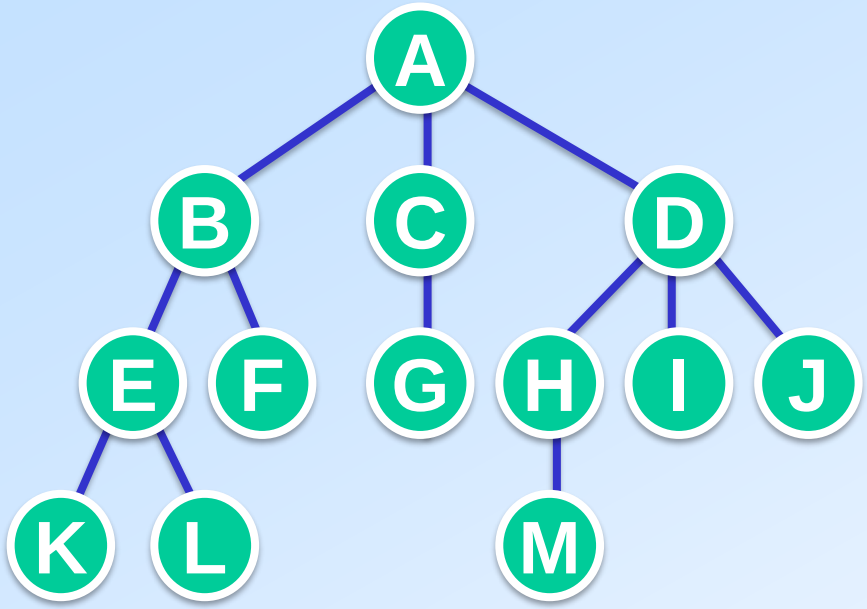
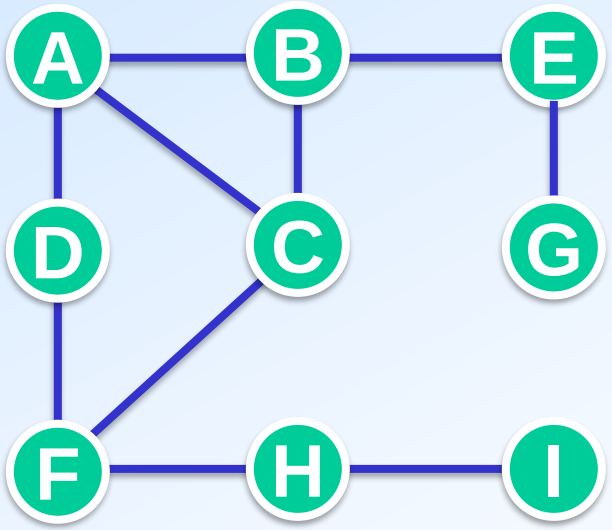
第七章 图

- 7.1 图的基本概念
- 7.2 图的存储结构
  - 邻接矩阵
  - 邻接表
- 7.3 图的遍历
  - 深度优先遍历DFS
  - 广度优先遍历BFS
- 7.4 图的连通性
- 7.5 最小生成树
  - 普里姆算法Prim
- 7.6 活动网络
  - 拓扑排序
  - 关键路径
- 7.7 最短路径
  - 迪杰斯特拉算法 Dijkstra

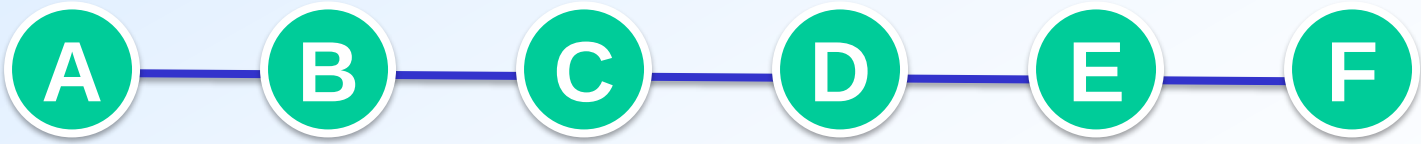
第七章 图—回顾与对比



树



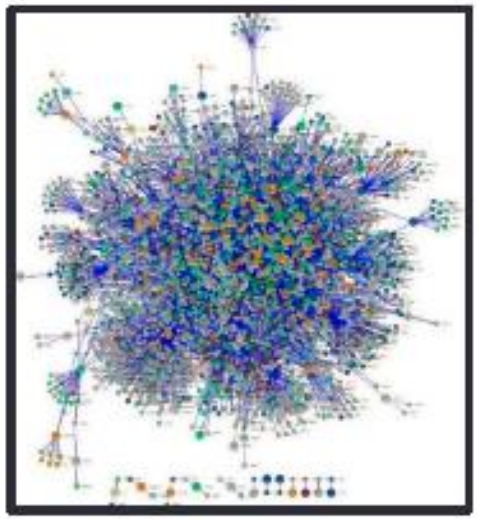
图



线性表

许多真实场景下，数据是有链接的

Social Networks



Biology Networks

Visual Web

A general language for describing and modeling complex systems

马克思主义哲学

唯物辩证法

- 联系具有普遍性：任何事物都处于普遍联系之中。
- 联系具有客观性：联系是事物本身所固有的，不以人的意志为转移。

李国杰院士

《大数据研究的科学价值》

- 数据背后的共性-关系网络。大数据面临的科学问题本质上可能就是网络科学问题。复杂网络分析应该是数据科学的重要基石。

图数据结构普遍存在

SPECIAL SECTION

INTRODUCTION

Connections

Complex Systems and Networks

We are caught in an inescapable network of mutuality. ... Whatever affects one directly, affects all indirectly. —Martin Luther King Jr.

FROM CHAOS COMES COMPLEXITY. FROM THE MOVEMENT OF MOLECULES WITHIN our cells to communication across an entire planet, we are part of networks. This special section shows how scientists are pushing network analysis to its limits across disciplinary fields.

Barabási published his seminal paper on scale-free networks a decade ago, and he starts us off by looking both at the past and the future (p. 412). The dramatic progress of researchers from disparate fields plunging into network analysis needs to be tempered by awareness of the potential dangers of misapplying fundamental assumptions (Butts, p. 414). Network analyses are also providing insights that will help us deal with our largest societal challenges. Bascompte (p. 416) confronts the effects of climate change on ecosystems and Ostrom (p. 419) examines organizing to maintain sustainability.

Physicists have taken up the challenge, too, as Adrian Cho reports (p. 406), aiming to use quantitative methods to forecast ethnic strife in Somalia, monitor surges of emotion in Internet users, and track the emergence of behavioral norms. Additionally, John Bohannon (p. 409) looks into one of the most controversial uses of network analysis: to identify key figures in terrorist organizations and eliminate them. Furthermore, networks can teach us about underlying mechanisms that affect us individually and as a society, such as monetary exchanges (Schweitzer *et al.*, p. 422) or transportation systems that promote viral transmission (Vespignani, p. 425).

At a microscopic level, molecular biologists are using networks to analyze basic cellular circuitry (Kim *et al.*, p. 429) to describe how interactions within a cell can be measured or modeled to generate predictions of responses to perturbation. Additionally, *Science Signaling* focuses on dynamics in signaling networks.

We need more and better data in many disciplines. It is not enough to look at patterns; we need to study how they evolve and change. The magnitude of the challenges we are facing shows how much we still need to learn. How can we move between levels of a complex system—to understand the transition from DNA sequence to disease symptoms or to predict the next economic recession? Network analysis is allowing us to understand how the world works from new vantage points, and it is exciting to think about what we will learn in the next 10 years.

– BARBARA R. JASNY, LAURA M. ZAHN, ELIOT MARSHALL

CONTENTS

See also related Policy Forum on p. 396 and go to http://www.sciencemag.org/complexity/ for additional online material from Science Signaling and http://sciencecareers.sciencemag.org/career_magazine/previous_issues/articles/2009_07_24/caredit.a0900091 for a look at Science Careers in Network Sciences.

