北京理工大学
BEIJING INSTITUTE OF TECHNOLOGY

数据结构与算法设计

课程内容

课程内容：数据结构部分

- 概述
- 线性表
- 栈与队列
- 数组与广义表
- 串
- 树
- 图
- 查找
- 内部排序
- 外部排序

课程内容：算法设计部分

- 概述
- 分治
- 动态规划
- 贪心
- 回溯
- ……
- ……
- ……
- ……
- 计算模型
- 可计算理论
- 计算复杂性

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

Contents

本章内容

6.1 图的定义与基本术语

图的定义

图(Graph)——图G是由两个集合V(G)和E(G)组成的,记为G=(V,E)

其中：V(G)是顶点的非空有限集

E(G)是边的有限集合，边是顶点的无序对或有序对。

图的分类

有向图

无向图

6.1 图的定义与基本术语

图的定义

有向图——有向图G是由两个集合V(G)和E(G)组成的。

其中：

V(G)是顶点的非空有限集。

E(G)是有向边（也称弧）的有限集合，弧是顶点的有序对，记为<v,w>，v,w是顶点，v为弧尾，w为弧头。

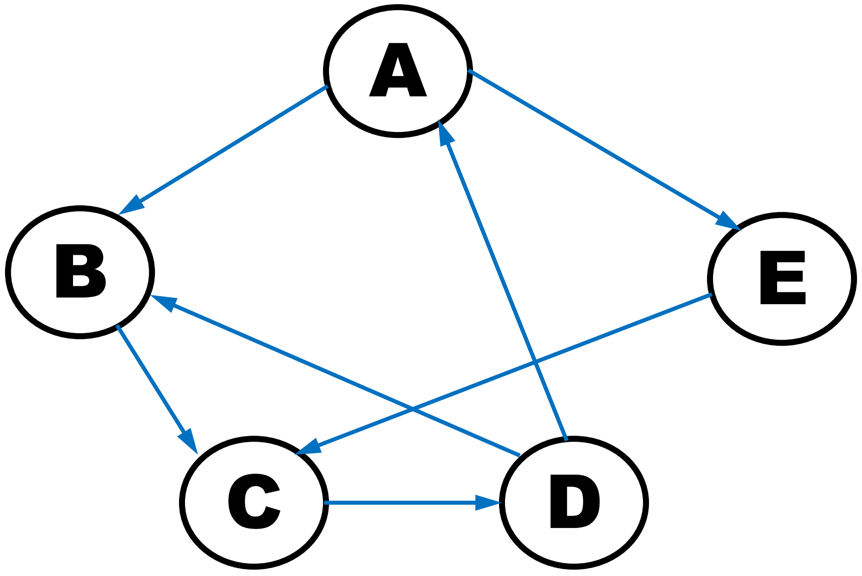
6.1 图的定义与基本术语

例如：有向图

G = <V, E>

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

E = {<A,B>, <A,E>, <B,C>, <C,D>, <D,B>, <D,A>, <E,C> }



6.1 图的定义与基本术语

图的定义

无向图——无向图G是由两个集合V(G)和E(G)组成的。

其中：

V(G)是顶点的非空有限集。

E(G)是边的有限集合，边是顶点的无序对，记为 (v,w) 或 (w,v)，并且(v,w) = (w,v)。

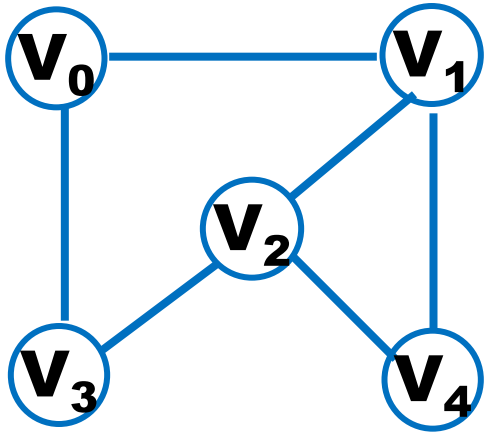
6.1 图的定义与基本术语

例如：无向图

$G = \langle V, E \rangle$

$V = \{ v_0, v_1, v_2, v_3, v_4 \}$

$E = \{ (v_0,v_1), (v_0,v_3), (v_1,v_2), (v_1,v_4), (v_2,v_3), (v_2,v_4) \}$



6.1 图的定义与基本术语

图的应用举例

例1. 交通图（公路、铁路）

顶点：地点　　　　边：连接地点的路

例2. 电路图

顶点：元件　　　　边：连接元件之间的线路

例3. 通讯线路图

顶点：地点　　　　边：地点间的连线

例4. 各种流程图

如产品的生产流程图。

顶点：工序　　　　边：各道工序之间的顺序关系

6.1 图的定义与基本术语

假定图中有n个顶点，e条边，则：

含有e=n(n-1)/2条边的无向图，称作无向完全图

含有e=n(n-1)条边的有向图，称之为有向完全图

若边或弧的数量e<nlogn，则称之为稀疏图（Sparse Graph），否则称作稠密图(Dense Graph)

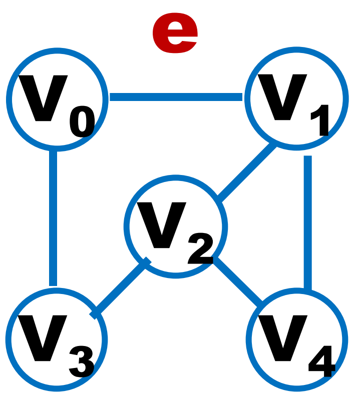
6.1 图的定义与基本术语

图的基本术语

邻接点及关联边

邻接点：边的两个顶点互为邻接点

关联边：若边 e = (v, u), 则称顶点v、u 关连边 e。



6.1 图的定义与基本术语

顶点的度、入度、出度

顶点V的度 = 与V相关联的边的数目

在有向图中：

顶点V的出度 = 以V为起点有向边数

顶点V的入度 = 以V为终点有向边数

顶点V的度 = V的出度+V的入度

设图G 的顶点数为 n，边数为 e

图的所有顶点的度数和 = 2*e

(每条边对图的所有顶点的度数和“贡献”2度)

路径、回路

无向图 G =（V，E）中的顶点序列$v_1, v_2, \ldots, v_k$，若 $(v_i,v_{i+1}) \in E$ ($i=1, 2, \ldots k-1$)， $v=v_1$, $u=v_k$，则称该序列是从顶点v到顶点u的路径；若v=u，则称该序列为回路。

有向图 D =（V，E）中的顶点序列 $v_1, v_2, \ldots, v_k$，若 $<v_i,v_{i+1}> \in E$（$i=1,2,\ldots k-1$)，$v=v_1$，$u=v_k$，则称该序列是从顶点v到顶点u的路径；若v=u，则称该序列为回路。

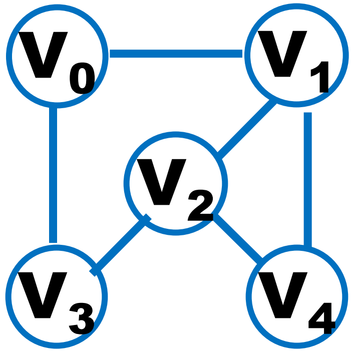
路径的边数称之为路径的长度。

6.1 图的定义与基本术语

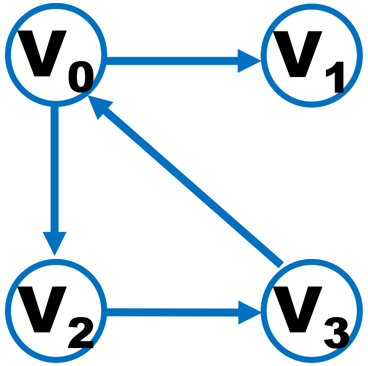
例如

在图G_1中，V_0, V_1, V_2, V_3 是 V_0 到 V_3 的路径；V_0, V_1, V_2, V_3, V_0 是回路。

在图G_2中，V_0, V_2, V_3 是 V_0 到 V_3 的路径； V_0, V_2, V_3, V_0 是回路。



无向图G_1



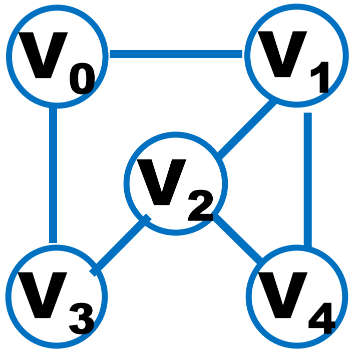
有向图G_2

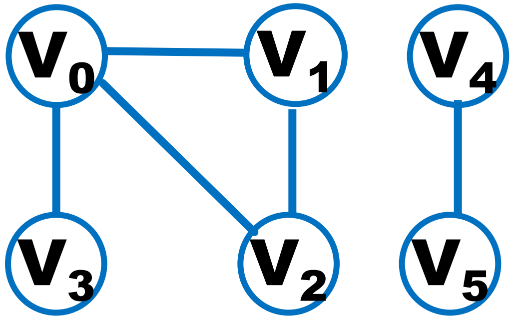
6.1 图的定义与基本术语

连通图（强连通图）

在无（有）向图 G=< V, E > 中，若对任何两个顶点 v、u 都存在从 v 到 u 的路径，则称G是连通图（强连通图）。

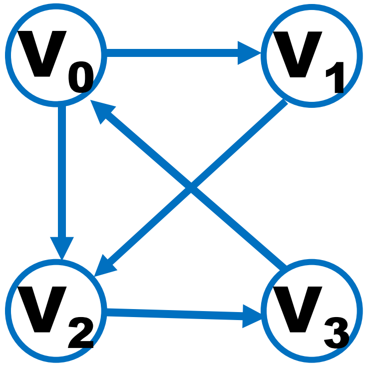
连通图

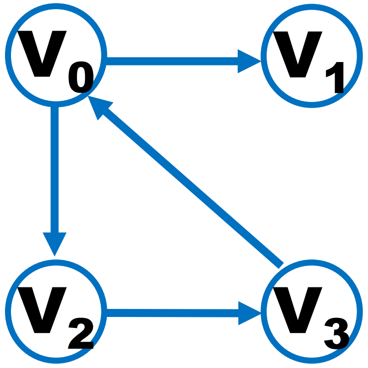




非连通图

强连通图





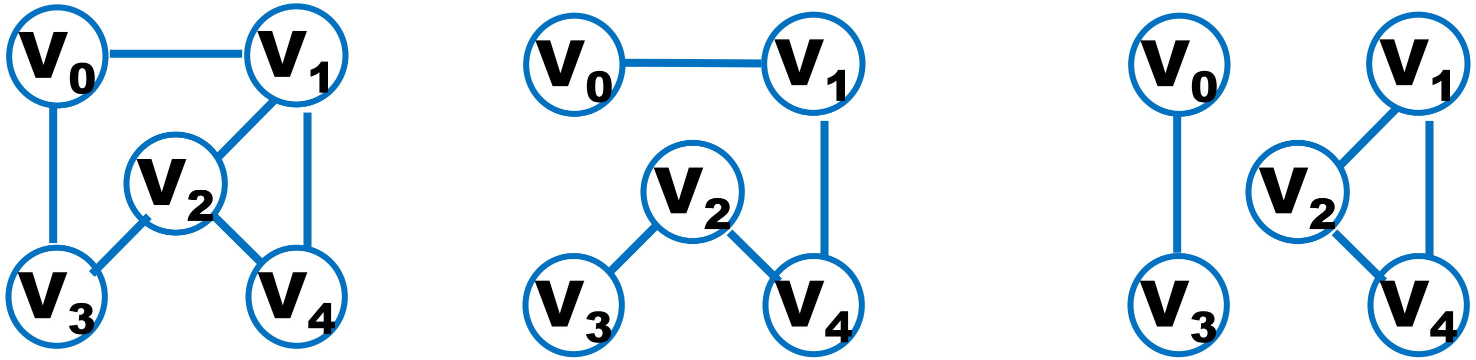
非强连通图

6.1 图的定义与基本术语

子图

设有两个图 G=(V，E)，G1=(V1，E1)，若 V1 ⊆ V，E1 ⊆ E，E1关联的顶点都在 V1 中，则称 G1 是 G 的子图。

例 (b)、(c) 是 (a) 的子图

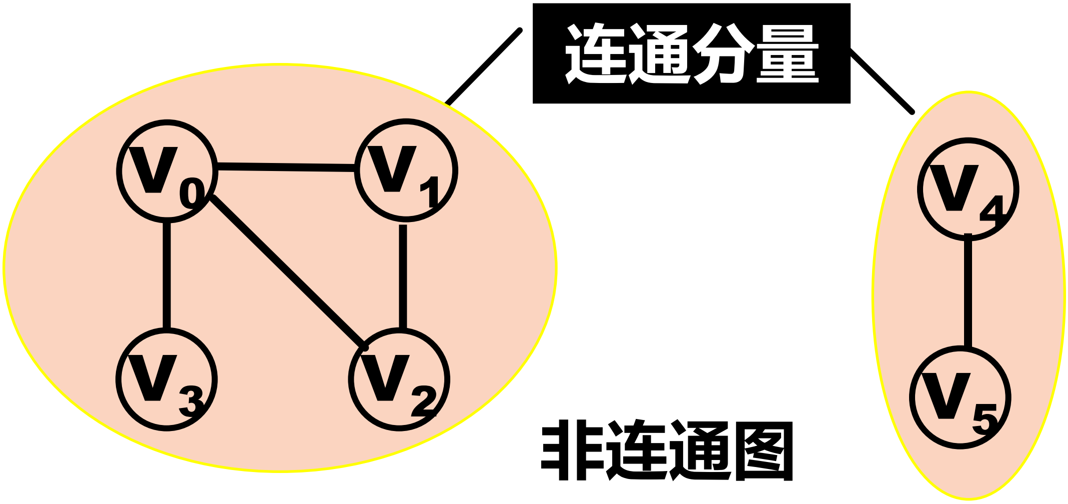


(a) (b) (c)

连通分量

无向图G的极大连通子图称为G的连通分量。任意两个顶点之间都有路径相通，则称此图为连通图(Connected graph)，

极大连通子图含义：该子图是G连通子图，将G的任何不在该子图中的顶点加入，子图不再连通。



连通分量

连通图G的连通分量，只有一个，就是G本身。

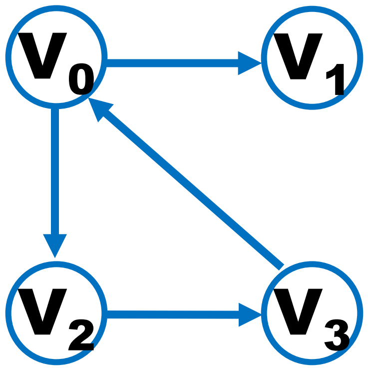
非连通图的连通分量，可以有多个。

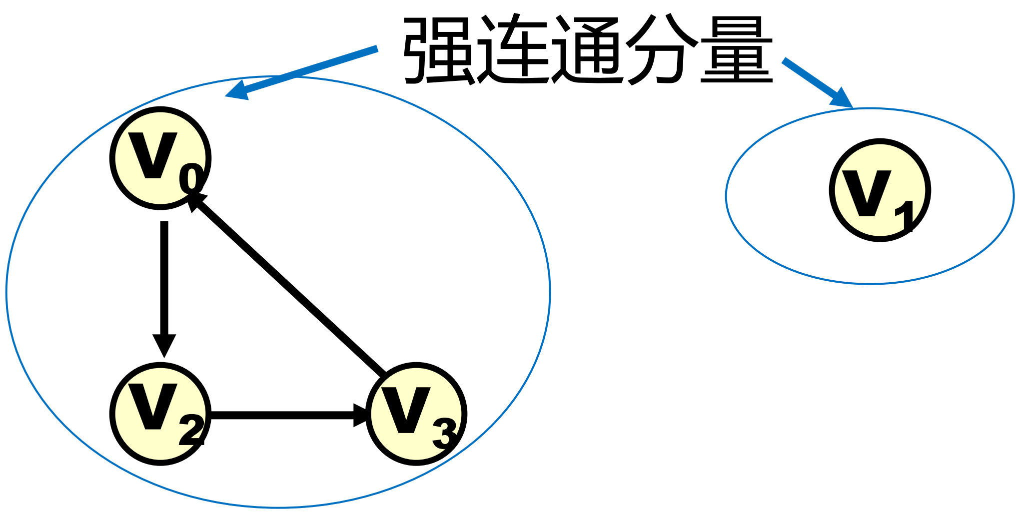
6.1 图的定义与基本术语

连通子图（强连通分量）若任意两个顶点之间都存在一条有向路径，则称此有向图为强连通图

有向图D的极大强连通子图称为D的强连通分量。

极大强连通子图含义：该子图是 D 的强连通子图，将 D 的任何不在该子图中的顶点加入，子图不再是强连通的。





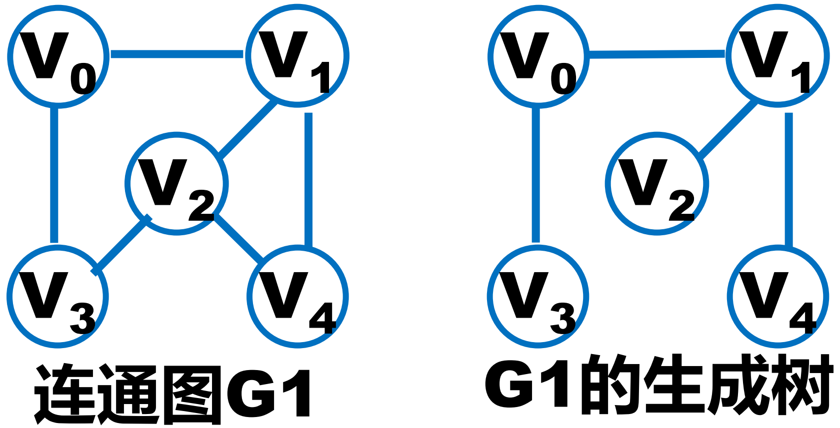
6.1 图的定义与基本术语

生成树

包含无向图 G 所有顶点的极小连通子图称为G生成树。

极小连通子图含义：该子图是G的连通子图，在该子图中删除任何一条边，子图不再连通，若T是G的生成树当且仅当T满足如下条件：

- T是G的连通子图
- T包含G的所有顶点
- T中无回路



6.1 图的定义与基本术语

图的基本操作

图的建立与销毁
对某个顶点的访问操作
对邻接点的操作
插入或删除顶点
插入或删除弧
遍历

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁：

CreateEmptyGraph(&G)：
创建一个空图
CreateGraph(&G,V,R)：
根据顶点集合V和边集合R，创建图
DestroyGraph(&G)：
给定图，删除所有顶点和边（清空）

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)

对某个顶点的访问操作：

LocateVex(G,v)：

给定图G，访问某顶点v，若不存在则返回空；否则返回地址

GetVex(G,v)：

给定图G，返回某顶点v的值，若顶点v不存在则报错；

PutVex(G,v，value)：

给定图G，为某顶点赋值value，若顶点v不存在则报错；

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)

对某个顶点的访问操作LocateVex(G,v),GetVex(G,v),PutVex(&G,v,value)

对邻接点的操作：

firstNeighbor(G,v)：

返回某个顶点v的第一个邻接点w；若不存在，则返回空；

nextNeighbor(G,v,w)：

返回某个顶点v的（相对于w的）下一个邻接点，若w已经时最后一个邻接点，则返回空；

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)

对某个顶点的访问操作LocateVex(G,v),GetVex(G,v),PutVex(&G,v,value)

对邻接点的操作firstNeighbor(G,v),nextNeighbor(G,v,r)

插入或删除顶点：

InsertVex(&G,v)：

在给定的图G中插入顶点v，若失败，则报错，若成功返回地址；

DeleteVex(&G,v)：

在给定的图G中删除顶点v，同时删除所有与v相邻的边

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)

对某个顶点的访问操作LocateVex(G,v),GetVex(G,v),PutVex(&G,v,value)

对邻接点的操作firstNeighbor(G,v),nextNeighbor(G,v,r)

插入或删除顶点InsertVex(&G,v),DeleteVex(&G,v)

插入或删除弧:

InsertArc(&G,v,w):

在给定图G中，在顶点v与顶点w之间，插入一条边，连接v和w

DeleteArc(&G,v,w):

在给定的图G中，删除顶点v和顶点w之间的边

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)

对某个顶点的访问操作LocateVex(G,v),GetVex(G,v),PutVex(&G,v,value)

对邻接点的操作firstNeighbor(G,v),nextNeighbor(G,v,r)

插入或删除顶点InsertVex(&G,v),DeleteVex(&G,v)

插入或删除弧InsertArc(&G,v,w),DeleteArc(&G,v,w)

遍历：

DFSTraverse(&G,v,Visit())：深度优先遍历

BFSTraverse(&G,v,Visit())：广度优先遍历

6.1 图的定义与基本术语

图的基本操作

图的建立与销毁CreateGraph(&G,V,R),DestroyGraph(&G)
对某个顶点的访问操作LocateVex(G,v),GetVex(G,v),PutVex(&G,v,value)
对邻接点的操作firstNeighbor(G,v),nextNeighbor(G,v,r)
插入或删除顶点InsertVex(&G,v),DeleteVex(&G,v)
插入或删除弧InsertArc(&G,v,w),DeleteArc(&G,v,w)
遍历DFSTraverse(&G,v,Visit()),BFSTraverse(&G,v,Visit())

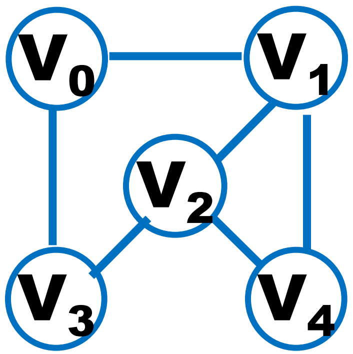
6.2 图的存储结构

一、邻接矩阵表示

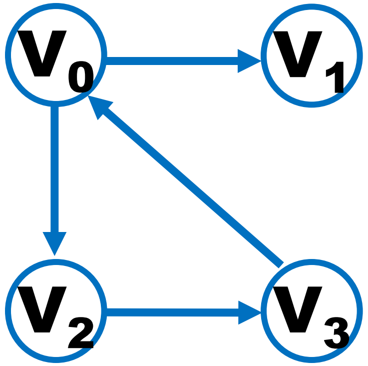
邻接矩阵： G的邻接矩阵是满足如下条件的n阶矩阵：

$$A[i][j]=\begin{cases}1 & 若(v_i,v_j)\subseteq E 或 <v_i,v_j>\subseteq E\\ 0 & 否则\end{cases}$$

在数组表示法中，用邻接矩阵表示顶点间的关系



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



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

6.2 图的存储结构

```
//邻接矩阵，可考虑分别存储两个矩阵，一个矩阵用于存顶点的信息
//另一个矩阵用于存弧的信息
#define INFINITY   65535
#define MAX_VERTEX_NUM 20

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

//弧矩阵
typedef struct ArcCell {
    VEType adj; //顶点关系类型，对于无权图，用0或1表示连接与否
                //对于带权图，则为其权重值
    InfoType *info; //该弧相关信息指针
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

//顶点信息矩阵
typedef struct {
    VertexType vexs[MAX_VERTEX_NUM];     //顶点向量
    AdjMatrix   arcs;                    //邻接矩阵
    int  vexnum, arcnum;                 //顶点数量和边的数量
    GraphKind kind;                      //图的种类标志
}MGraph;
```

6.2 图的存储结构

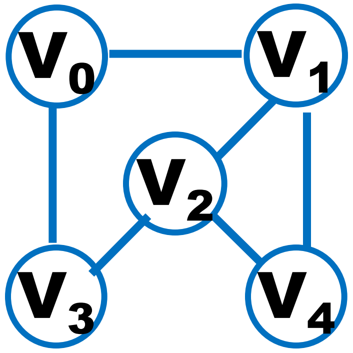
二、邻接表

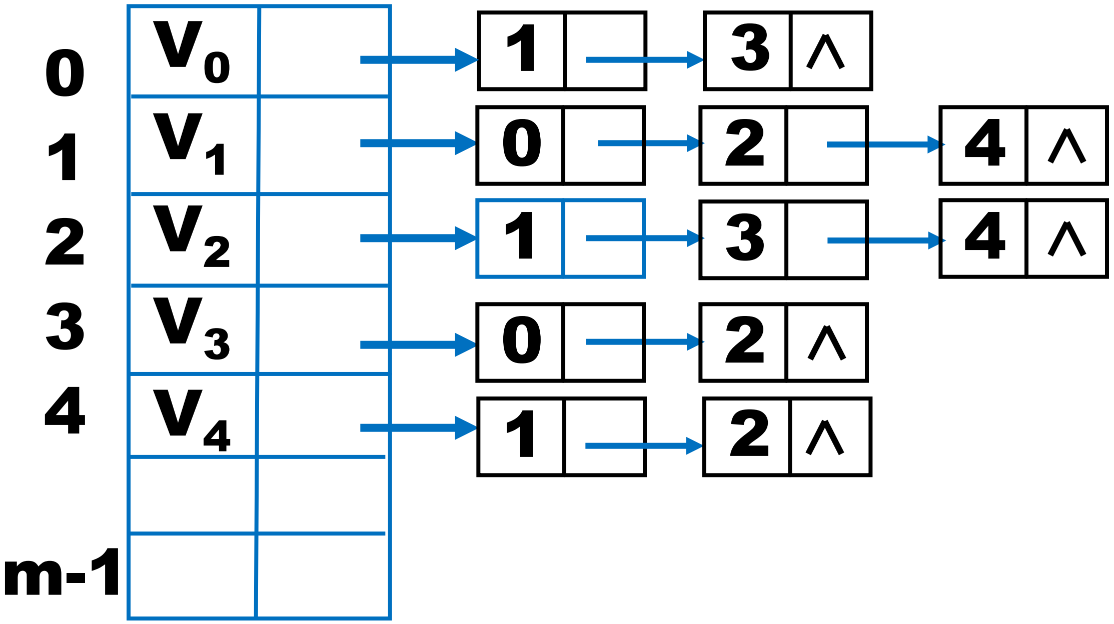
邻接表是图的链式存储结构

1、无向图的邻接表

顶点：通常按编号顺序将顶点数据存储在一维数组中；

关联同一顶点的边：用线性链表存储。





6.2 图的存储结构

adjvex	next

```
typedef struct ArcNode  // 结点定义
        {  int  adjvex;   // 邻接点域，存放与Vi邻接的点在表头数组中的位置
           struct  ArcNode  *next;  // 链域，下一条边或弧
        } ArcNode;
```

vexdata	firstarc

```
typedef struct tnode  // 表头结点
        {   int  vexdata;                  // 存放顶点信息
            ArcNode  * firstarc;        // 指向第一个邻接点
        } VNode, AdjList [ MAX_VERTEX_NUM ] ;

typedef struct
        {       AdjList  vertices;
                int          vexnum, arcnum;   // 顶点数和弧数
                int          kind;                    // 图的种类
        }
typedef enum {DG,DN,UDG,UDN} GraphKind
```

6.2 图的存储结构

无向图的邻接表的特点

1）在G邻接表中，同一条边对应两个结点；

2）顶点v的度：等于v对应线性链表的长度；

3）判定两顶点v，u是否邻接：要看v对应线性链表中有无对应的结点。

4）在G中增减边：要在两个单链表插入、删除结点；

5）设存储顶点的一维数组大小为 m（m≥图的顶点数n），图的边数为e，G占用存储空间为：m+2*e。G占用存储空间与G的顶点数、边数均有关；适用于边稀疏的图。

6.2 图的存储结构

	邻接矩阵	邻接表
firstNeighbor(G, v)	最坏O(n)	O(1)
nextNeighbor(G, v, w);	最坏O(n)	最坏O(e)
getWeight(G, v, w)	O(1)	最坏O(e)
printGraph(G)	$O(n^2)$	O(n+e)
空间效率	适合稠密图	适合稀疏图
时间效率	访问一条边 效率高	频繁访问邻接点 效率高

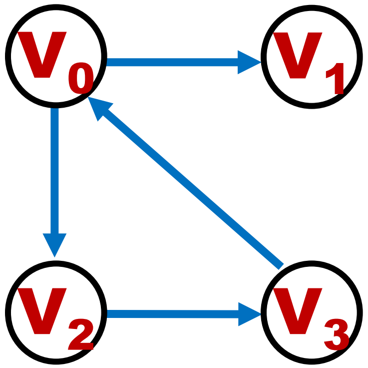
6.2 图的存储结构

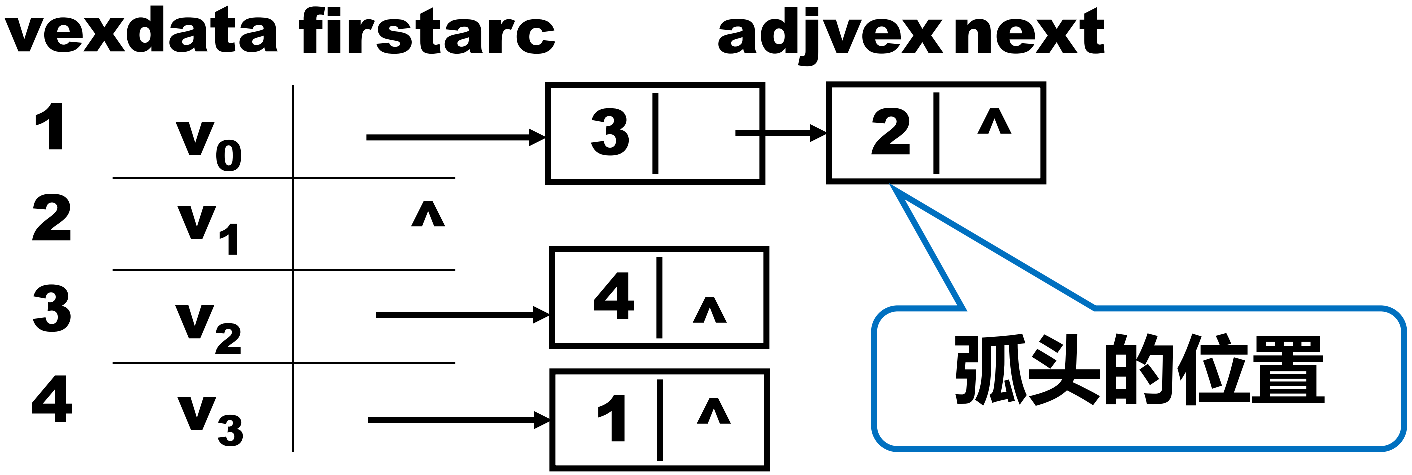
二、邻接表

2、有向图的邻接表

顶点：用一维数组存储（按编号顺序）

以同一顶点为起点的弧：用线性链表存储





类似于无向图的邻接表，所不同的是：

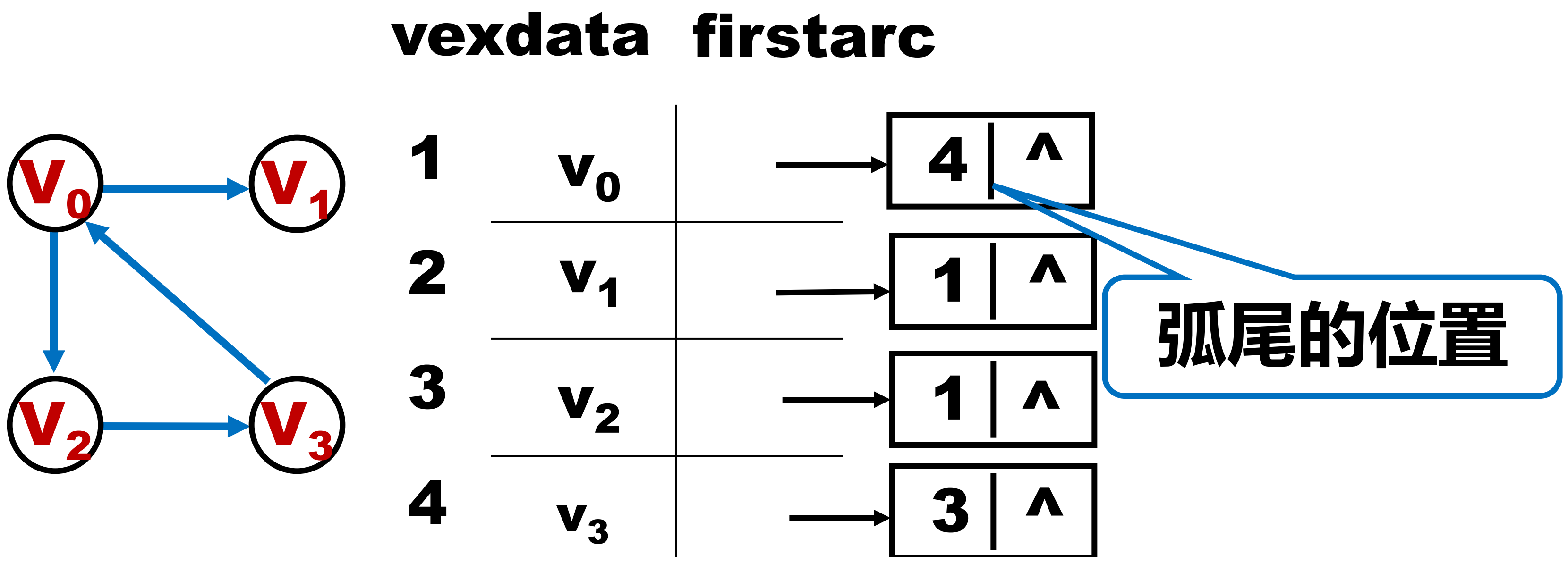
以同一顶点为**起点**的弧：用线性链表存储

二、邻接表

3、有向图的逆邻接表

顶点：用一维数组存储（按编号顺序）

以同一顶点为终点的弧：用线性链表存储。



类似于有向图的邻接表，所不同的是：

以同一顶点为终点弧：用线性链表存储

三、有向图的十字链表表示法

弧结点：

tailvex	headvex	hlink	tlink

```
typedef  struct  ArcBox
{   int   tailvex,  headvex;   // 弧尾、弧头在表头数组中位置
      struct   arcnode   *hlink;    // 指向弧头相同的下一条弧
      struct   arcnode   *tlink;     // 指向弧尾相同的下一条弧
} ArcBox;
```

顶点结点：

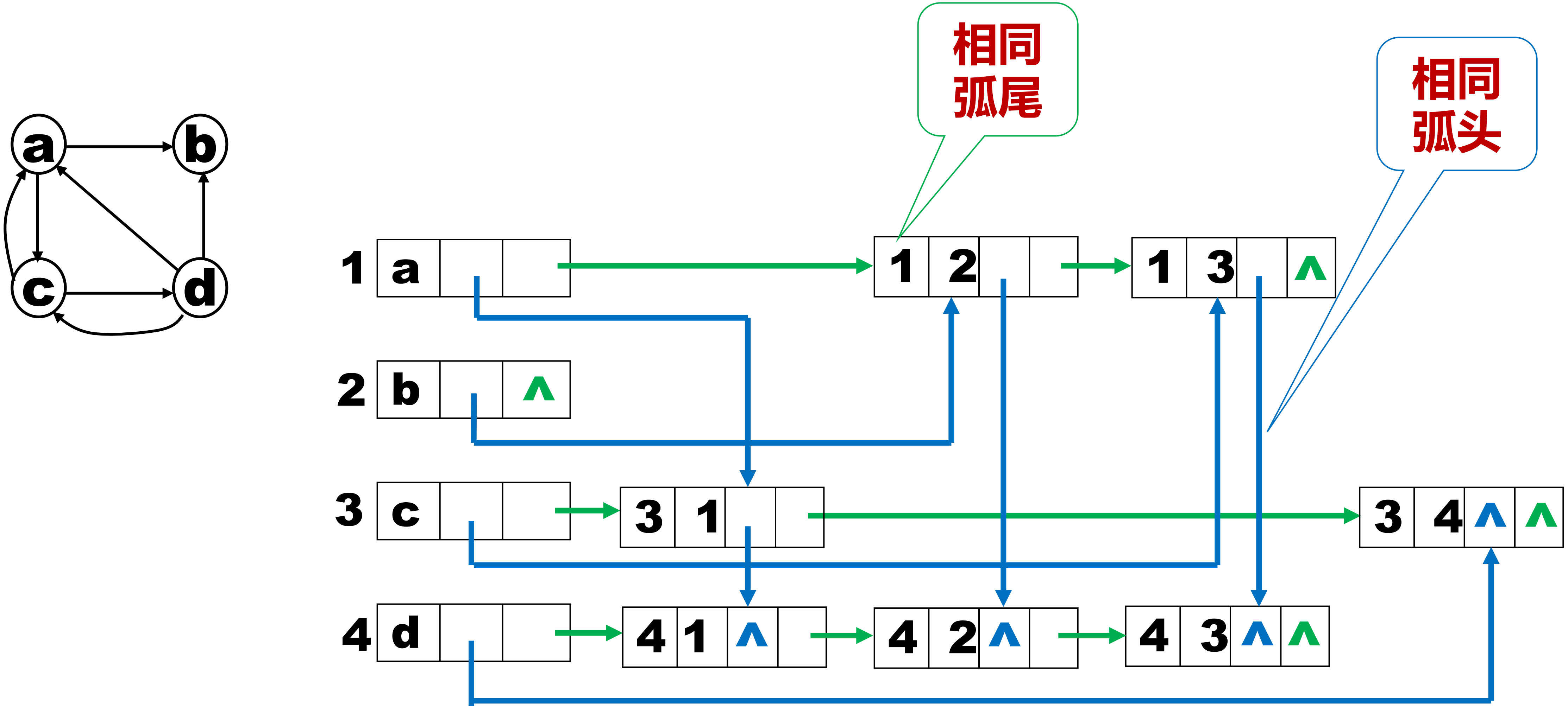
data	firstin	firstout

```
typedef  struct  VexNode
{  VertexType   data; // 存与顶点有关信息
    ArcBox *firstin;    // 指向以该顶点为弧头的第1个弧结点
    ArcBox *firstout;  // 指向以该顶点为弧尾的第1个弧结点
} VexNode;
VexNode  OLGraph[M];
```

6.2 图的存储结构

三、有向图的十字链表表示法

例



6.2 图的存储结构

四、无向图的邻接多重表表示法

边结点：

mark	**ivex**	**ilink**	**jvex**	**jlink**

```
typedef   struct  node
{   VisitIf  mark;   // 标志域，记录是否已经搜索过
    int  ivex, jvex;  // 该边依附的两个顶点在表头数组中位置
    struct  EBox   * ilink,  * jlink;
                             //分别指向依附于ivex和jvex的下一条边
} EBox;
```

顶点结点：

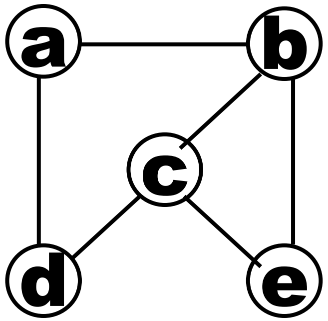
data	**firstedge**

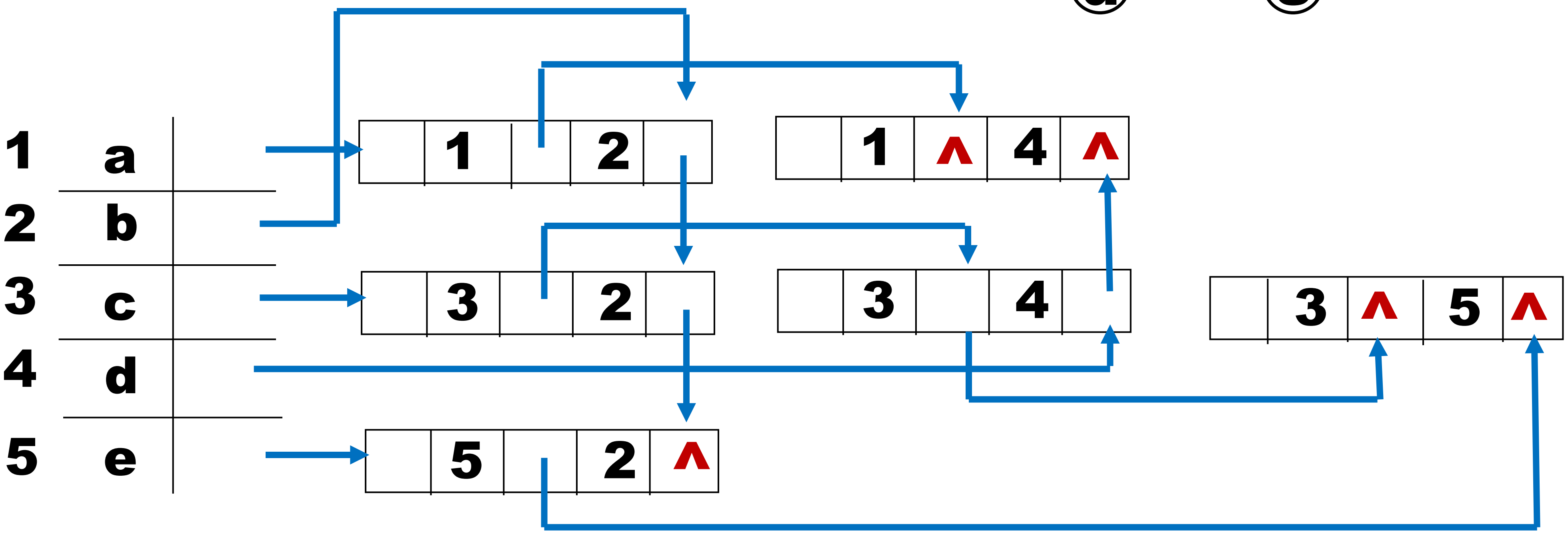
```
typedef   struct  VexBox
{    VertexType   data;          // 存与顶点有关的信息
     EBox  * firstedge;          // 指向第一条依附于该顶点的边
} VexBox;
VexBox  AMLGraph[M];
```

6.2 图的存储结构

四、无向图的邻接多重表表示法

例





6.3 图的遍历

图的遍历

访遍图中所有的顶点，并且使图中的每个顶点仅被访问一次。

遍历实质

找每个顶点的邻接点。

搜索路径

深度优先遍历（DFS）

广度优先遍历（BFS）

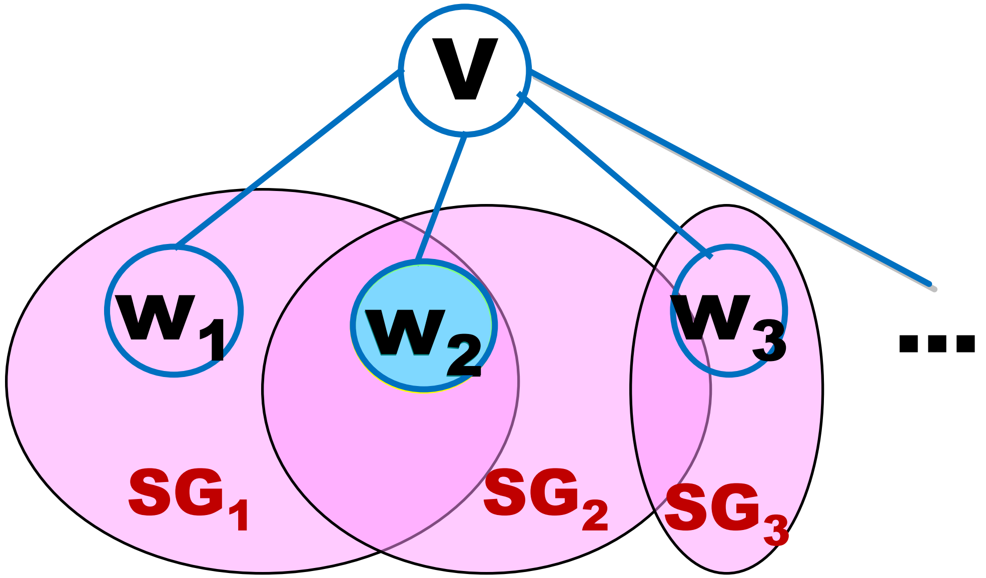
图的深度遍历（DFS）

从图的某顶点v出发，进行深度优先遍历

访问顶点 V；

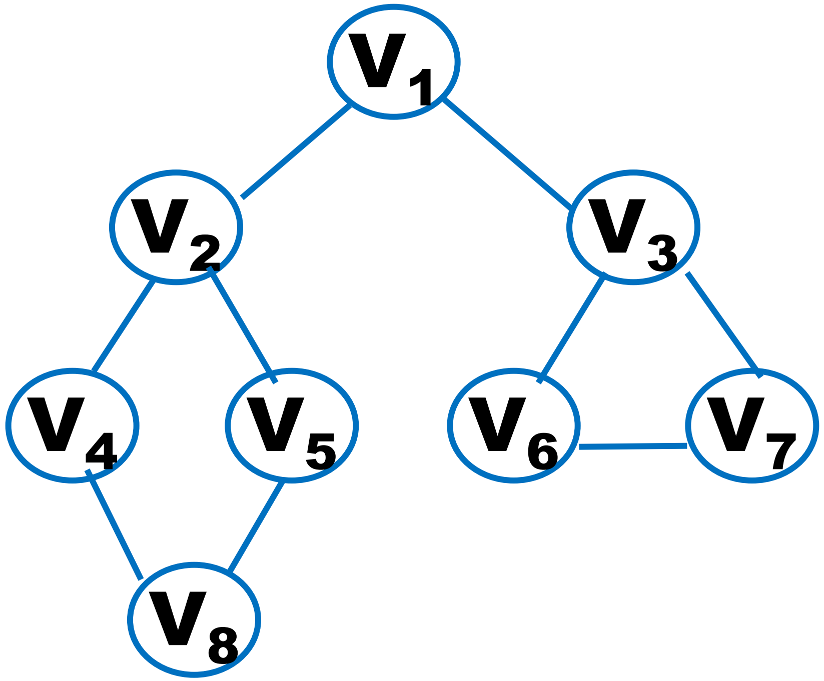
for (V的所有邻接点W_1、W_2、W_3...)

若 W_i 未被访问，则从 W_i 出发，进行深度优先遍历。



图的深度遍历（**DFS**）

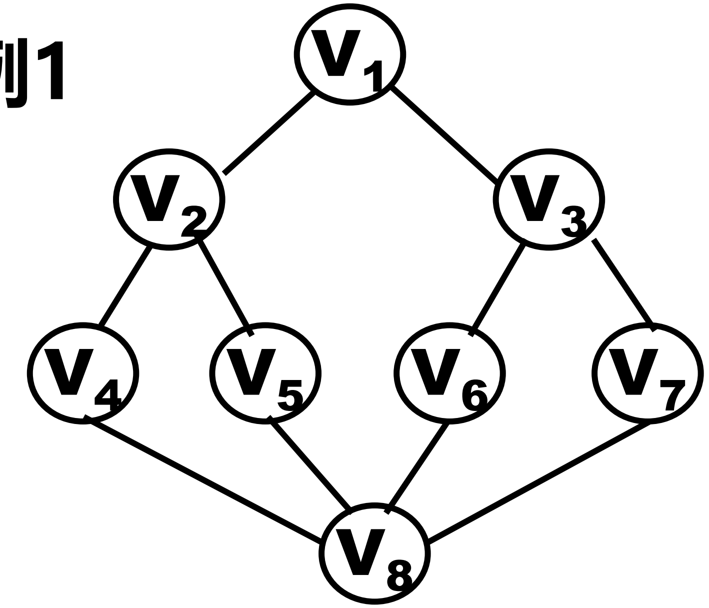
例：



深度遍历： $V_1 \Rightarrow V_2 \Rightarrow V_4 \Rightarrow V_8 \Rightarrow V_5 \Rightarrow V_3 \Rightarrow V_6 \Rightarrow V_7$

图的深度遍历（DFS）

例1



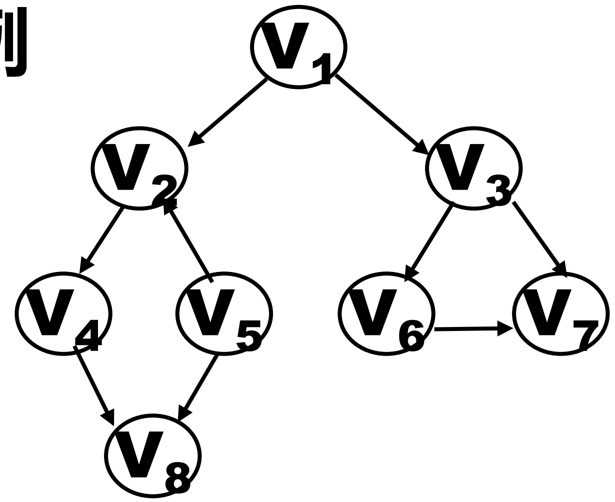
深度遍历1：$V_1 \Rightarrow V_2 \Rightarrow V_4 \Rightarrow V_8 \Rightarrow V_5 \Rightarrow V_6 \Rightarrow V_3 \Rightarrow V_7$

深度遍历2：$V_1 \Rightarrow V_3 \Rightarrow V_7 \Rightarrow V_8 \Rightarrow V_6 \Rightarrow V_5 \Rightarrow V_2 \Rightarrow V_4$

由于**没有规定访问邻接点的顺序**，所以深度优先序列不惟一。

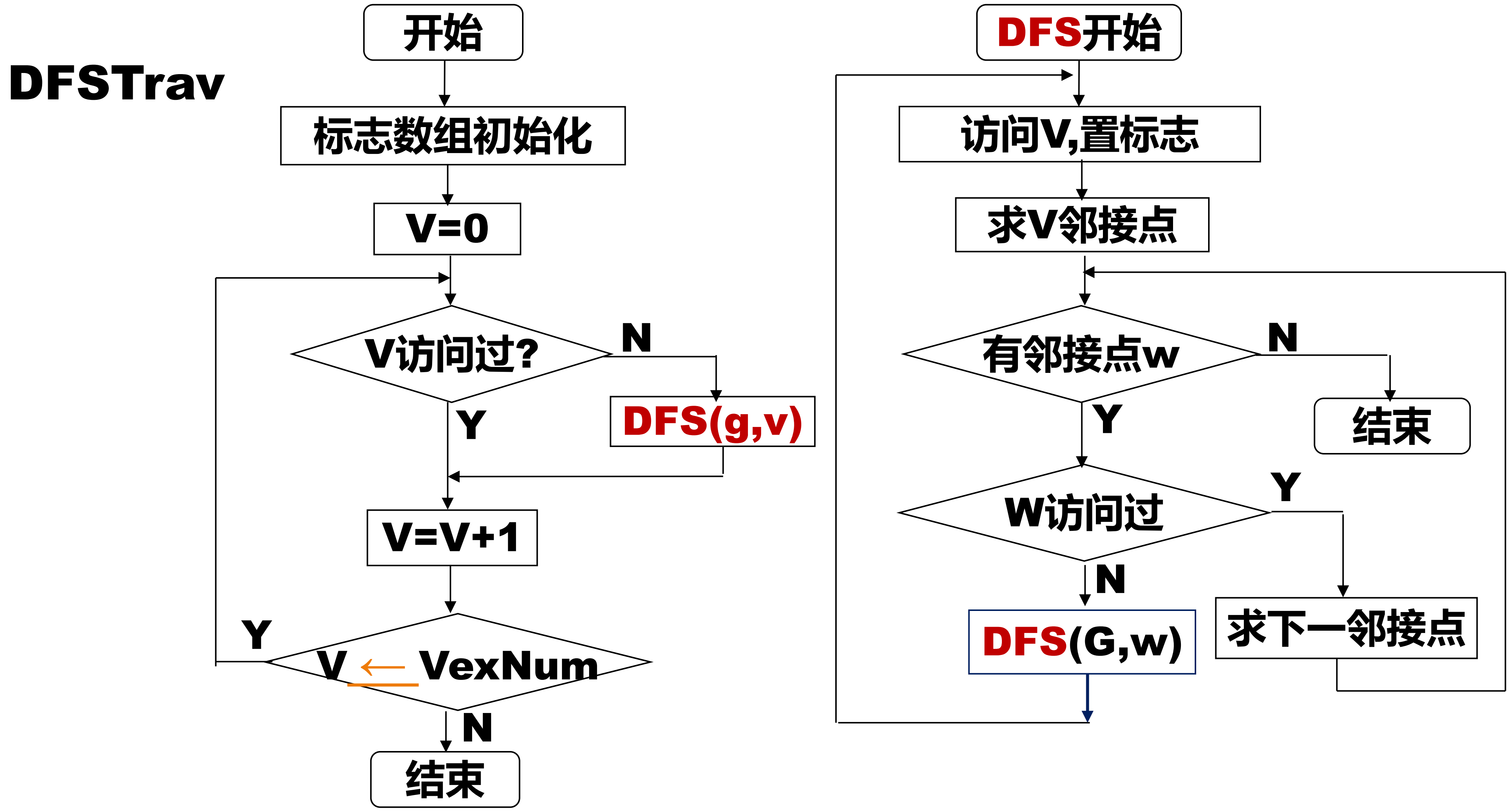
图的深度遍历（**DFS**）

例



深度遍历： $V_1 \Rightarrow V_2 \Rightarrow V_4 \Rightarrow V_8 \Rightarrow V_3 \Rightarrow V_6 \Rightarrow V_7 \Rightarrow V_5$

图的深度遍历（DFS）——算法6.4和6.5



图的深度遍历（DFS）——递归算法

```
void DFSTrav ( Graph G, Void ( * Visit ) ( VertexType e ) )
{
   for ( v=0; v< G.vexnum; ++v )
         visited[v] = FALSE;
   for ( v=0; v<G.vexnum; ++v )
         if ( ! visited[ v ] )
             DFS( G, v, Visit );
} //DFSTrav
```

图的深度遍历（DFS）——递归算法

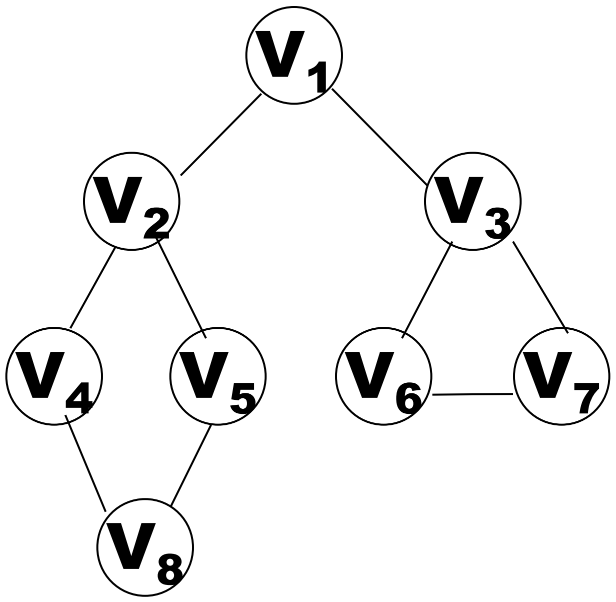
```
void DFS( Graph G, int v, void ( * Visit ) ( VertexType e )， int visited[] )
{  // 从v出发（v是顶点位置），深度优先遍历v所在的连通分量
   Visit( v );     //先根遍历
   visited[v] = TRUE;
   for ( w = FirstAdjVex( G, v ); w; w = NextAdjVex( G, v, w ) )
         if ( ! visited[ w ] )
               DFS( G, w, Visit( w ) ,visited);
} //DFS
```

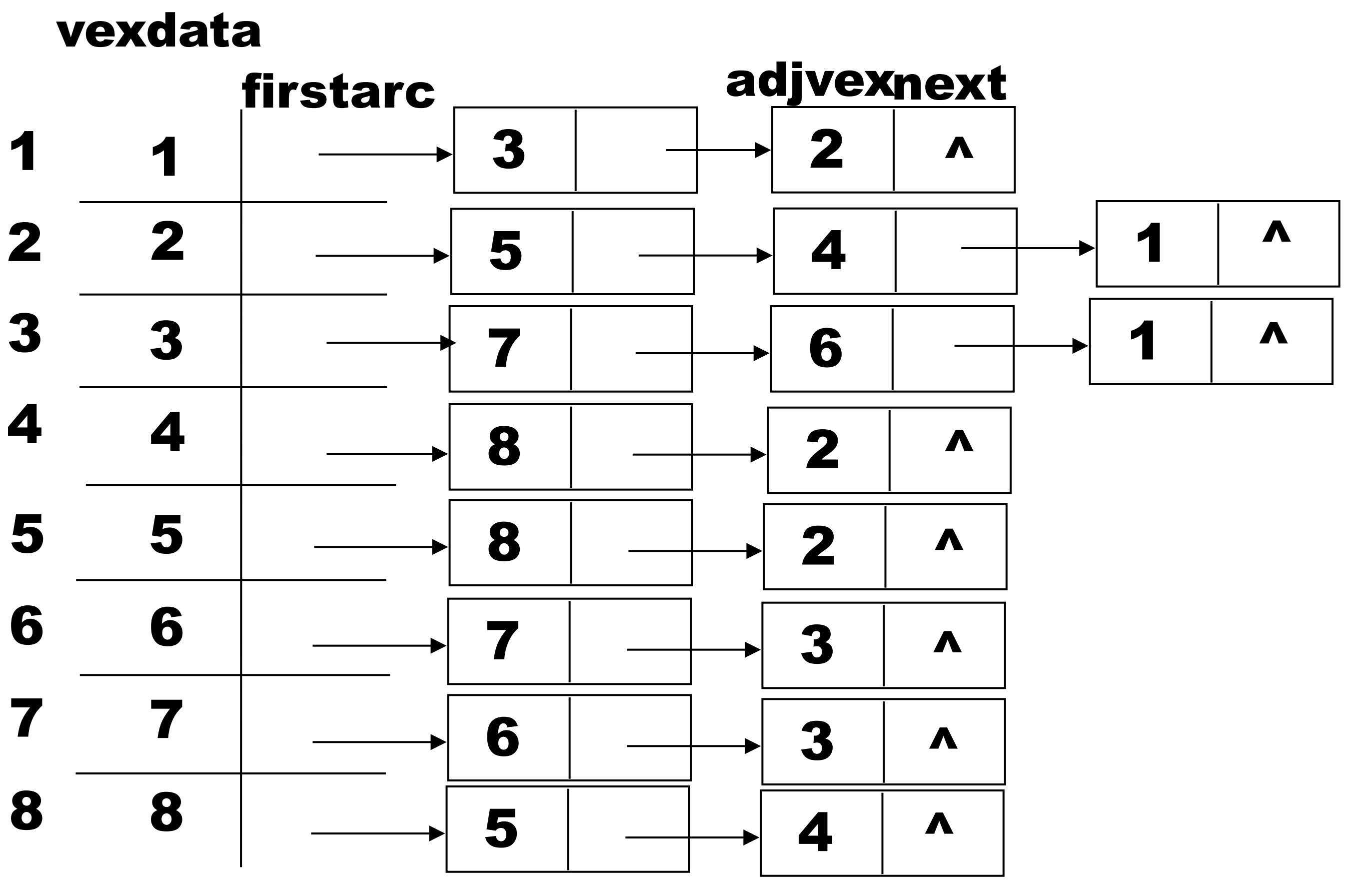
访问标志数组：int visited[ ] 初始时所有分量全为FALSE

6.3 图的遍历

图的深度遍历（DFS）——递归算法

例



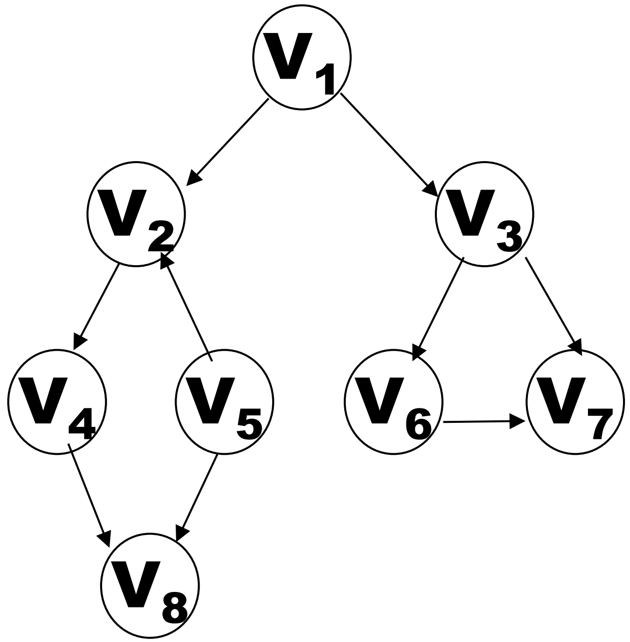


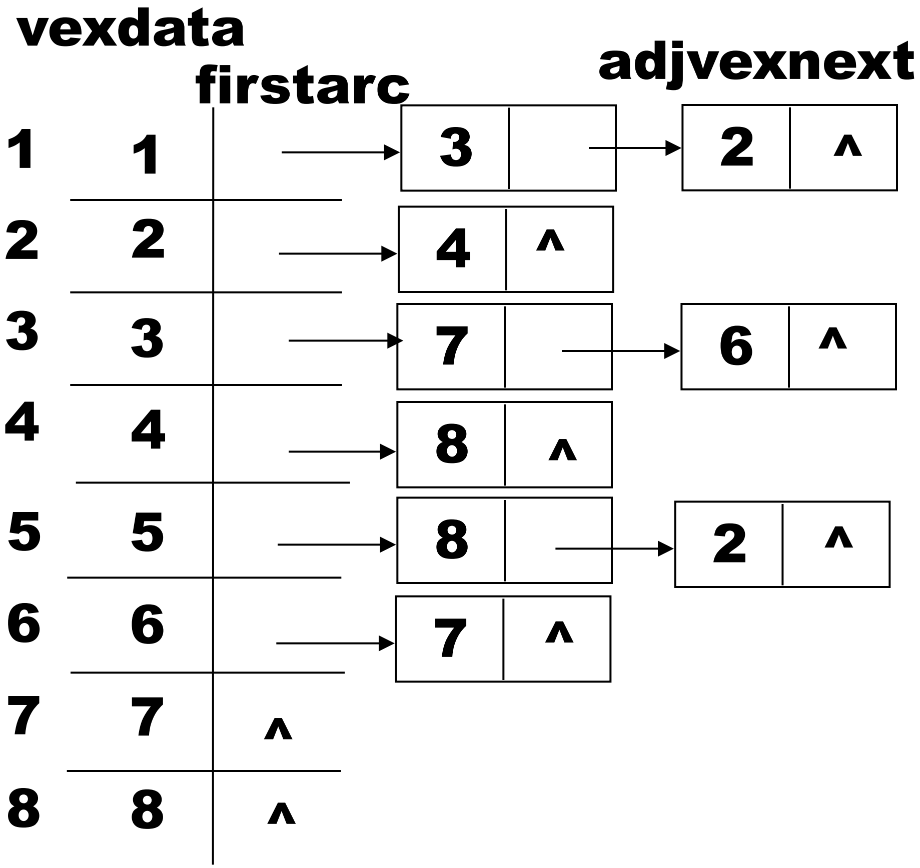
深度遍历： $V_1 \Rightarrow V_3 \Rightarrow V_7 \Rightarrow V_6 \Rightarrow V_2 \Rightarrow V_5 \Rightarrow V_8 \Rightarrow V_4$

6.3 图的遍历

图的深度遍历（DFS）——递归算法

例





深度遍历：$V_1 \Rightarrow V_3 \Rightarrow V_7 \Rightarrow V_6 \Rightarrow V_2 \Rightarrow V_4 \Rightarrow V_8 \Rightarrow V_5$

深度优先遍历的时间复杂度

DFS对每一条边处理一次，每个顶点访问一次。

邻接表表示总代价为：O(点数n + 边数e)

邻接矩阵表示：处理所有的边需要 $O(n^2)$ 的时间，所以总代价为 $O(n+n^2)= O(n^2)$。

6.3 图的遍历

图的广度遍历（BFS）

从图中某顶点v出发：

1)访问顶点v；

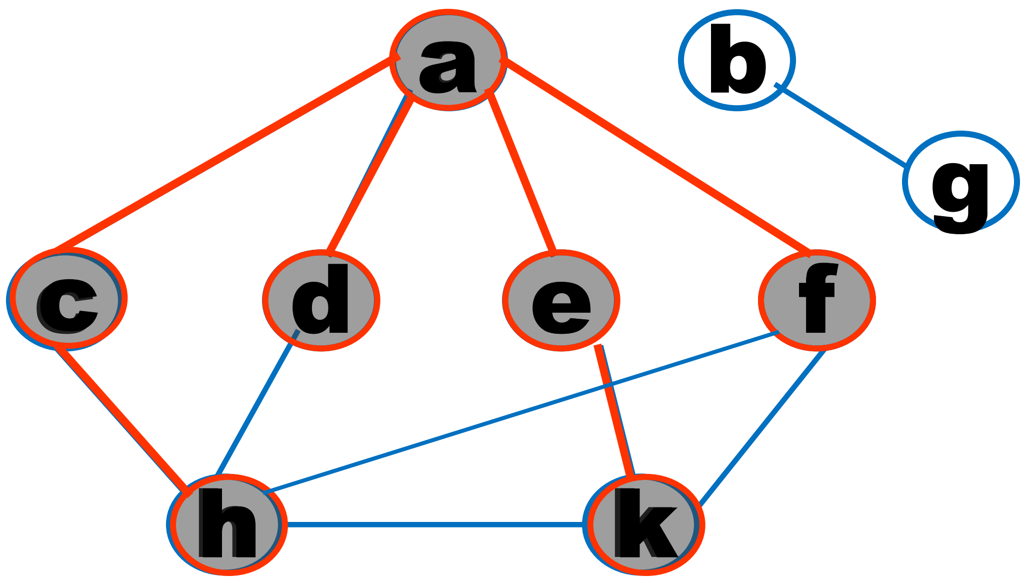
2)访问v所有未被访问的邻接点w1,w2,...wk；

3)依次从这些邻接点出发，访问其所有未被访问的邻接点。依此类推，直至图中所有和V0有路径相通的顶点都被访问到。

6.3 图的遍历

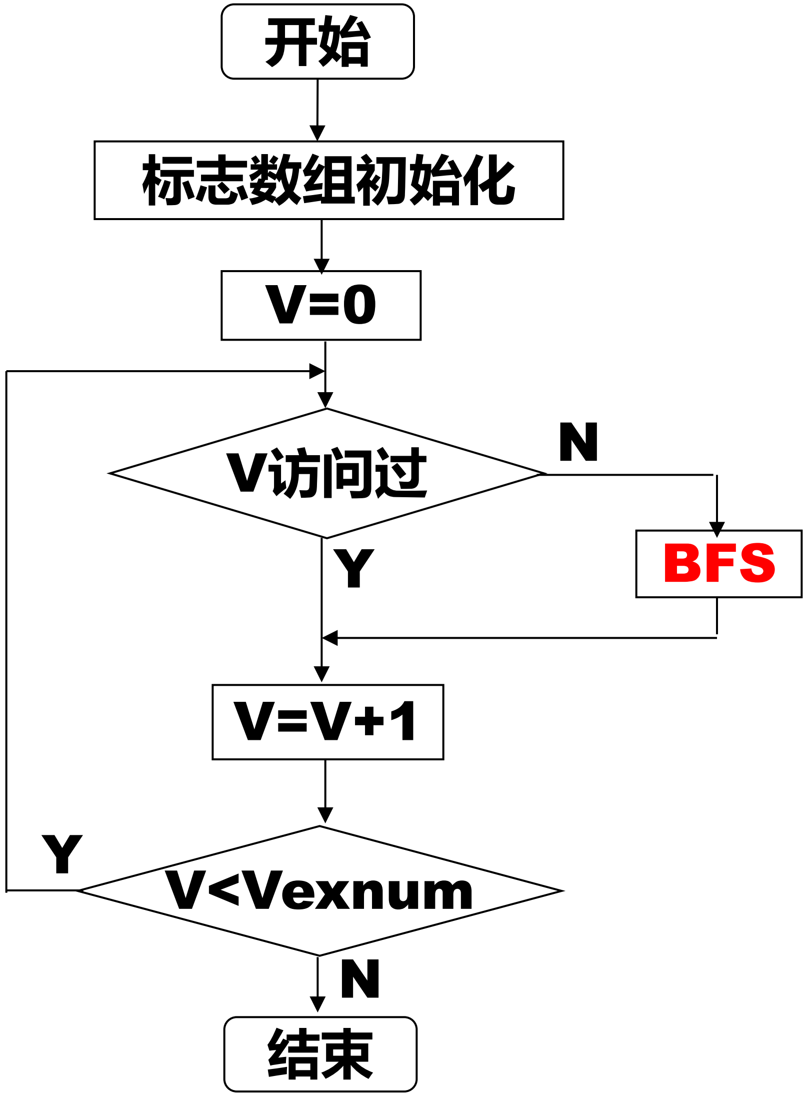
图的广度遍历（BFS）

例：



访问次序　a c d e f h k

图的广度遍历（BFS）

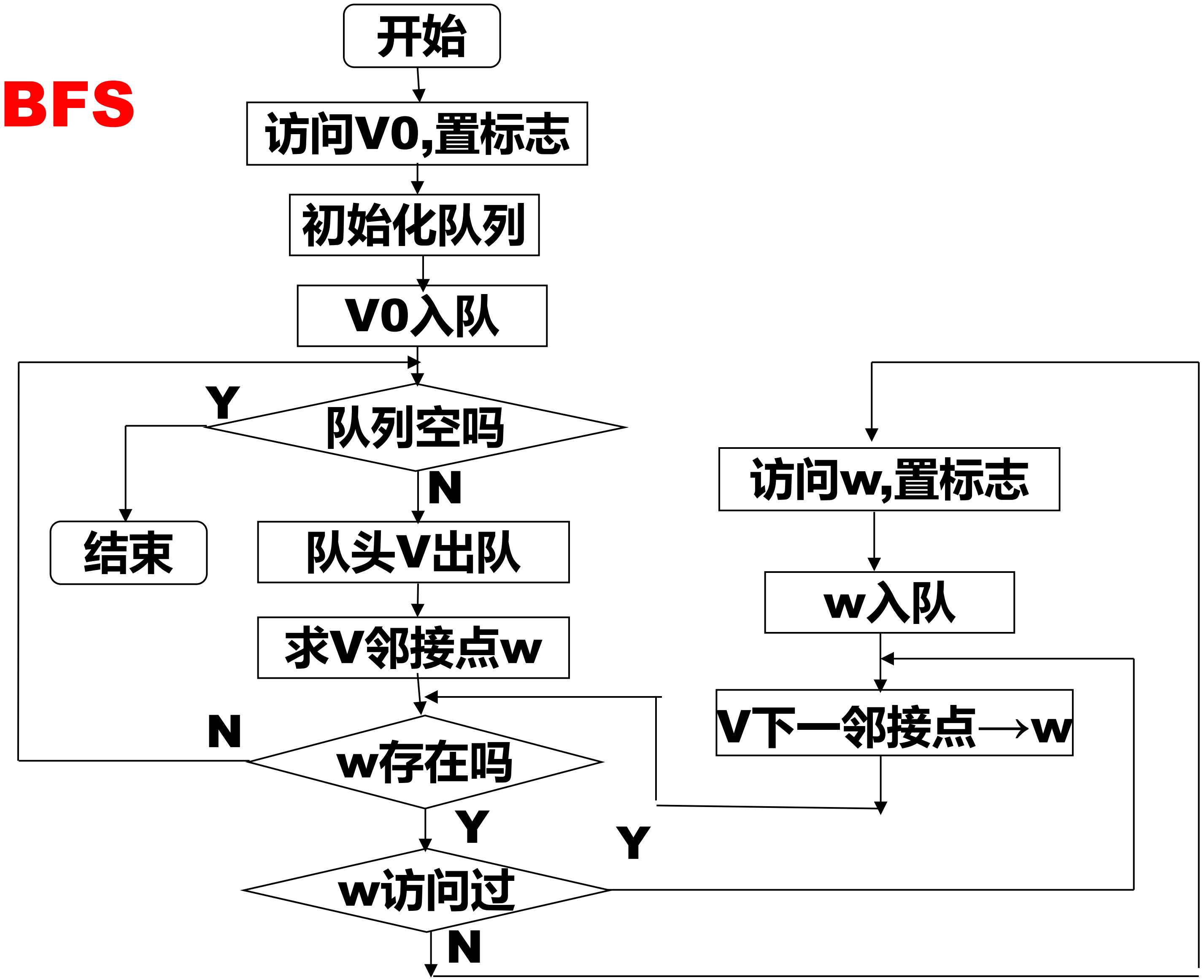


图的广度遍历（BFS）

```
void BFSTraverse ( Graph G, void (* Visit) ( VertexType ) )
{
   //本算法对图G进行广度优先遍历
   for ( v=0; v<G.vexnum; ++v )
        visited[v] = FALSE; // 访问标志数组初始化
   for ( v=0; v<G.vexnum; ++v )
        if ( ! visited[v] )
            BFS( G, v, Visit );
}  //BFSTraverse
```

