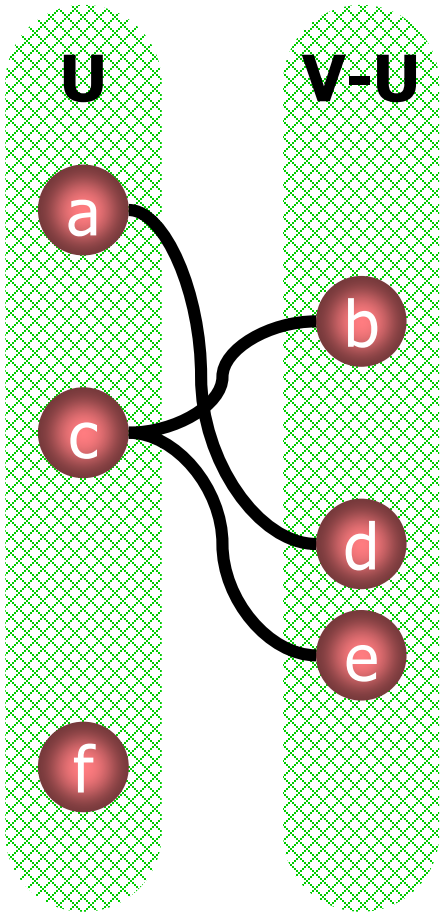
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	2	4

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f

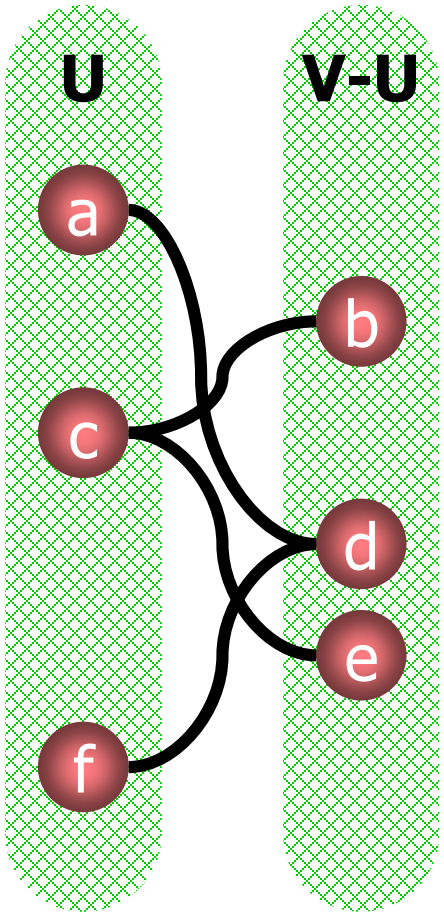




G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							114

❖ **Prim算法**

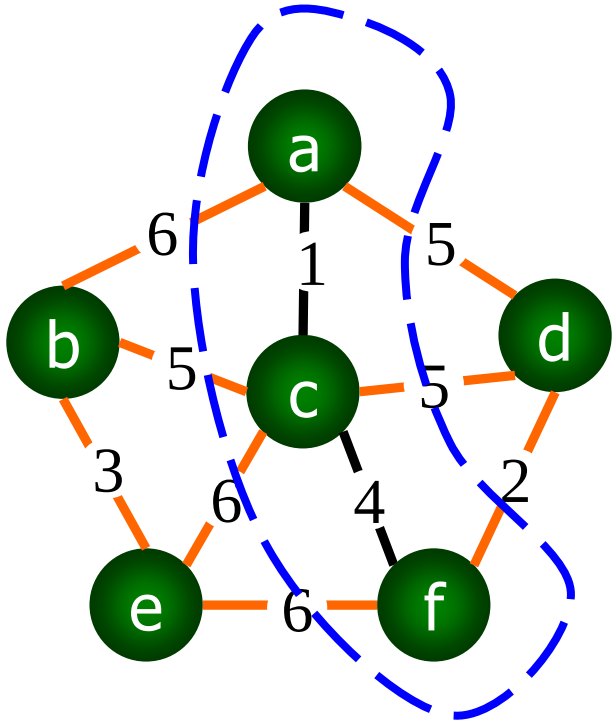


closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	2	0

G.vexs

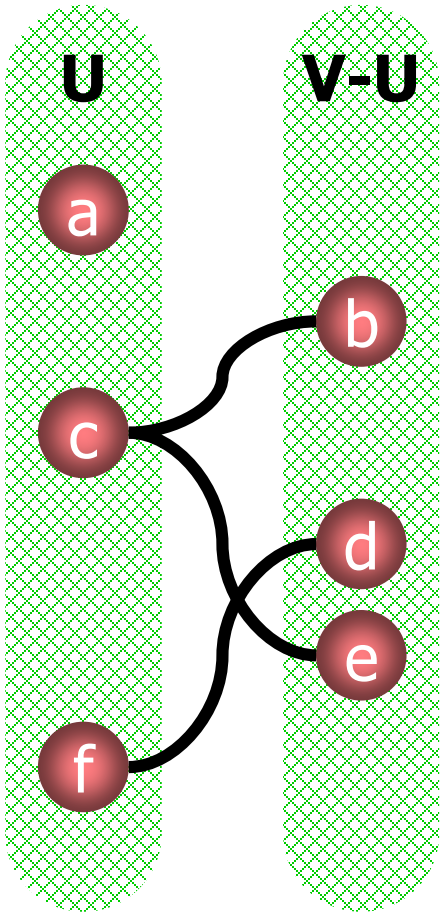
0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							115

❖ **Prim算法**

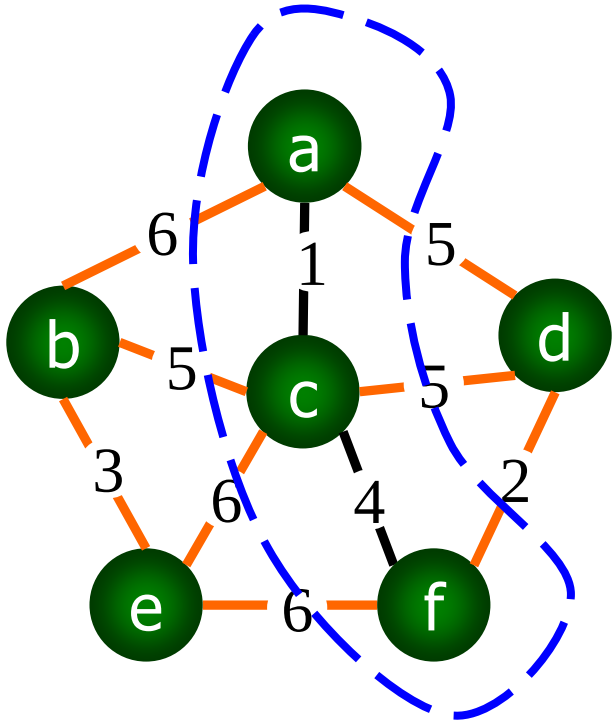


closedge

	adj	cost
0		0
1	2	5
2	0	0
3	0	5
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							116

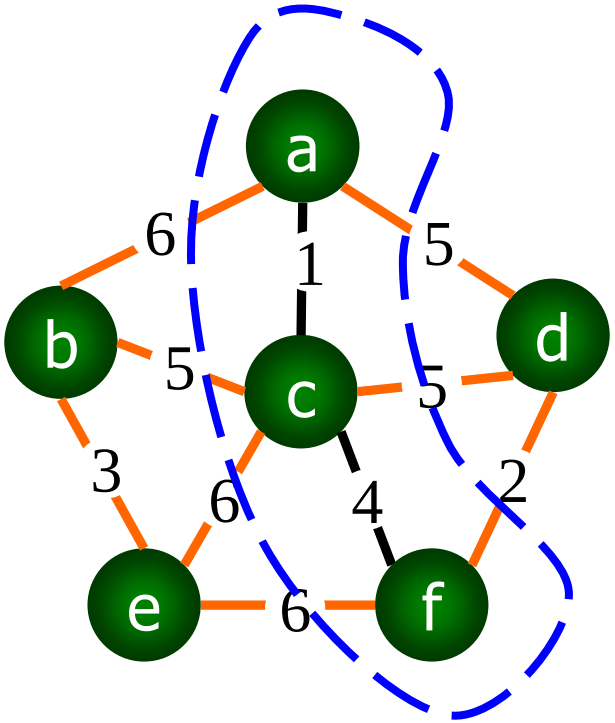
❖ **Prim算法**

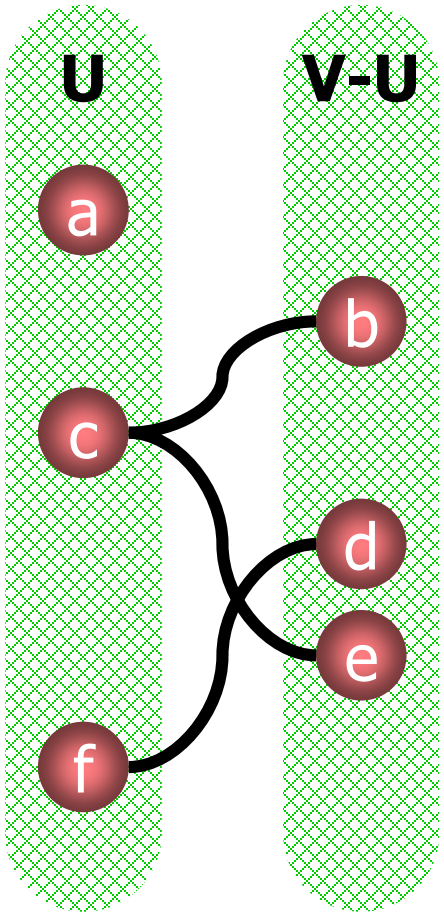
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	**5**	**2**
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							117

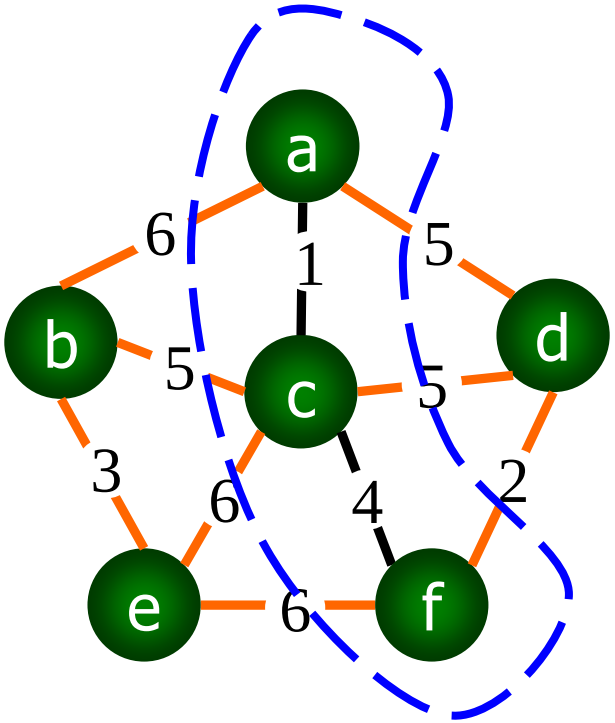
❖ **Prim算法**

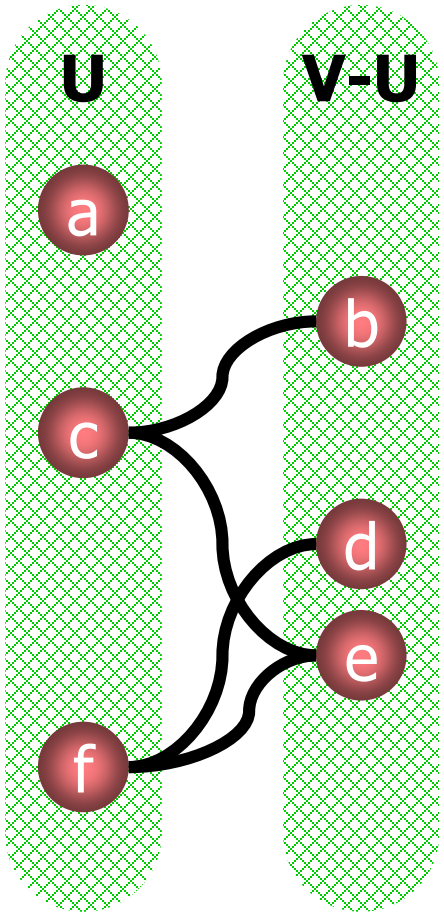
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	2
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							118

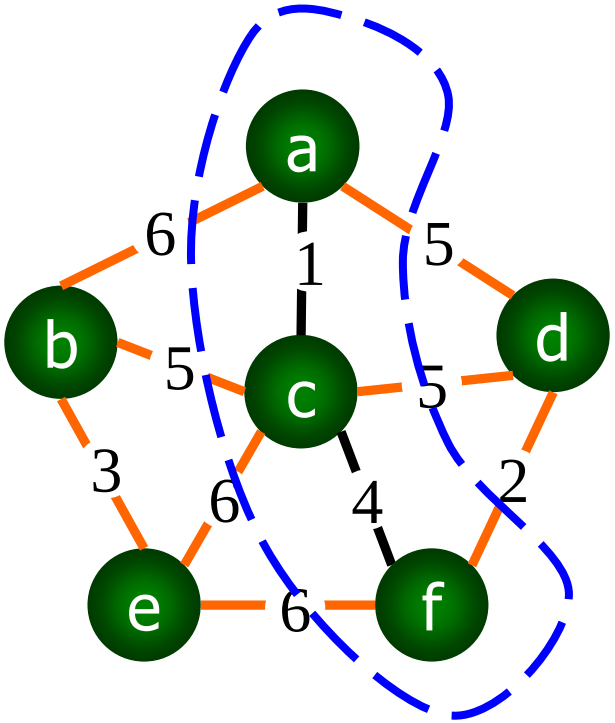
❖ **Prim算法**

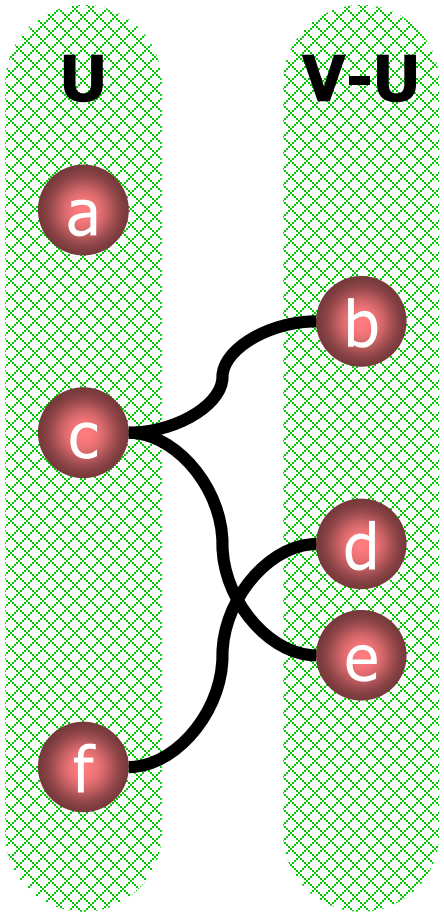
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	2
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							119

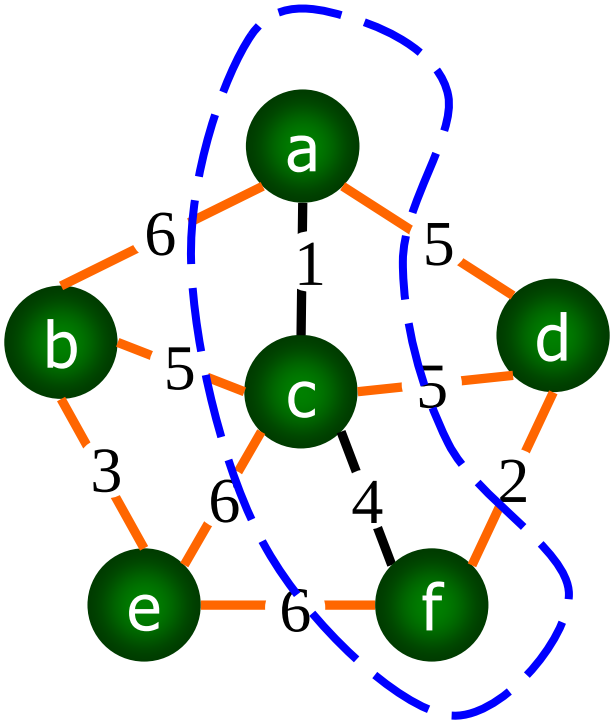
❖ **Prim算法**

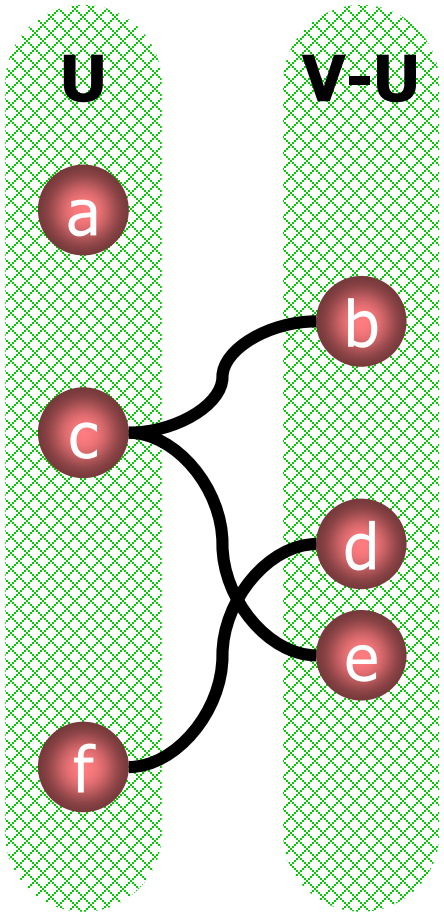
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	2
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							120

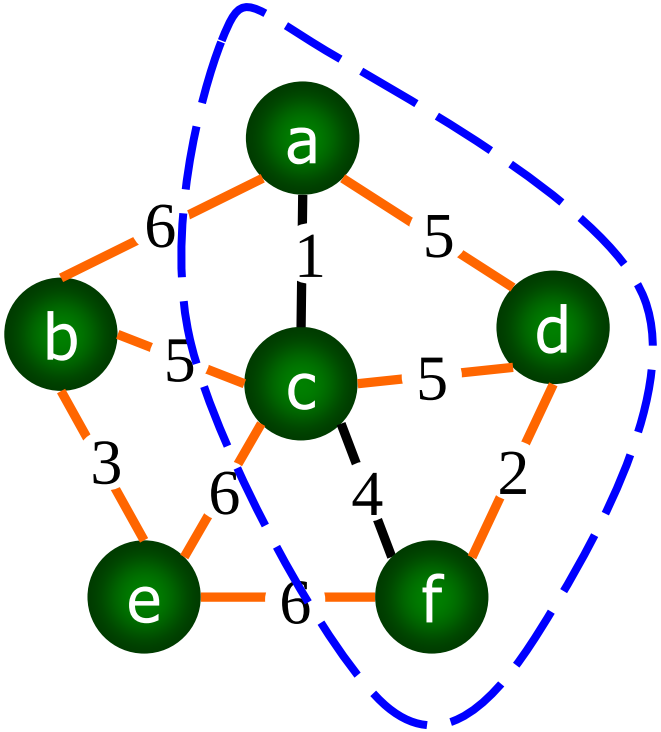
❖ **Prim算法**

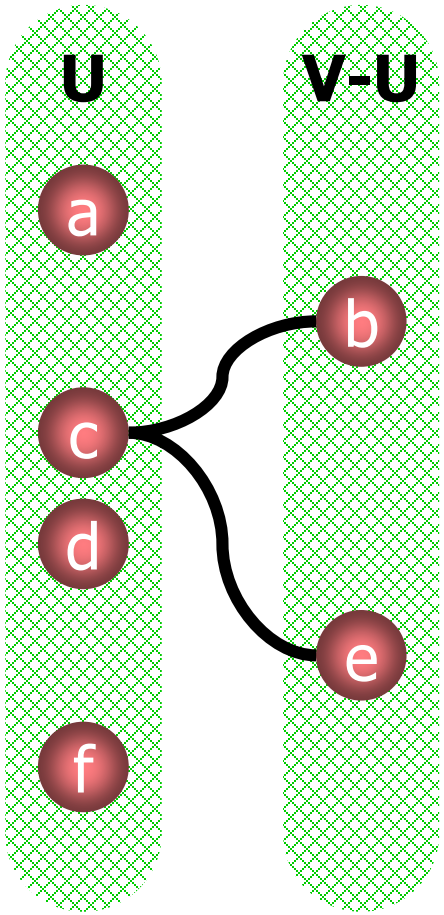
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	2
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f

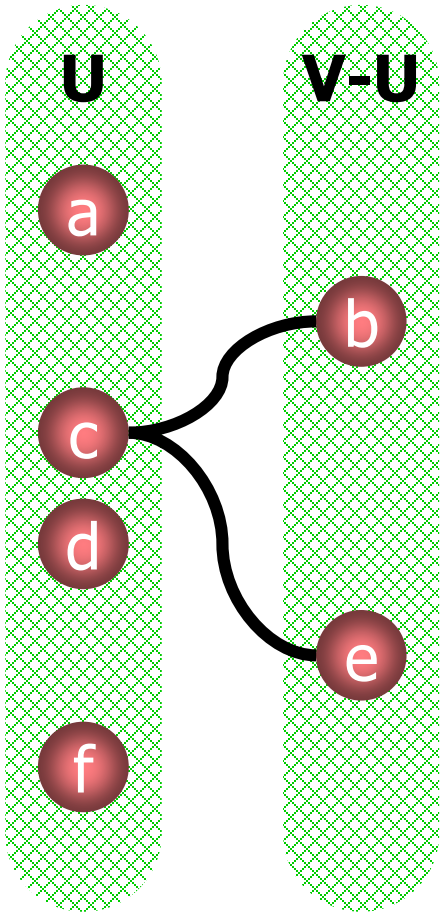




G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							121

❖ **Prim算法**

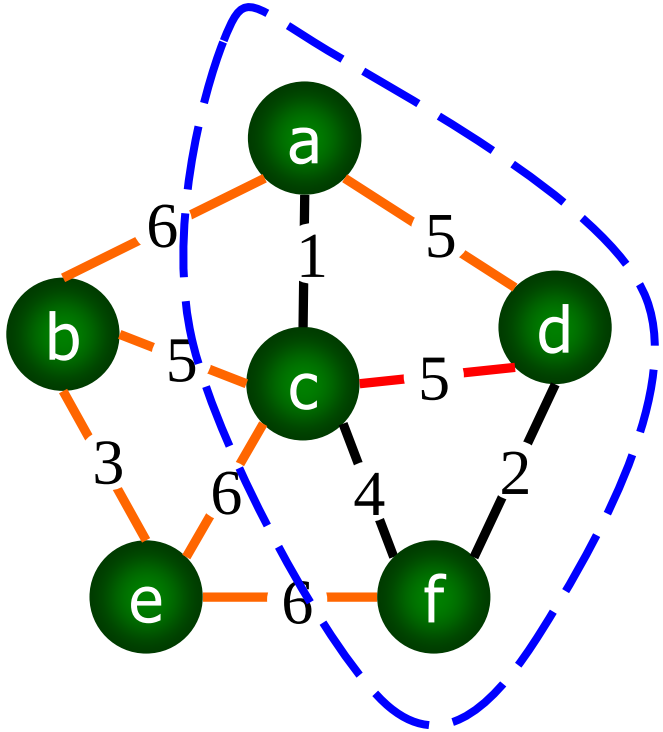


closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	0
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							122