CREDIT: CÉSAR HIDALGO

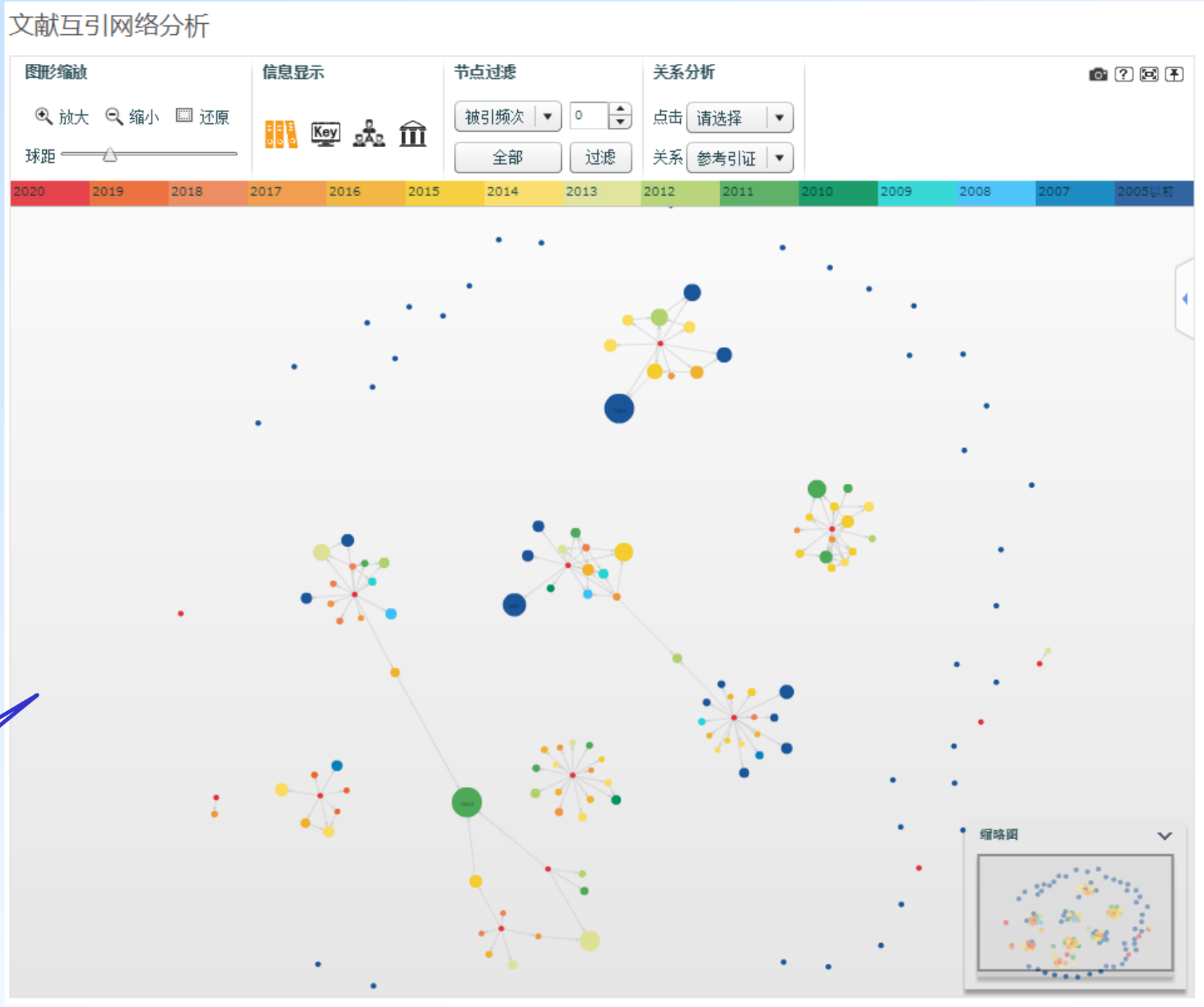
Science

我们被困于无法逃避的相互关联的网络中，任何事情，如果直接影响了一个人，就会间接影响到所有的人。

—马丁路德金

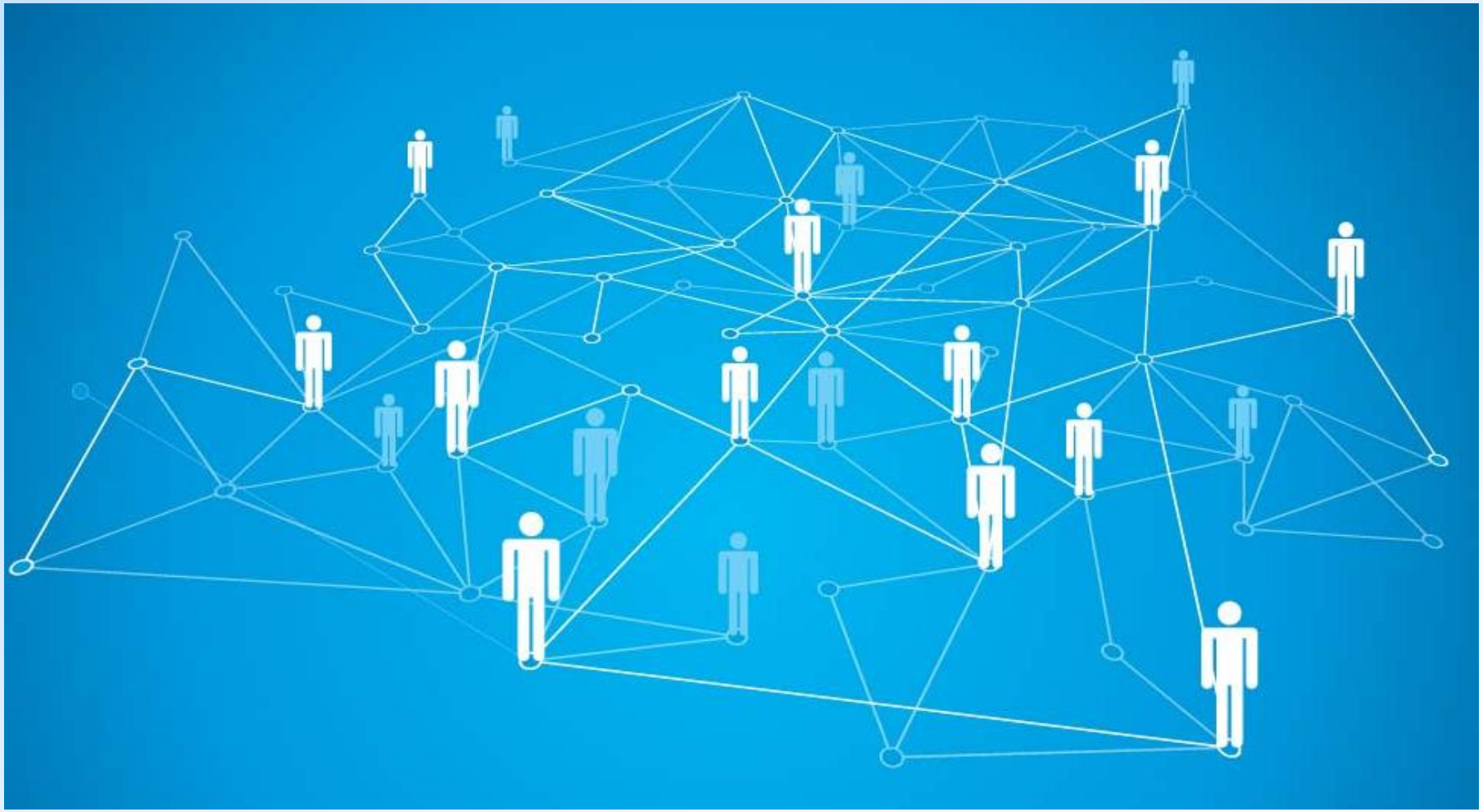
第七章 图—应用实例

- 2020年度《软件学报》在知网上所有论文引用分析
- 《软件学报》：中国计算机三大学报之一；
- 截至4月15日共收录57篇论文。



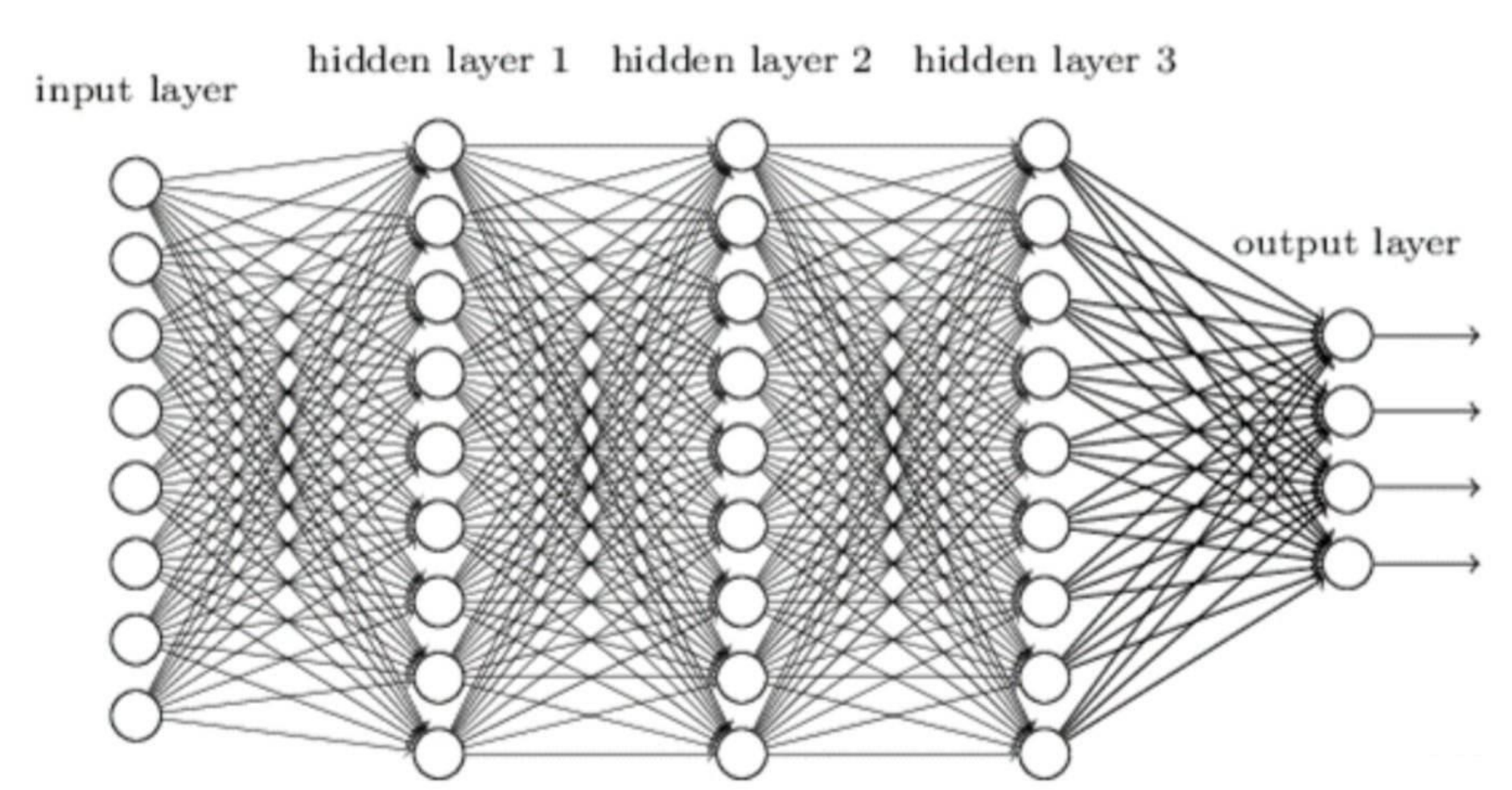
献互引网络分析图

第七章 图—应用实例



社交网络

第七章 图—应用实例



神经网络

第七章 图—应用实例

CNGI-CERNET2主干网

7.1 图的基本概念

- **图定义**　图是由顶点集合及顶点间的关系集合组成的一种数据结构：

$$G = (V, E)$$

$V = \{ v \mid v \in$ 某个数据对象$\}$ 是顶点的有穷非空集合；

$E1 = \{(v, w) \mid v, w \in V \}$

或　　$E2 = \{<v, w> \mid v, w \in V \ \&\& \ P(v, w)\}$

其中，E1 是顶点之间关系的有穷集合（边集合），此时图称为**无向图**；

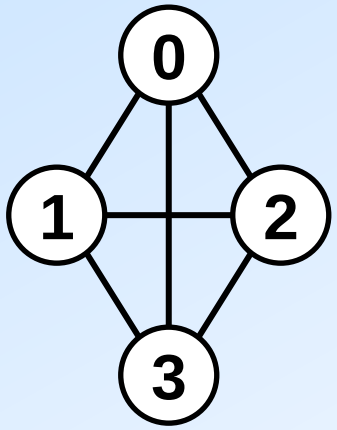
E2 中<v, w>表示从 v 到 w 的一条弧，且称 v 为弧尾， w 为弧头，这样的图称为**有向图**。谓词P(v,w)表示从v到w的一条单向通道。

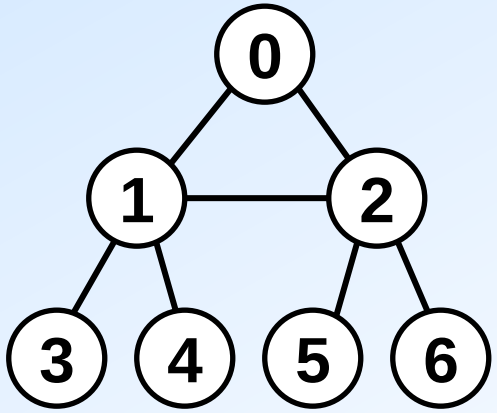
- **有向图与无向图**　在有向图中，顶点对 <v, w> 是**有序**的；在无向图中，顶点对 (v, w) 是**无序**的。

7.1 图的基本概念

完全图

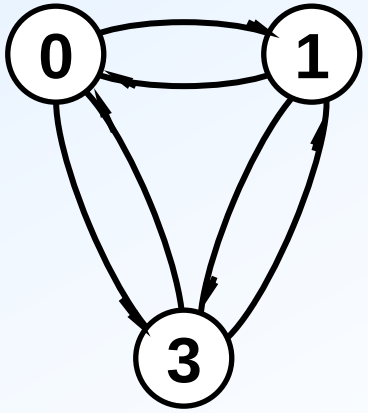
- 若有 n 个顶点的无向图有 n(n-1)/2 条边，则此图为无向完全图。有 n 个顶点的有向图有 n(n-1) 条边，则此图为有向完全图。

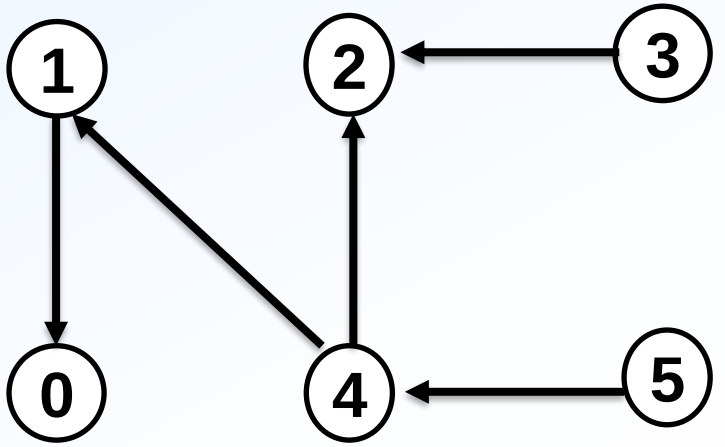




稀疏图与稠密图

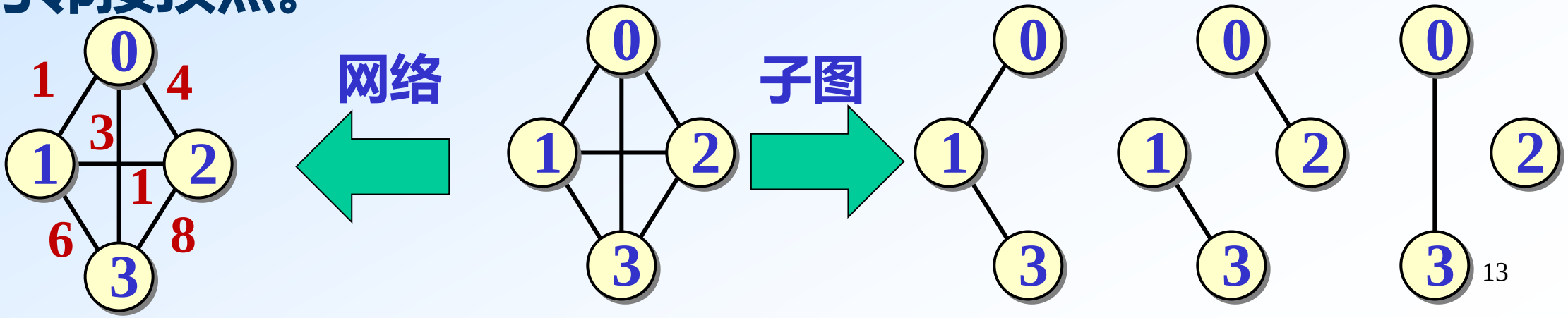
- 若有很少的边或弧的图称为稀疏图；
- 反之称为稠密图。





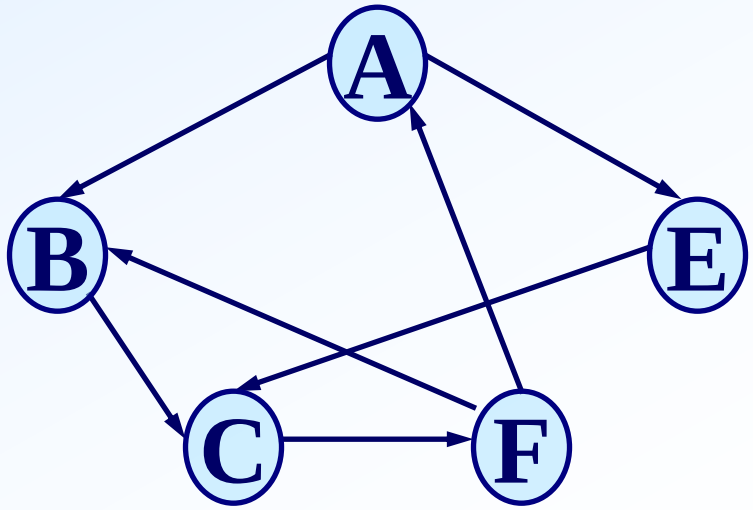
7.1 图的基本概念

- **网络与权** 某些图的边具有与它相关的数，称之为**权**。这种带权图叫做**网络**。
- **子图** 设有两个图 G = (V, E) 和 G' = (V', E')。若 V'⊆V 且 E'⊆E，则称图 G' 是图 G 的子图。
- **邻接顶点** 如果 (v, w) 是图 G 中的一条边，则称 v 与 w 互为邻接顶点。



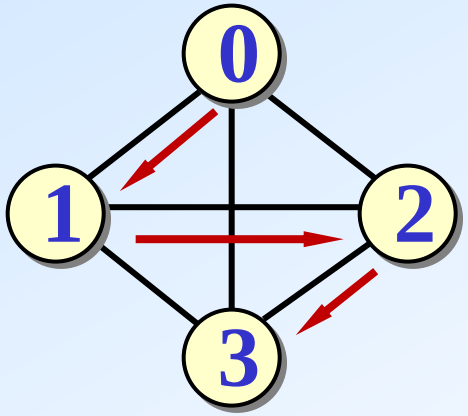
7.1 图的基本概念

- **顶点的度** 一个顶点 v 的度是与它相关联的边的条数。记作 TD(v)；
- **顶点 v 的入度**是以 v 为弧头(终点)的有向边的条数，记作 ID(v)；
- **顶点 v 的出度**是以 v 为弧尾(始点)的有向边的条数，记作 OD(v)。
- 对于有向图有：
  - TD = ID + OD
  - 如图：TD(B) = ID(B) + OD(B) = 2 + 1 = 3
- 有n个顶点，e条边或弧的图，满足：
  - $e = \frac{1}{2}\sum_{i=1}^{n} TD(v_i)$

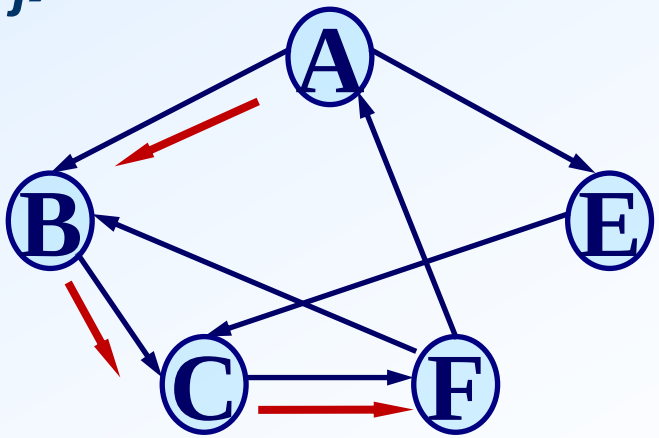


7.1 图的基本概念

□路径 在图 G = (V, E) 中，若从顶点 v_i 出发，沿一些边经过一些顶点 v_{i+1}, v_{i+2}, …, v_{j-1}，到达顶点v_j，则称顶点序列 (v_i, v_{i+1}, v_{i+2}, …, v_{j-1}, v_j) 为从顶点 v_i 到顶点 v_j 的**路径**。它经过的边(v_i, v_{i+1})、(v_{i+1}, v_{i+2})、…、(v_{j-1}, v_j) 应是属于E的边。



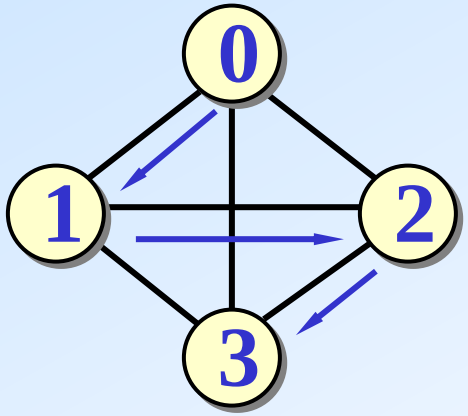
例1. 无向图从 0 到 3 的路径 {0,1,2,3}



例2. 有向图从 A 到 F 的路径{A,B,C,F}

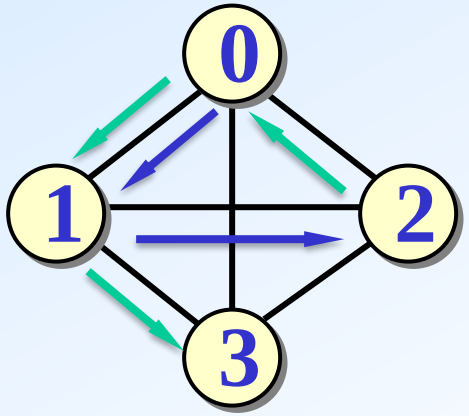
7.1 图的基本概念

- **简单路径** 若路径上各顶点 $v_i, v_{i+1}, \ldots, v_j$ 均不互相重复，则称这样的路径为**简单路径**。
- **简单回路** 若简单路径上第一个顶点 v_i 与最后一个顶点 v_j 重合，则称这样的简单路径为简单**回路**或简单**环**。

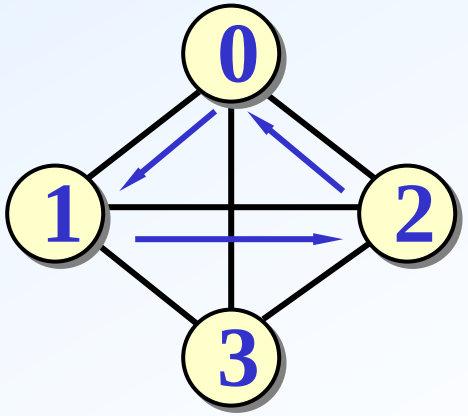


{0,1,2,3}

从顶点 0 到顶点 3 的路径



{0,1,2,0,1,3}

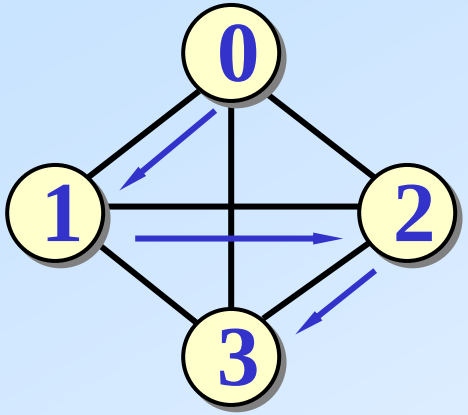


{0,1,2,0}

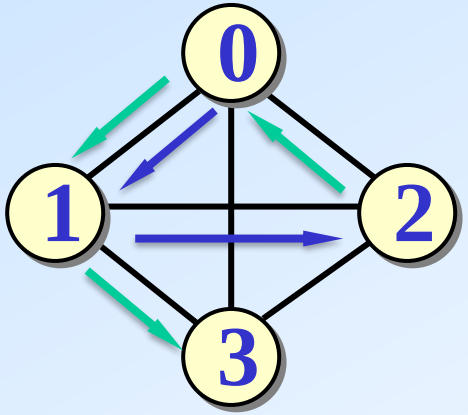
回路

7.1 图的基本概念

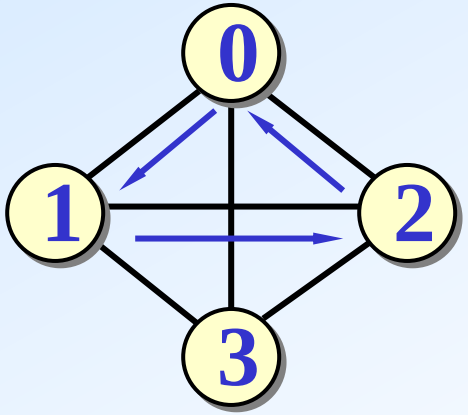
- 路径长度 非带权图的路径长度是指此路径上边的条数。带权图的路径长度是指路径上各边的权之和。