图的广度遍历（BFS）——算法6.6

BFS

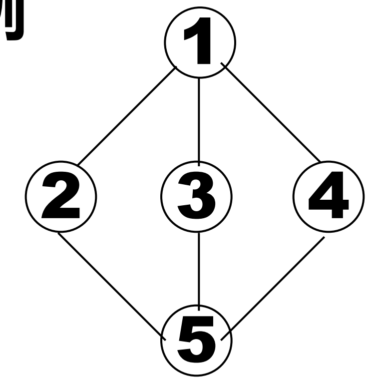


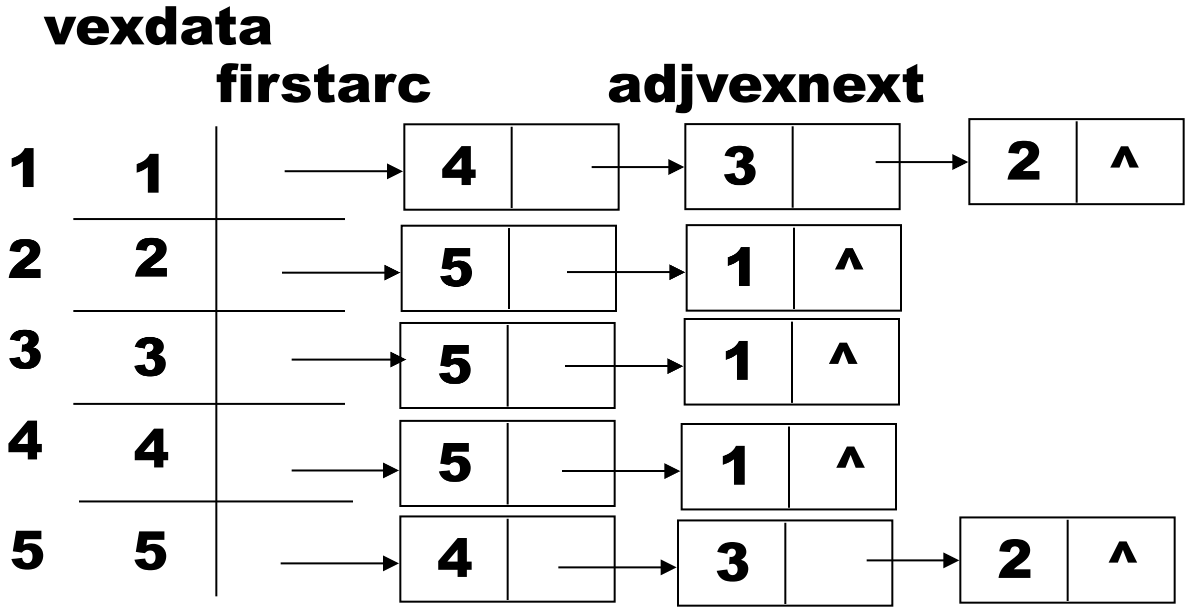
6.3 图的遍历

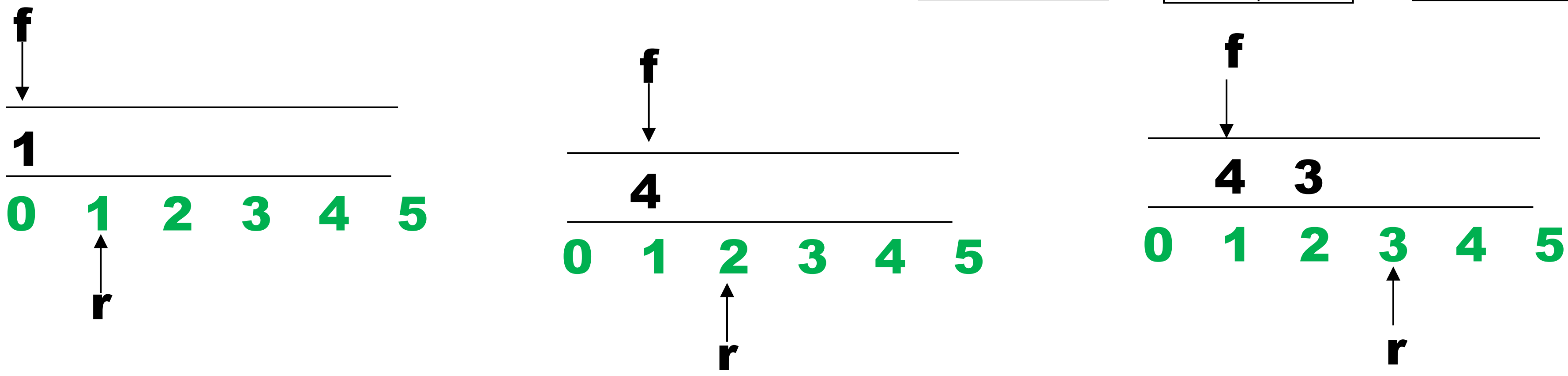
```
void BFS( Graph G, int v, void(* Visit) (VertexType e) ,int visited[])
{ // 从第v个顶点出发
   InitQueue(Q);  // 建立辅助空队列Q
   Visit(v);   visited[v]=TRUE;   // 访问u，访问标志数组
   EnQueue(Q,v);    // v入队
   while ( ! QueueEmpty( Q ) )
   { DeQueue(Q,u);   // 队头元素出队，并赋值给u
     for ( w=FirstAdjVex(G,u); w; w=NextAdjVex(G,u,w) )
          if ( ! visited[w] )
          {   Visit(w);
              visited[w]=TRUE;    // 访问u
              EnQueue(Q,w);
          }
  } //while
}  //BFS
```

图的广度遍历（BFS）

例

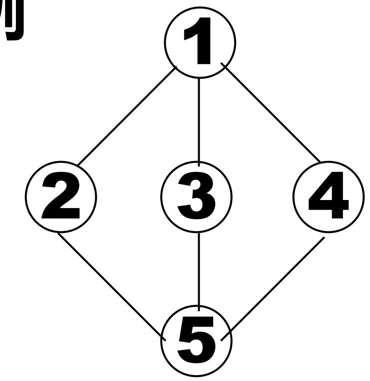

1
2
3
4
5

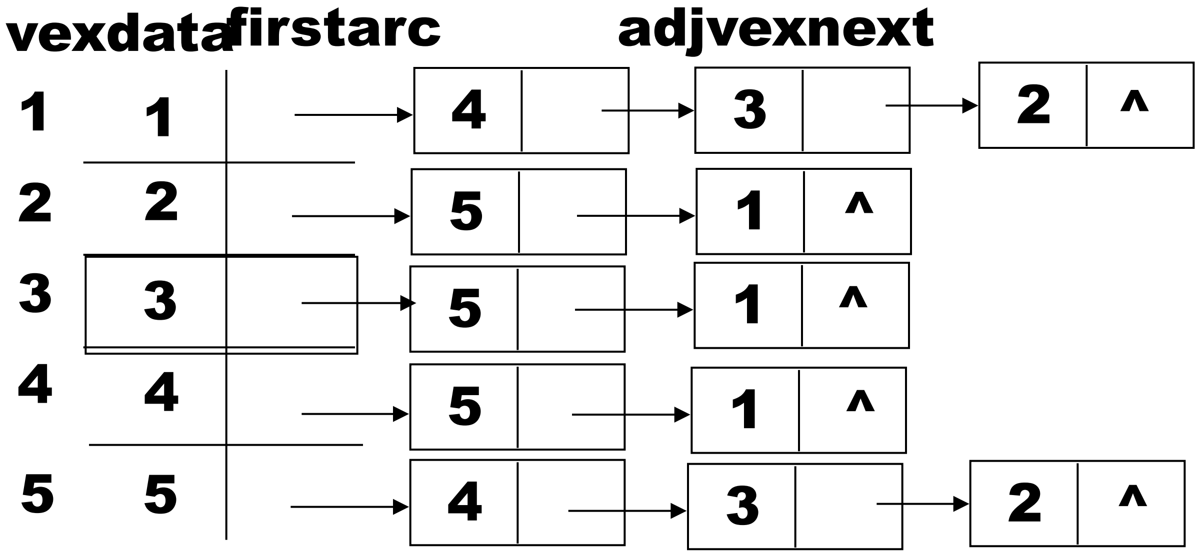

vexdata
firstarc
adjvex next
1 1 → 4 → 3 → 2 ^
2 2 → 5 → 1 ^
3 3 → 5 → 1 ^
4 4 → 5 → 1 ^
5 5 → 4 → 3 → 2 ^

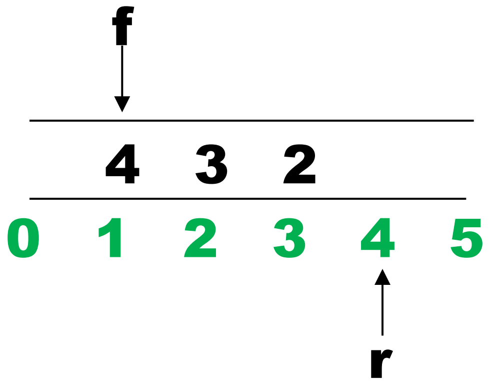

f
1
0 1 2 3 4 5
r
f
4
0 1 2 3 4 5
r
f
4 3
0 1 2 3 4 5
r

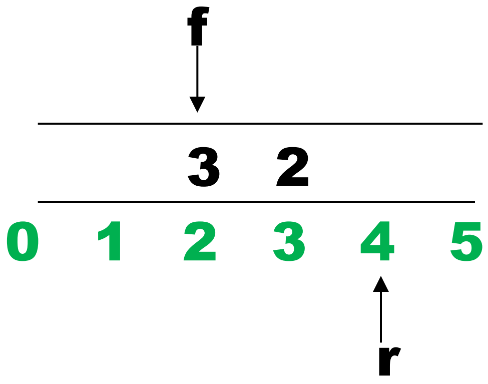
图的广度遍历（BFS）

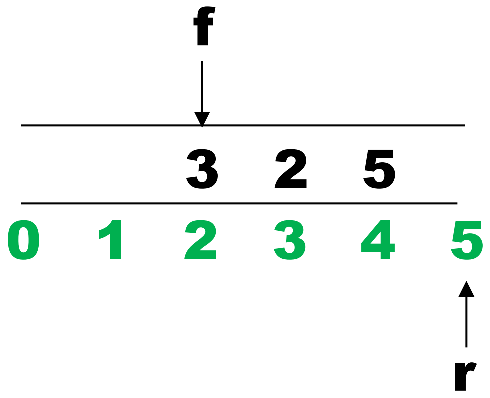
例


1
2
3
4
5


vexdata firstarc
adjvex next
1 1 → 4 → 3 → 2 ^
2 2 → 5 → 1 ^
3 3 → 5 → 1 ^
4 4 → 5 → 1 ^
5 5 → 4 → 3 → 2 ^

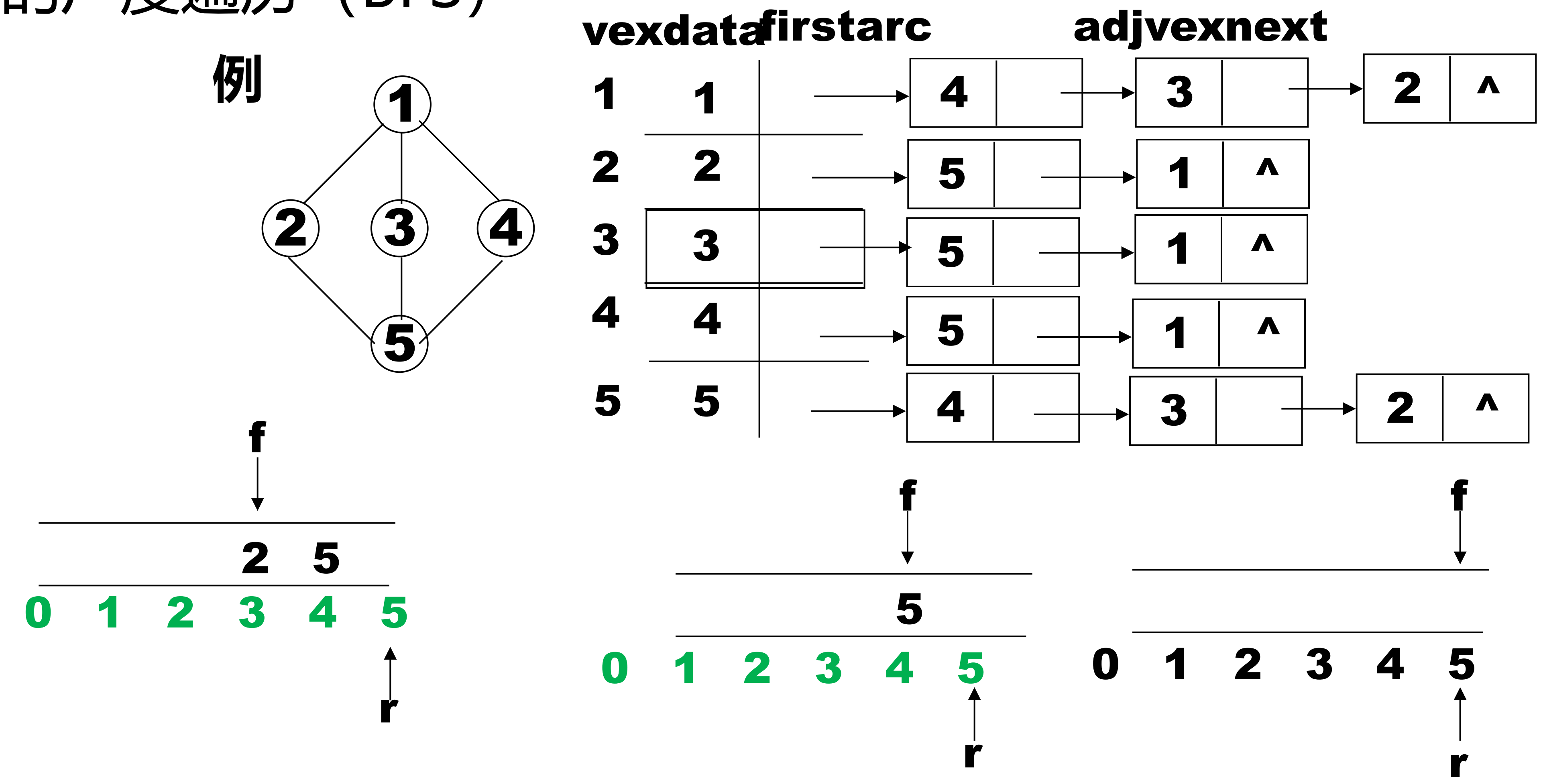

f
4 3 2
0 1 2 3 4 5
r


f
3 2
0 1 2 3 4 5
r


f
3 2 5
0 1 2 3 4 5
r

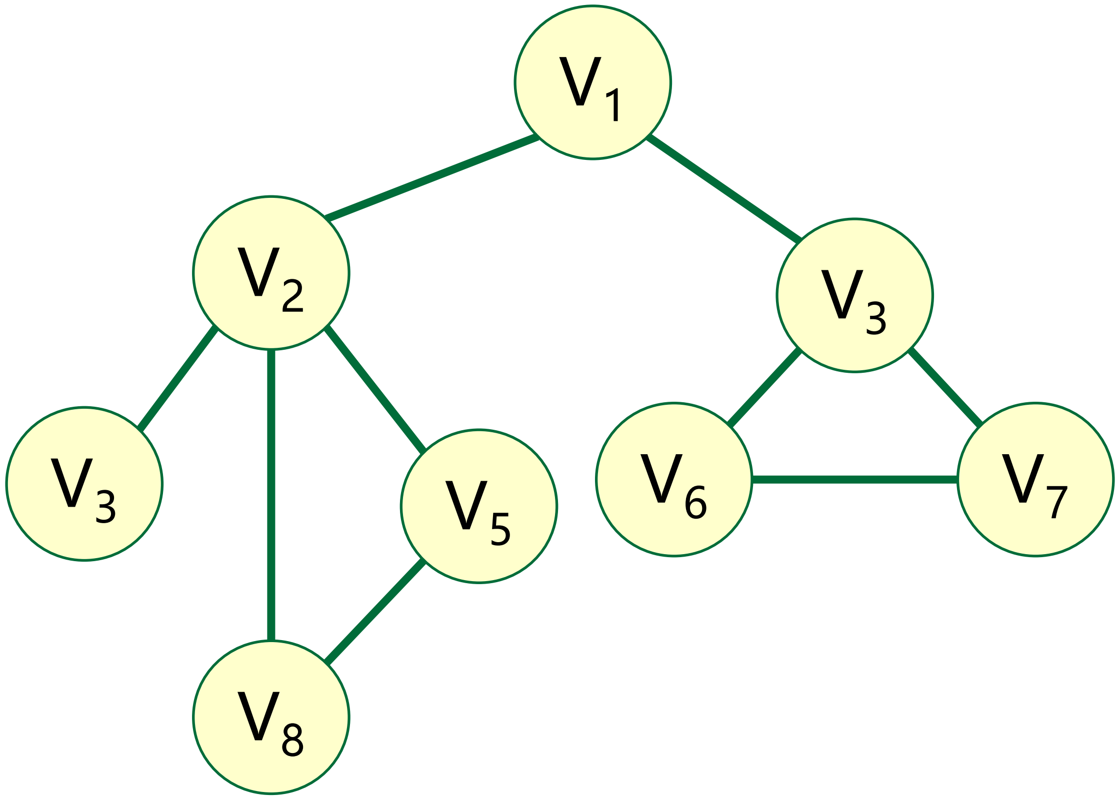
6.3 图的遍历

图的广度遍历（BFS）

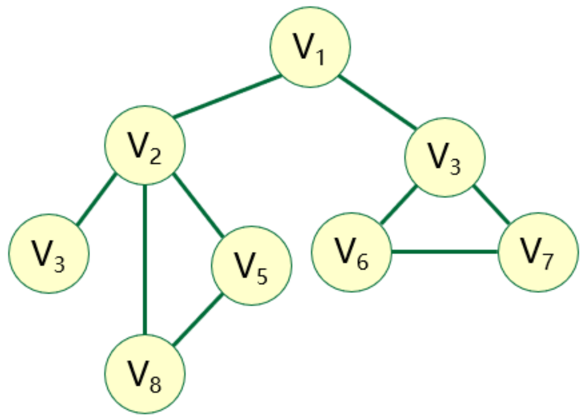


遍历的应用举例1：

求一条从顶点v到顶点s的简单路径



6.3 图的遍历

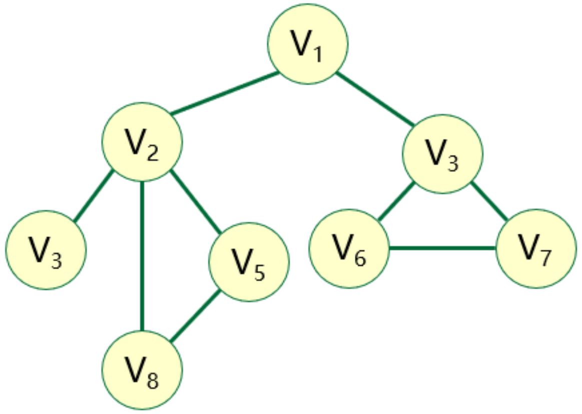


遍历的应用举例1：

求一条从顶点**v**到顶点**s**的简单路径

```
Status DFSearch(Graph G, VertexType v, VertexType s，, SqList &PATH) {
        //从v开始深度优先搜索，找到s为止
        v1 = LocateVex(G, v);     //找到v
        if ( v1 == -1) return FALSE;
        for (i=0; i<G.numVertices; ++i)    visited[i] = FALSE; // 访问标志数组初始化
        InitList_Sq(PATH);
        return _DFSearch(G, v1, s, PATH);    //从v开始深度优先搜索
}// DFSearch
```

6.3 图的遍历



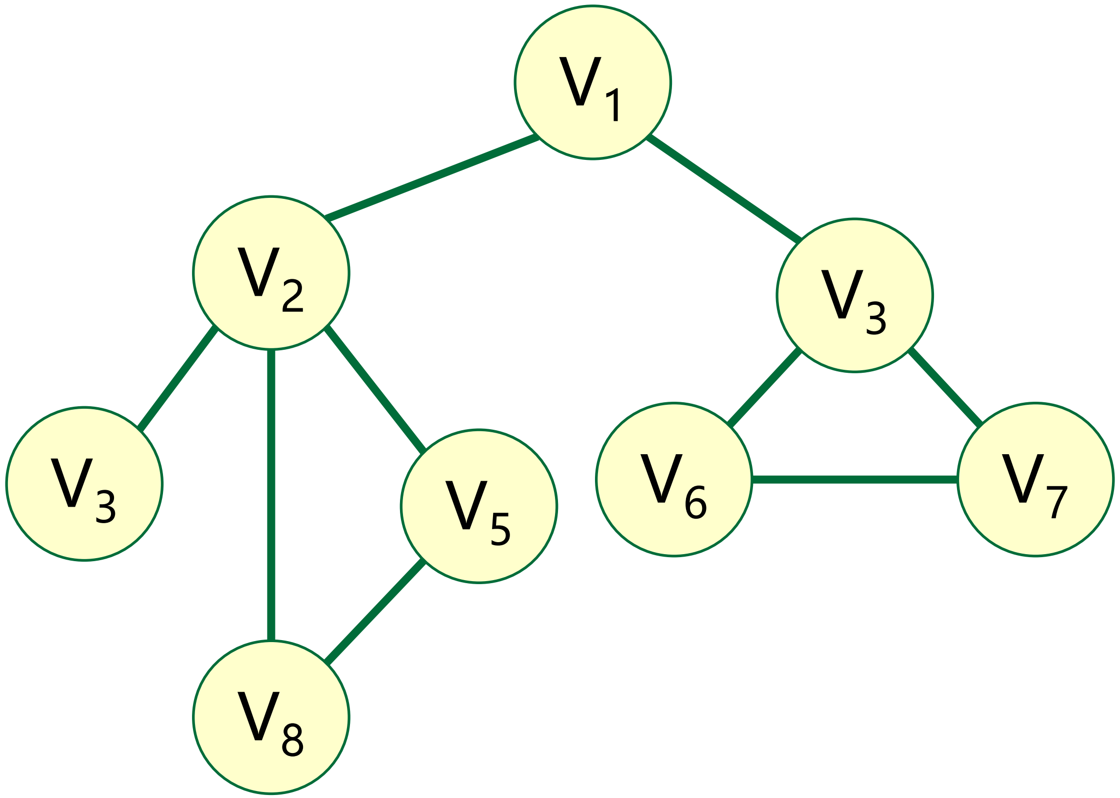
遍历的应用举例1：

求一条从顶点v到顶点s的简单路径

```
Status _DFSearch(Graph G,intv,VertexTypes,SqList&PATH) {
        //深度优先搜索的递归程序
        visited[v] = TRUE;   //访问第v个顶点
        ListAppend_Sq(PATH,G.vertices[v].data); //将点v添加到路径
        if (G.vertices[v].data == s)  return TRUE;//找到路径
        for(w =firstNeighbor(G, v); w !=-1; w =nextNeighbor(G, v, w))
                if (!visited[w])
                        if (DFSearch(G, w, s, PATH)) return TRUE;
        ListDelete_Sq(PATH,G.vertices[v]. data);
        return FALSE;
}//DFSearch
```

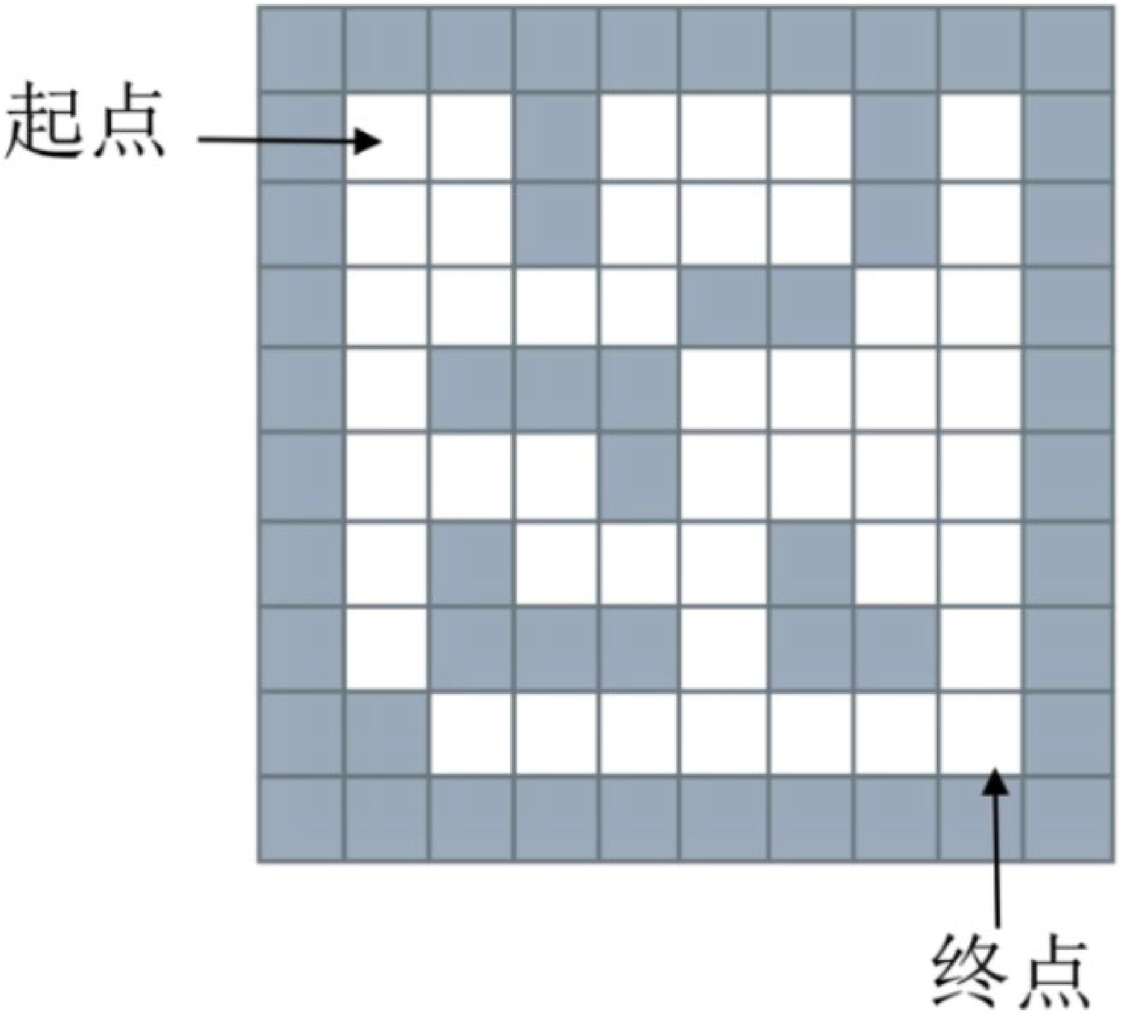
遍历的应用举例2：

求两个顶点之间的一条路径长度最短的路径



6.3 图的遍历

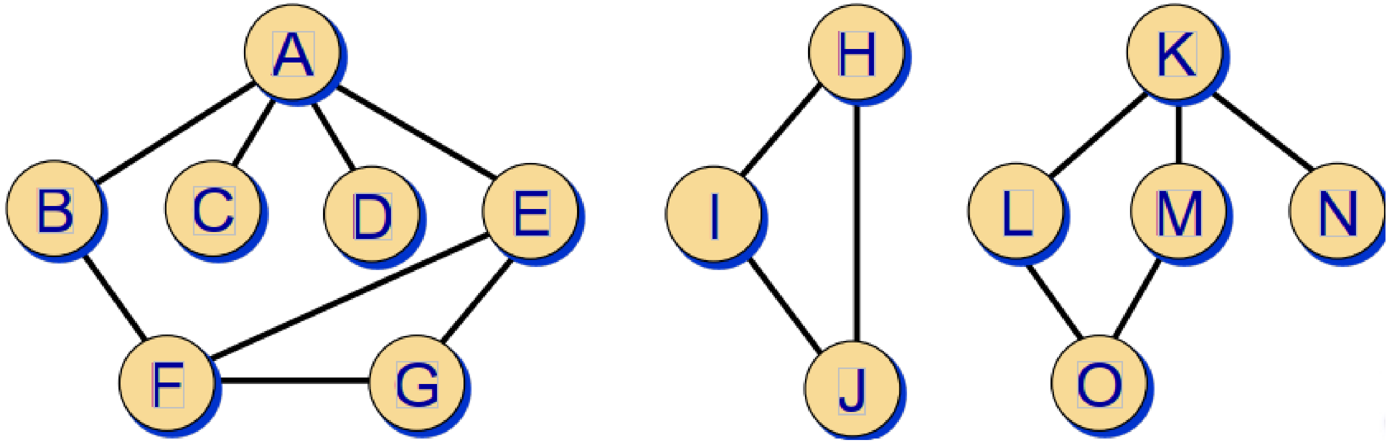
遍历的应用举例3：
迷宫问题



遍历的应用举例4：

寻找连通分量

1、从无向图的每一个连通分量重的一个顶点出发进行遍历，可求得无向图的所有连通分量



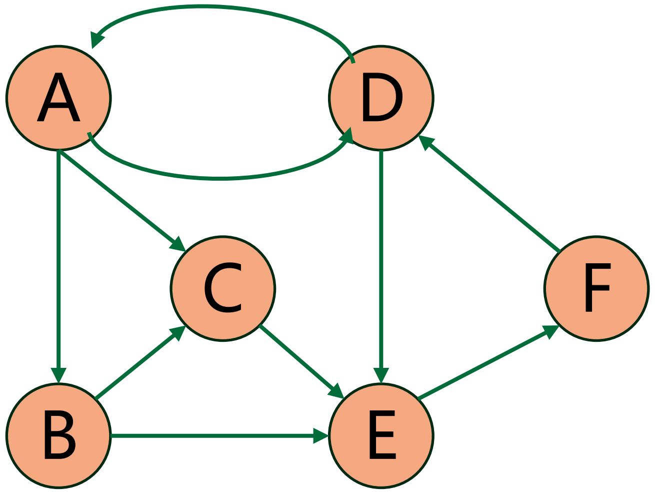
6.3 图的遍历

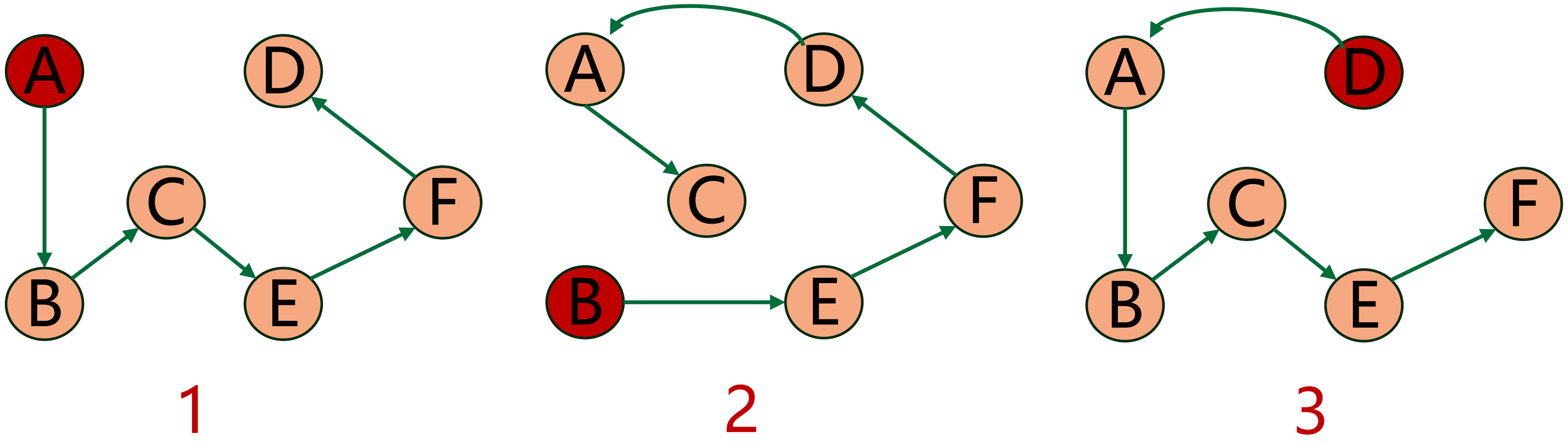
遍历的应用举例4：

寻找连通分量

2、从不同的顶点出发，对有向强连接图进行遍历，可以得到不同的**深度优先生成树**

3种深度优先生成树示例





6.3 图的遍历

遍历的应用举例4：

寻找连通分量

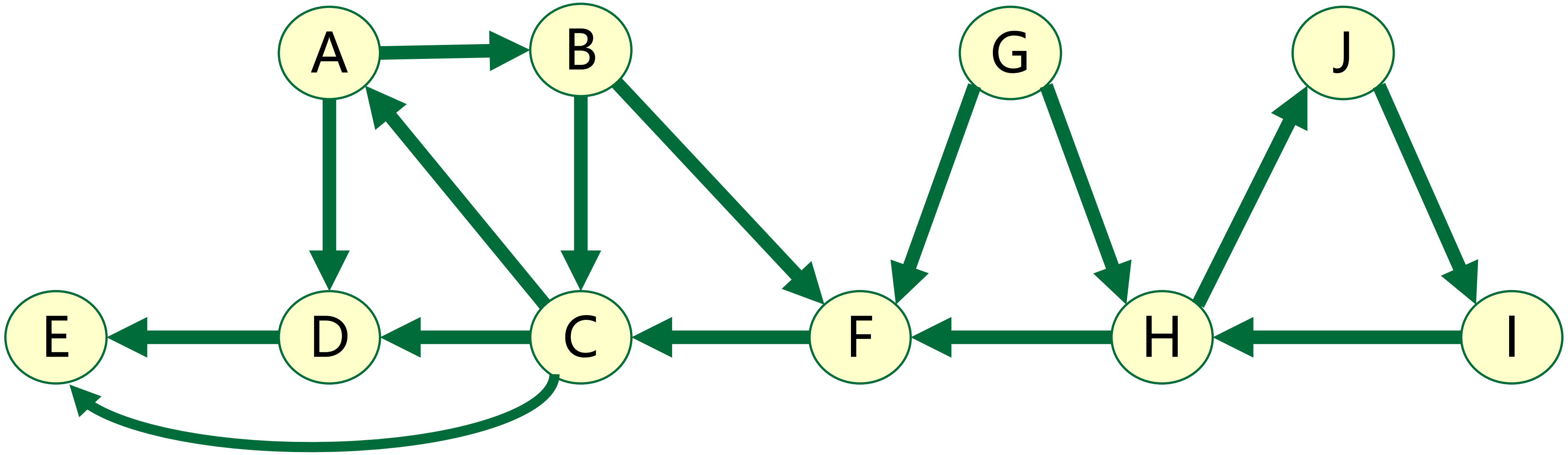
3、非强连通有向图的遍历一般得到的结果是生成森林

对于非强连通图，从某个顶点出发，只能遍历一个弱连通分量

6.3 图的遍历

遍历的应用举例**5**：

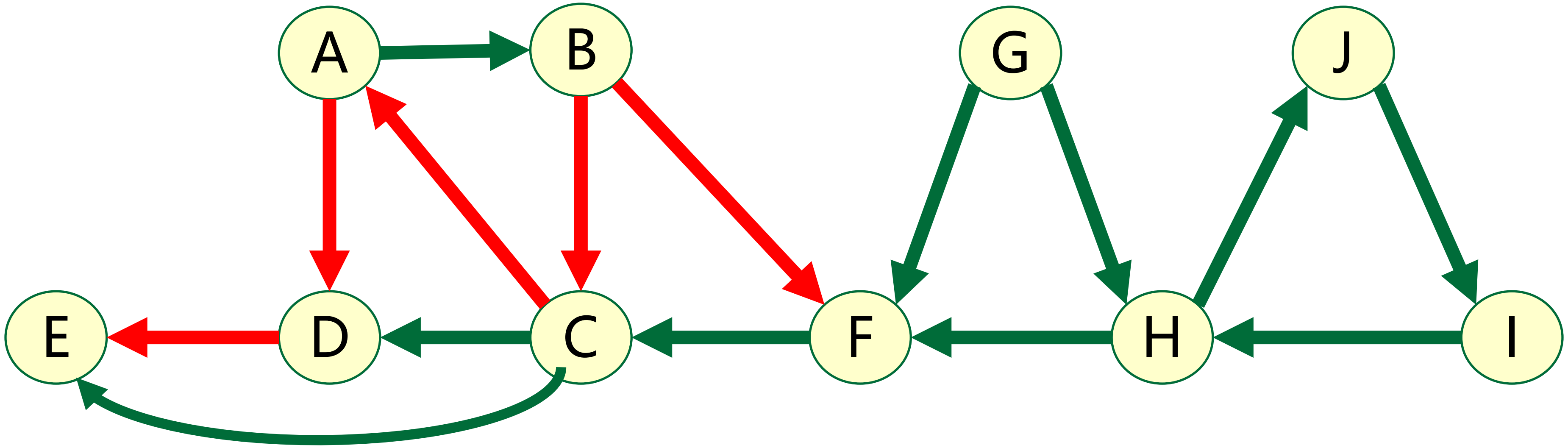
寻找强连通分量的**Kosaraju**算法



遍历的应用举例5：

寻找强连通分量的**Kosaraju**算法

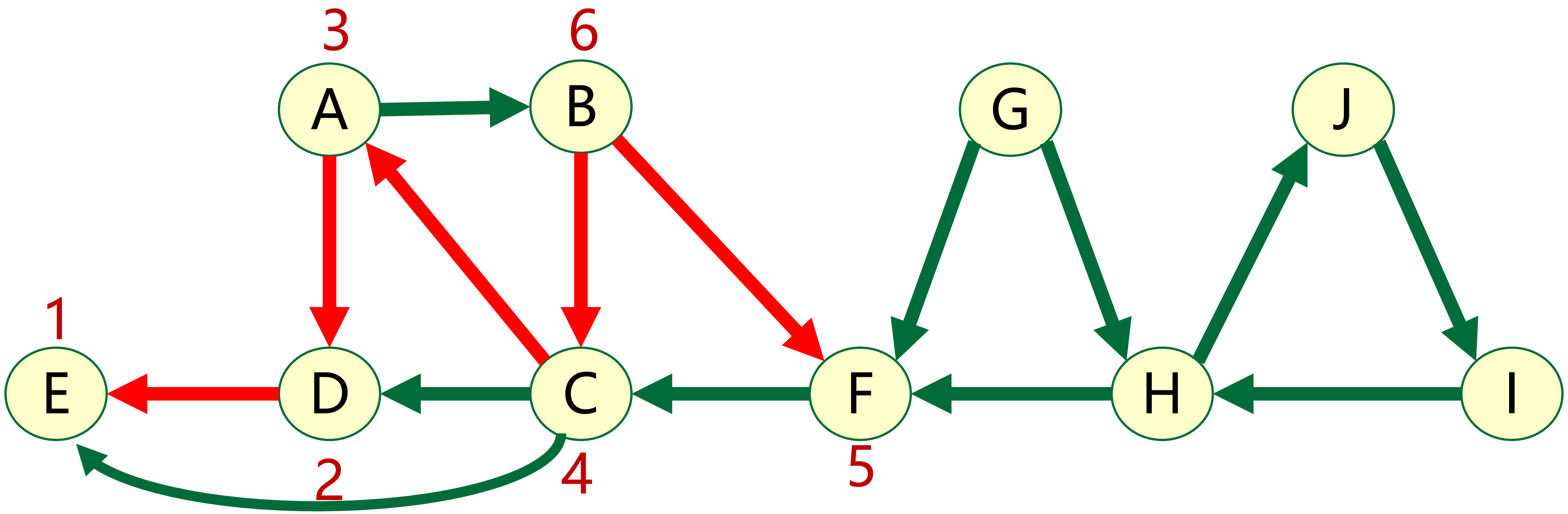
1、首先从任一顶点（A）开始对图进行一次DFS，在回退时记录对顶点回溯的顺序：**EDACFBIJHG**



遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

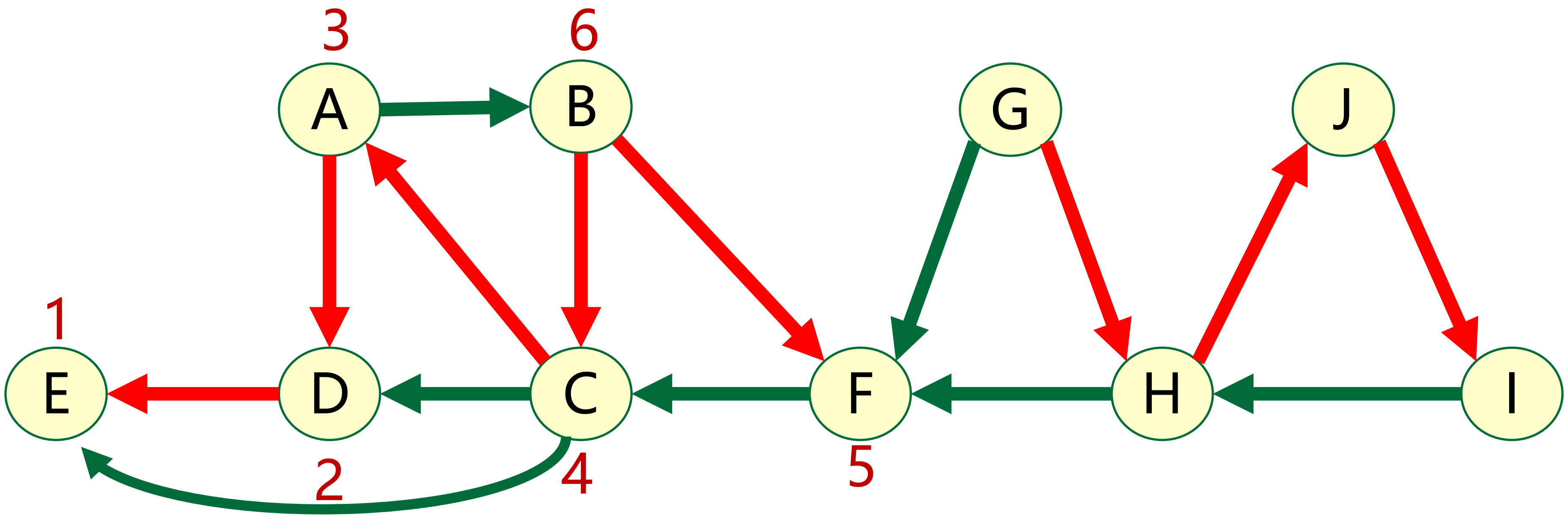
1、首先从任一顶点（A）开始对图进行一次DFS，在回退时记录对顶点回溯的顺序：**EDACFBIJHG**



遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

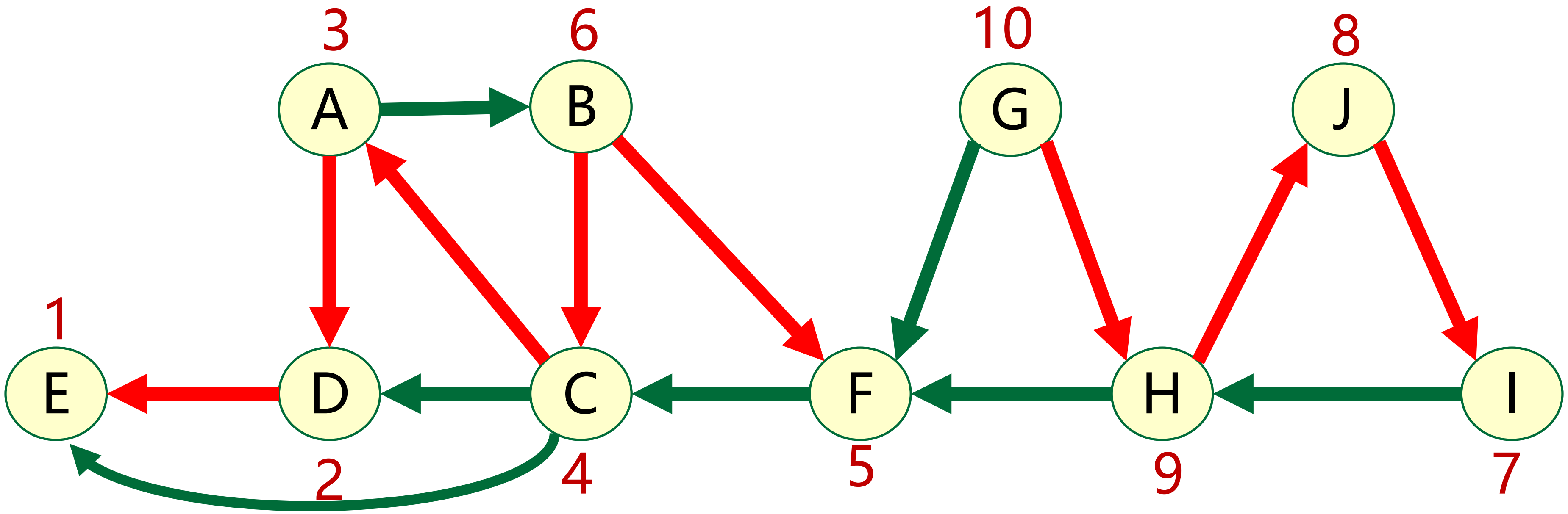
1、首先从任一顶点（A）开始对图进行一次DFS，在回退时记录对顶点回溯的顺序：**EDACFBIJHG**



遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

1、首先从任一顶点（A）开始对图进行一次DFS，在回退时记录对顶点回溯的顺序：**EDACFBIJHG**

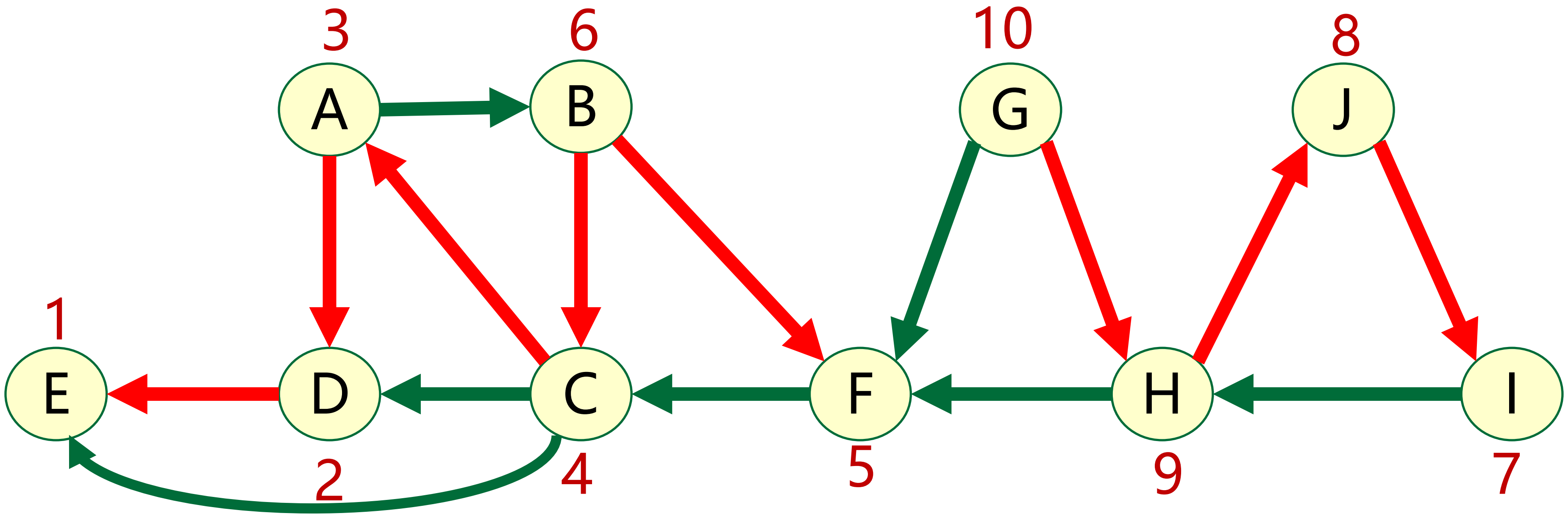


6.3 图的遍历

遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

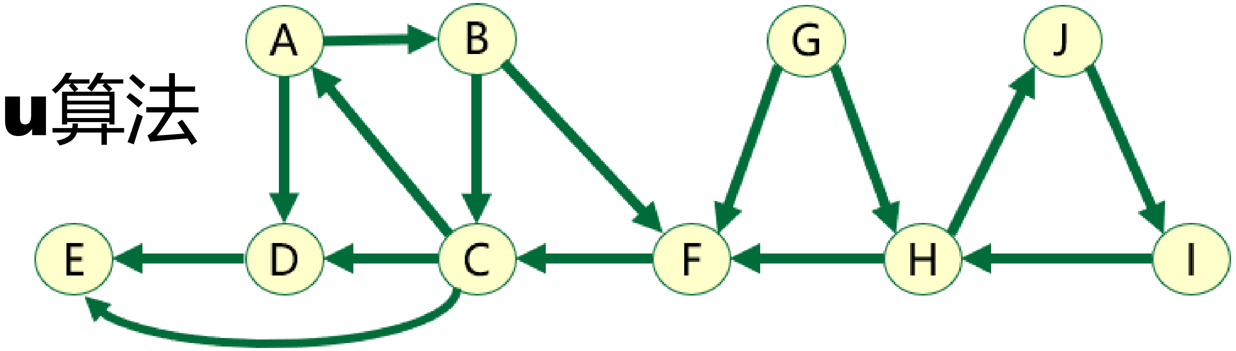
2、把图中所有的有向边逆转

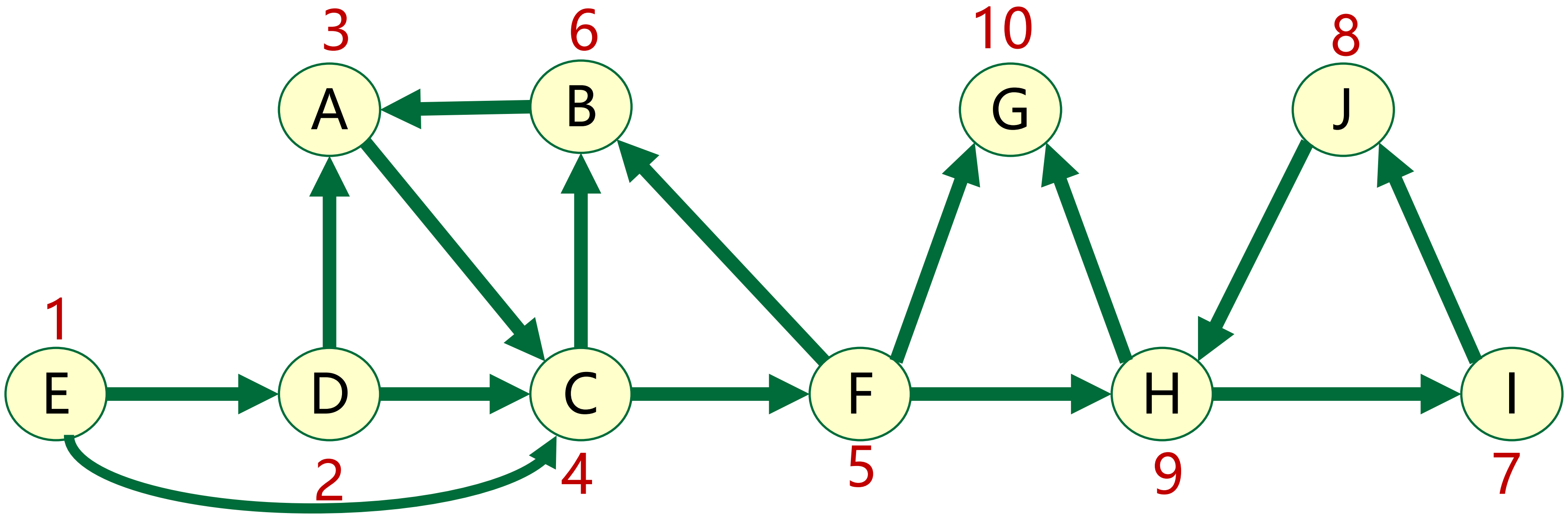


遍历的应用举例**5**：

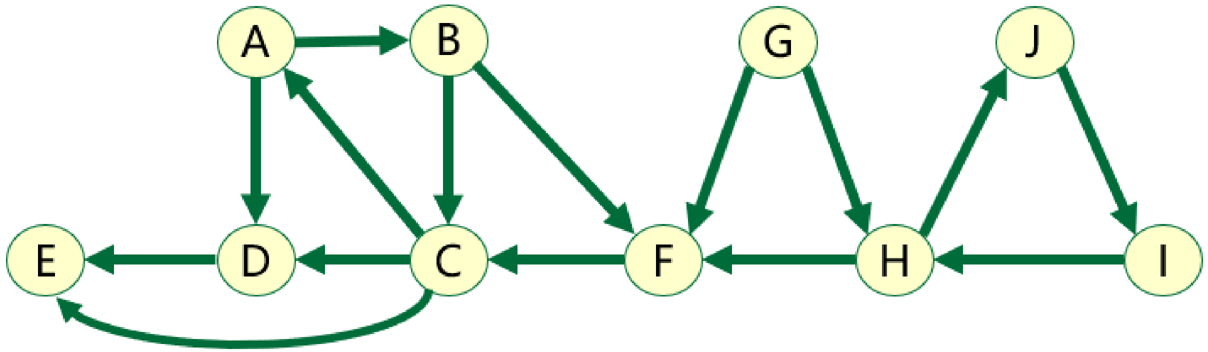
寻找强连通分量的**Kosaraju**算法

2、把图中所有的有向边逆转





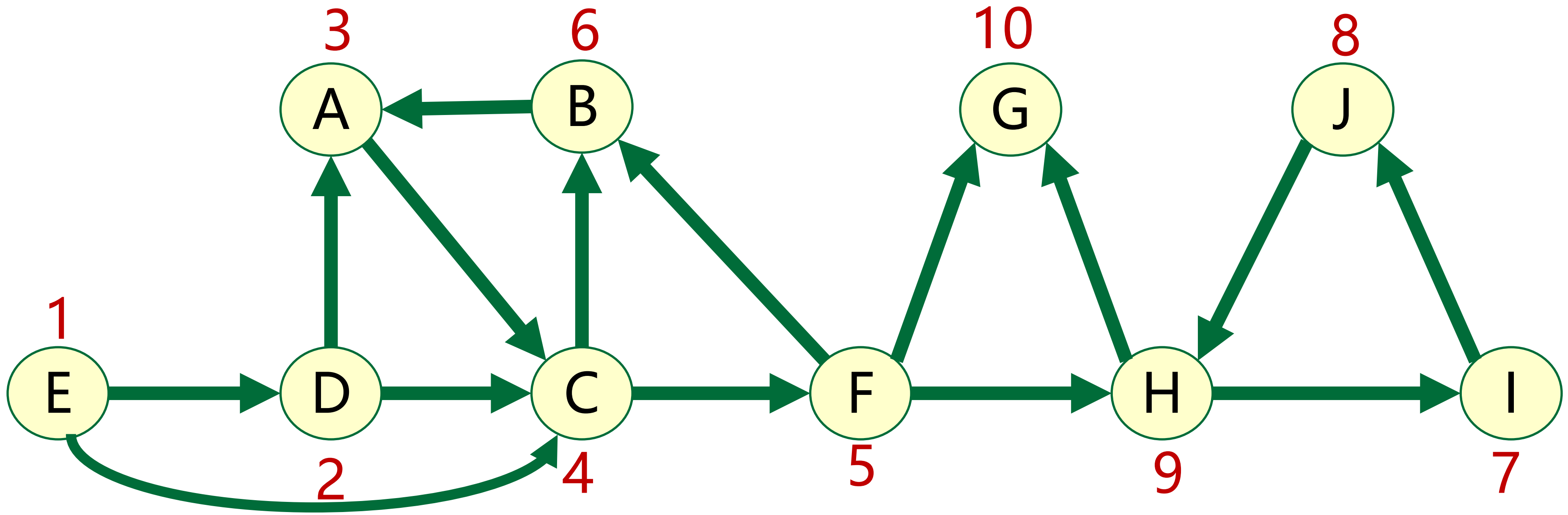
6.3 图的遍历



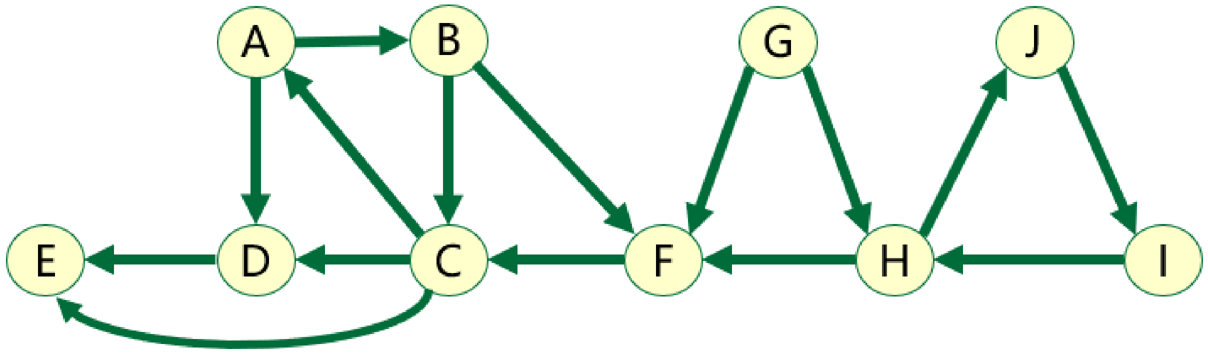
遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

3、对得到的新图沿回溯顺序**EDACFBIJHG，从最后一个顶点G开始，再进行一次DFS，所得到的深度优先森林（树），即为强连通分量的划分。**



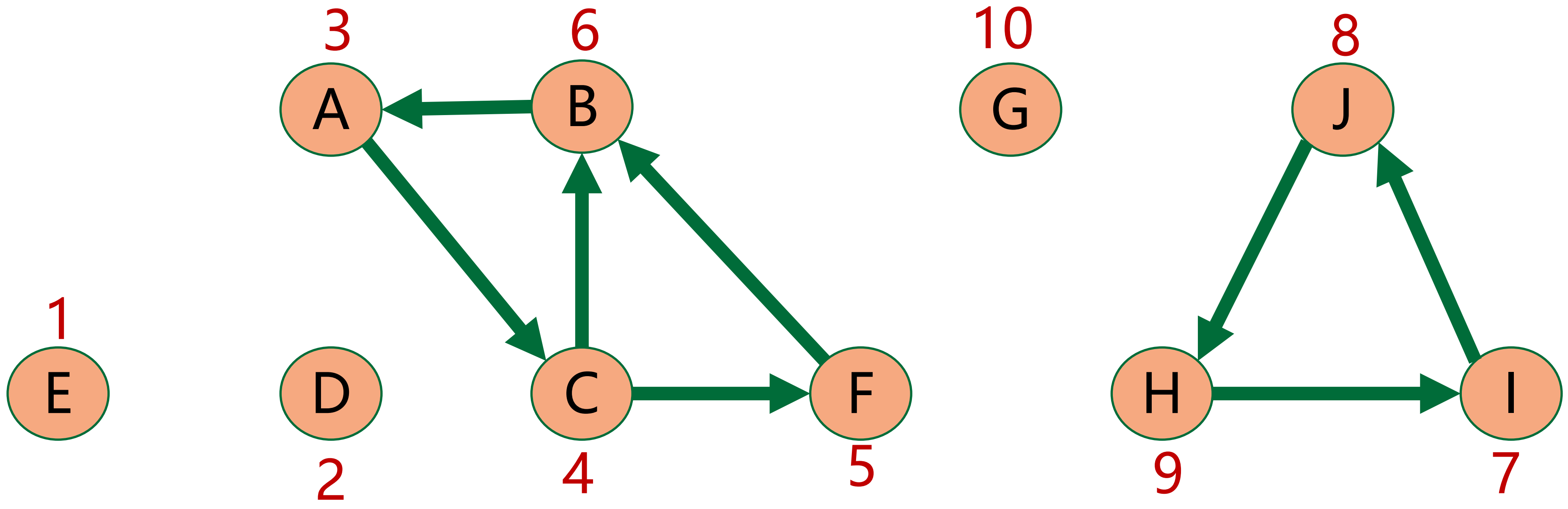
6.3 图的遍历



遍历的应用举例**5**：

寻找强连通分量的**Kosaraju**算法

3、对得到的新图沿回溯顺序**EDACFBIJHG，从最后一个结点G开始，再进行一次DFS，所得到的深度优先森林（树），即为强连通分量的划分。**



6.3 图的遍历

遍历的应用举例6：

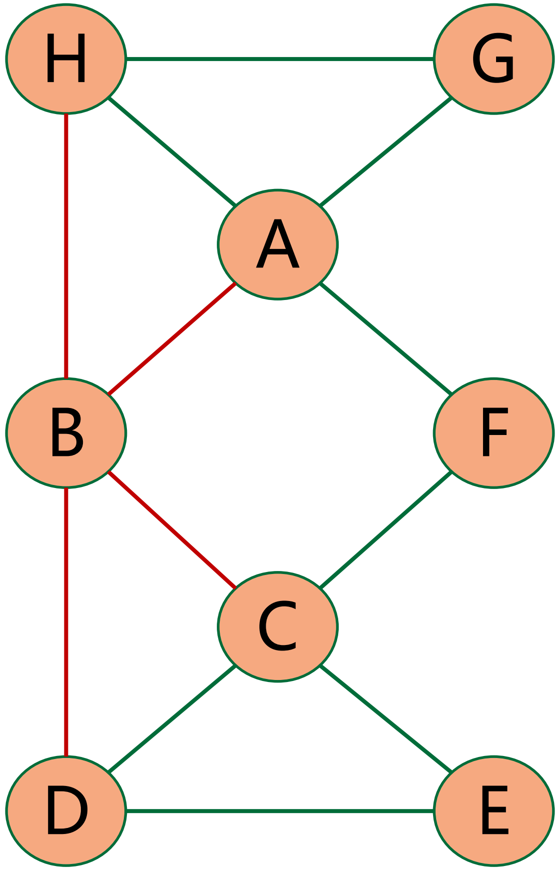
定义：若从一个连通图中删去一个顶点及其关联的边，连通图成为两个或多个连通分量，则称该顶点为关节点。

定义：若从一个连通图中删去任意一个顶点及其关联的边，它仍是一个连通图的话，则该连通图称为重（双连通图，二连通图）。

可知：

重连通图没有关节点

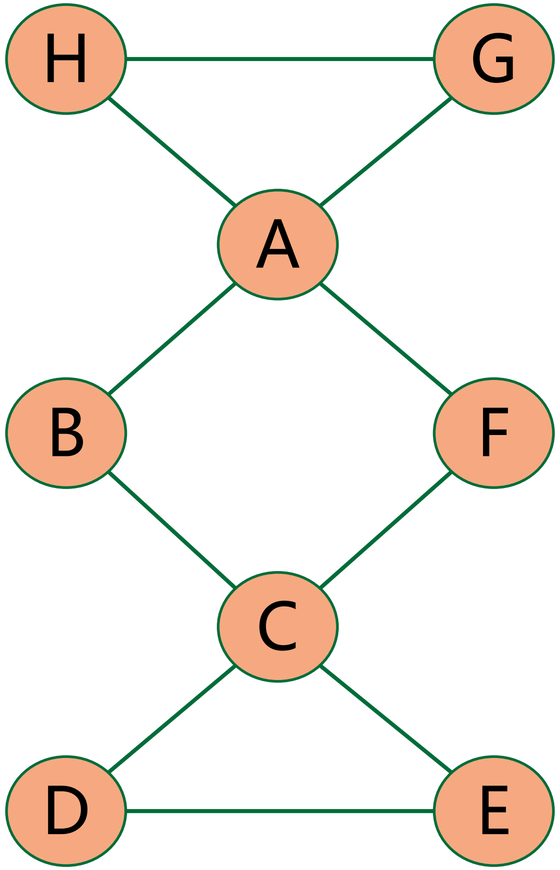
没有关节点的图为重连接图



6.3 图的遍历

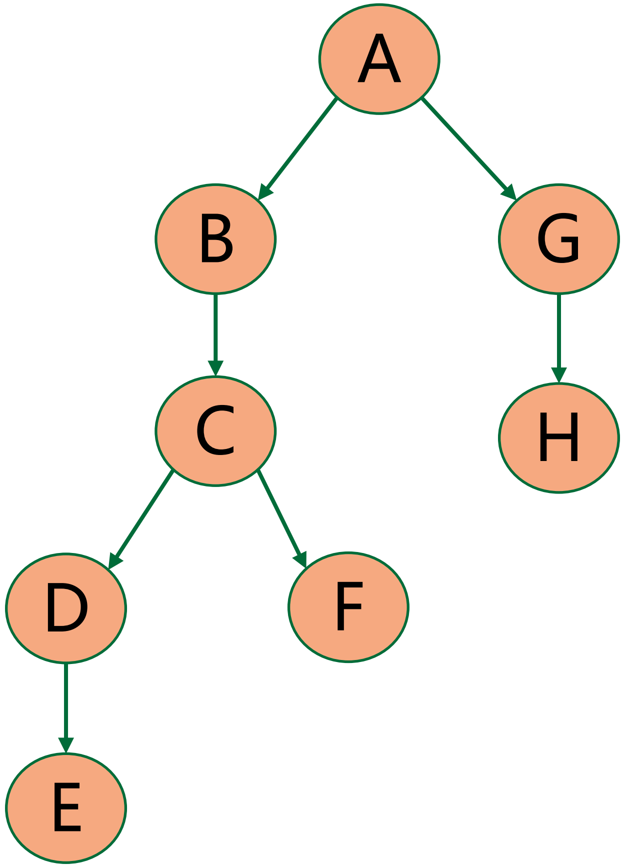
遍历的应用举例6：

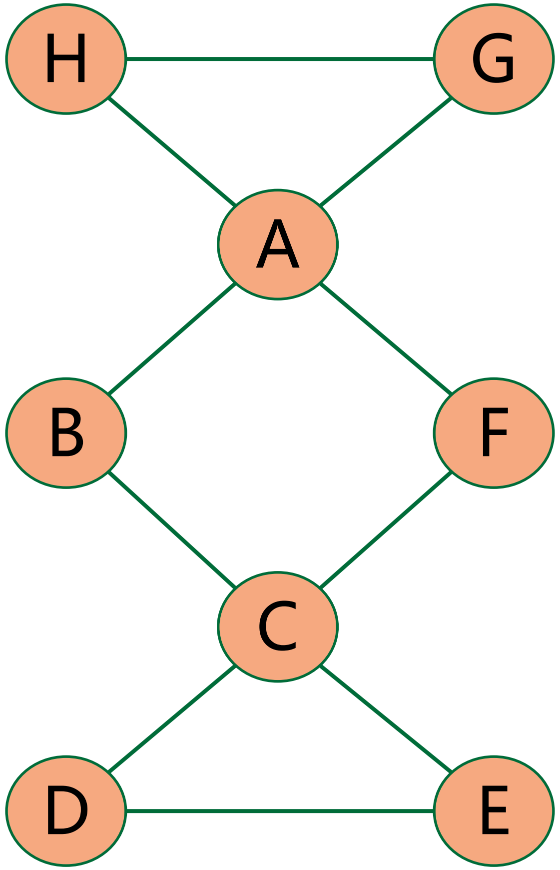
对右图G进行深度优先遍历，得到深度优先生成树T。



遍历的应用举例6：

对右图G进行深度优先遍历，得到深度优先生成树T。

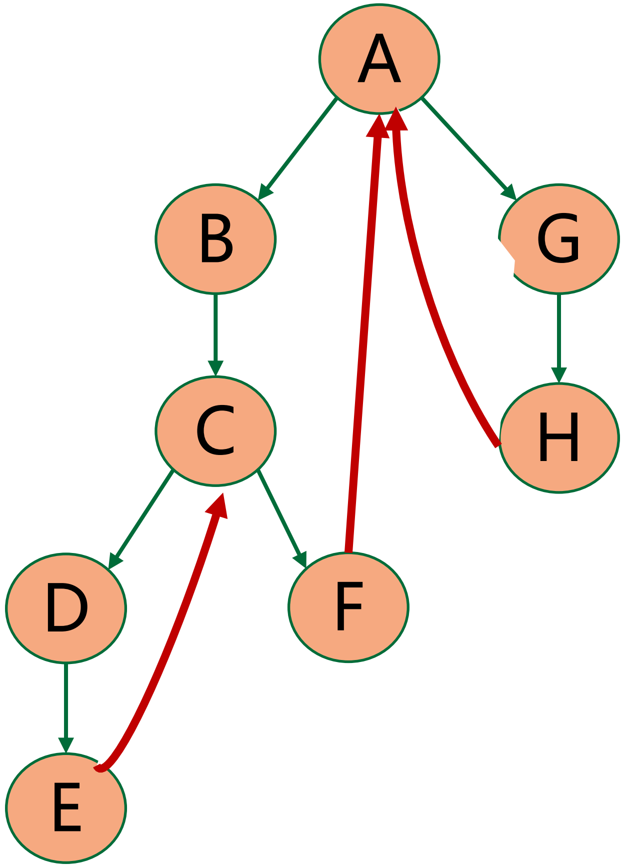


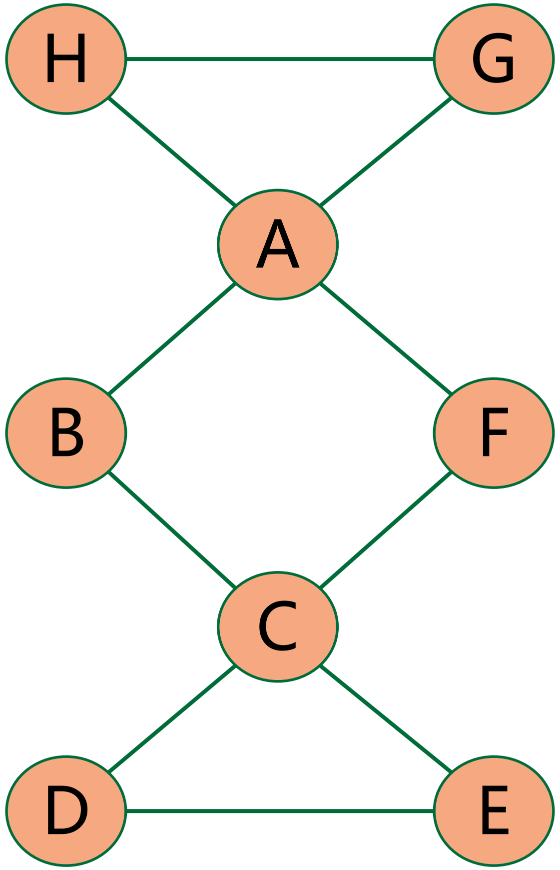


6.3 图的遍历

遍历的应用举例6：

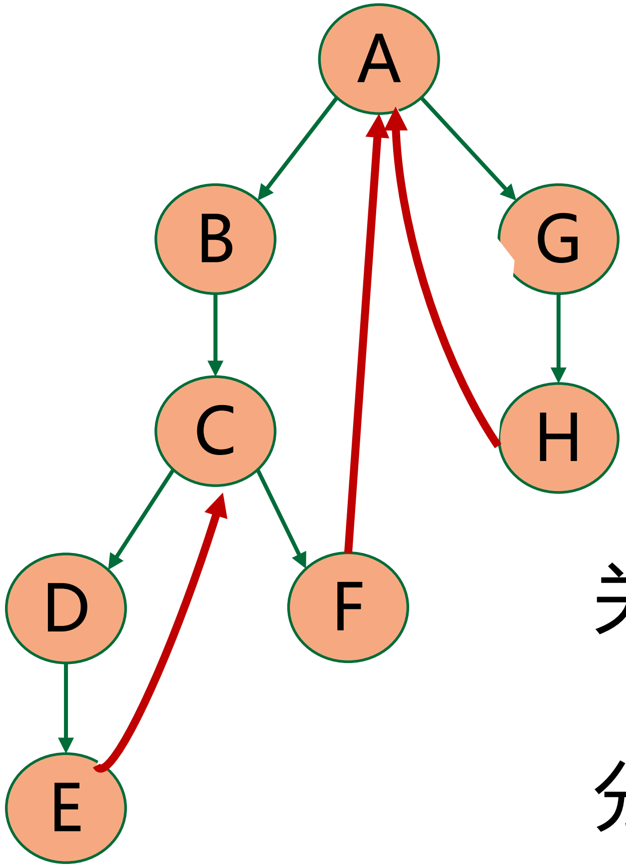
对右图G进行深度优先遍历，得到深度优先生成树T。在遍历树T中添加回联边。即在G中，但不在T中的边。