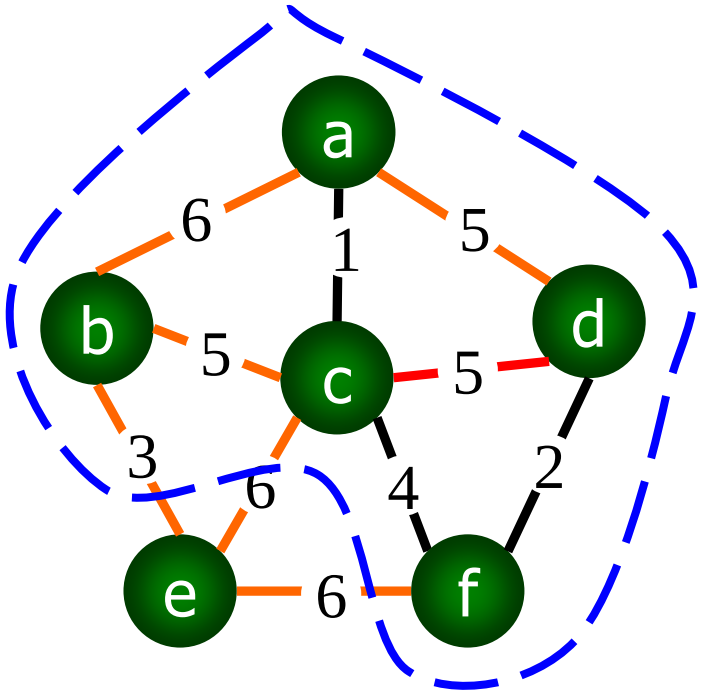
❖ **Prim算法**

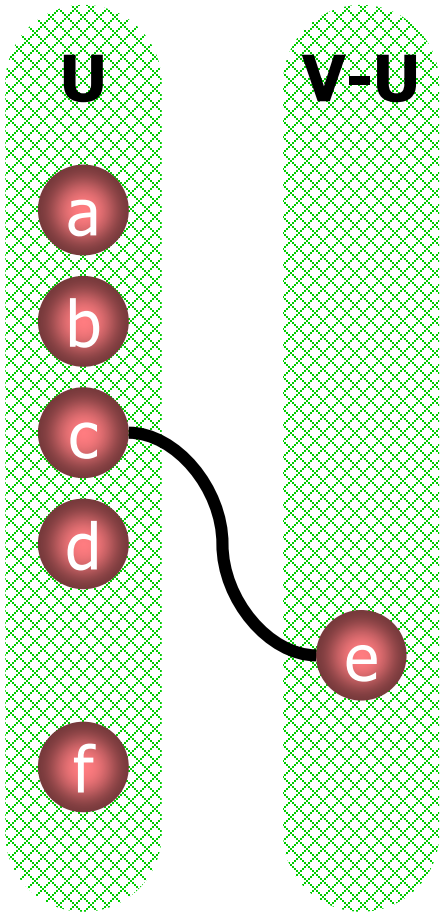
closedge

	adj	cost
0		0
1	2	5
2	0	0
3	5	0
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f

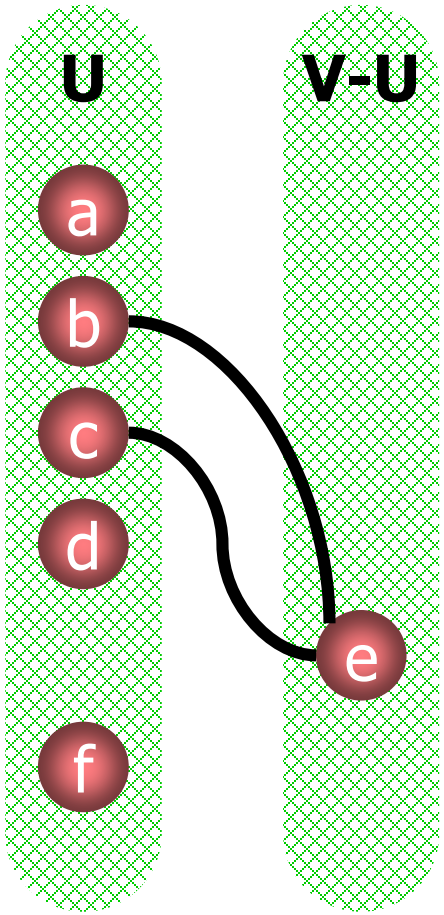




G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							123

❖ **Prim算法**

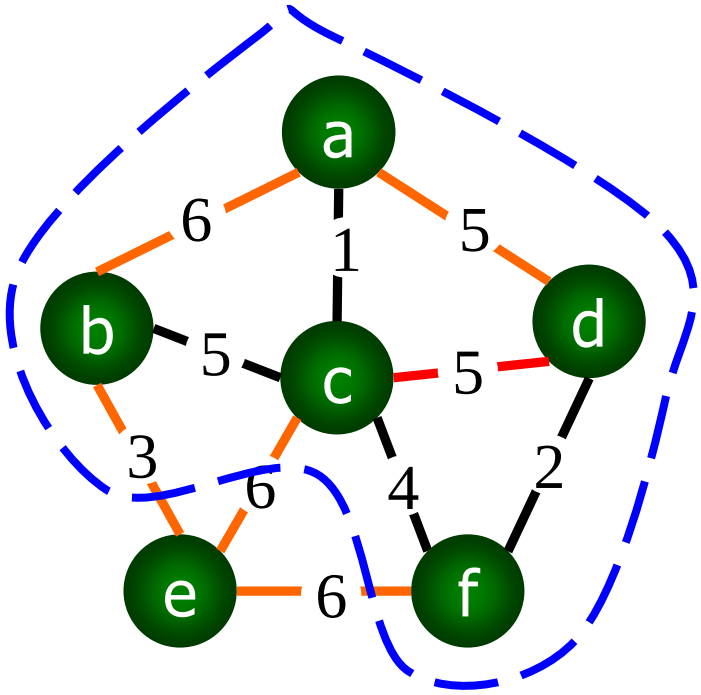


closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f



G.adj

	0	1	2	3	4	5
0	∞	6	1	5	∞	∞
1	6	∞	5	∞	3	∞
2	1	5	∞	5	6	4
3	5	∞	5	∞	∞	2
4	∞	3	6	∞	∞	6
5	∞	∞	4	2	6	∞

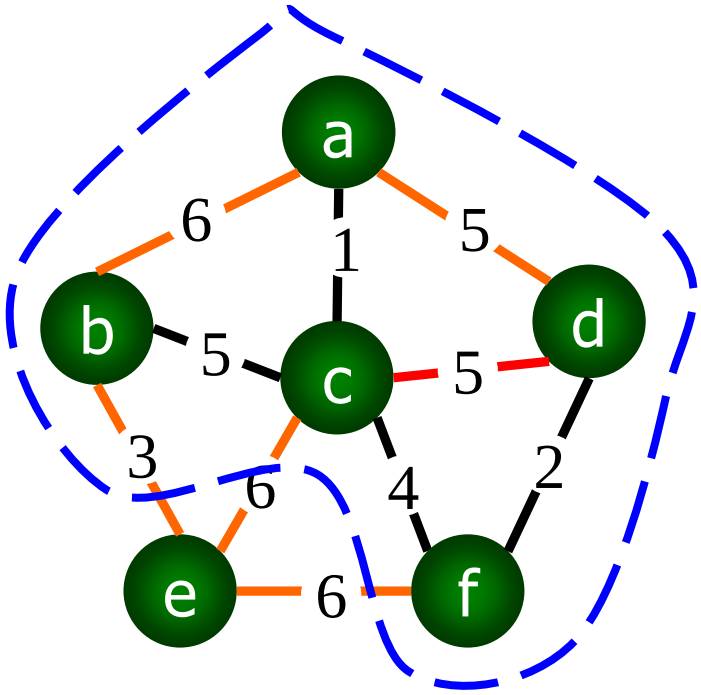
❖ **Prim算法**

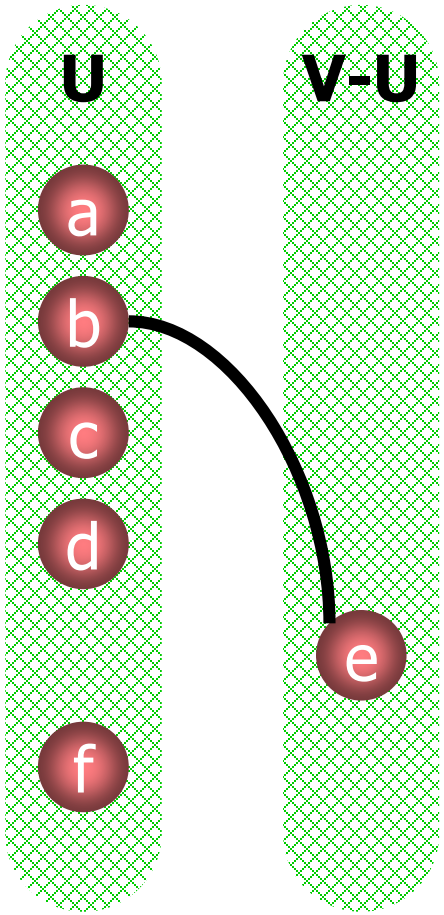
closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	2	6
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							125

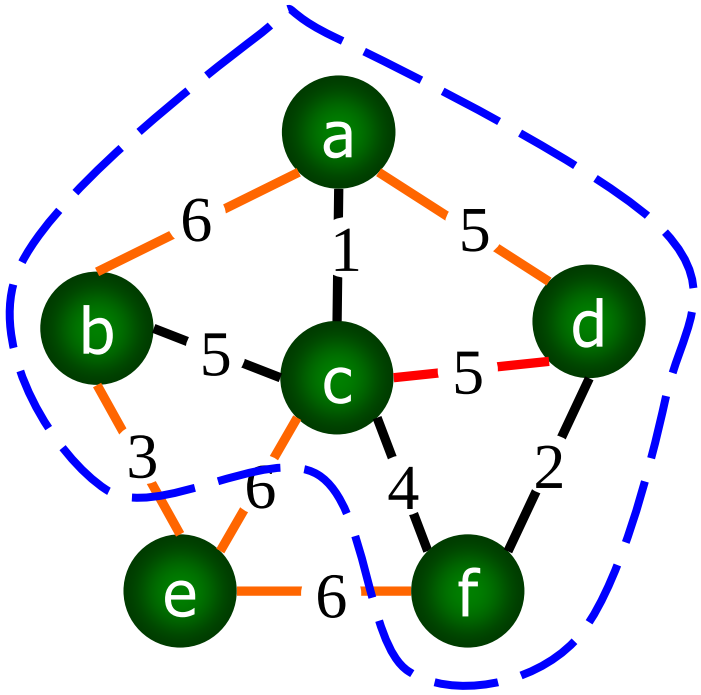
❖ **Prim算法**

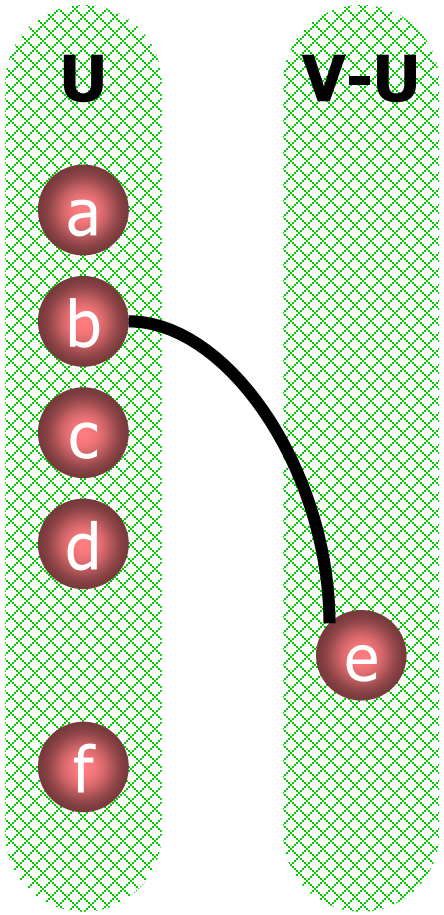
closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	1	3
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							126

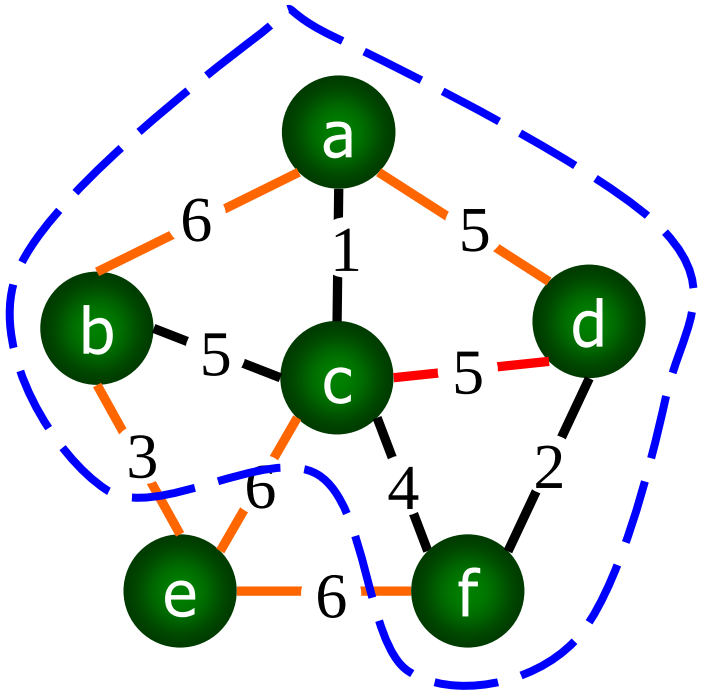
❖ **Prim算法**

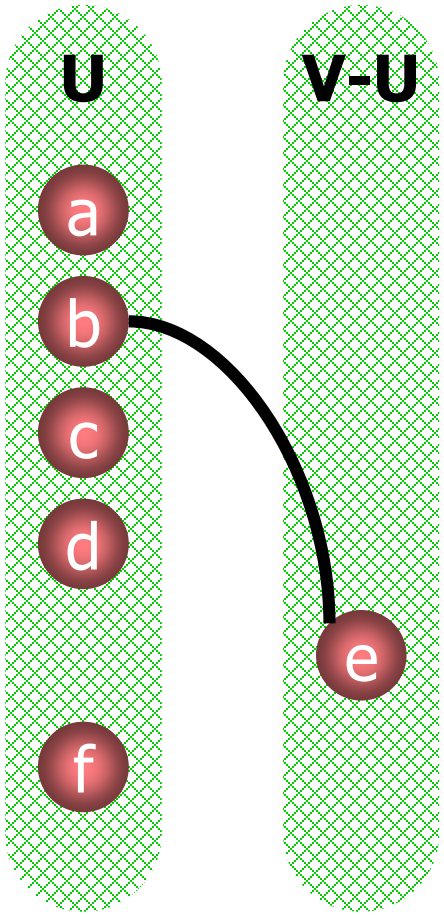
closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	1	3
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							127

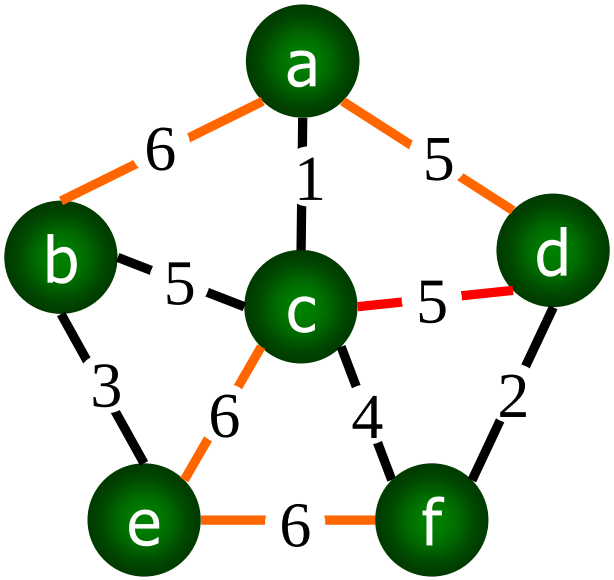
❖ **Prim算法**

closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	1	3
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f



U

a b c d e f

V-U

G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							128

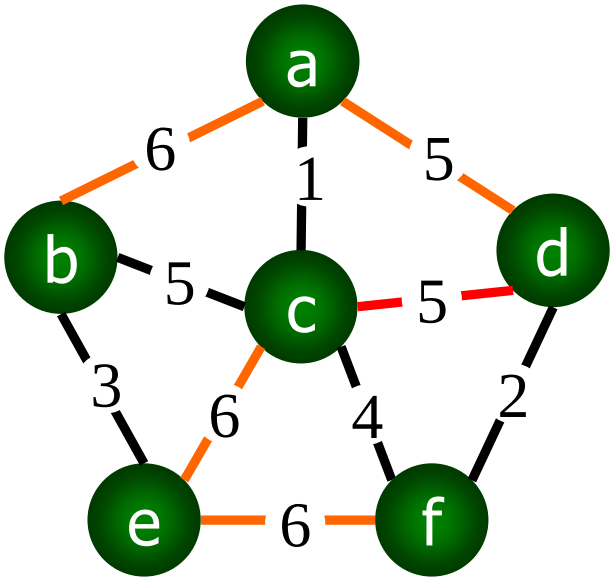
❖ **Prim算法**

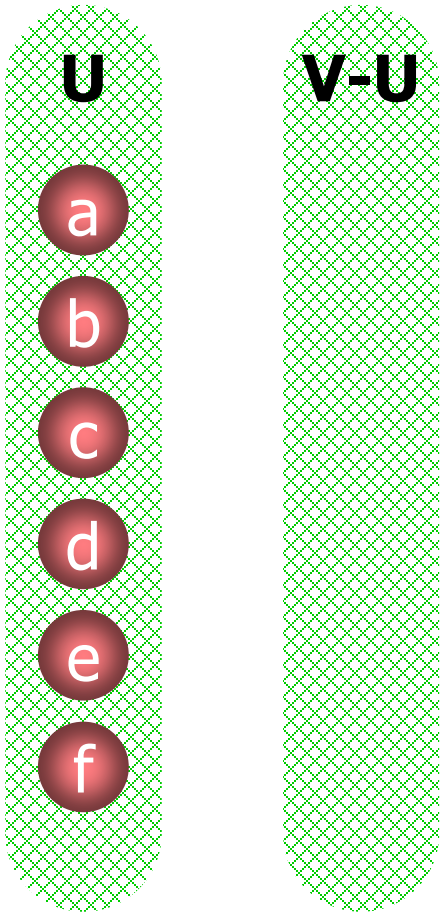
closedge

	adj	cost
0		0
1	2	0
2	0	0
3	5	0
4	1	0
5	2	0

G.vexs

0	a
1	b
2	c
3	d
4	e
5	f





G.adj

	0	1	2	3	4	5	
0	∞	6	1	5	∞	∞	
1	6	∞	5	∞	3	∞	
2	1	5	∞	5	6	4	
3	5	∞	5	∞	∞	2	
4	∞	3	6	∞	∞	6	
5	∞	∞	4	2	6	∞	
							129

Prim算法伪代码(假设存储结构为邻接矩阵)

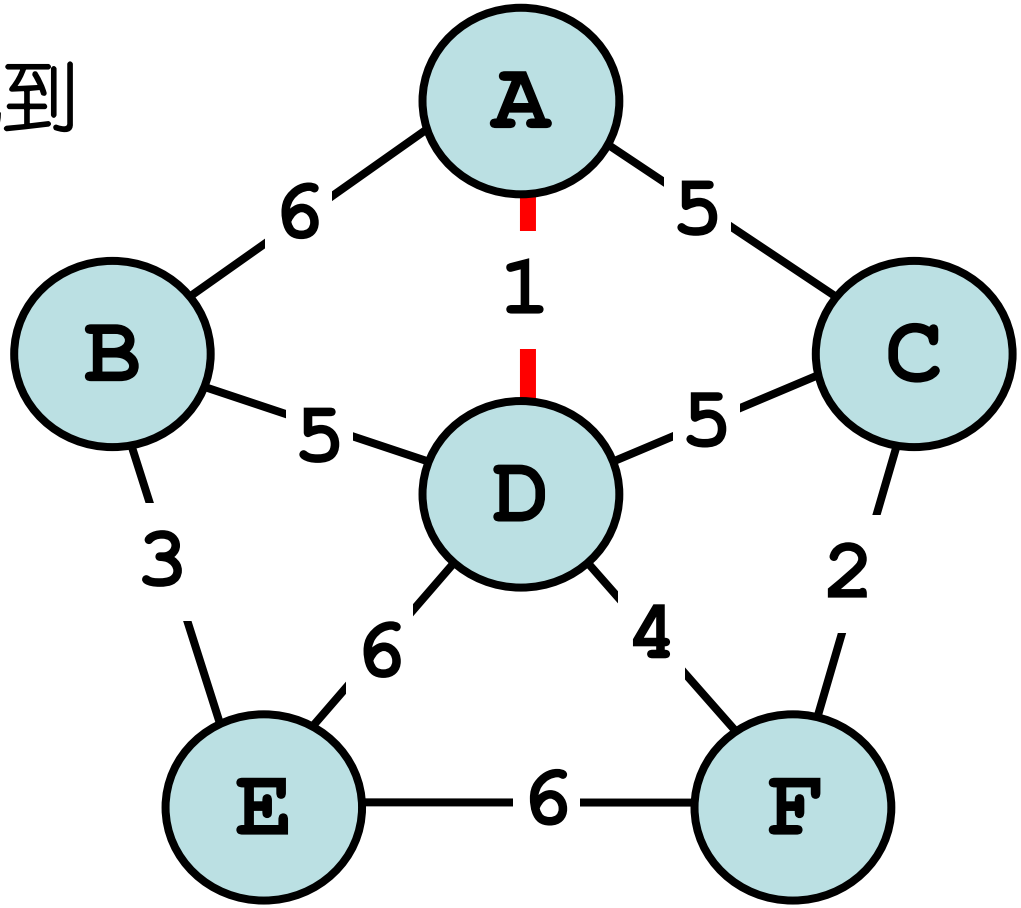
```
MiniCost_Prem(G,u){
  k = LocateVex ( G, u ); // 顶点u为构造生成树的起始点
  for ( j=0; j<G.vexnum; ++j ) // 辅助数组初始化
     if (j!=k) closedge[j] = { u, G.arcs[k][j].adj };
  closedge[k].lowcost = 0; // 初始，U = {u}
  for (i=0; i<G.vexnum; ++i) { //在其余顶点中选择
     k = minimum(closedge); // 求出T的下一个结点(k)
     printf(closedge[k].adjvex, G.vexs[k]); // 输出生成树的边
     closedge[k].lowcost = 0; // 第k顶点并入U集
     for (j=0; j<G.vexnum; ++j)
        if (G.arcs[k][j].adj < closedge[j].lowcost)
              closedge[j] = { G.vexs[k], G.arcs[k][j].adj };
}
```

时间复杂度：$O(n^2)$

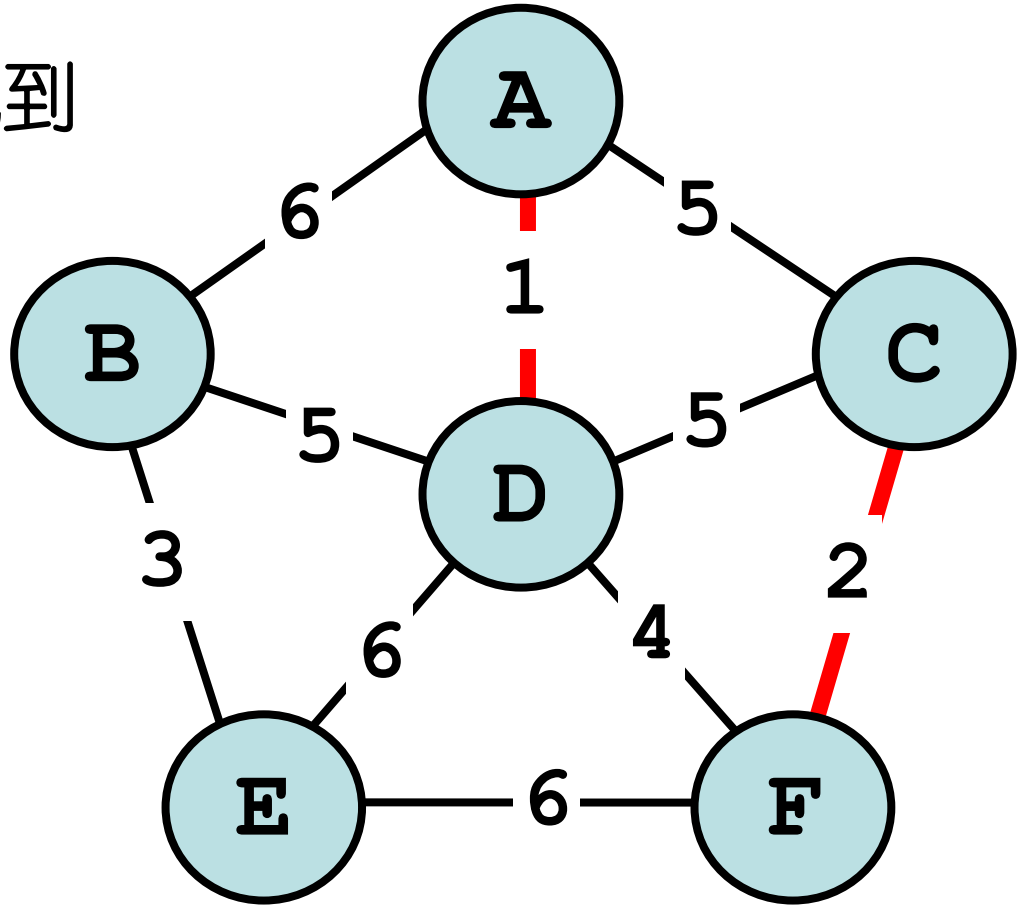
最小生成树

- **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
  - ...