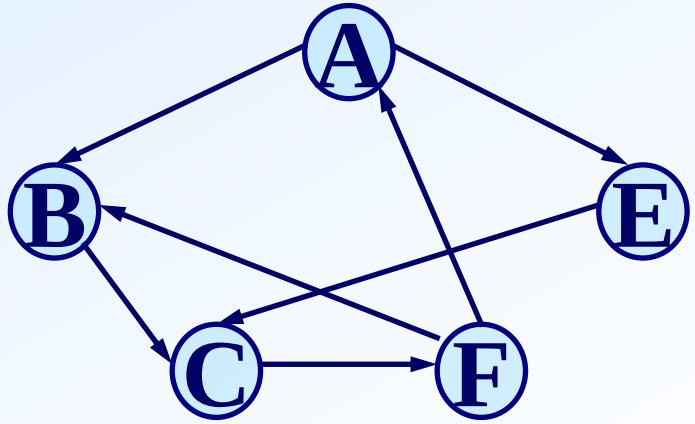
长度为3



长度为5



长度为3

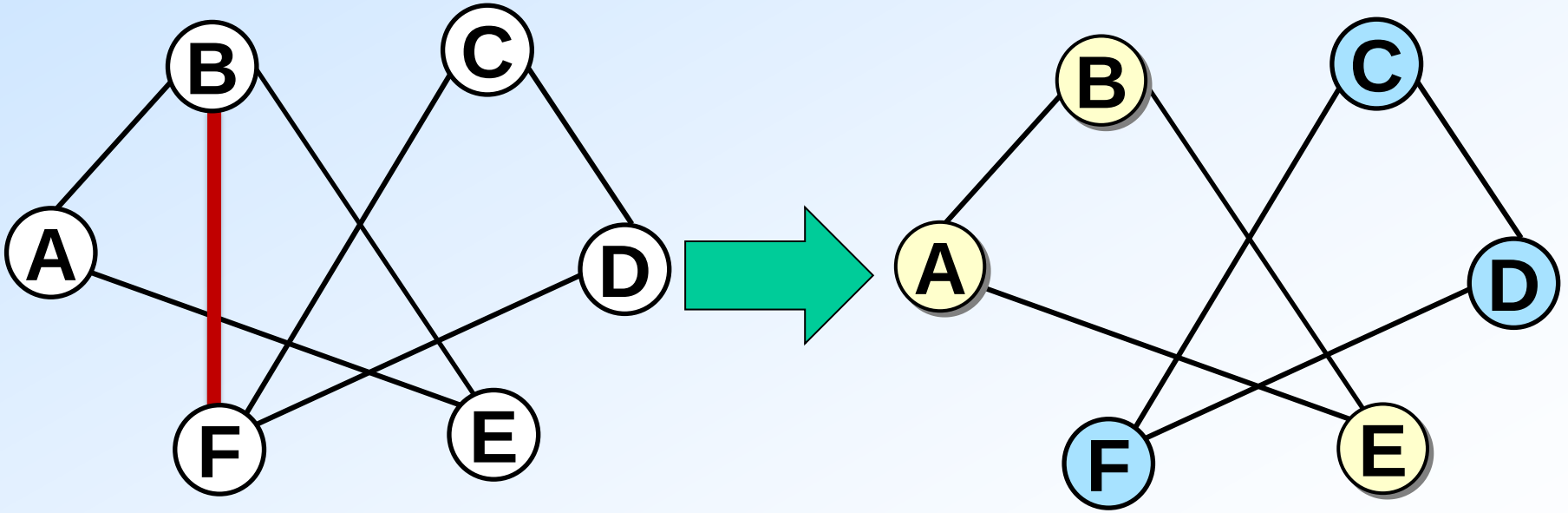


从 A 到 F 长度为 3 的路径{A,B,C,F}或{A,E,C,F}

7.1 图的基本概念

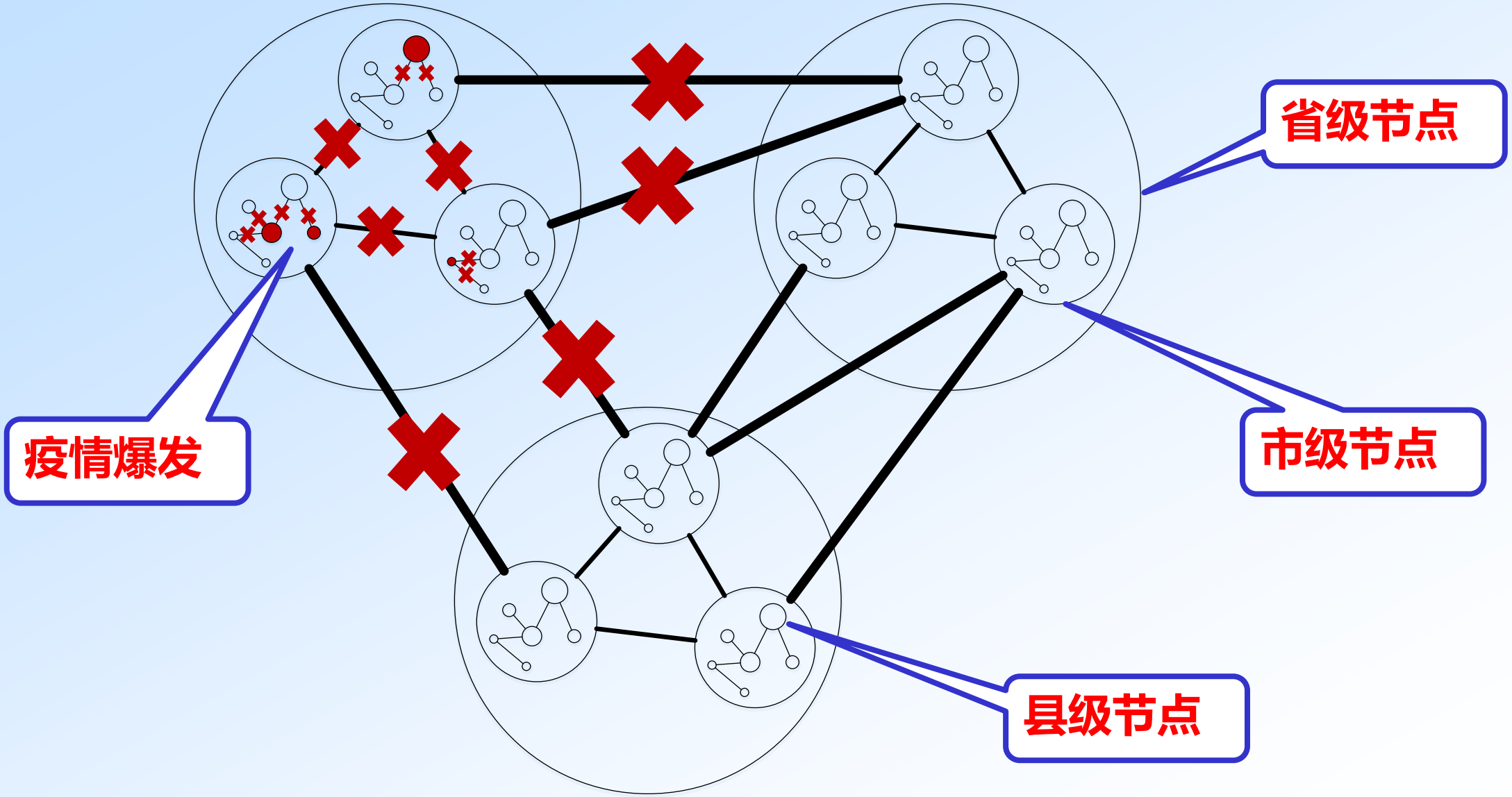
□连通图与连通分量　在无向图中，

- ■若从顶点 v_i 到顶点 v_j 有路径，则称顶点 v_i 与 v_j 是连通的。
- ■如果图中任意一对顶点都是连通的，则称此图是连通图。
- ■非连通图的**极大连通子图**叫做连通分量。



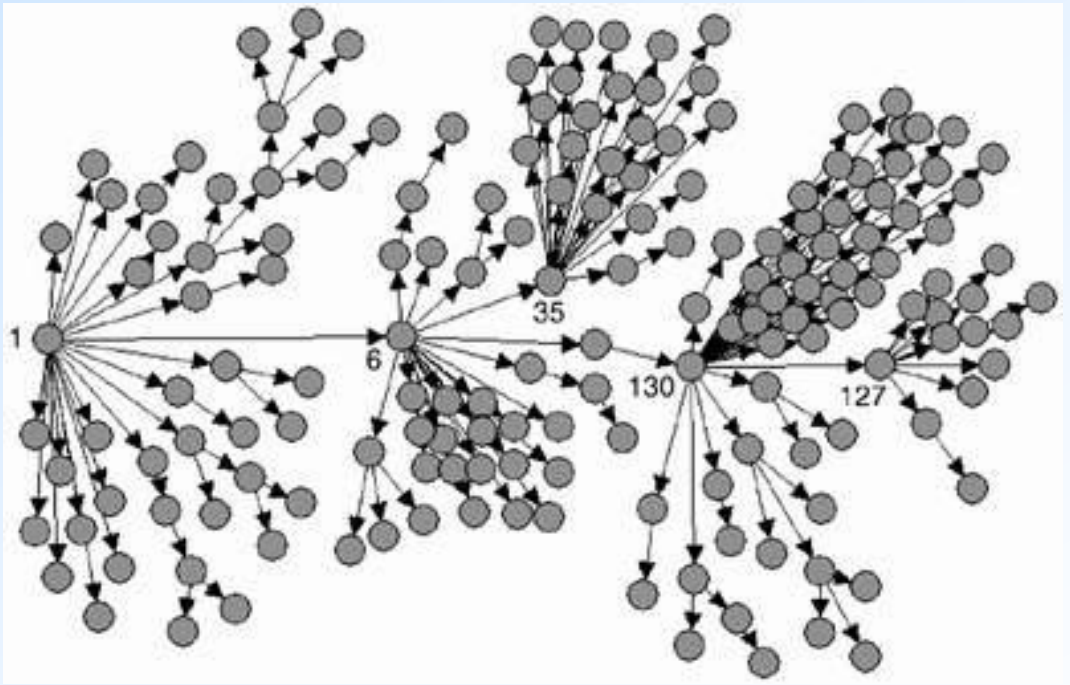
连通图　　非连通图，含2个连通分量

7.1 图的基本概念—连通图举例



7.1 图的基本概念—连通图举例

COVID-19传播模型（来源Wikipedia）

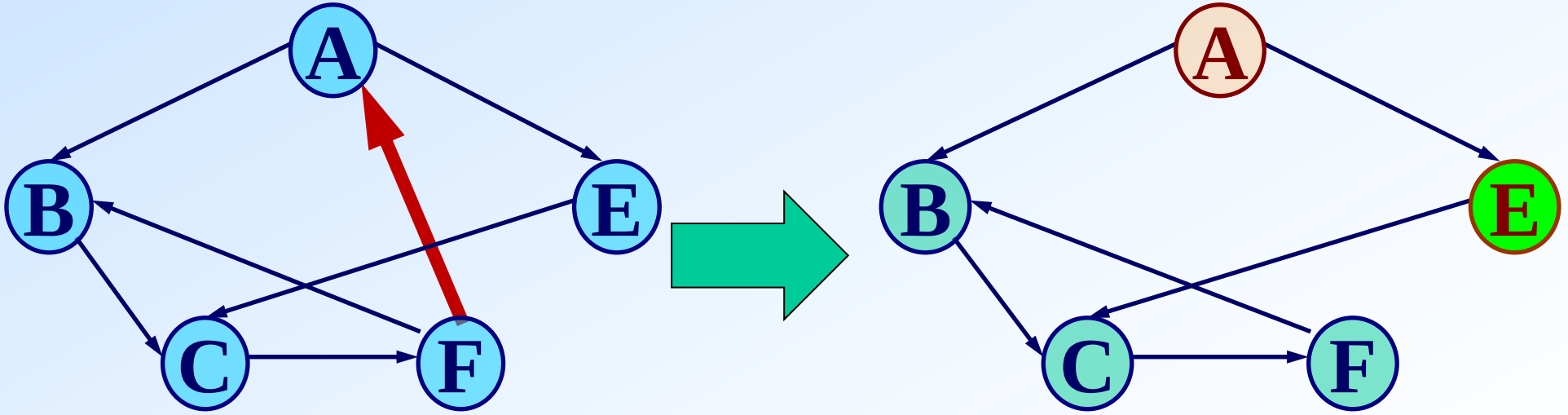


新加坡非典“超级传播者”的示意图

7.1 图的基本概念

□强连通图与强连通分量　在有向图中，

- ■若对于每一对顶点 v_i 和 v_j，都存在一条从 v_i 到 v_j 和从 v_j 到 v_i 的路径，则称此图是强连通图。
- ■非强连通图的**极大强连通子图**叫做强连通分量。

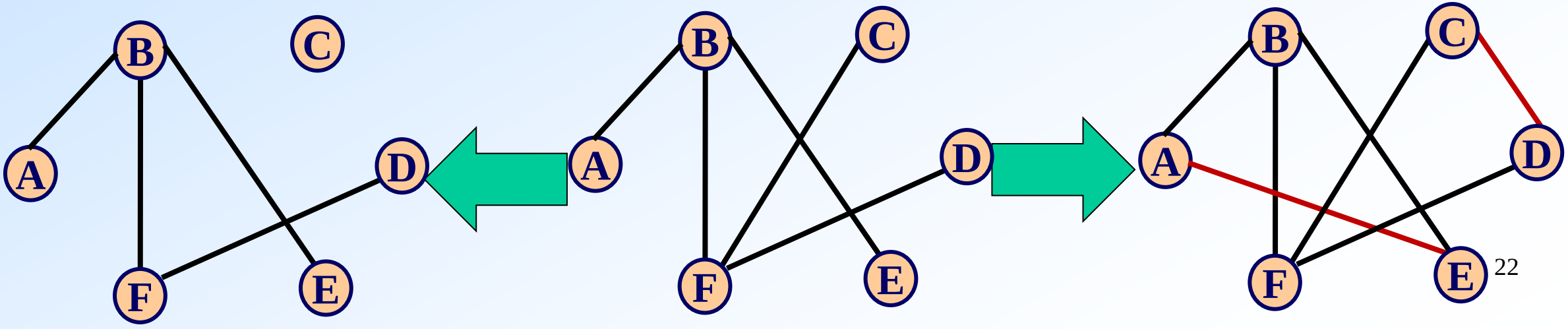


强连通图　　非强连通图，含3个强连通分量

7.1 图的基本概念

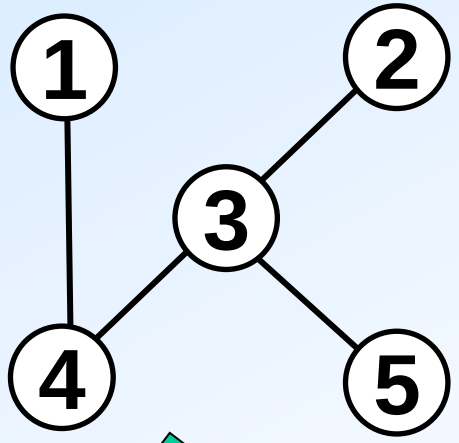
□生成树 假设一个连通图有 n 个顶点和 e 条边，其中 n-1 条边和 n 个顶点构成一个**极小连通子图**，称该极小连通子图为此连通图的生成树。