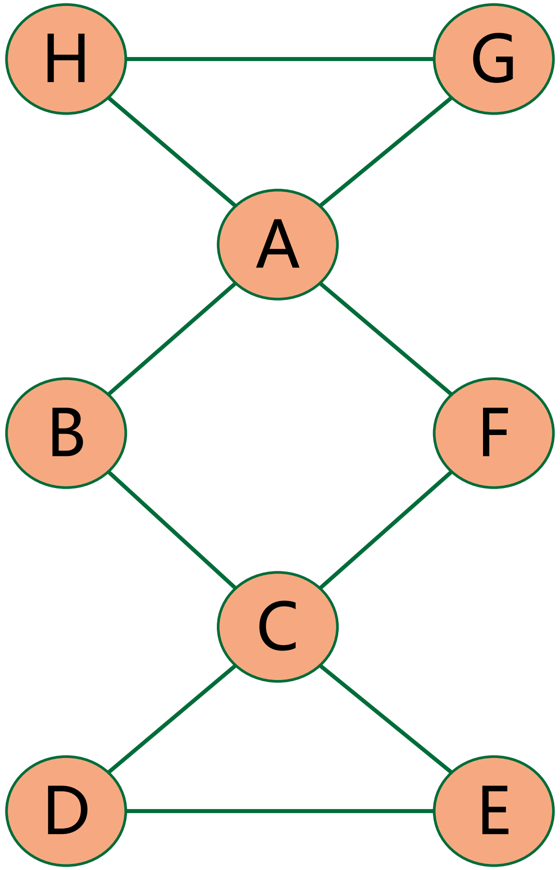




遍历的应用举例6：

对右图G进行深度优先遍历，得到深度优先生成树T。在遍历树T中添加回联边。即在G中，但不在T中的边。





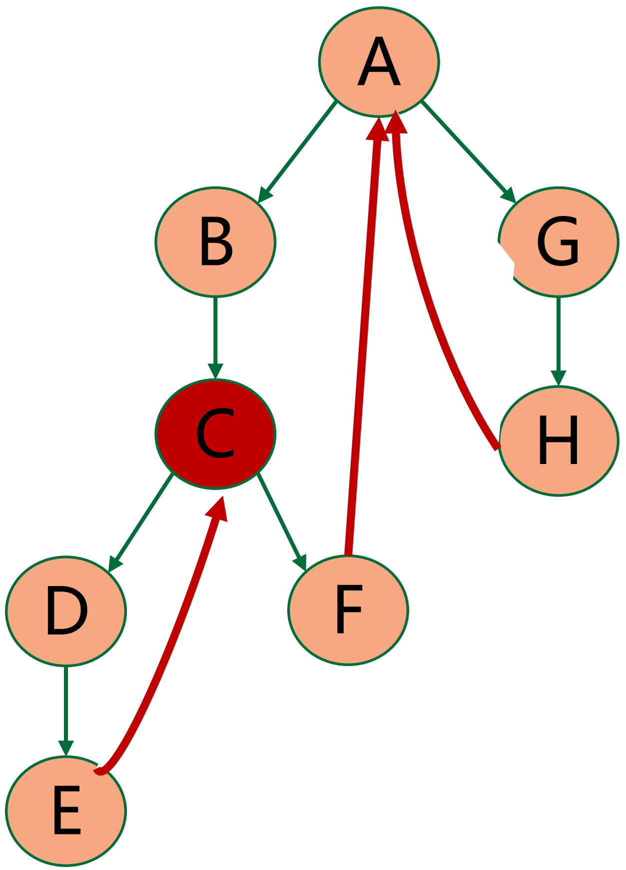
关节点的特征1：

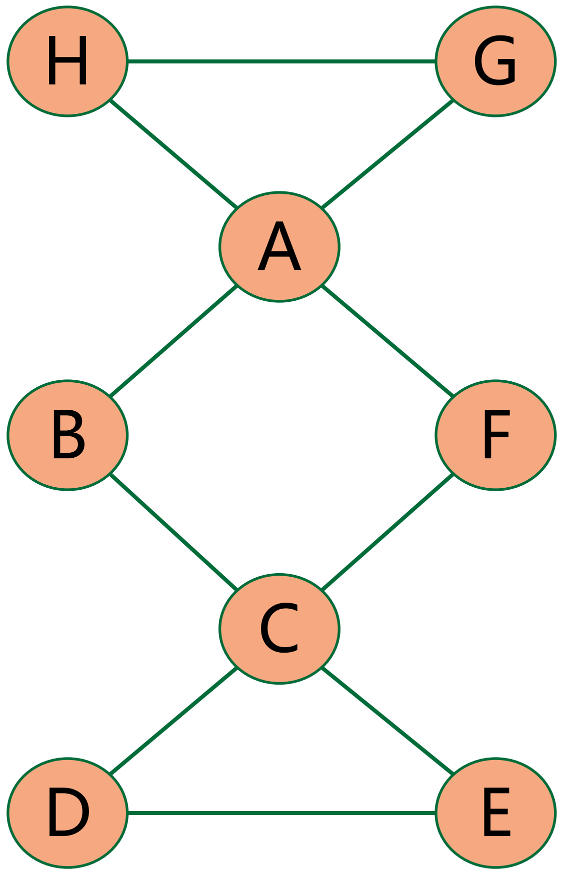
若生成树的根节点，有两个或两个以上的分支，则此顶点（生成树的根）必为关节点。

6.3 图的遍历

遍历的应用举例6：

对右图G进行深度优先遍历，得到深度优先生成树T。在遍历树T中添加回联边。即在G中，但不在T中的边。





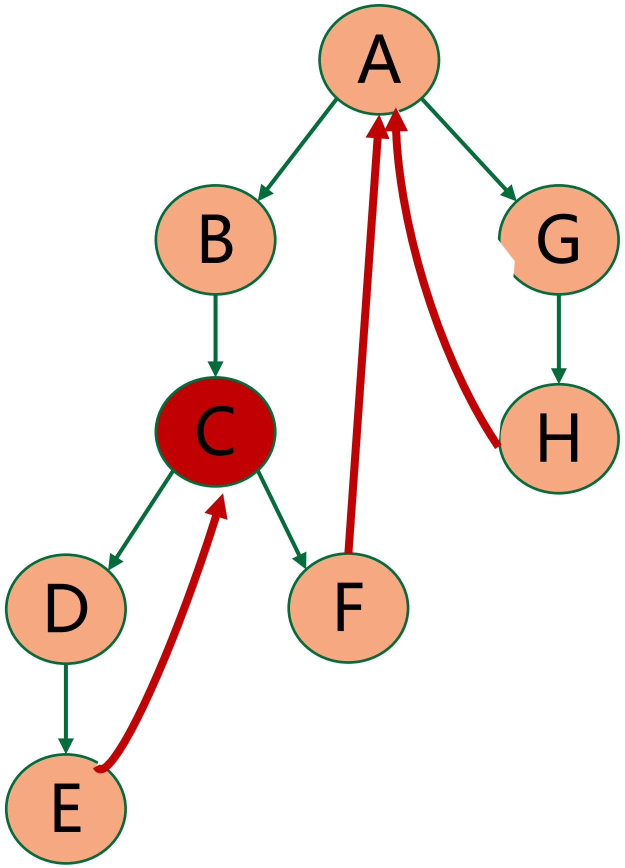
关节点的特征2：

对生成树上除了根以外的其他任意顶点v，若顶点v的所有子节点中存在一个子节点w_i，w_i及其子孙都没有指向v祖先节点的回边，则顶点v必为关节点。

6.3 图的遍历

关节点的特征2：

对生成树上除了根以外的其他任意顶点v，若顶点v的所有子节点中存在一个子节点w_i，w_i及其子孙都没有指向v祖先节点的回边，则顶点v必为关节点。



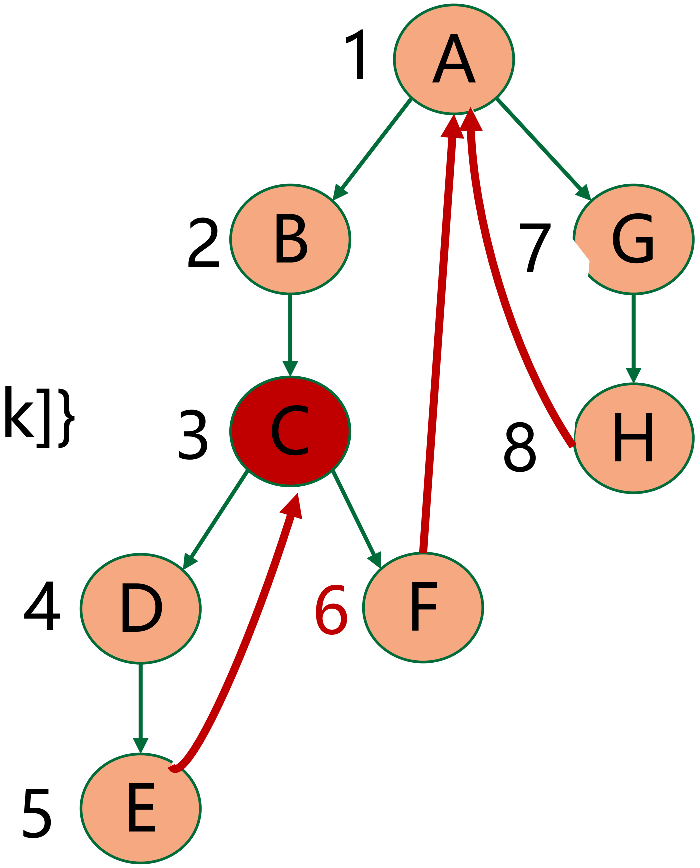
定义最低深度优先数low[v]:

low[v] = min{visited[v],low[w],visited[k]}

w是v在T中的子结点；

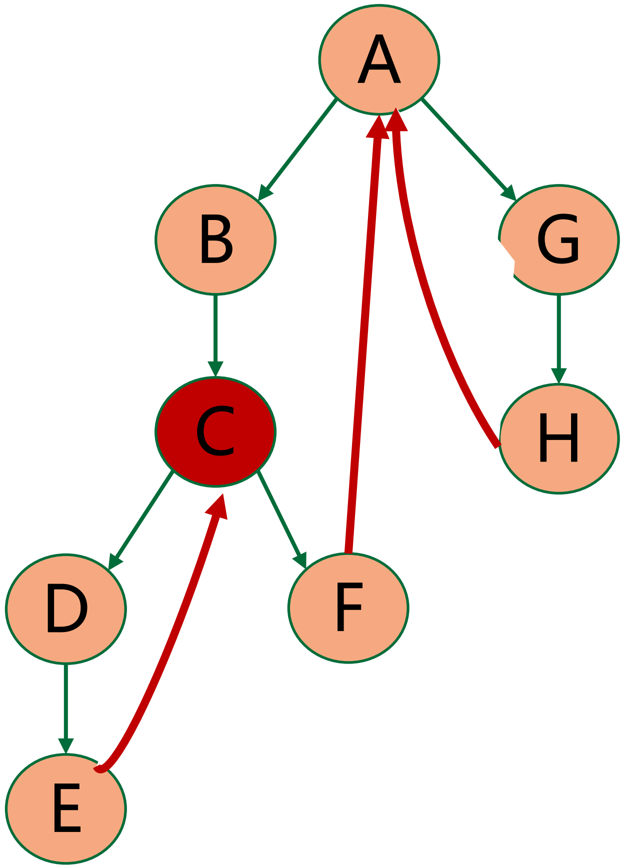
k是v在T中回联的祖先结点；

low[v]后序遍历T过程中求得



关节点的特征2：

对生成树上除了根以外的其他任意顶点v，若顶点v的所有子节点中存在一个子节点w_i，w_i及其子孙都没有指向v祖先节点的回边，则顶点v必为关节点。



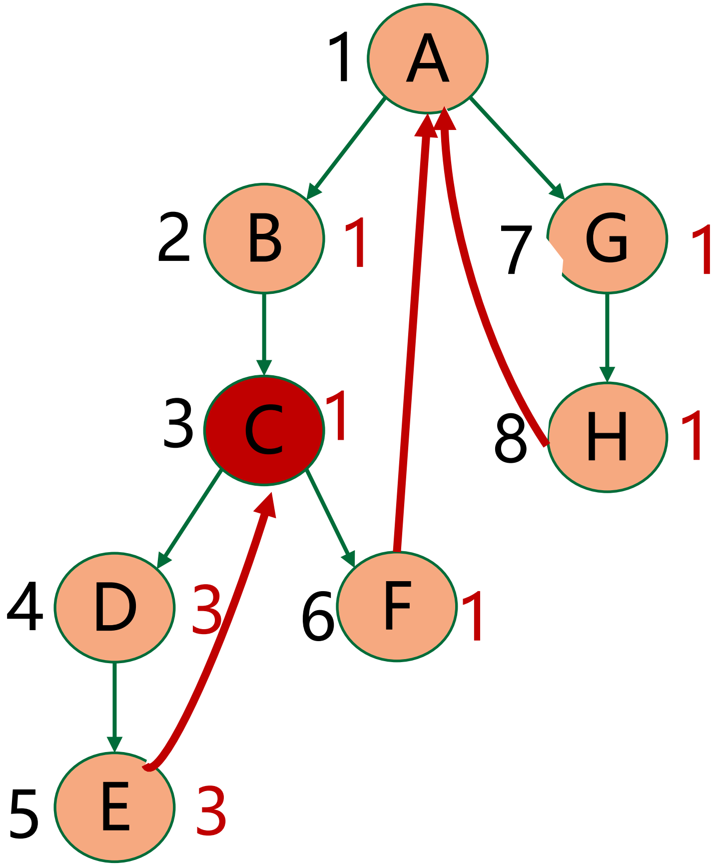
定义最低深度优先数low[v]:

low[u]：表示顶点u及其子树中的点，通过反向边，能够回溯到的最早的点（visited最小）的visited值。

若:

low[w]>=visited[v]

w是v在T中的子孙结点；则v是关节点。



low[E]==3 >= visited[C]== 3

6.3 图的遍历

	深度优先遍历	图的广度优先遍历
算法描述	从某顶点v出发 1)访问顶点v 2)依次从v的所有未被访问的邻接点$w_1,w_2,w_3.....w_n$出发，深度优先遍历该结点。 直到图中所有访问过的顶点的邻接点都被访问。	从某顶点v出发 1)访问顶点v 2)访问v所有未被访问的邻接点$w_1,w_2,w_3.....w_n$ 3)依次从这些邻接点出发，访问其所有未被访问的邻接点。依此类推，直到图中所有访问过的顶点的邻接点都被访问
特点	栈	队列
时间复杂度	邻接矩阵$O(n^2)$	邻接矩阵$O(n^2)$
	邻接表O(n+e)	邻接表O(n+e)

6.4 图的最小生成树

问题提出

要在n个城市间建立通信联络网，如何省钱?

顶点——表示城市

权——城市间建立通信线路所需花费代价

希望找到一棵生成树，它的每条边上的权值之和（即建立该通信网所需花费的总代价）最小——最小代价生成树

MST(Minimum cost Spanning Tree)

网络中的SpaningTree Protocol

6.4 图的最小生成树

生成树（Spanning tree）：包含无向连通图G所有顶点的极小连通子图称为G的生成树。

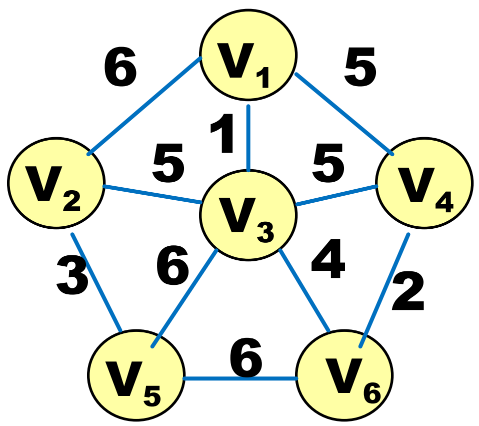
特点:

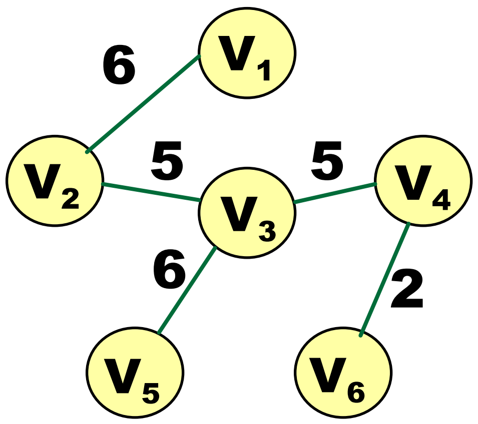
1）T是G的连通子图

2）T包含G的所有顶点

3）T中无回路

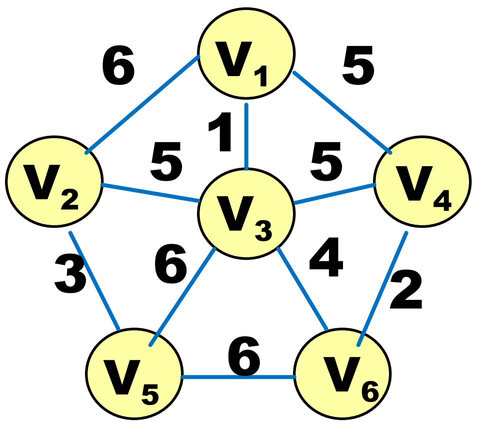
4）T中有n-1条边

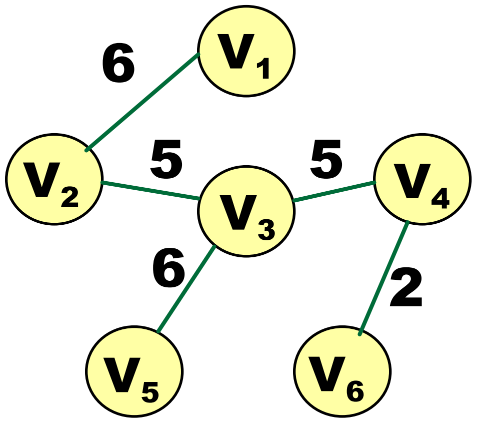




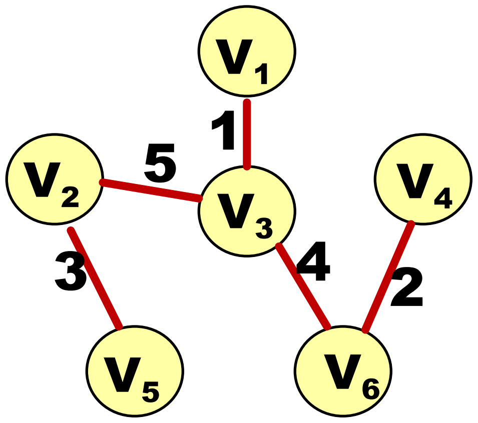
6.4 图的最小生成树

最小生成树（Least Weight Spanning Tree）
Minimun Spanning Tree（MST）
权之和最小的生成树





权之和：24



权之和：15

6.4 图的最小生成树

典型算法

- **普里姆(Prim)算法**

将顶点归并，与边数无关，适于稠密网。

- **克鲁斯卡尔(Kruskal)算法**

将边归并，适于求稀疏网的最小生成树。

6.4 图的最小生成树-普里姆算法

普里姆算法（Prim）

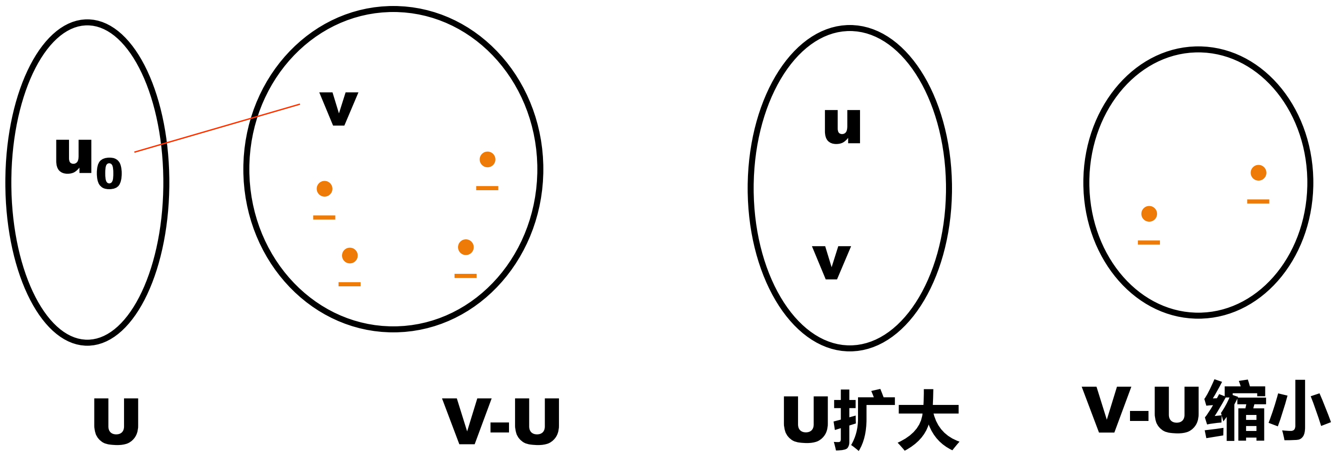
设 G=(V, GE) 为一个具有 n 个顶点的连通网络，T=(U, TE) 为构造的生成树。

(1) 初始时，$U = \{u_0\}$，$TE = \phi$；

(2) 在所有 $u \in U$ 且 $v \in V-U$ 的边 (u, v) 中选择一条权值最小的边，不妨设为(u,v)；

(3) (u,v) 加入TE，同时将 v 加入U；

(4) 重复(2)(3)，直到 U=V 为止；

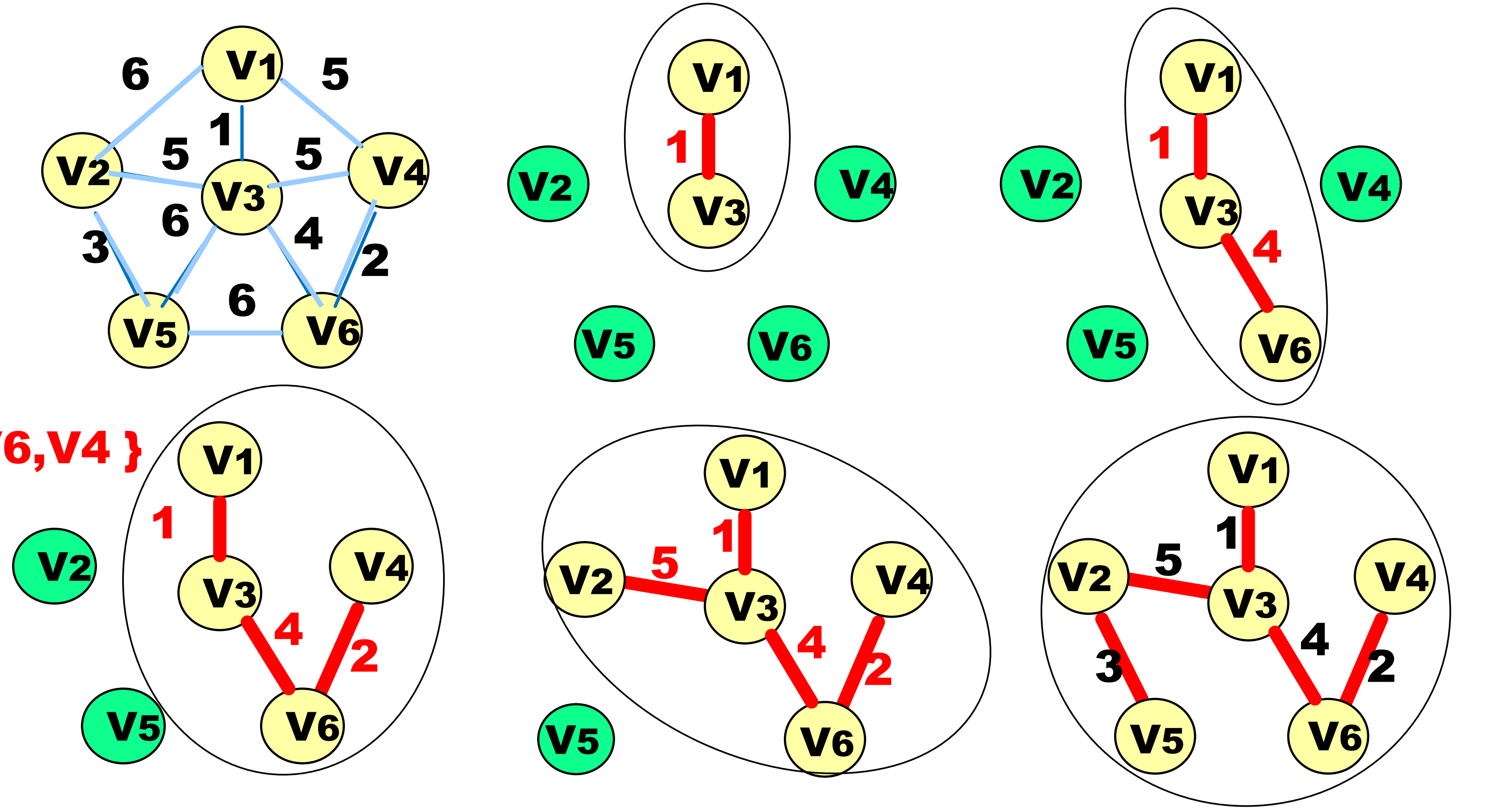


6.4 图的最小生成树-普里姆算法

普里姆算法（Prim）

U={ V1 }　　U={ V1,V3 }　　U={ V1,V3,V6}

U={ V1,V3,V6,V4 }



U={ V_1,V_3,V_6,V_4,V_2 }　　U={ V_1,V_3,V_6,V_4,V_2,V_5 }

6.4 图的最小生成树-普里姆算法

辅助数组closedge[ ]

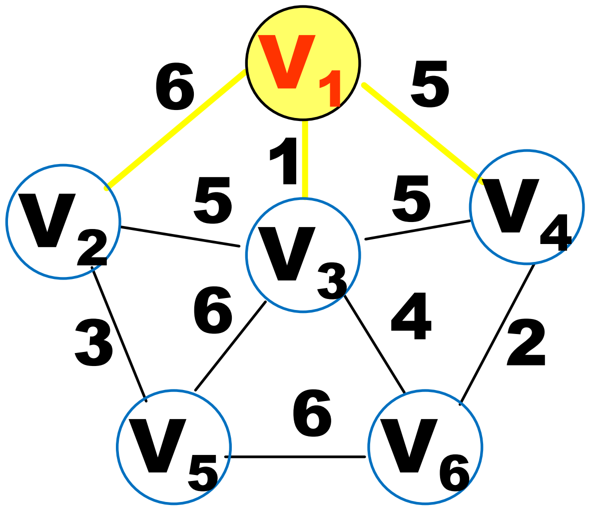
```
struct {
    VertexType  Adjvex;    // 相关顶点
    VRType      lowcost;   // 最小边的权值
} closedge[ MAX_VERTEX_NUM ];
```

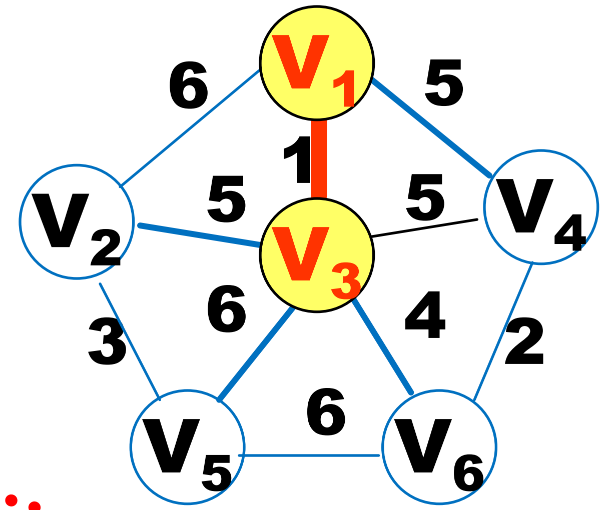
Closedge. Adjvex[v]:
  顶点v到子集U中权最小边 (v, u) 关联的顶点u

Closedge.lowcost[v]:
  顶点v到子集U权最小边 (v, u) 的权值(距离)

	1	2	3	4	5	6
closedge.Adjvex						
closedge.Lowcost						

6.4 图的最小生成树

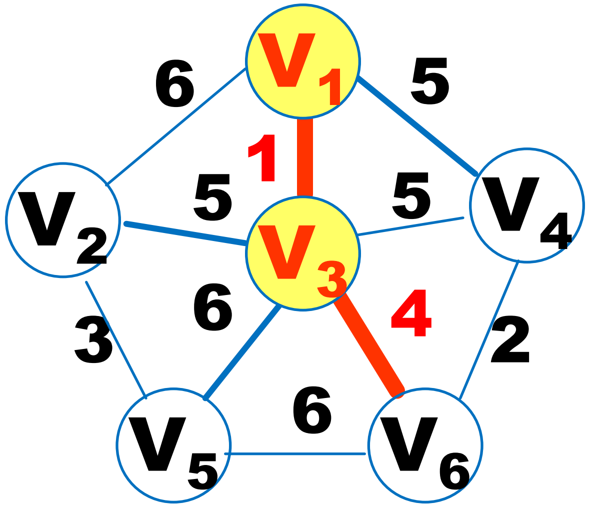


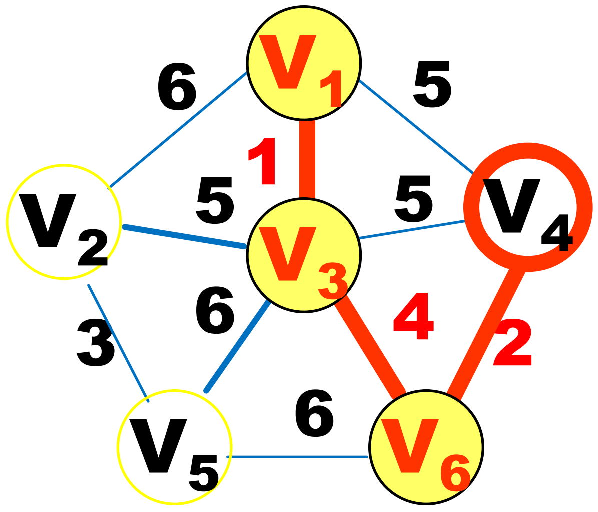


	0(V_1)	1(V_2)	2(V_3)	3(V_4)	4(V_5)	5(V_6)
closedge.Adjvex	V_1	V_1	V_1	V_1	V_1	V_1
closedge.Lowcost	0	6	1	5	max	max

	0(V_1)	1(V_2)	2(V_3)	3(V_4)	4(V_5)	5(V_6)
closedge.Adjvex	V_1	V_3	V_1	V_1	V_3	V_3
closedge.Lowcost	0	5	1	5	6	4

6.4 图的最小生成树

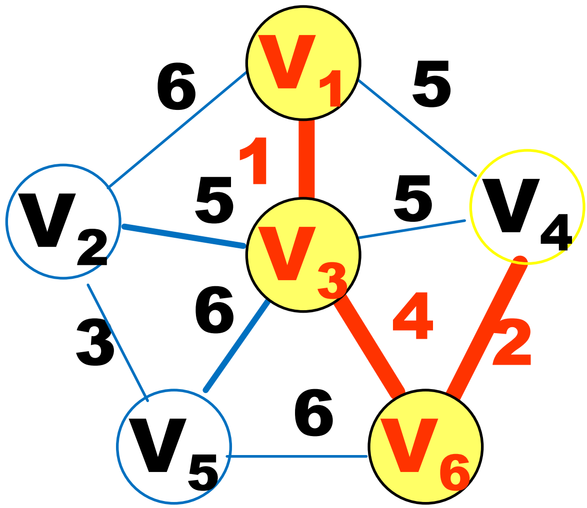


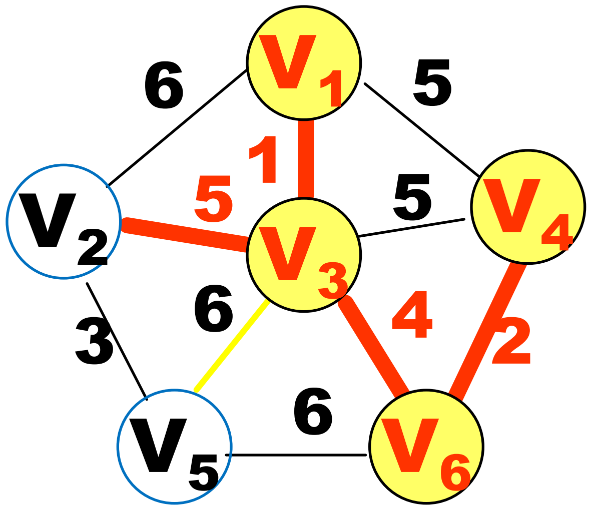


	$0(V_1)$	$1(V_2)$	$2(V_3)$	$3(V_4)$	$4(V_5)$	$5(V_6)$
closedge.Adjvex		V_3	V_1	V_1	V_3	V_3
closedge.Lowcost	0	5	1	5	6	4

	$0(V_1)$	$1(V_2)$	$2(V_3)$	$3(V_4)$	$4(V_5)$	$5(V_6)$
closedge.Adjvex		V_3	V_1	V_6	V_3	V_3
closedge.Lowcost	0	5	1	2	6	4

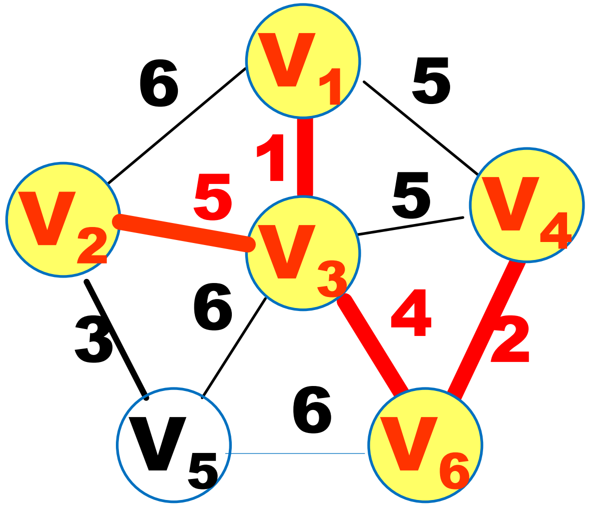
6.4 图的最小生成树

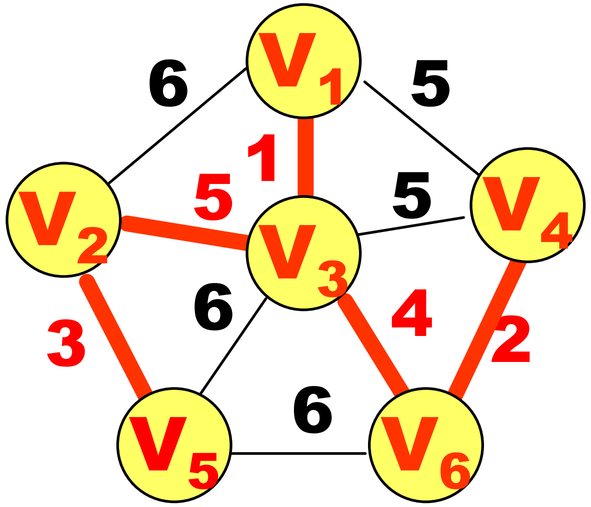




	0(V_1)	1(V_2)	2(V_3)	3(V_4)	4(V_5)	5(V_6)
closedge.Adjvex		V_3	V_1	V_6	V_3	V_3
closedge.Lowcost	0	5	1	2	6	4

6.4 图的最小生成树





	0(V_1)	1(V_2)	2(V_3)	3(V_4)	4(V_5)	5(V_6)
closedge.Adjvex		V_3	V_1	V_6	V_3	V_3
closedge.Lowcost	0	5	1	2	3	4

6.4 图的最小生成树

```
void MiniSpanTree_P( MGraph G, VertexType u )
 {
 //用普里姆算法从顶点u出发构造网G的最小生成树
  k = LocateVex ( G, u );
  for ( j=0; j<G.vexnum; ++j )  // 辅助数组初始化
      if (j!=k)
          closedge[j] = { u, G.arcs[k][j] };
  closedge[k].Lowcost = 0;      // 初始，U={u}
  for ( i=0; i<G.vexnum-1; ++i )
  {

      继续向生成树上添加顶点;

  }
```

6.4 图的最小生成树

```
  k = minimum(closedge);
        // 求出加入生成树的下一个顶点(k)
  printf(closedge[k].Adjvex, G.vexs[k]);
// 输出生成树上一条边
  closedge[k].Lowcost = 0;    // 第k顶点并入U集
  for (j=0; j<G.vexnum; ++j) //修改其它顶点的最小边
    if ( G.arcs[k][j] < closedge[j].Lowcost )
      closedge[j] = { G.vexs[k], G.arcs[k][j] };
```

6.4 图的最小生成树

普里姆算法的性能

设**n**是图的顶点数，普里姆算法的时间复杂度为**O(n^2)**。

与边数无关，适用于求边稠密的网的最小生成树。

6.4 图的最小生成树-Kruskal算法

克鲁斯卡尔(Kruskal)算法

设连通网 N = (V, {E})。

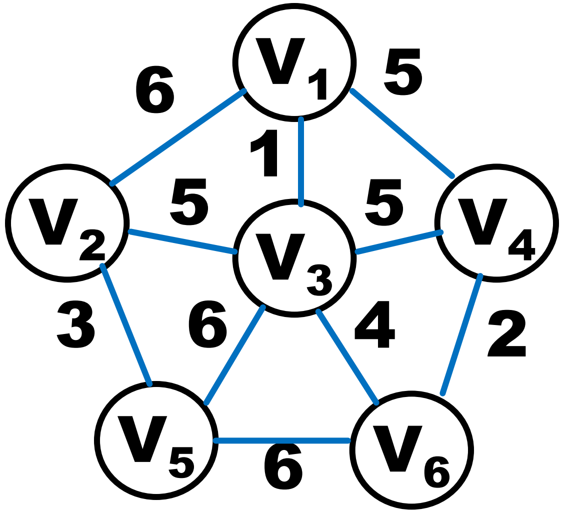
1) 初始时最小生成树只包含图的n个顶点，每个顶点为一棵子树;

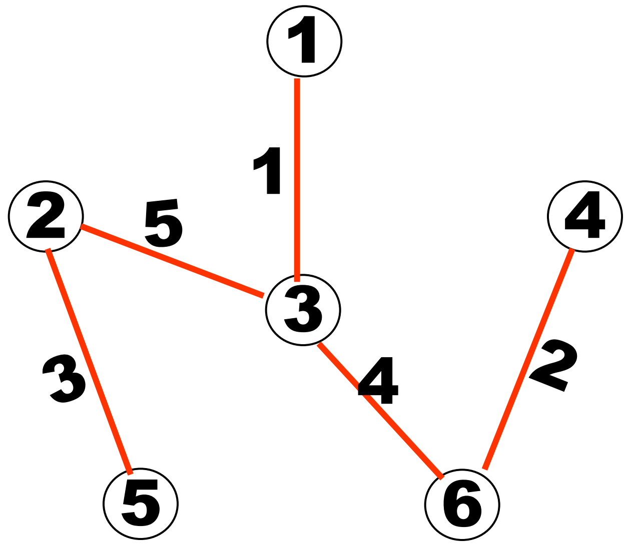
2) 选取权值较小且所关联的两个顶点不在同一子树的边，将此边加入到最小生成树中;

3) 重复2）n-1次，即得到包含n个顶点和n-1条边的最小生成树。

6.4 图的最小生成树-Kruskal算法

克鲁斯卡尔(Kruskal)算法

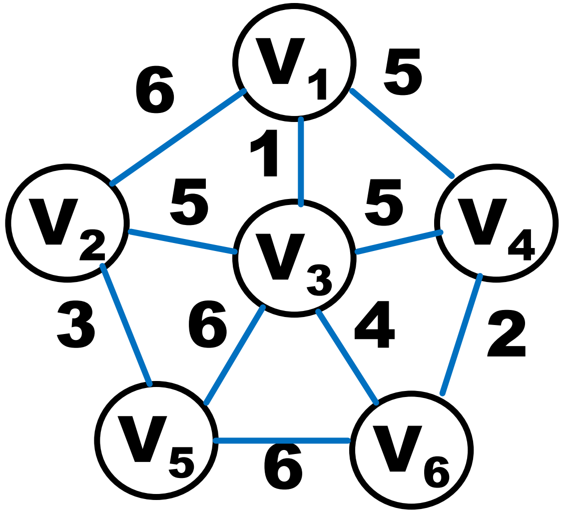


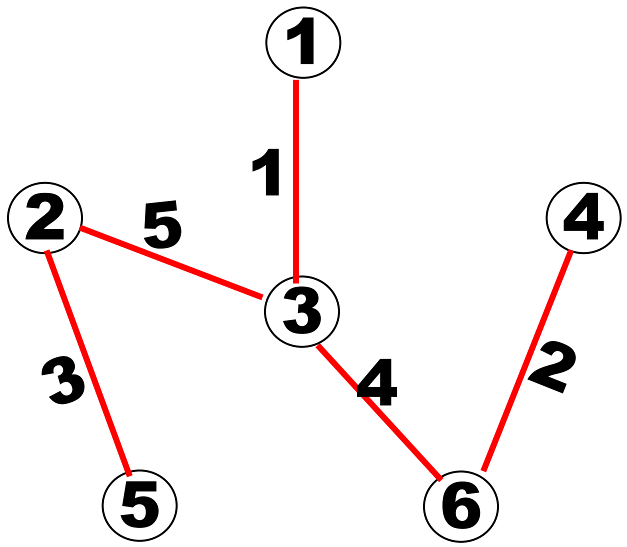


6.4 图的最小生成树-Kruskal算法

克鲁斯卡尔(Kruskal)算法

采用边集数组的形式保存图：





	data	set
1	1	~~1~~ 2
2	2	2
3	3	~~3~~ 1
4	4	~~4~~ 1
5	5	~~5~~ 2
6	6	~~6~~ 4

2

2

1 2

	vexh	vext	weight	flag
0	1	2	6	0
1	1	3	1	~~0~~ 1
2	1	4	5	~~0~~ 2
3	2	3	5	~~0~~ 1
4	2	5	3	~~0~~ 1
5	3	4	5	0
6	3	5	6	0
7	3	6	4	~~0~~ 1
8	4	6	2	~~0~~ 1
9	5	6	6	0

6.4 图的最小生成树-Kruskal算法

```
StatusKruskal_MST(MSTEdgeNodeEV[ ],int n,int e, ,MinSpanTree&T){
    //从已经按权值排序的边数组中顺序取出边，建立最小生成树
    UFSetsVset;Inital(Vset);    //初始化并查集
    InitMinSpanTree(T);         //初始化MST
    j = 0; k = 0;               //j记录加入MST中的边数; k记录当前扫描的边
    while(k <e && j < n){
        u = Find(Vset,EV[k].v1);      v = Find(Vset,EV[k].v2);
        if (u != v ) {                //如果k不是内边，将k加入MST中
            T.edgeValue[T.n++]=EV[k];
            Merge(Uset, u, v);
            j++;
        }   //if ( u != v)
        k++;
    }    //while(k < e)
    if ( j < n-1 ) return-1;
    else return 1;
}//Kruskal_MST
```

6.4 图的最小生成树

克鲁斯卡尔的性能

设图的边数是**e**，克鲁斯卡尔算法的时间复杂度为**O(elog e)**。

适用于求边稀疏的网的最小生成树。

6.4 图的最小生成树-破圈法

基本思想：

对一个有n个顶点的连通带权图，按其权值从大到小顺序逐个删除各边，直到剩下n-1条边。

删除的原则是：删除该边后各个顶点之间还是连通的。

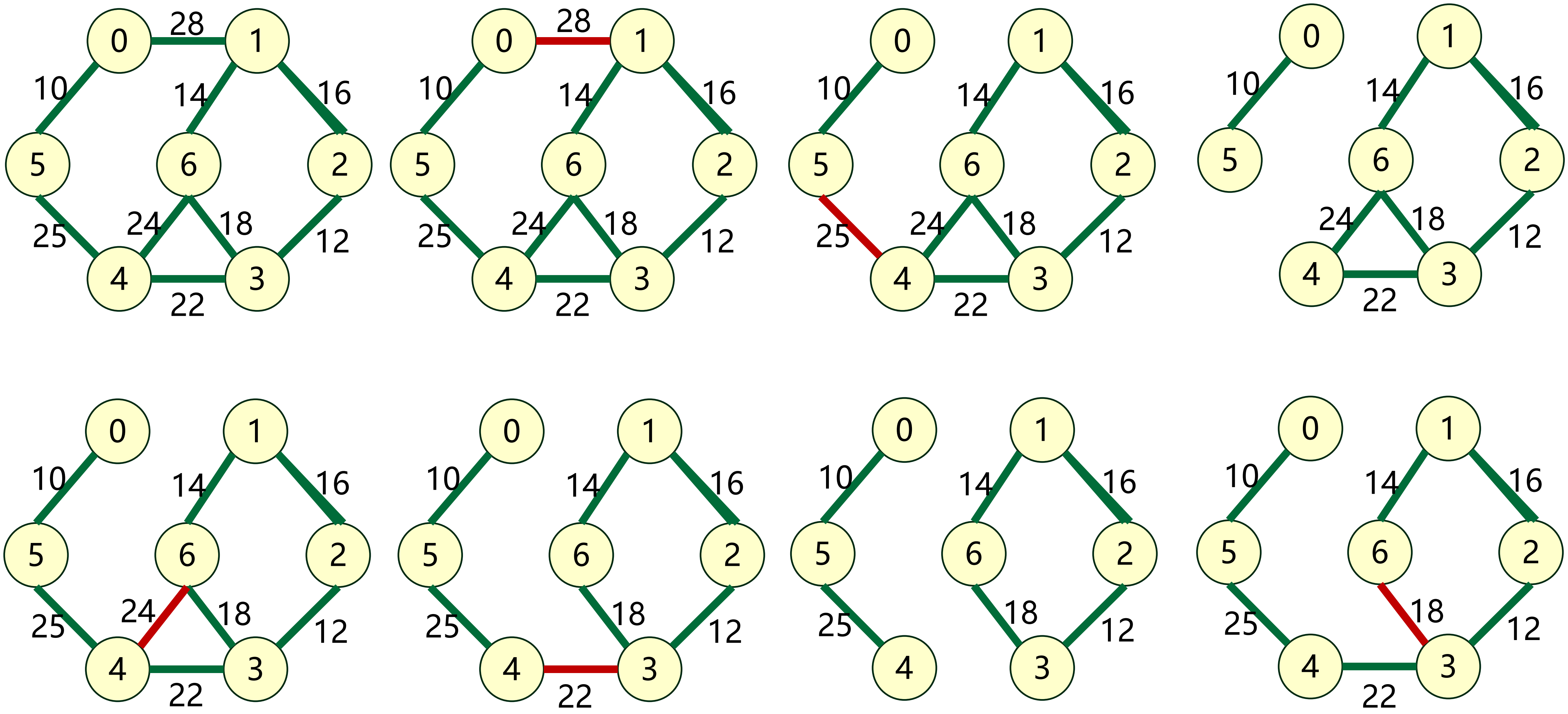
判断图连通性的方法：每删除一条权值最大的边，就

调用DFS遍历图，如果能够访遍图中所有顶点则表明删除此边没有破坏图的连通性，此边可删，否则撤销删除，恢复此边。

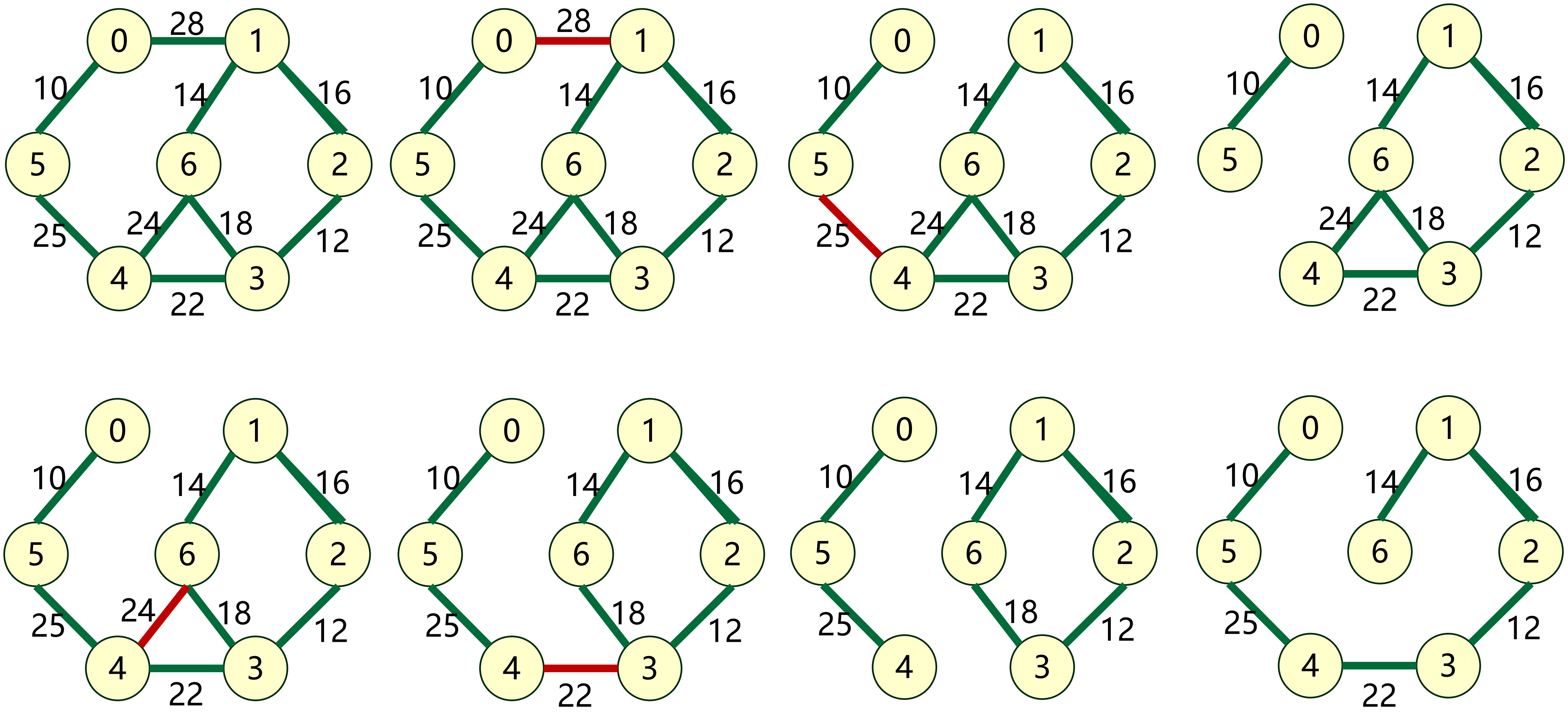
若采用邻接矩阵存储图，DFS的时间代价$O(n^2)$

若采用邻接表，DFS的时间代价O(n+e)。

6.4 图的最小生成树-破圈法



6.4 图的最小生成树-破圈法



6.4 图的最小生成树-迪杰斯特拉(Dijkstra)算法

基本思想：

1、将带权图中每个顶点视为一个独立的连通分量；

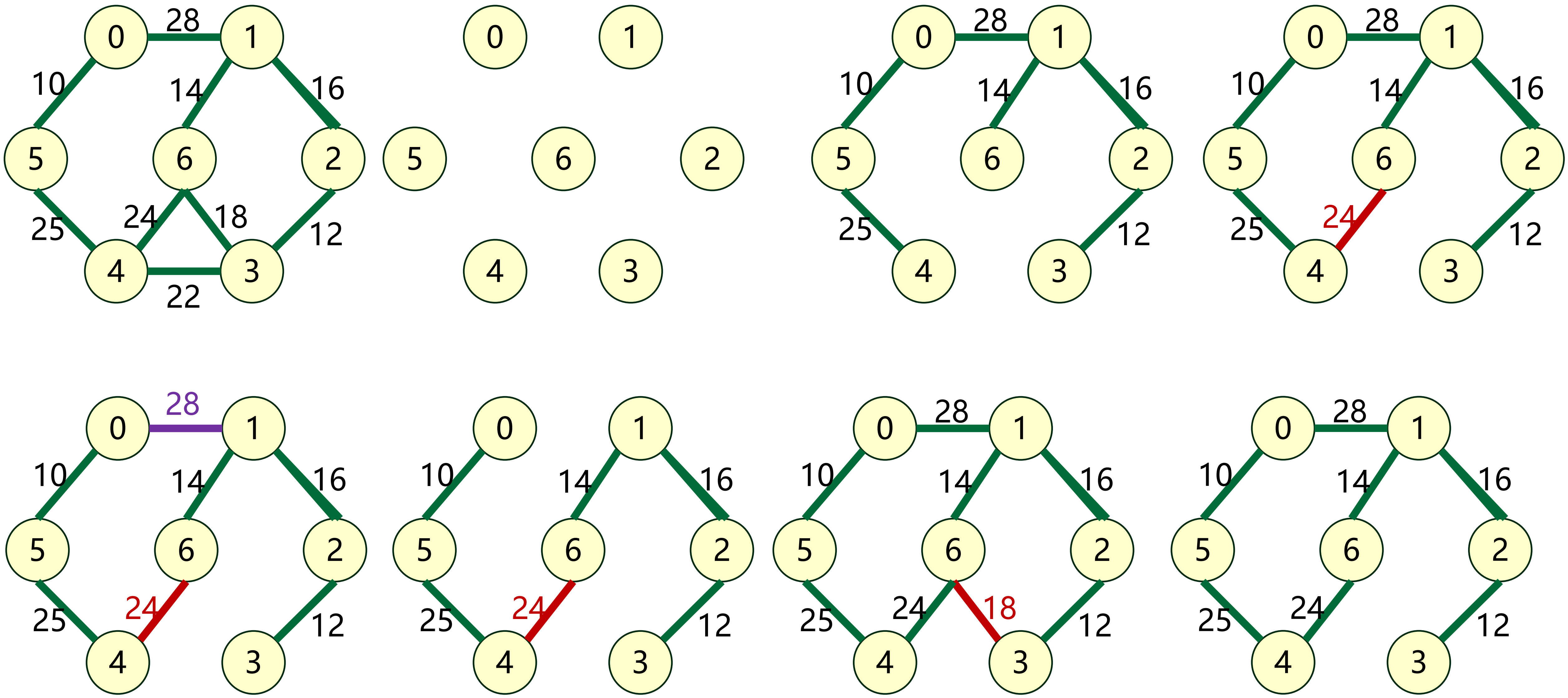
2、然后逐个检查各条边，直到所有边都检查完为止：

　　2.1如果该边的两个端点在不同的连通分量上，直接把它加入生成树；

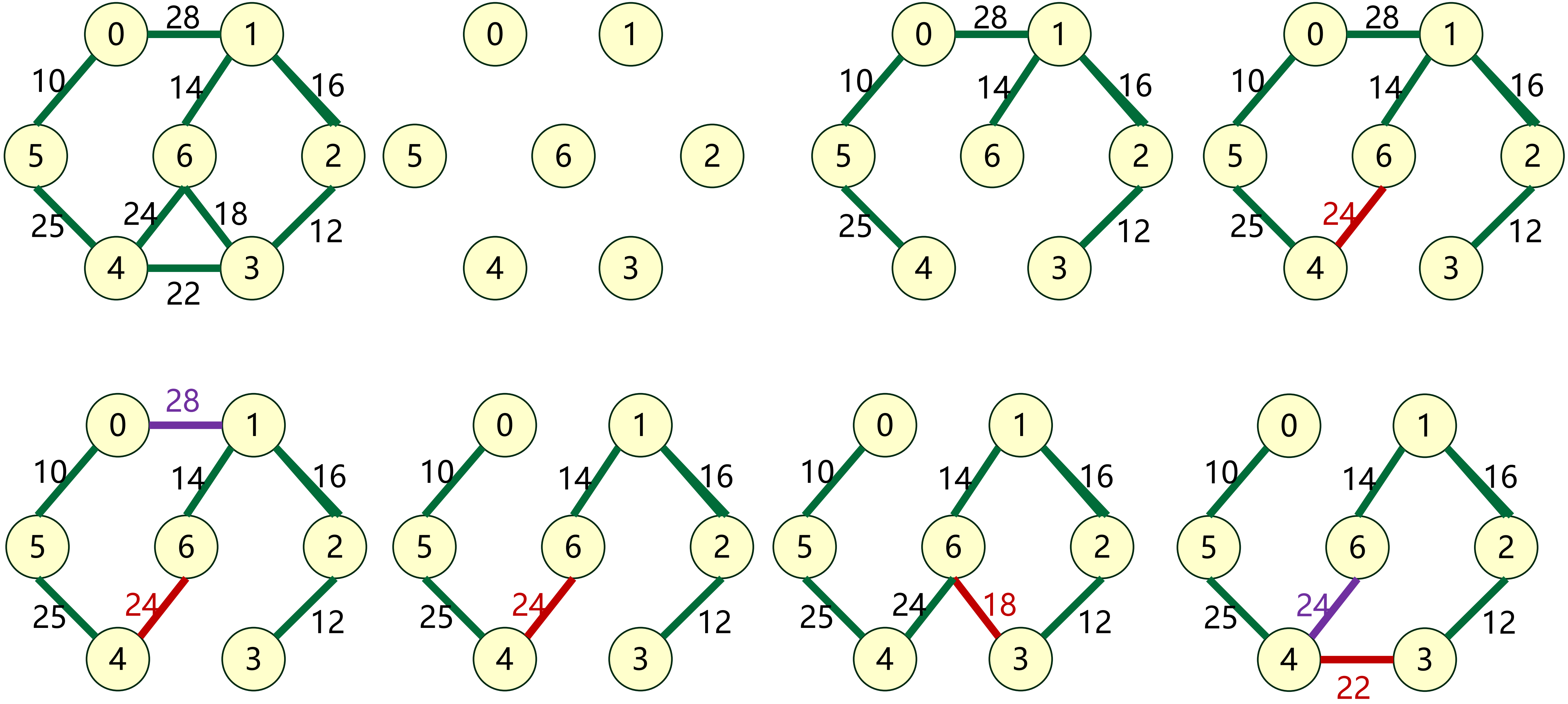
　　2.2如果该边的两个端点在同一个连通分量上，加入它后会形成一个环，把该环上权值最大的边删除。

在此算法中，边的检查顺序没有限制，只要出现圈就破圈。为此又用到了并查集。

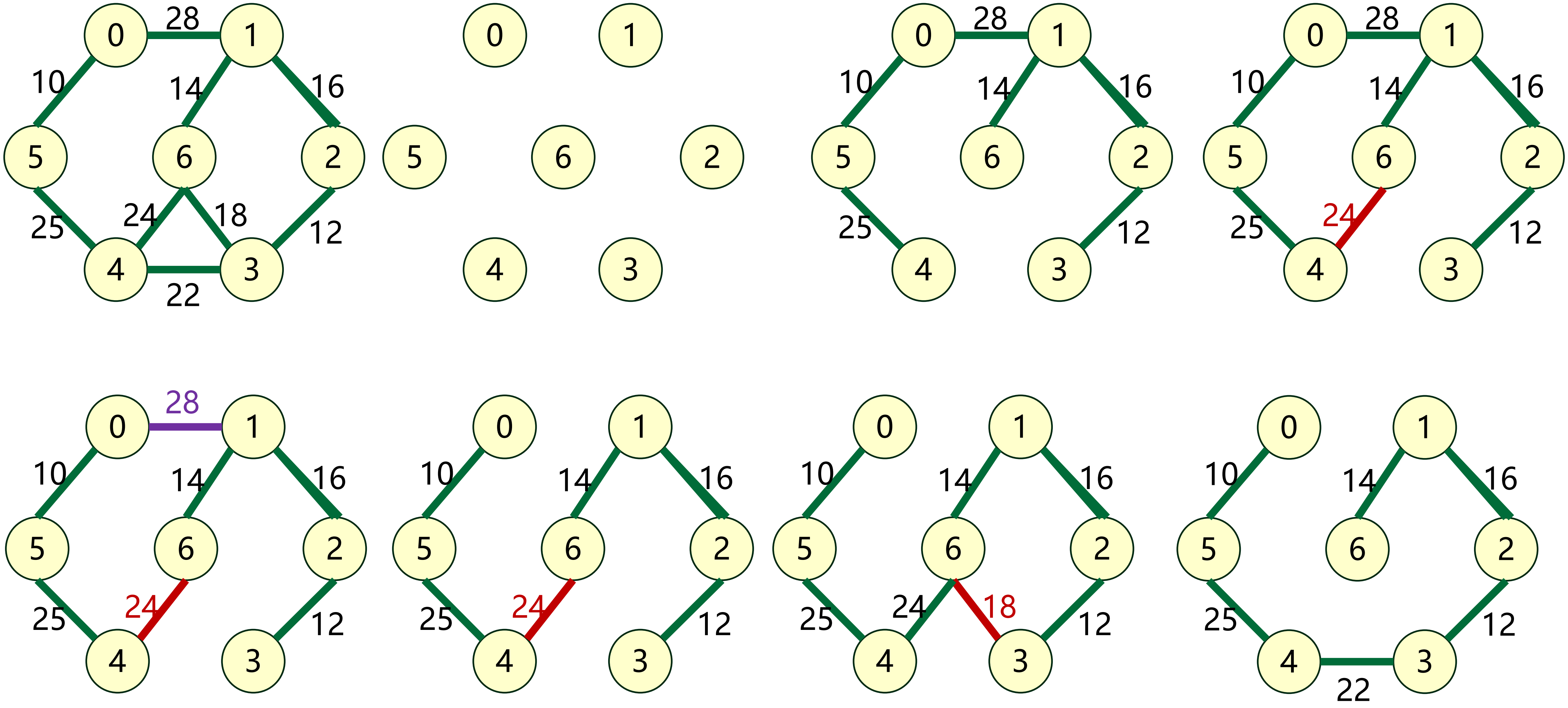
6.4 图的最小生成树-迪杰斯特拉(Dijkstra)算法



6.4 图的最小生成树-迪杰斯特拉(Dijkstra)算法



6.4 图的最小生成树-迪杰斯特拉(Dijkstra)算法



6.4 图的最小生成树

两种算法比较

	普里姆算法	克鲁斯卡尔算法
时间复杂度	$O(n^2)$	$O(eloge)$
适应范围	稠密图	稀疏图

6.5 有向无环图——拓扑排序

问题提出：学生选修课程问题

顶点——表示课程

有向弧——表示先决条件，若 课程i 是 课程j 的先决条件，则图中有弧<i,j>。

学生应按怎样的顺序学习这些课程，才能无矛盾、顺利地完成学业——拓扑排序。

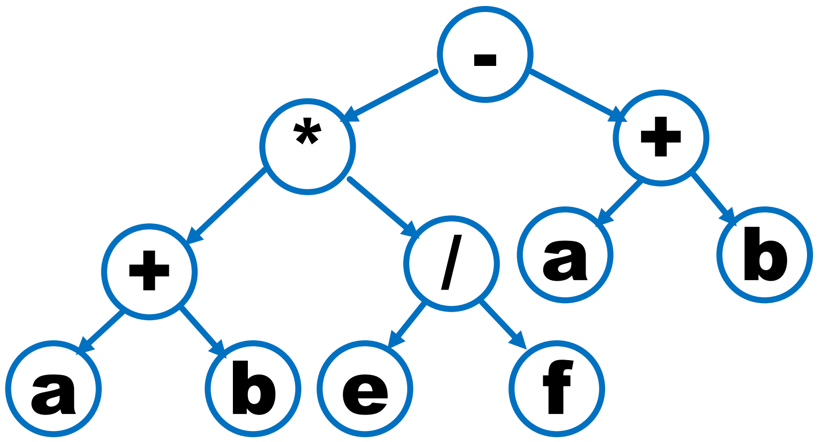
6.5 有向无环图——拓扑排序

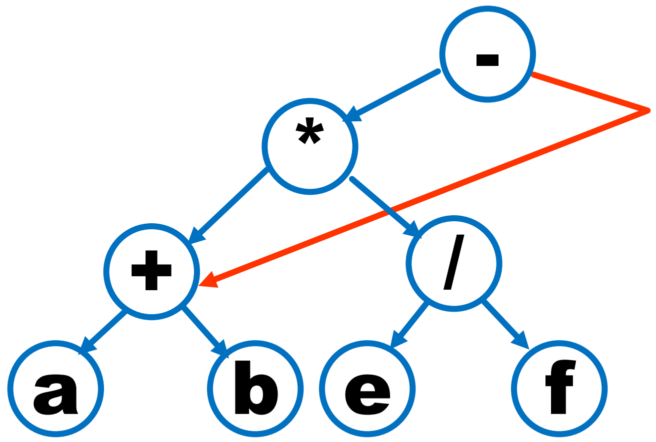
有向无环图(DAG)

没有回路的有向图。

含有公共子式的表达式

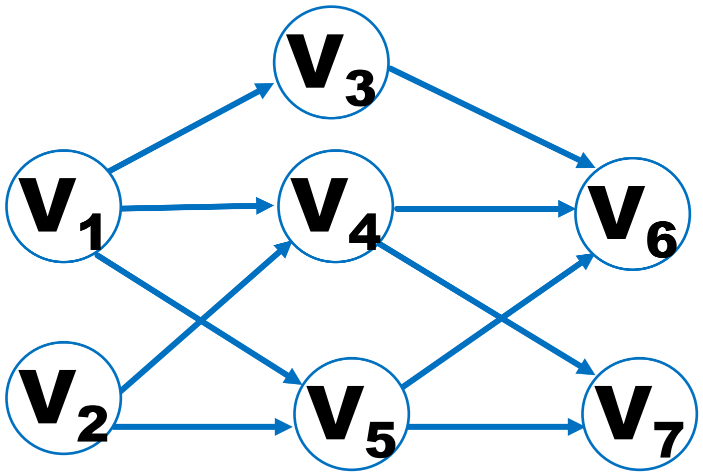
(a + b) * (e / f) - (a + b)





有向无环图(DAG)

某工程可分为7个子工程，工程流程图。



6.5 有向无环图——拓扑排序

定义

AOV网——用顶点表示活动，用弧表示活动间优先关系的有向图称为顶点表示活动的网（Activity On Vertex network），简称AOV网。

若 $<v_i,v_j>$ 是图中有向边，则 v_i 是 v_j 的直接前驱；v_j 是 v_i 的直接后继。

AOV网中不允许有回路，这意味着某项活动以自己为先决条件。

拓扑排序

把AOV网络中各顶点按照它们相互之间的优先关系排列成一个线性序列的过程。

检测AOV网中是否存在环方法：对有向图构造其顶点的拓扑有序序列，若网中所有顶点都在它的拓扑有序序列中，则该AOV网必定不存在环。

6.5 有向无环图——拓扑排序

拓扑排序的方法

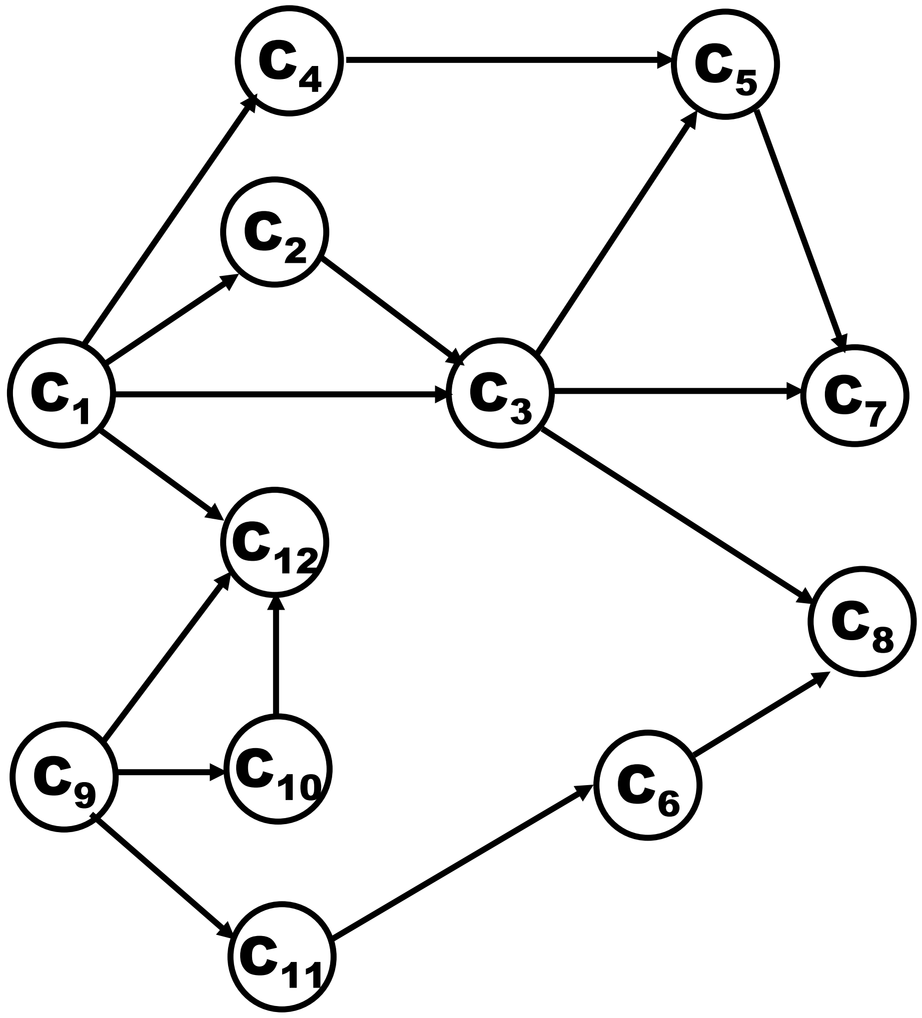
在有向图中选一个没有前驱的顶点且输出。

从图中删除该顶点和所有以它为尾的弧。

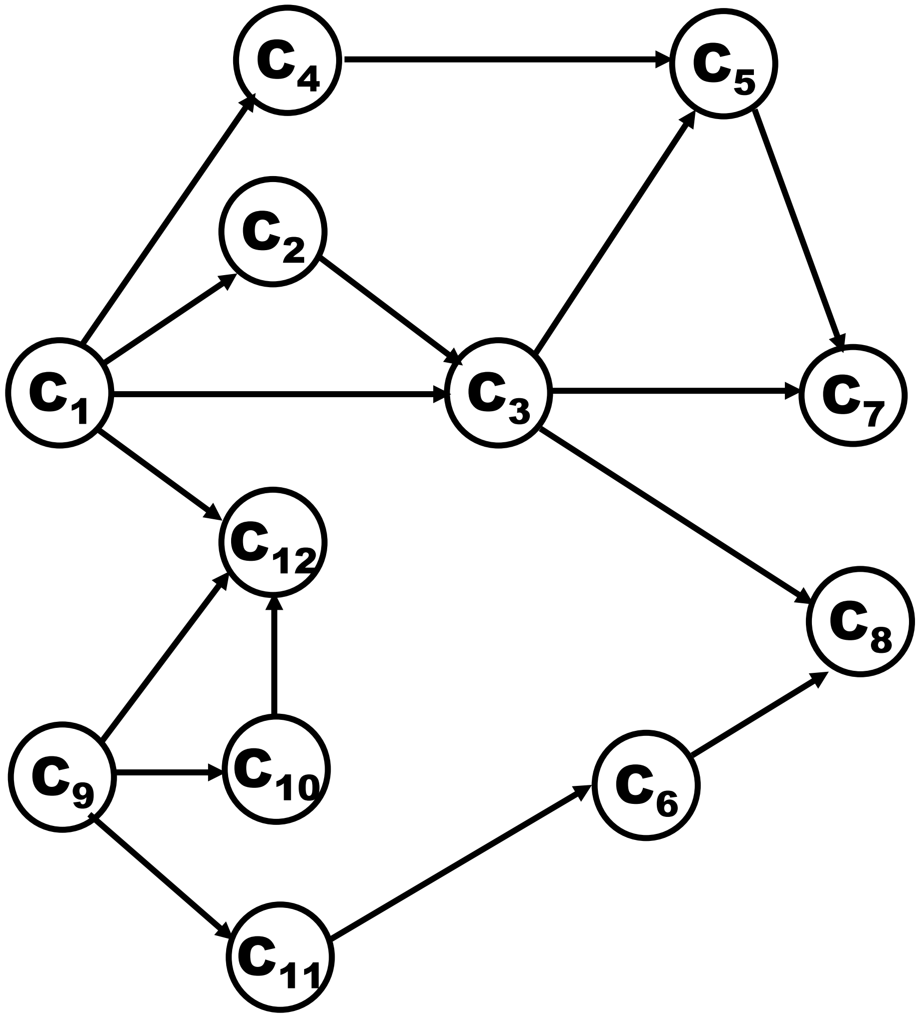
重复上述两步，直至全部顶点均已输出；或者当图中不存在无前驱的顶点为止。

6.5 有向无环图——拓扑排序

课程代号	课程名称	先修课
C_1	程序设计基础	无
C_2	离散数学	C_1
C_3	数据结构	C_1和C_2
C_4	汇编语言	C_1
C_5	语言的设计和分析	C_3和C_4
C_6	计算机原理	C_{11}
C_7	编译原理	C_3和C_5
C_8	操作系统	C_3和C_6
C_9	高等数学	无
C_{10}	线性代数	C_9
C_{11}	普通物理	C_9
C_{12}	数值分析	C_1和C_9和C_{10}