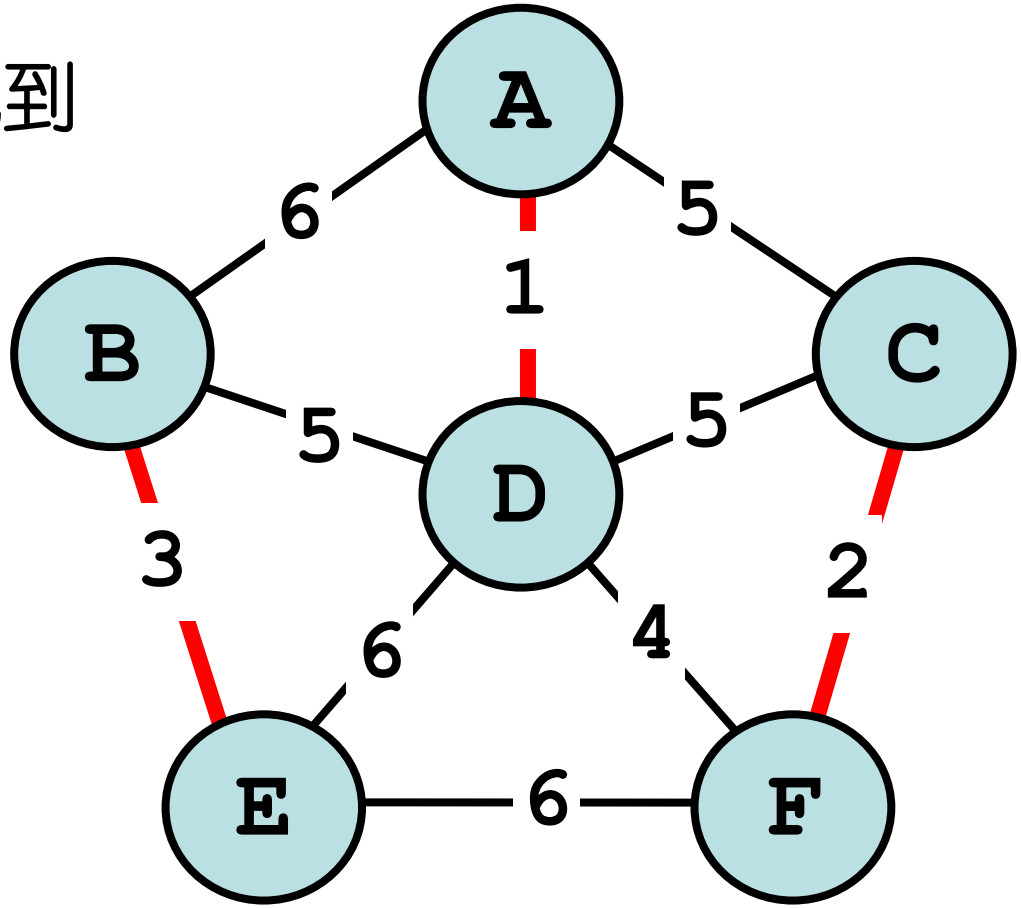
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - 但不能产生回路
  - . . .



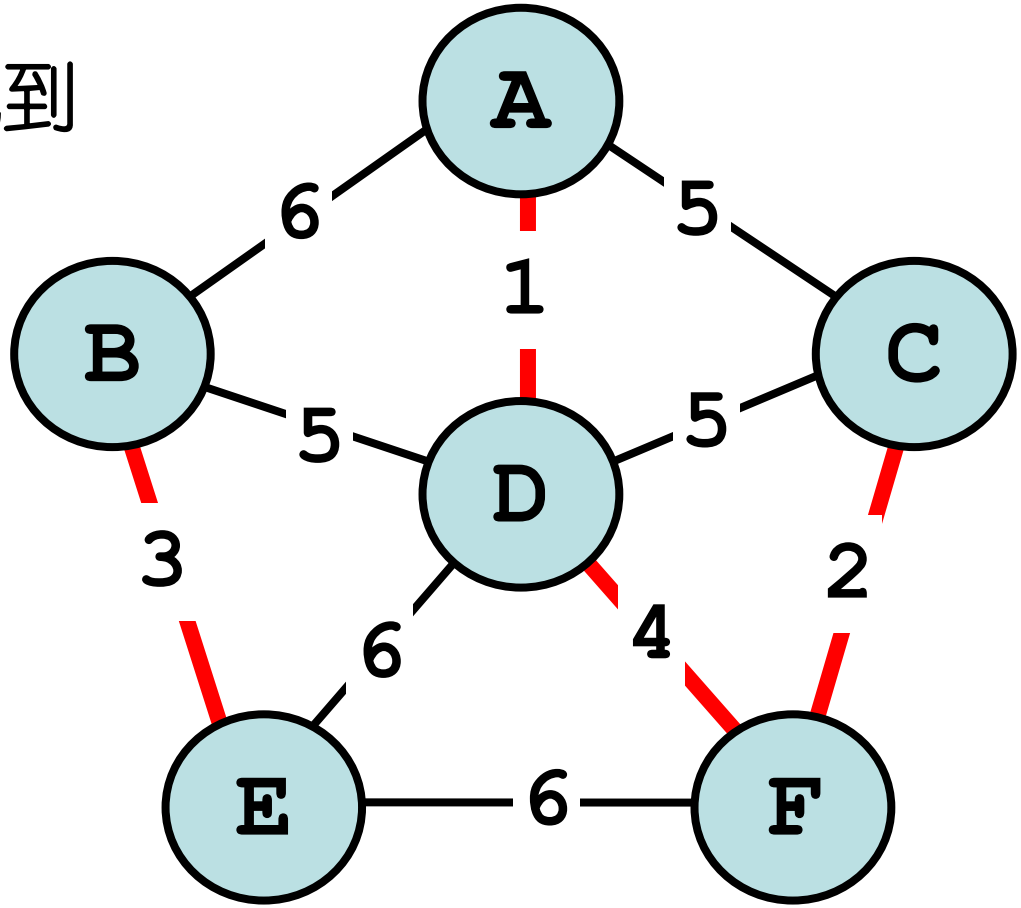
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - 但不能产生回路
  - . . .



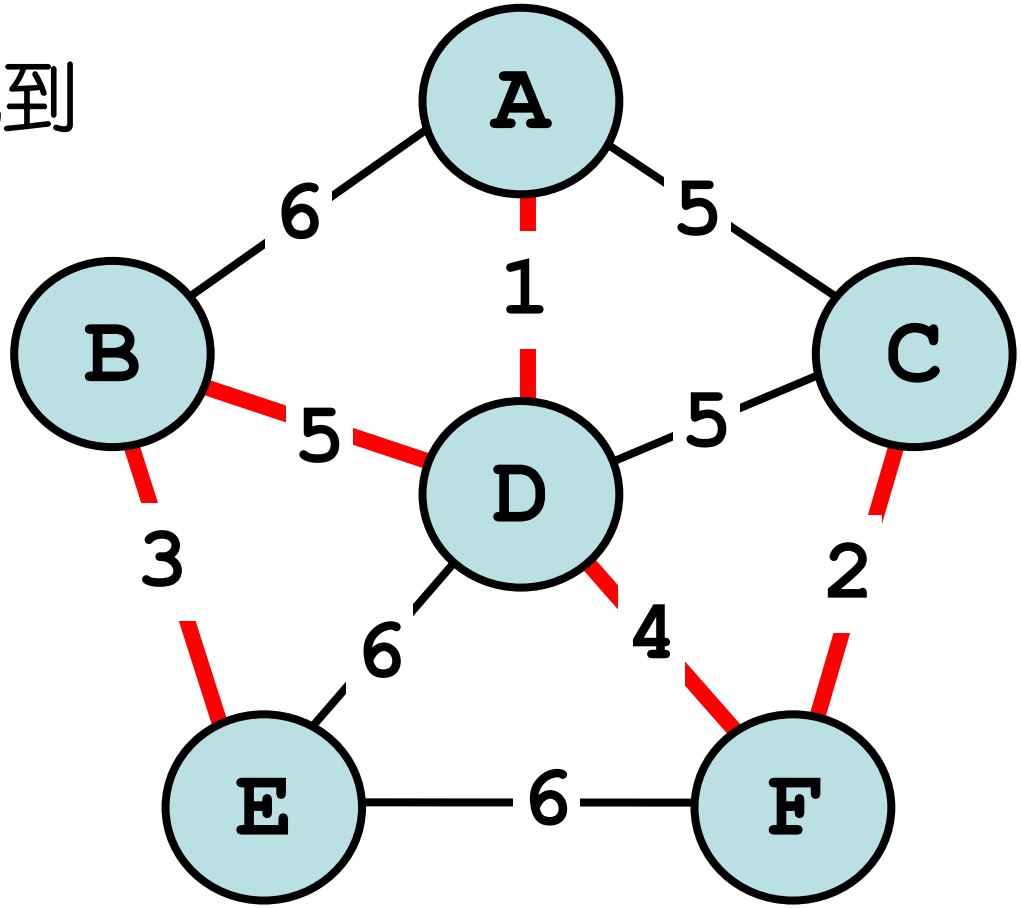
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - 但不能产生回路
  - . . .



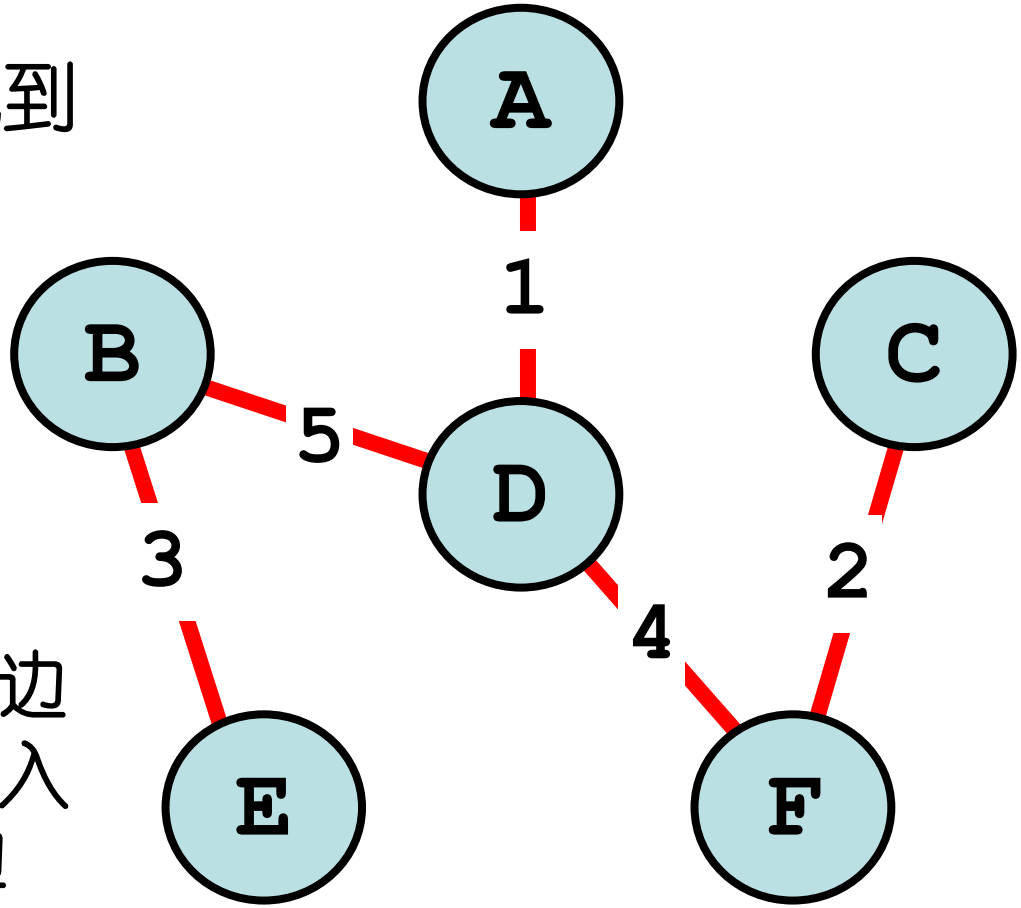
- **Kruskal算法**
  - 找到图中权最小的一条边，加入生成树
  - 再在剩下的边中找到权最小的，加入
  - 但不能产生回路
  - . . .



- **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



时间复杂度在于找最小边判断是否可以加入并加入到生成树中，设有e条边

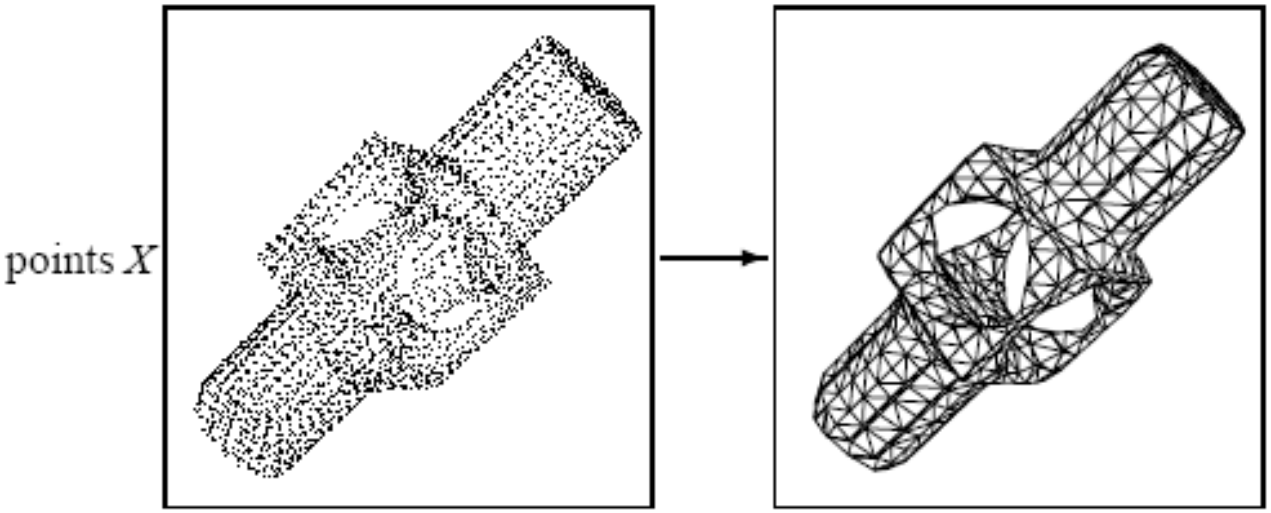
O(e loge)

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，对于无向图，凡是能遍历到的就是一个连通分量，对于有向图呢？
  - 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**

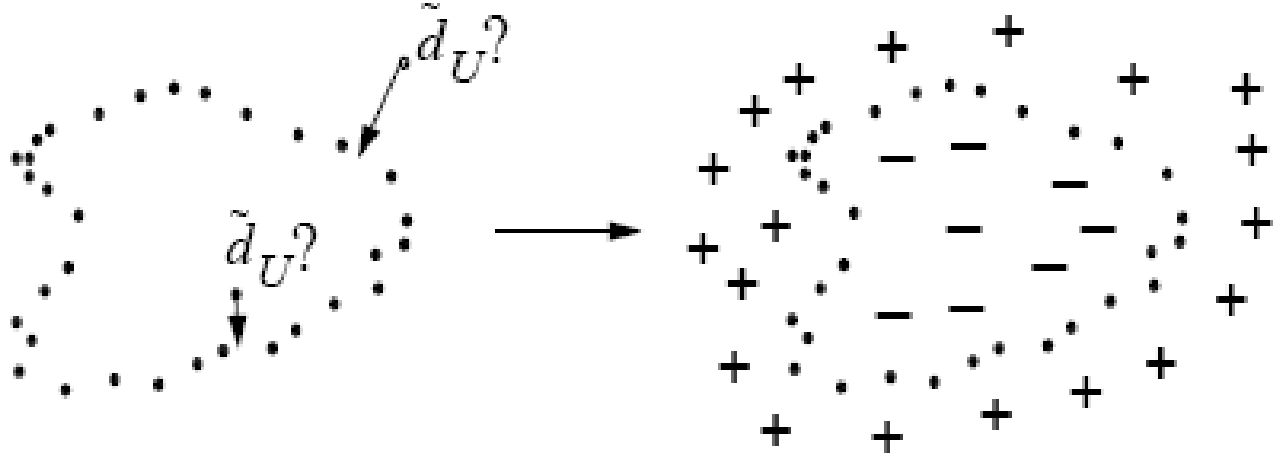
图形学中的应用

- Hooppe曲面重建算法(Surface reconstruction from unorganized points.H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, W. Stuetzle. *ACM SIGGRAPH 1992*, 71-78.)

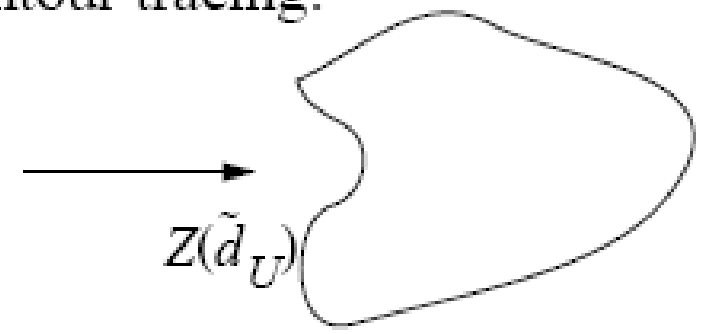


图形学中的应用

1. Estimate d_U from data points:



2. Use contour tracing:

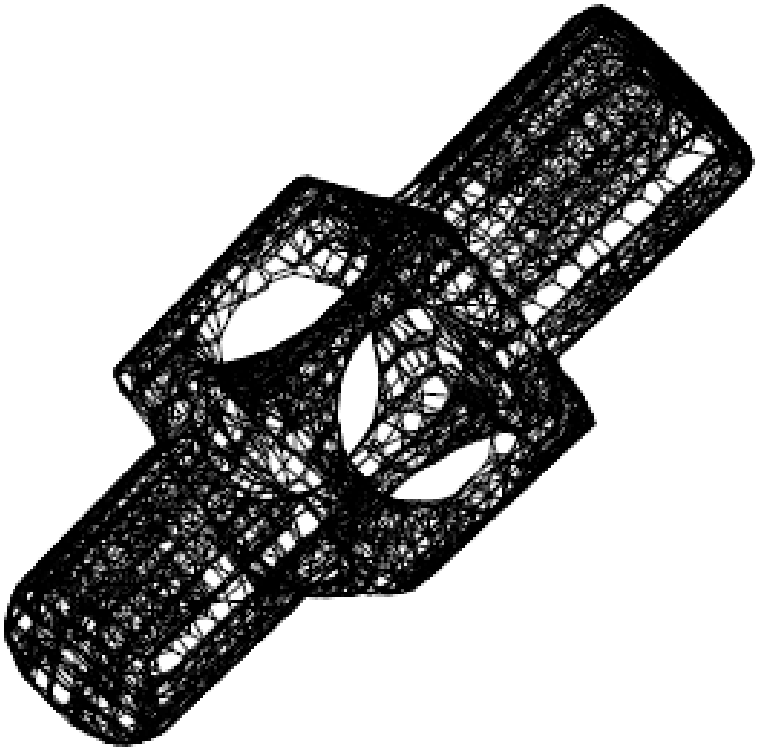


图形学中的应用

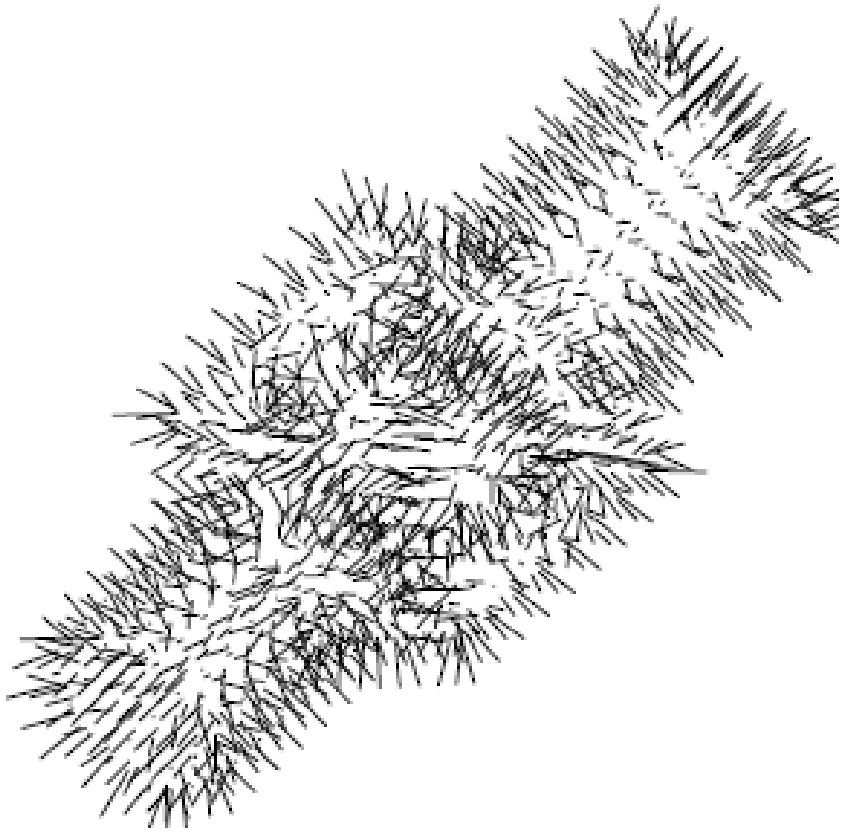
- 步骤：

1) 构造Riemannian图
2) 计算各点附近的切平面
3) 切平面法矢量的一致化(基于最小生成树的遍历定向)
4) Marching Cube等值面抽取网格生成