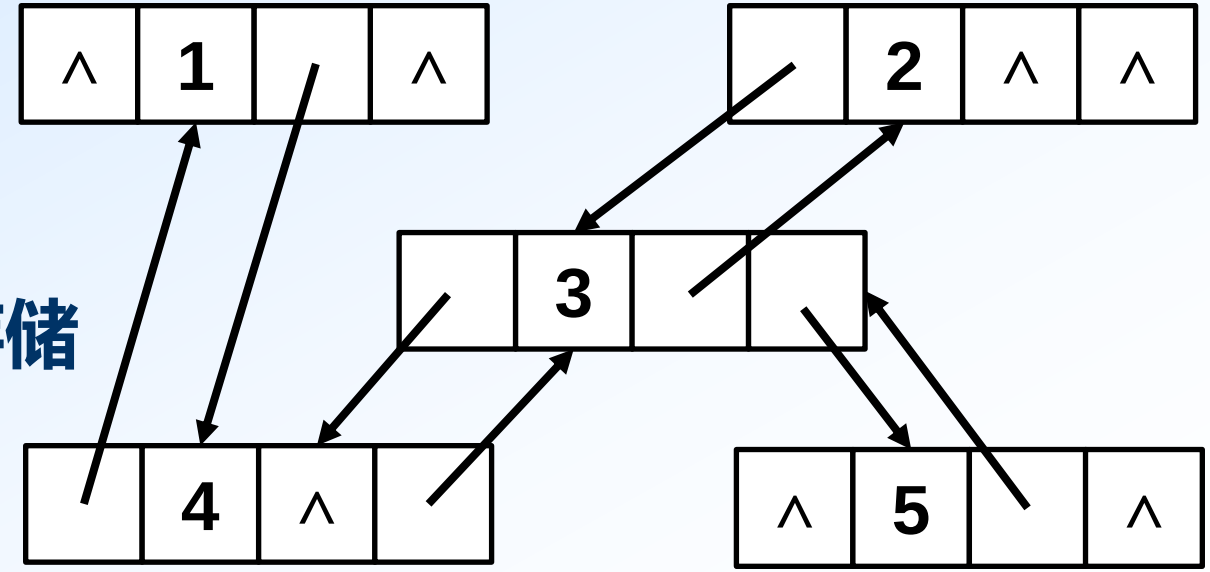
- ■在极小连通子图中增加一条边，则一定有环。
- ■在极小连通子图中去掉一条边，则成为非连通图。



7.2 图的存储结构—回顾

- 线性表、栈、队列、二叉树存储结构
  - 顺序表示、链式表示
- 树存储结构
  - 双亲表示法、孩子表示法、孩子兄弟表示法
- 广义表存储结构
  - 2种链式存储方法
- 图存储结构
  - 难以顺序存储和链式存储
  - 邻接矩阵、邻接表、
    十字链表、邻接多重表





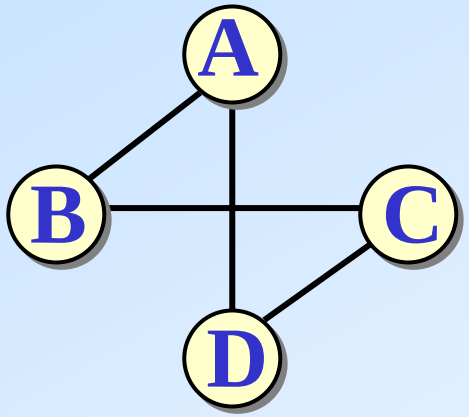
7.2.1 邻接矩阵

- 在图的邻接矩阵表示中，有一个记录各个顶点信息的**顶点表**，还有一个表示各个顶点之间关系的**邻接矩阵**。
- 设图 G = (V, E) 是一个有 n 个顶点的图，图的邻接矩阵是一个二维数组 G.edge[n][n]，定义：

$$G.edge[i][j] = \begin{cases} 1, & \text{若} < i, j > \in E \text{ 或 } (i, j) \in E \\ 0, & \text{其他} \end{cases}$$

7.2.1 无向图邻接矩阵

无向图邻接矩阵



$$G.edge = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix}$$

- 无向图的邻接矩阵是对称的
- 在无向图中，统计第 i 行 (列) 1 的个数可得顶点 v_i 的度。

7.2.1 有向图邻接矩阵

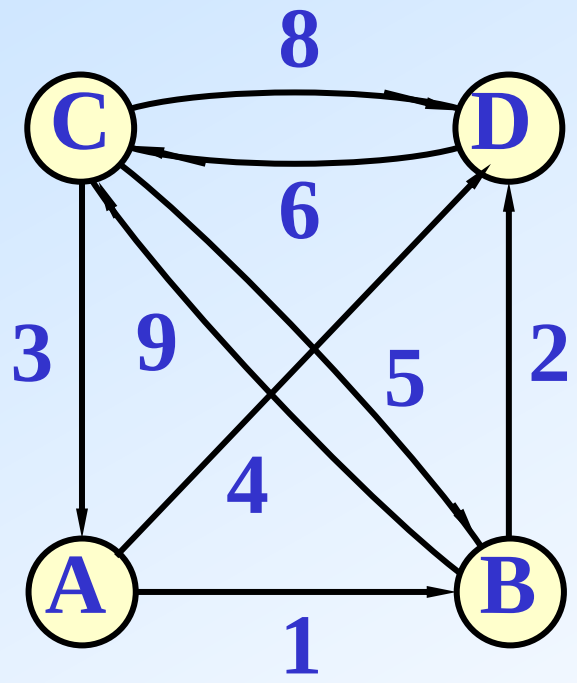
- 有向图邻接矩阵

$$G.edge = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$

- 有向图的邻接矩阵可能是不对称的；
- 在有向图中，统计第 i 行 1 的个数可得顶点 v_i 的出度，统计第 j 列 1 的个数可得顶点 v_j 的入度。

7.2.1 网络的邻接矩阵

$$G.edge[i][j]=\begin{cases} W(i,j), & \text{若} i\neq j \text{且} <i,j>\in E \text{或} (i,j)\in E \\ \infty, & \text{若} i\neq j \text{且} <i,j>\notin E \text{或} (i,j)\notin E \\ 0, & \text{若} i==j \end{cases}$$



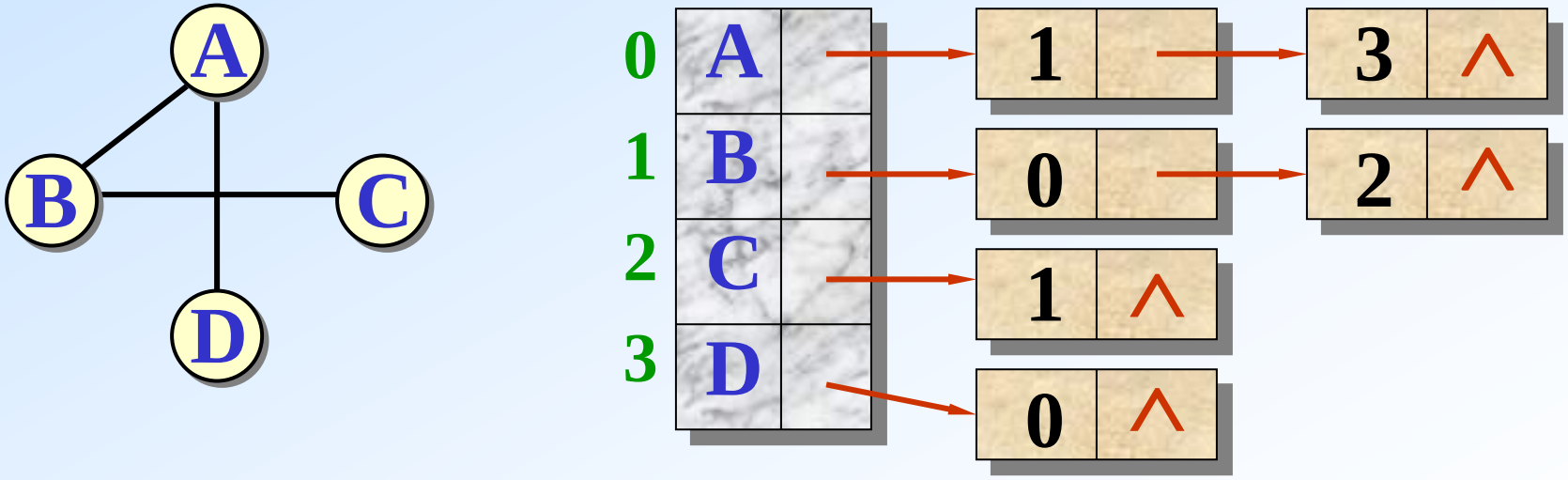
$$G.edge=\begin{bmatrix} 0 & 1 & \infty & 4 \\ \infty & 0 & 9 & 2 \\ 3 & 5 & 0 & 8 \\ \infty & \infty & 6 & 0 \end{bmatrix}$$

7.2.1 邻接矩阵的形式描述

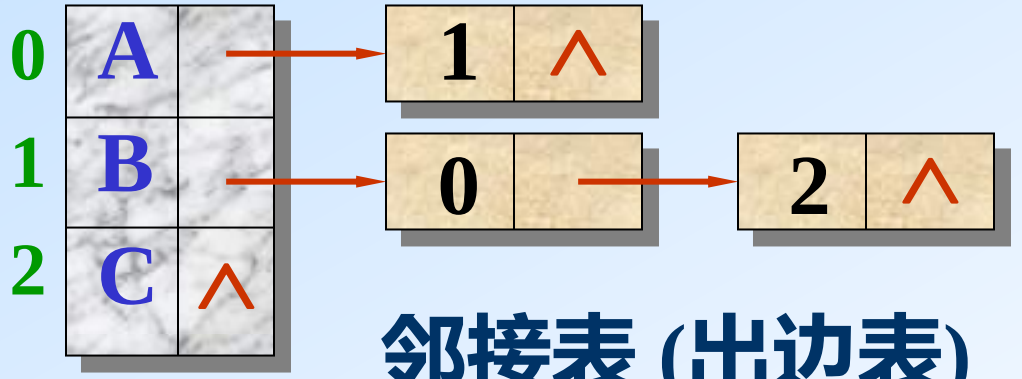
```
#define MaxValue INT_MAX     // 最大值∞
const int MaxVexNum = 10;    // 最大顶点个数
typedef char VertexData;     // 顶点数据类型
typedef int EdgeData;        // 边上权值类型
typedef struct {
   VertexData VexList[MaxVexNum];                    // 顶点表
   EdgeData edge[MaxVexNum][MaxVexNum];  // 邻接矩阵
   int n, e;                                         // 图中当前的顶点数与边数
} MTGraph;
```

7.2.2 邻接表

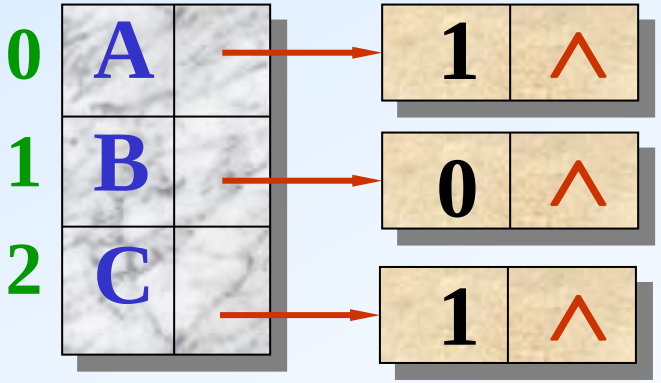
- 邻接表：是图的一种链式存储结构。
- 无向图的邻接表



7.2.2 有向图的邻接表



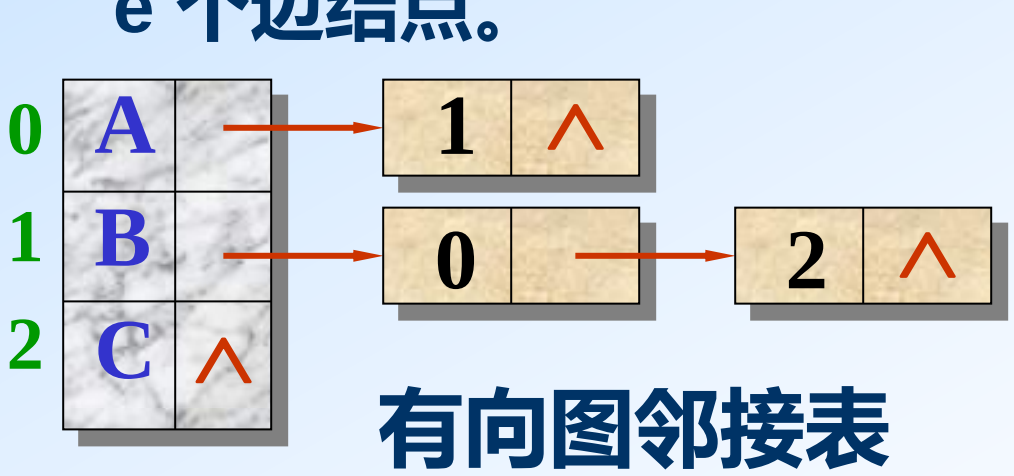
邻接表 (出边表)

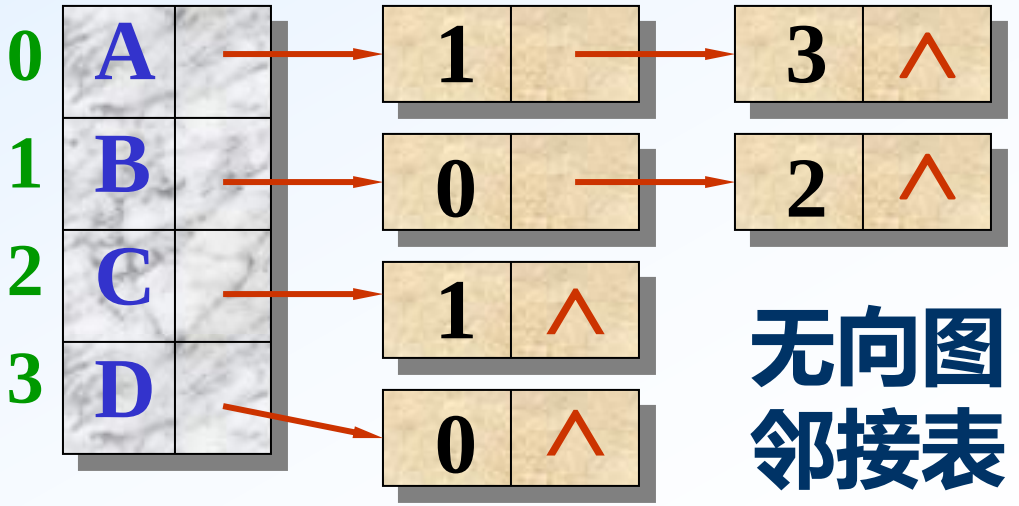


逆邻接表 (入边表)

7.2.2 邻接表—小结

- 在邻接表的边链表中，**边结点的链入顺序任意**，视边结点输入次序而定。
- 设图中有 n 个顶点，e 条边，则
  - 用**邻接表**表示**无向图**时，需要 n 个顶点结点，2e 个边结点；
  - 用**邻接表**表示**有向图**时，若不考虑逆邻接表，只需 n 个顶点结点，e 个边结点。