6.5 有向无环图——拓扑排序

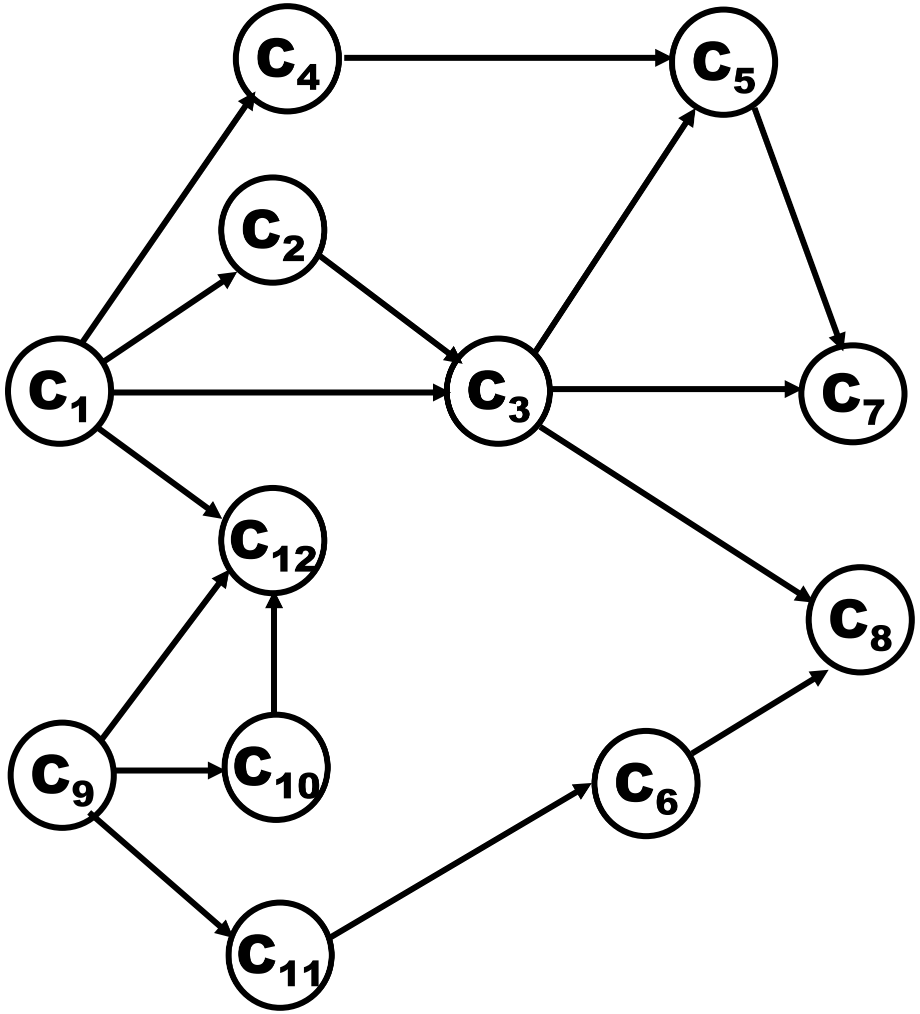


拓扑序列： C_1--C_2--C_3--C_4--C_5--C_7--C_9--C_{10}--C_{11}--C_6--C_{12}--C_8

或： C_9--C_{10}--C_{11}--C_6--C_1--C_{12}--C_4--C_2--C_3--C_5--C_7--C_8

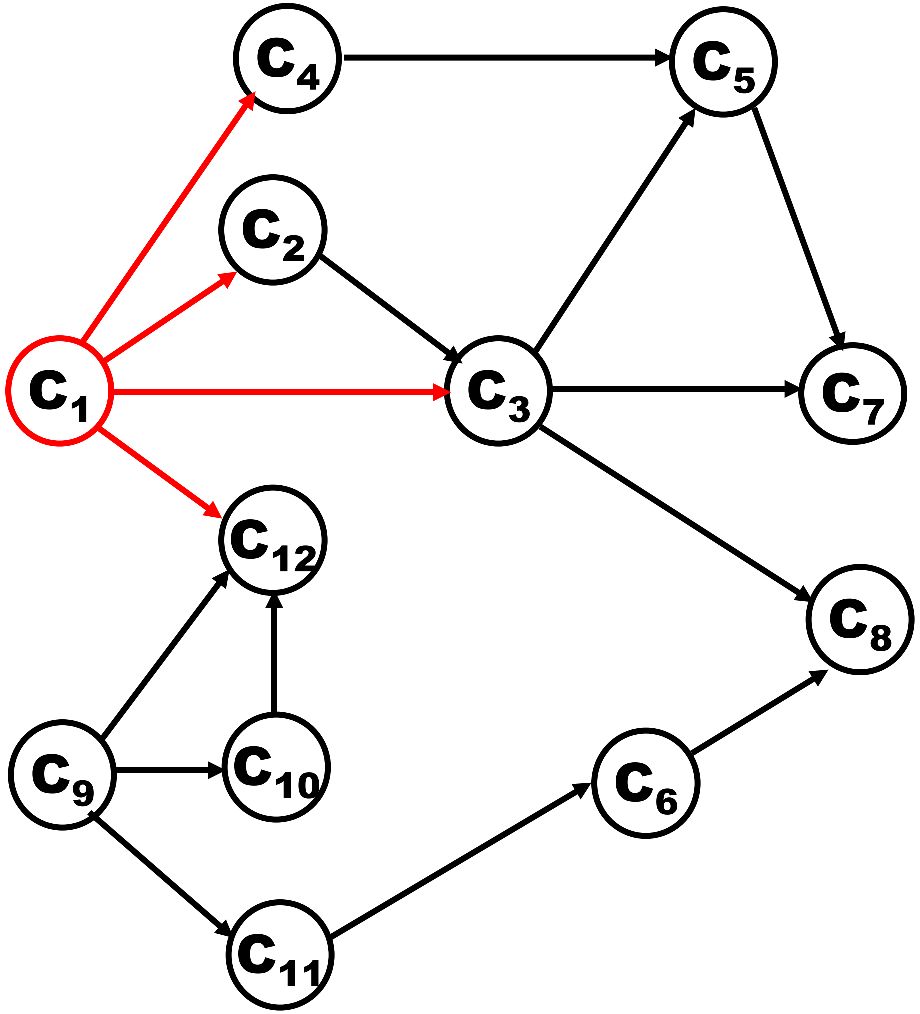
一个AOV网的拓扑序列不是唯一的

6.5 有向无环图——拓扑排序



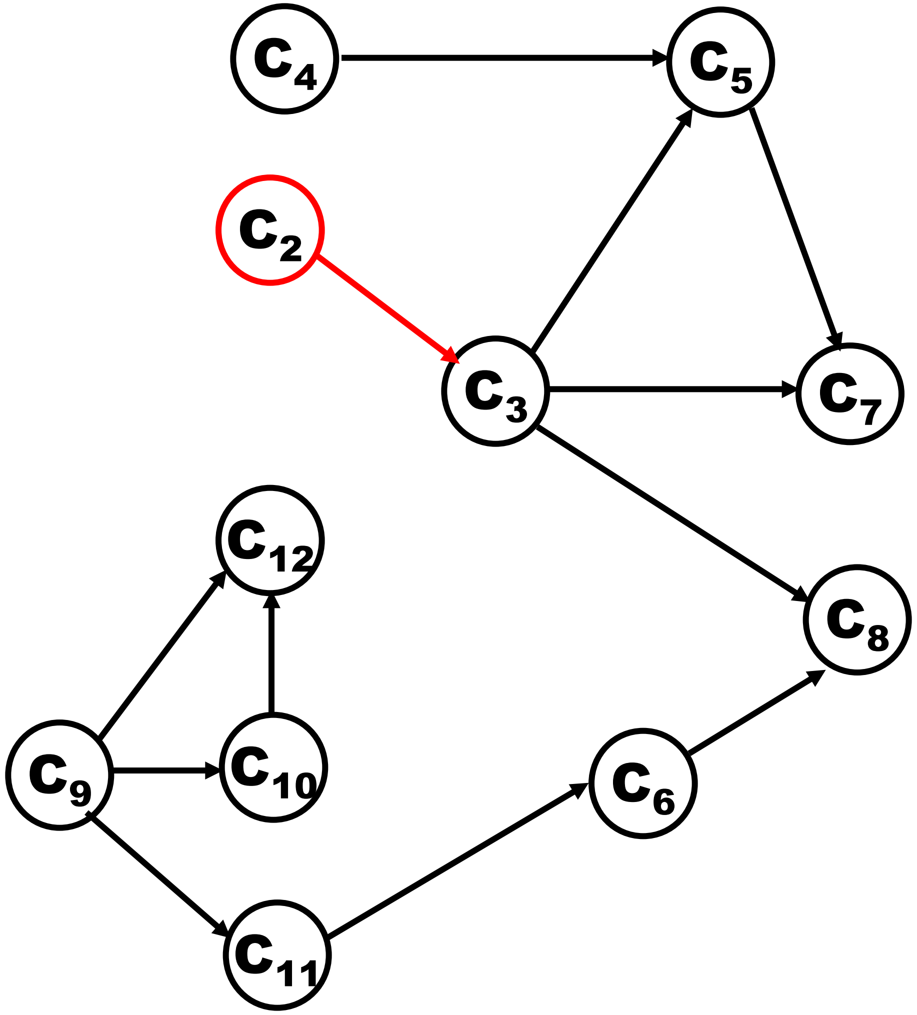
拓扑序列：C_1

6.5 有向无环图——拓扑排序



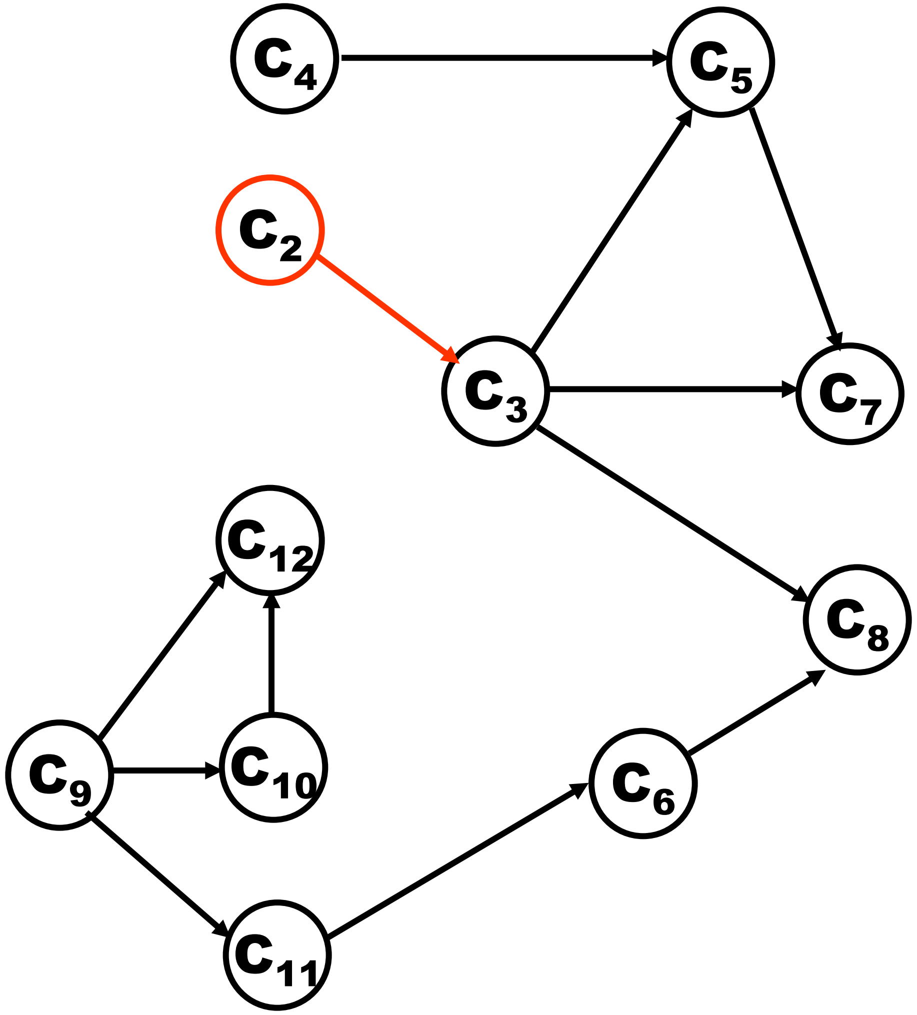
拓扑序列：c_1

6.5 有向无环图——拓扑排序



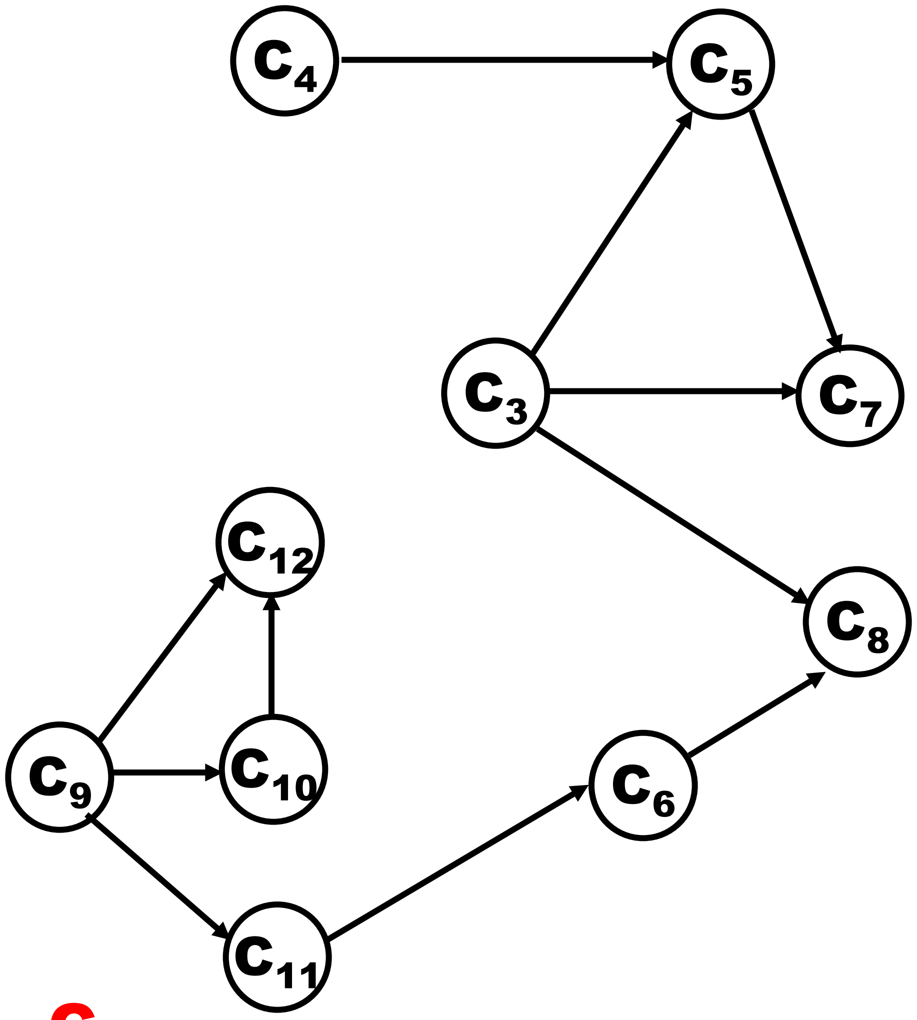
拓扑序列：C_1 --C_2

6.5 有向无环图——拓扑排序



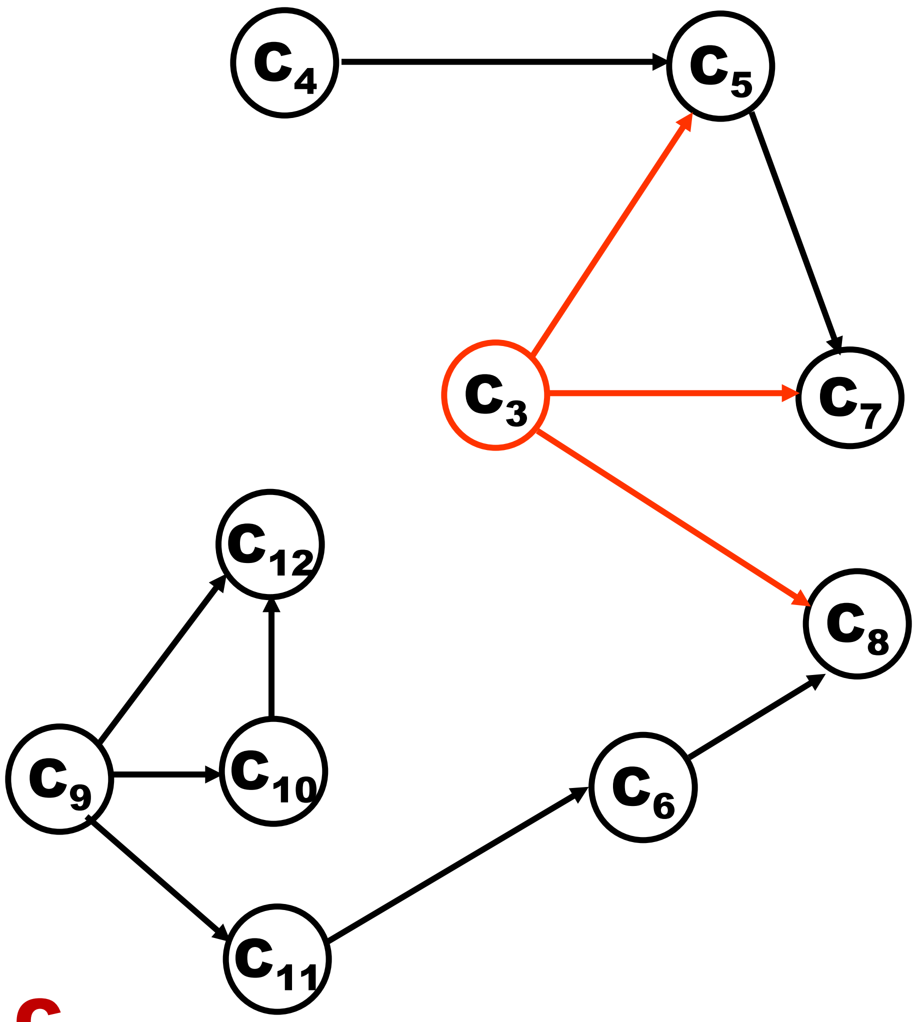
拓扑序列：c_1 --c_2

6.5 有向无环图——拓扑排序



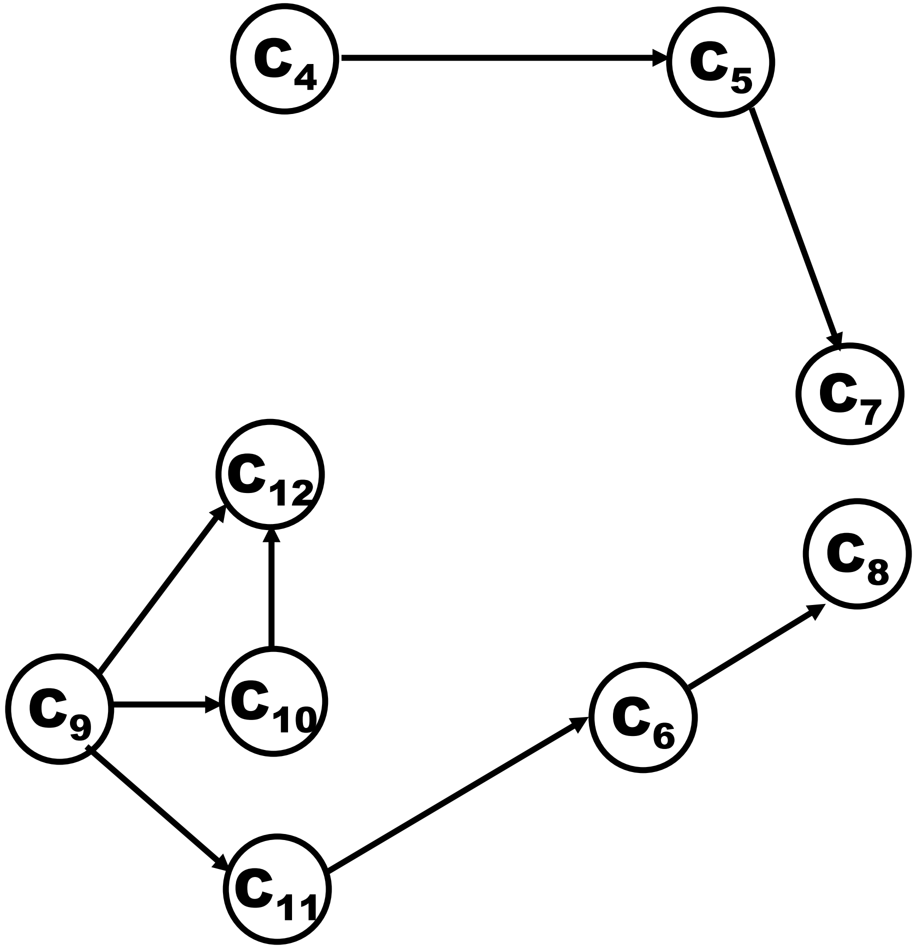
拓扑序列：C_1 --C_2 --C_3

6.5 有向无环图——拓扑排序



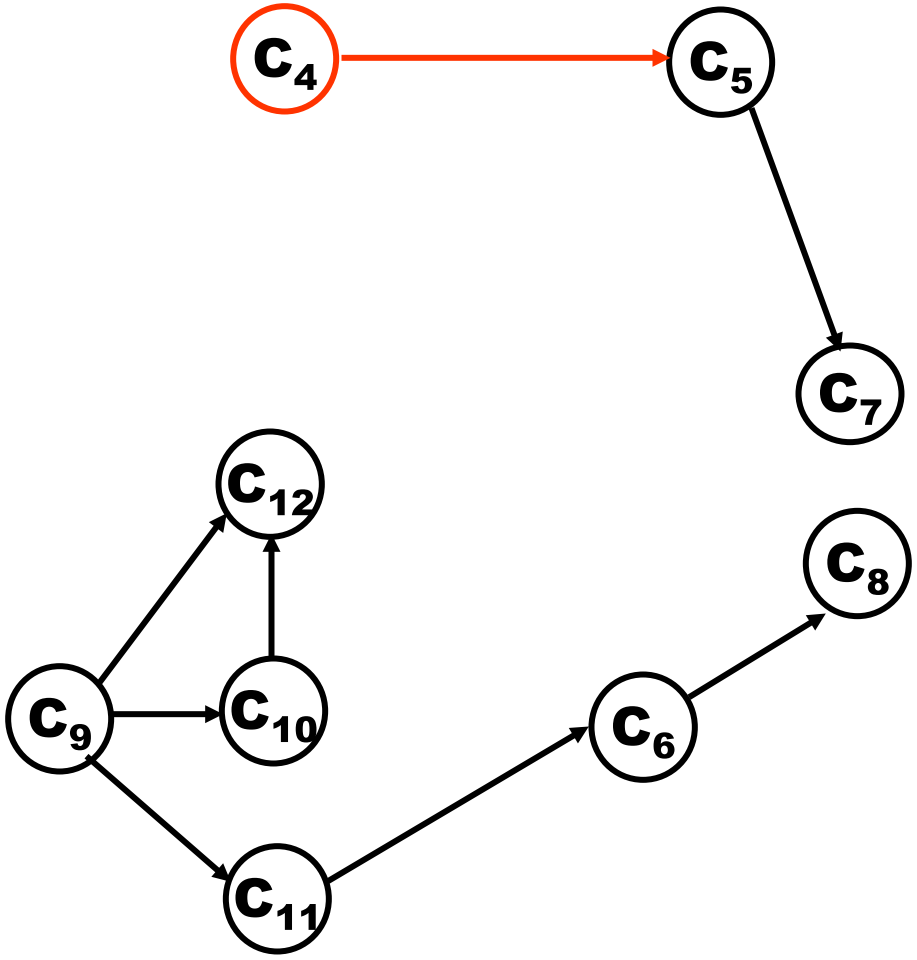
拓扑序列：c_1 --c_2 --c_3

6.5 有向无环图——拓扑排序



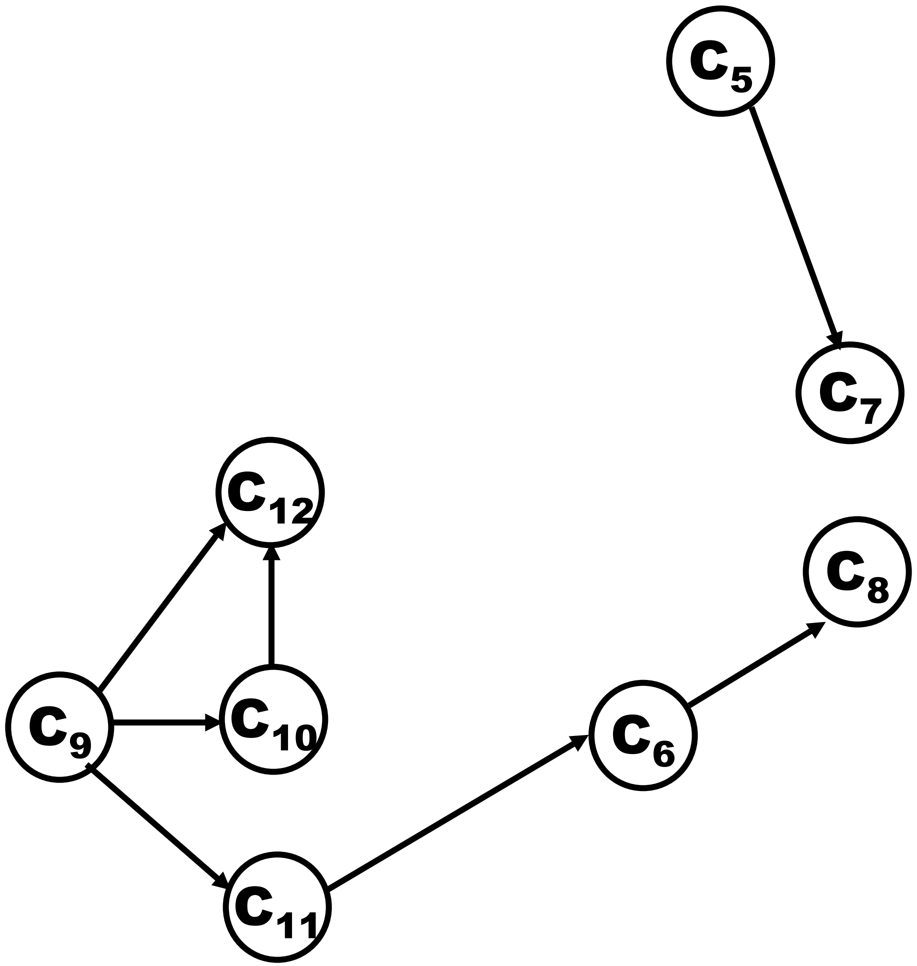
拓扑序列：C_1 --C_2 --C_3 --C_4

6.5 有向无环图——拓扑排序



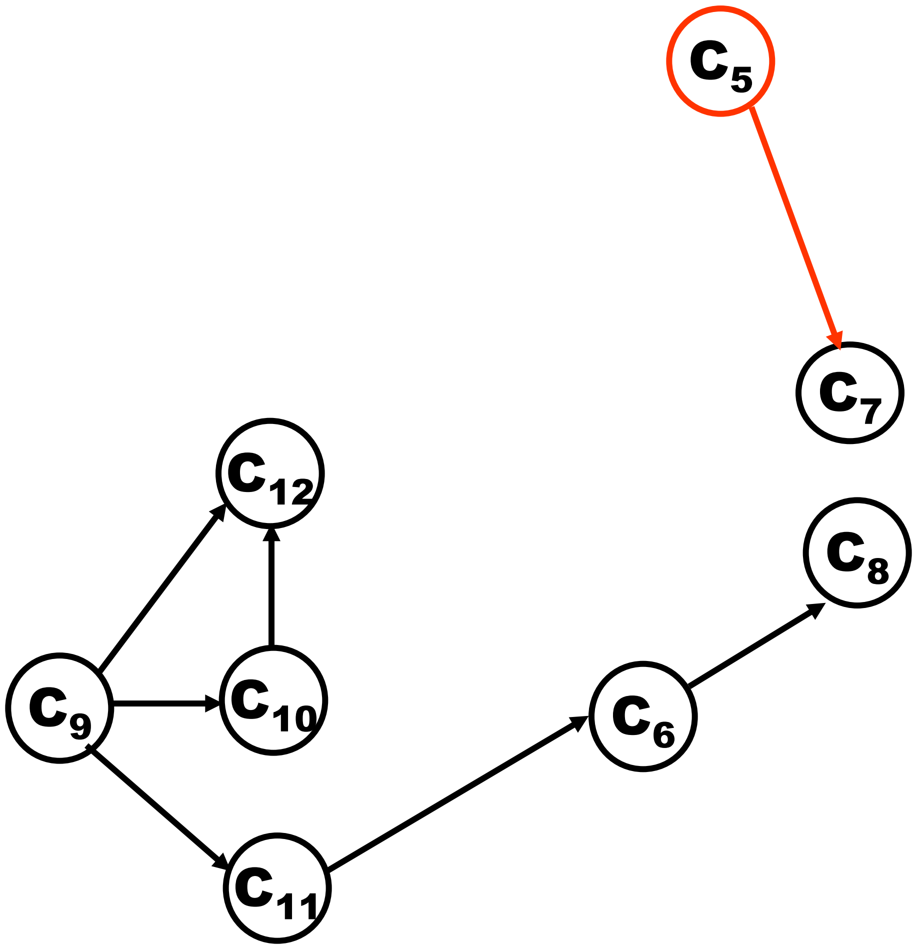
拓扑序列：c_1 --c_2 --c_3 --c_4

6.5 有向无环图——拓扑排序



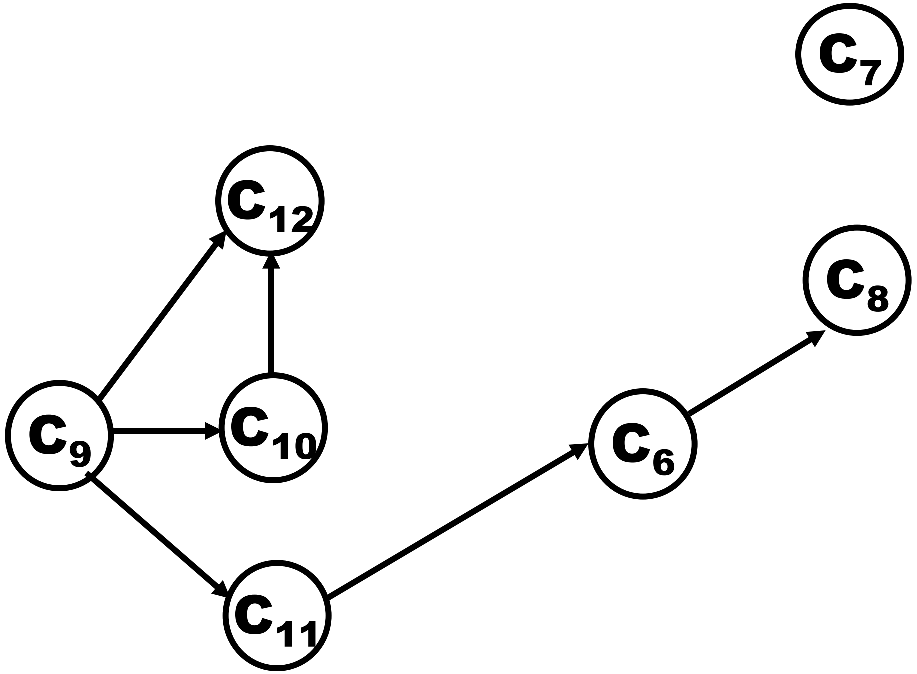
拓扑序列：c_1 --c_2 --c_3 --c_4 --c_5

6.5 有向无环图——拓扑排序



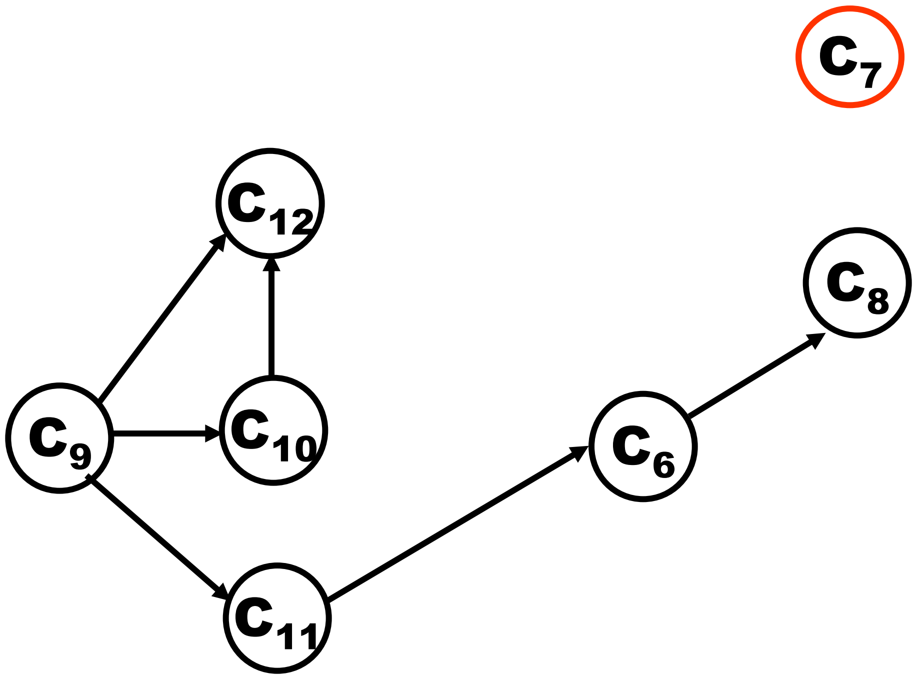
拓扑序列：c_1 --c_2 --c_3 --c_4 --c_5

6.5 有向无环图——拓扑排序



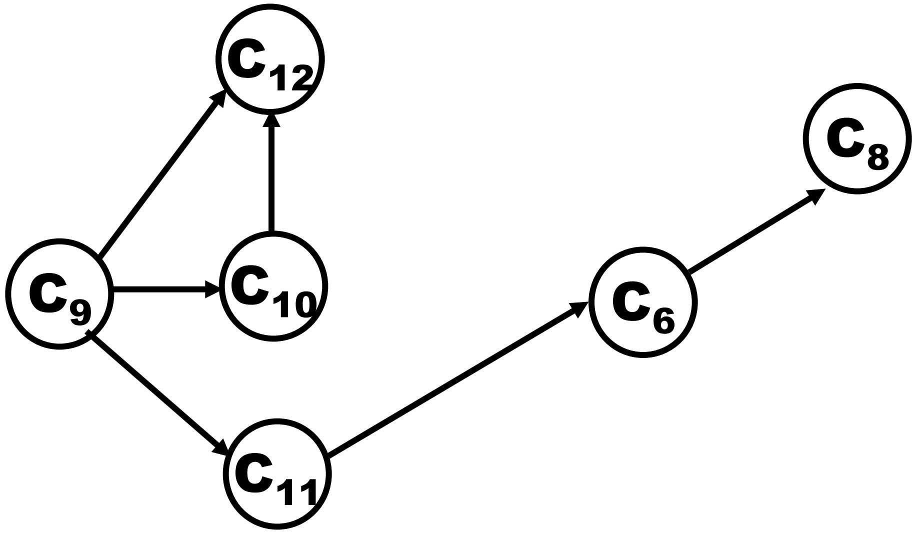
拓扑序列：C_1 --C_2 --C_3 --C_4 --C_5 --C_7

6.5 有向无环图——拓扑排序



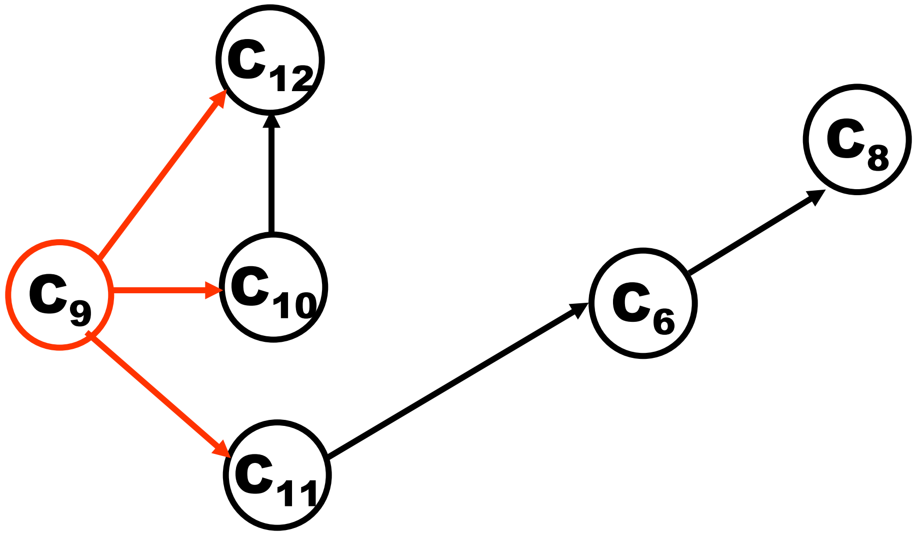
拓扑序列：c_1 --c_2 --c_3 --c_4 --c_5 --c_7

6.5 有向无环图——拓扑排序



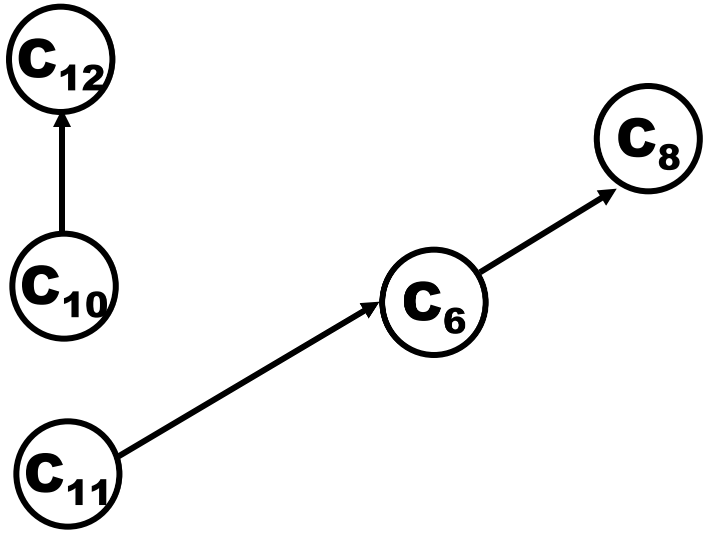
拓扑序列：C_1 --C_2 --C_3 --C_4 --C_5 --C_7 --C_9

6.5 有向无环图——拓扑排序



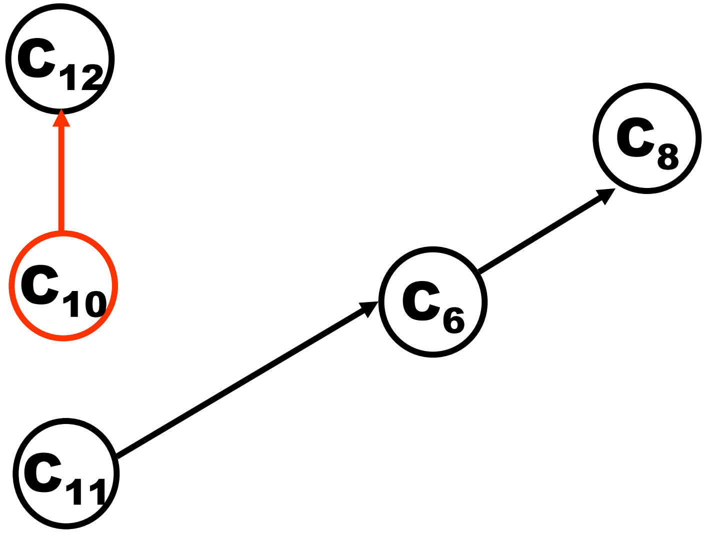
拓扑序列：C_1 --C_2 --C_3 --C_4 --C_5 --C_7 --C_9

6.5 有向无环图——拓扑排序



拓扑序列：C_1 --C_2 --C_3 --C_4 --C_5 --C_7 --C_9 --C_{10}

6.5 有向无环图——拓扑排序



拓扑序列：C_1 --C_2 --C_3 --C_4 --C_5 --C_7 --C_9 --C_{10} --C_{11} --C_6 --C_{12} --C_8

6.5 有向无环图——拓扑排序

拓扑排序算法实现

以邻接表作存储结构。

把邻接表中所有入度为0的顶点进栈。

栈非空时，输出栈顶元素 V_j 并退栈；在邻接表中查找 V_j 的直接后继 V_k，把 V_k 的入度 减1；若 V_k 的入度为 0 则进栈。

重复上述操作直至栈空为止。

若栈空时输出的顶点个数不是 n，则有向图有环；

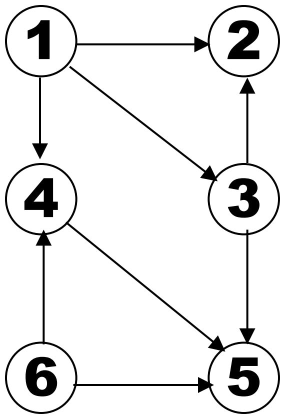
否则，拓扑排序完毕。

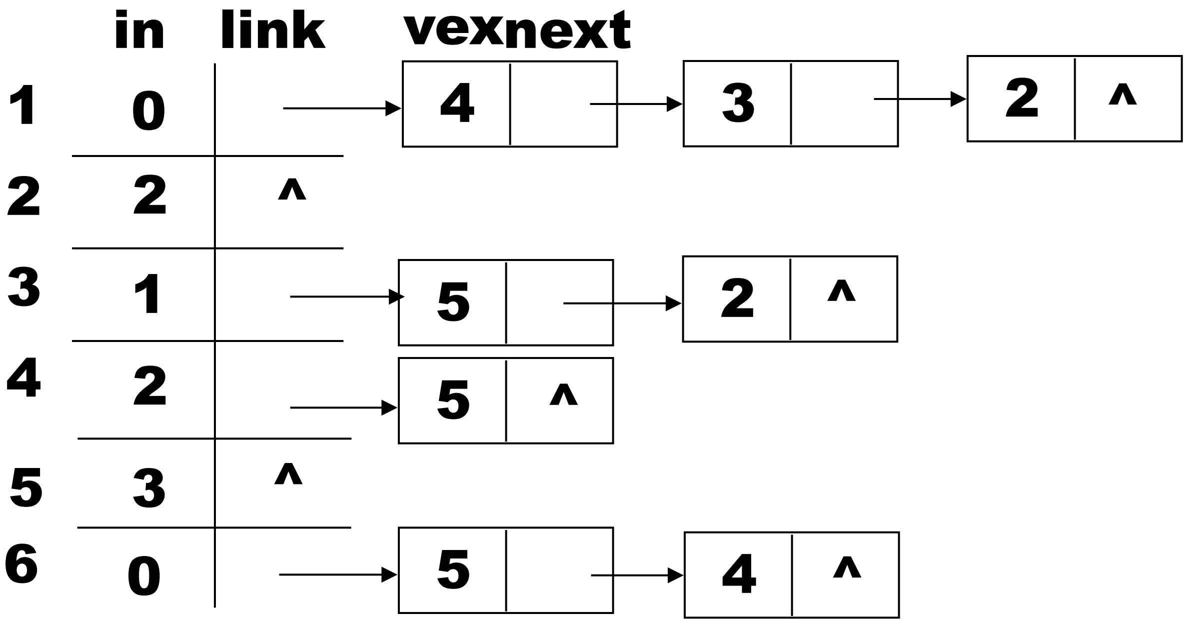
6.5 有向无环图——拓扑排序

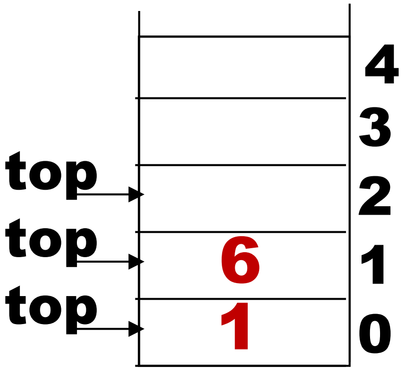
```
Status ToplogicalSort(ALGraph G)
//有向图采用邻接图存储结构
//若G无回路，则输出G的顶点的一个拓扑序列并返回OK，否则返回Error
FindInDegree（G，indegree）；
InitStack(S);
for(i=0;i<G.vexnum;i++) if (!indegree[i]) Push(S,i);
count=0;
while(!StackEmpty(S)){
    Pop(S,i); printf(i,G.vertices[i].data);  count++;
    for(p=G.vertices[i].firstarc; p; p=p->nextarc){
        k=p->adjvex;
        if (!(--indegree[k]) Push(S,k);
    }
}
if (count <G.vexnum) return Error;
else return OK;
```