图形学中的应用



Riemannian图



一致的法矢量

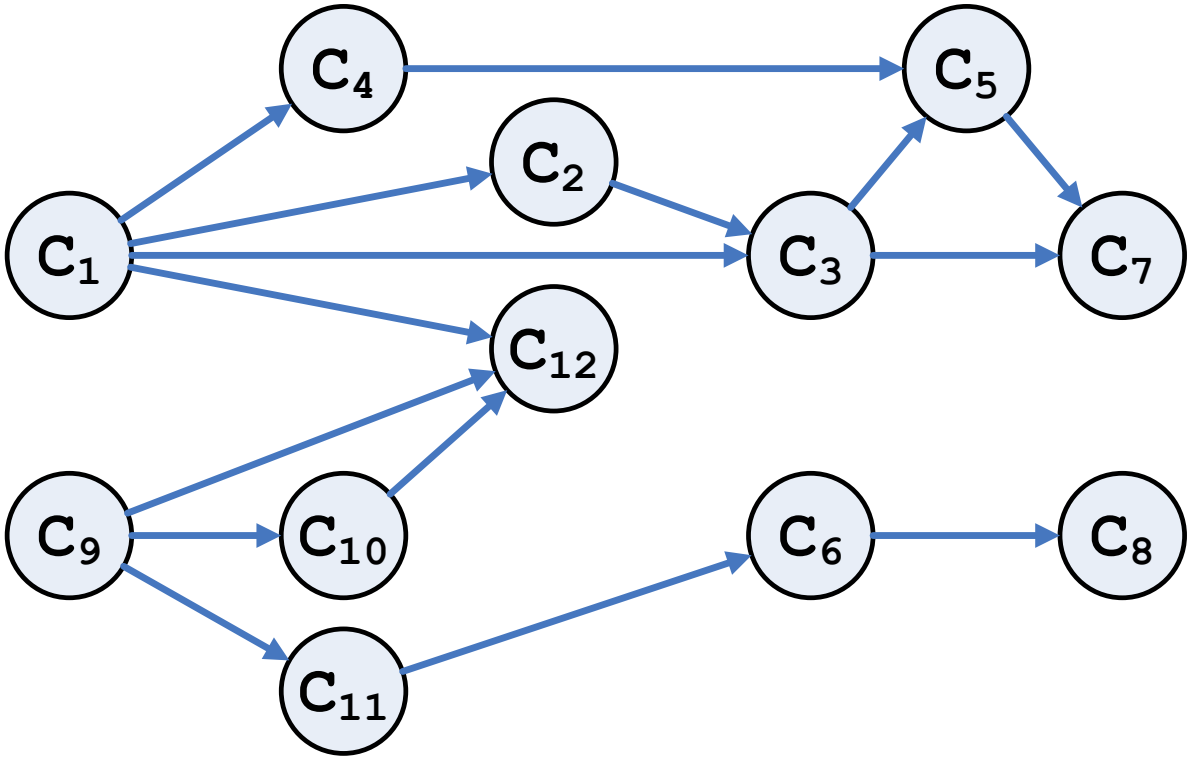
有向无环图的应用：拓扑排序

- 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

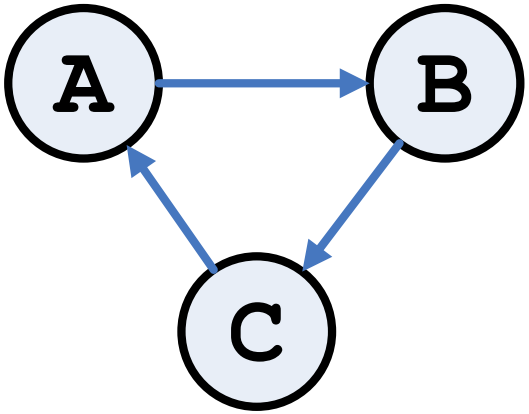
有向无环图的应用：拓扑排序

- 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**



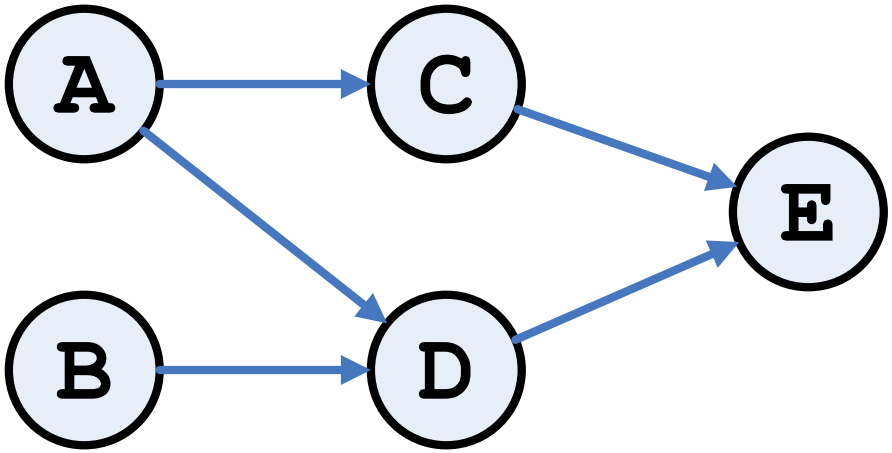
有向无环图的应用：拓扑排序

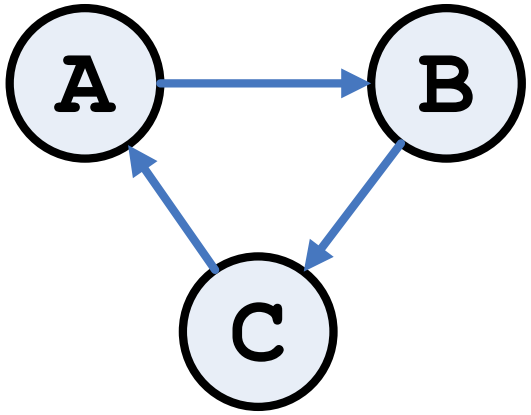
- 问题：（拓扑排序）
  - 假设学生每学期只上一门课，应该如何安排课程？
  - A到B有一条有向路径，说B依赖于A.
  - 拓扑排序即活动排成一个序列，被依赖活动一定位于依赖的活动前面.---不矛盾！



有向无环图的应用：拓扑排序

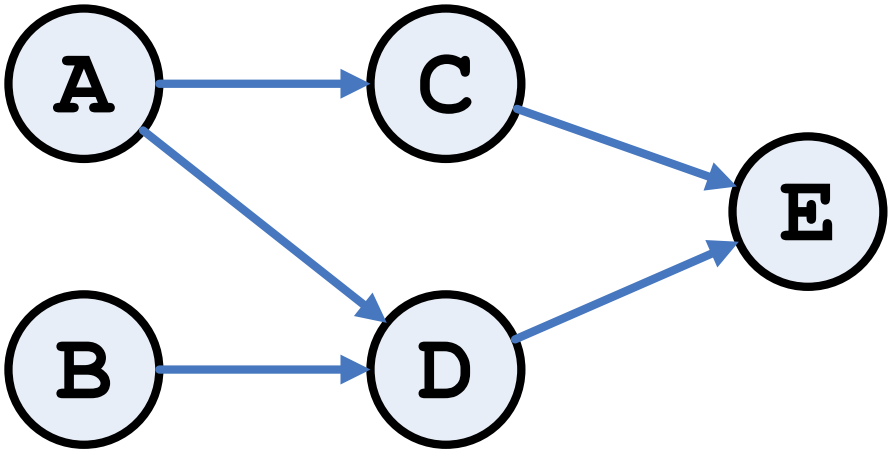
- 条件：　　此图必须是有向无环图

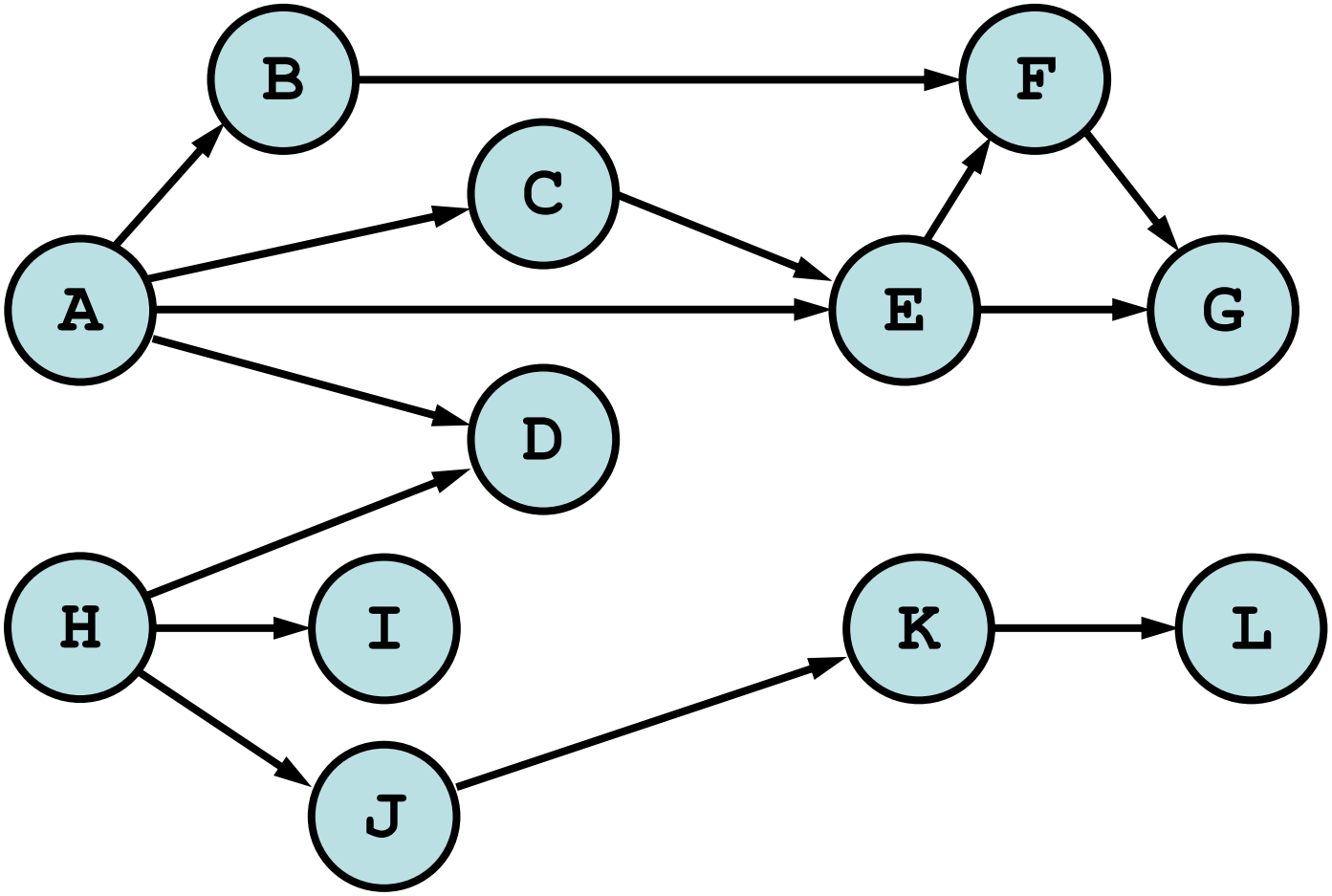




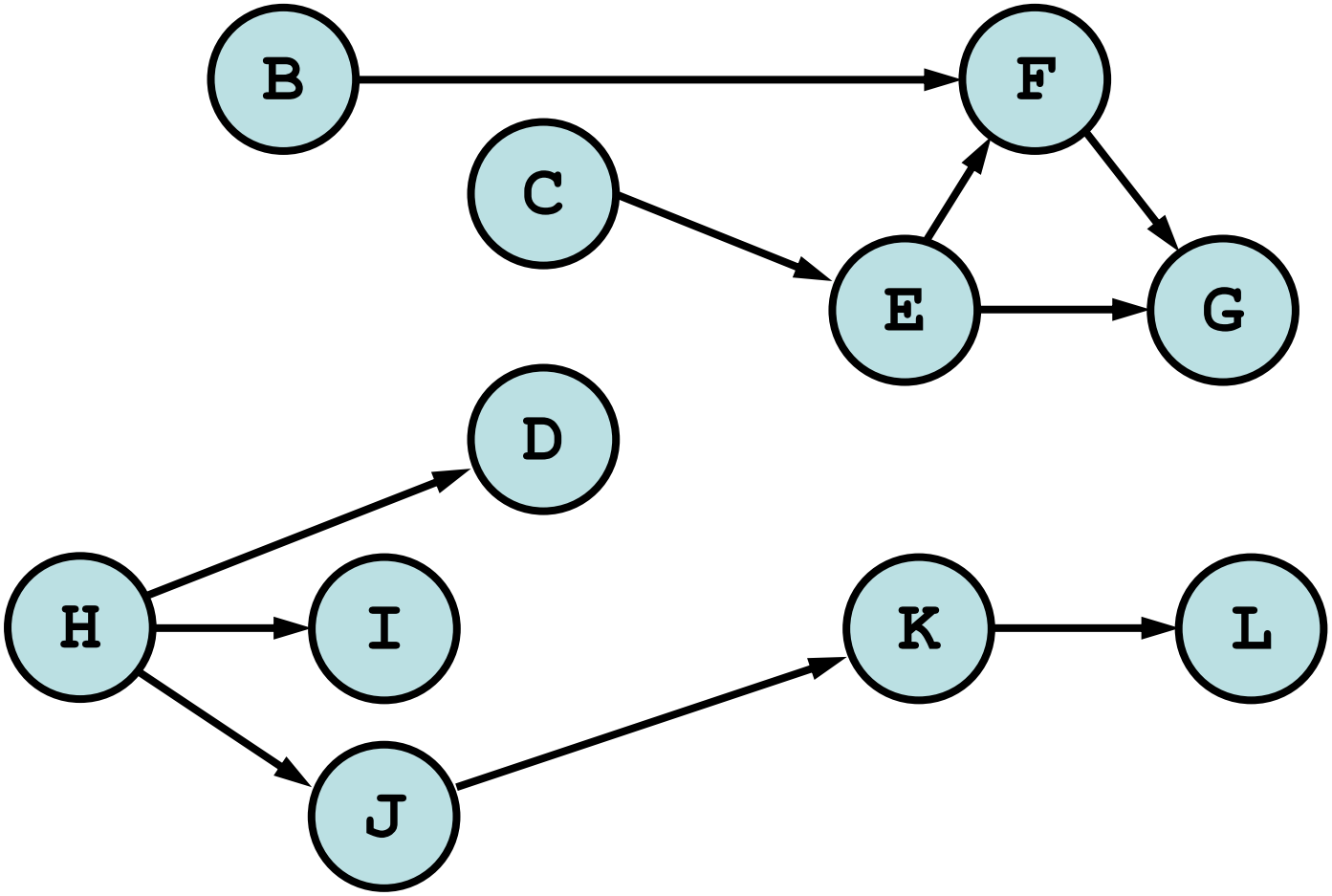
有向无环图的应用：拓扑排序

- 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

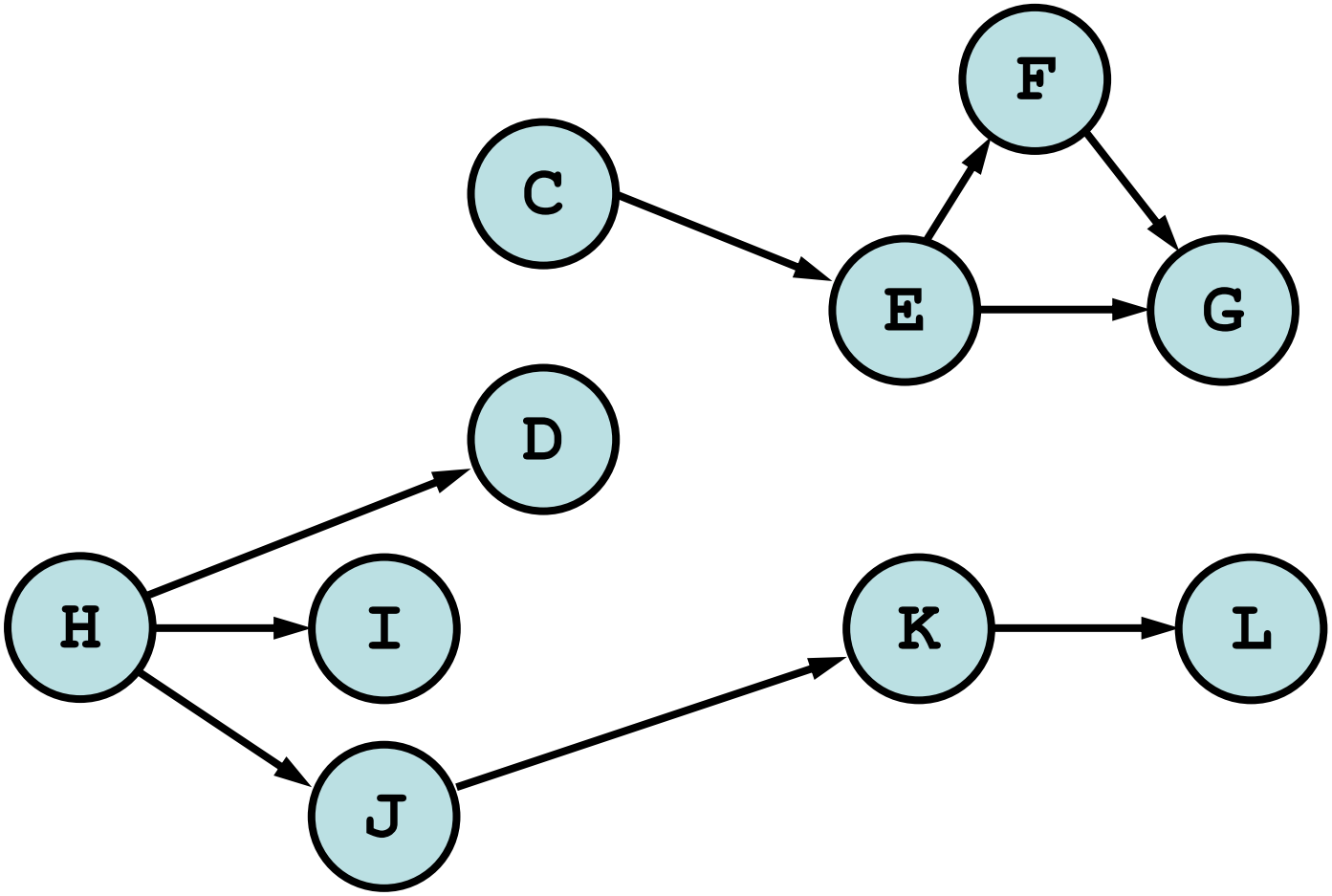




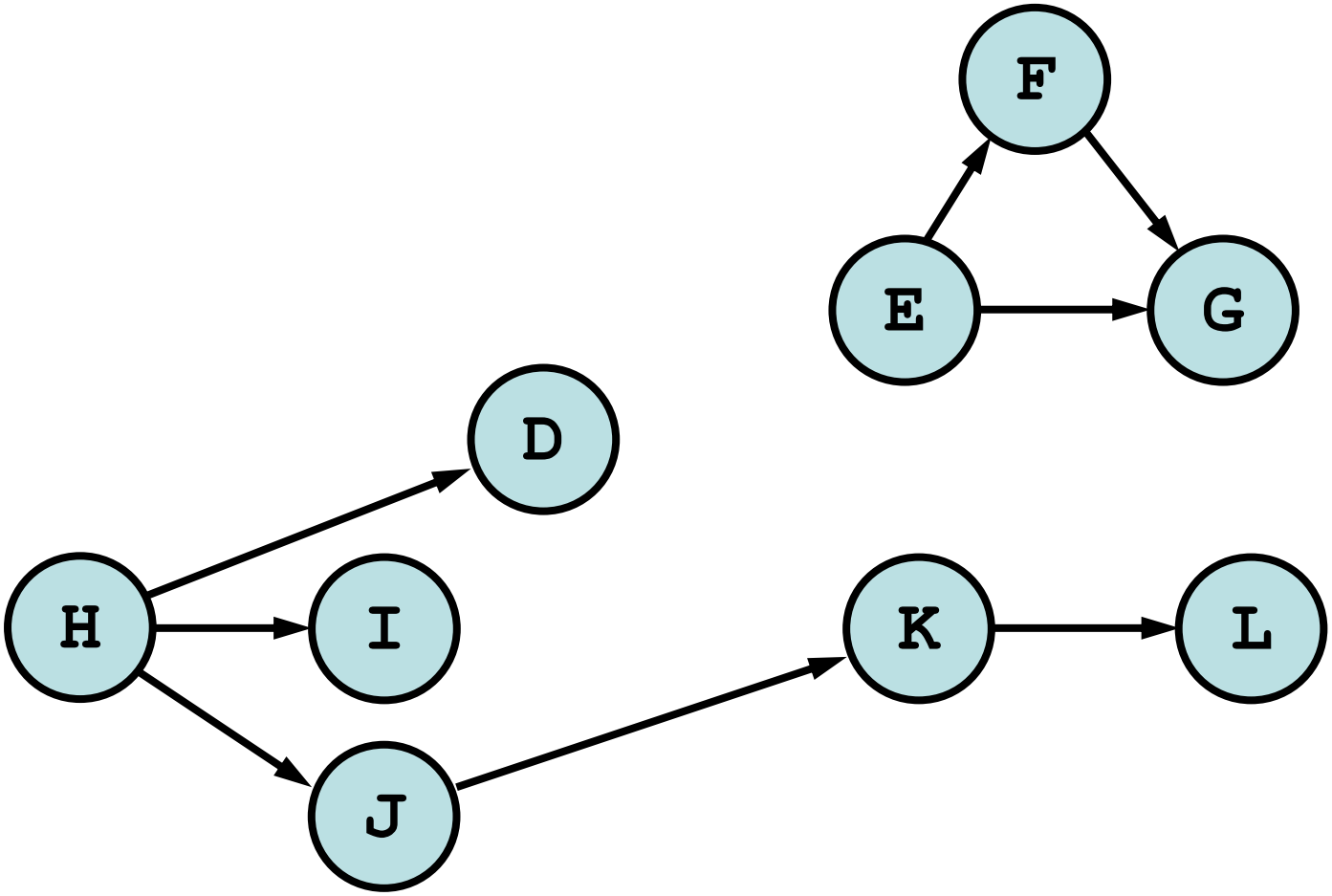
- 无前驱的顶点有：A，H
- 输出：A
- 删除A和以A为弧尾的弧



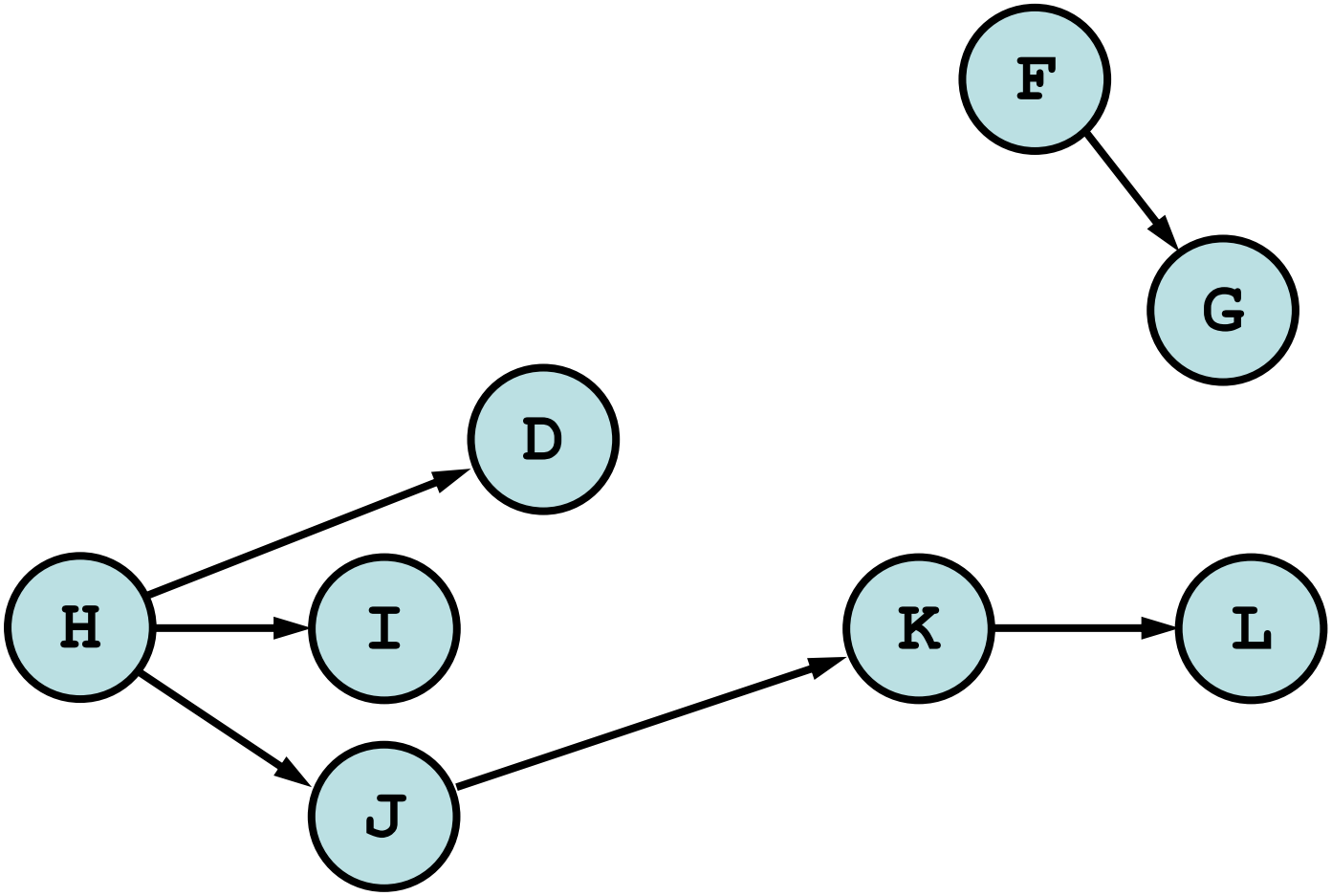
- 无前驱的顶点有：**B，H**
- 输出：**A,B**
- 删除**B**和以**B**为弧尾的弧