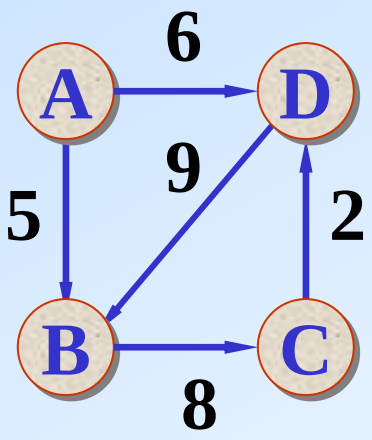
7.2.2 网络的邻接表

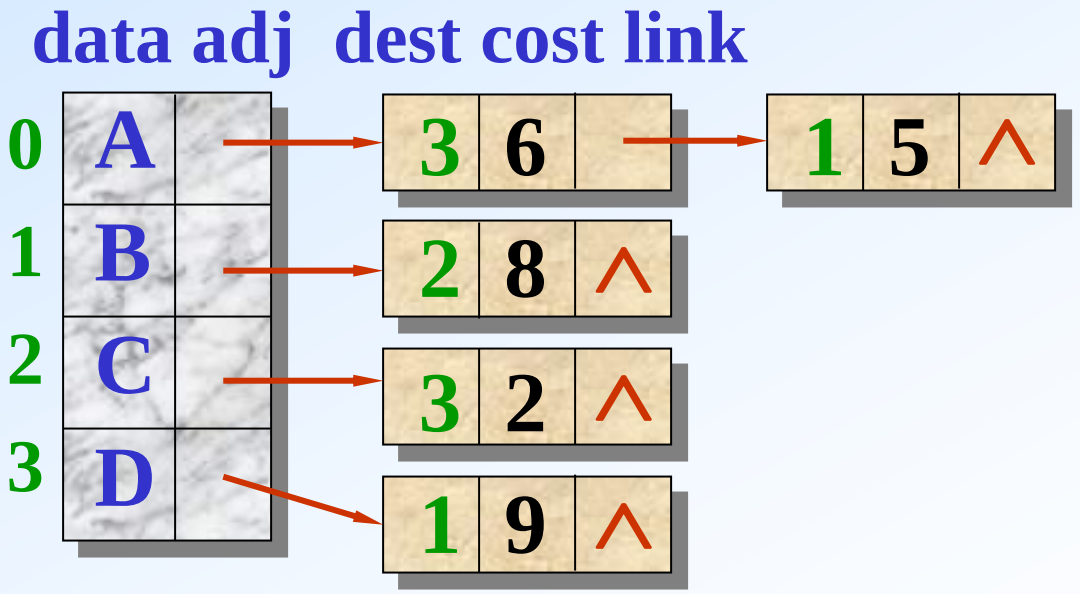
data	adj

图顶点的结构

dest	cost	link

边结点的结构





(顶点表)　　(邻接表)

- 带权图的边结点中保存该边上的权值 cost

7.2.2 邻接表的形式描述

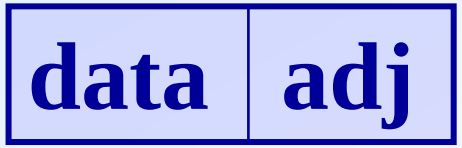
```
typedef char VertexData;        // 顶点数据类型
typedef int EdgeData;           // 边权值的类型

typedef struct node {           // 边结点
    int dest;                   // 目标顶点位置
    EdgeData cost;              // 边的权值
    struct node * link;         // 下一边链接指针
} EdgeNode;
```

边结点的结构

7.2.2 邻接表的形式描述

```
typedef struct {          // 顶点结点
   VertexData data;       // 顶点数据域
   EdgeNode * adj;        // 边链表头指针
} VertexNode;
```

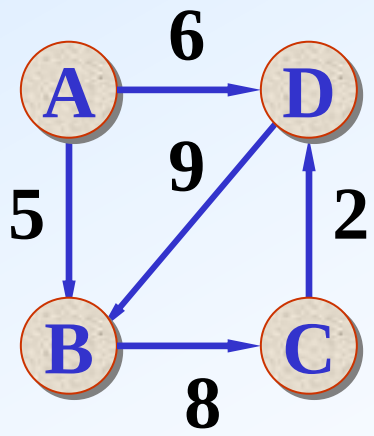


图顶点的结构

```
typedef struct {          // 图的邻接表
    VertexNode VexList [MaxVexNum];   // 邻接表
    int n, e;             // 图中当前的顶点个数与边数
} AdjGraph;
```

7.2.2 邻接表的构造算法

```
void CreateGraph (AdjGraph &G) {
   scanf ("%d %d", &G.n, &G.e);          // 输入顶点个数和边数
   for (int i = 0; i < G.n; i++) {
      scanf ("%c", &G.VexList[i].data);  // 输入顶点信息
      G.VexList[i].adj = NULL;
   }
```

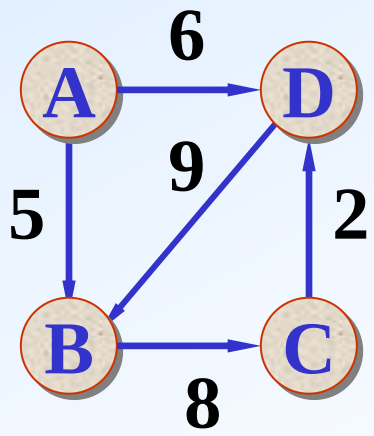


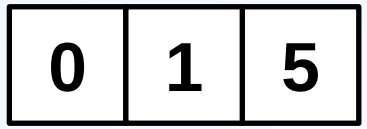
data adj　　dest cost link

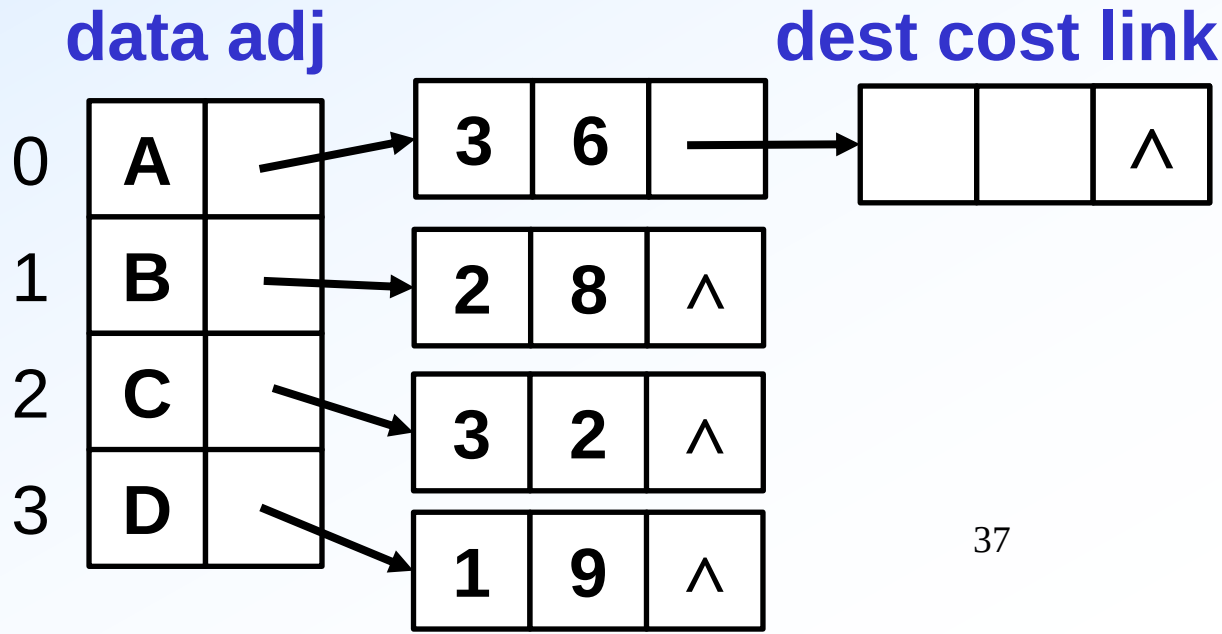
	data	adj
0	A	∧
1	B	∧
2	C	∧
3	D	∧

7.2.2 邻接表的构造算法（续）

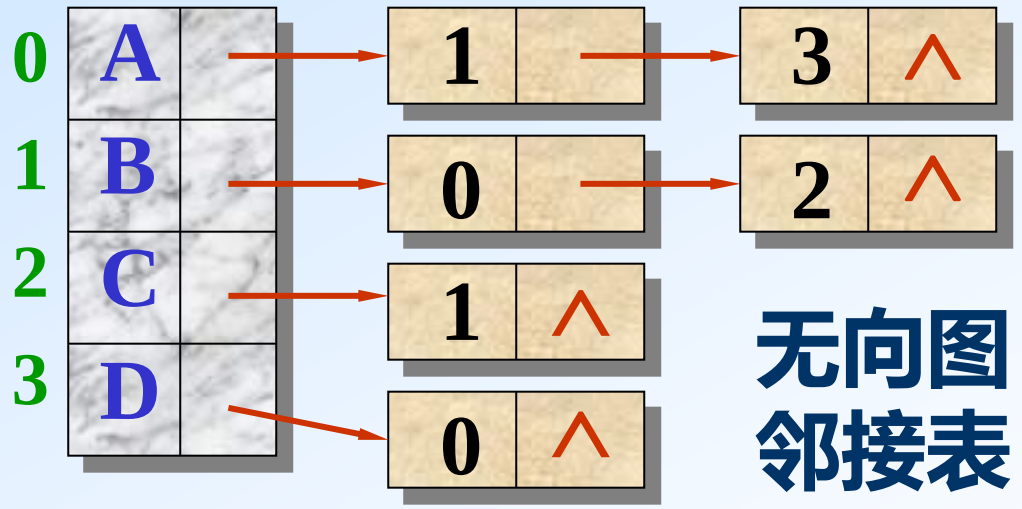
```
for (i = 0; i < G.e; i++) {
    scanf ("%d %d %d", &tail, &head, &weight); // 逐条边输入
    EdgeNode *p = new EdgeNode;
    p->dest = head;   p->cost = weight;
    // 链入第 tail 号链表的前端（回想链表插入算法）
    p->link = G.VexList[tail].adj;
    G.VexList[tail].adj = p;
} // end for
} // end CreateGraph
```







7.2.2 无向图邻接表的构造算法




```
for (i = 0; i < G.e; i++) {
   EdgeNode *p = new EdgeNode;
   p->dest = head;
   p->cost = weight;
   p->link = G.VexList[tail].adj;  G.VexList[tail].adj = p;

   p = new EdgeNode;
   p->dest = tail;   p->cost = weight;
   // 链入第 head 号链表的前端
   p->link = G.VexList[head].adj;  G.VexList[head].adj = p;
} // end for
```

7.3 图的遍历

回顾

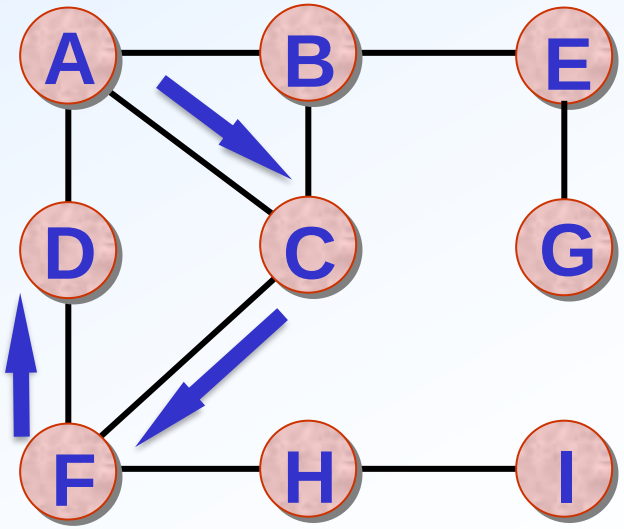
- 单链表遍历
- 二叉树遍历
  - 树的遍历就是按某种次序访问树中的结点，要求每个结点访问一次且仅访问一次。
  - 前序、中序与后序遍历
  - 线索二叉树时使用
- 树的遍历
  - 深度优先遍历
    - 先根次序遍历
    - 后根次序遍历

对比

- 从图中某一顶点出发访遍图中所有的顶点，且使每个顶点仅被访问一次，这一过程就叫做**图的遍历**。
- 图中**可能存在回路**，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。
- **两种图的遍历方法：**
  - 深度优先搜索 DFS
  - 广度优先搜索 BFS

7.3 图的遍历

- 为了避免重复访问，可设置一个标志顶点是否被访问过的**辅助数组 visited[ ]**。
- 辅助数组 visited[ ] 的初始状态为 0，在图的遍历过程中，一旦某一个顶点 i 被访问，就立即让 visited[i] 为 1，防止它被多次访问。

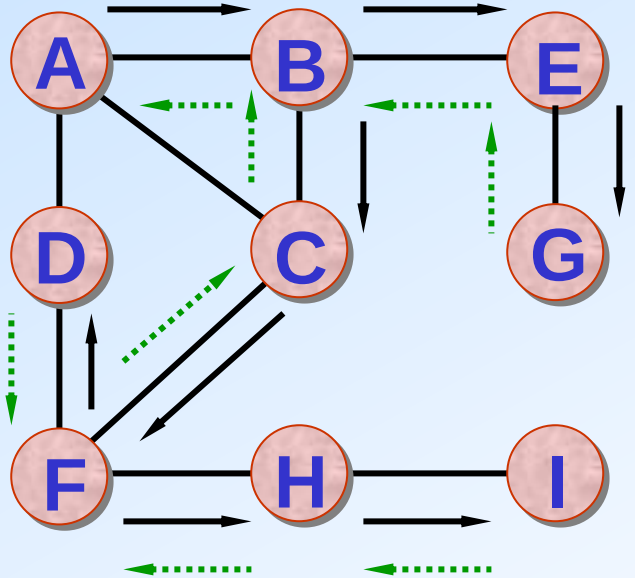


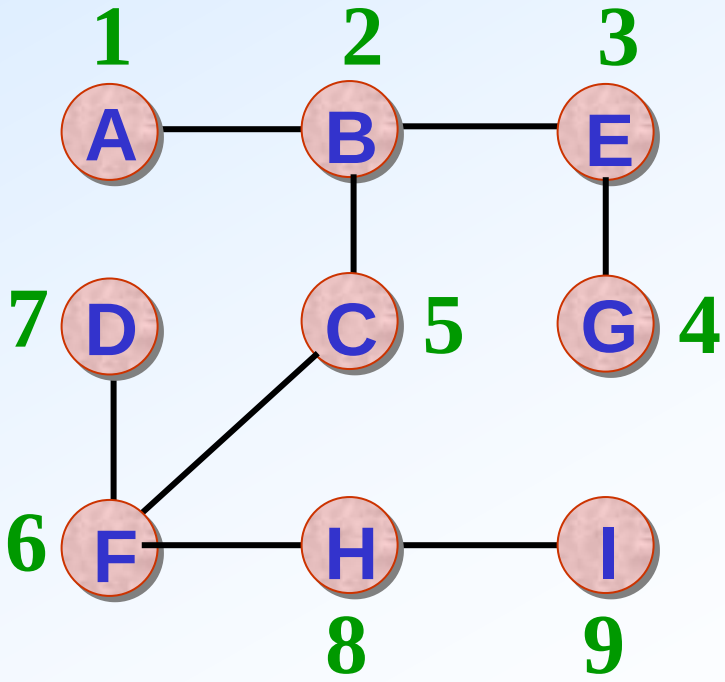
- 沿路径访问 A，C，F 和 D 后，只能返回到 A；
- visited[] 已存储 [1, 0, 1, 1, 0, 1, 0, 0, 0]

7.3.1 深度优先搜索DFS

□深度优先搜索过程

顶点访问顺序：ABEGCFDHI





深度优先生成树

7.3.1 深度优先搜索DFS

DFS算法基本步骤

1. 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一未访问过的邻接顶点 w_1；
2. 再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2；
3. 然后再从 w_2 出发，进行类似的访问，… 如此进行下去；
4. 直至到达所有的邻接顶点都被访问过的顶点 u 为止。

7.3.1 深度优先搜索DFS

□DFS算法基本步骤

5. 接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点；
6. 如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；
7. 如果没有，就再退回一步进行搜索；
8. 重复上述过程，直到连通图中所有顶点都被访问过为止。

7.3.1 DFS算法—主函数

```
void Graph_Traverse (AdjGraph G) {
   int *visited = new int [MaxVexNum];
   for (int i = 0; i < G.n; i++)
   visited[i] = 0;                          // 访问数组 visited 初始化
   for (int i = 0; i < G.n; i++)            // 解决连通分量问题
      if (!visited[i])
         DFS (G, i, visited);               // 从顶点 i 出发开始搜索
   delete [ ] visited;                      // 释放 visited
}
```

7.3.1 DFS函数

□DFS算法基本步骤

1. 在访问图中某一起始顶点 v 后，由 v 出发，访问它的任一未访问过的邻接顶点 w_1；
2. 再从 w_1 出发，访问与 w_1 邻接但还没有访问过的顶点 w_2；
3. 然后再从 w_2 出发，进行类似的访问，… 如此进行下去；
4. 直至到达所有的邻接顶点都被访问过的顶点 u 为止。

```
void DFS (AdjGraph G, int v, int visited[ ]) {
   printf (" %c " , GetValue (G, v));     // 访问顶点 v
   visited[v] = 1;                        // 顶点 v 作访问标记
   int w = GetFirstNeighbor (G, v);       // 取 v 的第一个邻接顶点 w
   while (w != -1) {                      // 若邻接顶点 w 存在
      if (!visited[w])                    // 若顶点 w 未访问过，递归访问顶点 w
         DFS (G, w, visited);
      // 取顶点 v 排在 w 后的下一个邻接顶点
      w = GetNextNeighbor (G, v, w);
   }
}
```