6.5 有向无环图——拓扑排序

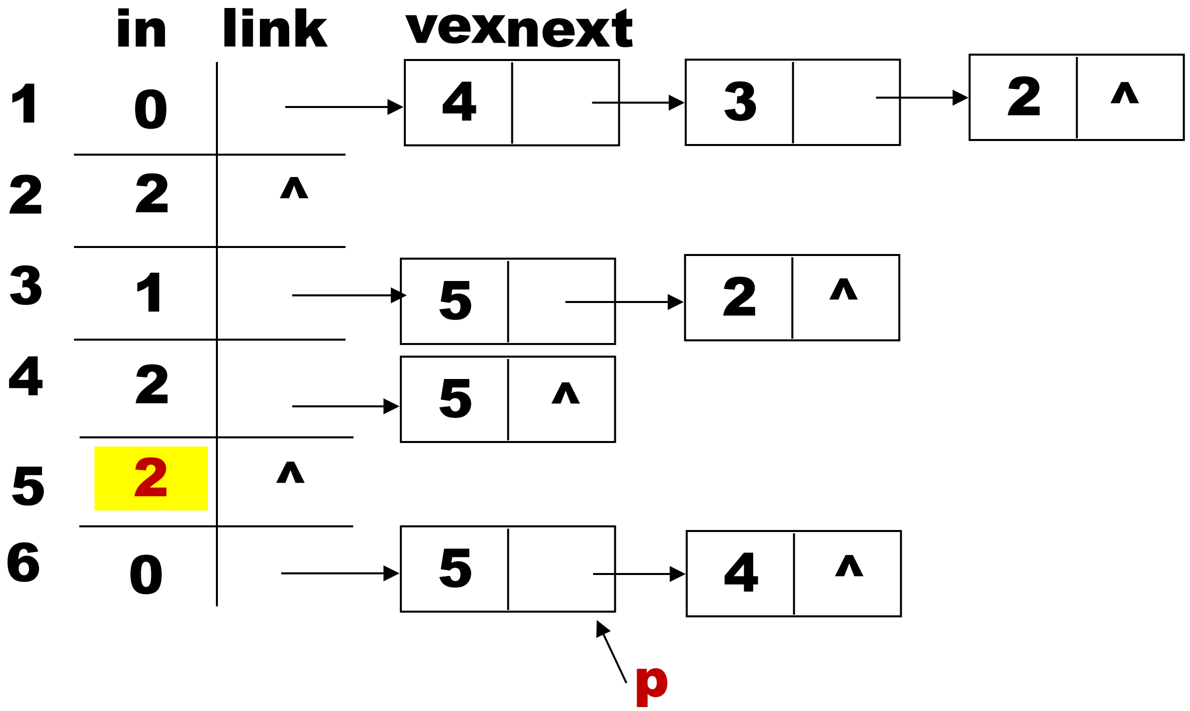
例

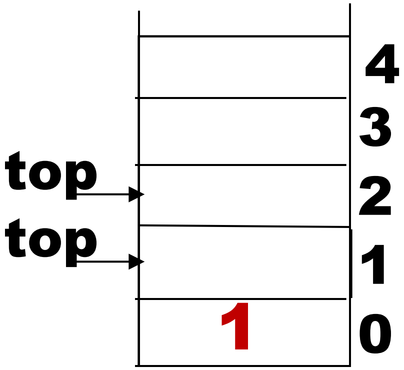






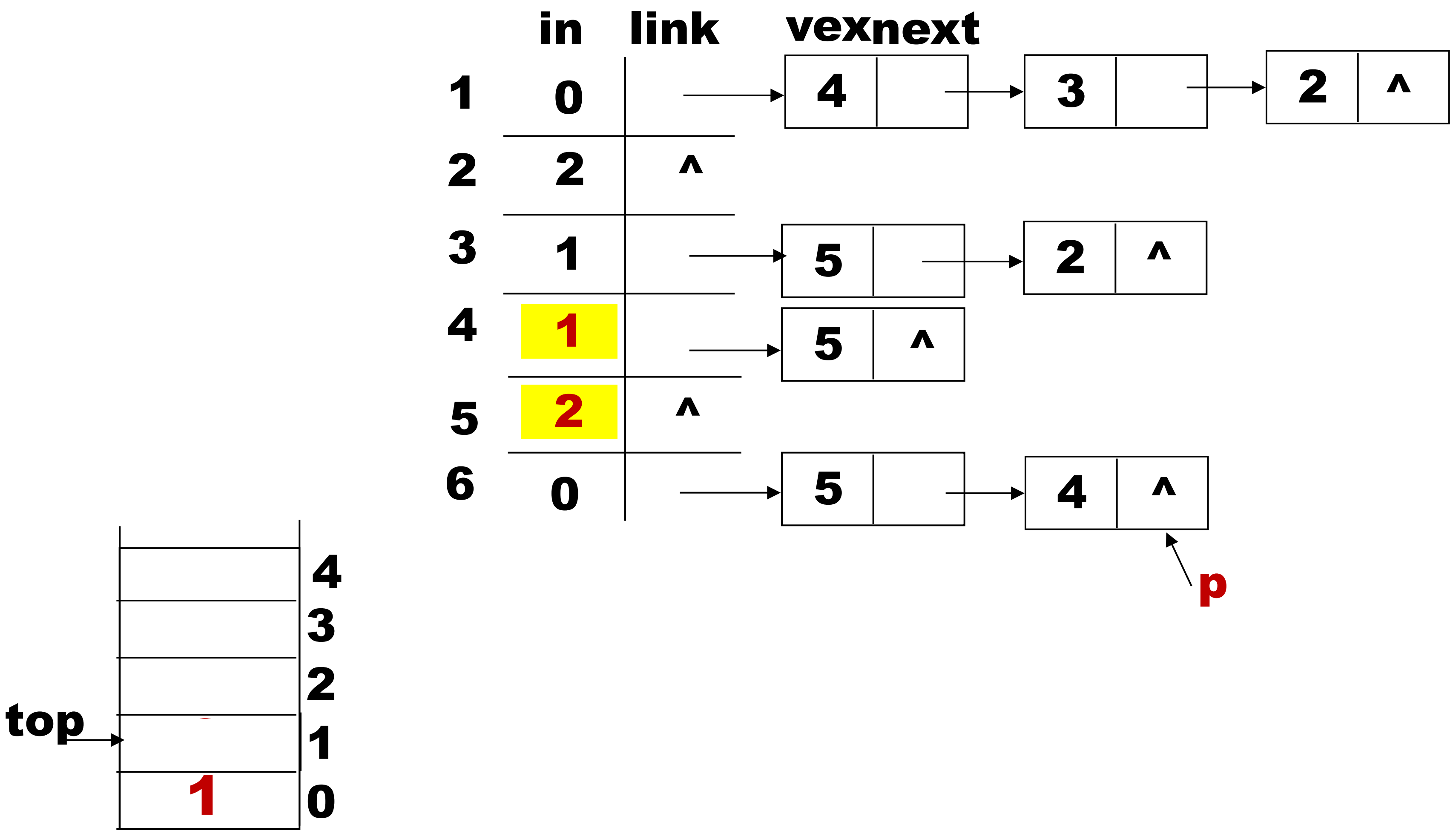
6.5 有向无环图——拓扑排序





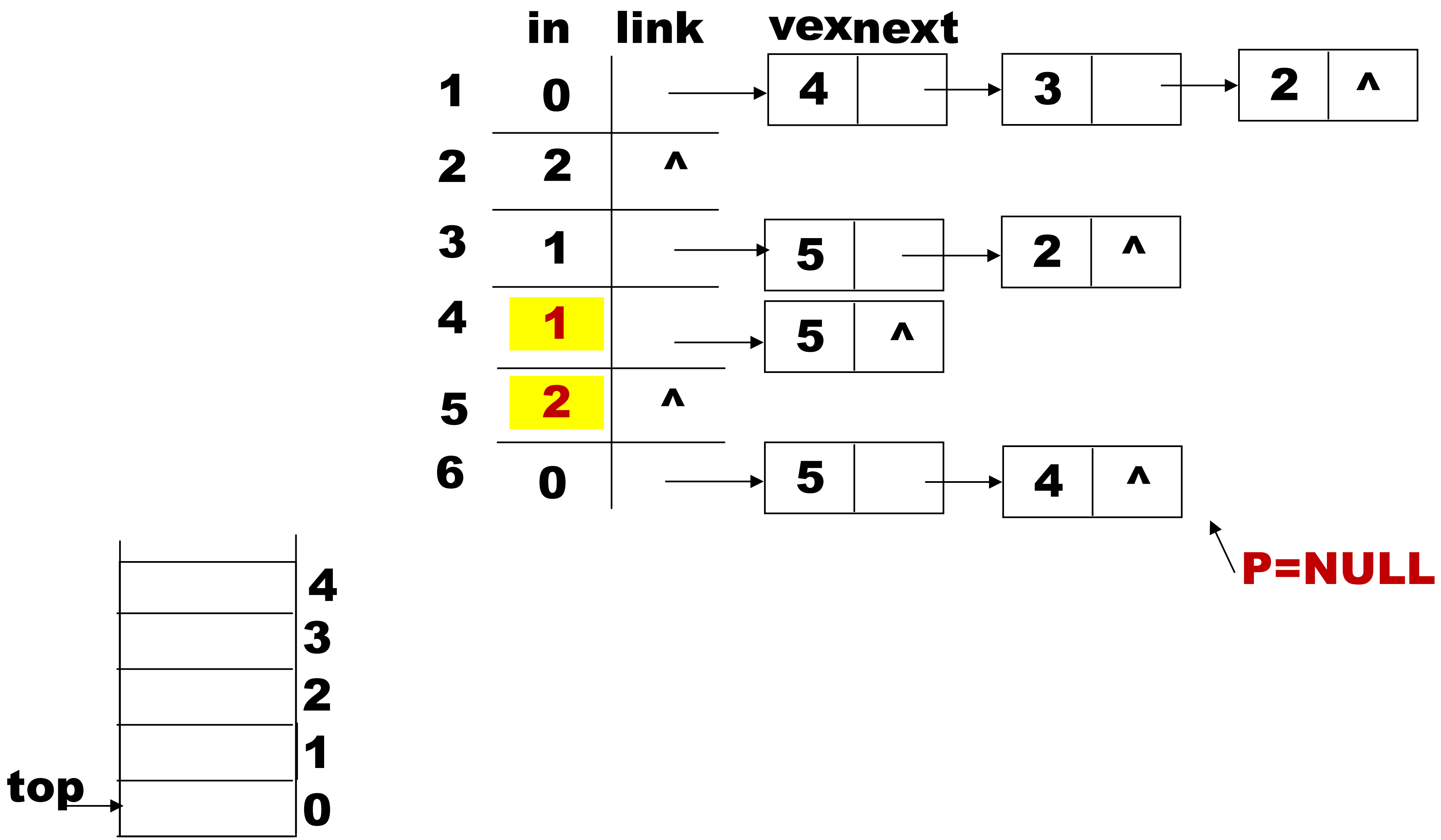
输出序列：6

6.5 有向无环图——拓扑排序



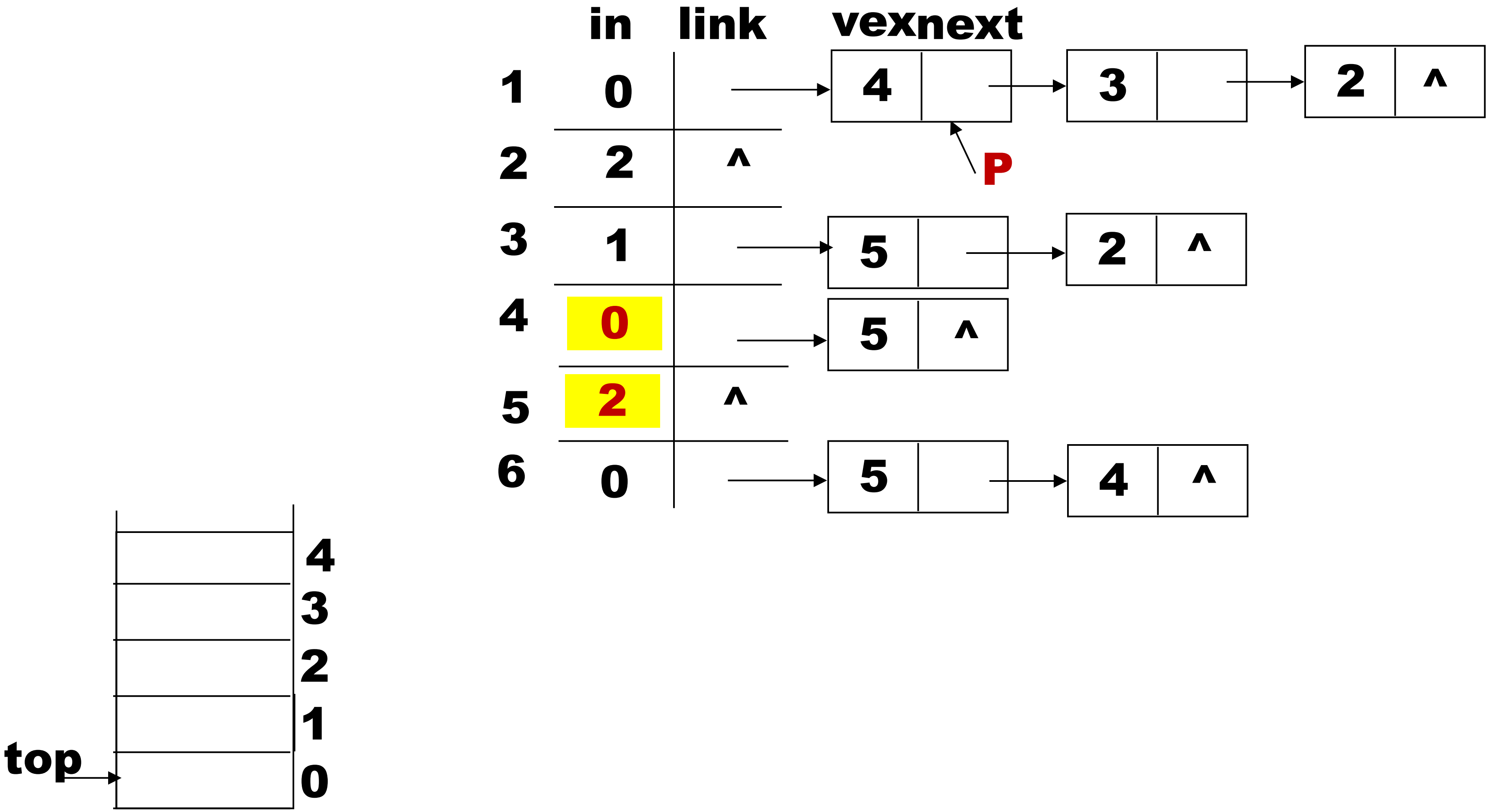
输出序列：6

6.5 有向无环图——拓扑排序



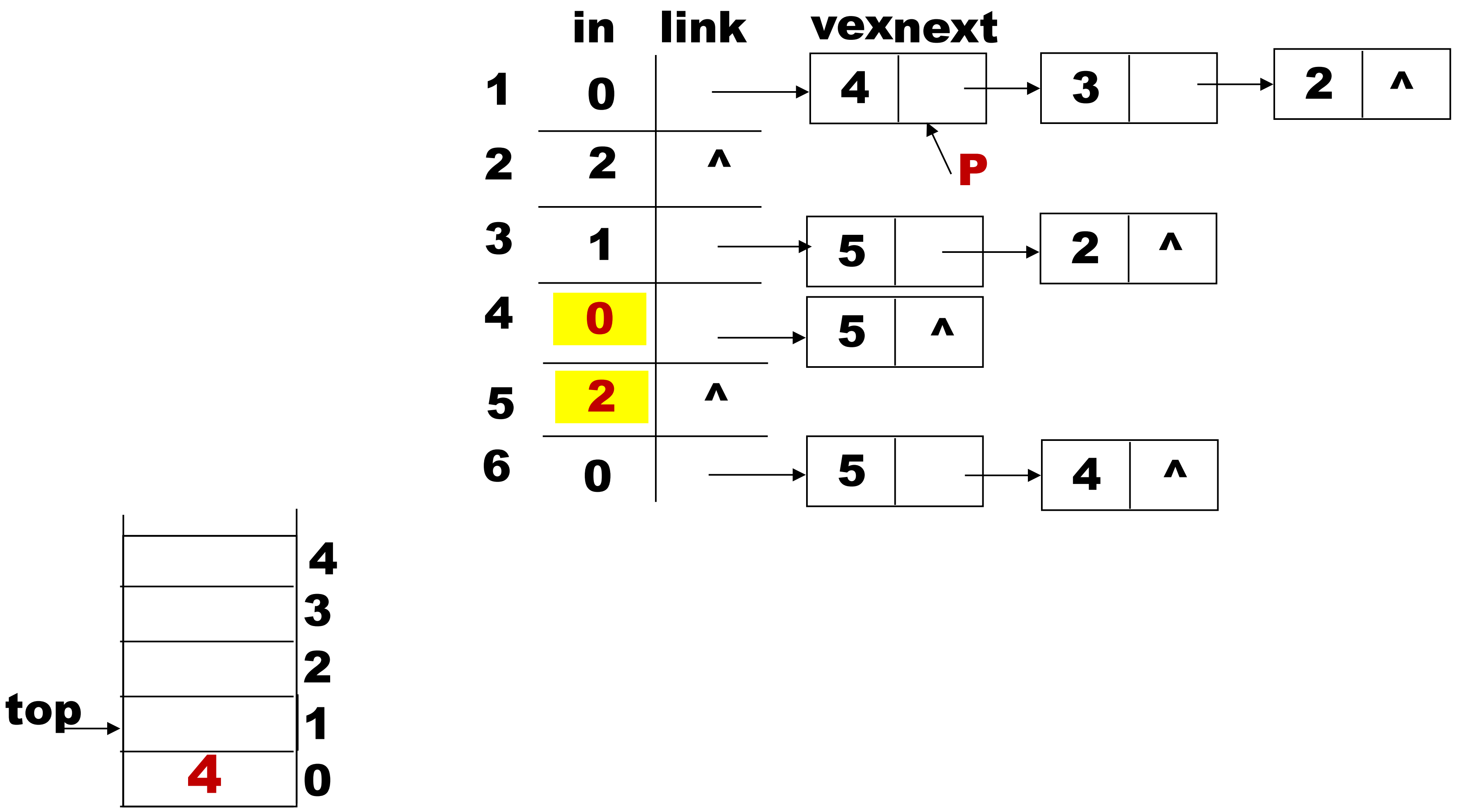
输出序列：6 1

6.5 有向无环图——拓扑排序



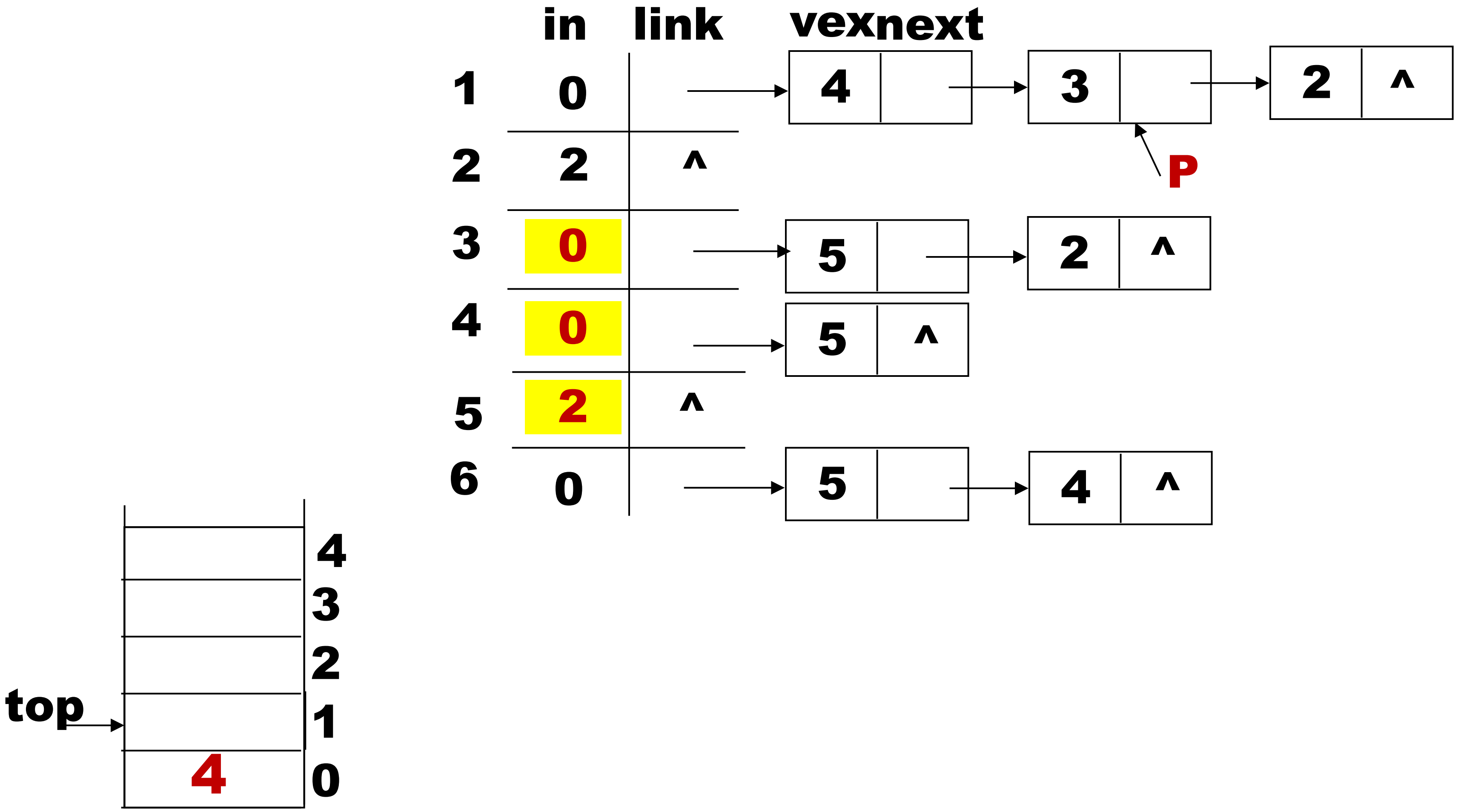
输出序列：6 1

6.5 有向无环图——拓扑排序



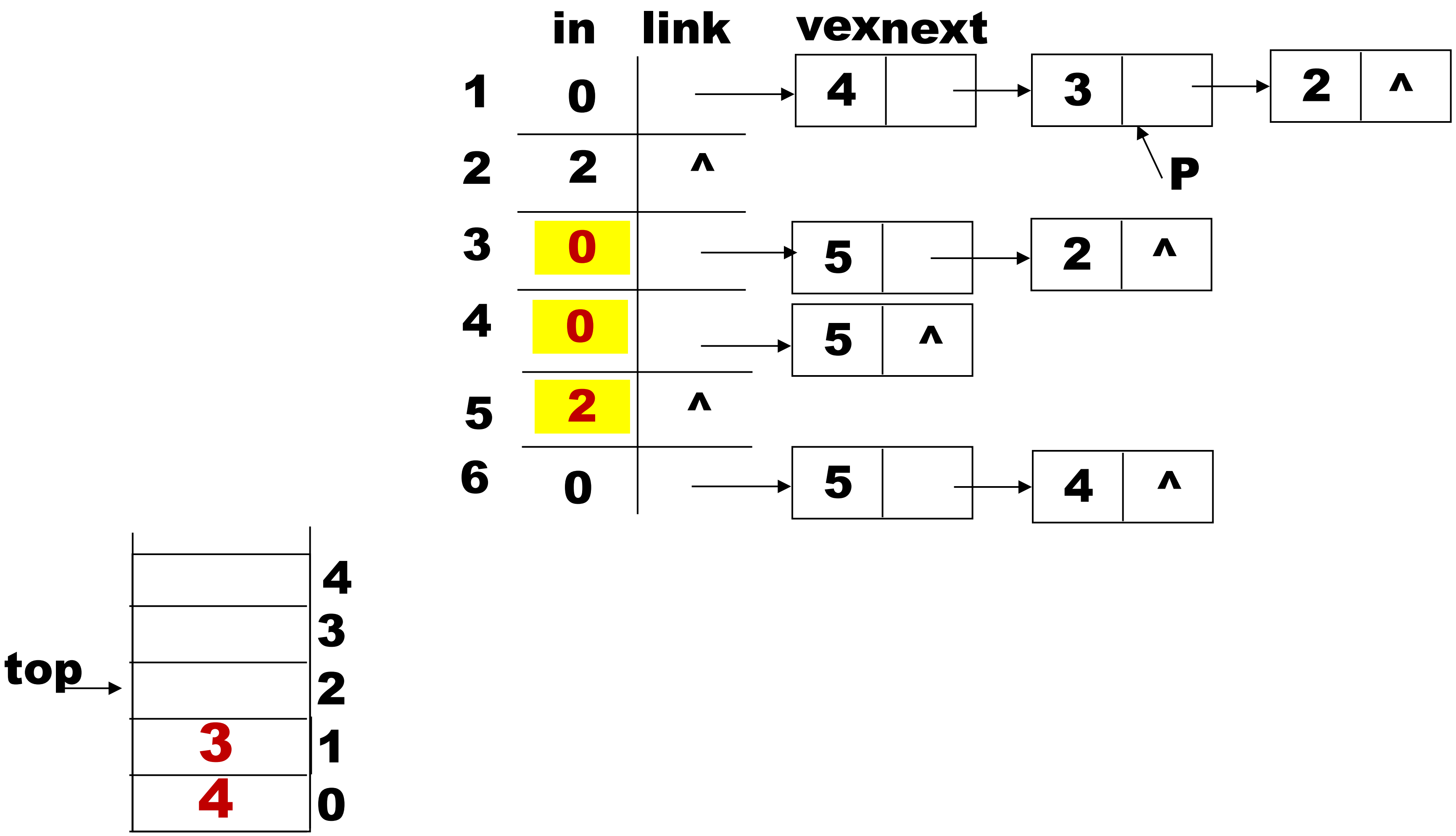
输出序列：6 1

6.5 有向无环图——拓扑排序



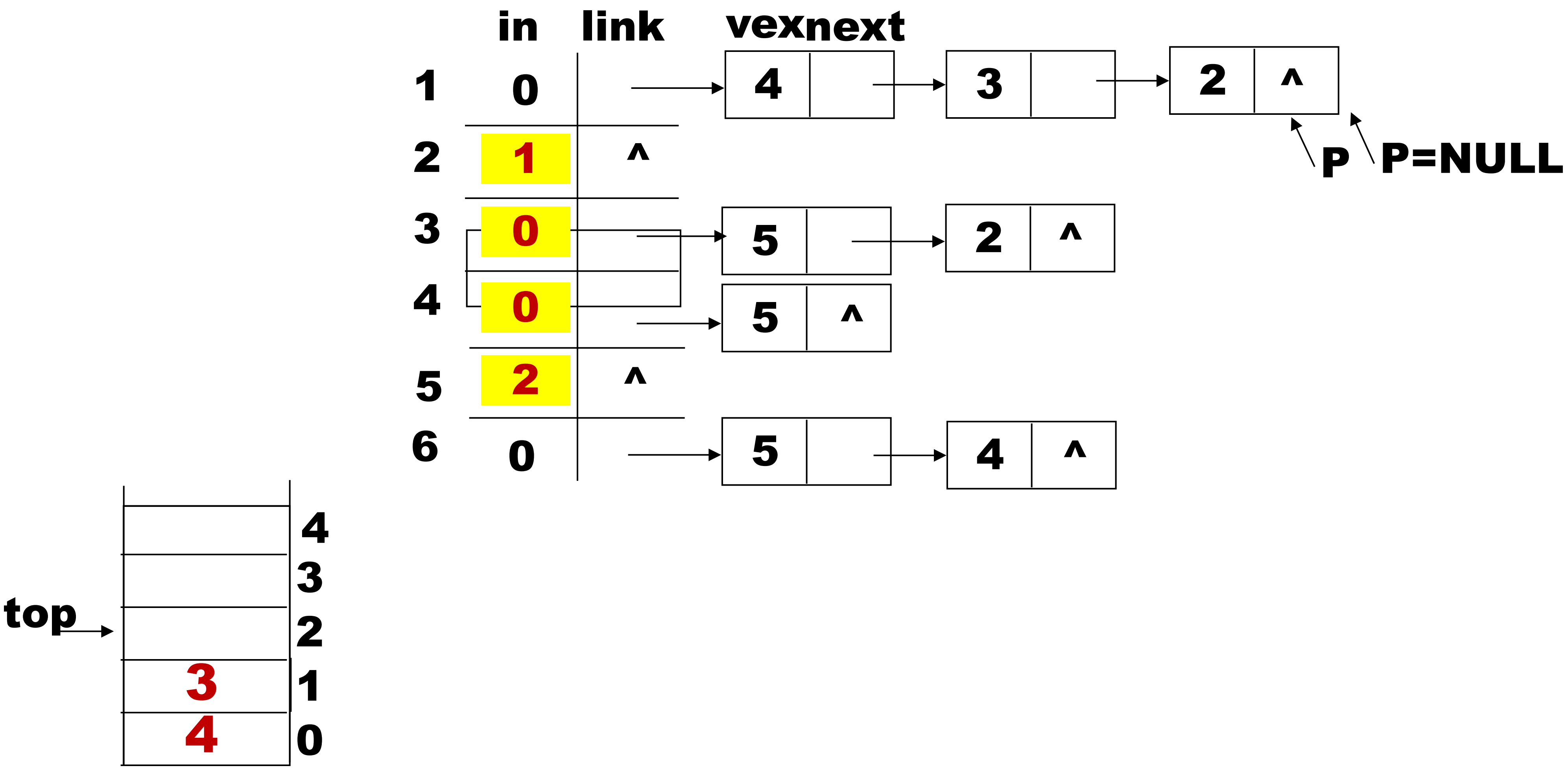
输出序列：6 1

6.5 有向无环图——拓扑排序



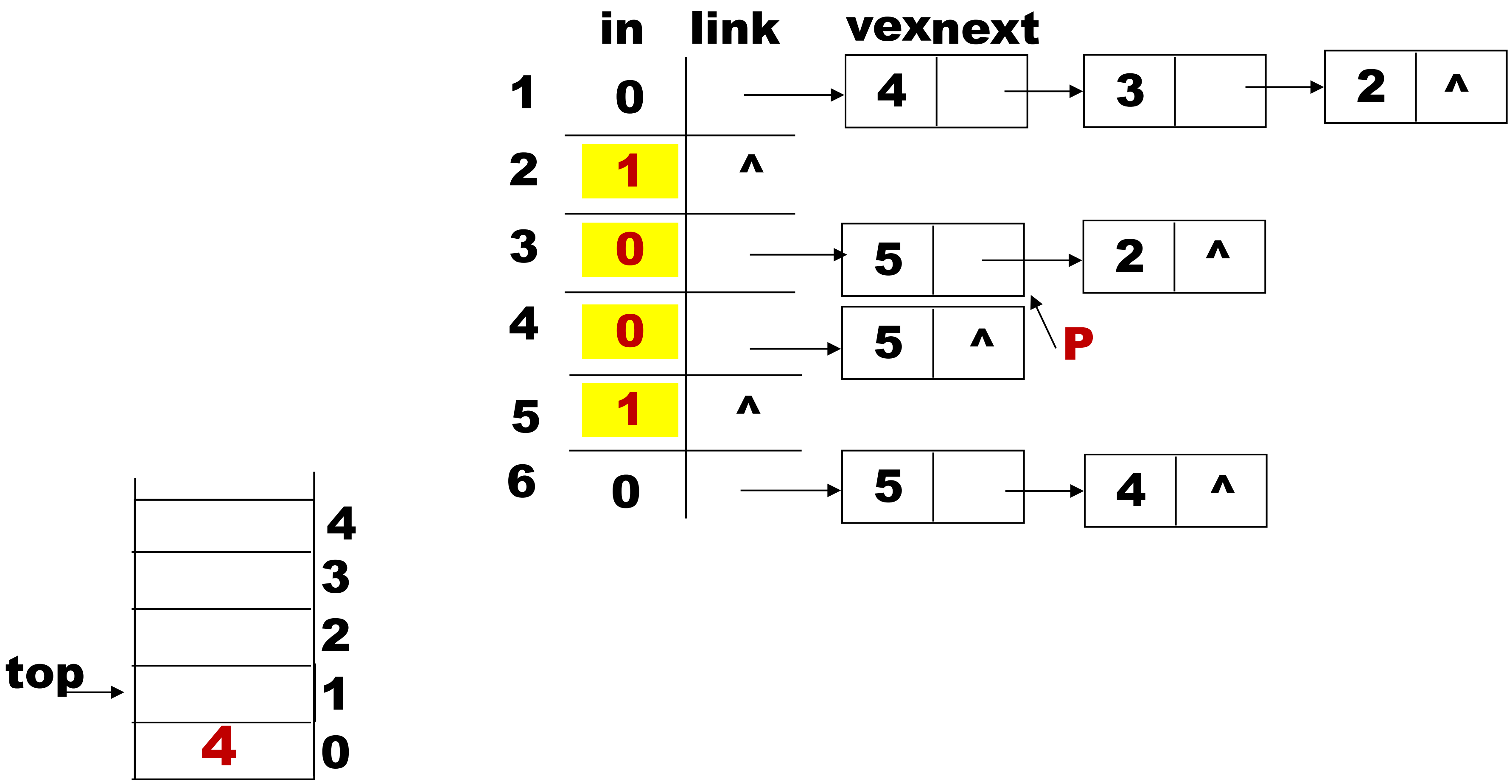
输出序列：6 1

6.5 有向无环图——拓扑排序



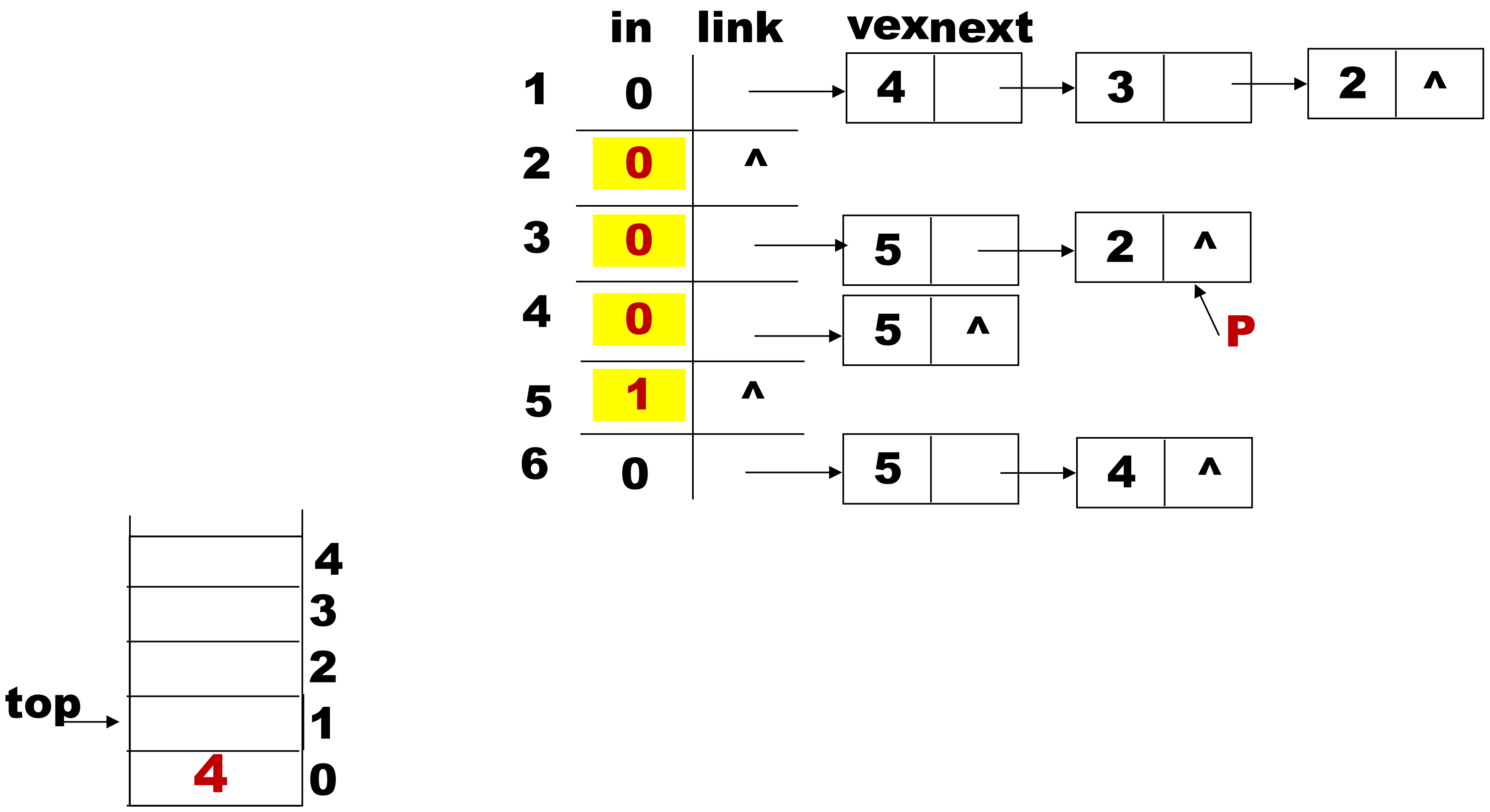
输出序列：6 1

6.5 有向无环图——拓扑排序



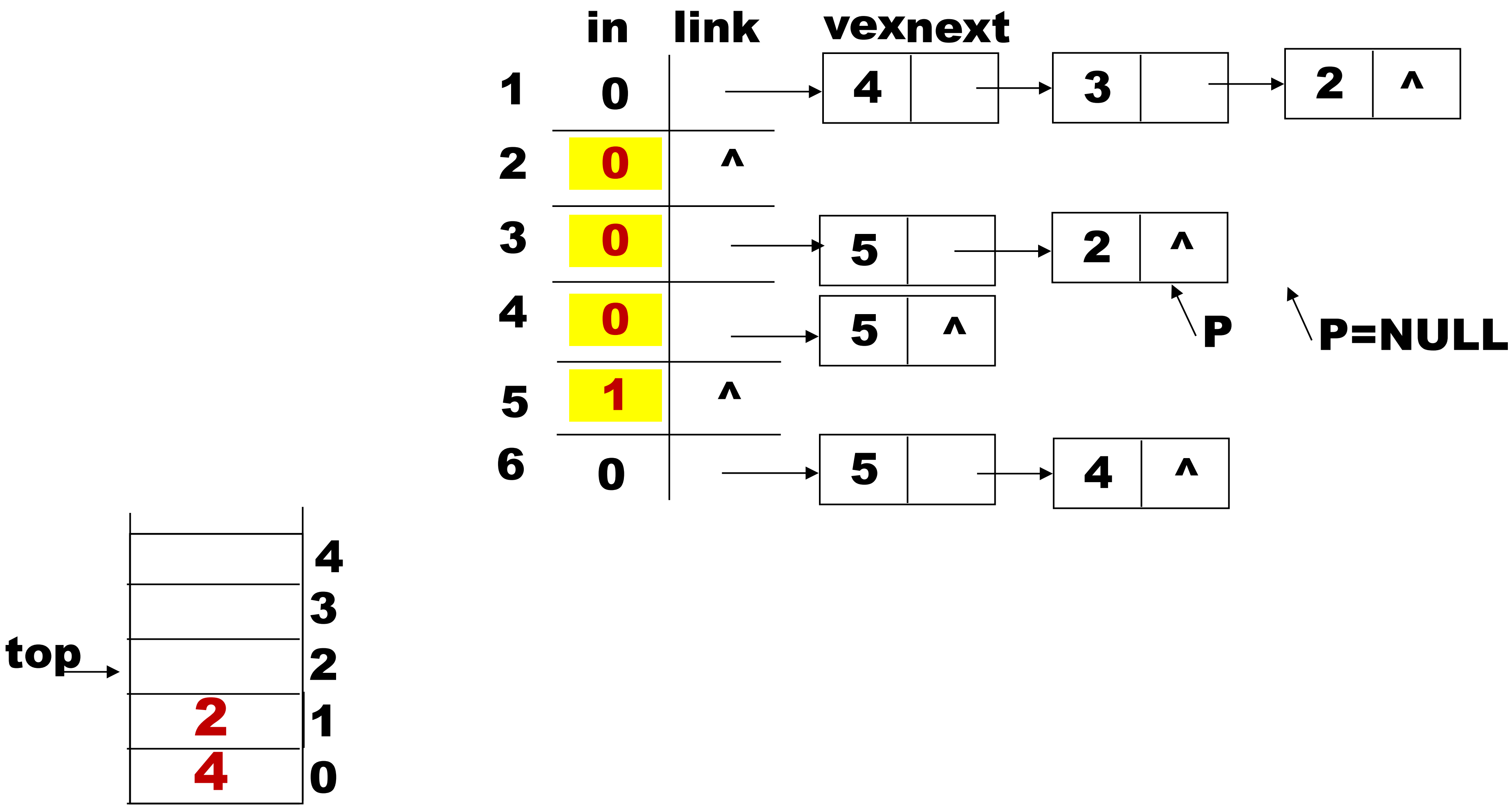
输出序列：6 1 3

6.5 有向无环图——拓扑排序



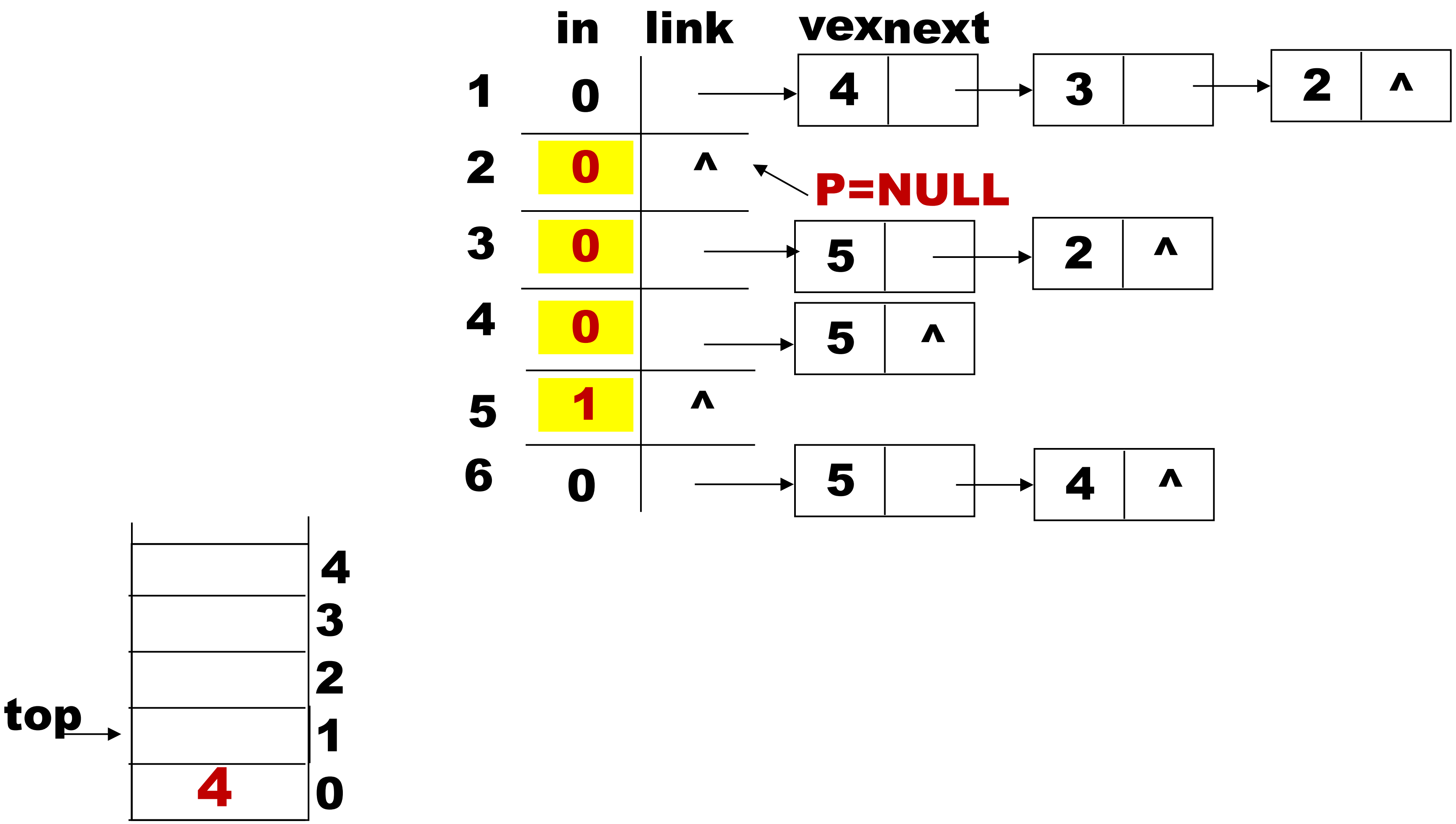
输出序列：6 1 3

6.5 有向无环图——拓扑排序



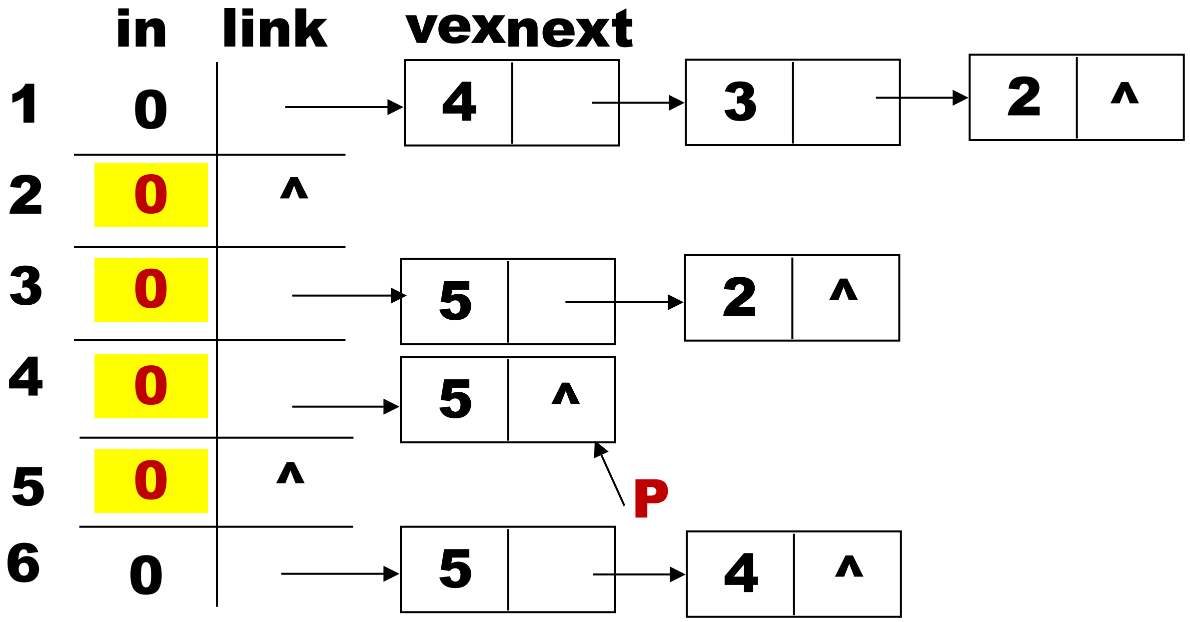
输出序列：6 1 3

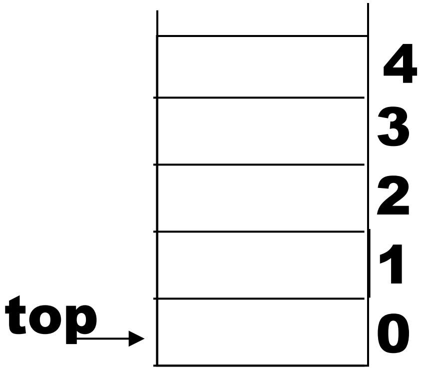
6.5 有向无环图——拓扑排序



输出序列：6 1 3 2

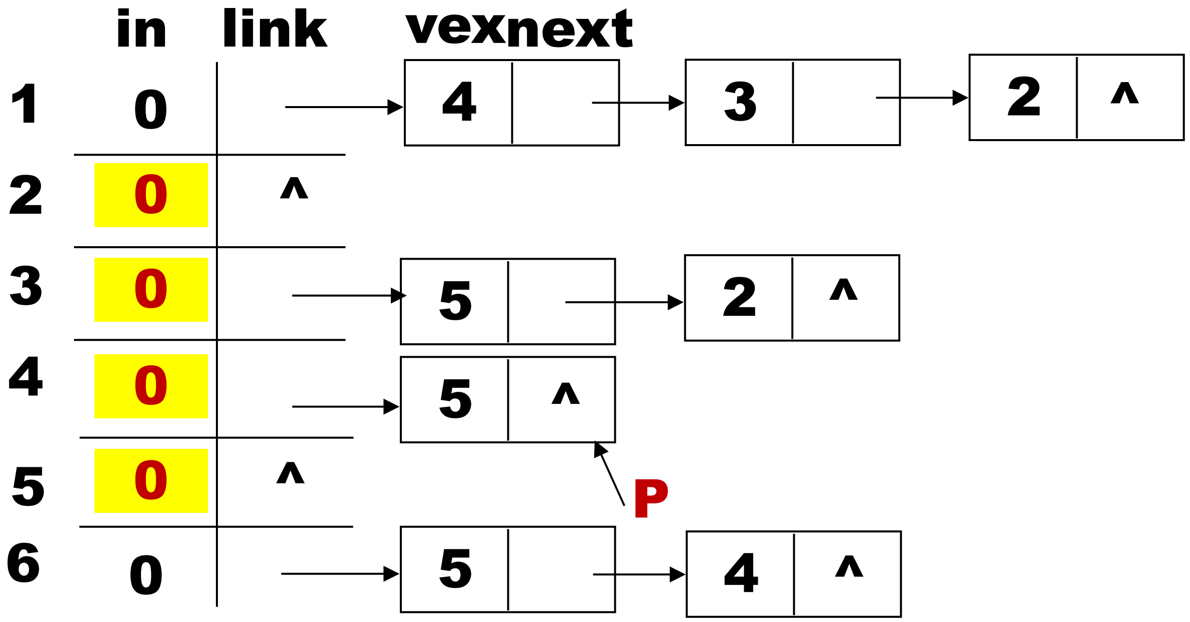
6.5 有向无环图——拓扑排序

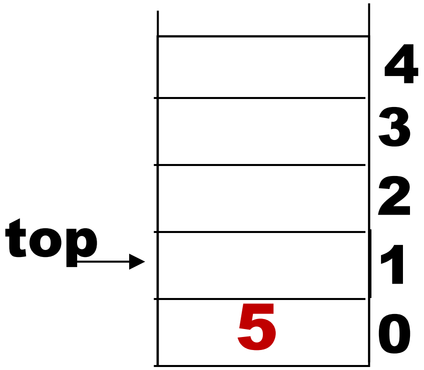




输出序列：6 1 3 2 4

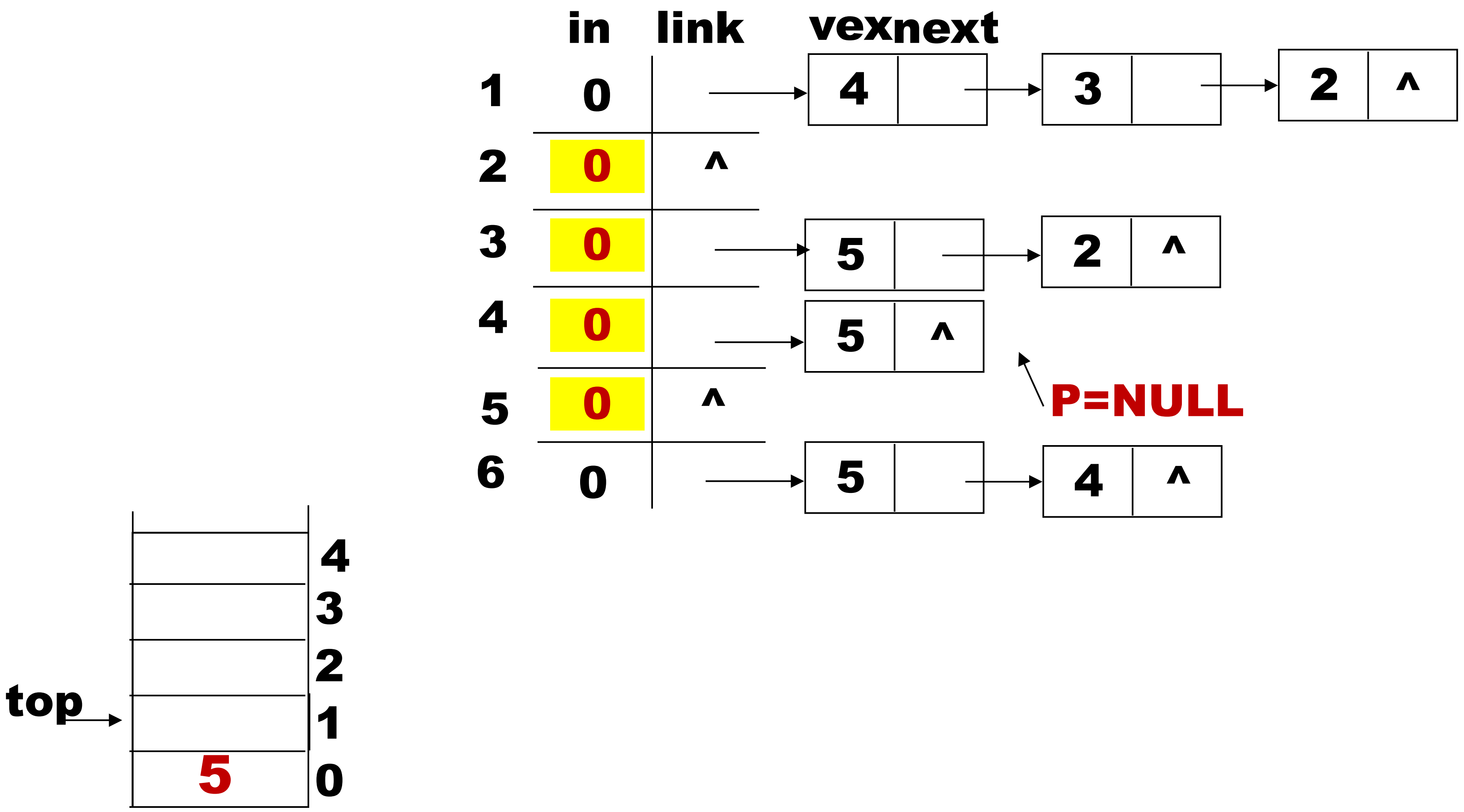
6.5 有向无环图——拓扑排序





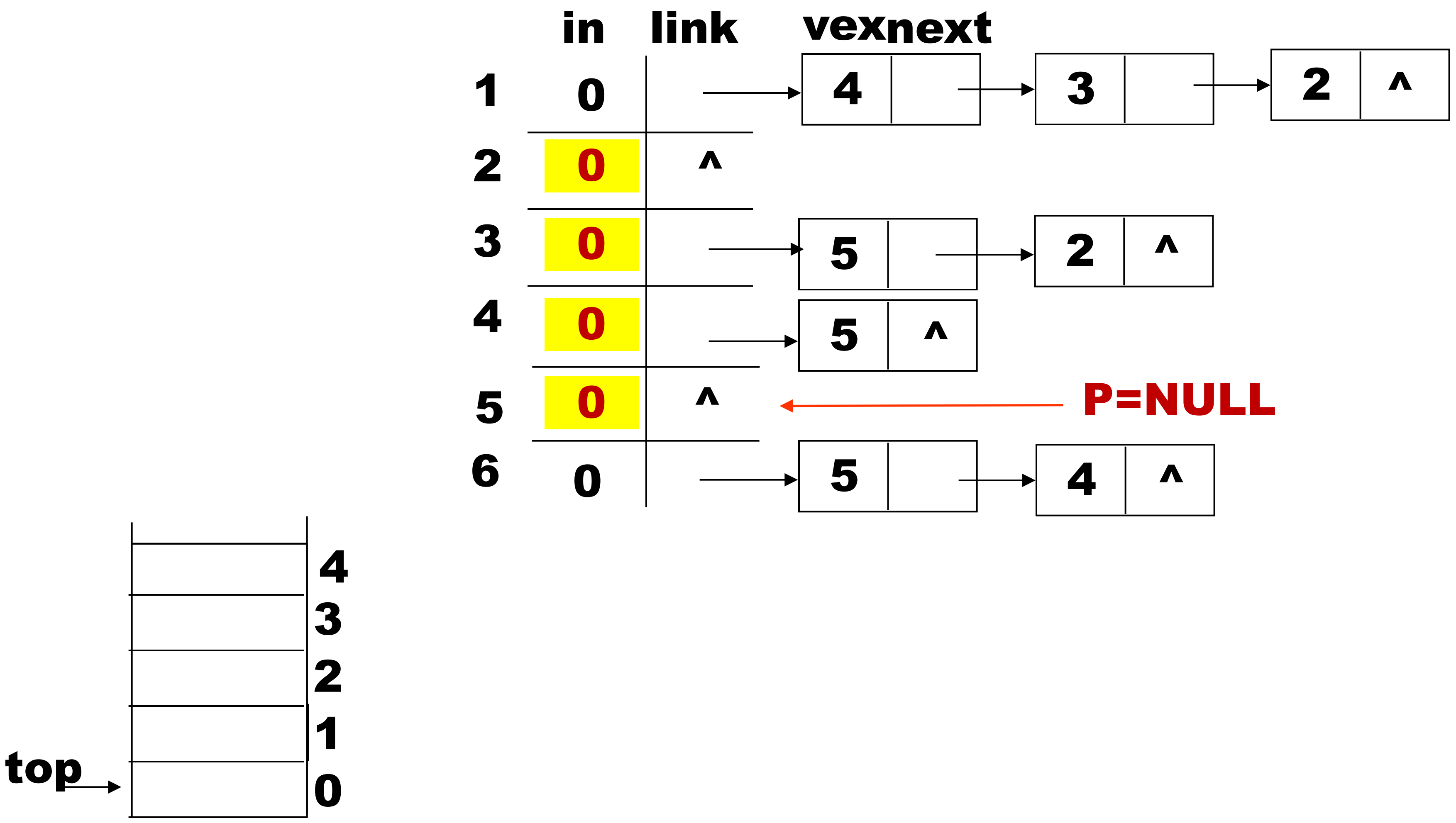
输出序列：6 1 3 2 4

6.5 有向无环图——拓扑排序



输出序列：6 1 3 2 4

6.5 有向无环图——拓扑排序



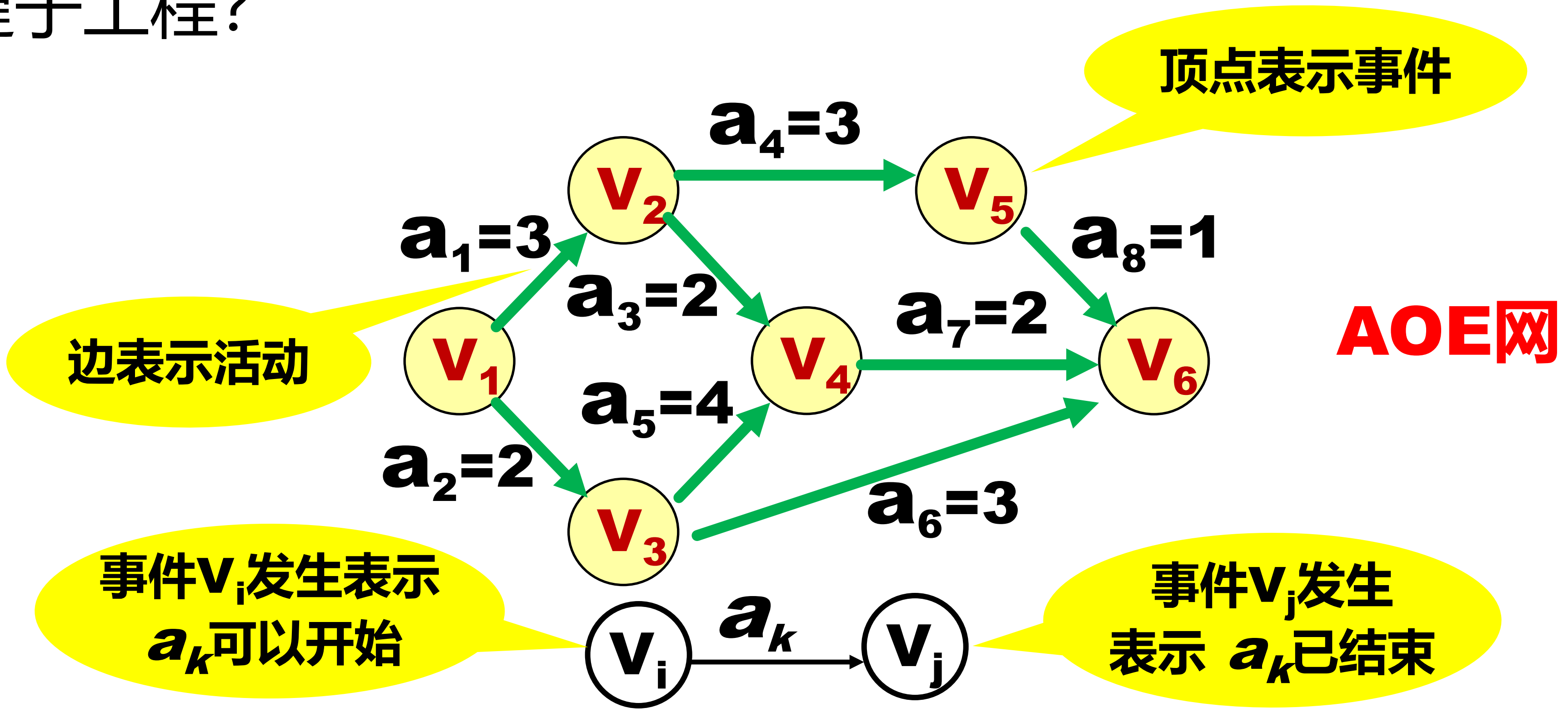
输出序列：6 1 3 2 4 5

6.5 有向无环图——关键路径

问题提出：

1）工程能否顺序进行，即工程流程是否“合理”

2）完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程？



6.5 有向无环图——关键路径

AOE网

AOE——用边表示活动的网。它是有一个带权的有向无环图。

顶点——表示事件，弧——表示活动，

权值——活动持续的时间。

路径长度——路径上各活动持续时间之和

关键路径——路径长度最长的路径叫关键路径

6.5 有向无环图——关键路径

AOV网和AOE网的区别

都能用来表示工程中的各子工程的流程：

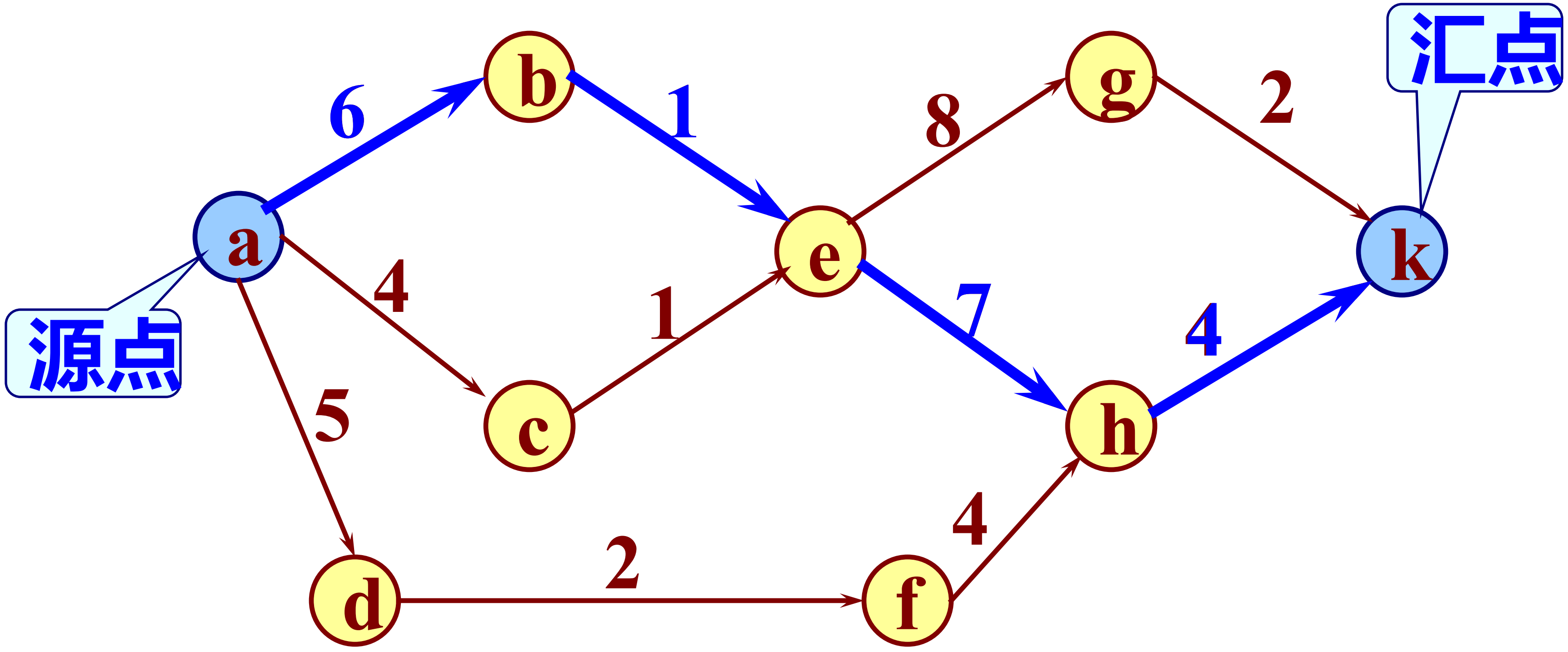
一个用顶点表示活动，

一个用边表示活动，

但AOV网侧重表示活动的前后次序，AOE网除表示活动先后次序，还表示了活动的持续时间等。

因此可利用 AOE网 解决工程所需最短时间及哪些子工程拖延会影响整个工程按时完成等问题。

整个工程完成的时间为：从有向图的源点到汇点的最长路径。



“关键活动”指的是：

该弧上的权值增加 将使有向图上的最长路径的长度增加。

6.5 有向无环图——关键路径

如何求关键活动?

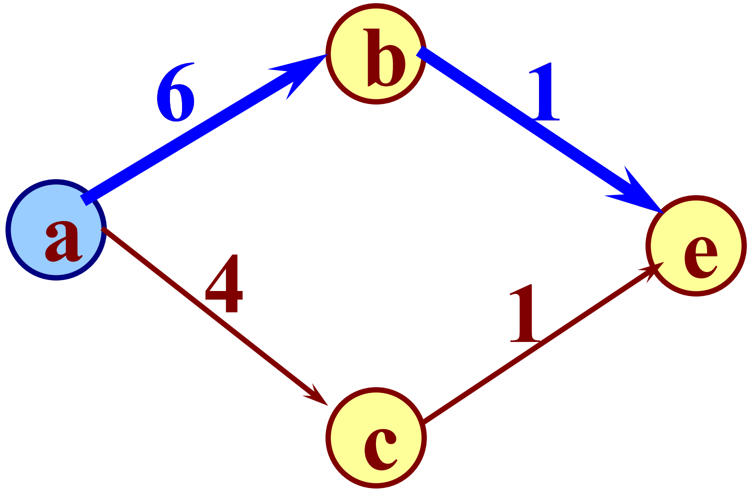
“活动(弧)”的 最早开始时间 e(<j, k>)

“活动(弧)”的 最迟开始时间 l(<j, k>)

关键活动： e(<j, k>) == l(<j, k>)

“事件(顶点)” 的最早发生时间 $v_e(j)$

“事件(顶点)” 的最迟发生时间 $v_l(k)$

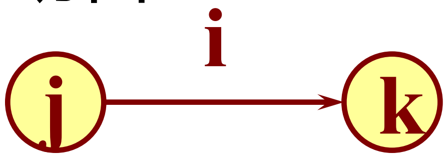


活动(弧)发生时间的计算公式

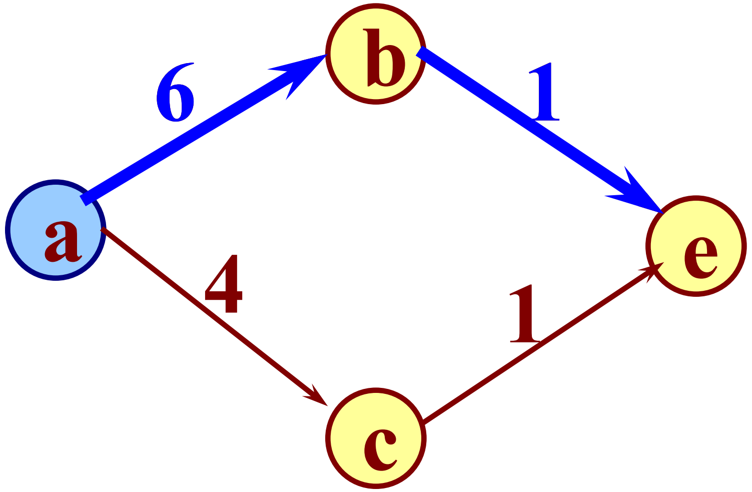
假设第 i 条弧为 <j, k>， 则 对第 i 项活动言

e(<j, k>) = ve(j)；

l(<j, k>) = vl(k) – dut(<j,k>)；



如何求关键活动?



事件(顶点)发生时间的计算公式

最早开始时间:

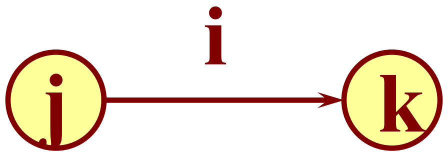
ve(源点) = 0;

ve(k) = Max{ve(j) + dut(<j, k>)}

最迟开始时间:

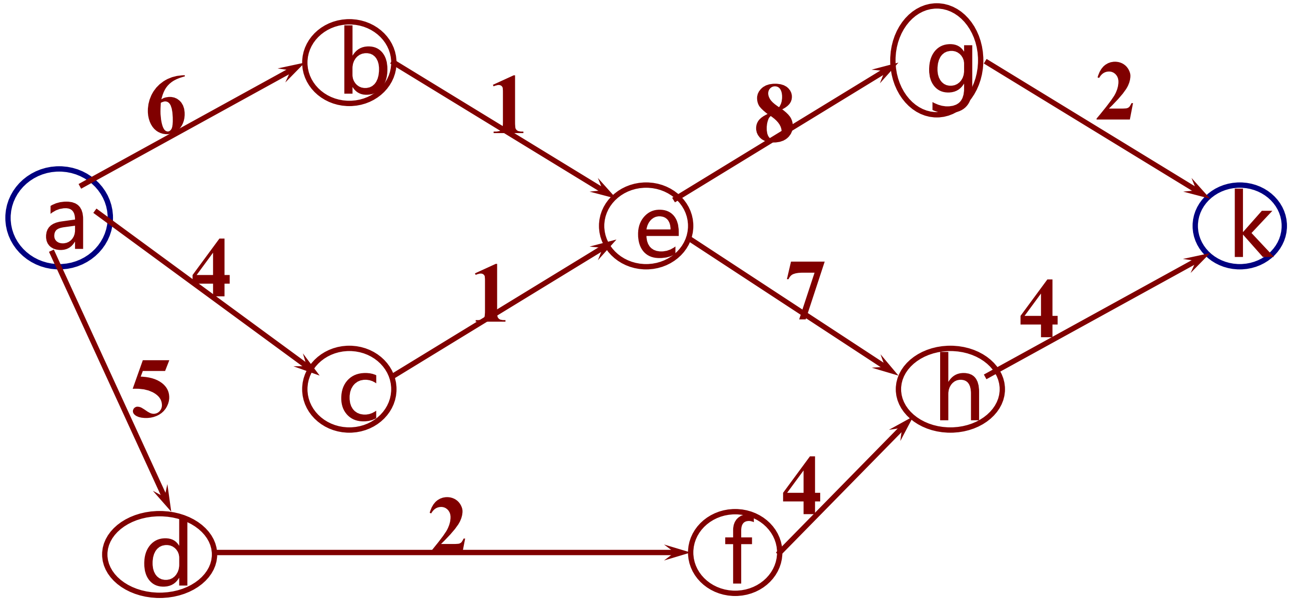
vl(汇点) = ve(汇点);

vl(j) = Min{vl(k) – dut(<j, k>)}



6.5 有向无环图——关键路径

如何求关键活动?



	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

6.5 有向无环图——关键路径

如何求关键活动?

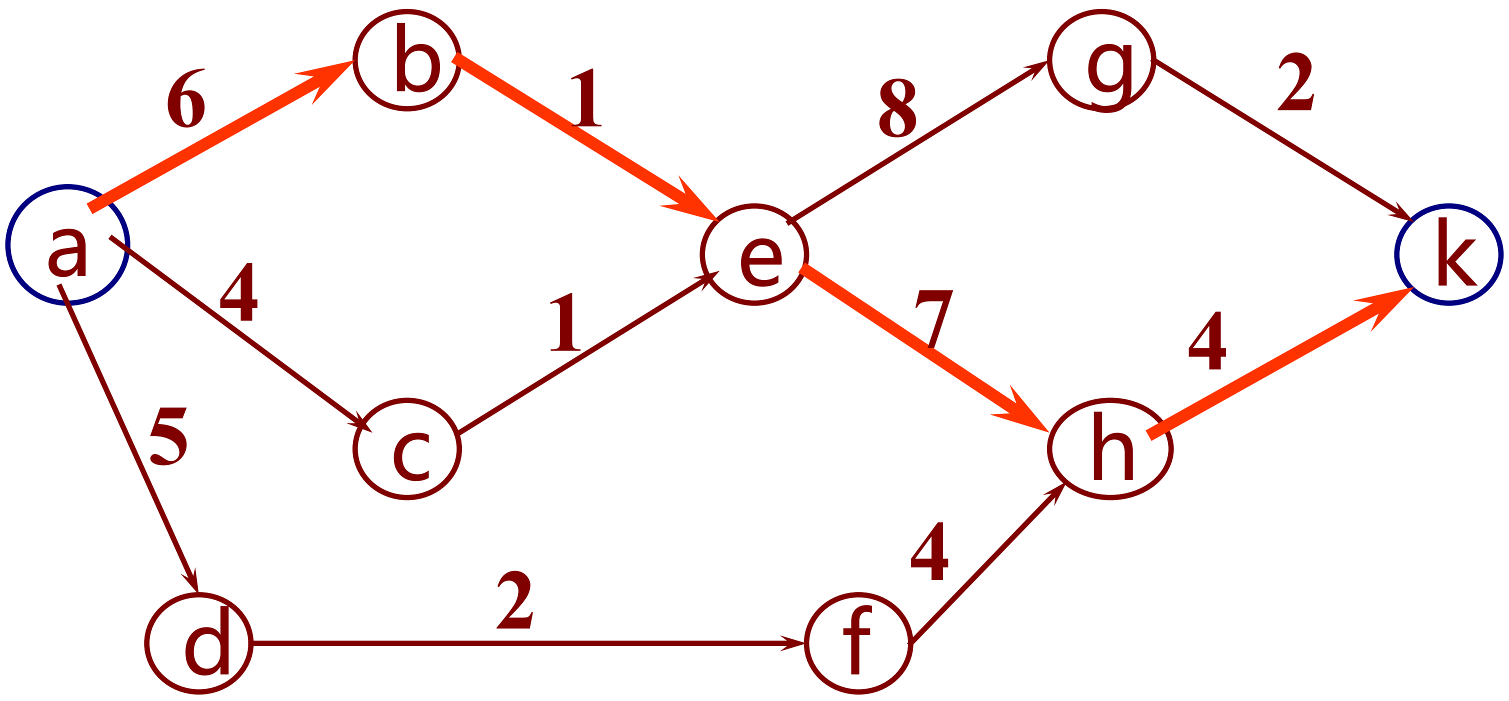
	a	b	c	d	e	f	g	h	k
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

	ab	ac	ad	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
e	0	0	0	6	4	5	7	7	7	15	14
l	0	2	3	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

- e(<j, k>) = ve(j)；

- l(<j, k>) = vl(k) – dut(<j,k>)；

如何求关键活动?



整个工期需要18天

如何缩短整个工期?

6.6 最短路径

问题模型：

用带权的有向图表示一个交通运输网，图中：

顶点——表示城市

边——表示城市间的交通联系

权——表示此线路的长度或沿此线路运输所花的时间或费用等。

问题：

从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径——最短路径。

6.6 最短路径

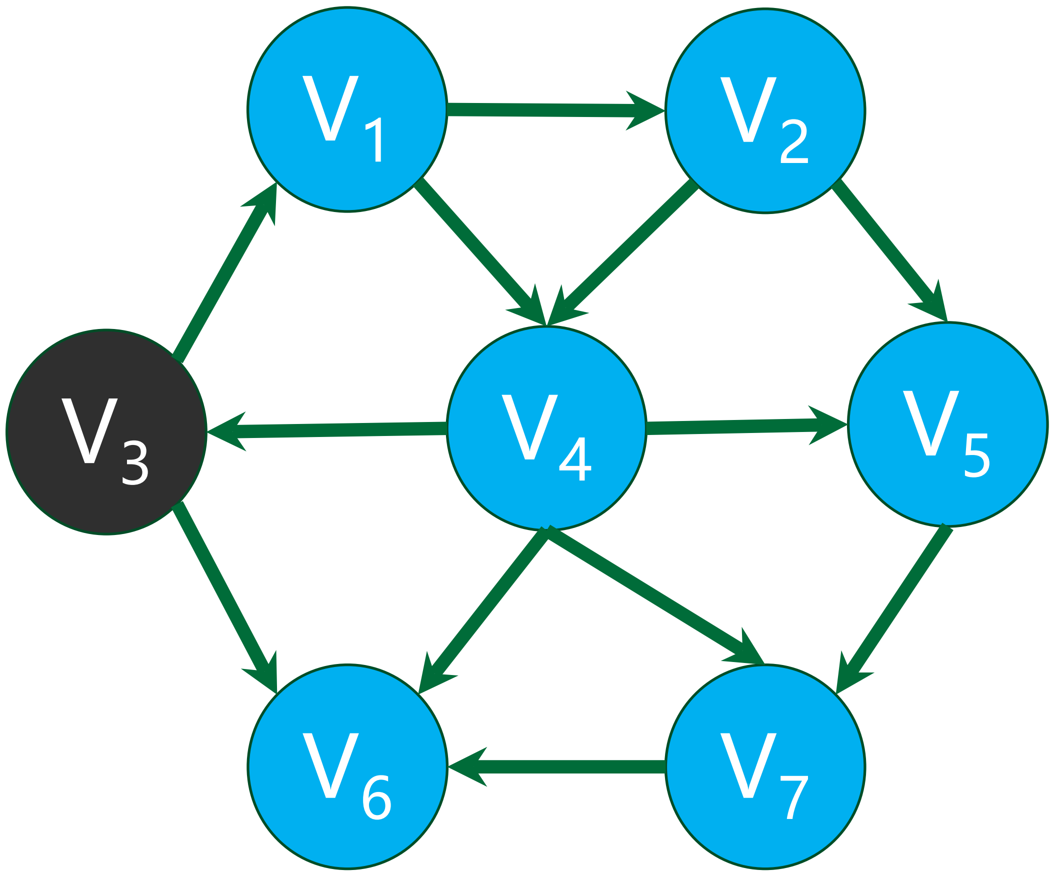
问题模型：

无权图的最短路径；

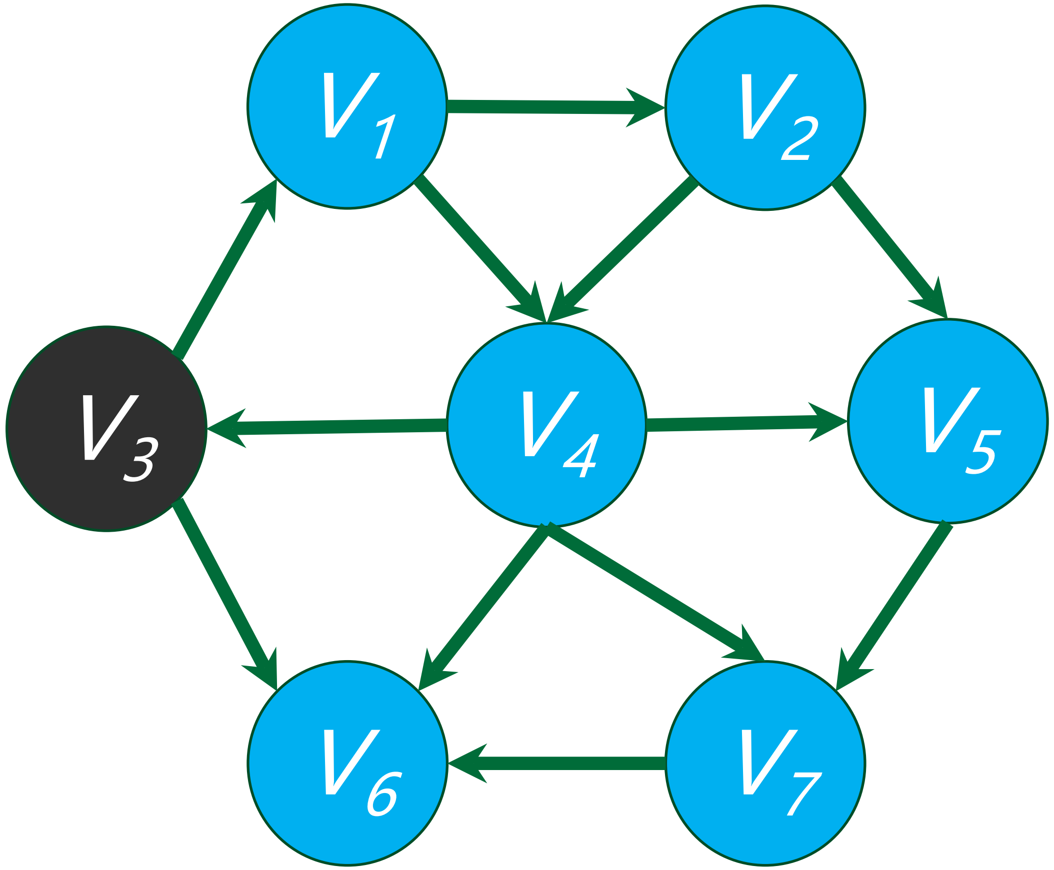
有权图的最短路径；

无权图的最短路径

- 问题：
- 寻找从V_3到其他节点的最短路径
- 要求：
- 1、计算出最短路径长度;
- 2、给出具体的路径输出

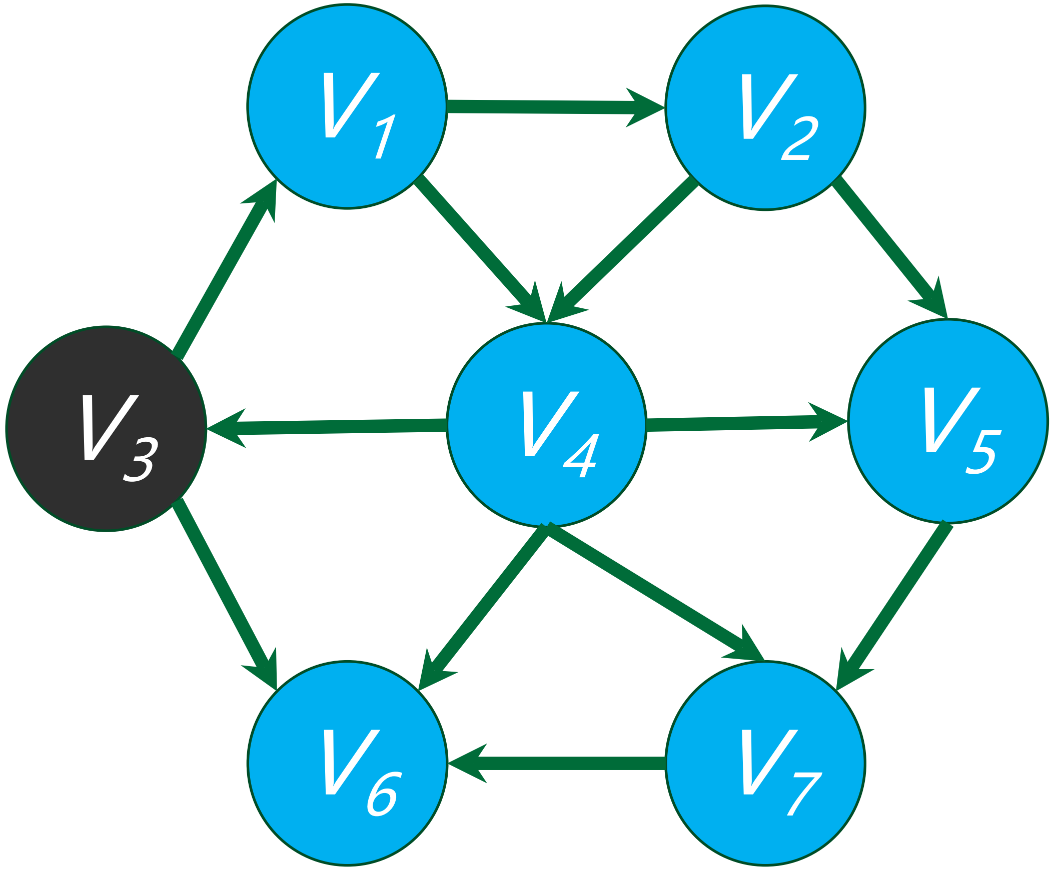


无权图的最短路径



vertex	dist	path
V_1	1	V_3
V_2	2	V_1
V_3	0	0
v_4	2	V_1
V_5	3	V_2
V_6	1	V_3
V_7	3	V_4

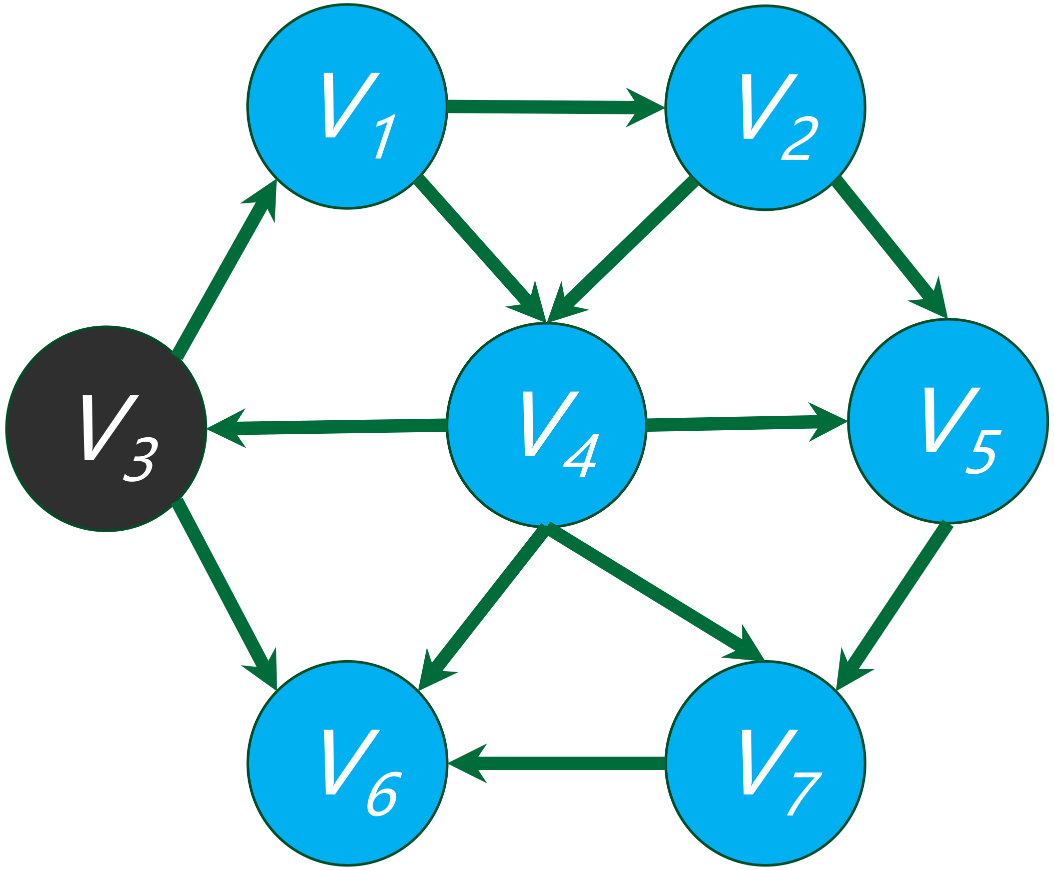
无权图的最短路径



queue

vertex	visit	dist	path
V_1	no	∞	0
V_2	no	∞	0
V_3	no	∞	0
V_4	no	∞	0
V_5	no	∞	0
V_6	no	∞	0
V_7	no	∞	0

无权图的最短路径



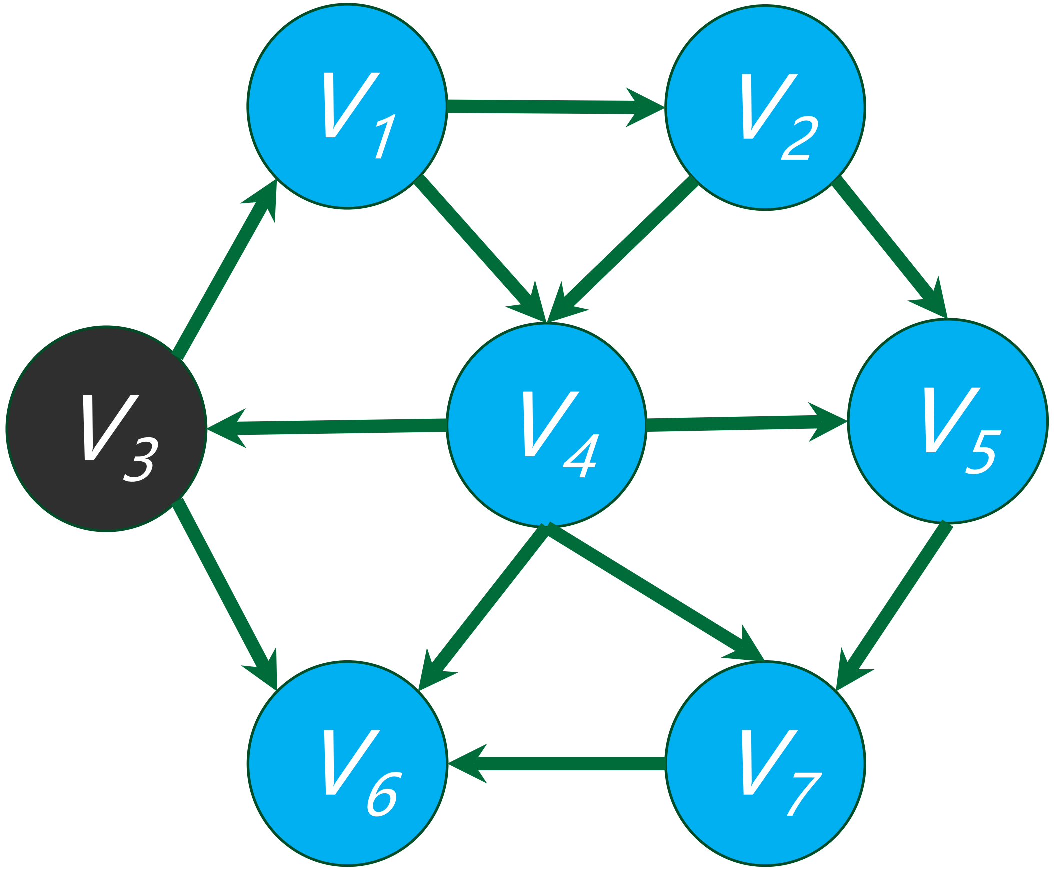
queue

queue
V_3

vertex	visit	dist	path
V_1	*no*	∞	*0*
V_2	*no*	∞	*0*
V_3	*yes*	*0*	*0*
V_4	*no*	∞	*0*
V_5	*no*	∞	*0*
V_6	*no*	∞	*0*
V_7	*no*	∞	*0*

V_3入队

无权图的最短路径

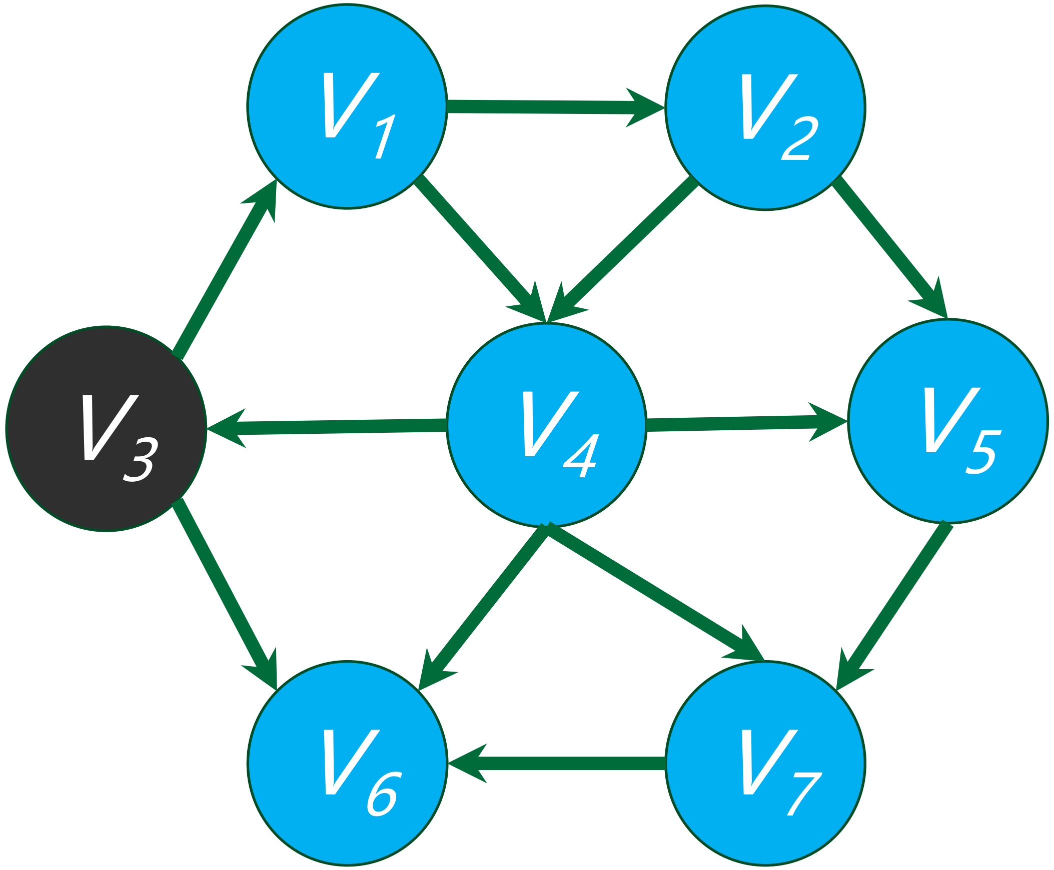


queue
V_1
V_6

vertex	visit	dist	path
V_1	no	∞	0
V_2	no	∞	0
V_3	yes	0	0
V_4	no	∞	0
V_5	no	∞	0
V_6	no	∞	0
V_7	no	∞	0