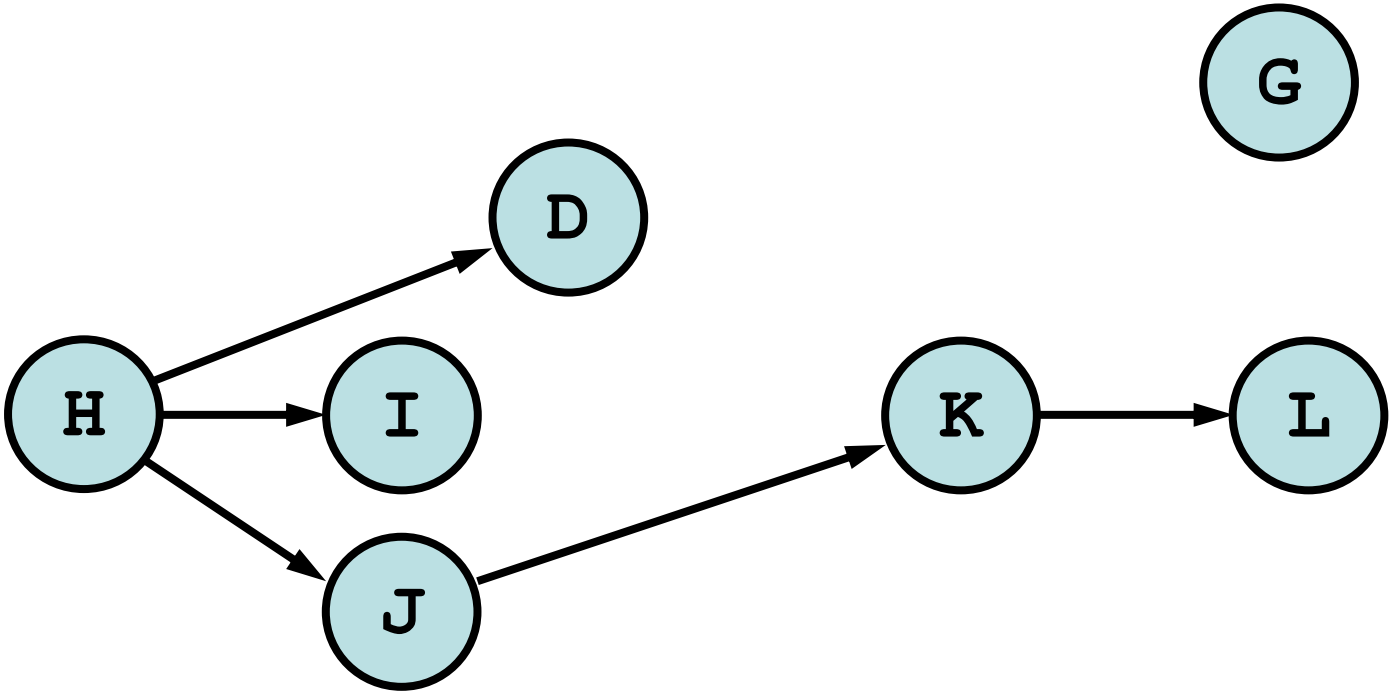
- 无前驱的顶点有：C，H
- 输出：A,B,C
- 删除C和以C为弧尾的弧



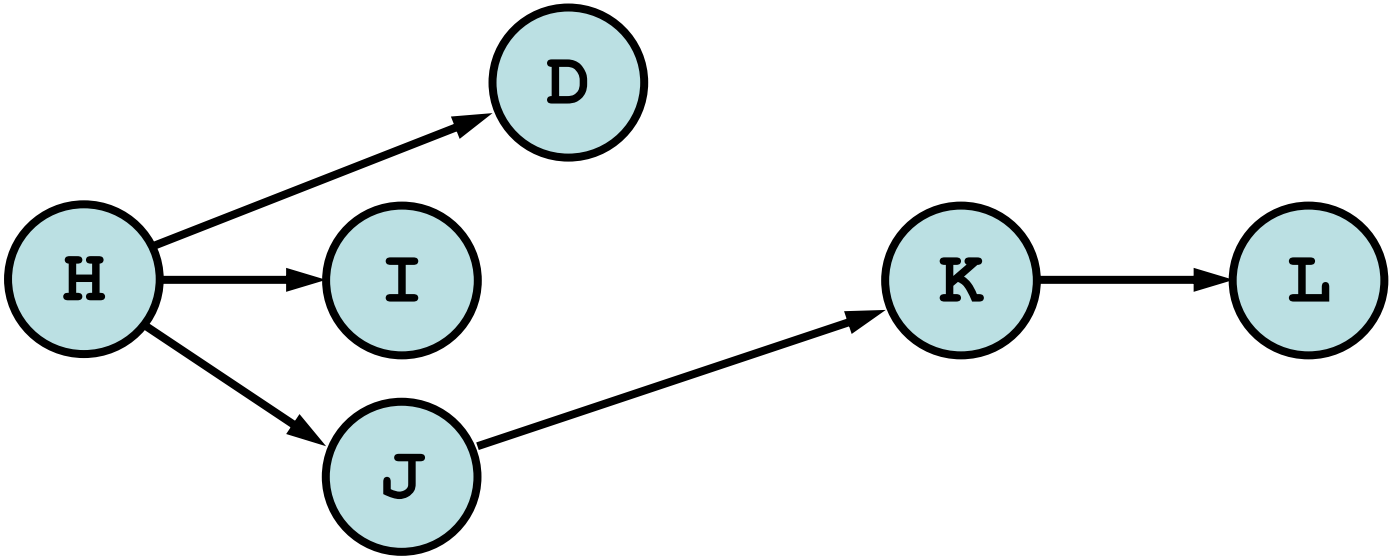
- 无前驱的顶点有：**E，H**
- 输出：**A,B,C,E**
- 删除**E**和以**E**为弧尾的弧



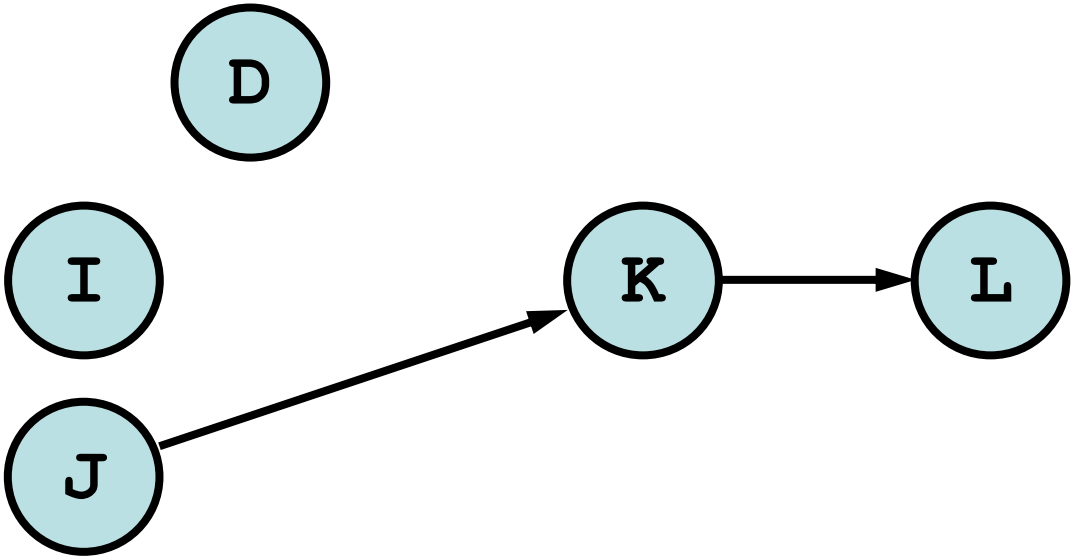
- 无前驱的顶点有：F，H
- 输出：A,B,C,E,F
- 删除F和以F为弧尾的弧



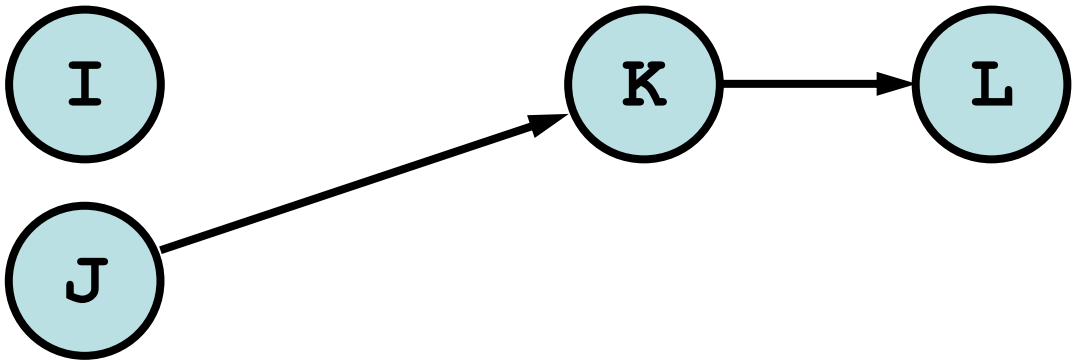
- 无前驱的顶点有：G,H
- 输出：A,B,C,E,F,G
- 删除G和以G为弧尾的弧



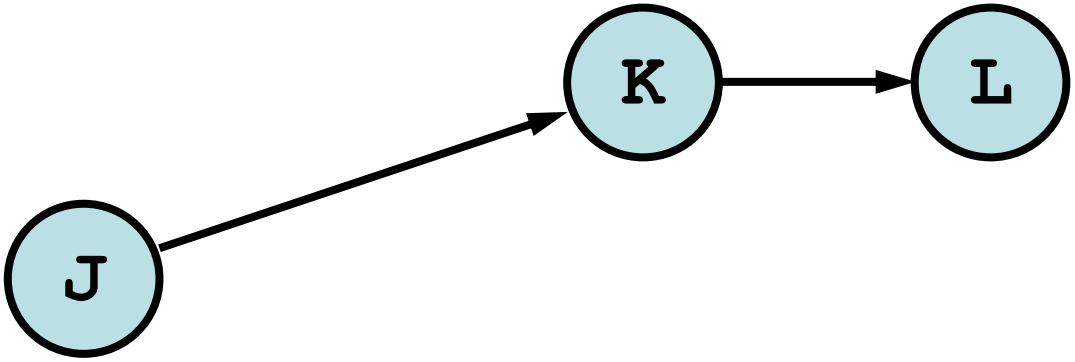
- 无前驱的顶点有：H
- 输出：A,B,C,E,F,G,H
- 删除H和以H为弧尾的弧



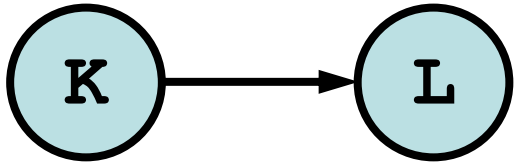
- 无前驱的顶点有：D, I, J
- 输出：A,B,C,E,F,G,H,D
- 删除D和以D为弧尾的弧



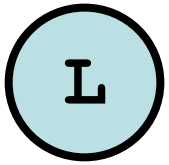
- 无前驱的顶点有：I，J
- 输出：A,B,C,E,F,G,H,D,I
- 删除I和以I为弧尾的弧



- 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



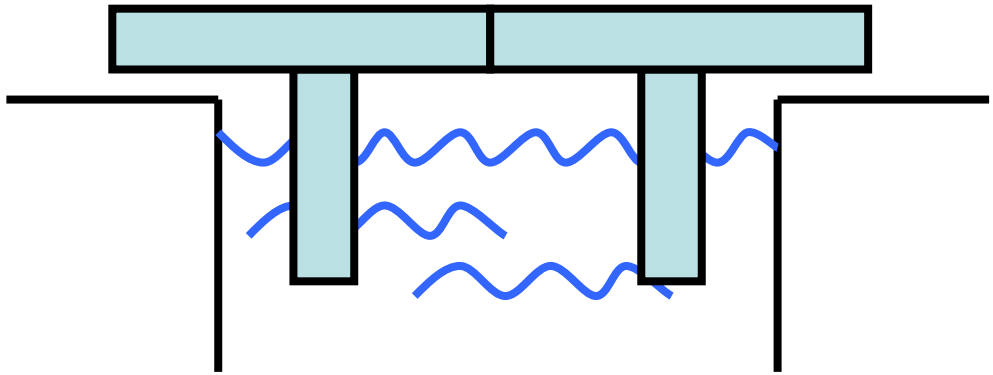
- 无前驱的顶点有：**L**
- 输出：**A,B,C,E,F,G,H,D,I,J,K,L**
- 删除**L**和以**L**为弧尾的弧

拓扑排序算法

```
Top_Sort (Grah G ){
建立入度为0的顶点栈S;  m=0;
  while (!EmptyStack(S)) {
     Pop(S,v);   printf(v); ++m;        //输出入度为0的顶点
     w:=FirstAdj(v);                    //取v的第一个邻接点
     while (w) {
        inDegree[w]--;                          //w的入度 –
        if(inDegree[w] ==0)   Push(S,w);
        w:=nextAdj(v,w);                  //取v的下一个邻接点
     }
  }
 if (m<n) printf( “图中有回路” );
}
```

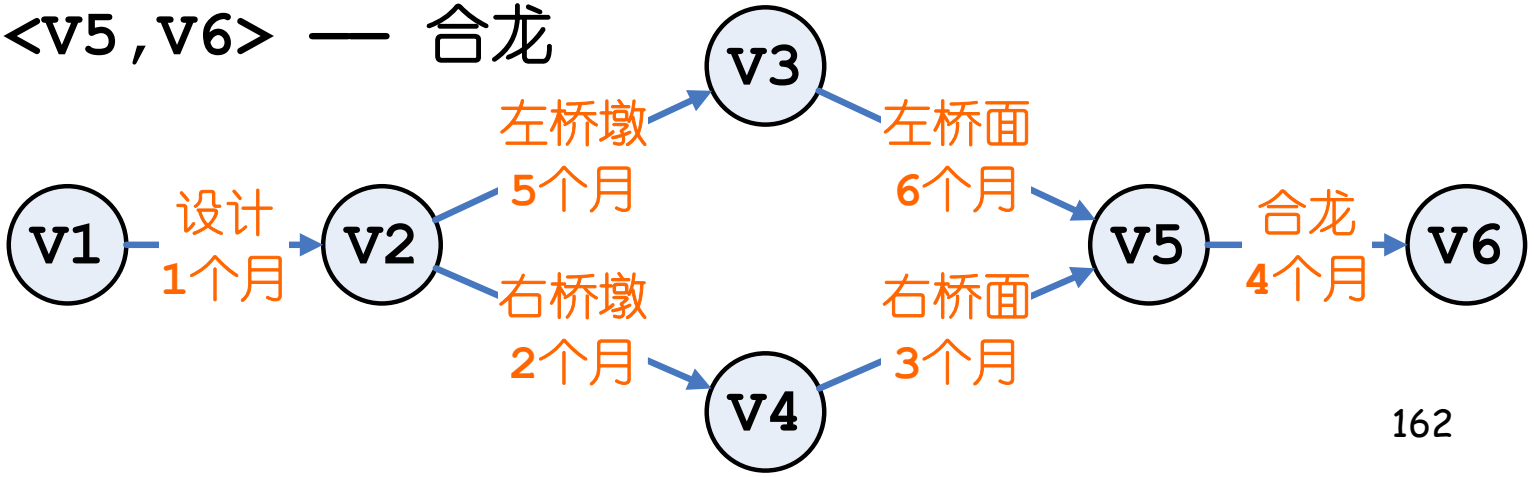
有向无环图的应用：关键路径

- 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



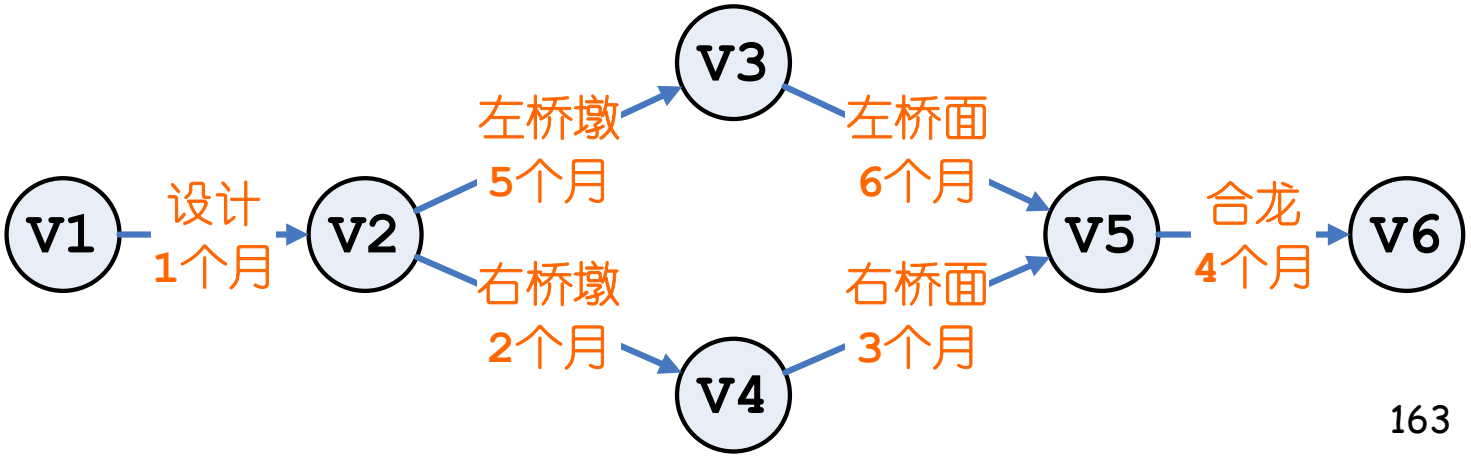
有向无环图的应用：关键路径

- 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
    - <V2,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



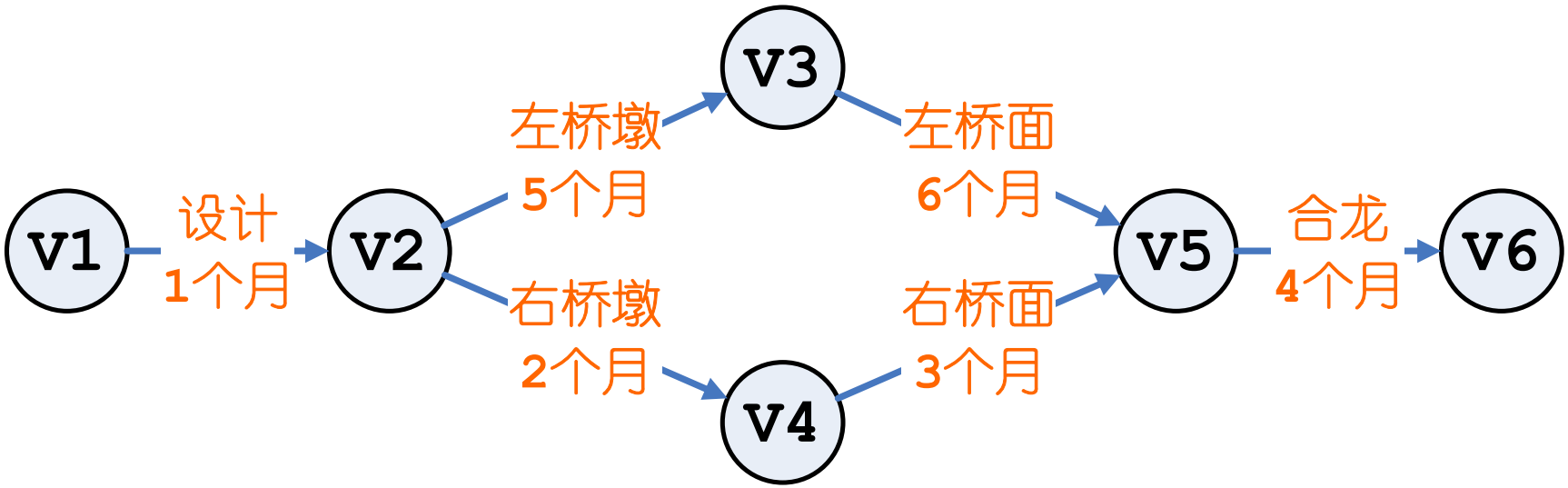
有向无环图的应用：关键路径

- 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

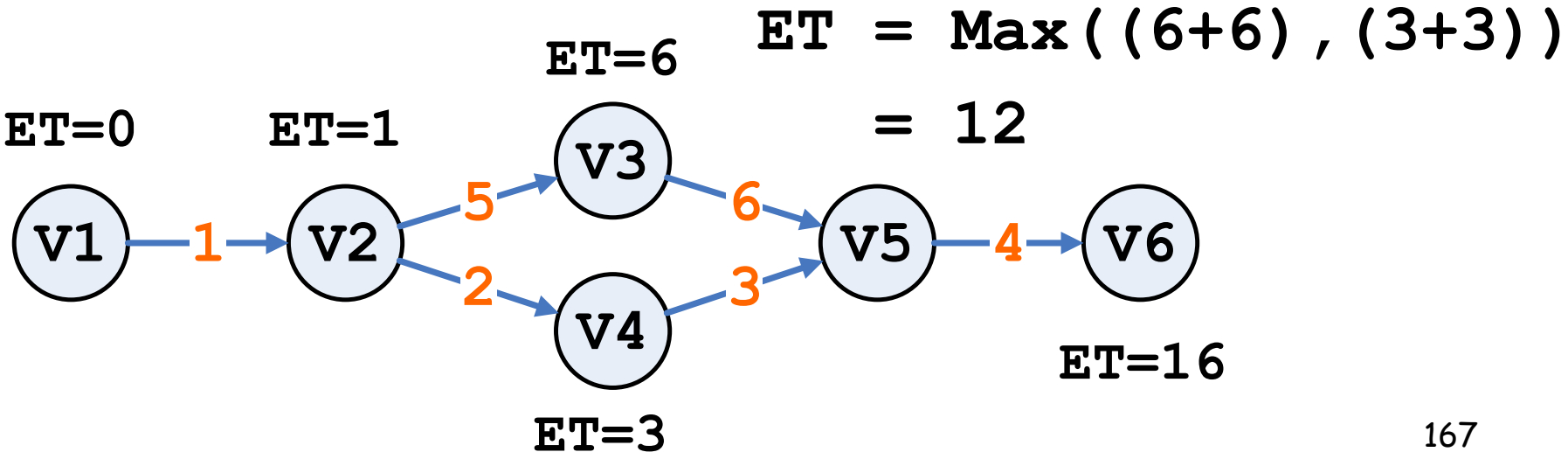
- **AOE**图在某些工程估算方面非常有用。如：
  （1） 完成整个工程至少需要多少时间(假设网络中没有环)？
  （2） 那些活动是影响工程进度的关键活动？
- 工程的最短完成时间：从开始点到完成点的最长路径长度—称为关键路径.
- 从开始点**v1**到**vi**的最长路径长度称为事件**vi**的最早发生时间， 而在不影响整个工程进度情况下时间**vi**有一个最迟发生时间.