□DFS算法基本步骤

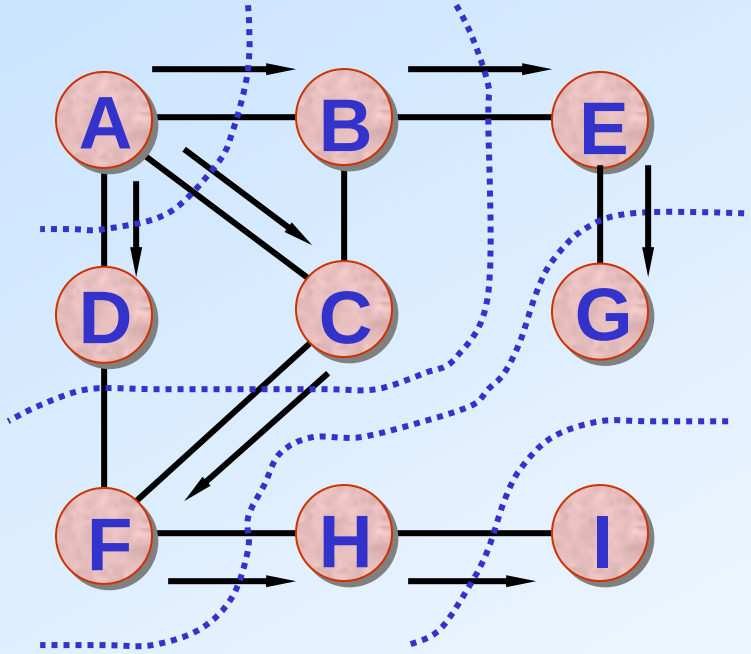
5. 接着，退回一步，退到前一次刚访问过的顶点，看是否还有其它没有被访问的邻接顶点；
6. 如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；
7. 如果没有，就再退回一步进行搜索；
8. 重复上述过程，直到连通图中所有顶点都被访问过为止。

7.3.1 DFS算法分析

- DFS算法采用**递归**或**栈**的方式
- 采用DFS算法遍历图时，对图中每个顶点至多调用一次DFS函数
- 耗费的时间取决于所采用的**存储结构**。
  - 假设图顶点个数为 n，边条数为 e。
  - 邻接矩阵作为图的存储结构时，其时间复杂度为$O(n^2)$
  - 邻接表作为的图的存储结构时，查找每个顶点的邻接顶点所需时间O(e)，其时间复杂度为 O(n + e)

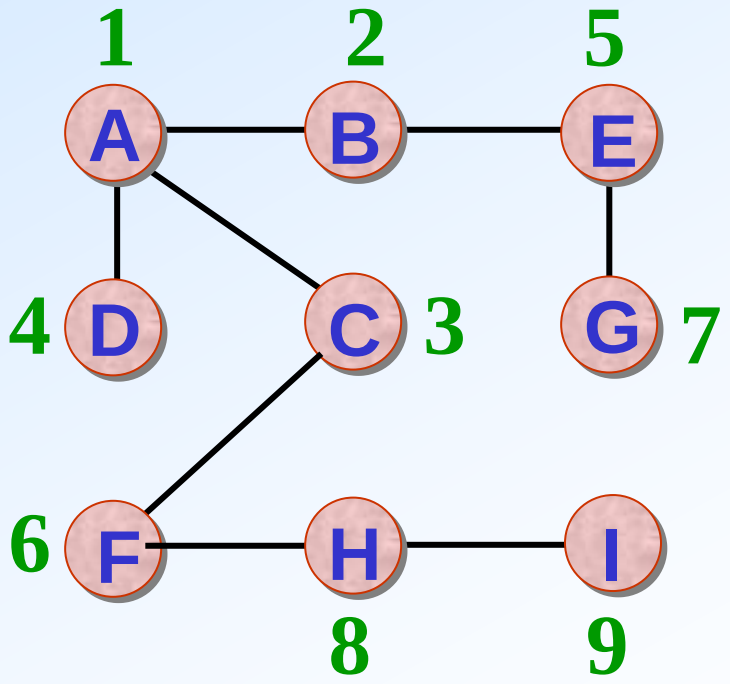
7.3.2 广度优先搜索BFS

□广度优先搜索过程　　顶点访问顺序：ABCDEFGHI



前进 ——→

分层 ⌒⌒⌒



广度优先生成树

7.3.2 广度优先搜索BFS

□BFS算法基本步骤

1. 访问起始顶点 v；
2. 由 v 出发，依次访问 v 的各个未被访问过的邻接顶点 w_1, w_2, …, w_t；
3. 再顺序访问 w_1, w_2, …, w_t 的所有还未被访问过的邻接顶点；
4. 再从这些访问过的顶点出发，再访问它们的所有还未被访问过的邻接顶点，… 如此做下去；
5. 直到图中所有顶点都被访问到为止。

7.3.2 广度优先搜索BFS

□BFS算法说明

- BFS是一种**分层**的搜索过程，每向前走一步可能访问一批顶点，不像BFS那样有往回退的情况。因此，**BFS不是一个递归的过程**。
- 为了实现逐层访问，算法中使用了一个**队列**，以记忆正在访问的这一层和下一层的顶点，以便于向下一层访问。
- 为避免重复访问，仍需要**辅助数组 visited[ ]**，给被访问过的顶点加标记。

7.3.2 BFS算法主函数

```
void Graph_Traverse (AdjGraph G) {
   int *visited = new int [MaxVexNum];
   for (int i = 0; i < G.n; i++ )
      visited [i] = 0;                    // 访问数组 visited 初始化
   for ( int i = 0; i < G.n; i++ )
      if (!visited[i])
         BFS (G, i, visited );            // 从顶点 i 出发开始搜索
   delete [ ] visited;                    // 释放 visited
}
```

7.3.2 BFS函数

```
void BFS (AdjGraph G, int v, int visited[ ]) {
   printf(" %d ",  GetValue (v));
   visited[v] = 1;                                  // 先访问顶点v
   Queue<int> q;
   InitQueue(&q);
   EnQueue (&q, v);                                 // 再令顶点v进队列
   while (!QueueEmpty (&q)  {
      DeQueue (&q, v);                              // 队头出列并赋值给v
      int w = GetFirstNeighbor (G, v);              // v的第1个邻居赋值给w
```

7.3.2 BFS函数（续）

```
    while ( w != -1 ) {    // 若邻接顶点 w 存在
        if ( !visited[w] ) { // 未访问过
            printf(" %d ", GetValue (w) );
            visited[w] = 1;
            EnQueue (&q, w);
        }
        w = GetNextNeighbor (G, v, w);
    }                  // 重复检测 v 的所有邻接顶点
  } // end while (!QueueEmpty (&q))
}
```

7.3.2 BFS算法分析

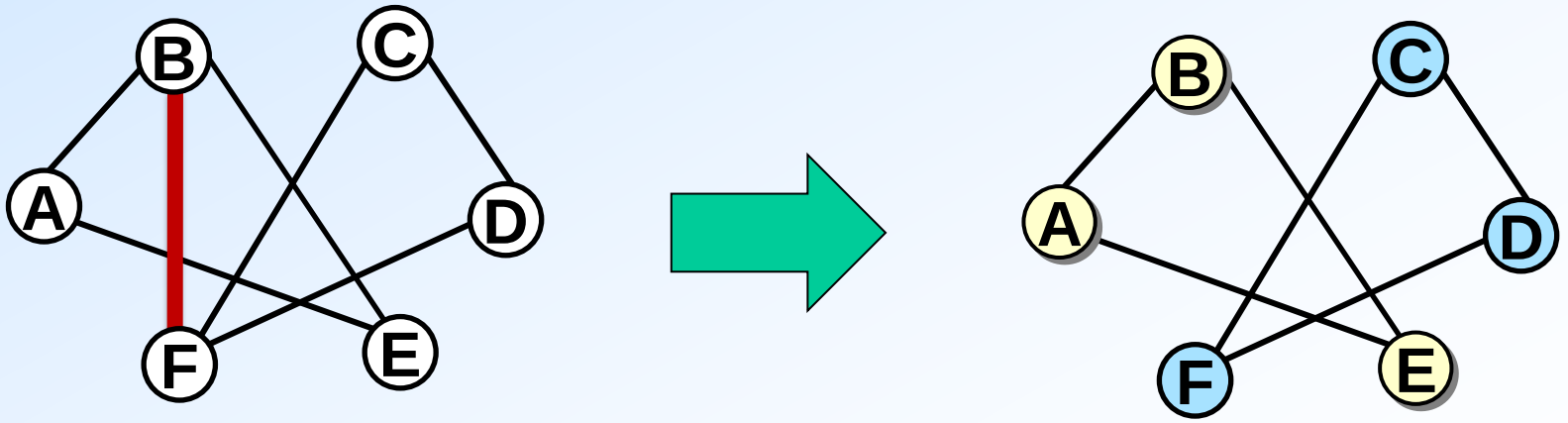
- BFS遍历图的时间复杂度和DFS遍历的时间复杂度相同；
- 耗费的时间均取决于所采用的存储结构
  - 假设图顶点个数为 n，边条数为 e
  - 邻接矩阵作为图的存储结构时，其时间复杂度为 $O(n^2)$
  - 邻接表作为的图的存储结构时，其时间复杂度为 $O(n + e)$
- 不同之处仅在于对顶点的访问顺序不同

7.3.2 DFS与BFS算法对比

- 深度优先搜索DFS
  - 对于解决遍历和求所有问题有效，对于问题搜索深度小的时候处理速度迅速，然而在深度很大的情况下效率不高。
- 广度优先搜索BFS
  - 对于解决最短或最少问题特别有效，而且寻找深度小，但缺点是内存耗费量大，需要开大量的数组单元用来存储状态。

7.4 图的连通性

- 连通分量
- 当**无向图**为**非连通图**时，调用一次DFS算法或BFS算法，只能访问到一个最大连通子图(**连通分量**)的所有顶点。
- 若从无向图的每个连通分量中的一个顶点出发进行遍历，可求得无向图的所有连通分量。



连通图　　非连通图，含2个连通分量

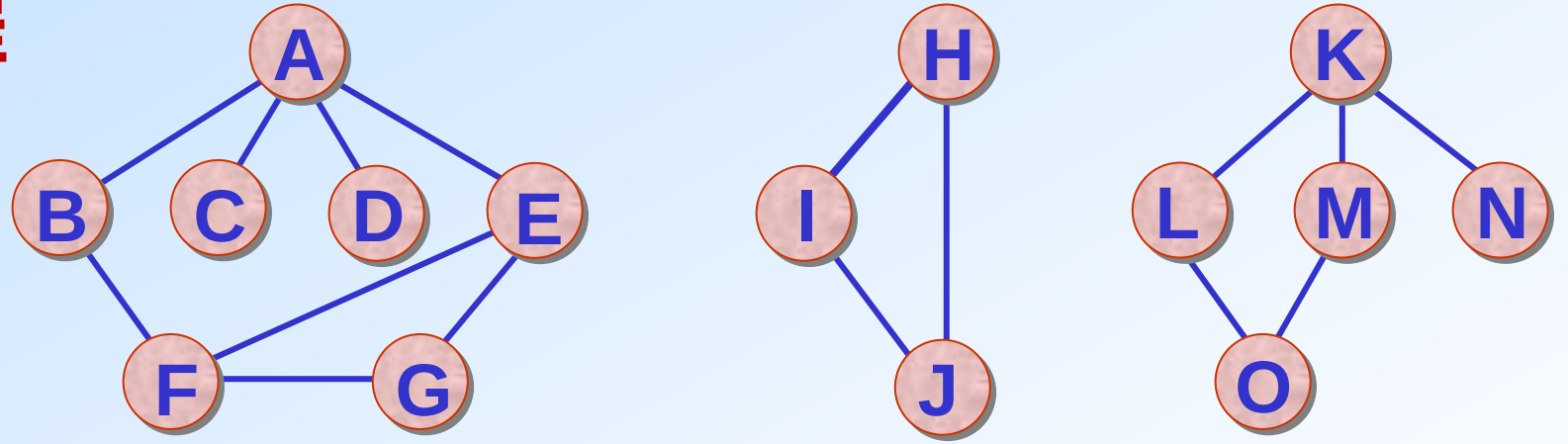
7.4.1 连通分量

- **连通分量求解基本思路：**
- **求连通分量的算法需要对图的每一个顶点进行检测：**
  - **若已被访问过，则该顶点一定是落在图中已求得的连通分量上；**
  - **若还未被访问，则从该顶点出发遍历图，可求得图的另一个连通分量。**
- **对于非连通的无向图，所有连通分量的生成树组成了非连通图的生成森林。**

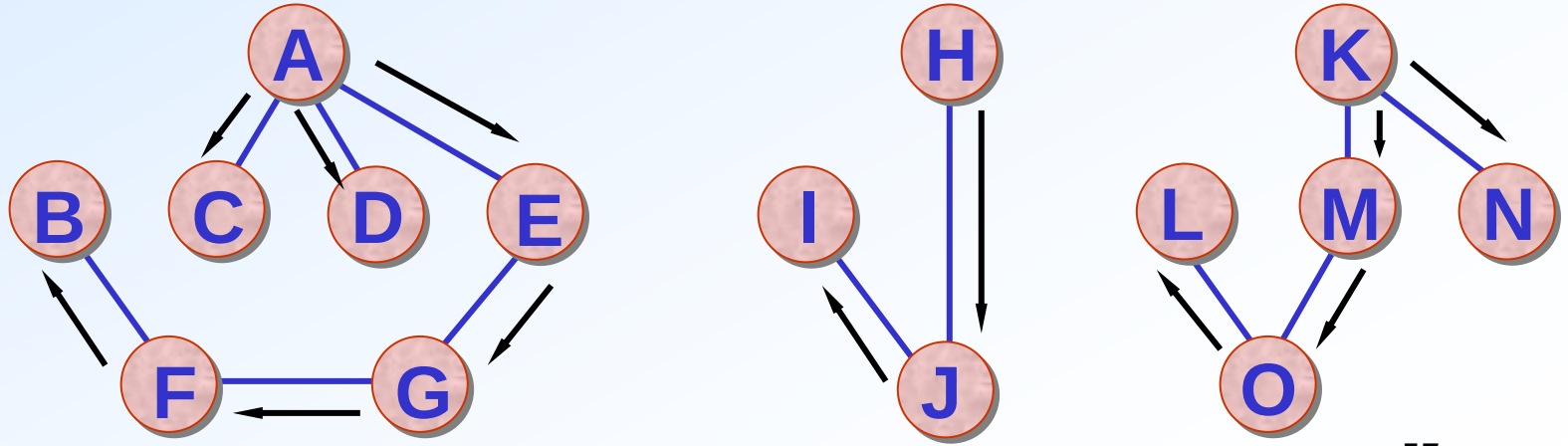
7.4.1 连通分量

□连通分量求解过程

非连通无向图的
三个连通分量



非连通图的连通分量
的极小连通子图

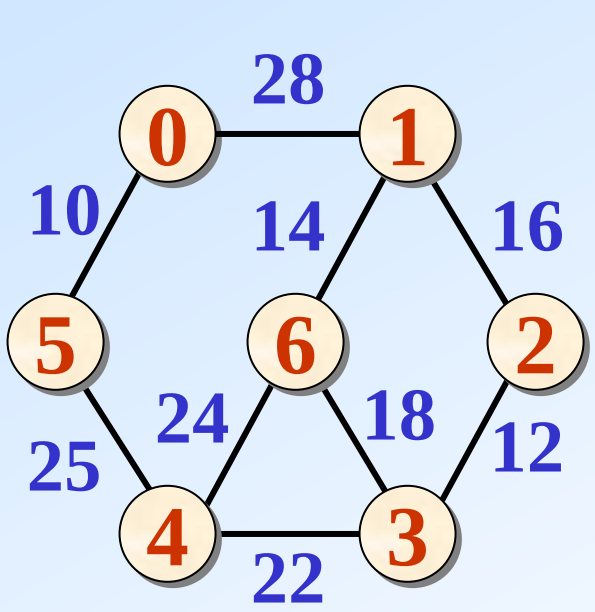


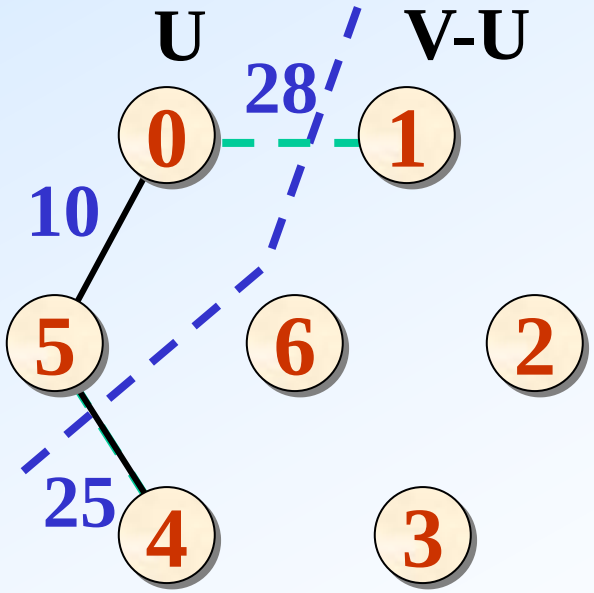
7.5 最小生成树

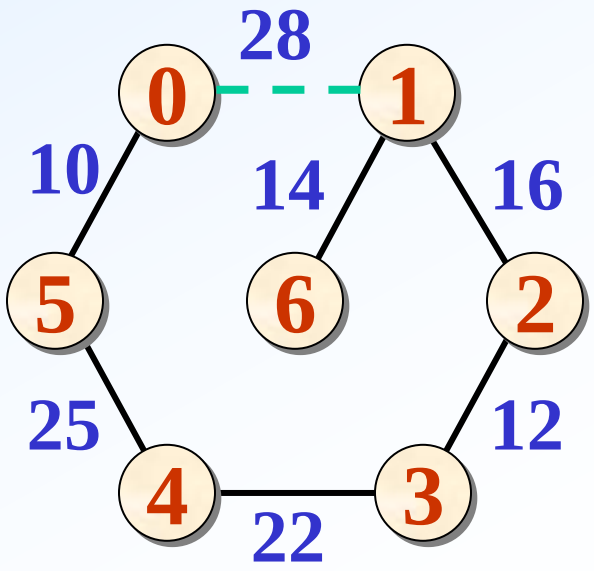
- 按照生成树的定义，
  - 使用不同的遍历图的方法，可以得到不同的生成树；
  - 从不同的顶点出发，也可能得到不同的生成树。
- 构造最小生成树MST的准则
  - 必须且仅用该网络中的 n-1 条边来联结网络中的 n 个顶点；
  - 不能产生回路；
  - 各边权值总和最小。
- 两种算法：
  - 普里姆(Prim)算法、克鲁斯卡尔 (Kruskal)算法