V_3出队,V_1,V_6进队

无权图的最短路径

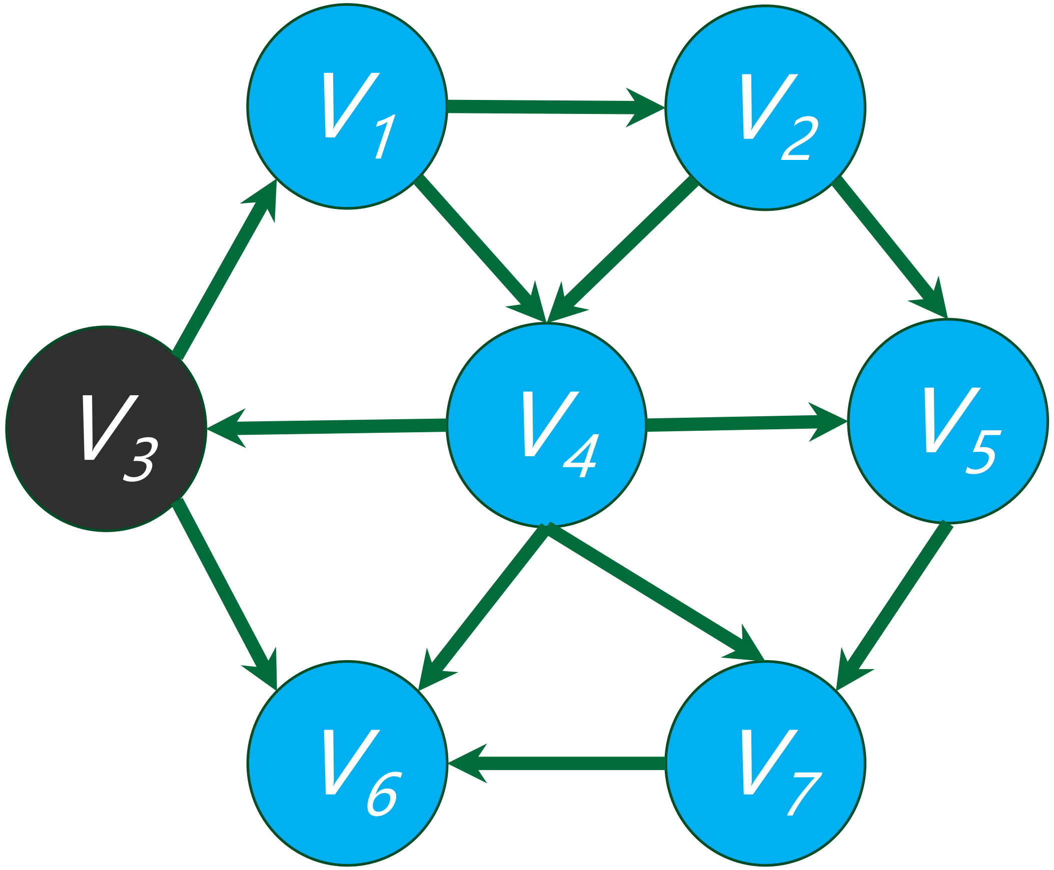


queue
V_1
V_6

vertex	visit	dist	path
V_1	Yes	1	V3
V_2	no	∞	0
V_3	yes	0	0
v_4	no	∞	0
V_5	no	∞	0
V_6	no	∞	0
V_7	no	∞	0

dist[1]=dist[3]+1

无权图的最短路径

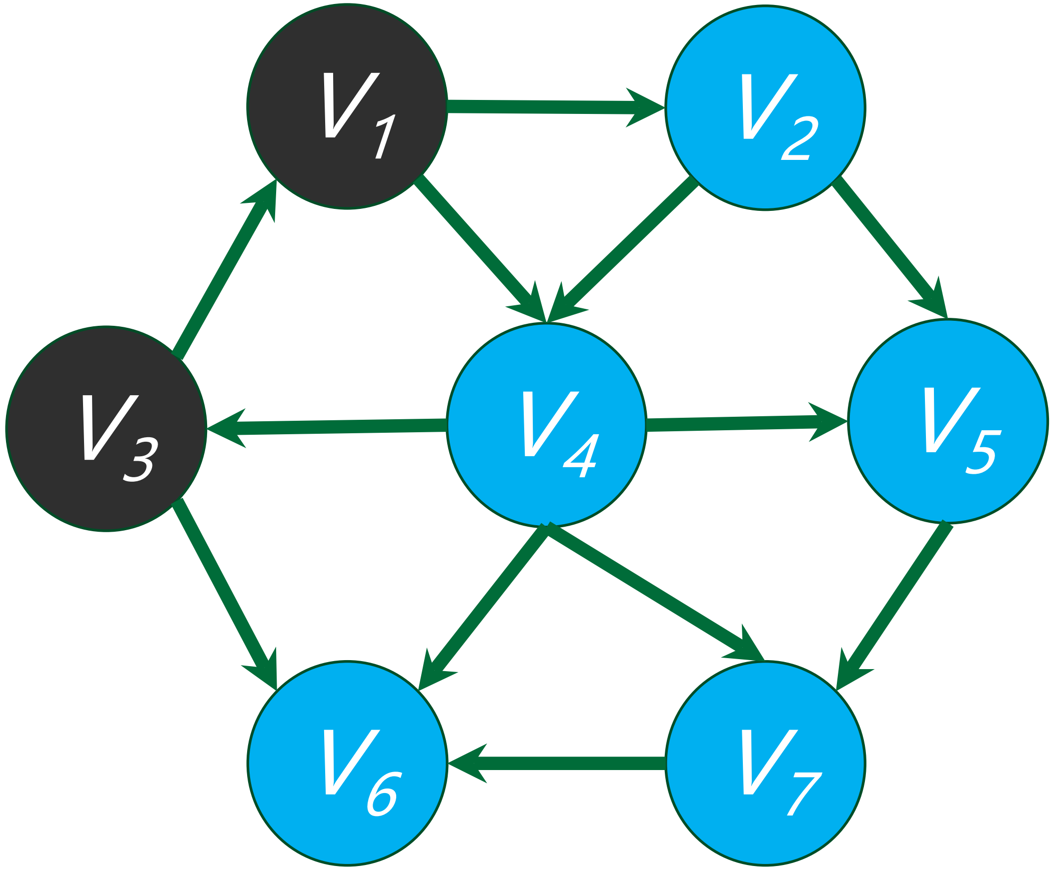


queue
V_1
V_6

vertex	visit	dist	path
V_1	*Yes*	1	V_3
V_2	*no*	∞	0
V_3	*yes*	0	0
V_4	*no*	∞	0
V_5	*no*	∞	0
V_6	*yes*	1	V_3
V_7	*no*	∞	0

dist[6]=dist[3]+1

无权图的最短路径

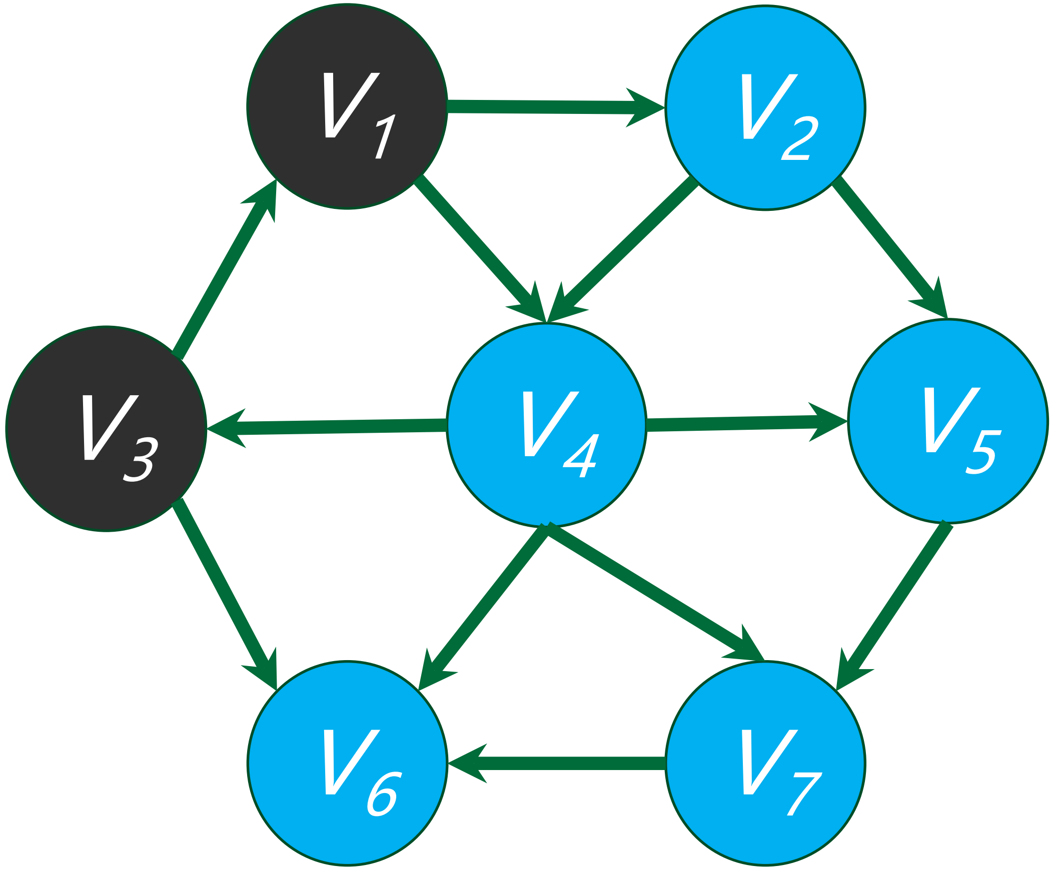


queue
V_6
V_2
V_4

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	no	∞	0
V_3	yes	0	0
v_4	no	∞	0
V_5	no	∞	0
V_6	yes	1	V_3
V_7	no	∞	0

V_1出队，V_2，V_4进队

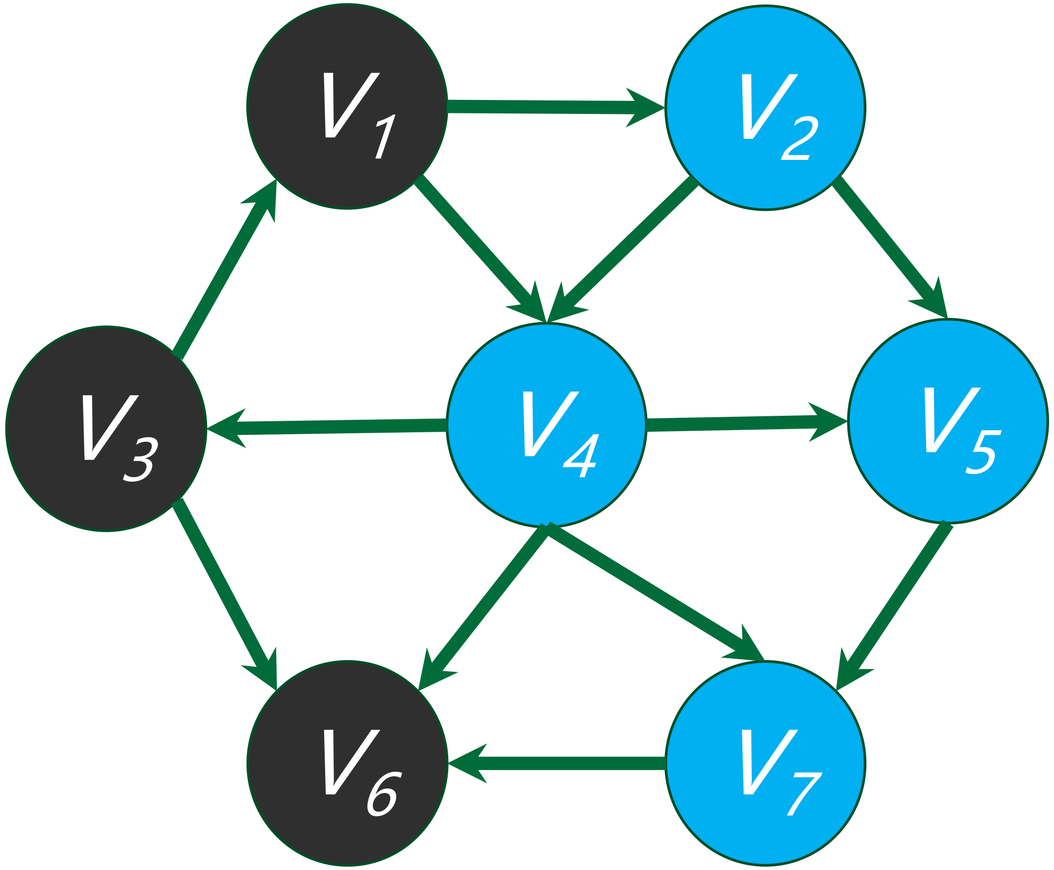
无权图的最短路径



queue
V_6
V_2
V_4

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
v_4	Yes	2	V_1
V_5	no	∞	0
V_6	yes	1	V_3
V_7	no	∞	0

无权图的最短路径



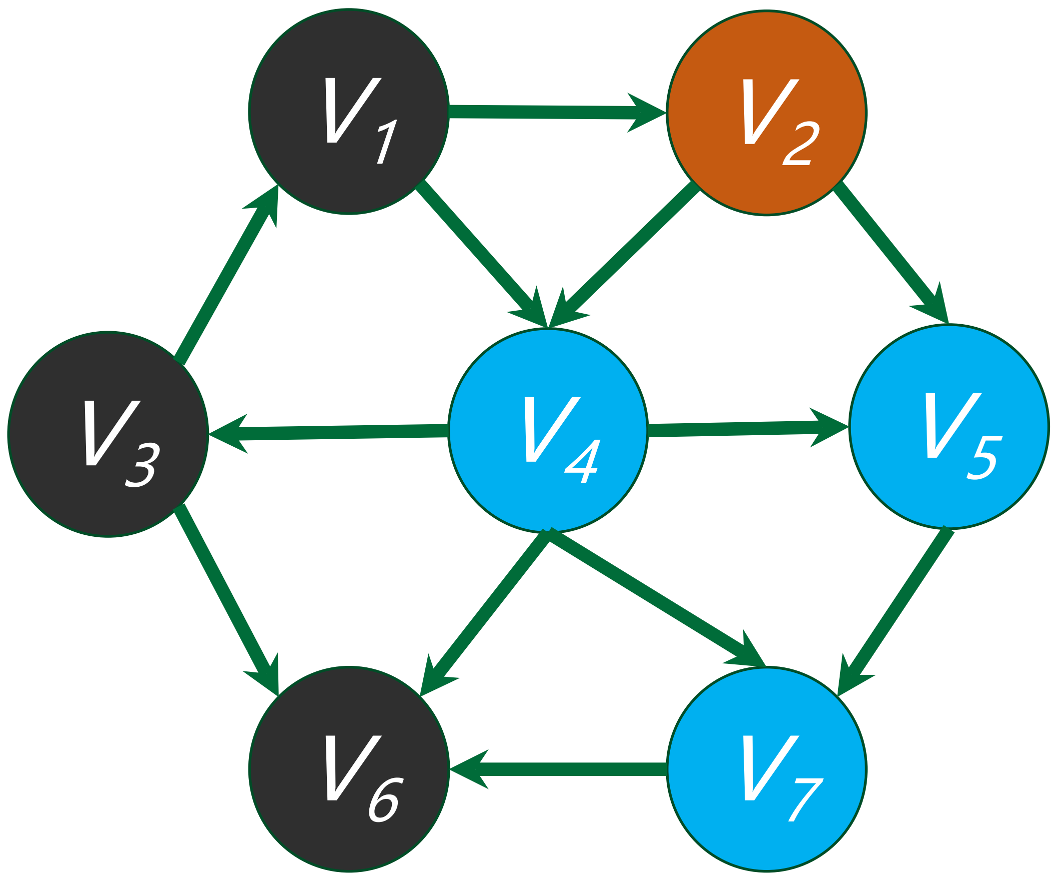
queue

V_2
V_4

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
V_4	Yes	2	V_1
V_5	no	∞	0
V_6	yes	1	V_3
V_7	no	∞	0

V_6出队

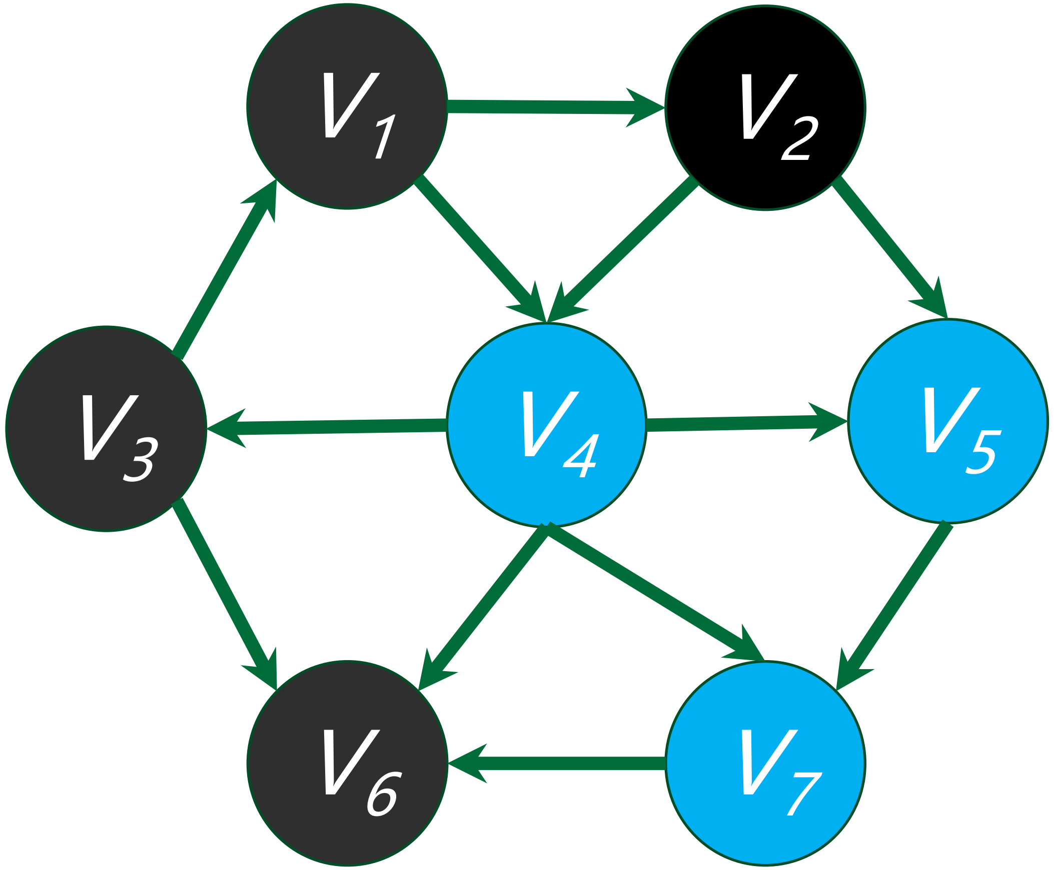
无权图的最短路径



queue
V_2
V_4

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
V_4	Yes	2	V_1
V_5	no	∞	0
V_6	yes	1	V_3
V_7	no	∞	0

无权图的最短路径

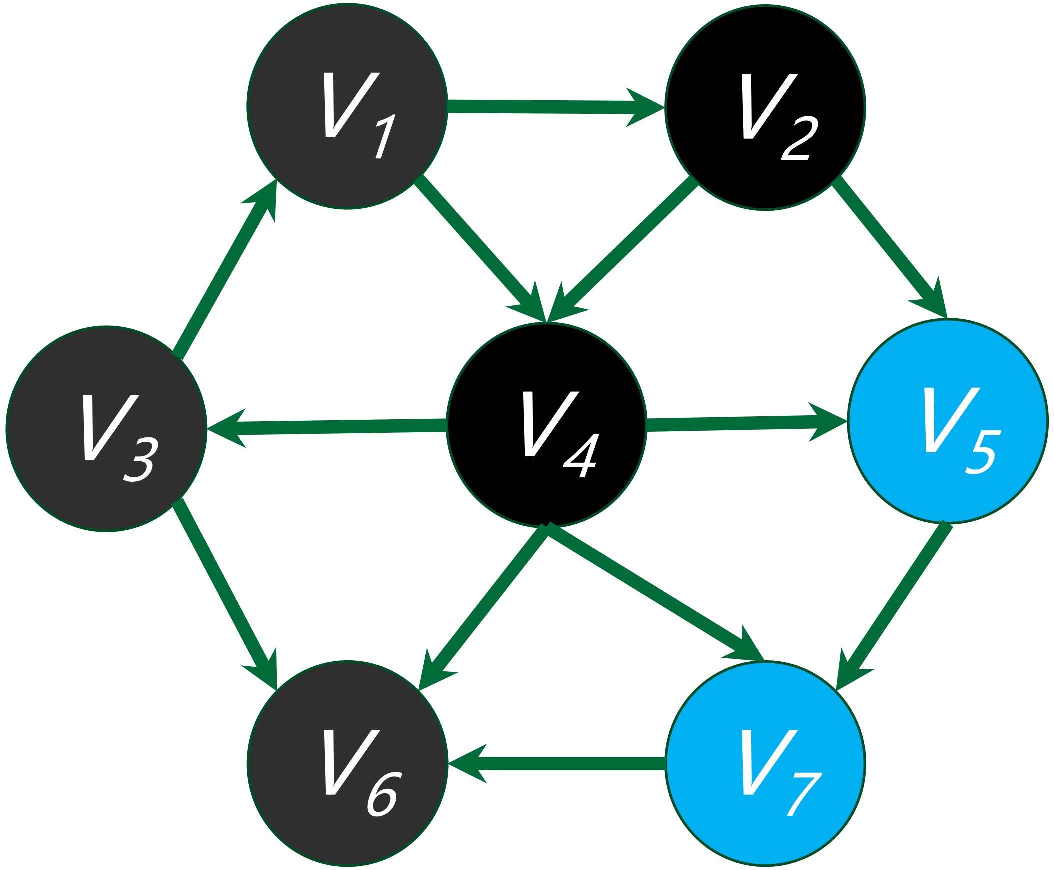


queue
V_4
V_5

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
v_4	Yes	2	V_1
V_5	Yes	3	V2
V_6	yes	1	V_3
V_7	no	∞	0

V_2出队，V_5入队

无权图的最短路径

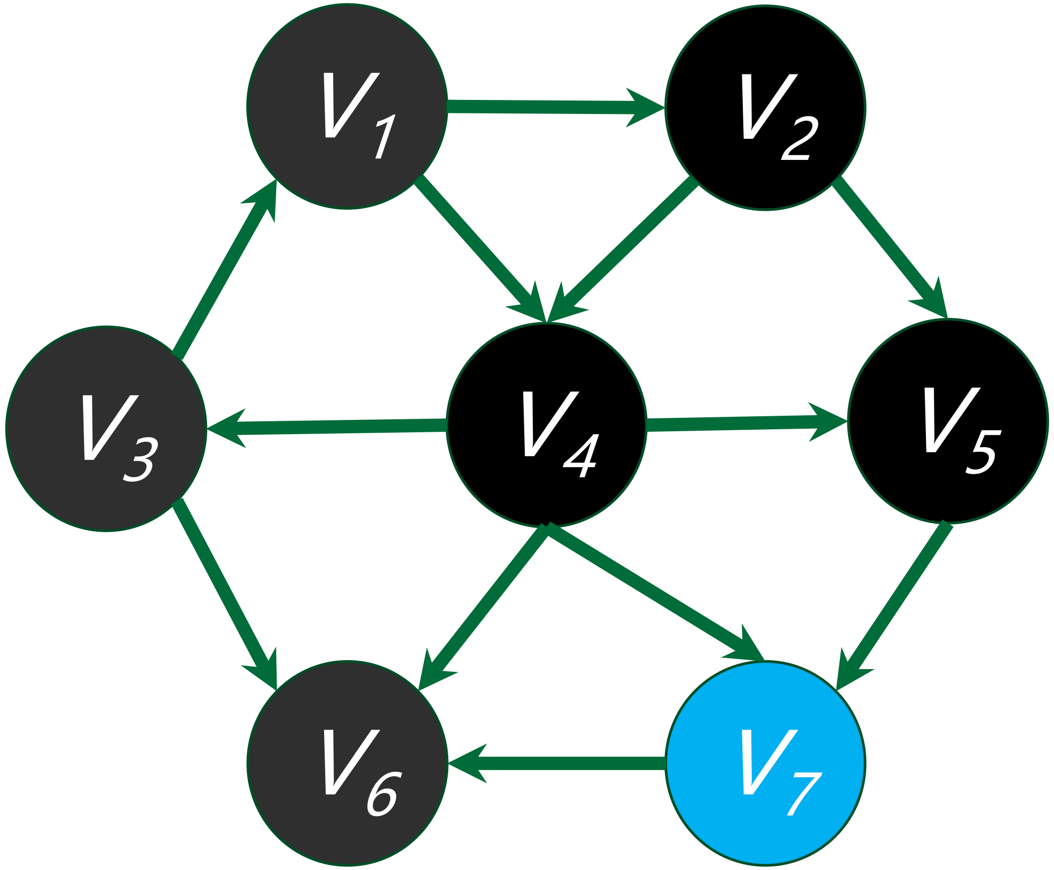


queue
V_5
V_7

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
v_4	Yes	2	V_1
V_5	Yes	3	V2
V_6	yes	1	V_3
V_7	Yes	3	V_4

V_4出队，V_7入队

无权图的最短路径

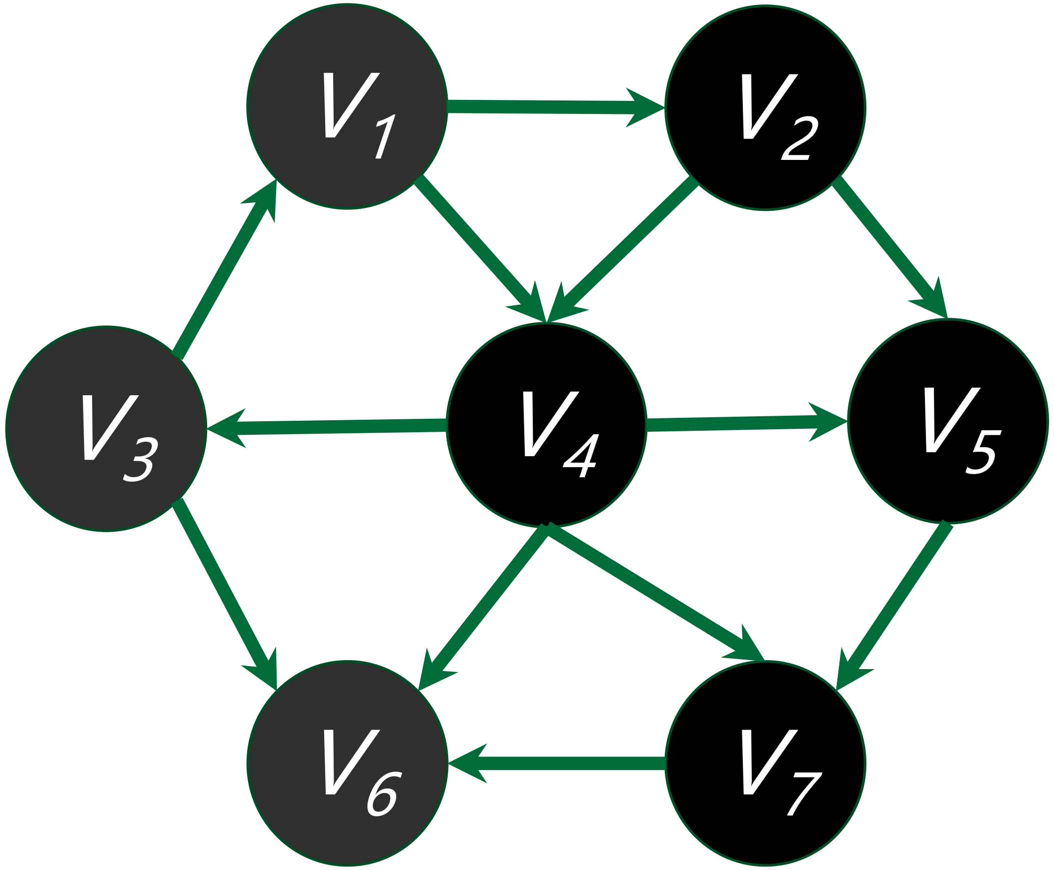


queue
V_5
V_7

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
v_4	Yes	2	V_1
V_5	Yes	3	V2
V_6	yes	1	V_3
V_7	Yes	3	V_4

V_5出队

无权图的最短路径



queue
V_5
V_7

vertex	visit	dist	path
V_1	Yes	1	V_3
V_2	Yes	2	V_1
V_3	yes	0	0
v_4	Yes	2	V_1
V_5	Yes	3	V_2
V_6	yes	1	V_3
V_7	Yes	3	V_4

V_7出队

6.6 最短路径

求从某个源点到其余各顶点的最短路径——迪杰斯特拉(Dijkstra)算法

指的是对已知图 G=（V，E），给定源顶点 s∈V，找出 s 到图中其它各顶点的最短路径。

求每一对顶点之间的最短路径——弗洛伊德(Floyd)算法

指的是对已知图 G=（V，E），任意的顶点 V_i，V_j∈V，找出从 V_i 到 V_j 的最短路径。

6.6 最短路径

迪杰斯特拉算法基本思想：按长度递增的顺序求解最短路径

把图中所有顶点分成两组

　　第1组包括已求得最短路径的顶点

　　第2组包括尚未求得最短路径的顶点；

每次从第2组中选择与源点距离最小的顶点，加入第1组， 直至把图的所有顶点都加到进第1组。

6.6 最短路径-迪杰斯特拉算法

全图中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。

下一段路径长度最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v_1，再到达该顶点（由两条弧组成）。

北京理工大学 BEIJING INSTITUTE OF TECHNOLOGY

再下一段路径长度最短的最短路径的特点：

它可能有三种情况：

1. 或者是直接从源点到该点（只含一条弧）；
2. 或者是从源点经过顶点v1，再到达该顶点（由两条弧组成）；
3. 或者是从源点经过顶点v2，再到达该顶点。

其余最短路径的特点：

它或者是直接从源点到该点（只含一条弧）；或者是从源点经过已求得最短路径的顶点，再到达该顶点。

6.6 最短路径-迪杰斯特拉算法

求最短路径步骤——从V_0到其他顶点

1. 初始时令 S = { V_0 }，T = { 其余顶点 }，T中顶点对应的距离值
   ① 若存在 $<V_0, V_i>$，为 $<V_0,V_i>$ 弧上的权值
   ② 若不存在 $<V_0, V_i>$，为∞
2. 从 T 中选取一个其距离值为最小的顶点 W，加入S，
3. 对 T 中顶点的距离值进行修改：若加进 W 作中间顶点，从 V_0 到 V_i 的距离值 比 不加 W 的路径要短，则修改此距离值；
4. 重复2-4上述步骤，直到S中包含所有顶点，即 S=V 为止。

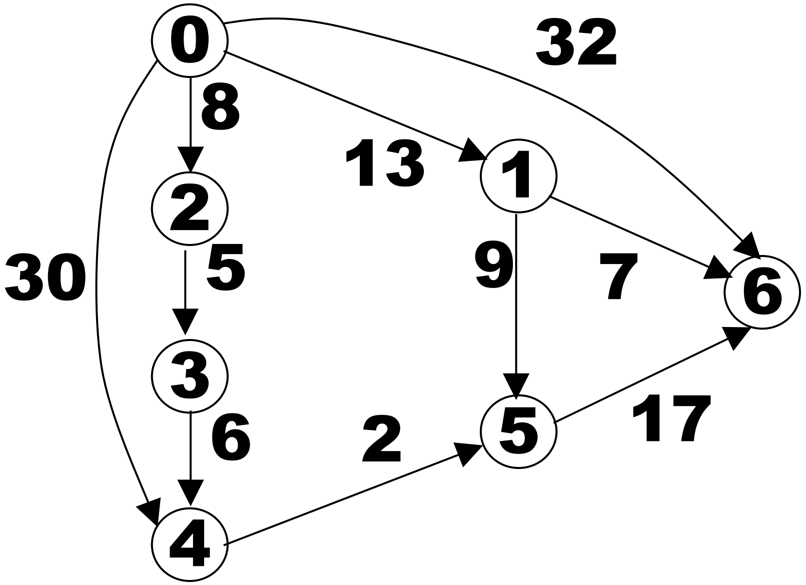
6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

1. 初始时令 S = { V_0 }，T = { 其余顶点 }，T中顶点对应的距离值

   ① 若存在 $<V_0,V_i>$，为 $<V_0,V_i>$ 弧上的权值

   ② 若不存在 $<V_0,V_i>$，为∞

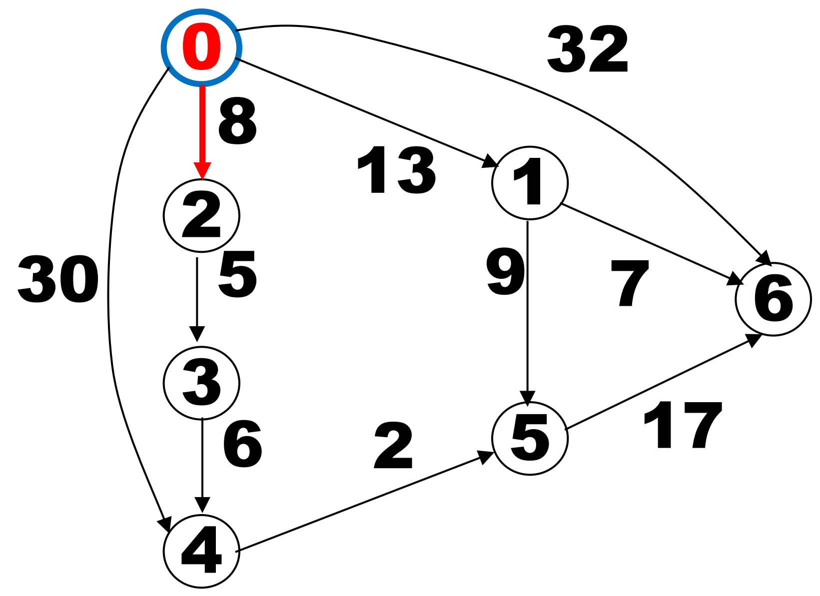


$$
adj[\,][\,]=\begin{vmatrix}
0 & 13 & 8 & \infty & 30 & \infty & 32\\
\infty & 0 & \infty & \infty & \infty & 9 & 7\\
\infty & \infty & 0 & 5 & \infty & \infty & \infty\\
\infty & \infty & \infty & 0 & 6 & \infty & \infty\\
\infty & \infty & \infty & \infty & 0 & 2 & \infty\\
\infty & \infty & \infty & \infty & \infty & 0 & 17\\
\infty & \infty & \infty & \infty & \infty & \infty & 0
\end{vmatrix}
$$

6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

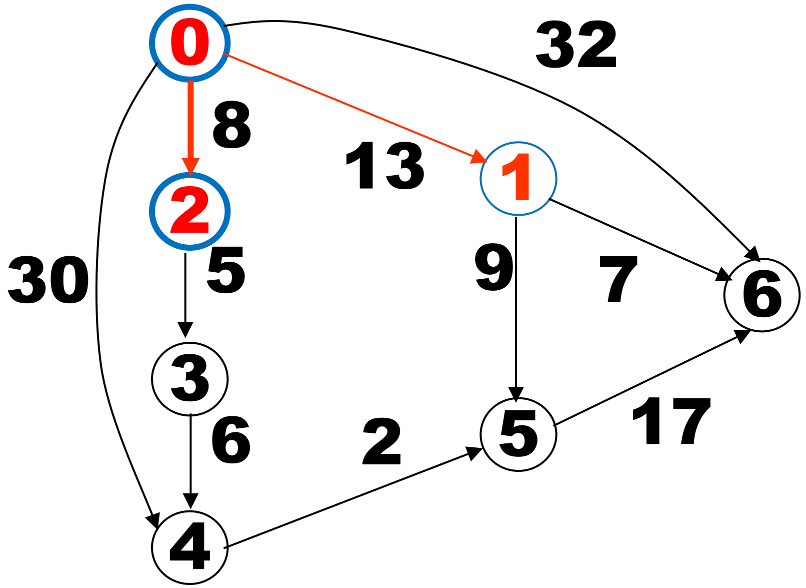
终点	从V0到各终点的最短路径及其长度					
V1	13 $<V_0,V_1>$					
V2	8 $<V_0,V_2>$					
V3	∞					
V4	30 $<V_0,V_4>$					
V5	∞					
V6	32 $<V_0,V_6>$					
Vj	V_2:8 $<V_0,V_2>$					



6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

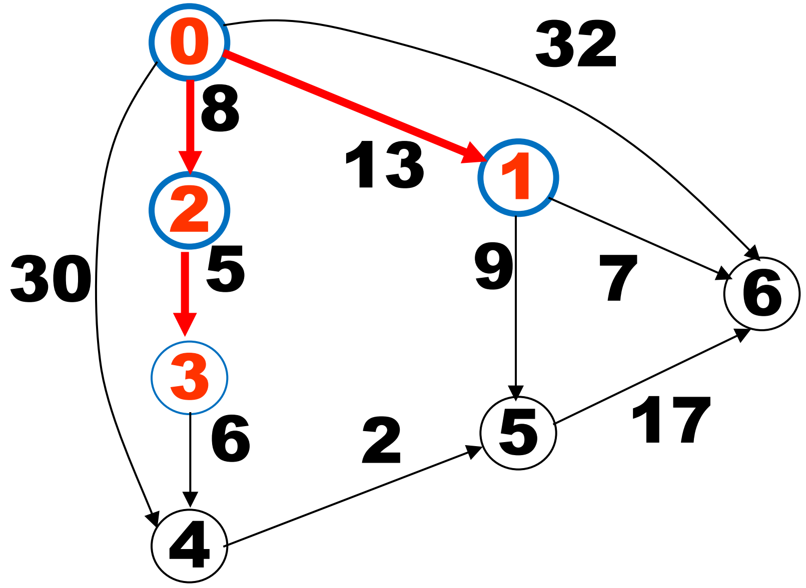
终点	从V0到各终点的最短路径及其长度				
V1	13 $<V_0,V_1>$	13 $<V_0,V_1>$			
V2	8 $<V_0,V_2>$	--------			
V3	∞	13 $<V_0,V_2,V_3>$			
V4	30 $<V_0,V_4>$	30 $<V_0,V_4>$			
V5	∞	∞			
V6	32 $<V_0,V_6>$	32 $<V_0,V_6>$			
Vj	V_2:8 $<V_0,V_2>$	V_1:13 $<V_0,V_1>$			



6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

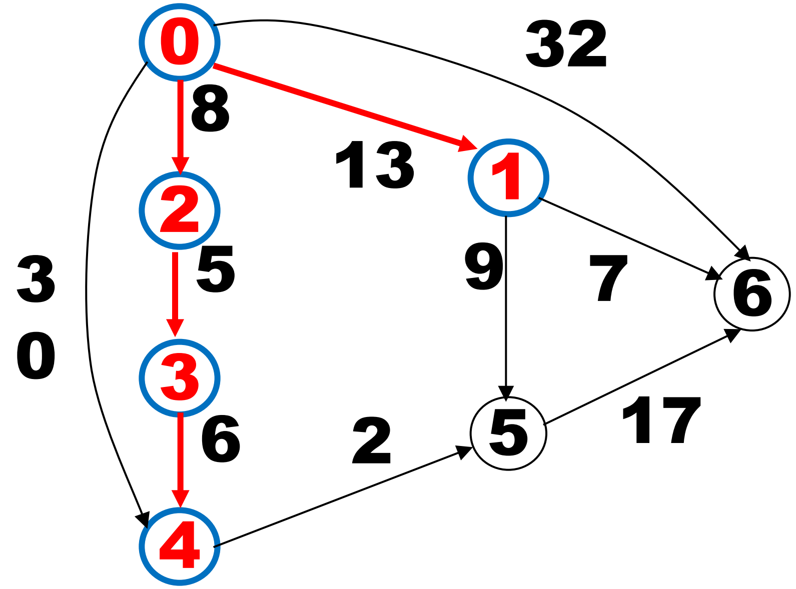
终点	从V0到各终点的最短路径及其长度				
V1	13 $<V_0,V_1>$	13 $<V_0,V_1>$	--------		
V2	8 $<V_0,V_2>$	--------	--------		
V3	∞	13 $<V_0,V_2,V_3>$	13 $<V_0,V_2,V_3>$		
V4	30 $<V_0,V_4>$	30 $<V_0,V_4>$	30 $<V_0,V_4>$		
V5	∞	∞	22 $<V_0,V_1,V_5>$		
V6	32 $<V_0,V_6>$	32 $<V_0,V_6>$	20 $<V_0,V_1,V_6>$		
Vj	V_2:8 $<V_0,V_2>$	V_1:13 $<V_0,V_1>$	V_3:13 $<V_0,V_2,V_3>$		



6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

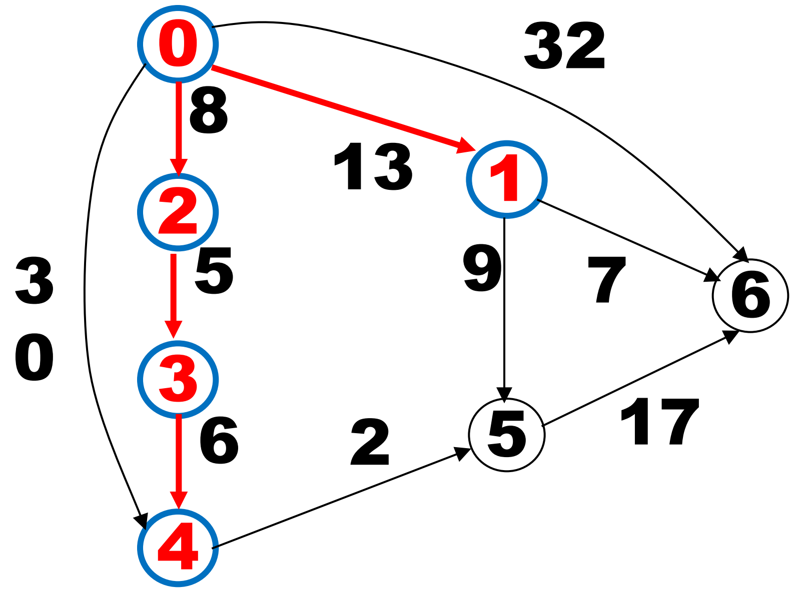
终点	从V0到各终点的最短路径及其长度				
V1	13 $<V_0,V_1>$	13 $<V_0,V_1>$	-------	-------	
V2	8 $<V_0,V_2>$	-------	-------	-------	
V3	∞	13 $<V_0,V_2,V_3>$	13 $<V_0,V_2,V_3>$	-------	
V4	30 $<V_0,V_4>$	30 $<V_0,V_4>$	30 $<V_0,V_4>$	19 $<V_0,V_2,V_3,V_4>$	
V5	∞	∞	22 $<V_0,V_1,V_5>$	22 $<V_0,V_1,V_5>$	
V6	32 $<V_0,V_6>$	32 $<V_0,V_6>$	20 $<V_0,V_1,V_6>$	20 $<V_0,V_1,V_6>$	
Vj	V_2:8 $<V_0,V_2>$	V_1:13 $<V_0,V_1>$	V_3:13 $<V_0,V_2,V_3>$	V_4:19 $<V_0,V_2,V_3,V_4>$	



6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

终点	从V0到各终点的最短路径及其长度				
V1	13 $\langle V_0,V_1\rangle$	13 $\langle V_0,V_1\rangle$	--------	--------	
V2	8 $\langle V_0,V_2\rangle$	--------	--------	--------	
V3	∞	13 $\langle V_0,V_2,V_3\rangle$	13 $\langle V_0,V_2,V_3\rangle$	--------	--------
V4	30 $\langle V_0,V_4\rangle$	30 $\langle V_0,V_4\rangle$	30 $\langle V_0,V_4\rangle$	19 $\langle V_0,V_2,V_3,V_4\rangle$	--------
V5	∞	∞	22 $\langle V_0,V_1,V_5\rangle$	22 $\langle V_0,V_1,V_5\rangle$	21 $\langle V_0,V_2,V_3,V_4,V_5\rangle$
V6	32 $\langle V_0,V_6\rangle$	32 $\langle V_0,V_6\rangle$	20 $\langle V_0,V_1,V_6\rangle$	20 $\langle V_0,V_1,V_6\rangle$	20 $\langle V_0,V_1,V_6\rangle$
Vj	V_2:8 $\langle V_0,V_2\rangle$	V_1:13 $\langle V_0,V_1\rangle$	V_3:13 $\langle V_0,V_2,V_3\rangle$	V_4:19 $\langle V_0,V_2,V_3,V_4\rangle$	V6:20 $\langle V_0, V_1,V_6\rangle$

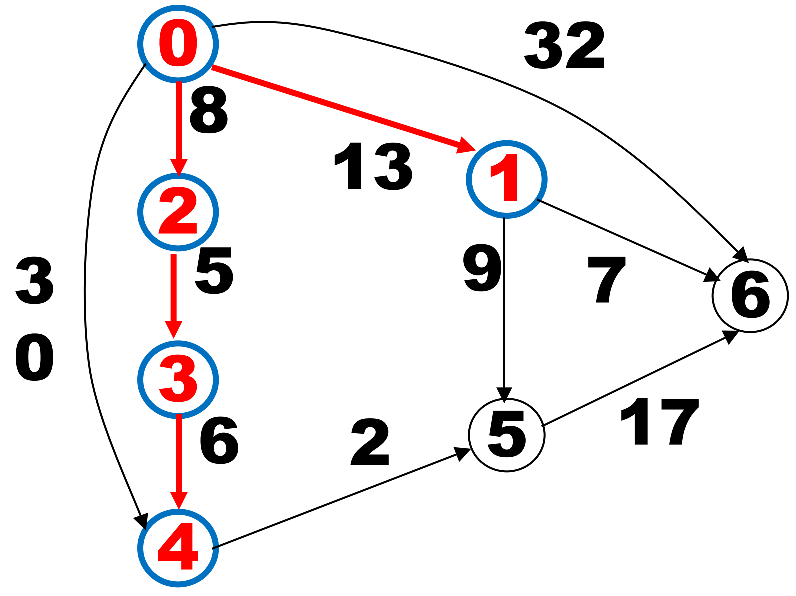


6.6 最短路径-迪杰斯特拉算法

求最短路径步骤

终点	从V0到各终点的最短路径及其长度				
V1	13 $<V_0,V_1>$	13 $<V_0,V_1>$	--------	--------	
V2	8 $<V_0,V_2>$	--------	--------	--------	
V3	∞	13 $<V_0,V_2,V_3>$	13 $<V_0,V_2,V_3>$	--------	--------
V4	30 $<V_0,V_4>$	30 $<V_0,V_4>$	30 $<V_0,V_4>$	19 $<V_0,V_2,V_3,V_4>$	--------
V5	∞	∞	22 $<V_0,V_1,V_5>$	22 $<V_0,V_1,V_5>$	21 $<V_0,V_2,V_3,V_4,V_5>$
V6	32 $<V_0,V_6>$	32 $<V_0,V_6>$	20 $<V_0,V_1,V_6>$	20 $<V_0,V_1,V_6>$	20 $<V_0,V_1,V_6>$
Vj	V_2:8 $<V_0,V_2>$	V_1:13 $<V_0,V_1>$	V_3:13 $<V_0,V_2,V_3>$	V_4:19 $<V_0,V_2,V_3,V_4>$	V6:20 $<V_0, V_1,V_6>$

21 $<V_0,V_2,V_3,V_4,V_5>$



6.6 最短路径-迪杰斯特拉算法

```
voidShortestPath(MGraph& G,intv,Weightdist[ ],intpath[ ])
{      //求带权有向图G中从源点v到其他顶点的最短路径
       intS[maxVertices];
       n =G.verticeNum;
      for(i= 0;i< n;i++ ) {//初始化数组S、path和dist
          dist[i] =G.Edge[v][i];  S[i] = 0;
          if (dist[i] <maxWeight)path[i] = v;
          else path[i] =-1;
       } //for
       path[v]= v;   S[v] = 1;dist[v] = 0;
       for (i= 0;i< n-1;i++ ) {
            //求到其他点最短路径求到其他点最短路径
       }//for (i= 0;...)
}//ShortestPath
```

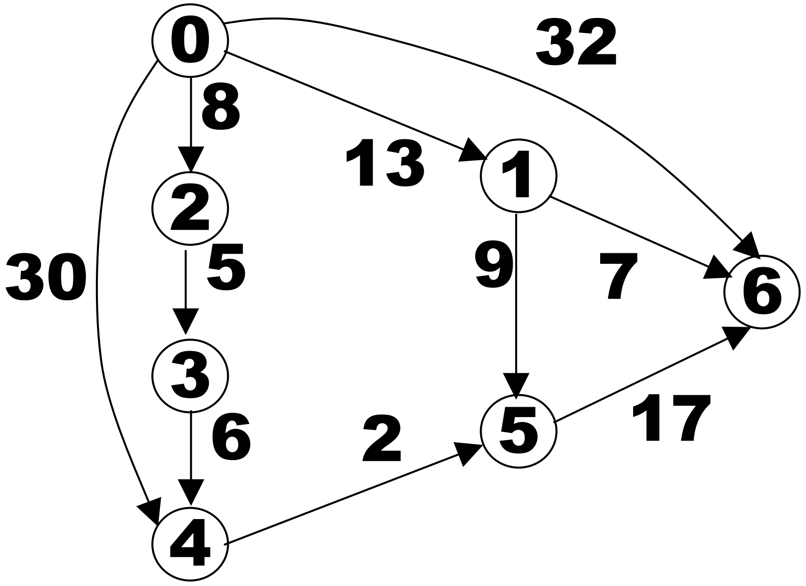
6.6 最短路径-迪杰斯特拉算法

```
voidShortestPath(MGraph& G,intv,Weightdist[ ],intpath[ ])
{
    ……
    for (i= 0;i< n-1;i++ ) {
        //求到其他点最短路径求到其他点最短路径
        u =SelectMin(dist);  //选不在S中具有最短路径顶点u
        S[u] = 1;//将顶点u加入集合S
        for( k = 0; k < n; k++ ){  //修改其他顶点的路径长度，松弛
            if(!S[k]&&dist[u] +G.Edge[u][k]<dist[k]) {
                //顶点k未加入S,且v到k经过u的路径更短
                dist[k] =dist[u] +G.Edge[u][k];
                path[k] = u;
            }  //if ( !S[k] &&...
        }
    }//for (i= 0;...)
}//ShortestPath
```

换一种方法

- 采用优先队列的迪杰斯特拉算法

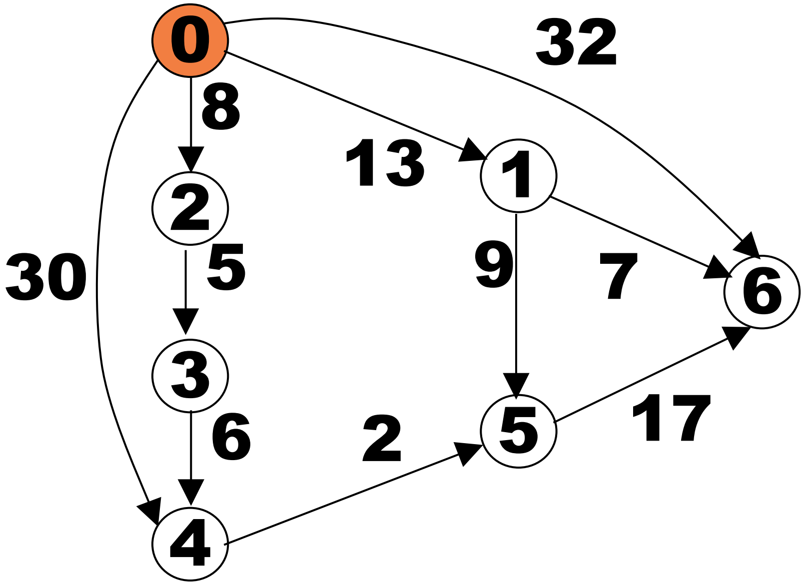
6.6 最短路径-迪杰斯特拉算法



Priority queue

vertex	dist	path
V_1	∞	0
V_2	∞	0
V_3	∞	0
V_4	∞	0
V_5	∞	0
V_6	∞	0
V_7	∞	0

6.6 最短路径-迪杰斯特拉算法



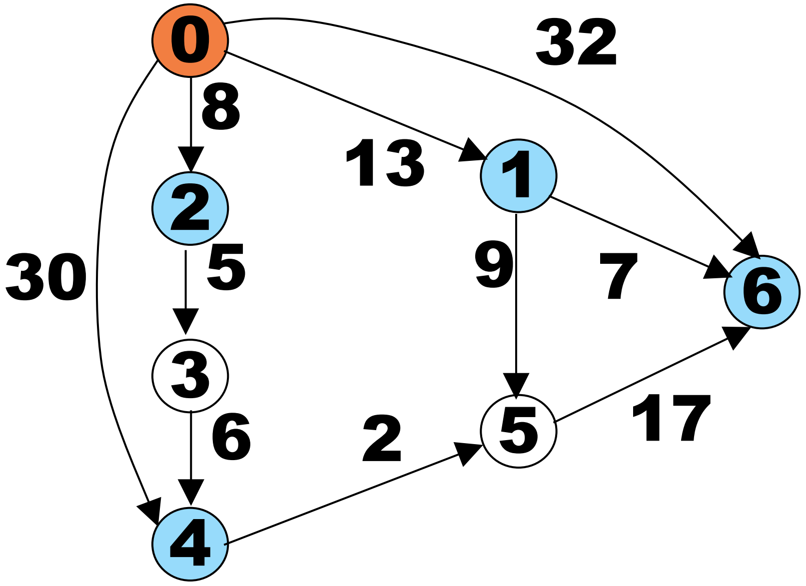
Priority queue

V_0	0

vertex	dist	path
V_0	0	0
V_1	∞	0
V_2	∞	0
V_3	∞	0
V_4	∞	0
V_5	∞	0
V_6	∞	0

入队：V_0

6.6 最短路径-迪杰斯特拉算法



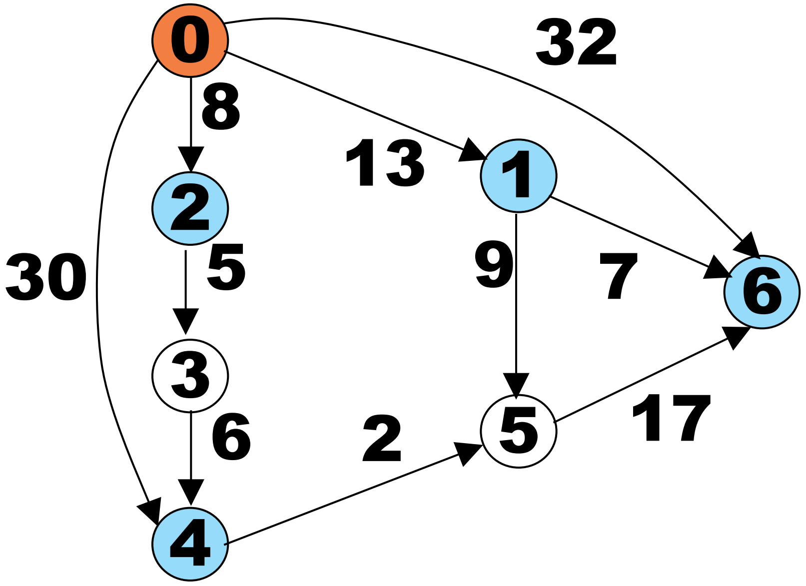
Priority queue

V_1	13
V_2	8
V_4	30
V_6	32

vertex	dist	path
V_0	*0*	*0*
V_1	*13*	V_0
V_2	*8*	V_0
V_3	*∞*	*0*
v_4	*30*	V_0
V_5	*∞*	*0*
V_6	*32*	V_0

V_0出队，V_1，V_2，V_4，V_6入队

6.6 最短路径-迪杰斯特拉算法

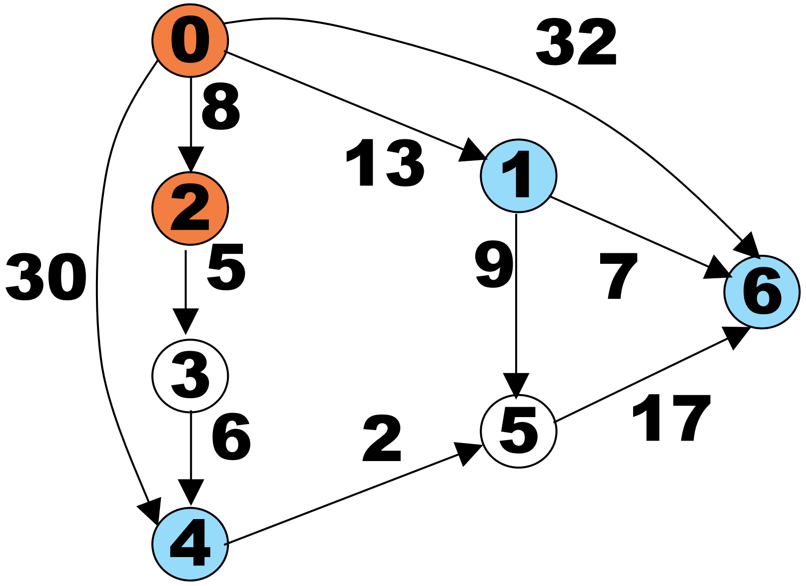


Priority queue

V_2	8
V_1	13
V_4	30
V_6	32

vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	∞	0
V_4	30	V_0
V_5	∞	0
V_6	32	V_0

6.6 最短路径-迪杰斯特拉算法



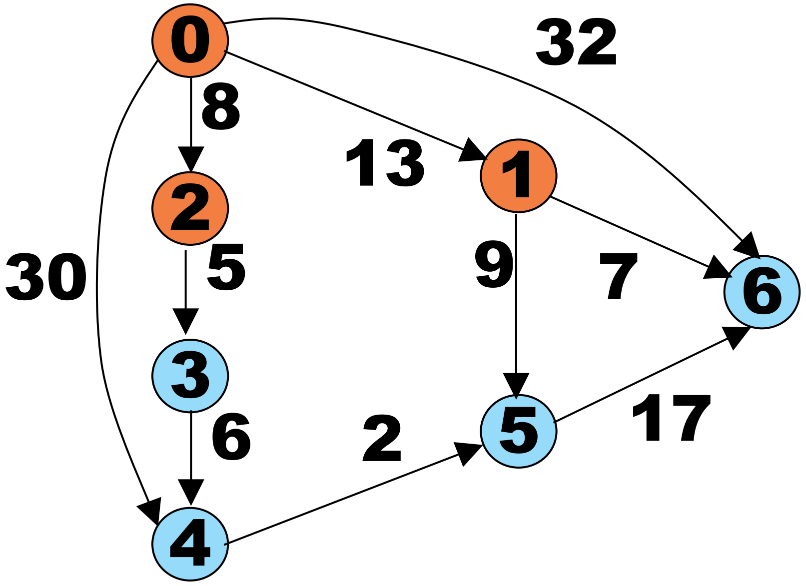
Priority queue

V_1	13
V_3	13
V_4	30
V_6	32

vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	13	V_2
V_4	30	V_0
V_5	∞	0
V_6	32	V_0

V_2出队，V_3入队

6.6 最短路径-迪杰斯特拉算法



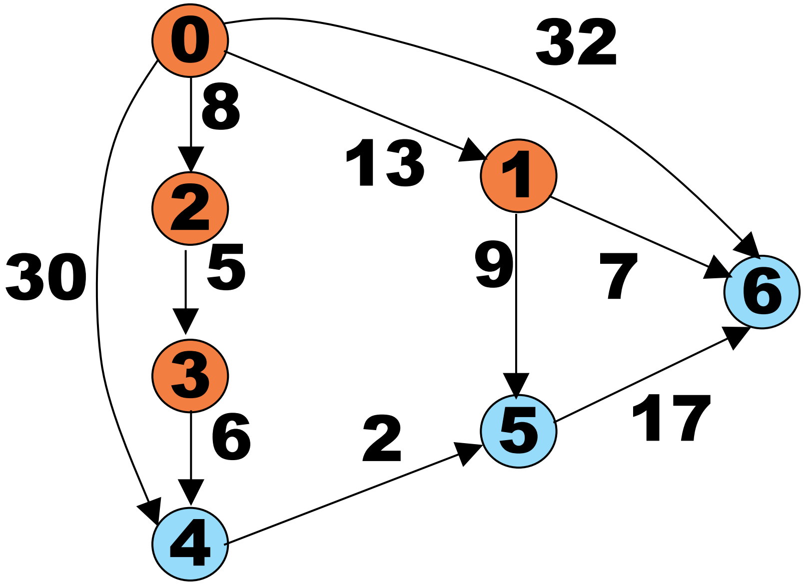
Priority queue

V_3	13
V_6	20
V_5	22
V_4	30

vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	13	V_2
v_4	30	V_0
V_5	22	V_1
V_6	20	V_1

V_1出队，V_5入队

6.6 最短路径-迪杰斯特拉算法



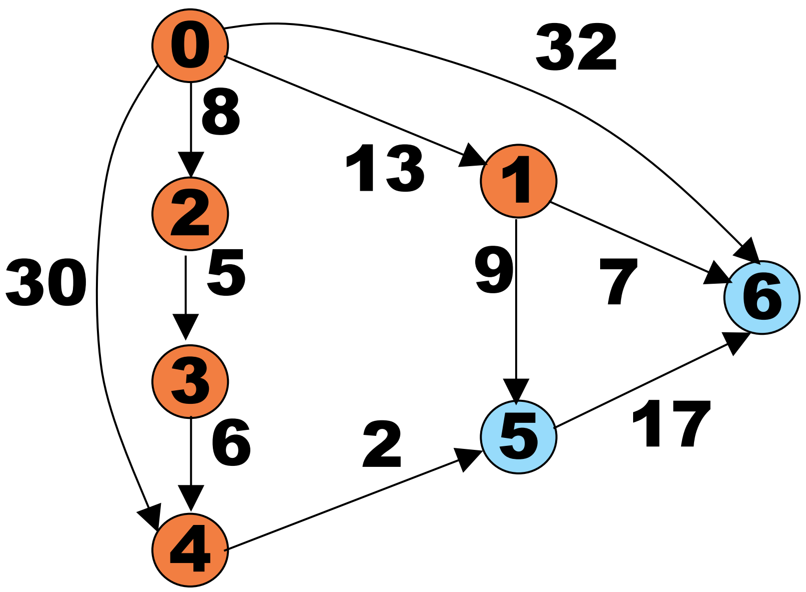
Priority queue

V_4	19
V_6	20
V_5	22

vertex	dist	path
V_0	*0*	*0*
V_1	*13*	V_0
V_2	*8*	V_0
V_3	*13*	*V2*
v_4	*19*	V_3
V_5	*22*	*V1*
V_6	*20*	V_1

V_3出队

6.6 最短路径-迪杰斯特拉算法



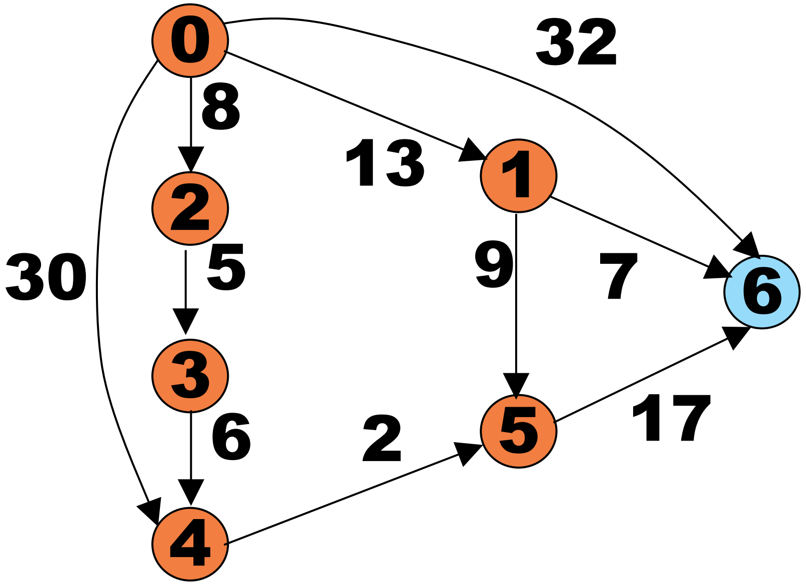
Priority queue

V_6	20
V_5	21

vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	13	V2
v_4	19	V_3
V_5	21	V4
V_6	20	V_1

V_4出队

6.6 最短路径-迪杰斯特拉算法



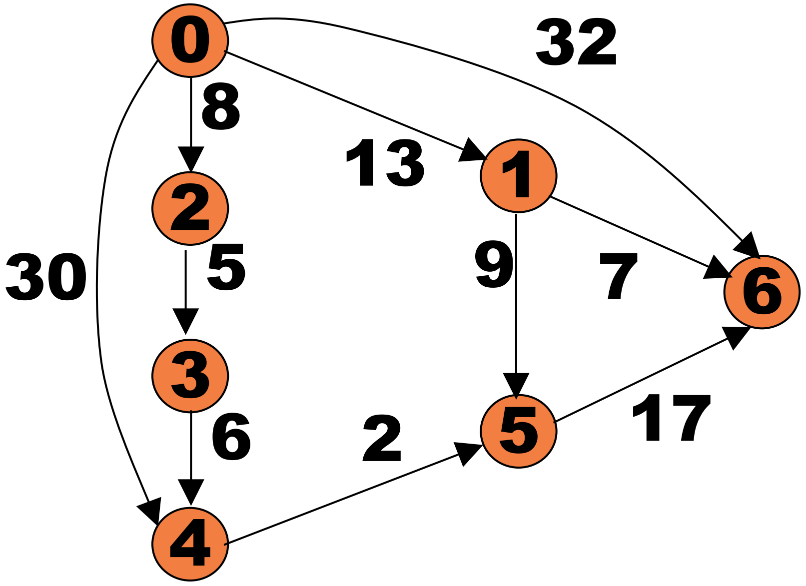
Priority queue

V_5	21

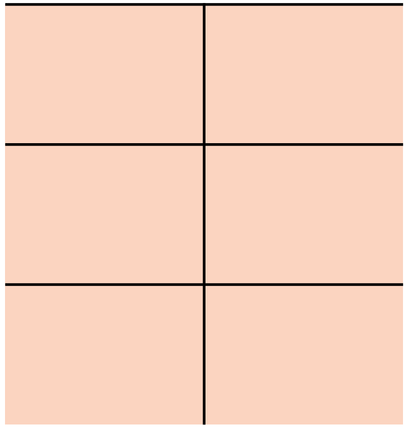
vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	13	V2
v_4	19	V_3
V_5	21	V4
V_6	20	V_1

V_6出队

6.6 最短路径-迪杰斯特拉算法



Priority queue



vertex	dist	path
V_0	0	0
V_1	13	V_0
V_2	8	V_0
V_3	13	V2
v_4	19	V_3
V_5	21	V4
V_6	20	V_1

V_5出队

栈空：结束

6.6 最短路径-迪杰斯特拉算法

Dijkstra算法

复杂度分析：

采用邻接矩阵的方法，复杂度为$O(n^2)$

采用优先队列的方法：

1. 优先队列中要访问所有顶点和所有边；。
2. 优先队列的排序采用堆来实现。
3. 最终的复杂度为$O((|V|+|E|)*\log|V|)$。

6.6 最短路径-迪杰斯特拉算法

带权有向图的某几条边或所有边的长度可能为负值。
用Dijkstra算法不一定能得到正确的结果。

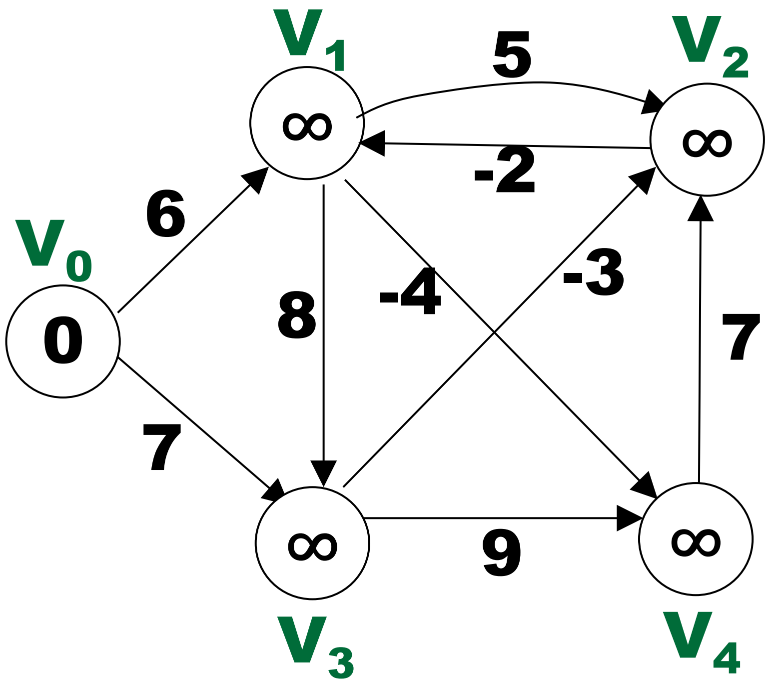
带权有向图的某几条边或所有边的长度可能为负值。
用Dijkstra算法不一定能得到正确的结果。

- 负回路—最短路径无定义
- 正回路—最短路径算法不可以求解
- 零回路—可以找到不包括回路的简单路径

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法

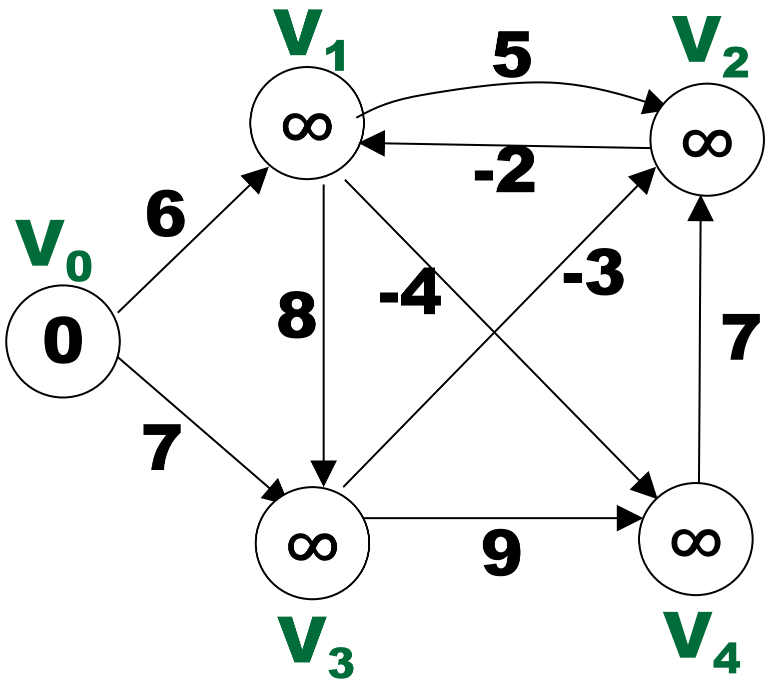


V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	∞	7	∞

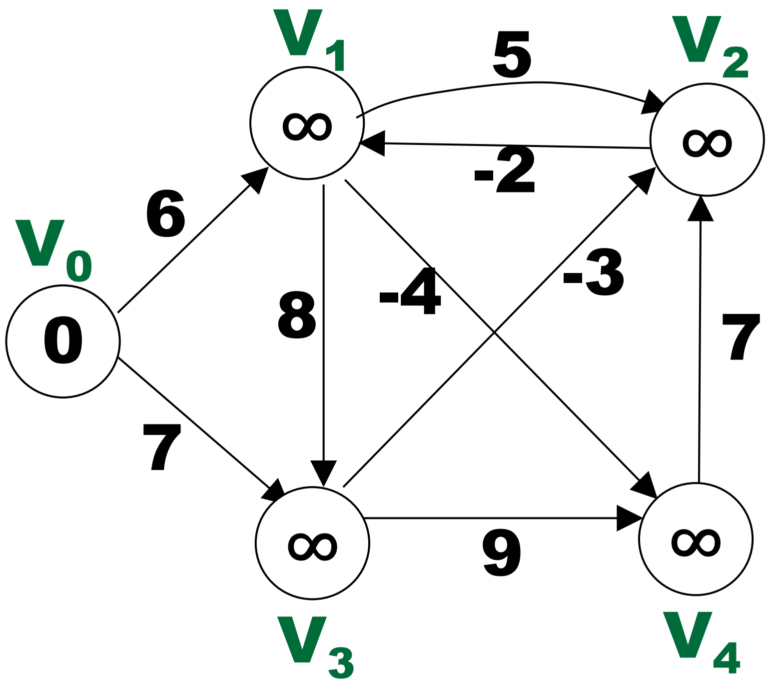
第1次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	11	7	∞