有向无环图的应用：关键路径

- 关键路径上的点vi的最早发生时间和最迟发生时间一定是相等的.马不停蹄!
- 即关键路径上的活动和事件分别是关键活动和关键事件. 非关键活动则可以悠闲点
- 如果能够缩短关键活动的时间,则可以提前完成工程. 规定你2个小时完成的考试,你半个小时就完成了.

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)

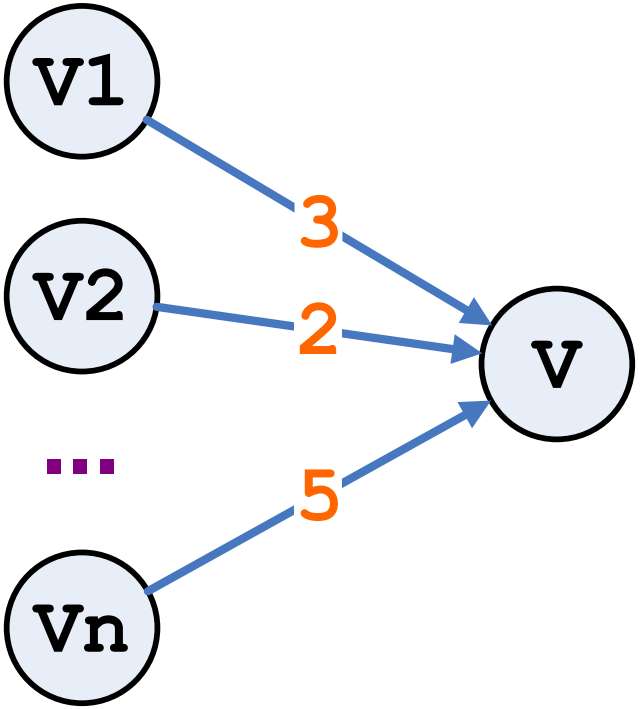
$$ET(V) = \max_i(ET(V_i) + dut(i, V))$$

ET(V1) = 2

ET(V2) = 4

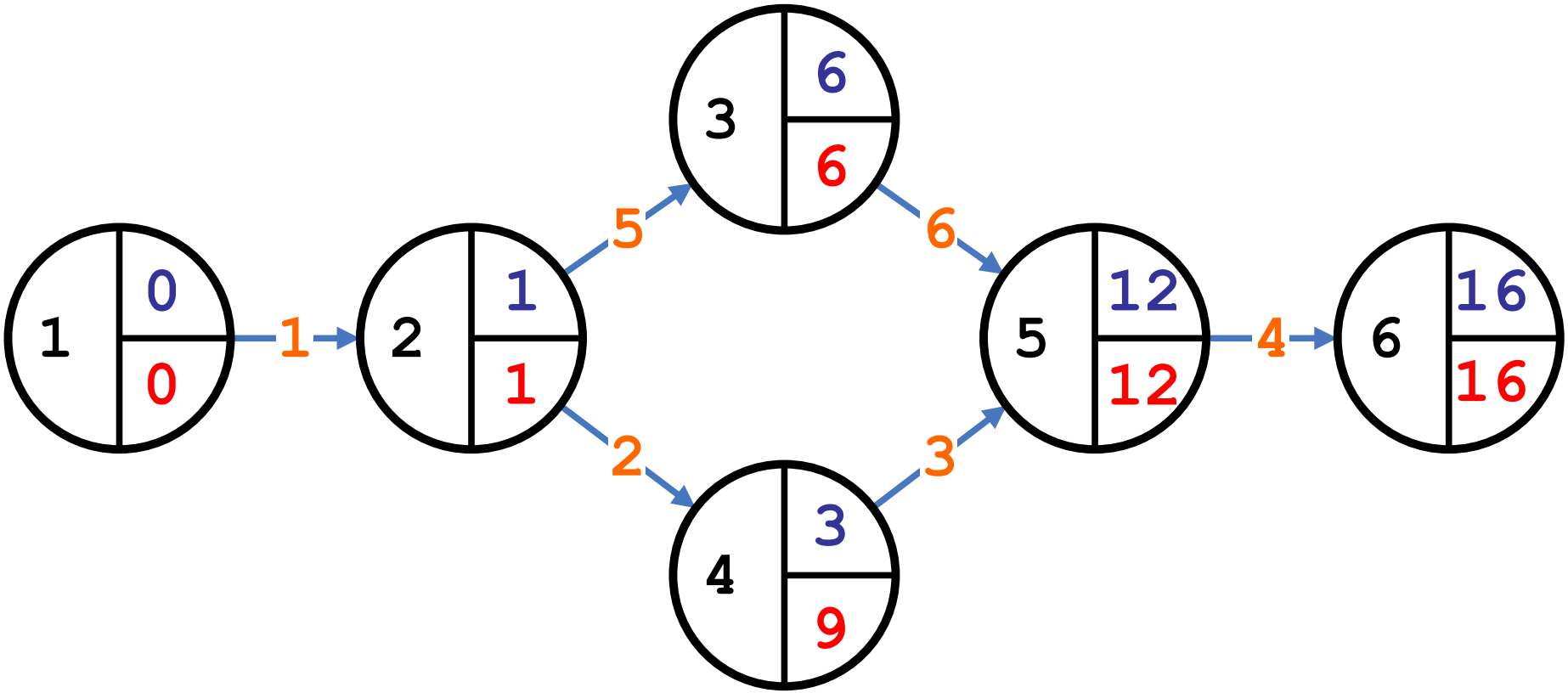
...

ET(Vn) = 1



有向无环图的应用：关键路径

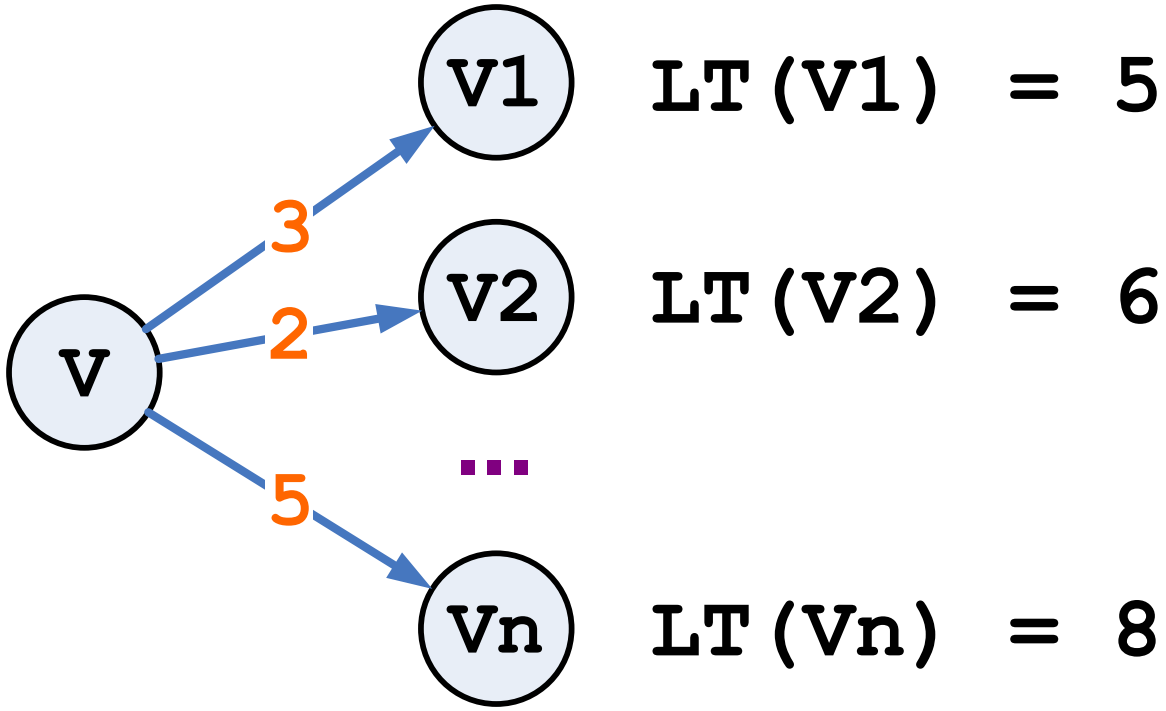
- 事件的最晚发生时间(Latest Time)



有向无环图的应用：关键路径

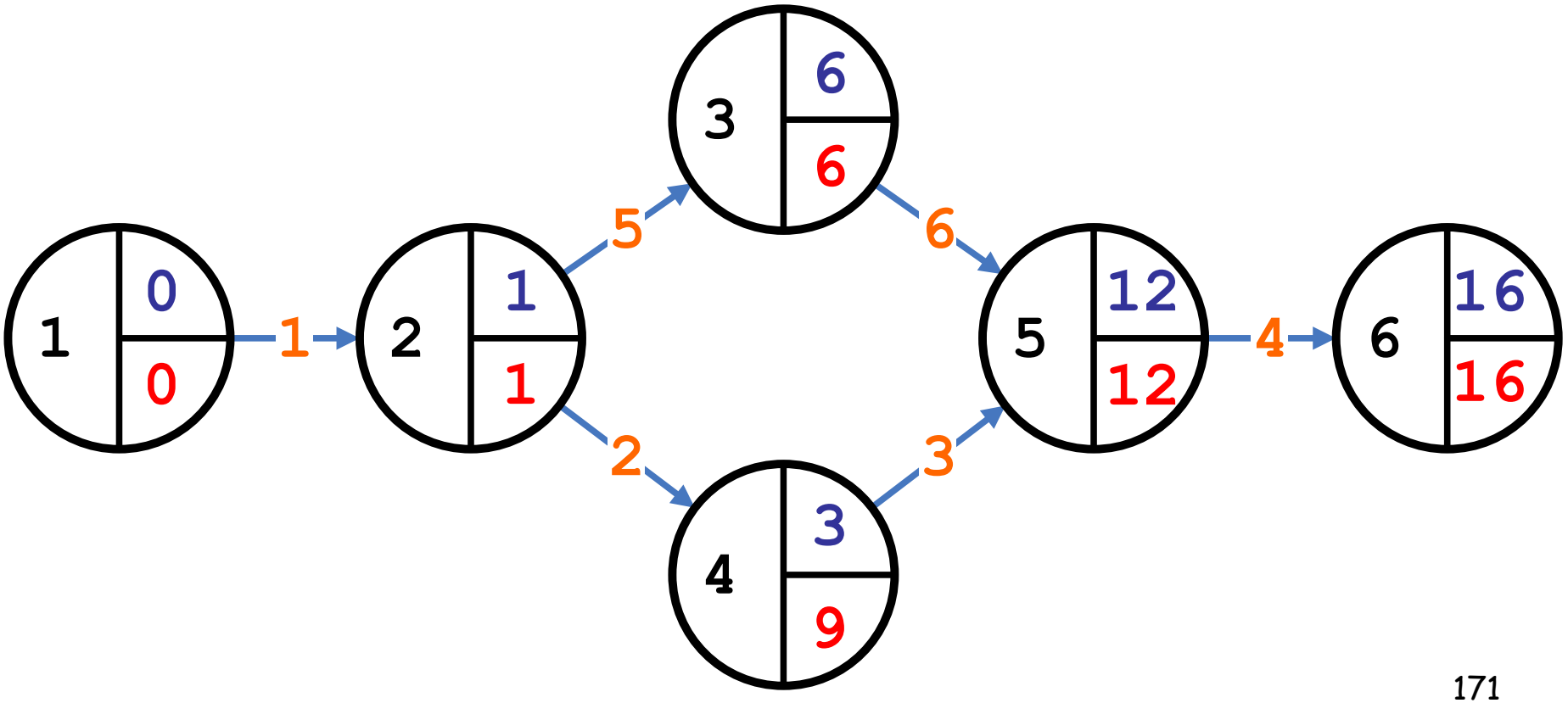
- 事件的最晚发生时间(Latest Time)

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



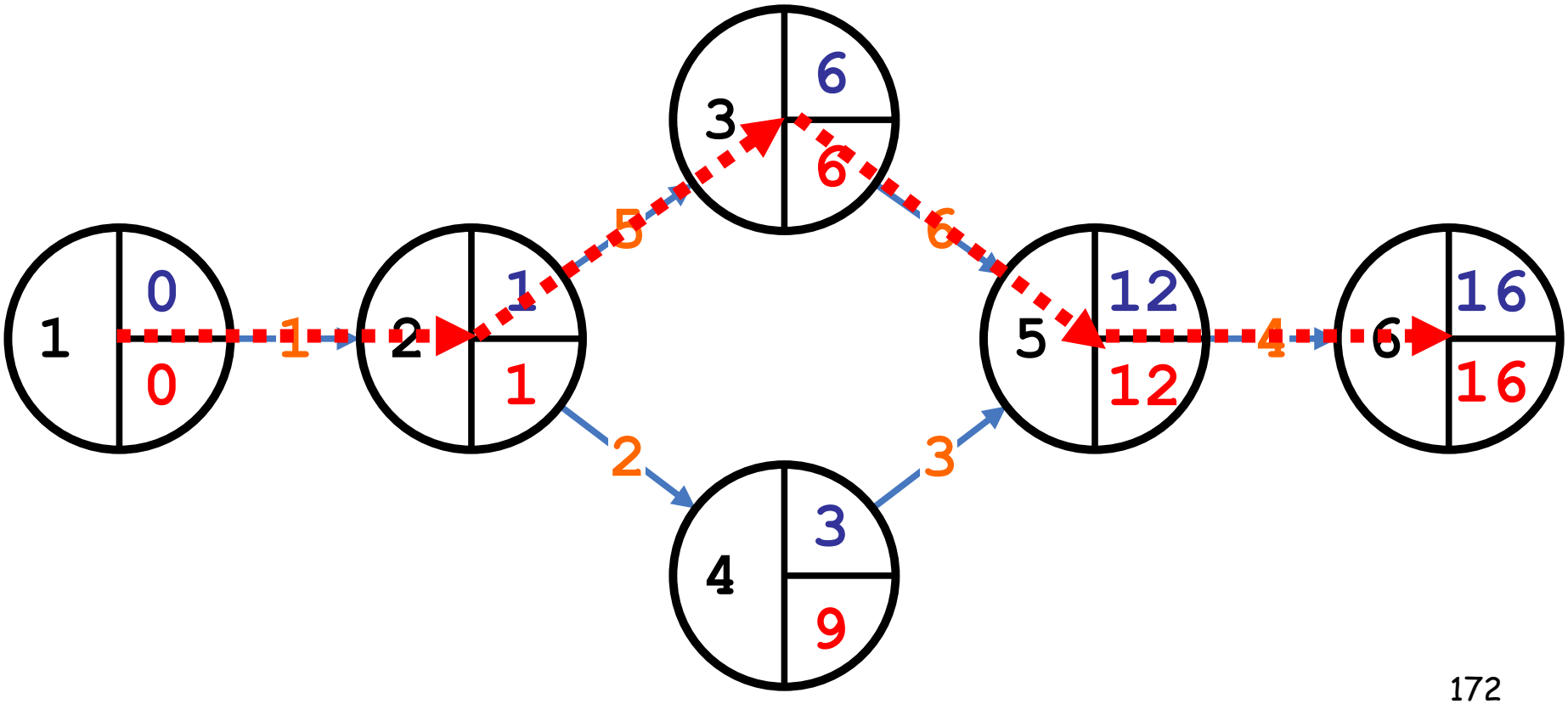
有向无环图的应用：关键路径

- 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：关键路径

- 求关键路径的算法：

扫描修改一下拓扑排序算法就行了.如：

a)初始化ve[i] = 0;开始拓扑排序

b)在删除<vj,vk>边时,若

ve[j]+dut(<j,k>)>ve[k]

则 ve[k] = ve[j]+dut(<j,k>)

求vl[i]不过是对逆拓扑序列进行

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

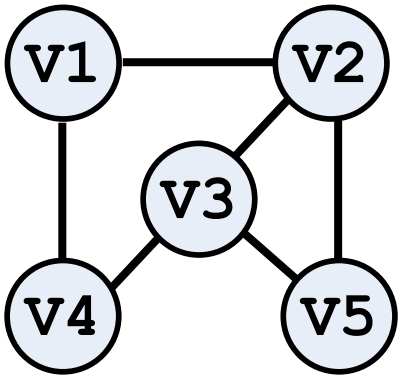
【复习提问】

- 图的最常用的两种存储结构是？ 31 37
- 如果从一个顶点出发访问它的邻接点？
- 如何区分已经访问和未访问过的邻接点？
- 路径长度和带权路径长度？

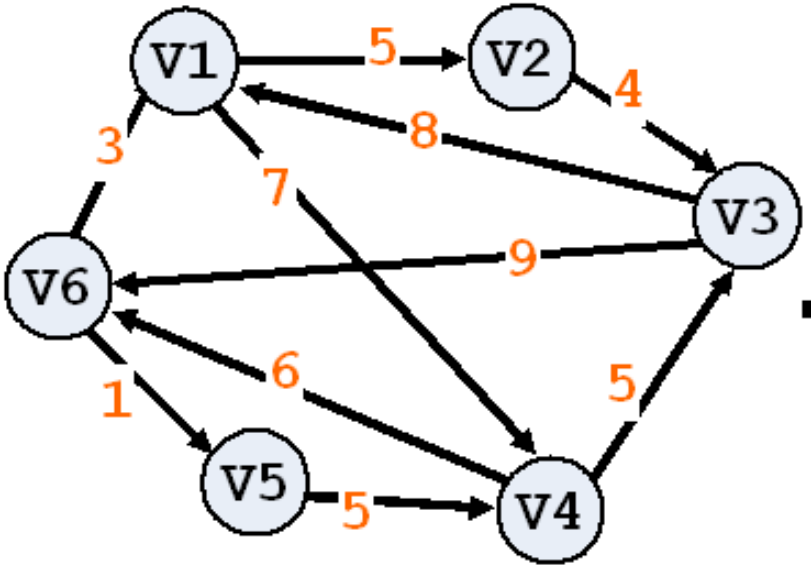
  路径长度：路径上边的数目

  带权路径长度：路径上边的权值之和

【复习提问】

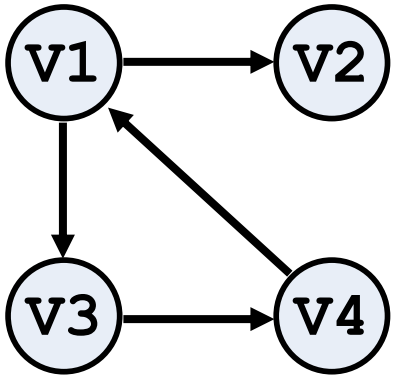


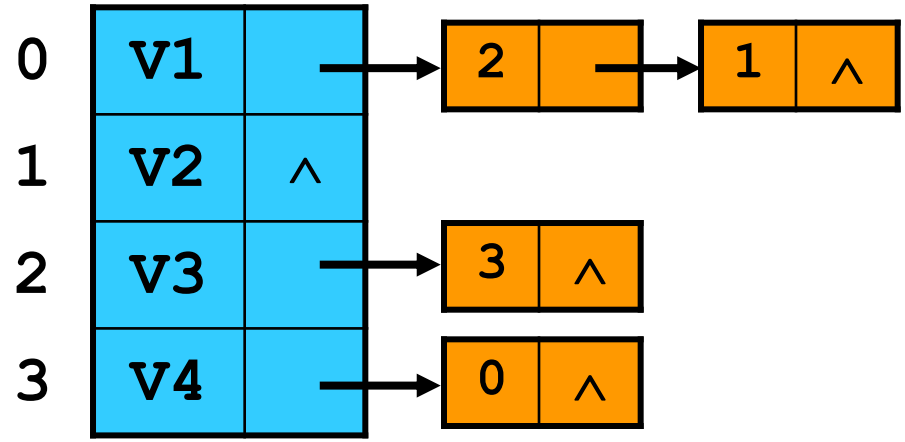
$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{pmatrix}$$

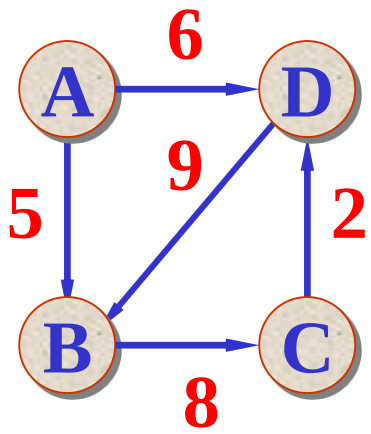


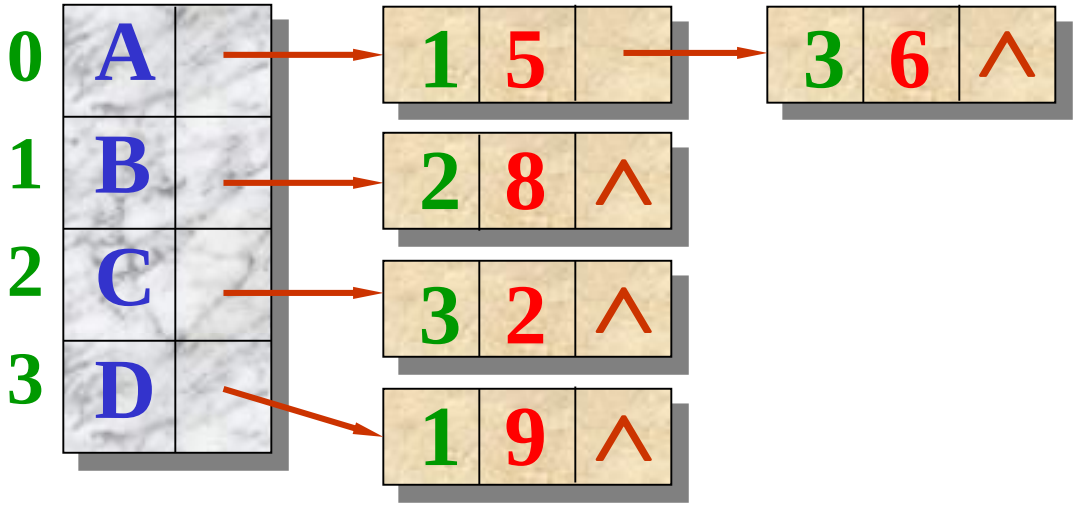
$$\begin{pmatrix} \infty & 5 & \infty & 7 & \infty & \infty \\ \infty & \infty & 4 & \infty & \infty & \infty \\ 8 & \infty & \infty & \infty & \infty & 9 \\ \infty & \infty & 5 & \infty & \infty & 6 \\ \infty & \infty & \infty & 5 & \infty & \infty \\ 3 & \infty & \infty & \infty & 1 & \infty \end{pmatrix}$$