7.5 最小生成树—MST性质

- 假设 $N=(V,E)$ 是一个连通网络，U 是顶点集 V 的一个非空子集。
- 若 (u,u') 是一条具有最小权值（代价）的边，其中 $u\in U$，$u'\in V-U$，则必存在一棵包含边(u,u')的最小生成树。



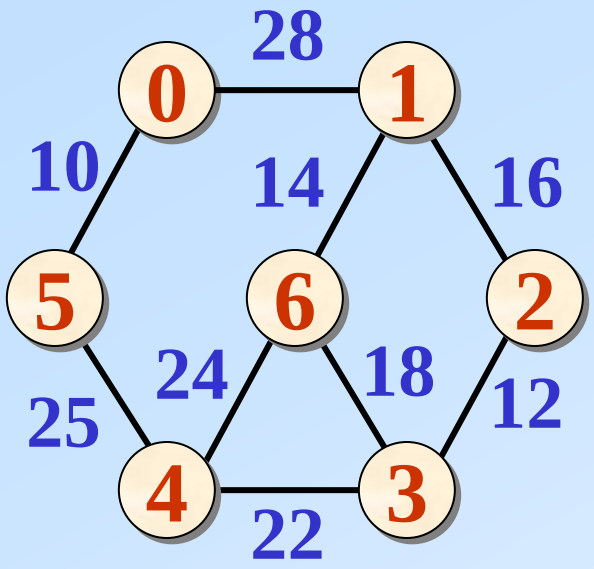




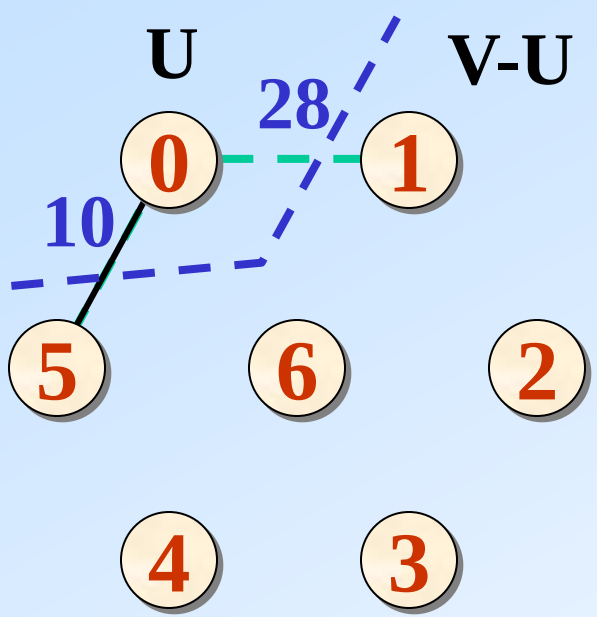
7.5 最小生成树—Prim算法基本思想

- 已知，连通网络 $N=(V,E)$，生成树顶点集合 U；

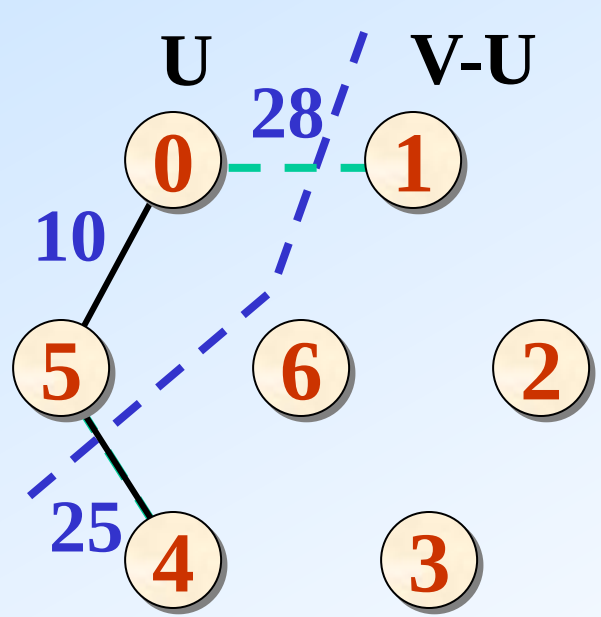
1. 从某一顶点 u_0 出发，选择与它关联的具有最小权值的边 (u_0,v)，将顶点 v 加入到生成树顶点集合 U 中。
2. 每次从一个顶点在 U 中，而另一个顶点不在 U (即 $V-U$) 中的各条边中选择**权值最小**的边 (u,u')，把它的顶点 v 加入到集合 U 中。
3. 直至网络中的所有顶点都加入到生成树顶点集合 U 中为止。


28
0
1
10
14
16
5
6
2
24
18
25
12
4
3
22

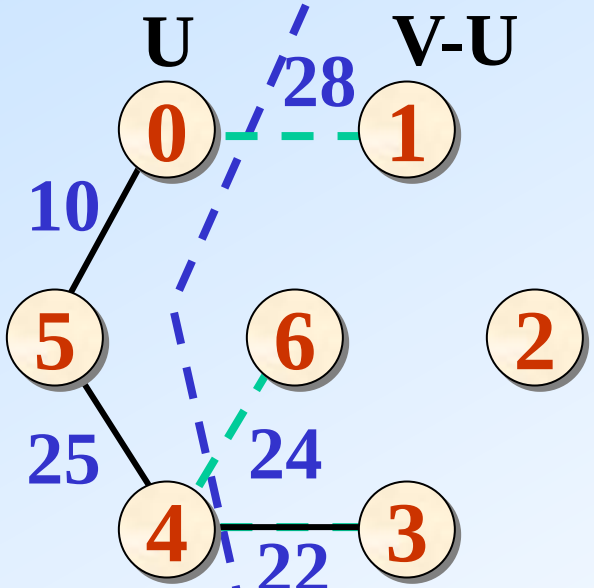
原图


U
V-U
28
0
1
10
5
6
2
4
3

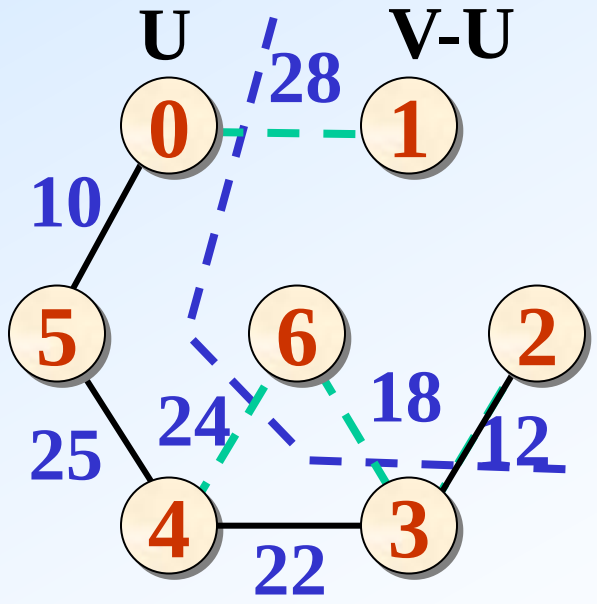
(a)


U
V-U
28
0
1
10
5
6
2
25
4
3

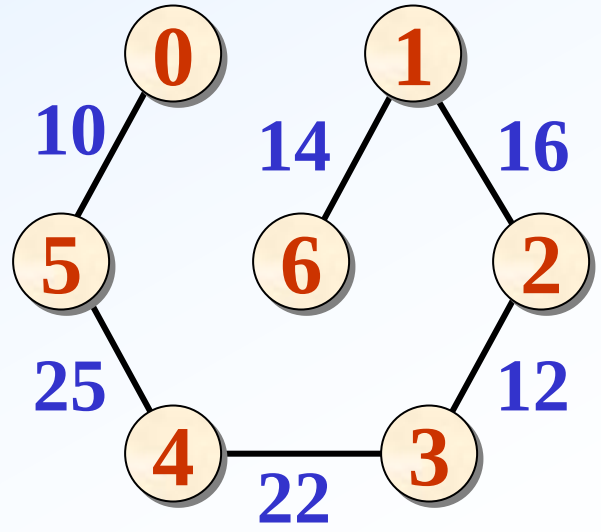
(b)


U
V-U
28
0
1
10
5
6
2
25
24
4
3
22

(c)


U
V-U
28
0
1
10
5
6
2
24
18
25
12
4
3
22

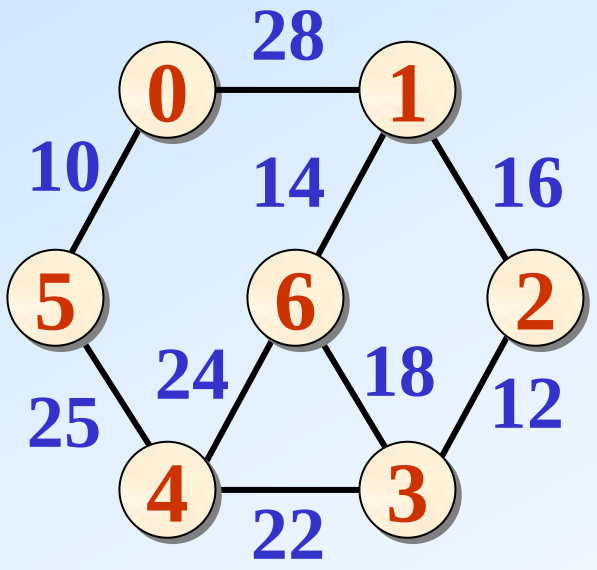
(d)


0
1
10
14
16
5
6
2
25
12
4
3
22

(e) (f)

7.5 最小生成树—Prim算法实现过程

- 在构造过程中，需要设置两个辅助数组：
  - lowcost[ ] 存放生成树顶点集合 U 内顶点到生成树外 $V-U$ 各顶点的各边上的**当前最小权值**；
  - adjvex[ ] 记录生成树顶点集合外各顶点 u' 距离集合内**哪个顶点 u** 最近，否则记为 -1。



邻接矩阵

$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

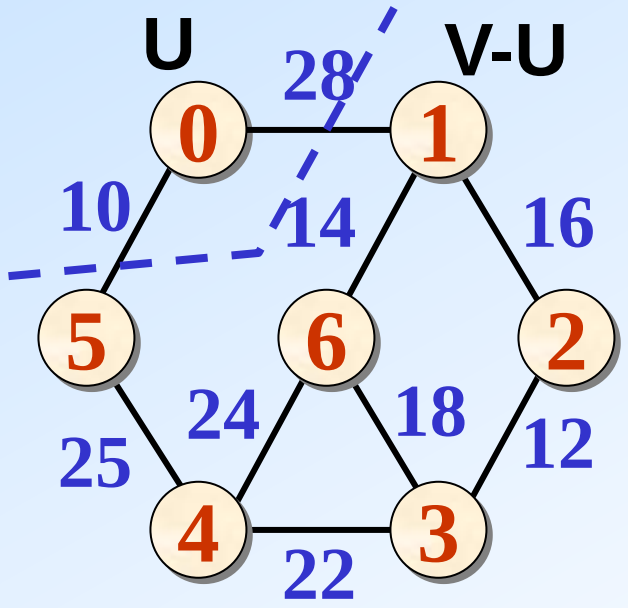
7.5 最小生成树—Prim算法实现过程

- 若选择从顶点0出发，即 $u_0 = 0$，则两个辅助数组的初始状态为：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
adjvex	−1	0	0	0	0	0	0

顶点集合V

顶点集合U

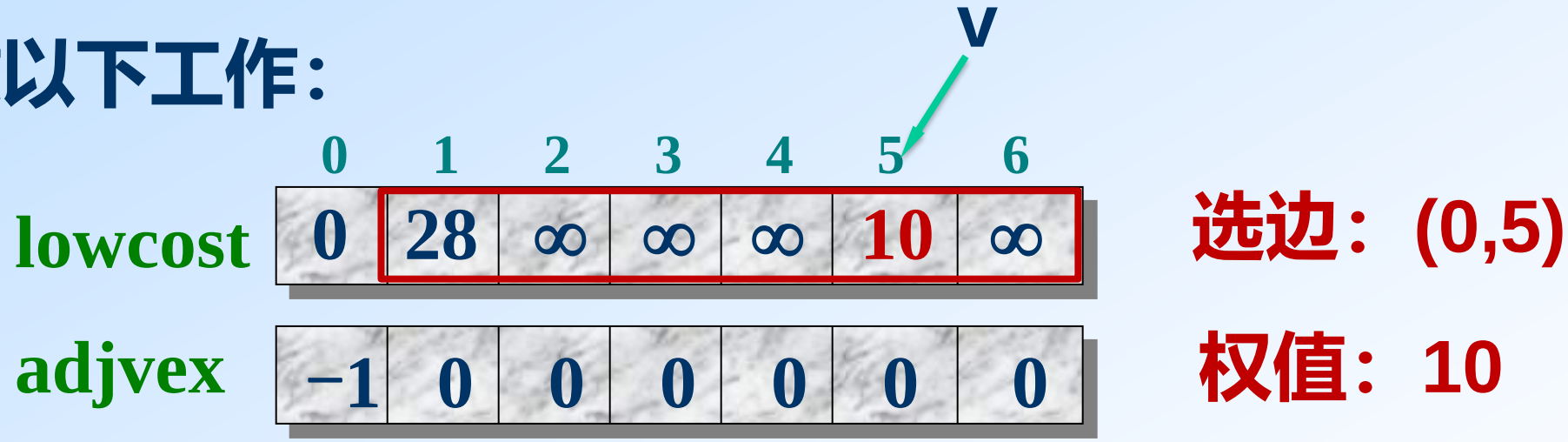


邻接矩阵

$$\begin{pmatrix} 0 & 28 & \infty & \infty & \infty & 10 & \infty \\ 28 & 0 & 16 & \infty & \infty & \infty & 14 \\ \infty & 16 & 0 & 12 & \infty & \infty & \infty \\ \infty & \infty & 12 & 0 & 22 & \infty & 18 \\ \infty & \infty & \infty & 22 & 0 & 25 & 24 \\ 10 & \infty & \infty & \infty & 25 & 0 & \infty \\ \infty & 14 & \infty & 18 & 24 & \infty & 0 \end{pmatrix}$$