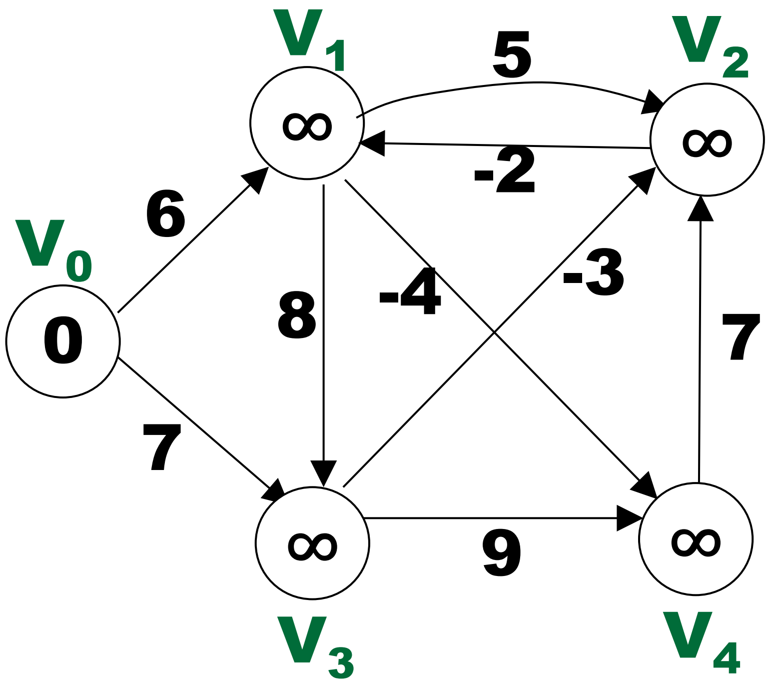
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	11	7	∞

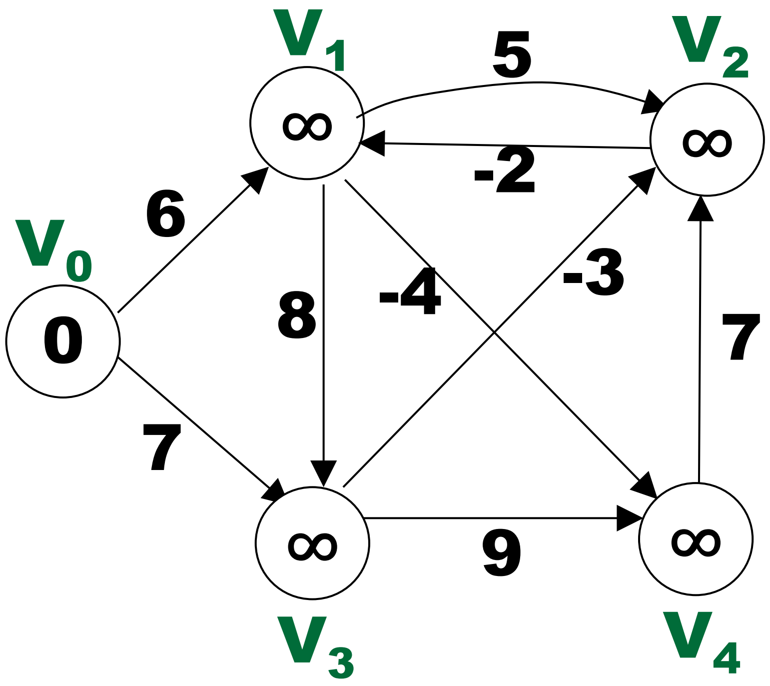
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	11	7	2

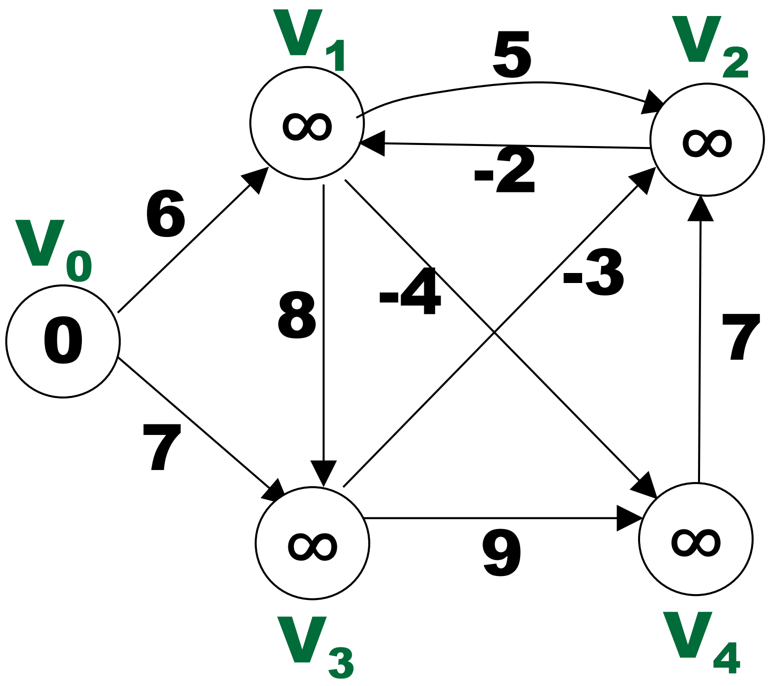
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	11	7	2

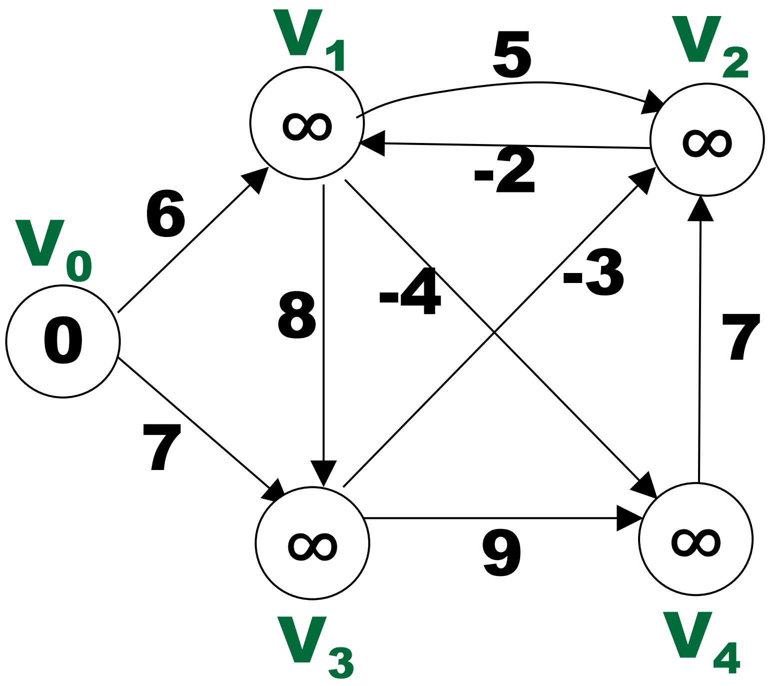
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	4	7	2

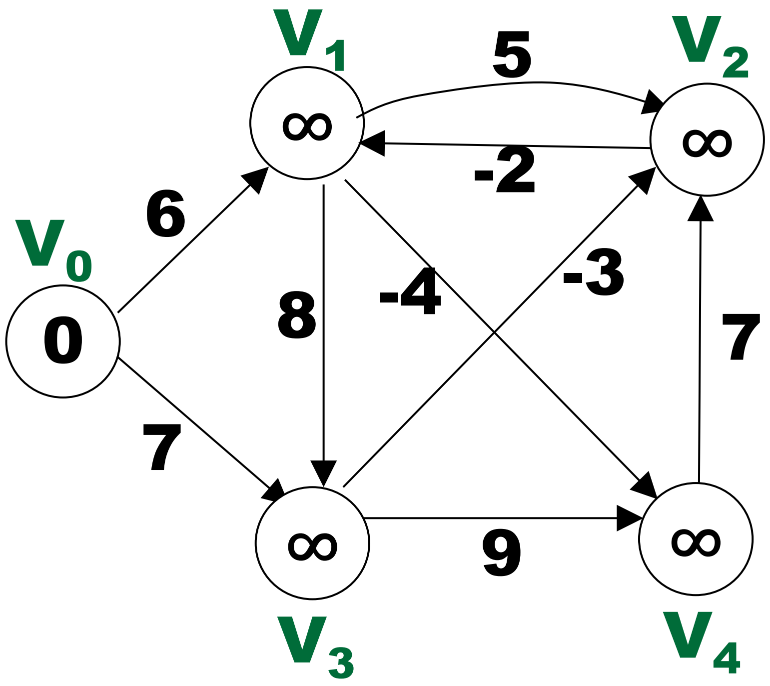
第2次考察：

V_1V_2， V_1V_3， V_1V_4， V_2V_1， V_3V_2， V_3V_4， V_4V_2， V_0V_1， V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	4	7	2

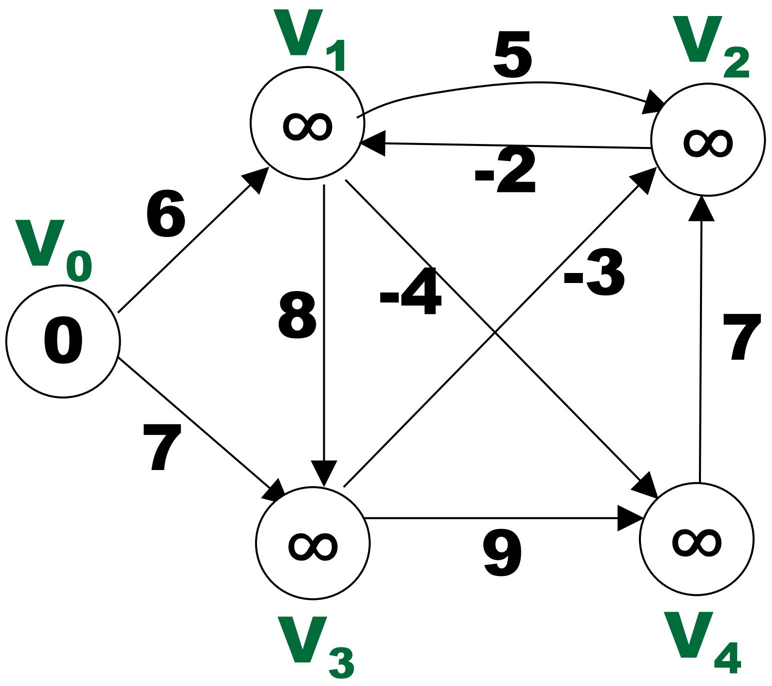
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	6	4	7	2

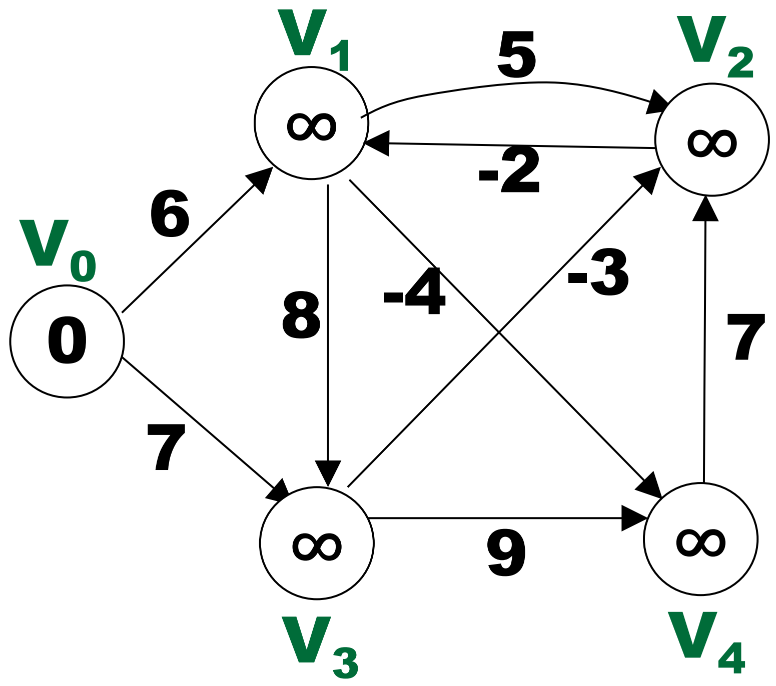
第2次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	2	4	7	2

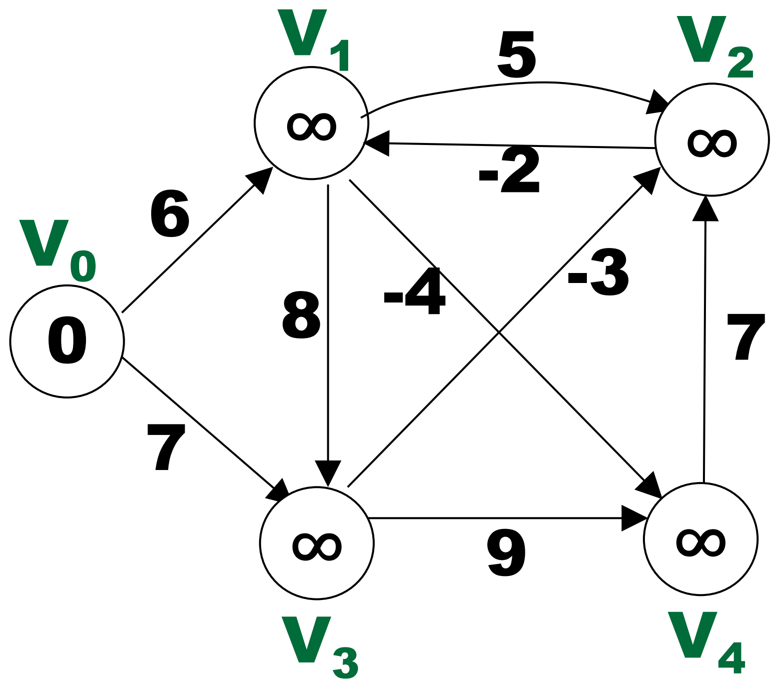
第3次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

单源最短路径问题，顶点V_0到其它顶点间最短路径

♦基本思想：逐条边试探法

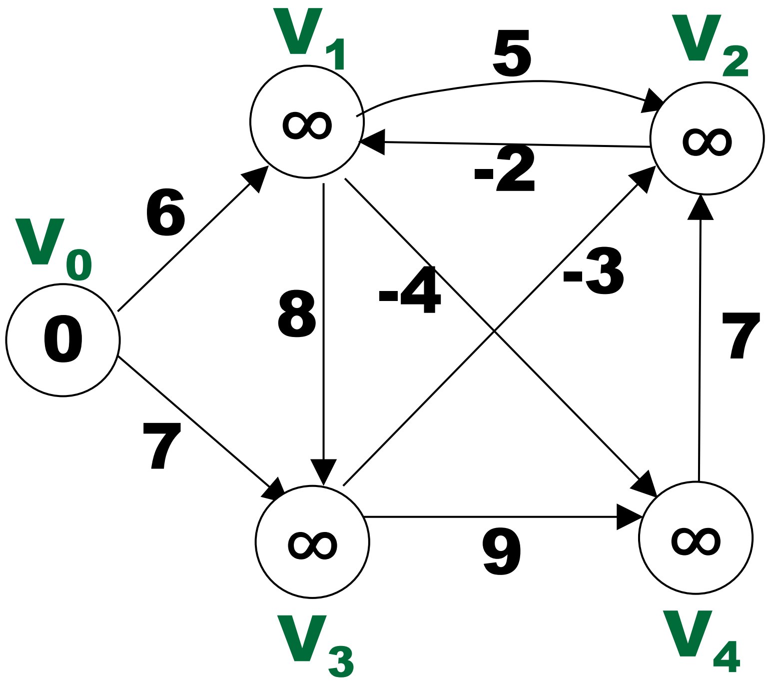


V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	2	4	7	-2

第4次考察：

V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法



V_0	V_1	V_2	V_3	V_4
0	∞	∞	∞	∞
0	2	4	7	-2

1：V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

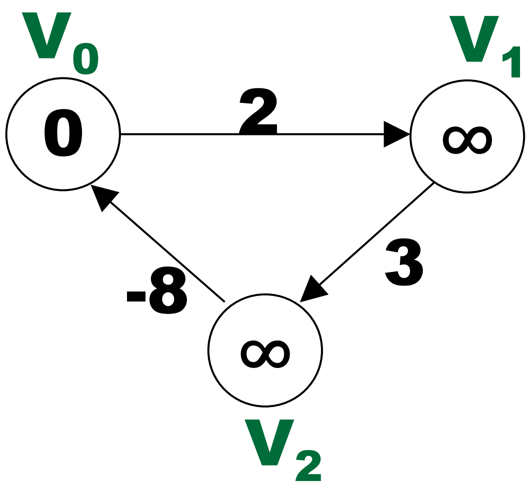
2：V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

3：V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

4：V_1V_2，V_1V_3，V_1V_4，V_2V_1，V_3V_2，V_3V_4，V_4V_2，V_0V_1，V_0V_3

6.6 最短路径-Bellman-Ford算法

图中有负回路的情况



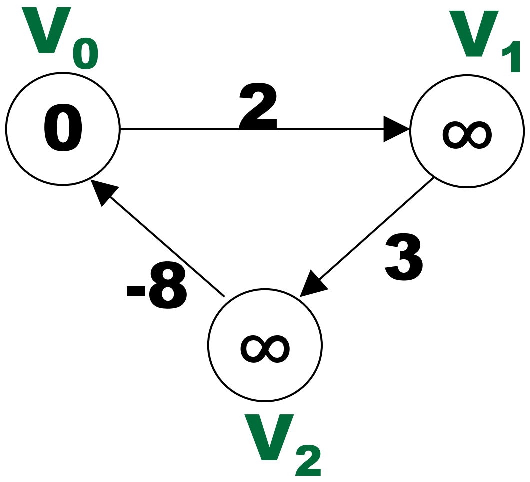
V_0	V_1	V_2
0	∞	∞
0	2	∞

第1次考察：

V_0V_1，V_1V_2，V_2V_0

6.6 最短路径-Bellman-Ford算法

图中有负回路的情况



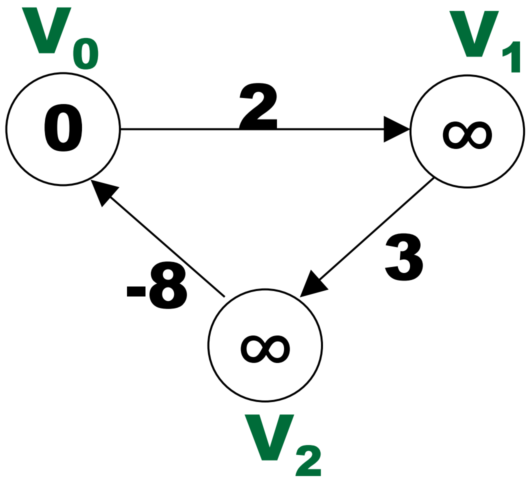
V_0	V_1	V_2
0	∞	∞
-3	2	5

第2次考察：

V_0V_1, V_1V_2, V_2V_3

6.6 最短路径-Bellman-Ford算法

图中有负回路的情况



V_0	V_1	V_2
0	∞	∞
-3	-1	2

第3次考察：

V_0V_1，V_1V_2，V_2V_3

若存在某个顶点，其距离再次减小，则有负回路。

6.6 最短路径-Bellman-Ford算法

算法过程：

♦1.初始化所有结点到源点距离为∞;

♦2. for(i=1;i<n;i++)

♦对每一条边<u, v>依次进行下列判断（松弛操作）

¶如果dist[v] >dist[u]+w(u, v)

¶则dist[v]=dist[u] +w(u, v) ;

♦3.判断图中是否有负回路：

在所有n-1个顶点循环完毕后，再次对每一条边<u,v>依次进行下列判断：

¶如果dist[v] >dist[u] + w(u, v)

¶则有负回路，return false;

时间复杂度：（n-1）*E+E = n*E

6.6 最短路径

弗洛伊德(Floyd)算法

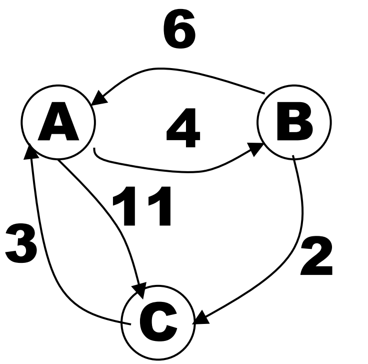
算法思想：逐个顶点试探法

求最短路径步骤

1. 初始时设置一个 n 阶方阵，令其对角线元素为 0，若存在弧 <Vi, Vj>，则对应元素为权值；否则为∞。
2. 逐步试着在原直接路径中增加中间顶点，若加入中间点后路径变短，则修改之；否则，维持原值。
3. 所有顶点试探完毕，算法结束。

6.6 最短路径

例



初始：

$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \infty & 0 \end{bmatrix}$$

路径：

0	AB	AC
BA	0	BC
CA	∞	0

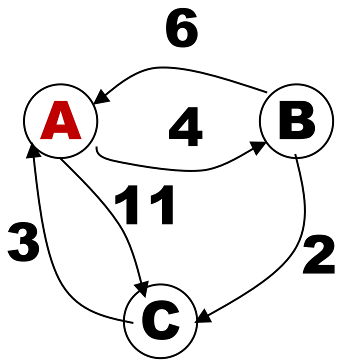
加入V1点（A）

考察：<v2, v3> = 2

<v2, v1> <v1,v3> = 17

<v3, v2> = ∞

<v3, v1> <v1,v2> = 7



加入V1：

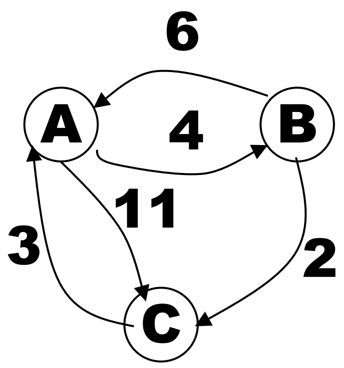
$$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$$

路径：

	AB	AC
BA		BC
CA	CAB	

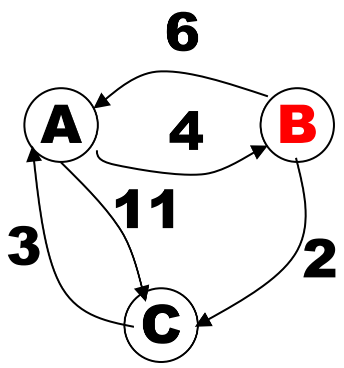
6.6 最短路径

例



加入V1：$\begin{bmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$ 路径：

	AB	AC
BA		BC
CA	CAB	



加入V2点（B）

考察：<v1, v3> = 11

<v1, v2> <v2, v3> = 6

<v3, v1> = 3

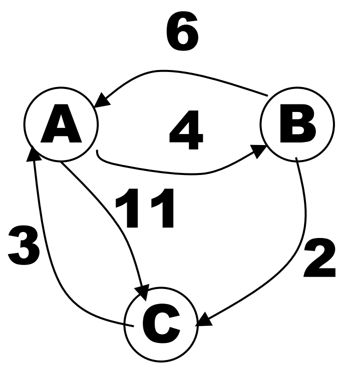
<v3, v2> <v2, v1> = ∞

加入V2：$\begin{bmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{bmatrix}$ 路径：

	AB	ABC
BA		BC
CA	CAB	

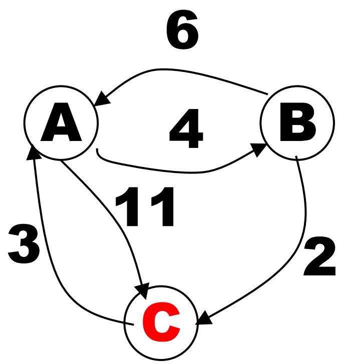
6.6 最短路径

例



加入V2: $\begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径:

	AB	ABC
BA		BC
CA	CAB	



加入V3点（C）

考察：<v1, v2> = 4

<v1, v3> <v3, v2> = 13

<v2, v1> = 6

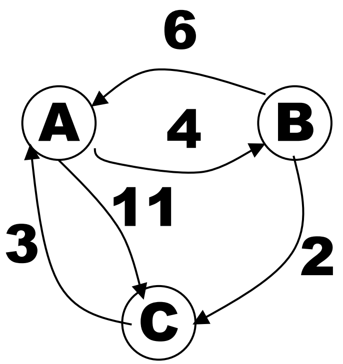
<v2, v3> <v3, v1> = 5

加入V3: $\begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径:

	AB	ABC
BCA		BC
CA	CAB	

6.6 最短路径

例



初始: $\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & \square & 0 \end{pmatrix}$ 路径:

	AB	AC
BA		BC
CA		

加入V1: $\begin{pmatrix} 0 & 4 & 11 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径:

	AB	AC
BA		BC
CA	CAB	

加入V2: $\begin{pmatrix} 0 & 4 & 6 \\ 6 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径:

	AB	ABC
BA		BC
CA	CAB	

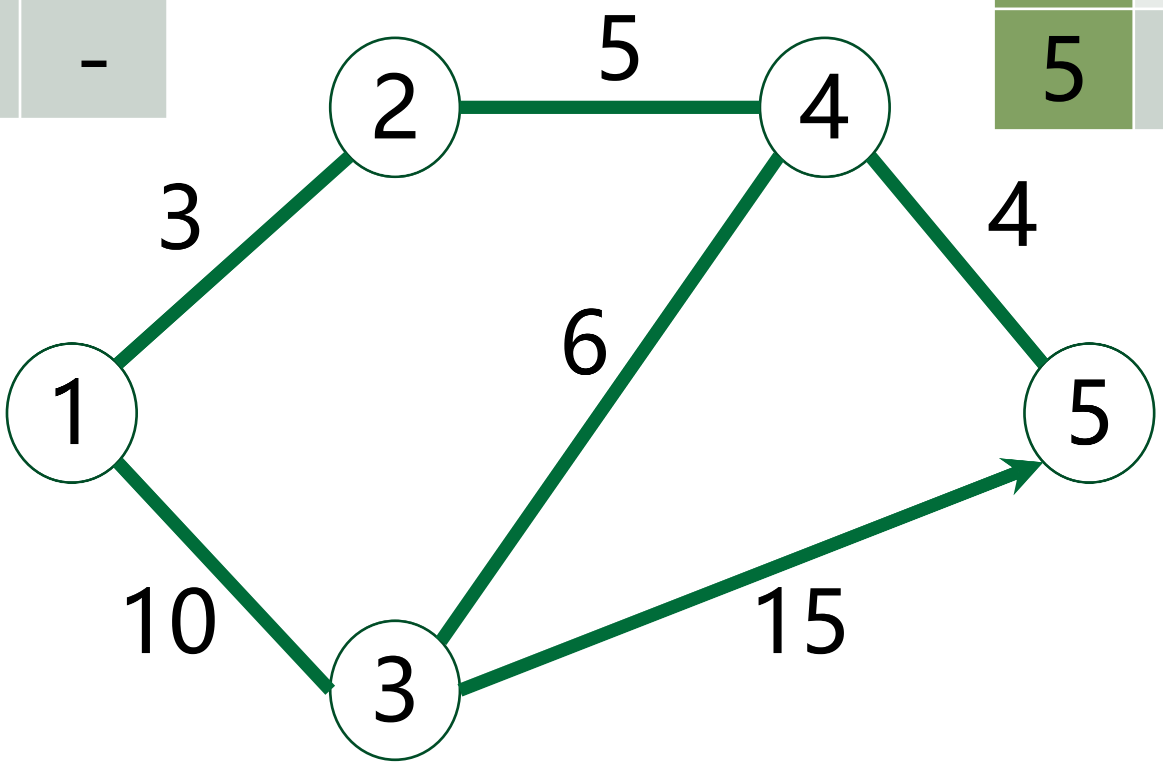
加入V3: $\begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 2 \\ 3 & 7 & 0 \end{pmatrix}$ 路径:

	AB	ABC
BCA		BC
CA	CAB	

换一种方法

D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

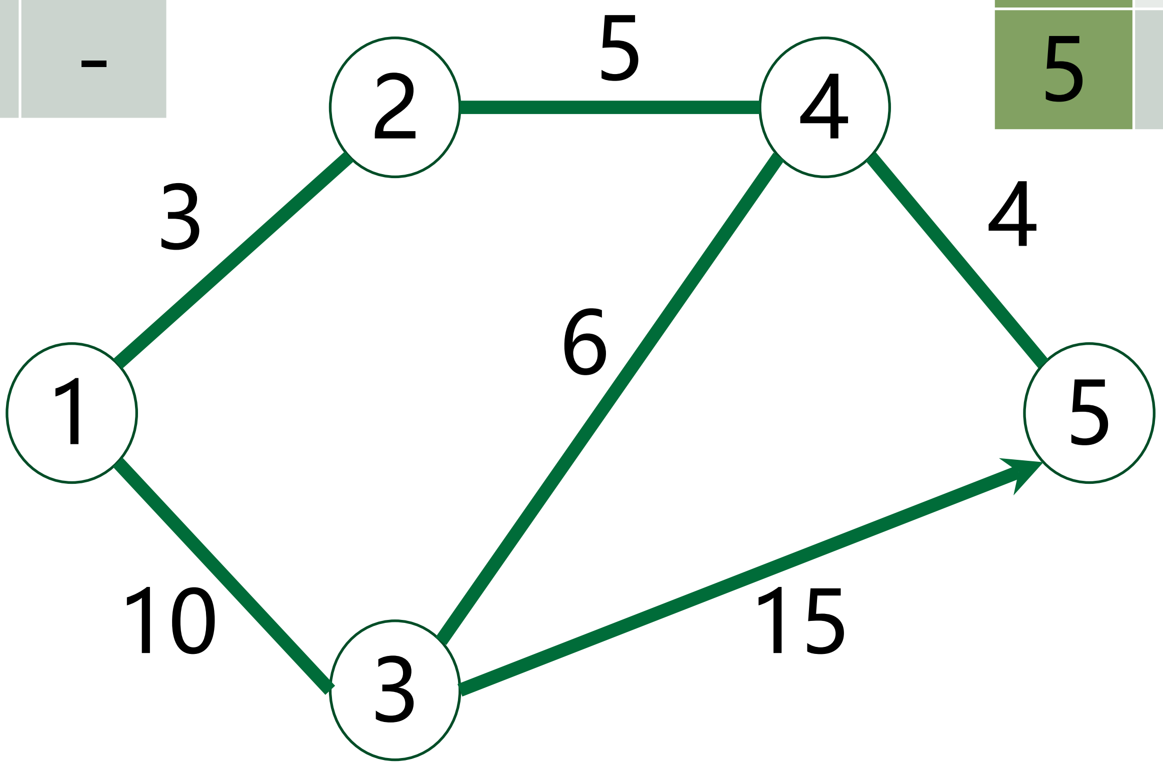
S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

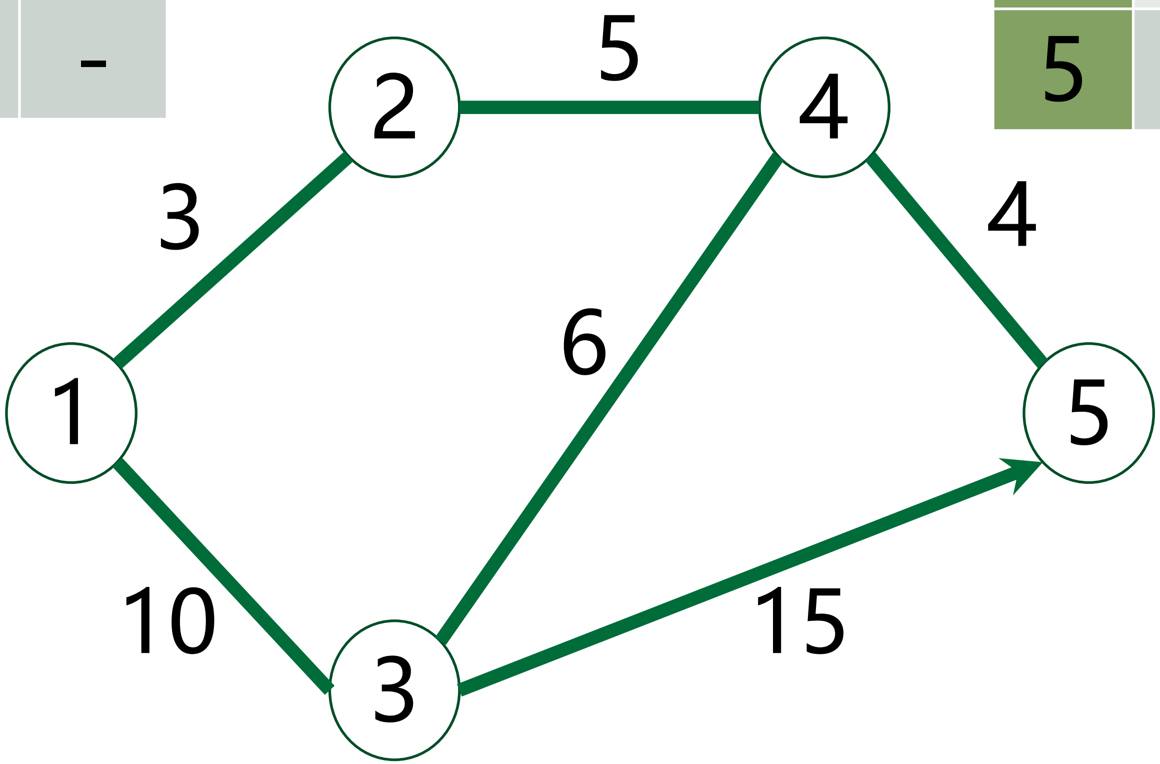


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5…**
对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**

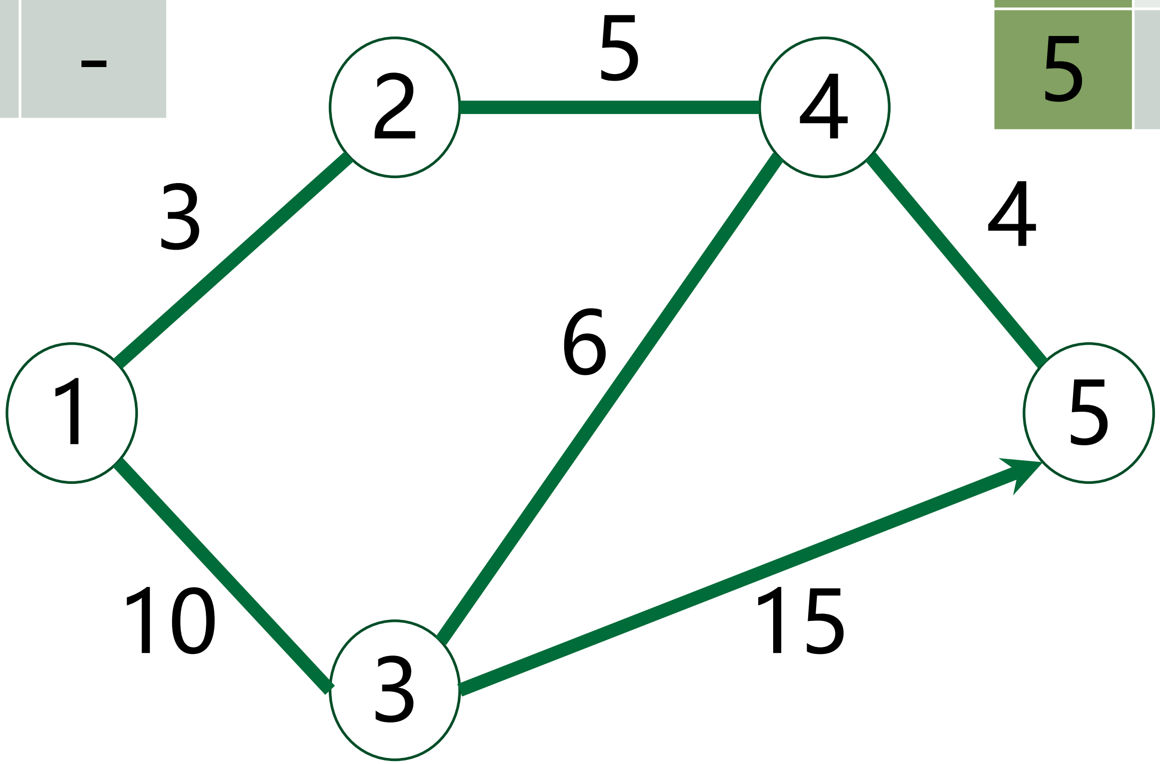


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

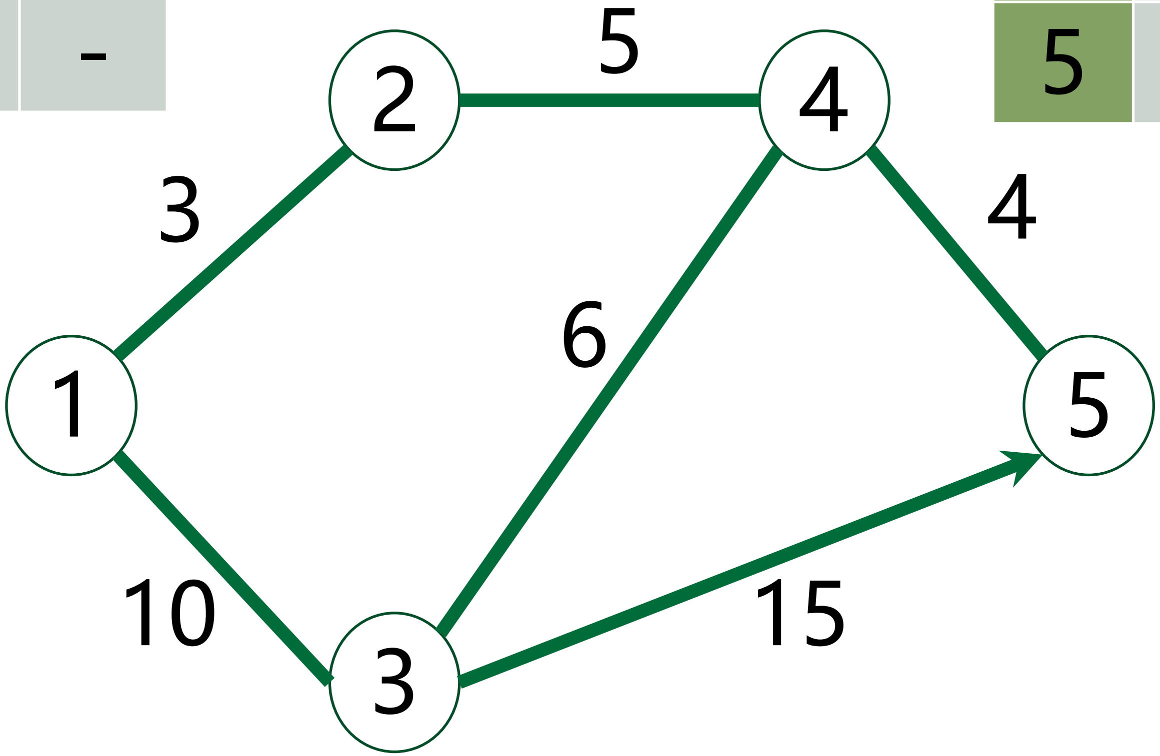


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5…**
对应的**i**列（$\mathbf{D_{ik}}$）**+** 对应的**j**行（$\mathbf{D_{kj}}$）
是否小于 $\mathbf{D_{ij}}$

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

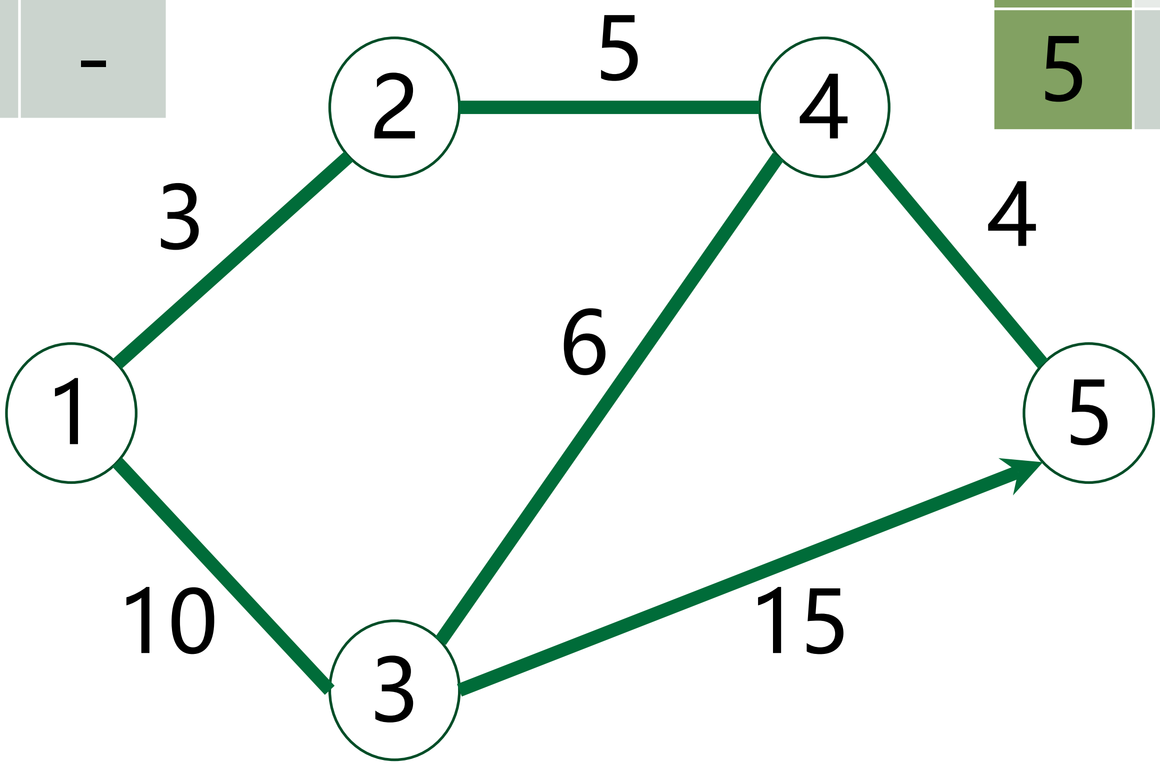


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**

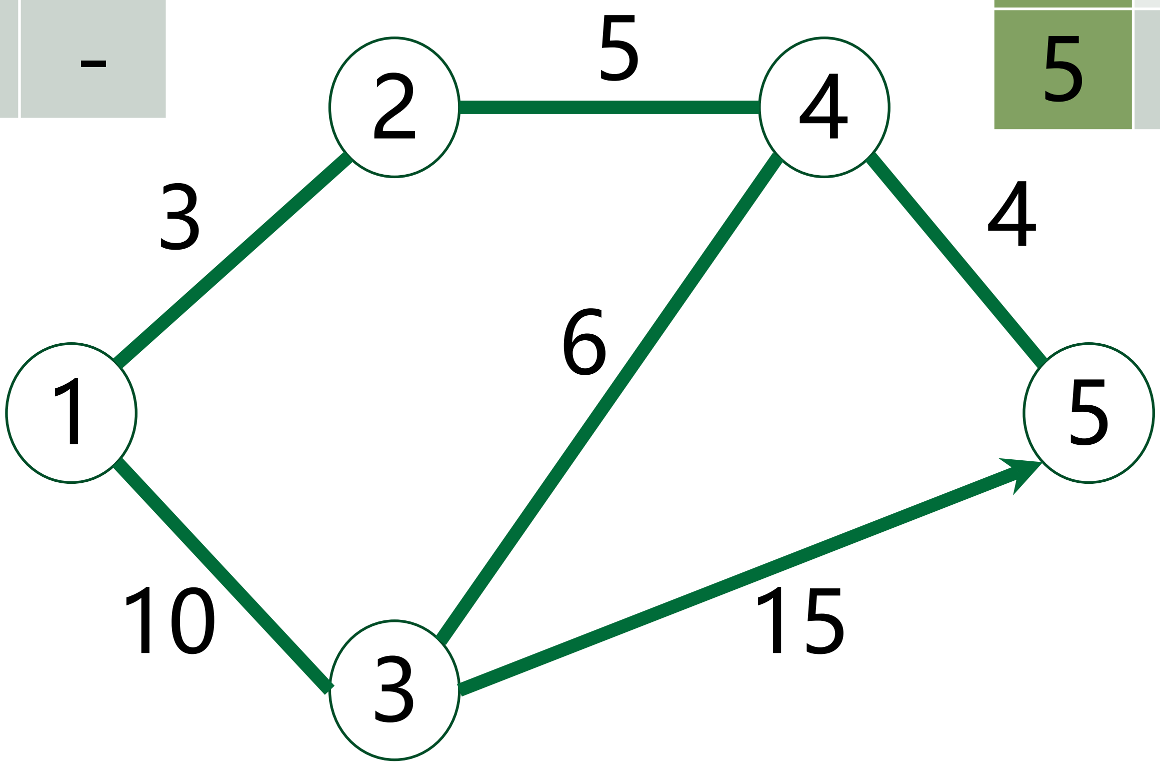


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（$\mathbf{D_{ik}}$）**+** 对应的**j**行（$\mathbf{D_{kj}}$）
是否小于 $\mathbf{D_{ij}}$

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察

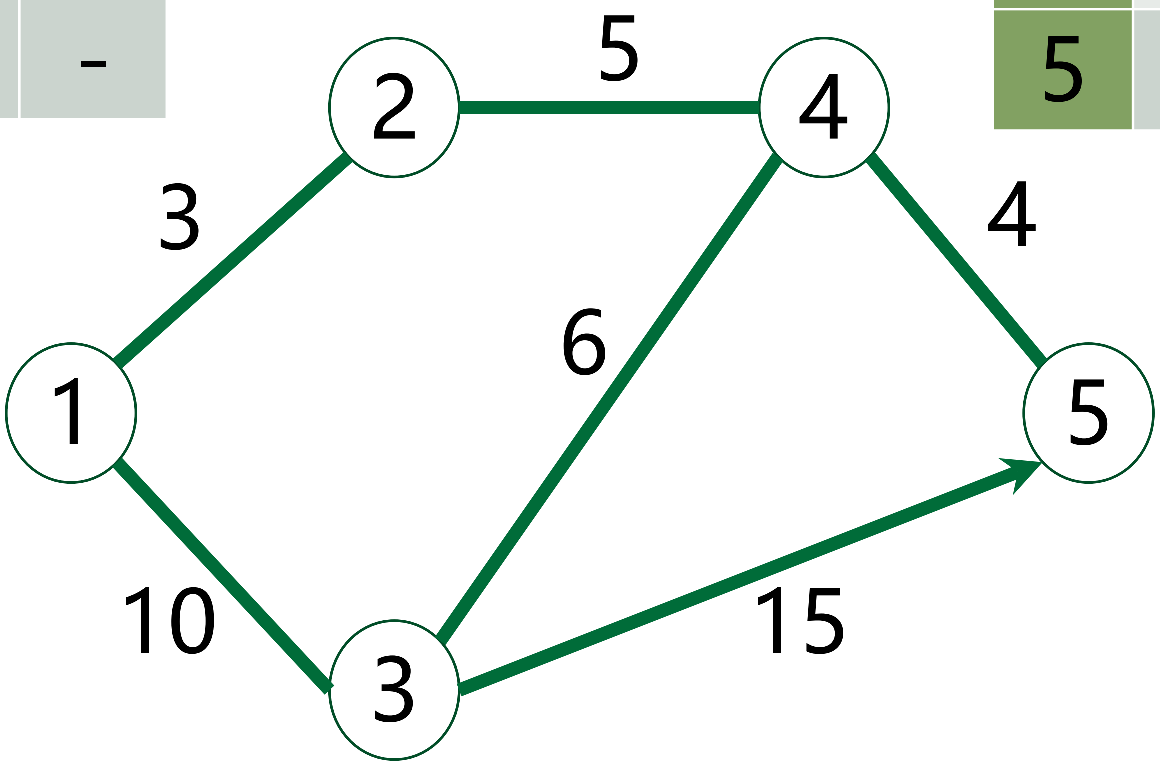
k=1的时候，**i,j**分别=**2**，**3**，**4**，**5...**

对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）

是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

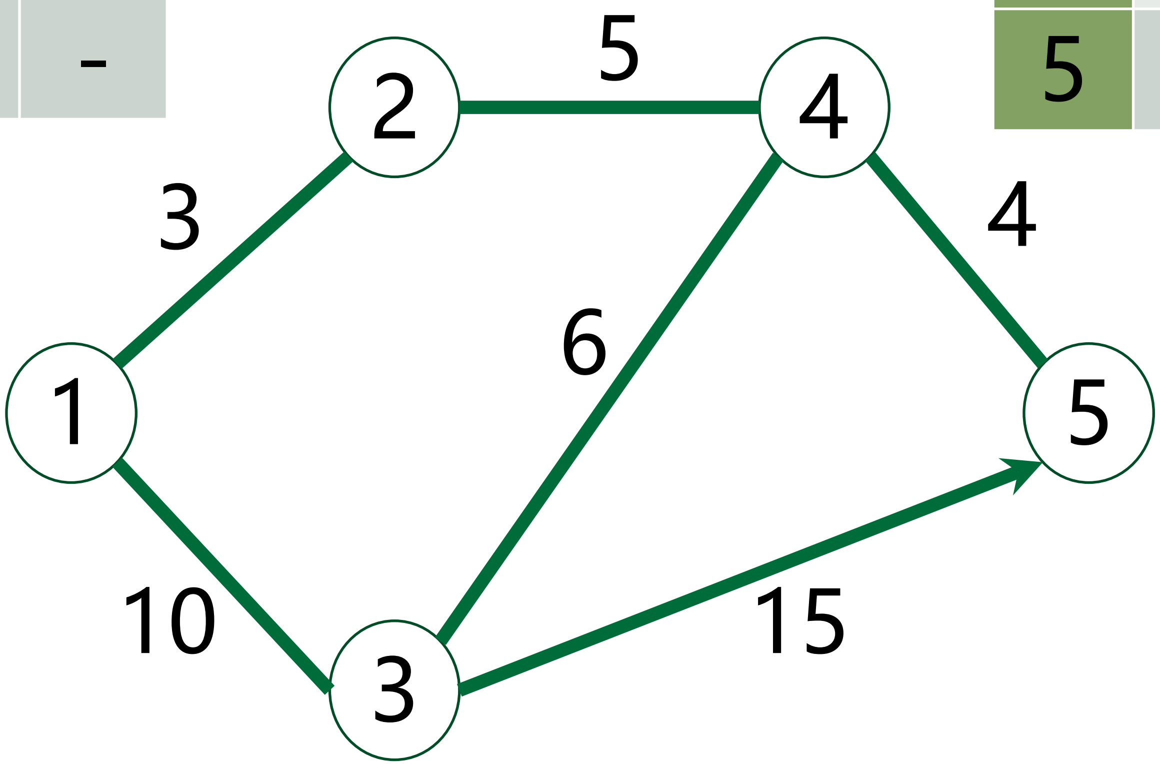


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**

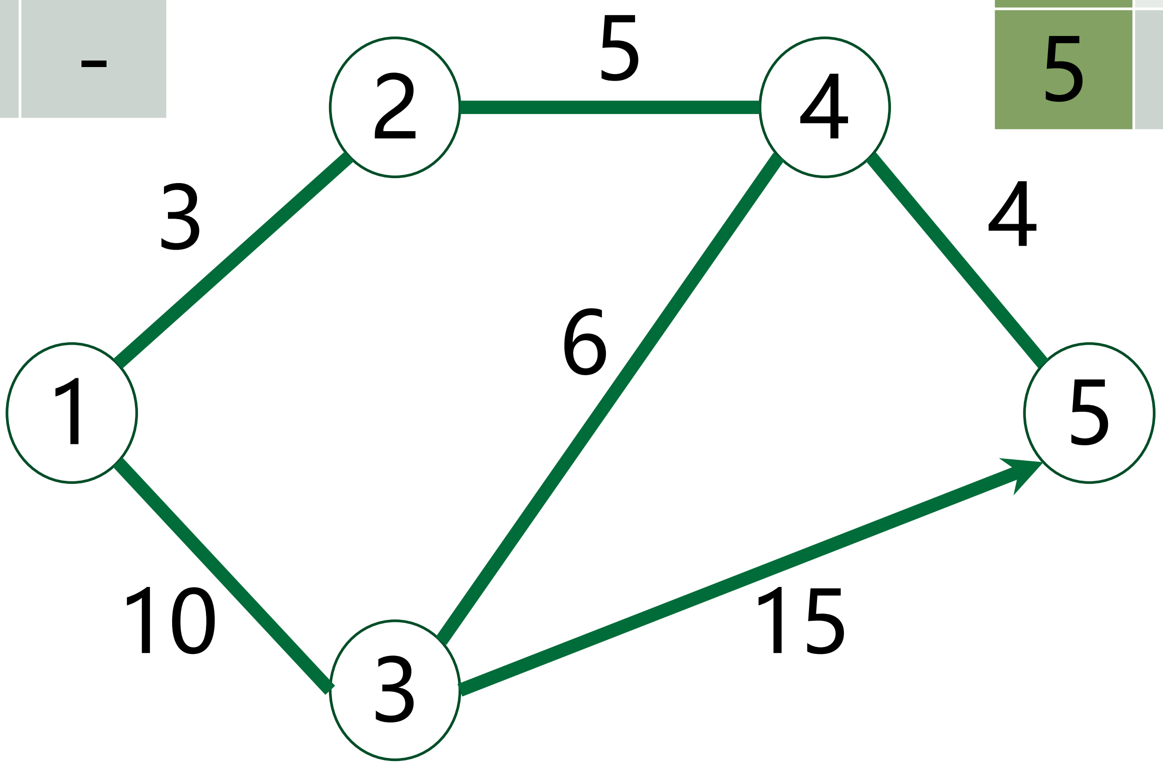


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5…**
对应的**i**列（$\mathbf{D_{ik}}$）**+** 对应的**j**行（$\mathbf{D_{kj}}$）
是否小于 $\mathbf{D_{ij}}$

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

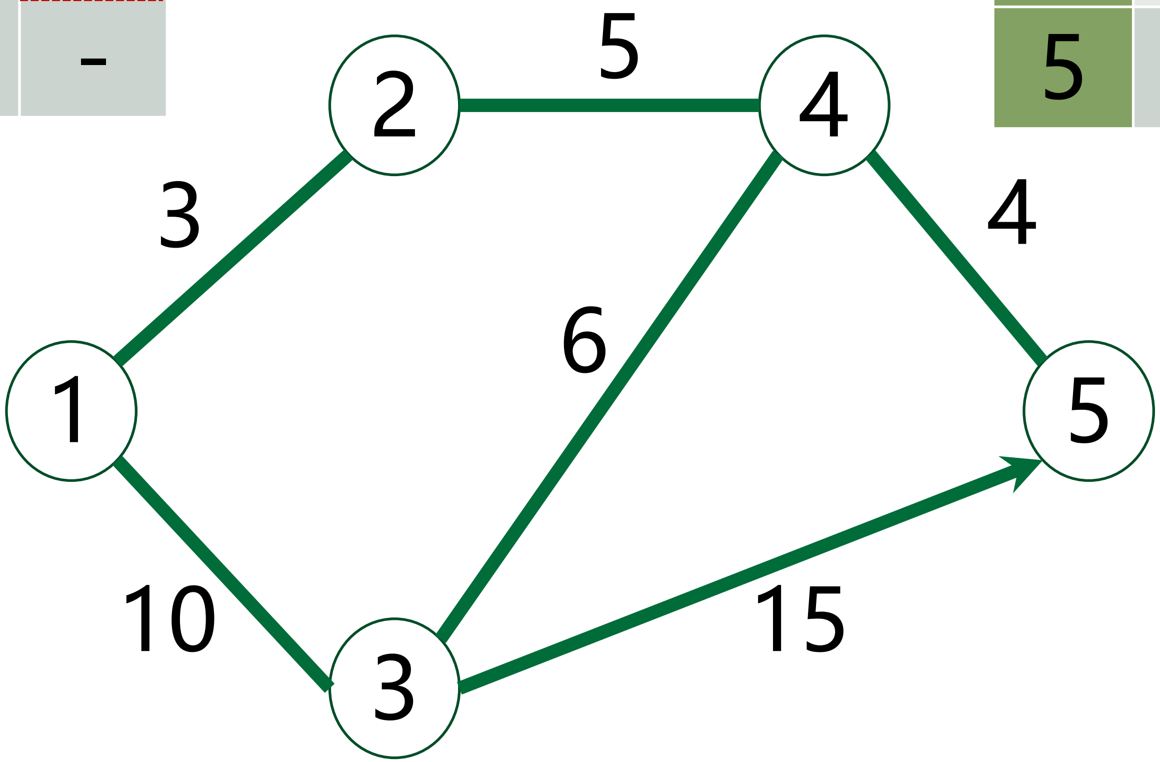


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（$\mathbf{D_{ik}}$）**+** 对应的**j**行（$\mathbf{D_{kj}}$）
是否小于 $\mathbf{D_{ij}}$

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察

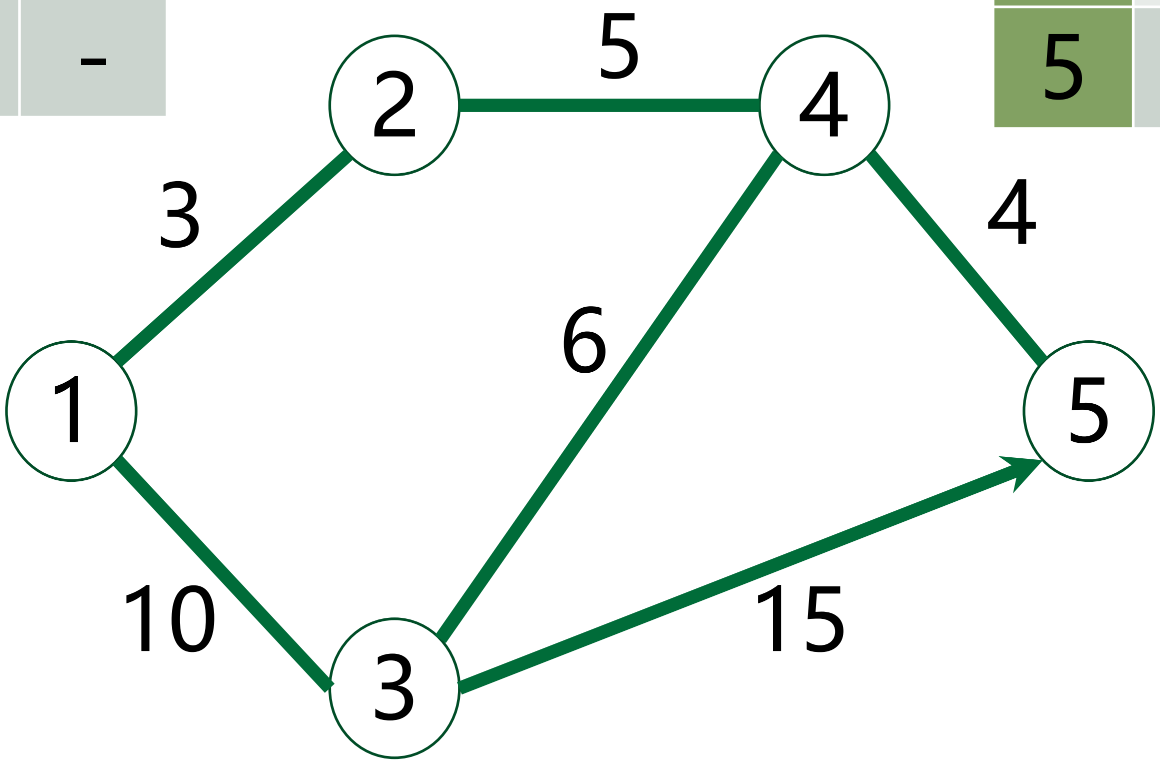
k=1的时候，**i,j**分别=**2**，**3**，**4**，**5...**

对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）

是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

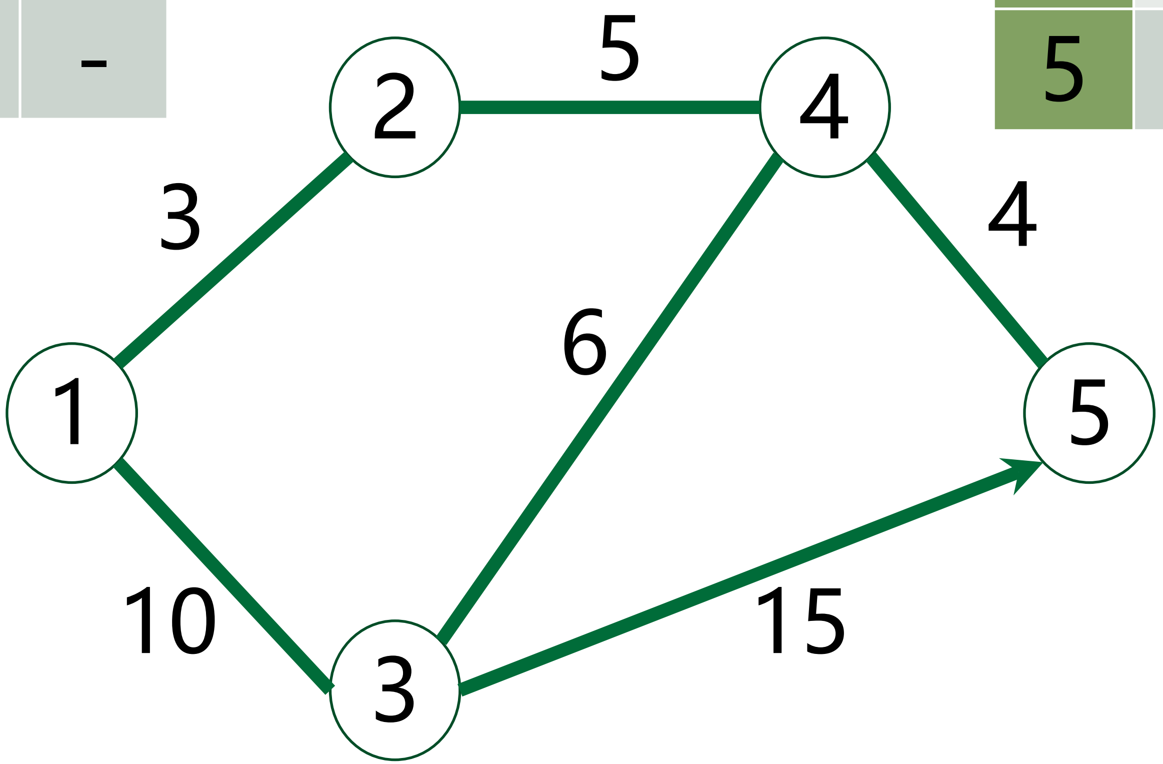


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，**i,j**分别**=2**，**3**，**4**，**5...**
对应的**i**列（D_{ik}）**+** 对应的**j**行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*3*	4	*5*
3	1	*2*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=1,**引入顶点**1**

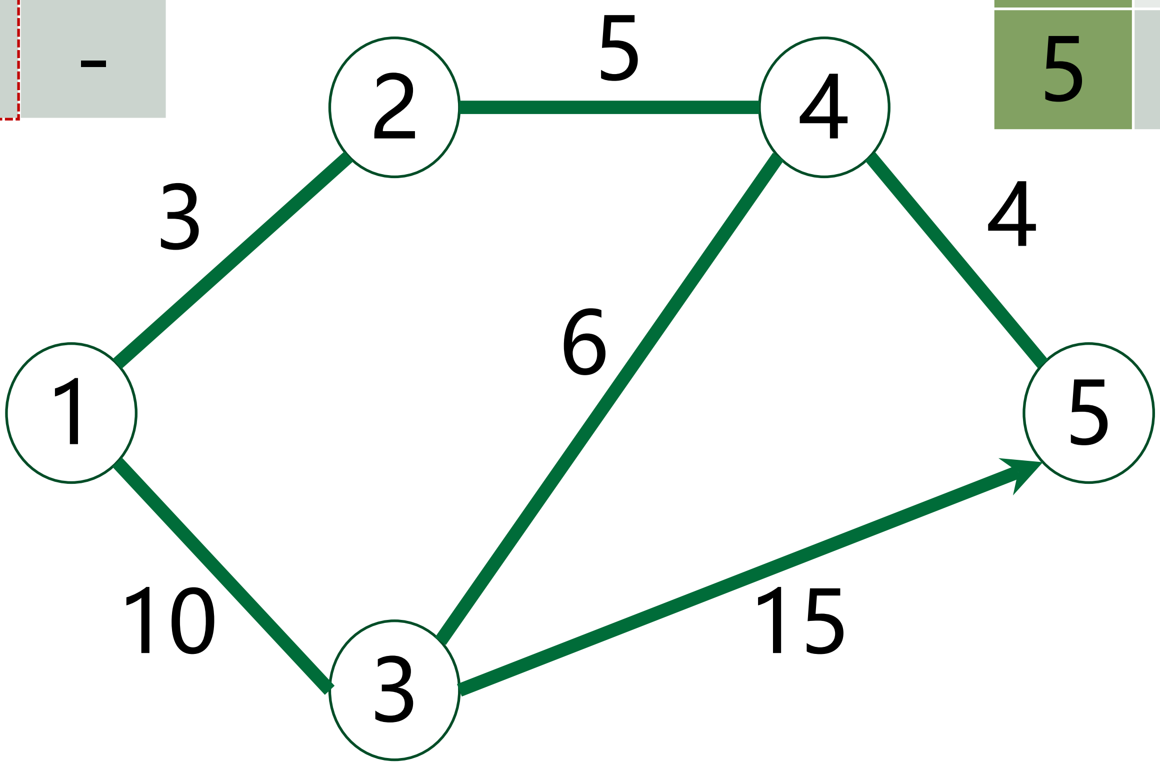


D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

分别考察
k=1的时候，i,j分别=2，3，4，5...
对应的i列（D_{ik}）+ 对应的j行（D_{kj}）
是否小于 D_{ij}

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	∞	5	∞
3	10	∞	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

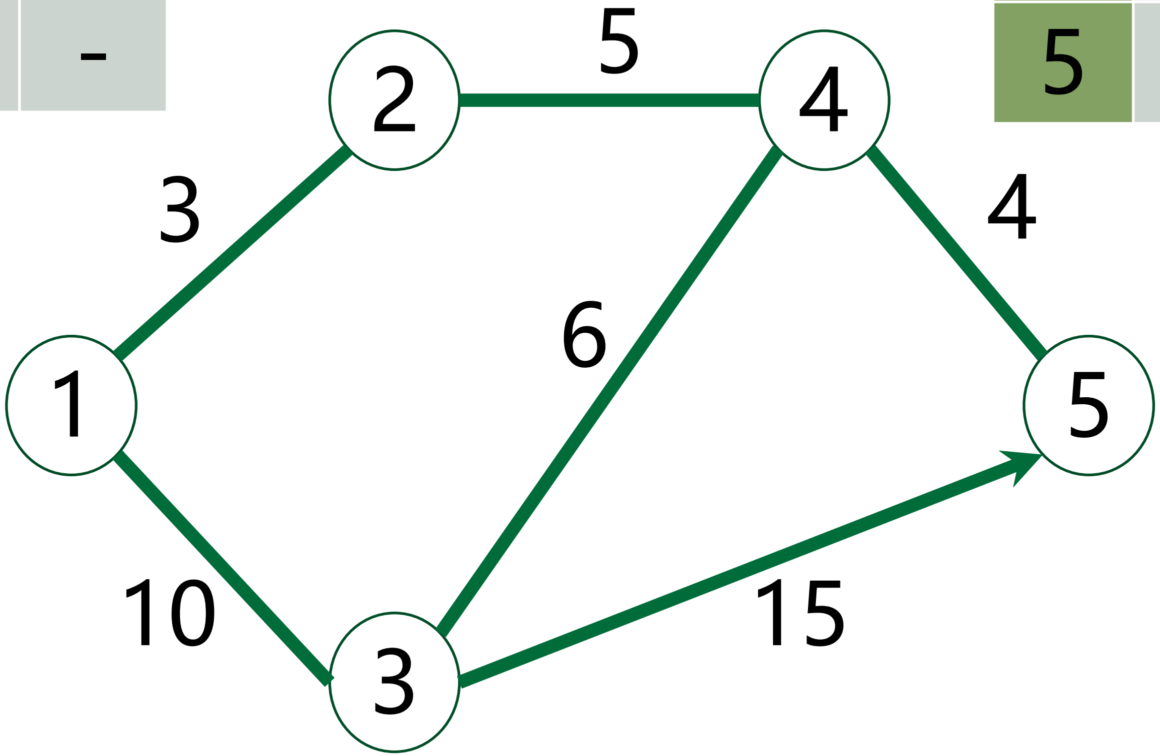
S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	3	4	5
3	1	2	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

令**k=1,**引入顶点**1**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现**i=2**，**j=3**



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	13	5	∞
3	10	13	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

S	1	2	3	4	5
1	-	2	3	4	5
2	1	-	1	4	5
3	1	1	-	4	5
4	1	2	3	-	5
5	1	2	3	4	-

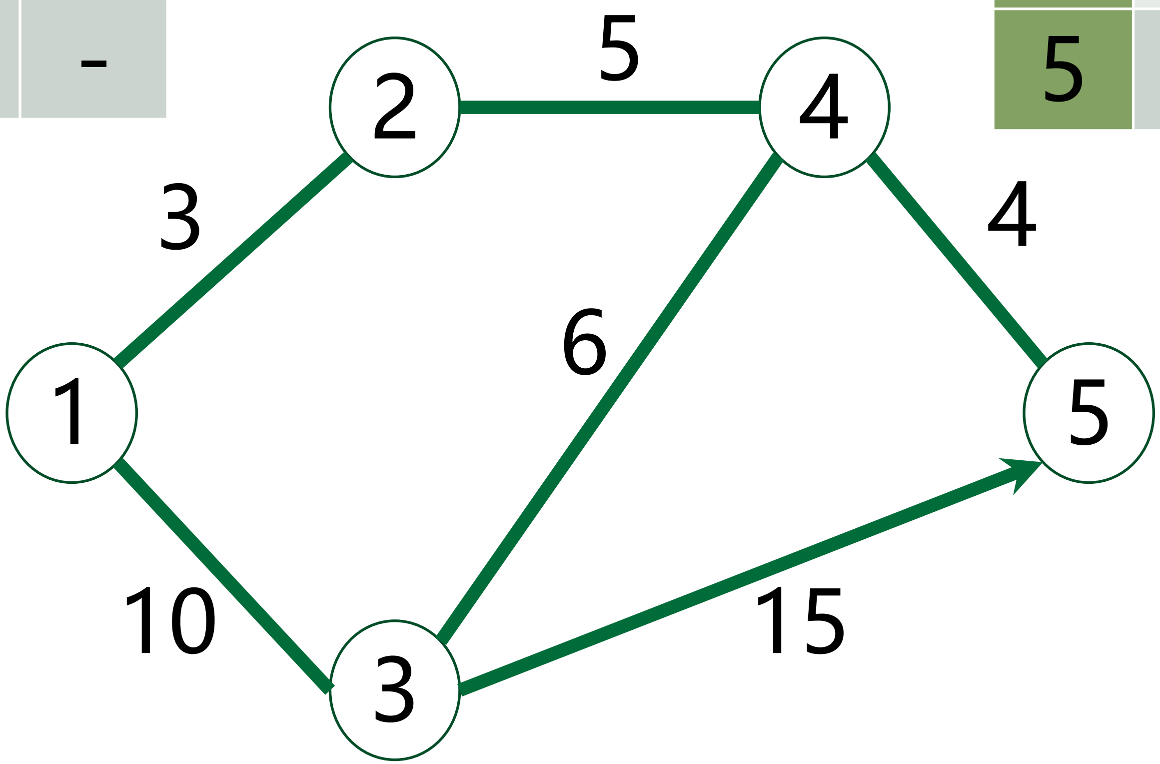
令**k=1**,引入顶点**1**

考察是否存在:

$D_{ik}+D_{kj}<D_{ij}$?