【复习提问】


V1
V2
V3
V4


0
1
2
3
V1
V2
V3
V4
2
1
3
0
^


A
D
B
C
6
9
5
2
8


0
1
2
3
A
B
C
D
1 5
3 6
2 8
3 2
1 9
^

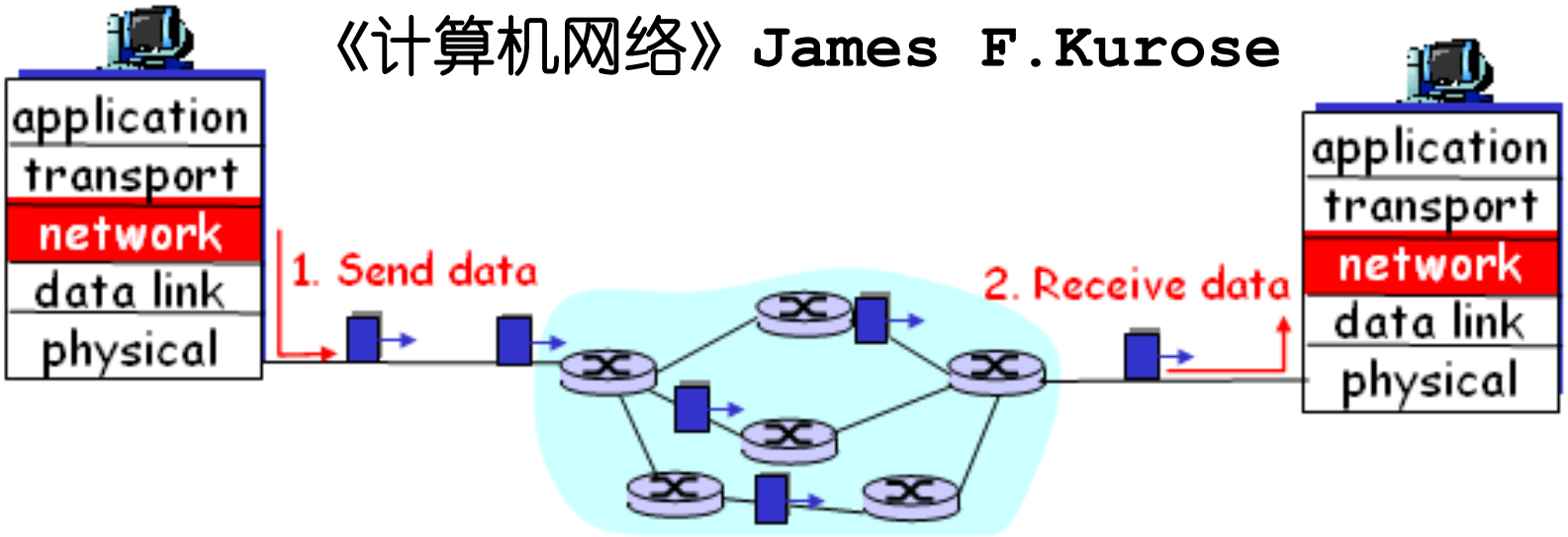
顶点表　　出边表

最短路径问题-典型应用

1．生活中的例子：日常出行，需要知道从A地到B的多条可能路径中那条最短？

最短路径问题-典型应用

2．网络路由（Routing Protocal）:数据报从源ip地址以最低代价传送到目的ip地址？



Global:

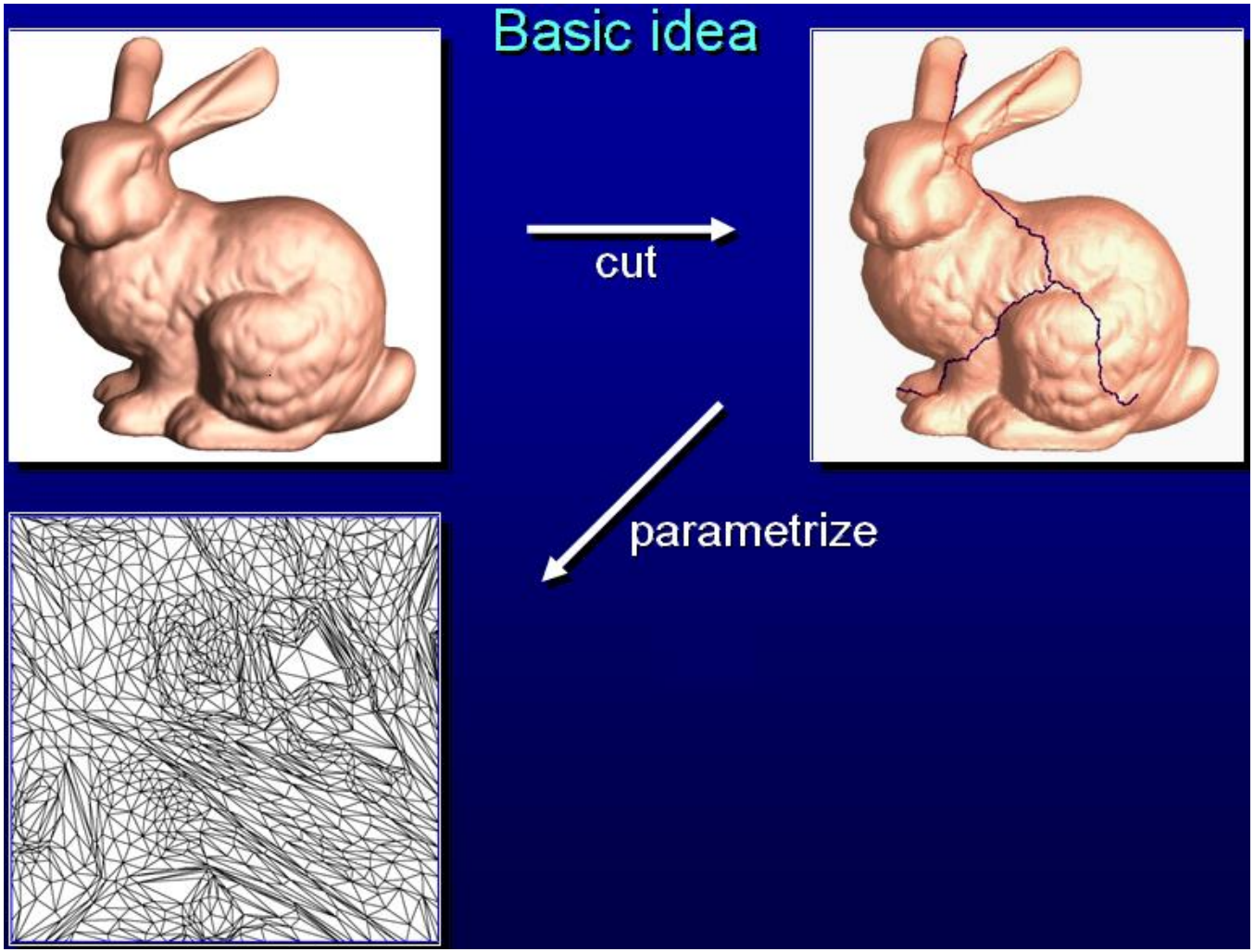
Link-State Routing Algorithm—Dijkstra

Decentralized:

Distance Vector Routing Algorithm

最短路径问题-典型应用

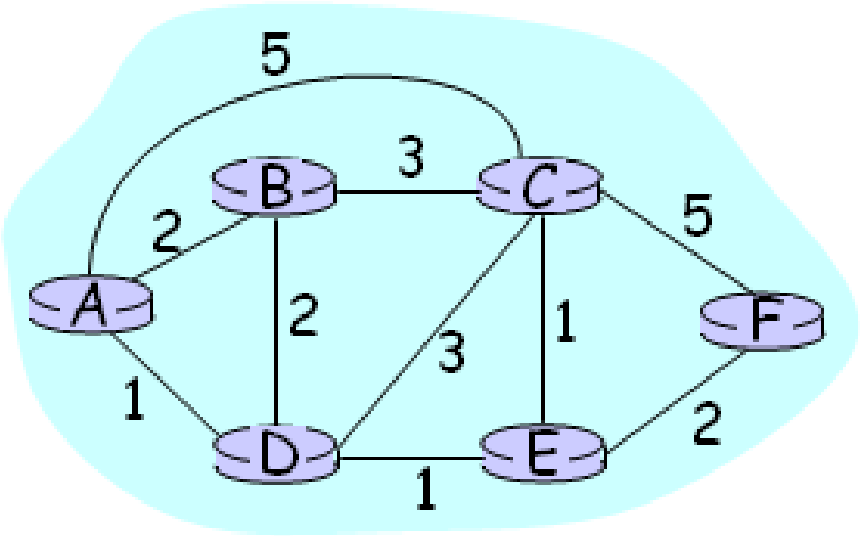
3． 3D网格模型的测地线－两点间的最短距离



最短路径问题-问题抽象

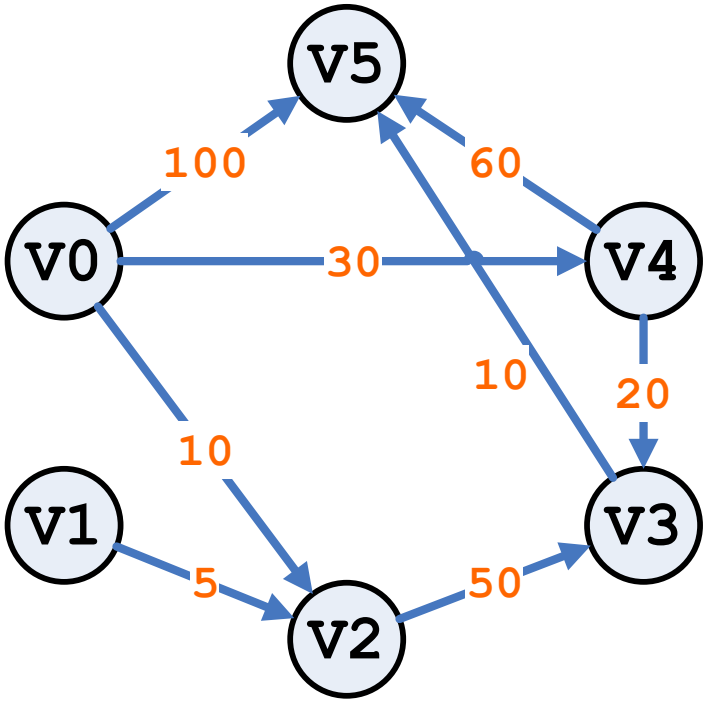
- 问题抽象：在带权图中A点（源点）到达B点（终点）的多条路径中，寻找一条各边权值之和最小的路径，即最短路径。
  带权路径长度
- 权值：路径、代价、时间等等，取决于具体应用

- graph nodes are routers
- graph edges are physical links
  - link cost: delay, $ cost, or congestion level



最短路径问题-分类

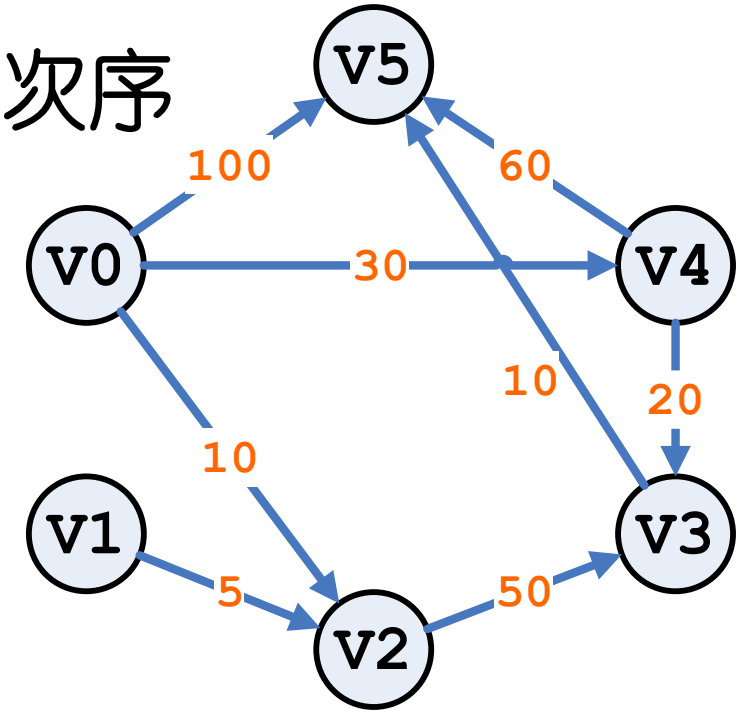
- 两种常见的最短路径问题
  - 从一个顶点出发，求去往其它所有顶点的最短路径及其长度(也称单源最短路径问题)
  - 求任意两个顶点的最短路径及其长度



最短路径问题：Dijkstra算法

- **Dijkstra算法**
  - 求从一个顶点出发到其它所有顶点的最短路径及其长度
    - 按最短路径长度递增的次序产生各个最短路径

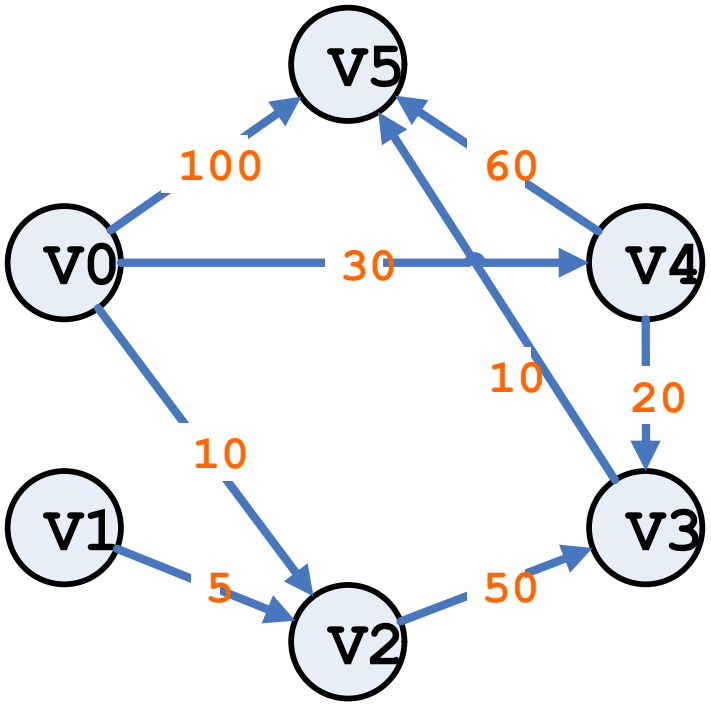
$D(V_2) < D(V_4) < D(V_3) < \ldots$



最短路径问题：Dijkstra算法

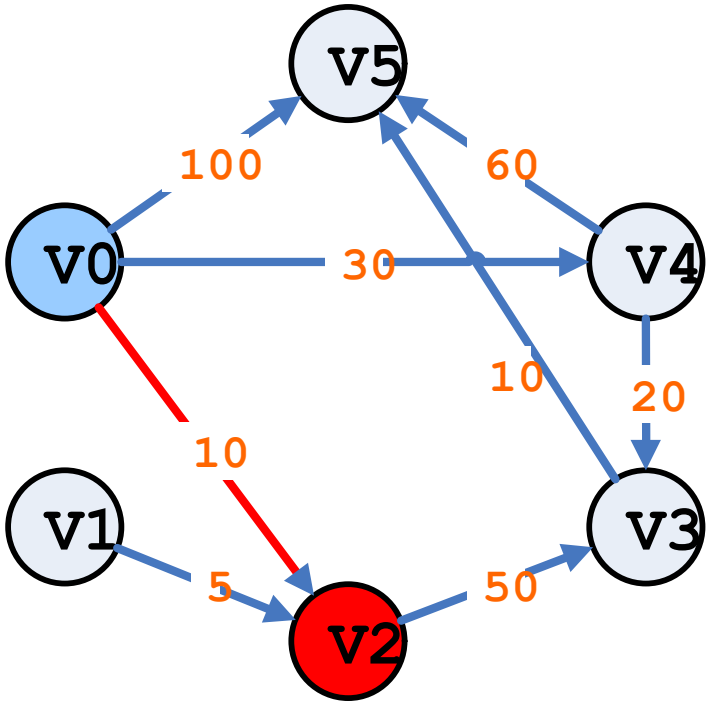
- 最短路径长度最短的最短路径的特点：

  在这条路径上，必定只含一条弧，并且这条弧的权值最小。



最短路径问题：Dijkstra算法

- 最短路径长度次短的最短路径的特点：
  它只可能有两种情况：或者是直接从源点到该点(只含一条弧)； 或者是，从源点经过顶点v2，再到达该顶点(由两条弧组成)。

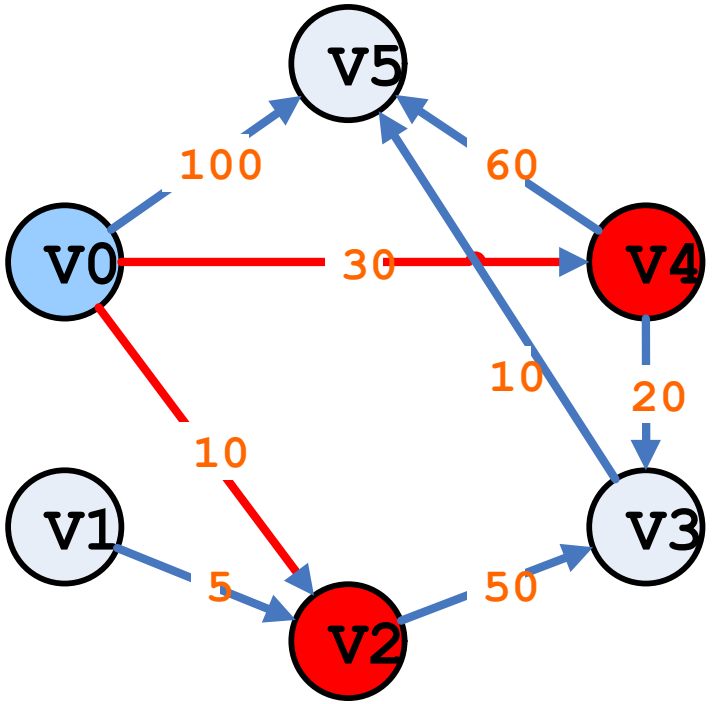


最短路径问题：Dijkstra算法

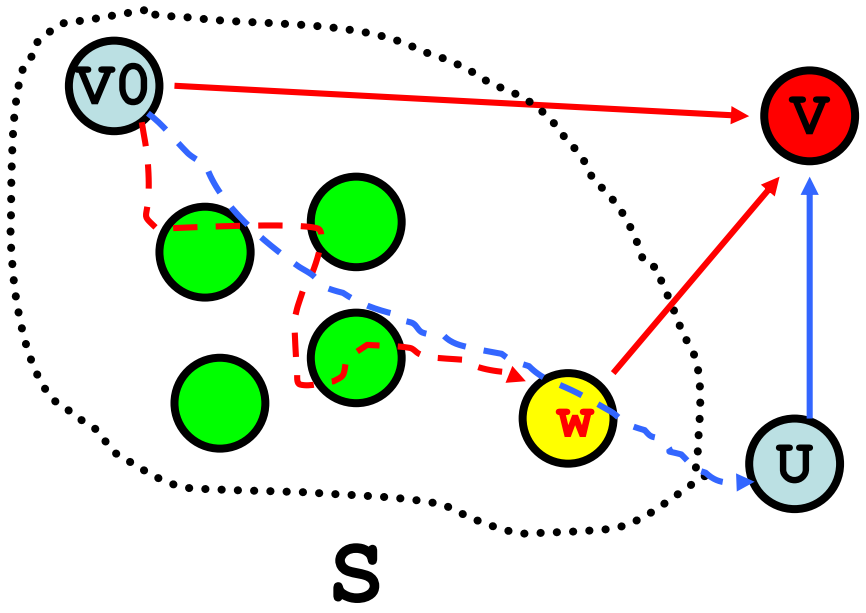
- 第3条最短路径的特点：

它只可能有两种情况：或者是直接从源点到该点(只含一条弧)； 或者是，从源点经过顶点(v2,v4)，再到达该顶点(由两条弧组成)。

v4的加入，是否会改变其他未知最短距离的顶点的最短距离呢？



最短路径问题：Dijkstra算法



反证：

设w是刚求得最短距离的点，v是下一条最短距离的终点，则其最短路径不可能绕道集合S之外的点如u到达.

那样的话，到u的最短距离更短了！

因此，当w加入S后，只要检查每个从w出发的直接邻接点v，看看绕道w会不会使D(v)的距离变得更小？

最短路径问题：Dijkstra算法

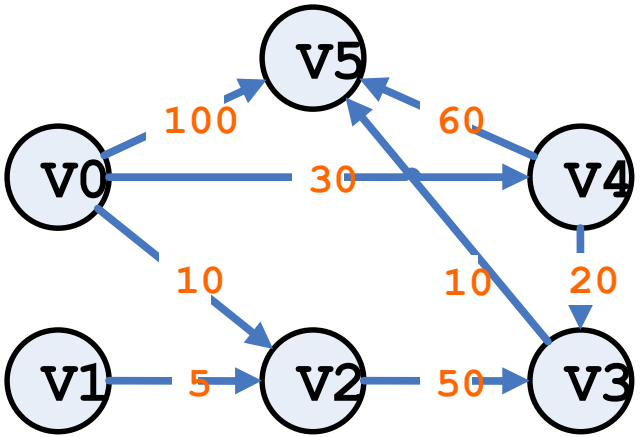
- 算法实现
  - 数据结构
    - **D = Distance**：起点到终点的当前距离
    - **P = Path**：终点的前一个顶点
    - **S = Set**：已知最短路径的目的顶点集合
  - 初始化
    - **S = {V0}**
    - 对于所有顶点**v**
      - 若存在弧**<V0,V>**，**D(v) = arcs(V0,V)**
      - 否则 **D(v) = ∞**

最短路径问题：Dijkstra算法

- 循环(直到找不到从v0可以达到的顶点)
  - 在不在S中的顶点中查找D(w)最小的顶点w
  - 把w加入S，即w已知最短路径
  - 更新w的所有邻接点v的D(v)和P(v)

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

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



P	v0	v0	v2	v0	v3

最短路径问题：Dijkstra算法

对于所有顶点v　　n次

```
   P(v) = v0;
   若存在弧<V0,V>, D(v) = arcs(V0,V)
   否则 D(v) = ∞
```

循环(直到找不到可加入S的顶点){　　n-1次

```
   在不在S中的顶点中查找D(w)最小的顶点w　　n次
   把w加入S，即w已知最短路径
   更新w的所有邻接点v的D(v)　　n次?
     if( D(w)+c(w,v) < D(v) )
       D(v) = D(w)+arcs(w,v); P(v) = w;
}
```

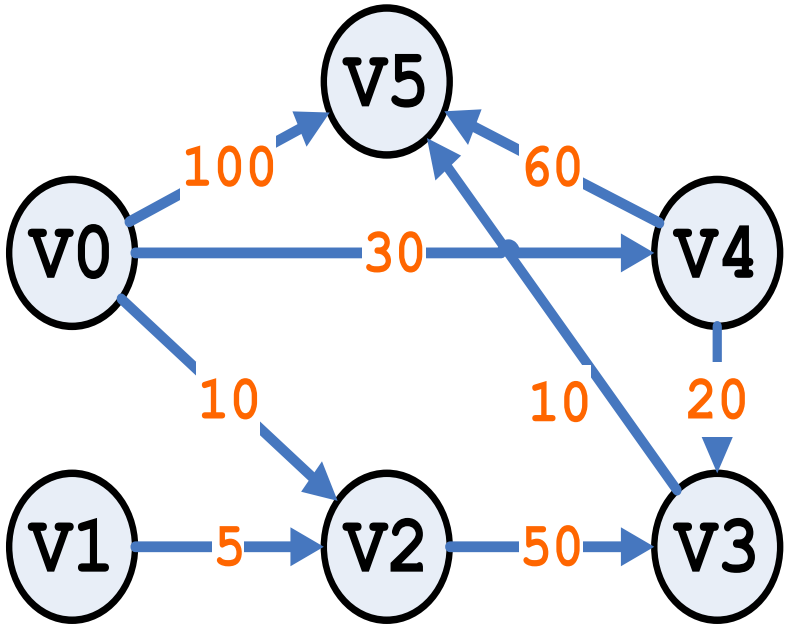
时间复杂度 = $O(n^2)$

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(n3)

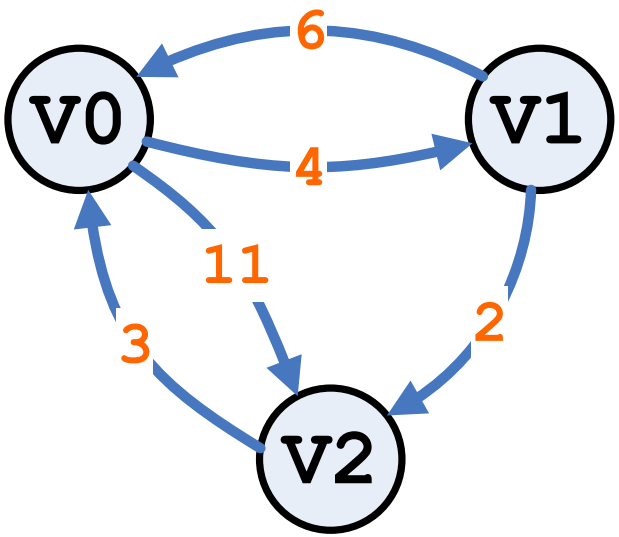
方法二：Floyd算法，时间复杂度也是O(n3)



最短路径问题：Floyd算法

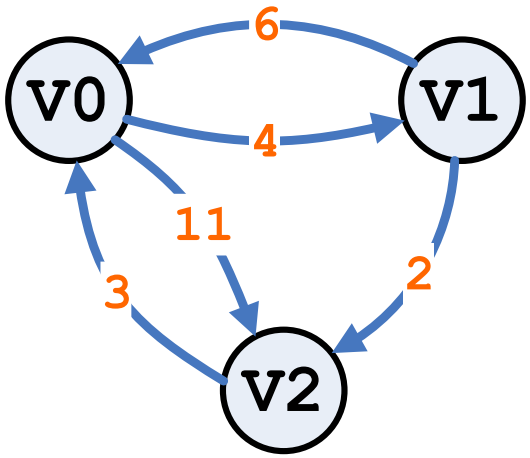
- 原理：用矩阵表示两点的距离
- 初始时 距离矩阵 = 邻接矩阵
  - 任意两个顶点的最短路径就是直接连接的弧

	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



D12 < D10 + D02 ?