发现**i=2**，**j=3**

和**i=3**，**j=2**



D	1	2	3	4	5
1	-	3	10	∞	∞
2	3	-	*13*	5	∞
3	10	*13*	-	6	15
4	∞	5	6	-	4
5	∞	∞	∞	4	-

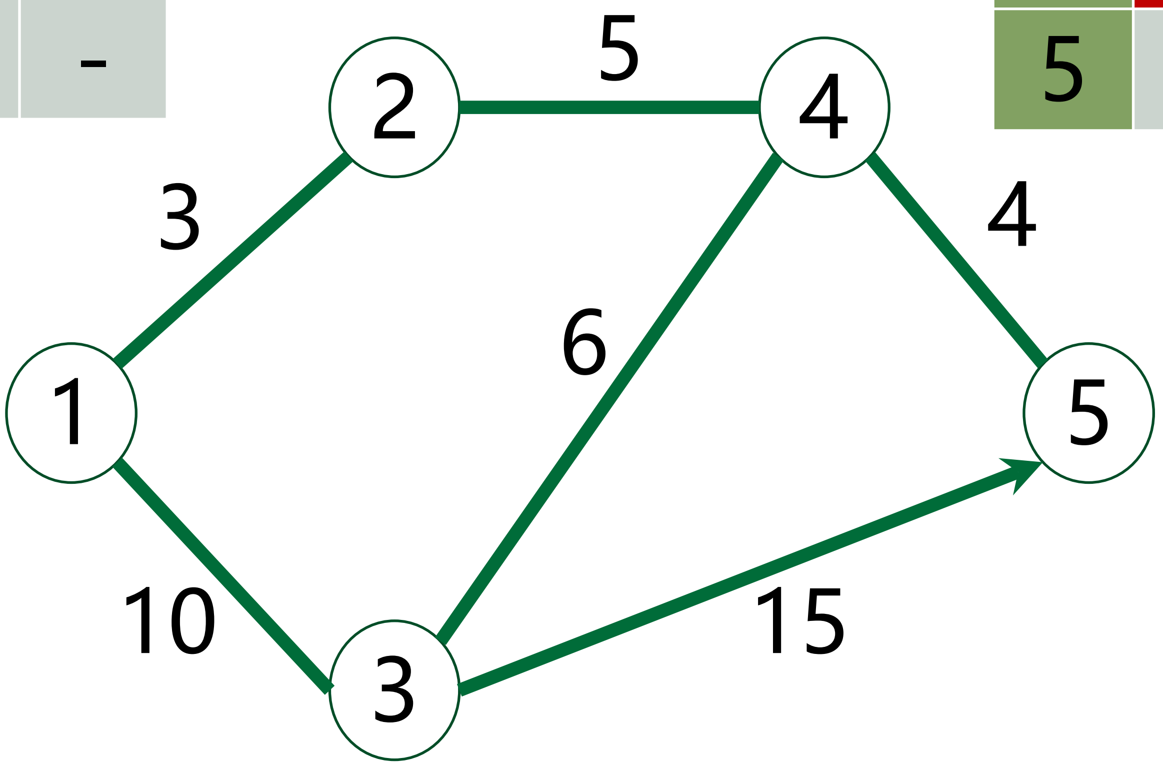
S	1	2	3	4	5
1	-	2	3	*4*	*5*
2	1	-	*1*	4	*5*
3	1	*1*	-	4	5
4	*1*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=2,**引入顶点**2**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现**i=1**，**j=4**



D	1	2	3	4	5
1	-	3	10	8	∞
2	3	-	13	5	∞
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

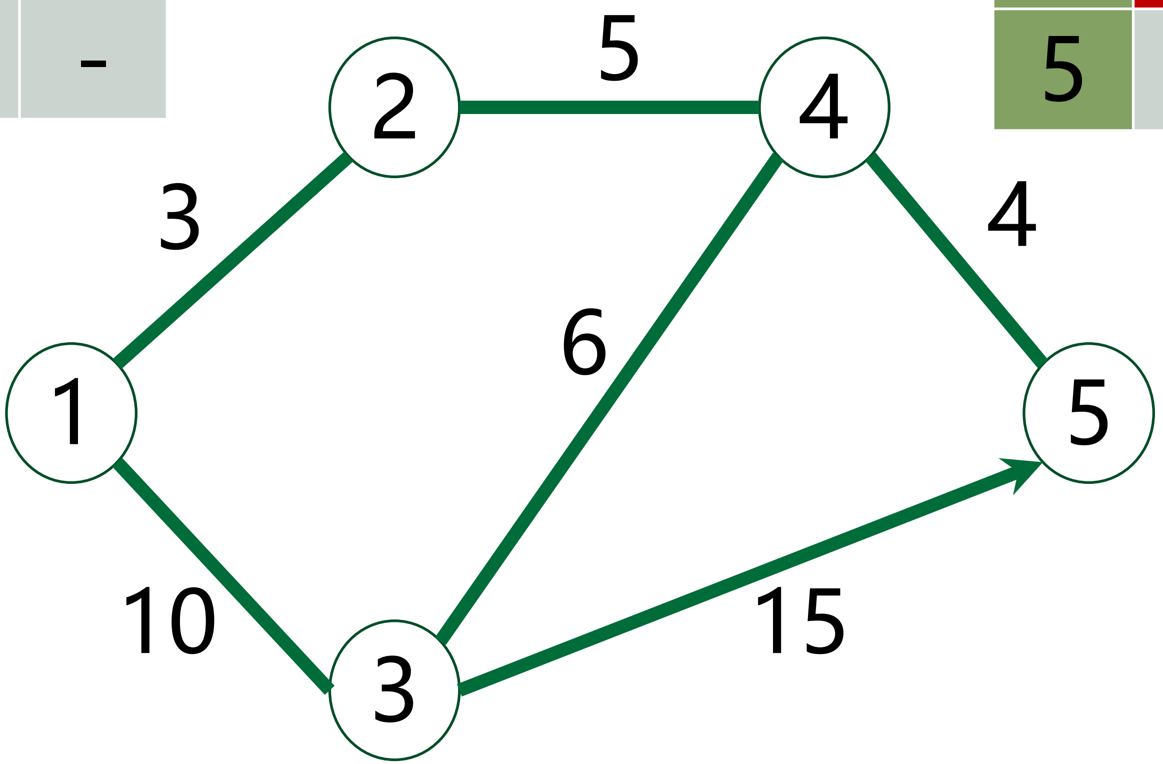
S	1	2	3	4	5
1	-	2	3	2	5
2	1	-	1	4	5
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令**k=2,**引入顶点**2**

考察是否存在：

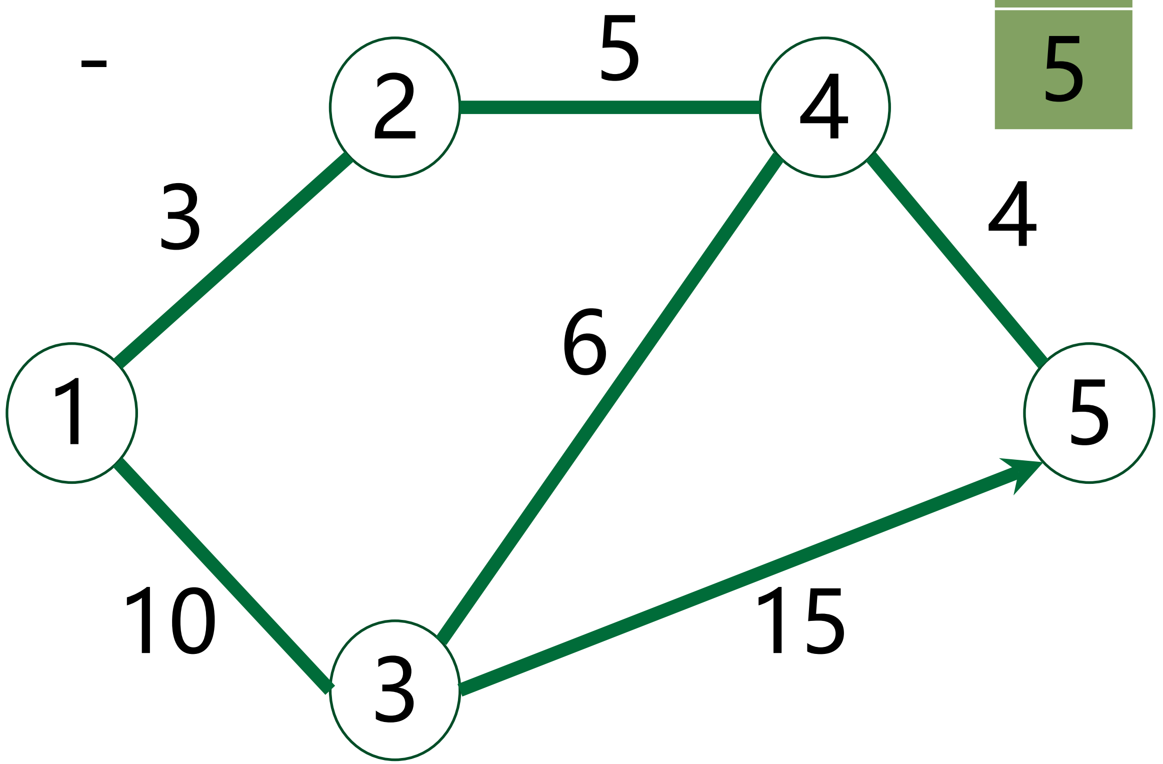
$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现**i=1**，**j=4**



D	1	2	3	4	5
1	-	3	10	*8*	∞
2	3	-	*13*	5	∞
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

S	1	2	3	4	5
1	-	2	3	*2*	*5*
2	1	-	*1*	4	*5*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-



令k=2,引入顶点**2**

考察是否存在：

$D_{ik}+D_{kj}<D_{ij}$?

发现**i=1，j=4**

D	1	2	3	4	5
1	-	3	10	8	∞
2	3	-	13	5	∞
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

S	1	2	3	4	5
1	-	2	3	2	5
2	1	-	1	4	5
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令k=3,引入顶点**3**

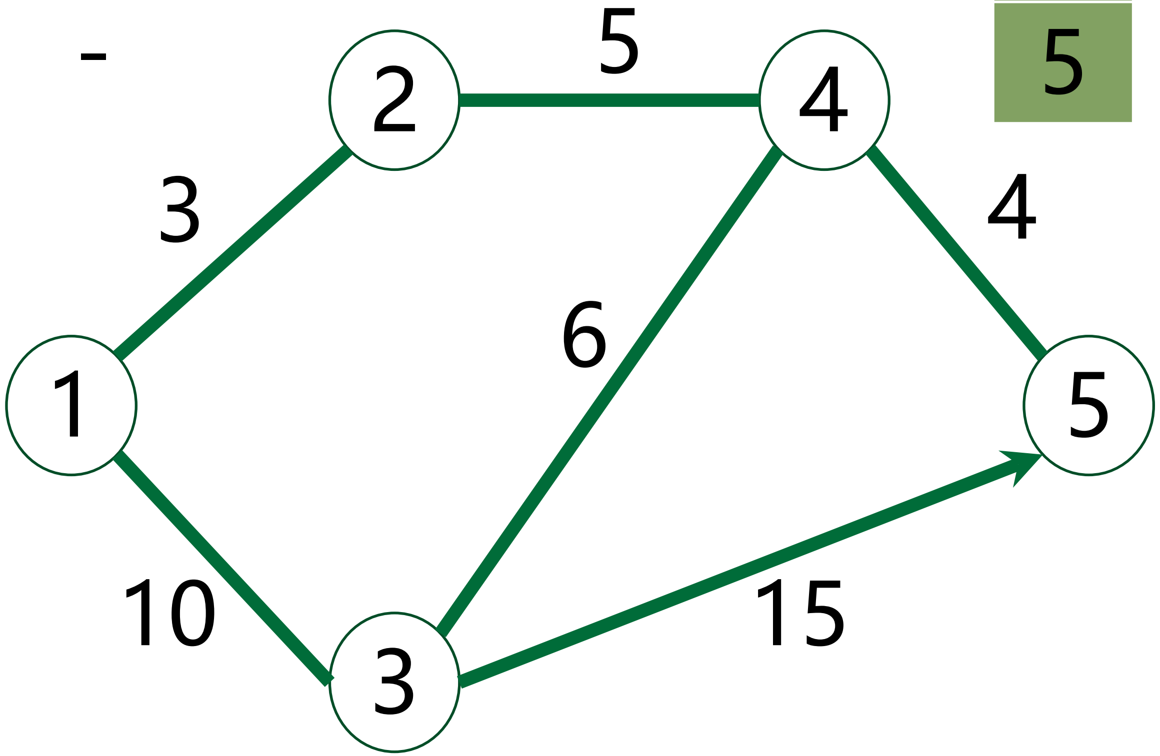
考察是否存在：

$D_{ik}+D_{kj}<D_{ij}$?

发现：

i=1，j=5

i=2，j=5



D	1	2	3	4	5
1	-	3	10	*8*	*25*
2	3	-	*13*	5	*28*
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	*∞*	*∞*	*∞*	4	-

S	1	2	3	4	5
1	-	2	3	*2*	3
2	1	-	*1*	4	*3*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=3,**引入顶点**3**

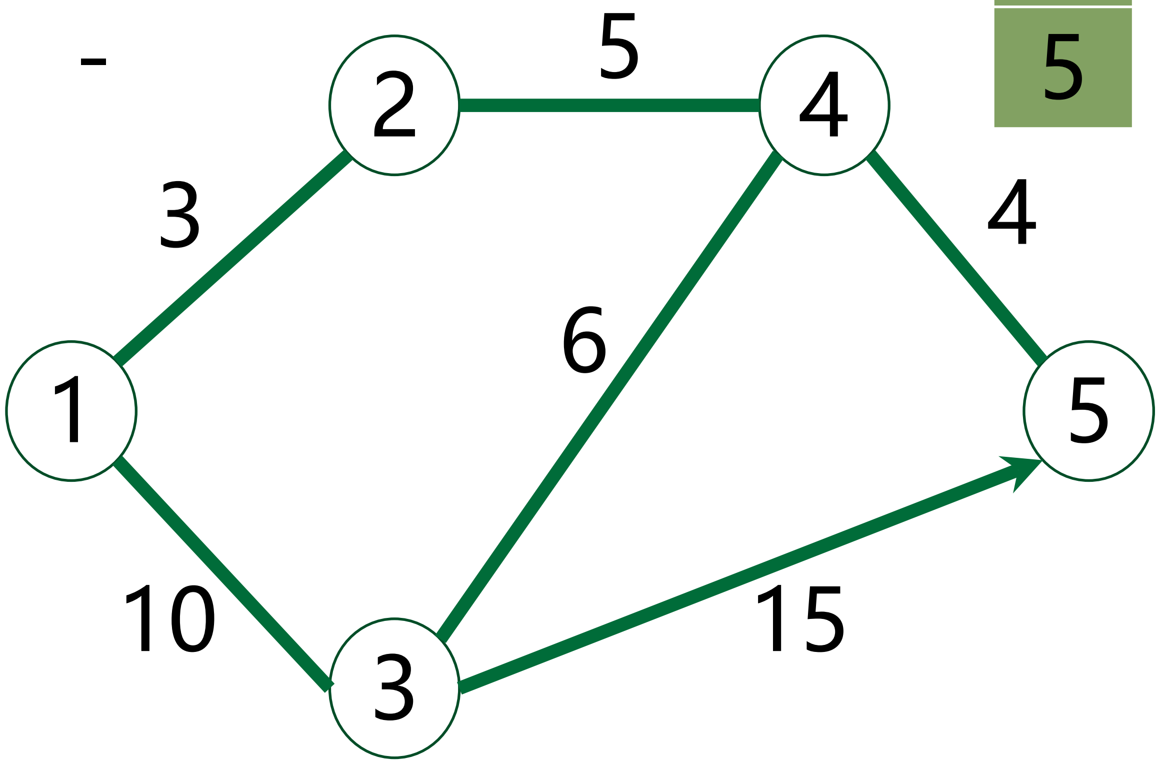
考察是否存在：

$D_{ik}+D_{kj}<D_{ij}$?

发现：

i=1，**j=5**

i=2，**j=5**



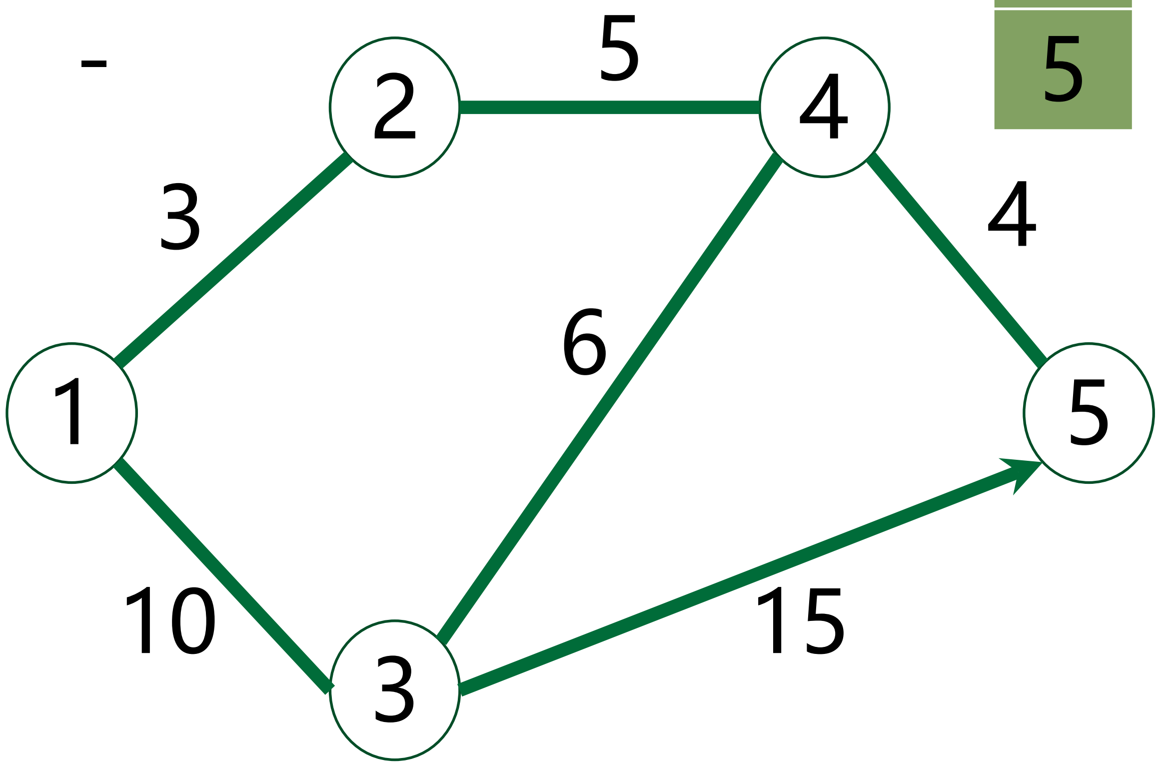
D	1	2	3	4	5
1	-	3	10	8	25
2	3	-	13	5	28
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

S	1	2	3	4	5
1	-	2	3	2	3
2	1	-	1	4	3
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$D_{ik}+D_{kj}<D_{ij}$?



D	1	2	3	4	5
1	-	3	10	8	25
2	3	-	13	5	28
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

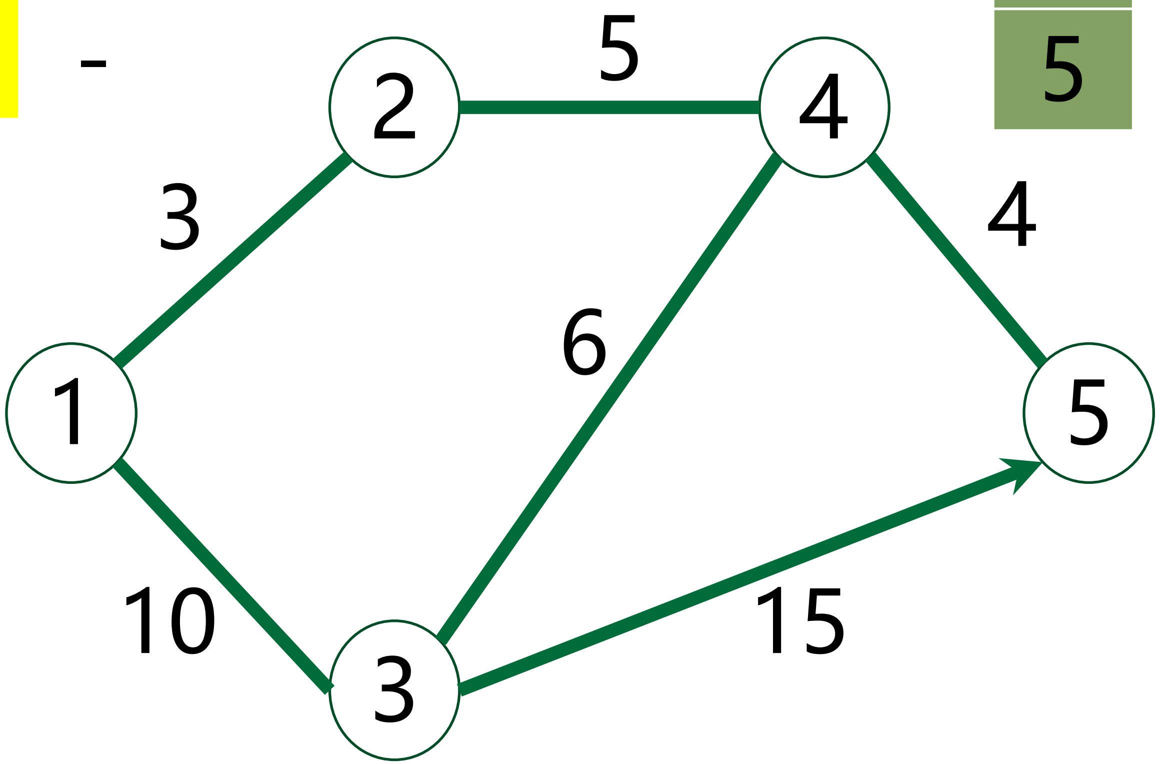
S	1	2	3	4	5
1	-	2	3	2	3
2	1	-	1	4	3
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$?

发现：**i=1**，**j=5**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*13*	5	*28*
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

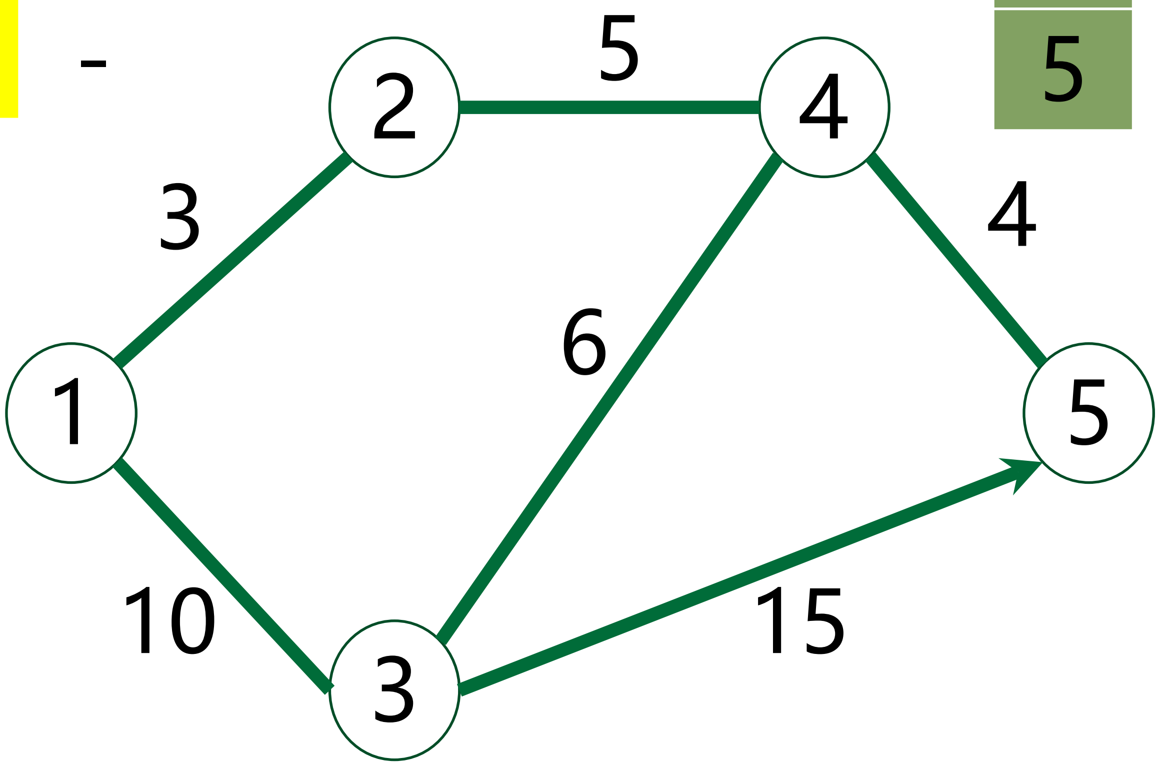
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*1*	4	*3*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=1**，**j=5**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*13*	5	*28*
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

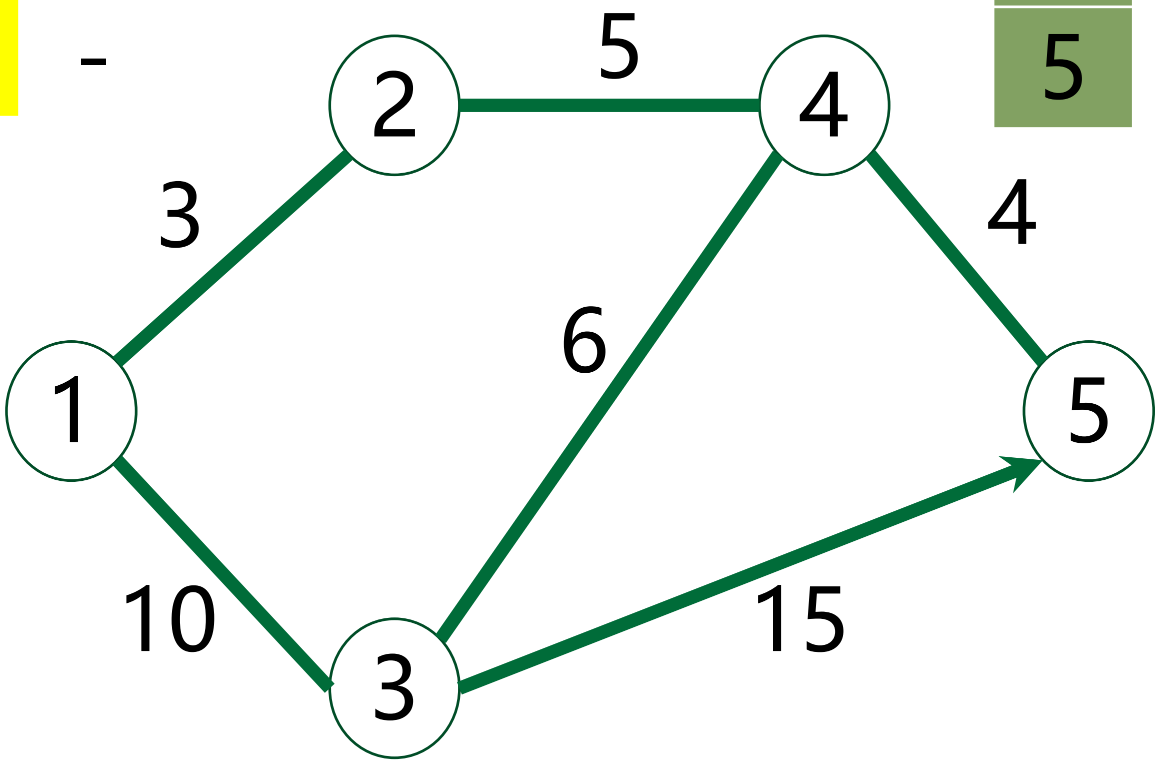
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*1*	4	*3*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现：**i=2**，**j=3**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*28*
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

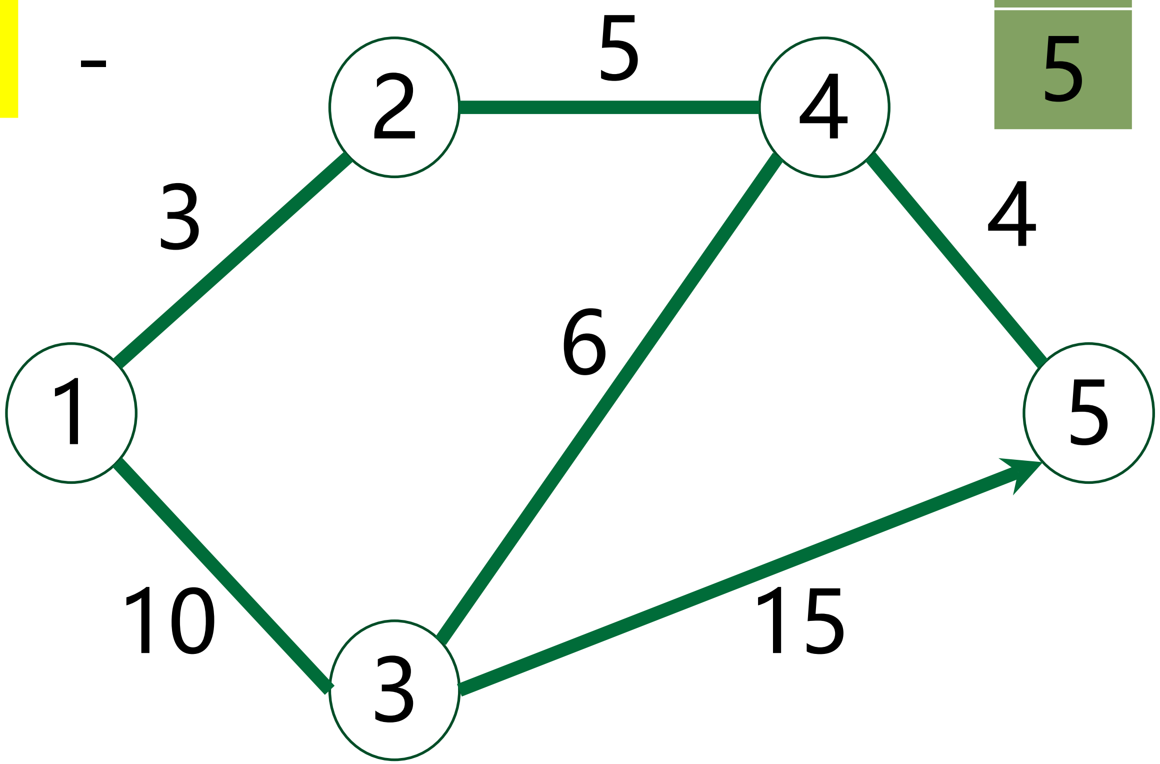
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*3*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=2**，**j=5**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	13	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

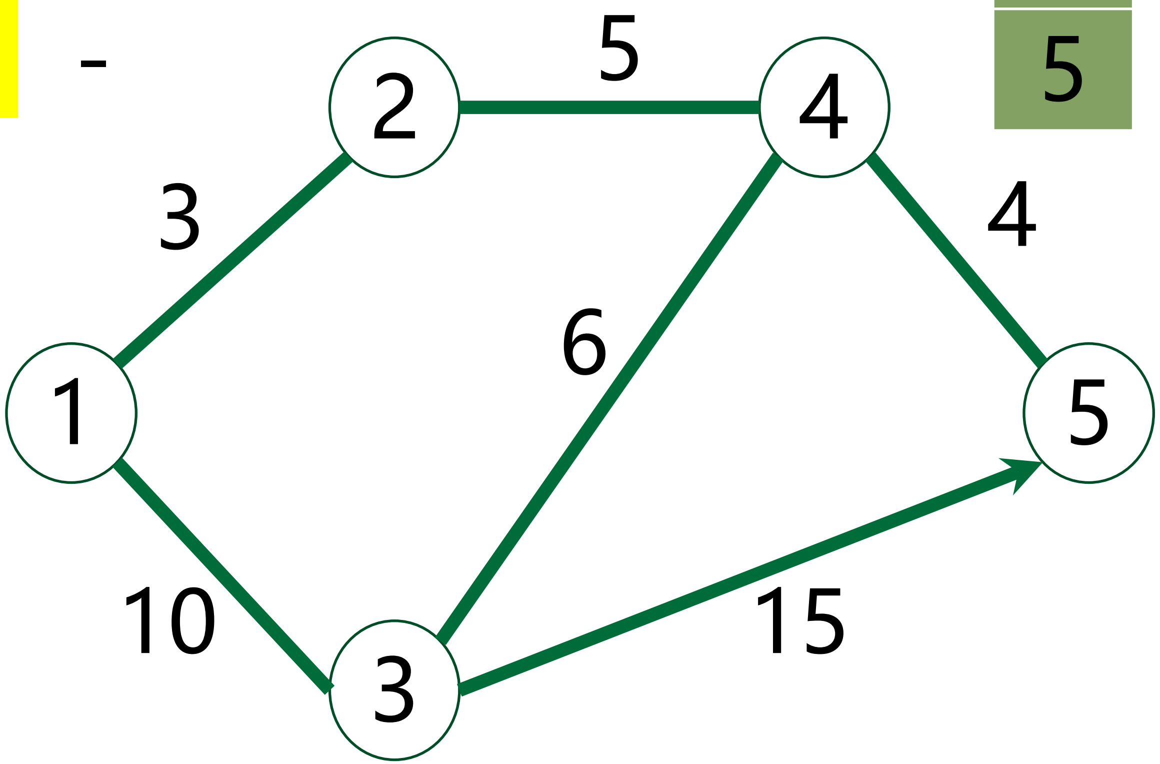
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	1	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=2**，**j=5**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*9*
3	10	*13*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

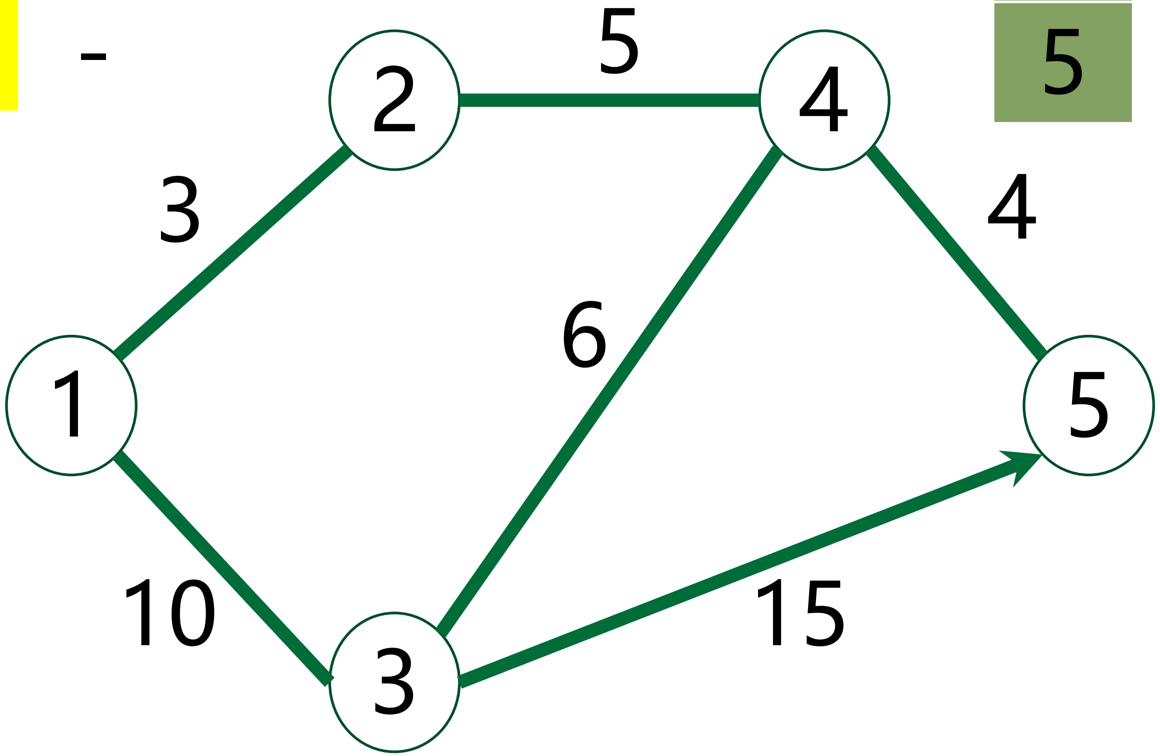
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*4*
3	1	*1*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=3**，**j=2**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*9*
3	10	*11*	-	6	15
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

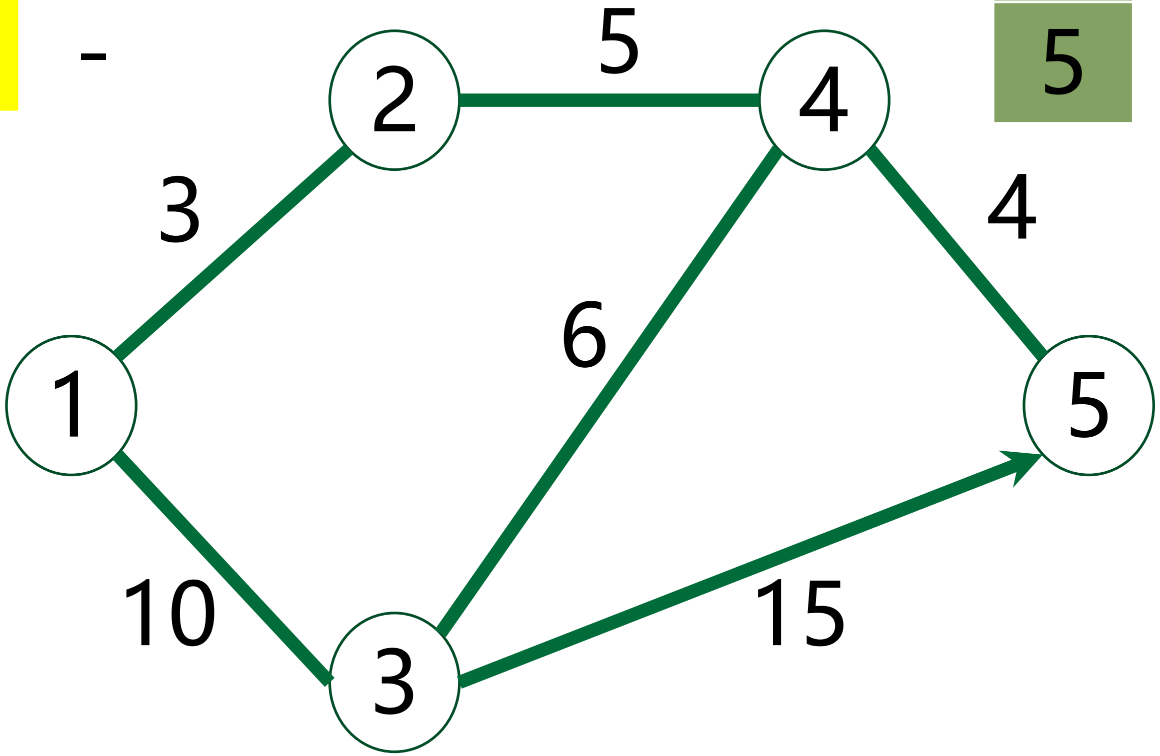
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*4*
3	1	*4*	-	4	5
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现：**i=3，j=2**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	15
4	8	5	6	-	4
5	∞	∞	∞	4	-

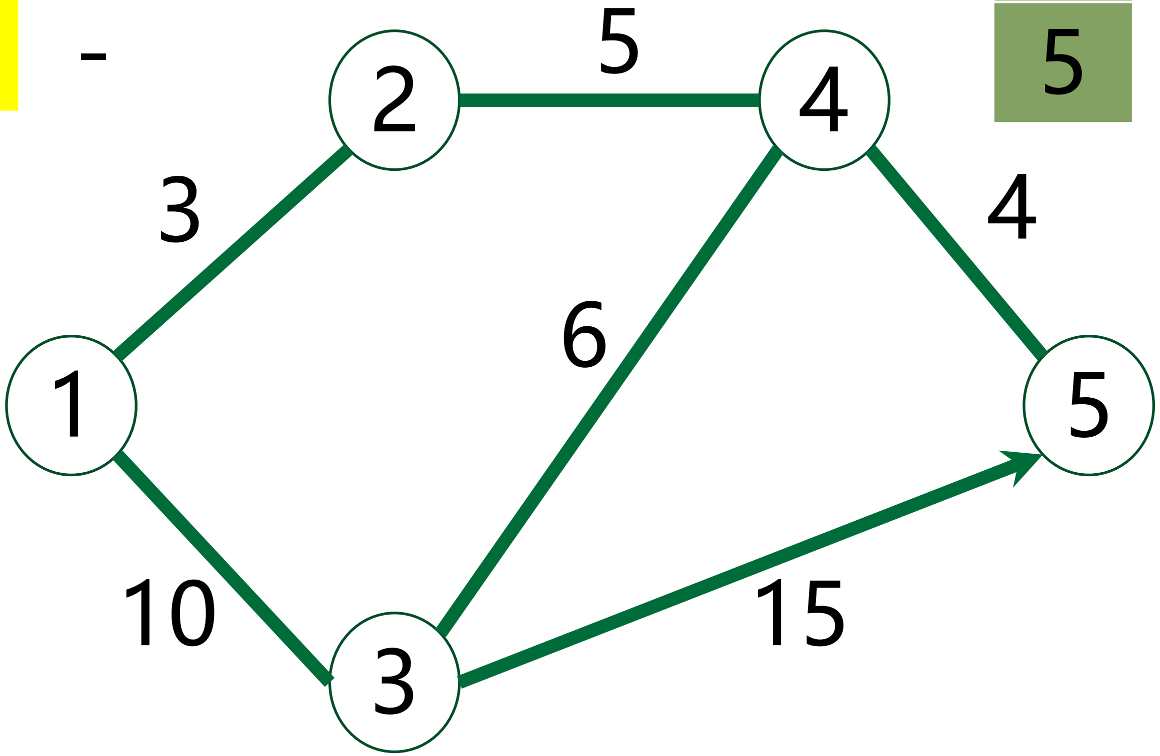
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	5
4	2	2	3	-	5
5	1	2	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现：**i=3，j=5**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*9*
3	10	*11*	-	6	10
4	*8*	5	6	-	4
5	∞	∞	∞	4	-

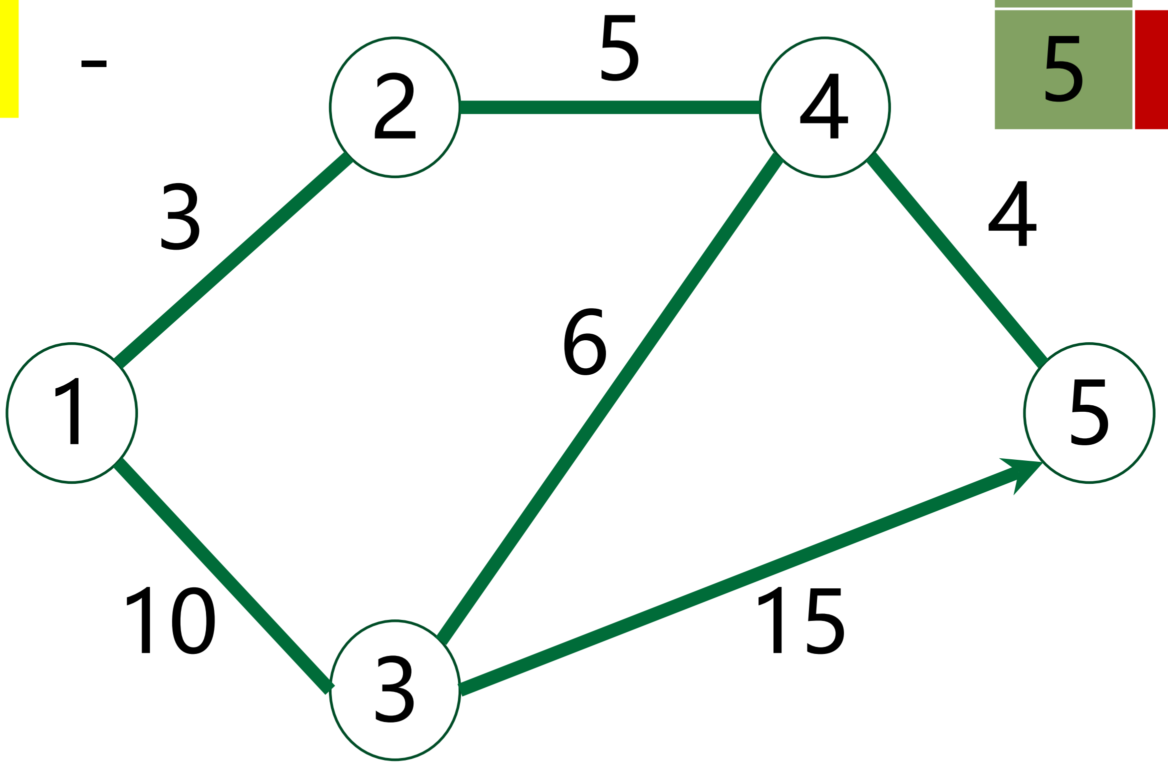
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*4*
3	1	*4*	-	4	4
4	*2*	2	3	-	5
5	*1*	*2*	*3*	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$?

发现：**i=5**，**j=1**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	∞	∞	4	-

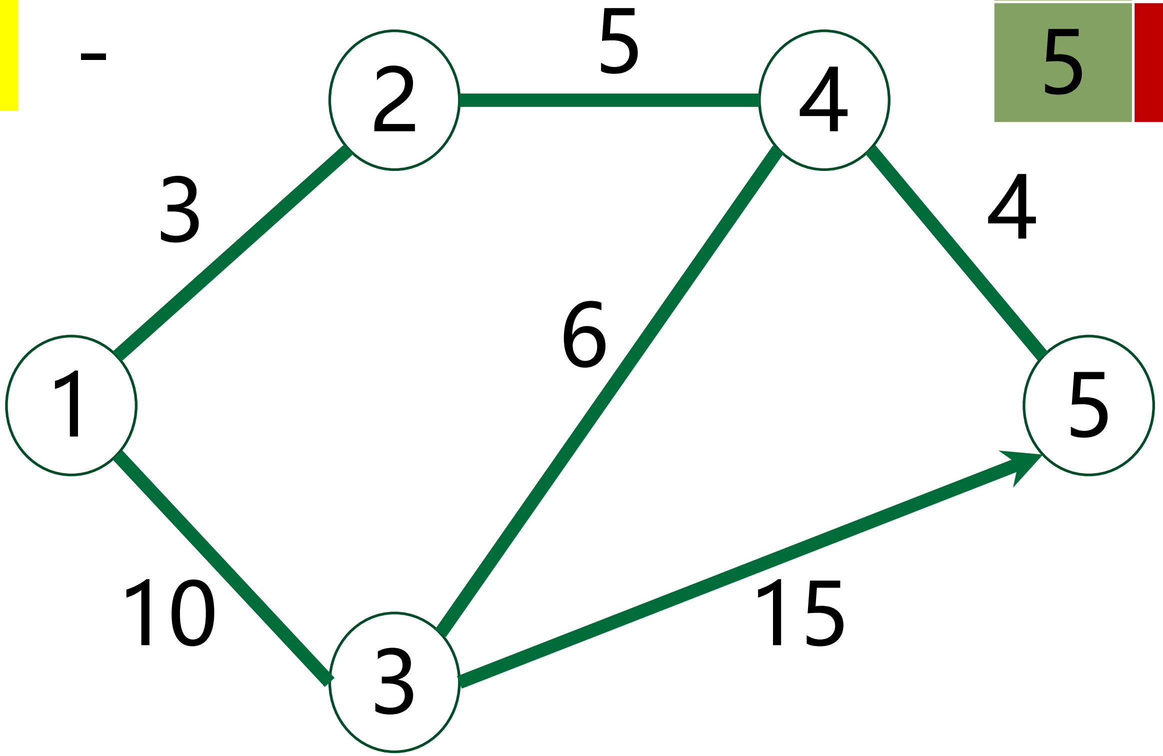
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	2	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=5**，**j=2**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	9	∞	4	-

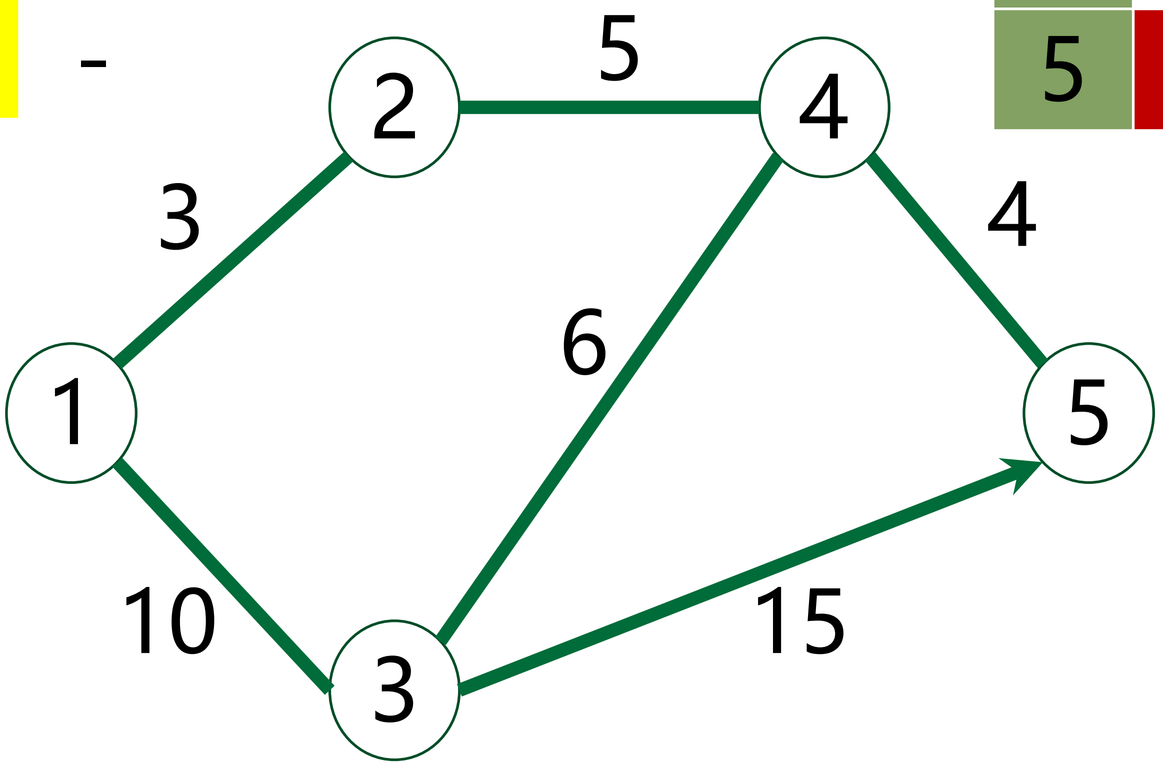
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	4	3	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$**?**

发现：**i=5**，**j=3**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	9	10	4	-

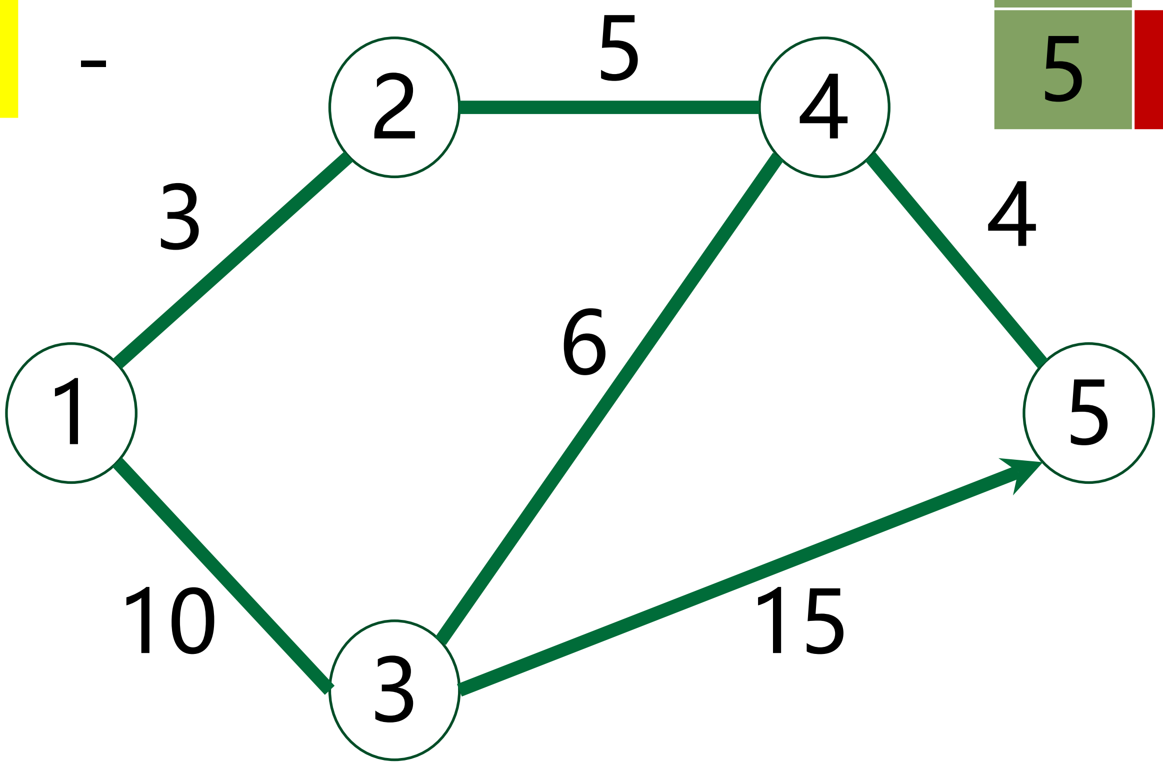
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	4	4	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现：**i=5**，**j=3**



D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	9	10	4	-

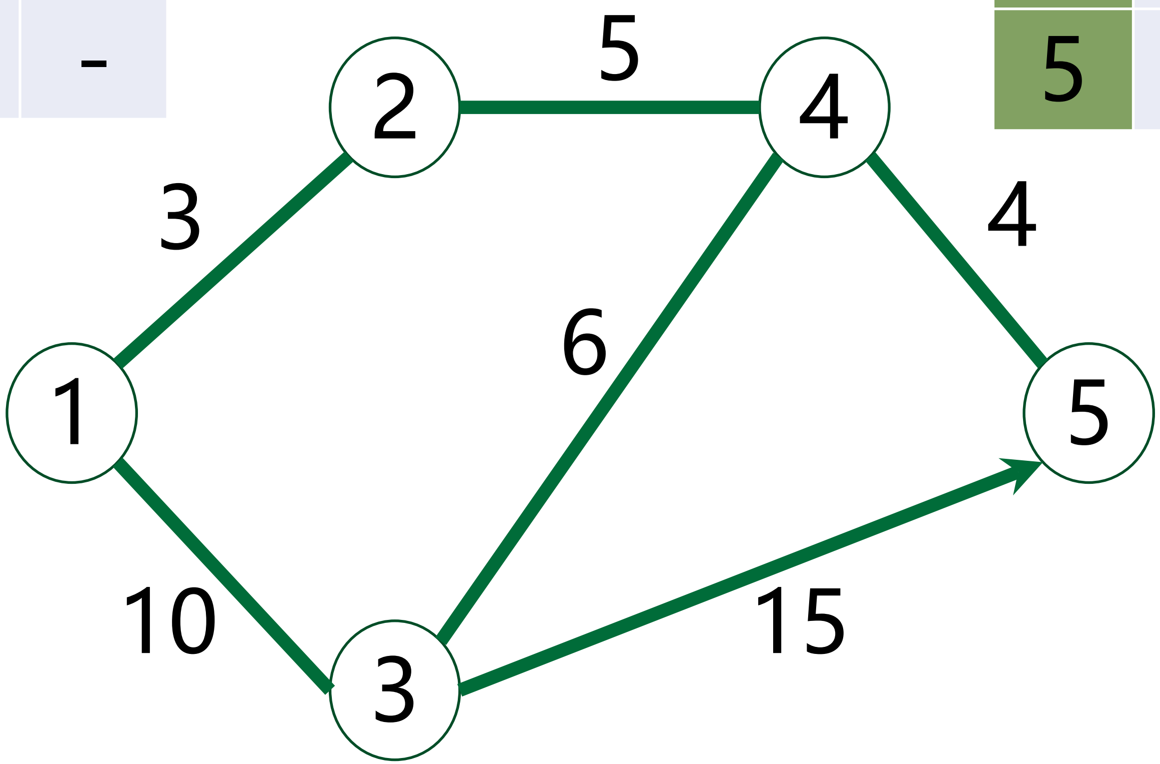
S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	4	4	4	-

令**k=4,**引入顶点**4**

考察是否存在：

$D_{ik}+D_{kj}<D_{ij}$?

发现：**i=5**，**j=3**



D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*9*
3	10	*11*	-	6	10
4	*8*	5	6	-	4
5	*12*	*9*	*10*	4	-

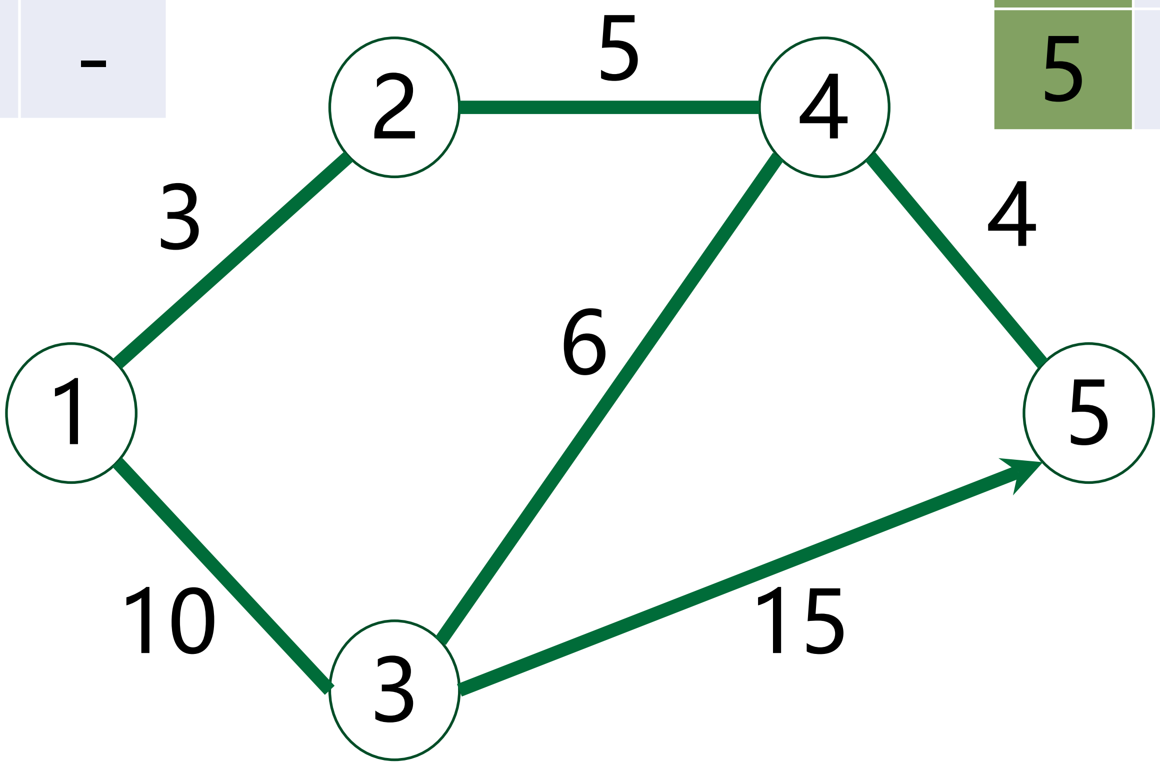
S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*4*
3	1	*4*	-	4	4
4	*2*	2	3	-	5
5	*4*	*4*	*4*	4	-

令**k=5,**引入顶点**5**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$

发现：**i=5**，**j=3**



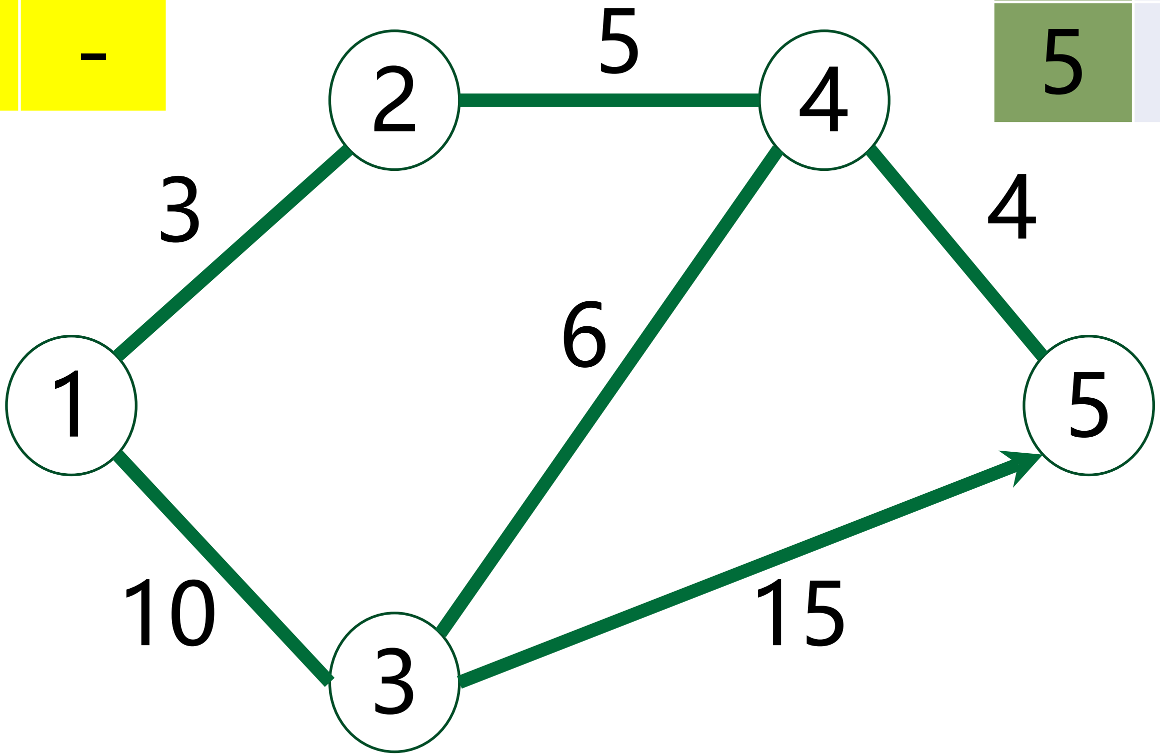
D	1	2	3	4	5
1	-	3	10	8	12
2	3	-	11	5	9
3	10	11	-	6	10
4	8	5	6	-	4
5	12	9	10	4	-

S	1	2	3	4	5
1	-	2	3	2	4
2	1	-	4	4	4
3	1	4	-	4	4
4	2	2	3	-	5
5	4	4	4	4	-

令**k=5,**引入顶点**5**

考察是否存在：

$\mathbf{D_{ik}+D_{kj}<D_{ij}?}$



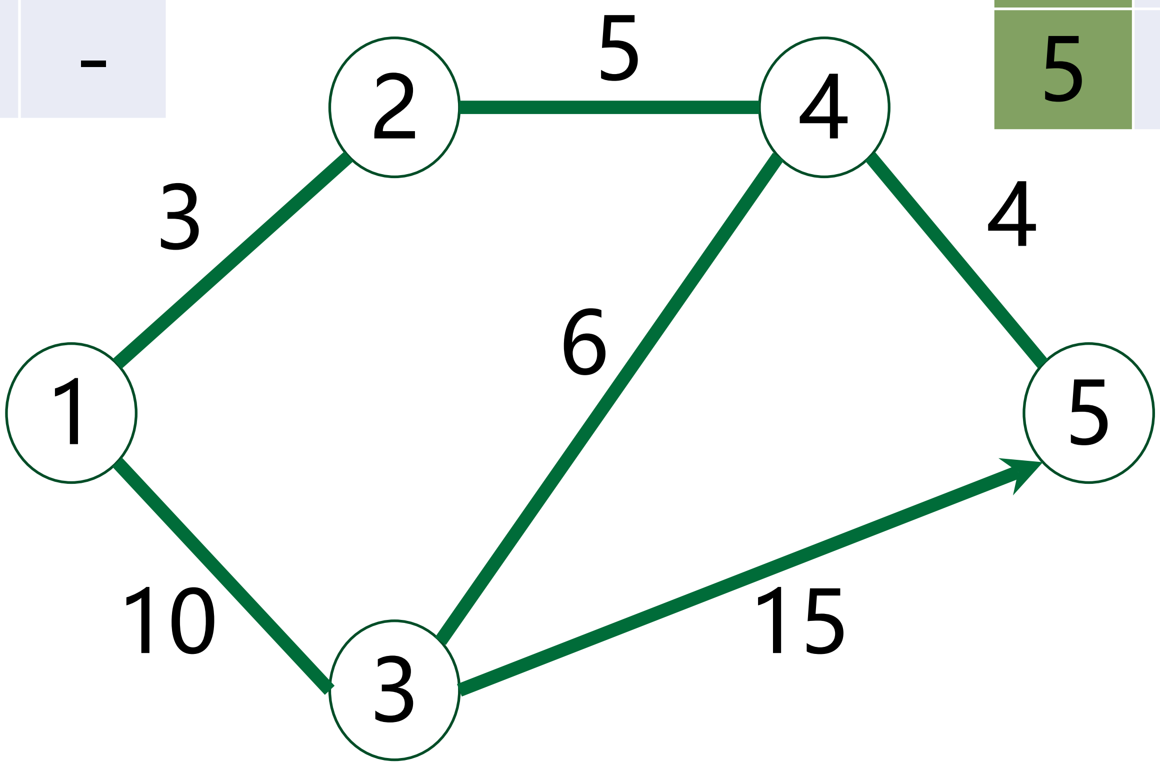
D	1	2	3	4	5
1	-	3	10	*8*	*12*
2	3	-	*11*	5	*9*
3	10	*11*	-	6	10
4	*8*	5	6	-	4
5	*12*	*9*	*10*	4	-

S	1	2	3	4	5
1	-	2	3	*2*	4
2	1	-	*4*	4	*4*
3	1	*4*	-	4	4
4	*2*	2	3	-	5
5	*4*	*4*	*4*	4	-

令**k=5,**引入顶点**5**

考察是否存在:

$\mathbf{D_{ik}+D_{kj}<D_{ij}}$?



6.6 最短路径

求每一对顶点之间的最短路径

1. 每次以一个顶点为源点，重复执行**Dijkstra**算法**n**次

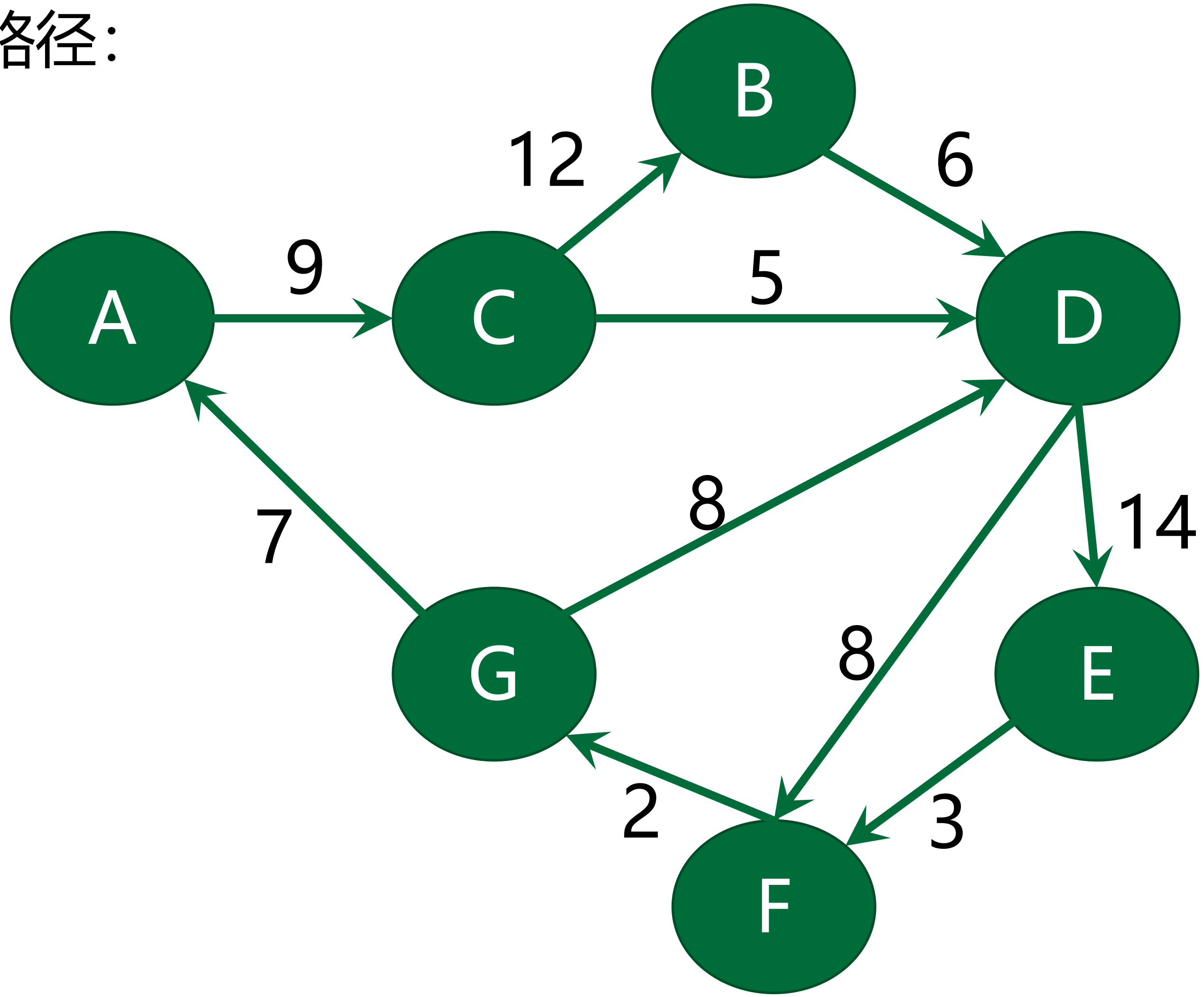
   $T(n)=O(n^3)$，不能有负权边

2. 弗洛伊德**(Floyd)**算法

   从v_i到v_j的所有可能存在的路径中，选出一条长度最短的路径。**$T(n)=O(n^3)$**，不可以有负权回路

6.6 最短路径

分别用（以A为出发点）：
Dijkstra算法和Floyd算法求取最短路径：



思考题

1、判断一个有向图是否有环（回路）的方法是

A）求结点的度　　B）拓扑排序

C）求关键路径　　D）求最短路径

答案：**B**

2、在有向图的邻接表存储结构中，顶点v在链表中出现的次数是

A）顶点的v的度　　B）顶点v的出度

C）顶点v的入度　　D）依附于顶点v的边数

答案：**C**

3、用邻接矩阵表示图时，若图中有100个顶点，100条边，则形成的矩阵有多少元素？有多少非零元素？

答案：邻接矩阵中的元素有$100^2 = 10000$个。

它有100个非零元素（对于有向图）或200个非零元素（对于无向图）。

要点提示

图

- 存储结构
  - 邻接矩阵
  - 邻 接 表
  - 十字链表
  - 邻接多重表
- 遍　历
  - 深度优先搜索
  - 广度优先搜索

无向图的应用

- 图的连通分量　（利用DFS）
- 图的关节点　　（利用DFS）
- 图的生成树 → 最小生成树
  - Prim算法
  - Kruskal算法

有向图的应用

- 拓扑排序和关键路径
- 最短路径
  - Dijkstra算法
  - Floyd算法