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

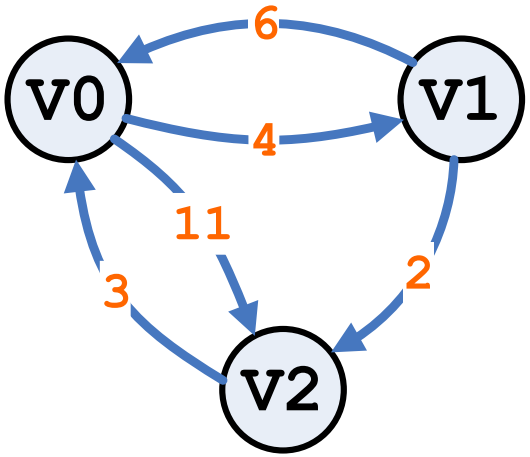
绕道V0 →

	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

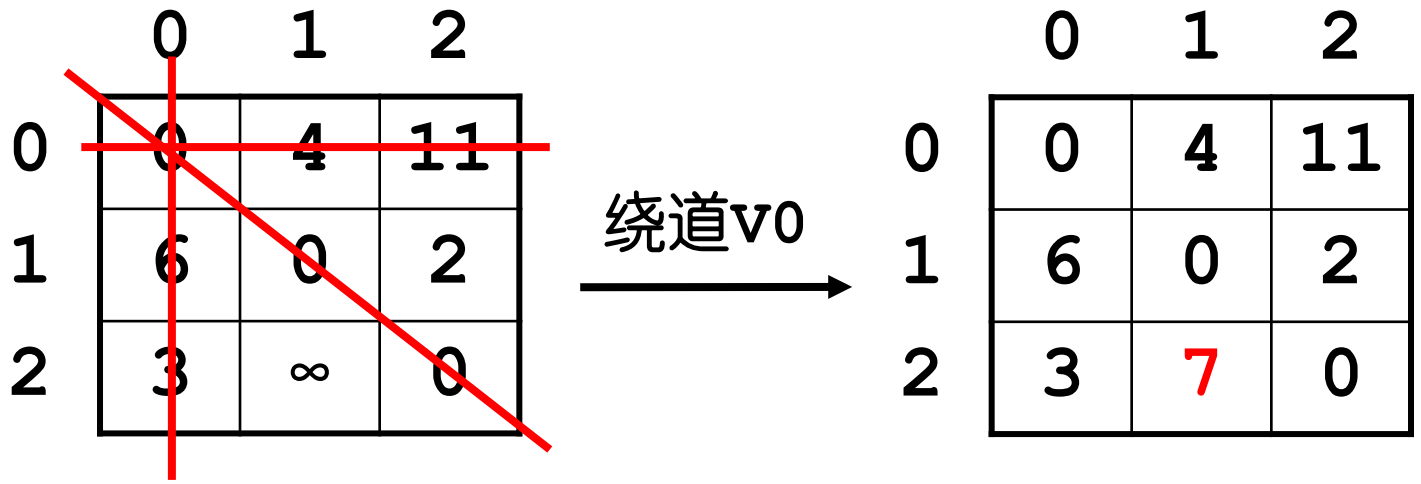
绕道v2

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

最短路径问题：Floyd算法

- 绕道Vk时
  - 不需要扫描整个矩阵，对角线，第k行，第k列可以省略
  - 这样每次只须扫描$n^2-(3n-2)$个单元

Dik+ Dkj< Dij



最短路径问题：Floyd算法

- 算法分析
  - 每尝试一个顶点，需$n^2-(3n-2)$
  - 分别要尝试从n个顶点绕道
  - 时间复杂度 = $O(n^3-3n^2+2n)= O(n^3)$

	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

1）初始化距离矩阵D和路径矩阵P

D = 邻接矩阵；P = 弧矩阵

2）考虑绕道每个顶点Vk：

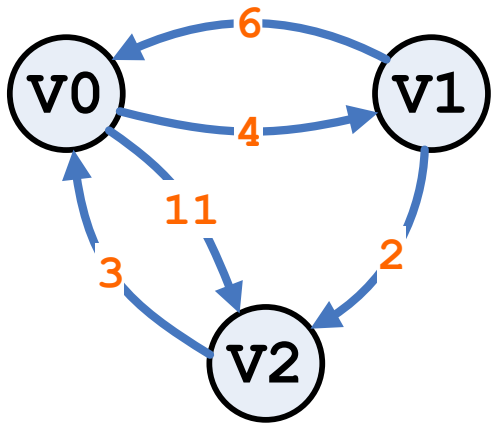
```
for(int k=0; k<n; k++)
  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      if(D[i][k]+D[k][j]<D[i][j]){
         D[i][j] = D[i][k]+D[k][j];
         P[i][j] = P[i][k]||P[k][j];
      }
```

绕道v0

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

→

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




```
D21 = D20+D02;
P21 = P20||P02;
```

P

	0	1	2
0		AB	AC
1	BA		BC
2	CA		

→

P

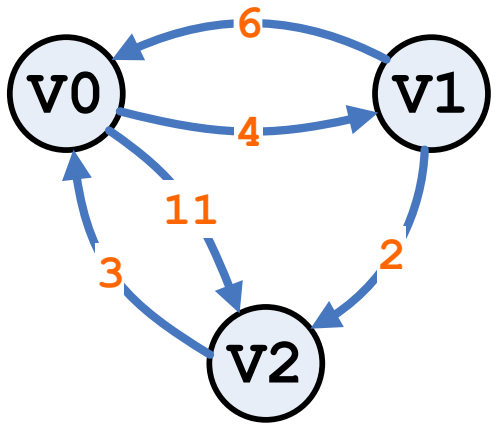
	0	1	2
0		AB	AC
1	BA		BC
2	CA	CAB	

绕道v1

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

→

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




```
D02 = D01+D12;
P02 = P01||P12;
```

	0	1	2
0		AB	AC
1	BA		BC
2	CA	CAB	

P

→

	0	1	2
0		AB	ABC
1	BA		BC
2	CA	CAB	

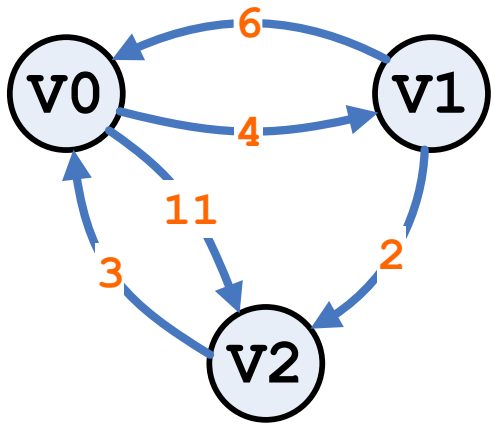
P

绕道v2

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

→

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




```
D10 = D12+D20;
P10 = P12||P20;
```

	0	1	2
0		AB	ABC
1	BA		BC
2	CA	CAB	

P

→

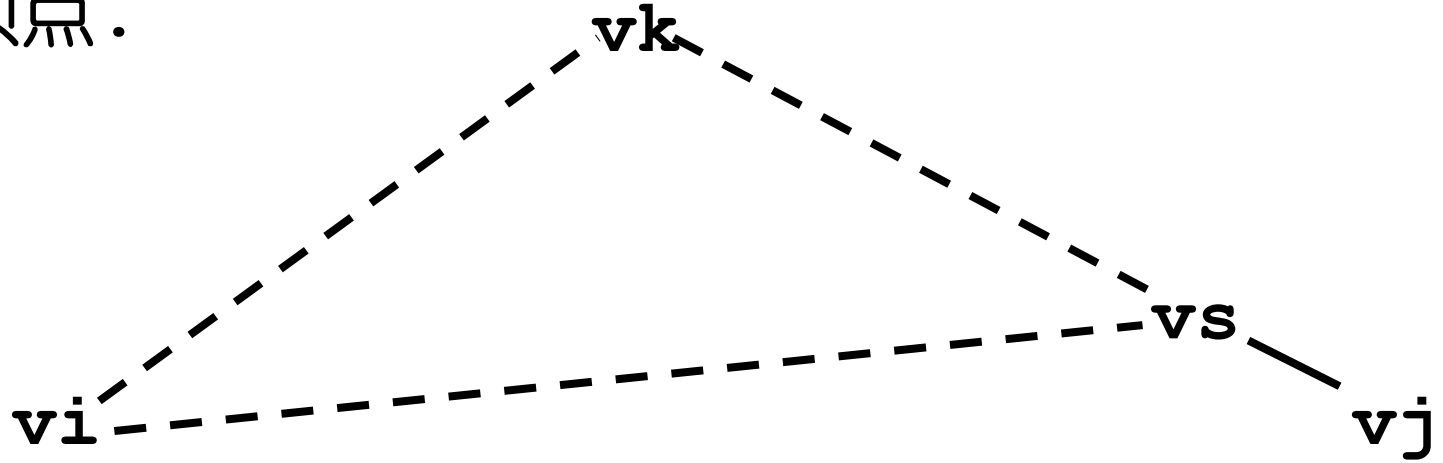
	0	1	2
0		AB	ABC
1	BCA		BC
2	CA	CAB	

P

最短路径问题：Floyd算法

- 改进的路径数组

  原来的路径数组元素P[i][j]记录了(绕道过一些顶点后)顶点vi到vj的当前最短路径上的所有顶点.



Dik+Dkj = Dik+Dks+w(s,j)<=Dis+Dsj

➔ Dik+Dks<=Dis (1)

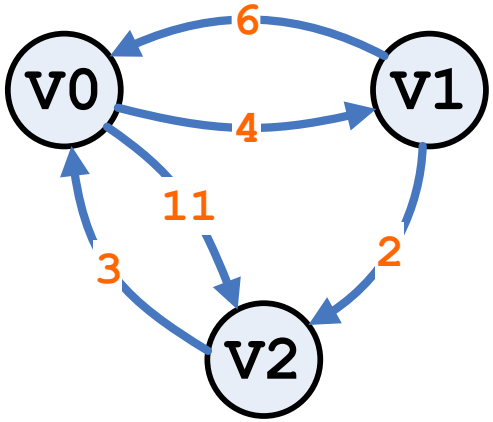
Dis<= Dik+Dks (2)

最短路径问题：Floyd算法

- 距离矩阵D和路径矩阵P

  P[i][j]表示

  路径上终点vj的前一顶点



	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

D

	0	1	2
0		A	A
1	B		B
2	C		

P

绕道vk

```
if( D[i][k]+D[k][j]< D[i][j] ){
   D[i][j] = D[i][k] + D[k][j];
// P[i][j] = P[i][k] ||P [k][j];
   P[i][j] = P[k][j];
}
```

绕道v0

	0	1	2
0	0	4	11
1	6	0	2
2	3	∞	0

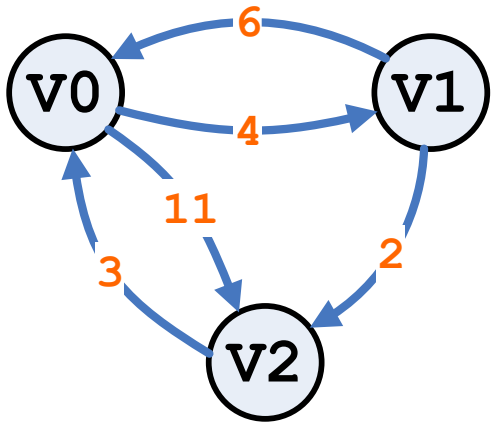
D

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

	0	1	2
0		A	A
1	B		B
2	C		

P

	0	1	2
0		A	A
1	B		B
2	C	**A**	




```
D21 = D20+D01;
P21 = P01;
```

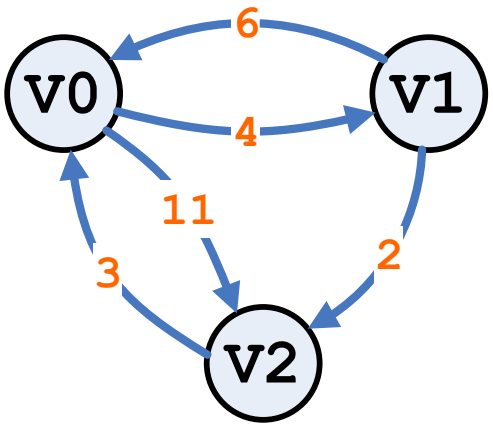
V2到V1的路径是：　C　A　B

绕道v1

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

D

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



	0	1	2
0		A	A
1	B		B
2	C	A	

P

	0	1	2
0		A	B
1	B		B
2	C	A	

```
D02 = D01+D12;
P02 = P12;
```

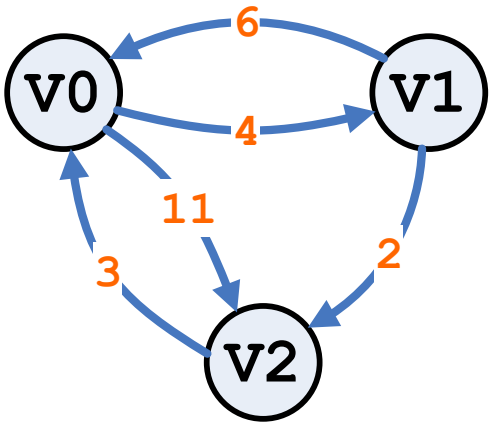
V0到v2的路径是： A B C

绕道V2

D

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

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



P

	0	1	2
0		A	B
1	B		B
2	C	A	

	0	1	2
0		A	B
1	B		B
2	C	A	

```
D10 = D12+D20;
P10 = P20;
```

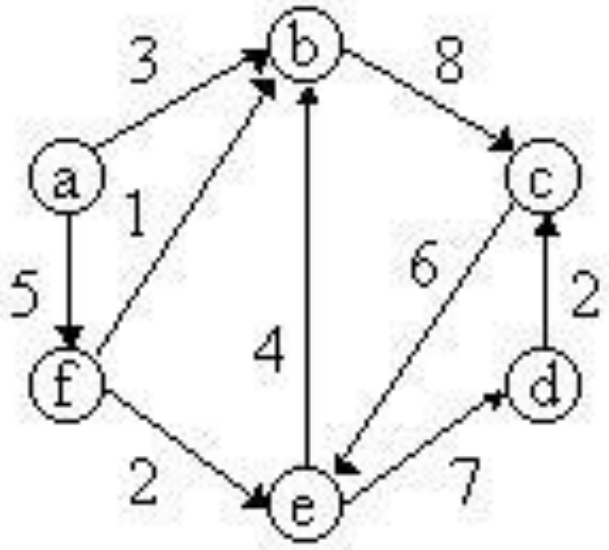
V1到v0的路径是：　　B　C　A

最短路径问题:小结和要求

- 迪杰斯特拉算法按照最短路径程度从小到大次序依次求出各个最短路径. 当一个顶点w的最短路径求出后,看看其邻接点v的距离会不会因为绕道w更加近?
- Floyd算法从初始邻接矩阵开始,依次加入个个顶点vk,看看D(vi,vj)会不会因为绕道vk更加近?
- 能手工给出有向网或无向网的两种最短路径算法的求解过程.
- 能够编写两个算法. ---实验之一

最短路径问题：练习

【例1】设有向网如下，1)用迪杰斯特拉算法求从顶点 a 出发到其余各顶点的最短路径。2)用Floyd算法求从任意2个顶点的最短路径。



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

1.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）

1. 在含n个顶点和e条边的无向图的邻接矩阵中,零元素个数为(　　)

A. e　　B. 2e　　C. n^2-e　　D. n^2-2e

2. 假设一个有n个顶点和e条弧的有向图用邻接表表示,则删除与某个顶点vi相关的所有弧的时间复杂度是(　　)

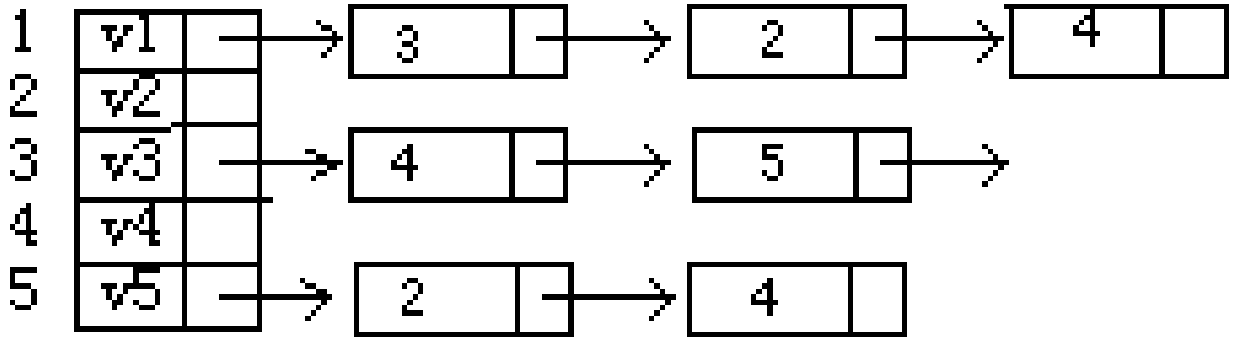
A. O(n)　　B. O(e)　　C. O(n+e)　　D. O(n*e)

作业

3．一有向图的邻接表存储结构如图1所示，按深度优先算法，从v1出发得到的顶点序列为（　　）

A) v1,v3,v2,v4,v5　　B) v1,v3,v4,v2,v5

C) v1,v2,v3,v4,v5　　D) v1,v3,v4,v5,v2



作业

4．在有向图中所有结点入度之和是所有结点出度之和的（　）倍

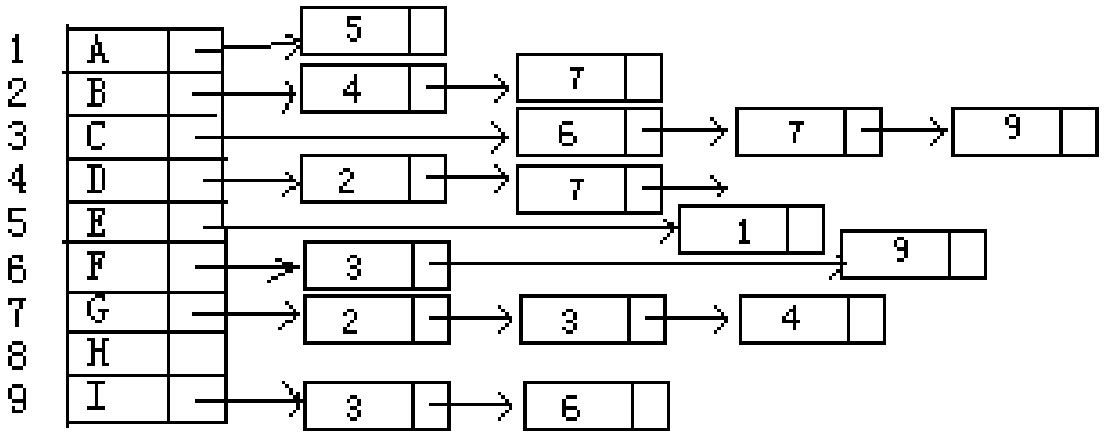
A）0.5　　B）1　　C）2　　D）4

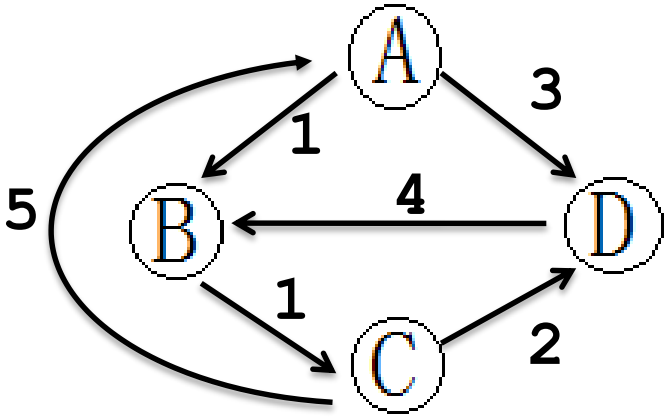
5．设G是一n个顶点的有向图，其存储结构分别为：

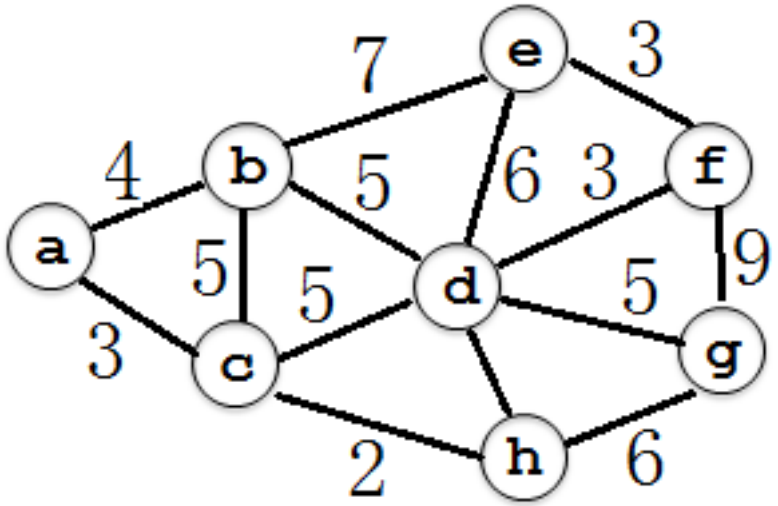
(1)邻接表　　(2) 邻接矩阵

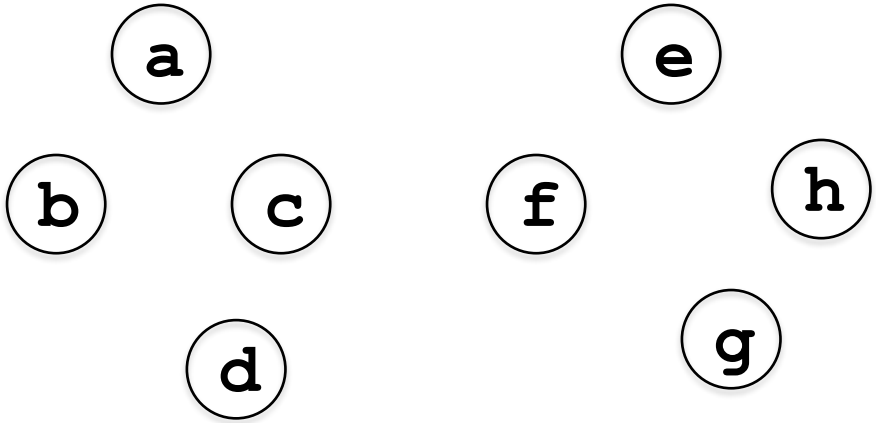
请写出相应结构上的计算有向图出度为0的顶点个数的算法。

6．已知图G的邻接表如下，画出图G的所有连通分量。




A
B
C
D
1
3
4
5
1
2


a
b
c
d
e
f
g
h
7
3
4
5
6
3
5
5
5
9
3
2
6


a
b
c
d
e
f
g
h