7.5 最小生成树—Prim算法实现过程

- 然后反复做以下工作：



选边：(0,5)

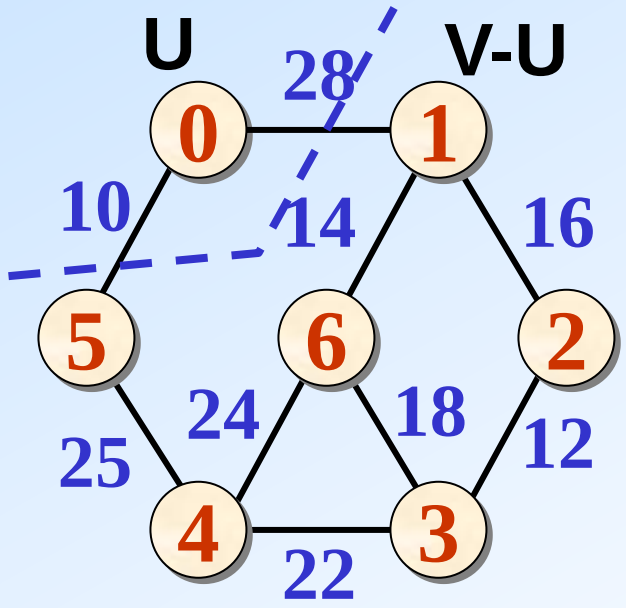
权值：10

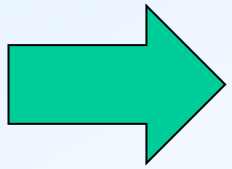
1. 在 lowcost[ ] 中选择 adjvex[i] ≠ -1 && lowcost[i] 最小的边，用 v 标记它。则选中的权值最小的边为 (adjvex[v], v)，相应的权值为 lowcost[v]。

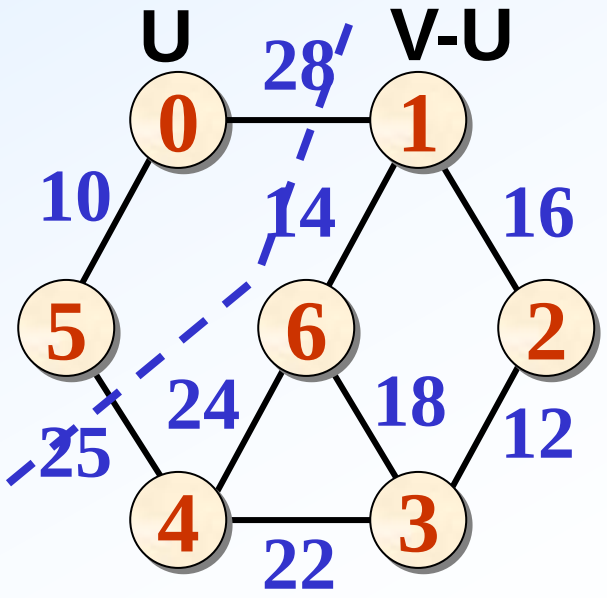
7.5 最小生成树—Prim算法实现过程

2. 将 adjvex[v] 改为-1，表示它已加入生成树顶点集合。

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	∞	10	∞
adjvex	−1	0	0	0	0	-1	0

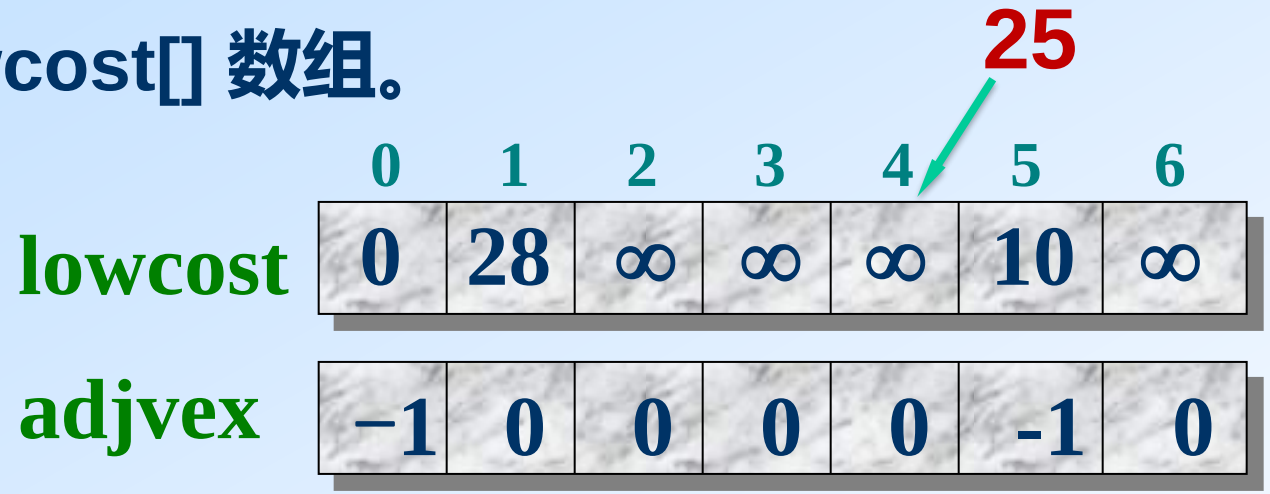




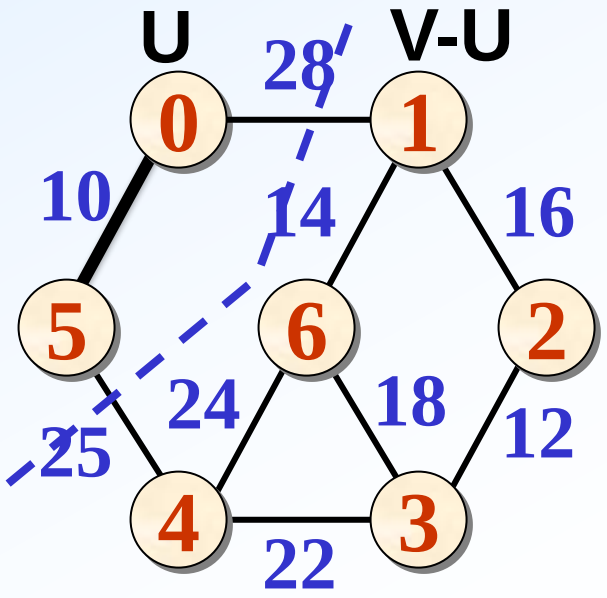


7.5 最小生成树—Prim算法实现过程

3. 更新 lowcost[] 数组。



- 取 lowcost[i] = min{ lowcost[i], edge[v][i] }，即用生成树顶点集合外各顶点 i 到刚加入该集合的新顶点 v 的距离 edge[v][i]，若比原lowcost[i] 更小，则更新lowcost[i]。

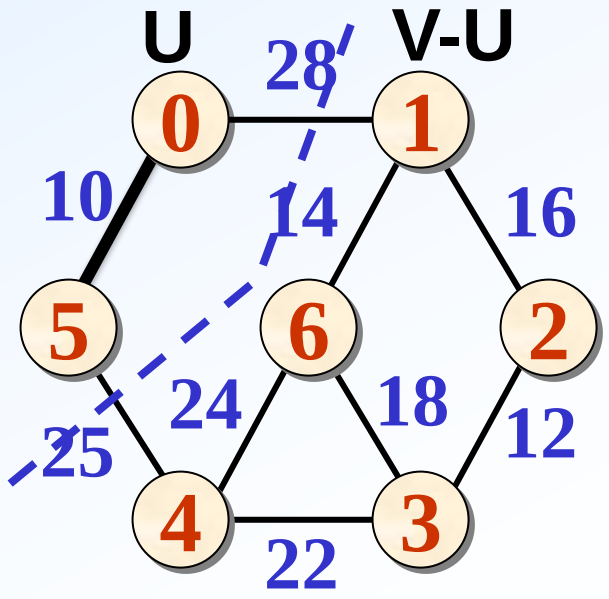


7.5 最小生成树—Prim算法实现过程

4. 更新 adjvex[] 数组。

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞
adjvex	−1	0	0	0	5	-1	0

- 若lowcost[i]被调整，则对应更新 adjvex[i] = v。表示生成树外顶点 i 到生成树内顶点 v 当前距离更近。



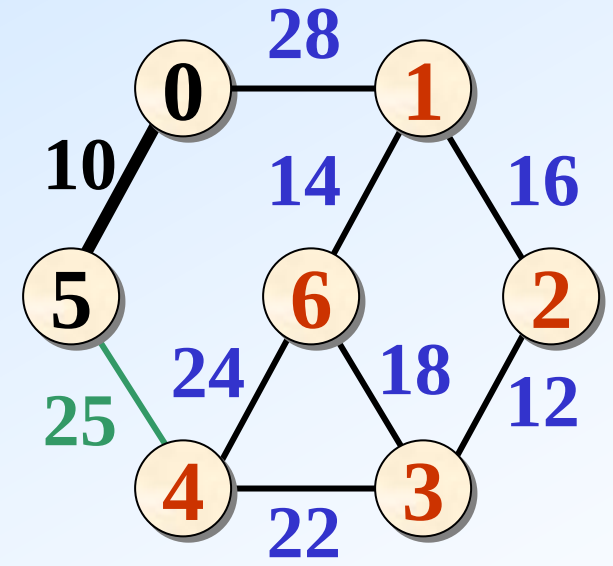
7.5 最小生成树—Prim算法实现过程

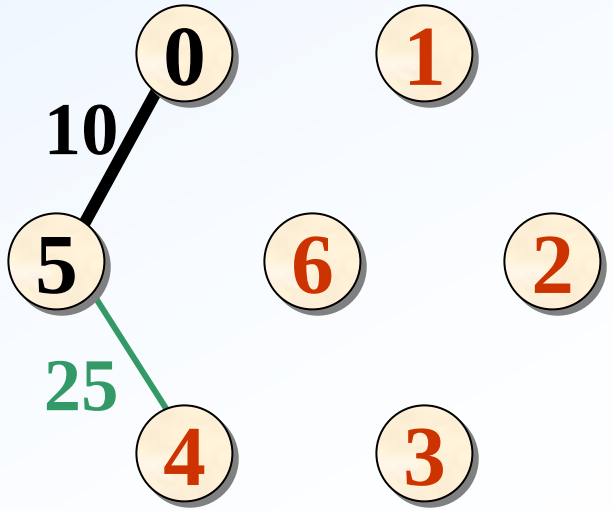
□顶点 v=5 加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	∞	25	10	∞
adjvex	−1	0	0	0	5	−1	0

选 v=4　　选边 (5,4)

边 (0,5,10) 加入生成树





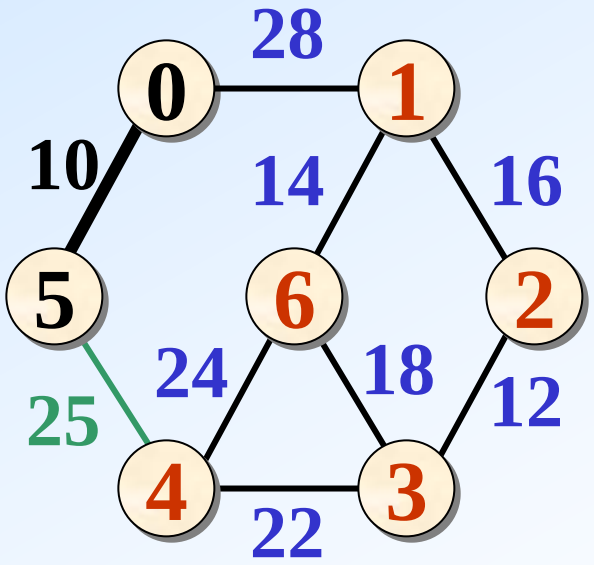
7.5 最小生成树—Prim算法实现过程

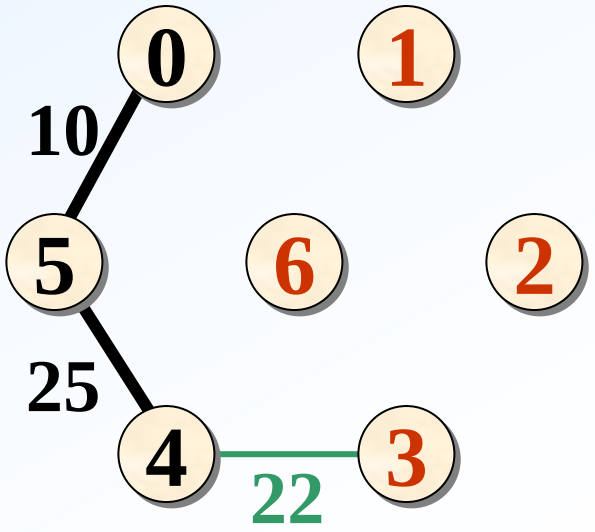
□顶点 v=4 加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	∞	22	25	10	24
adjvex	−1	0	0	4	−1	−1	4

选 v=3　　选边 (4,3)

边 (5,4,25) 加入生成树





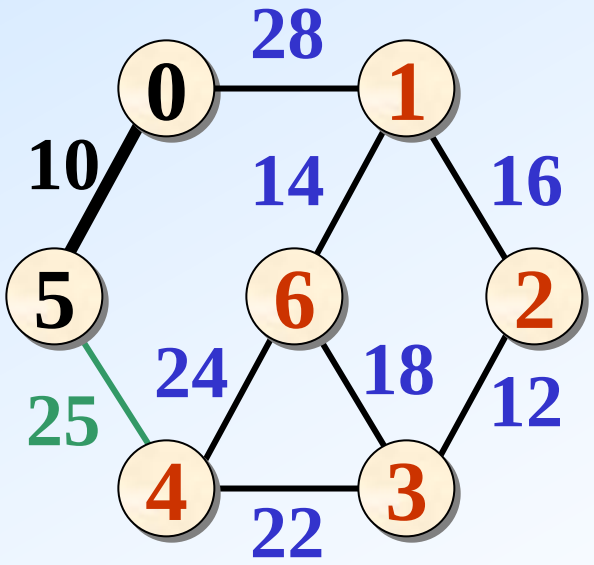
7.5 最小生成树—Prim算法实现过程

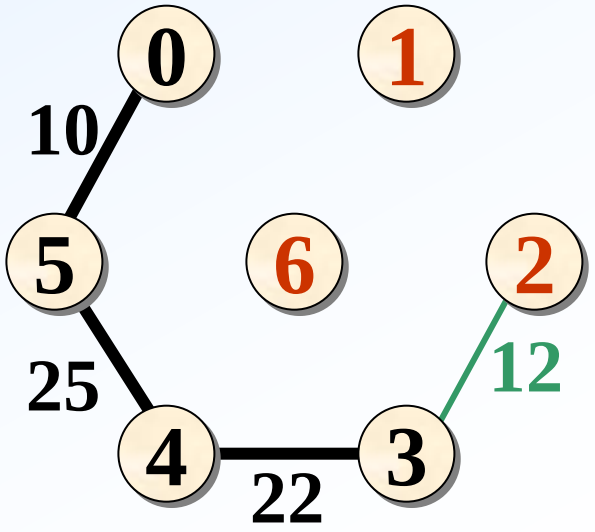
□顶点 v=3 加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	28	12	22	25	10	18
adjvex	−1	0	3	−1	−1	−1	3

选 v=2 ↑ 选边 (3,2)

边 (4,3,22) 加入生成树





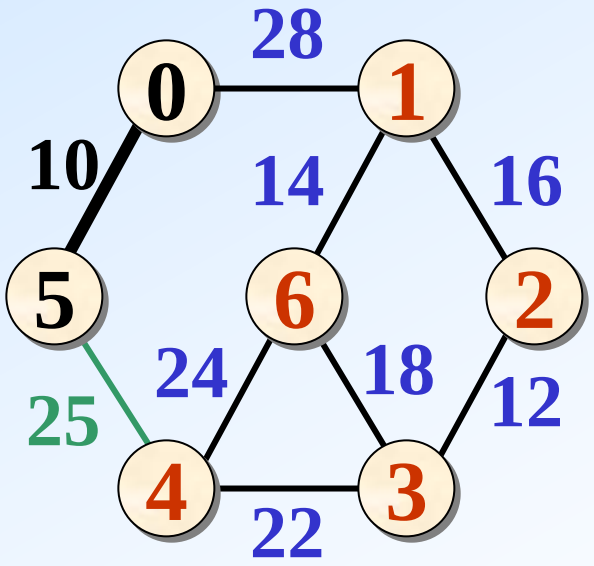
7.5 最小生成树—Prim算法实现过程

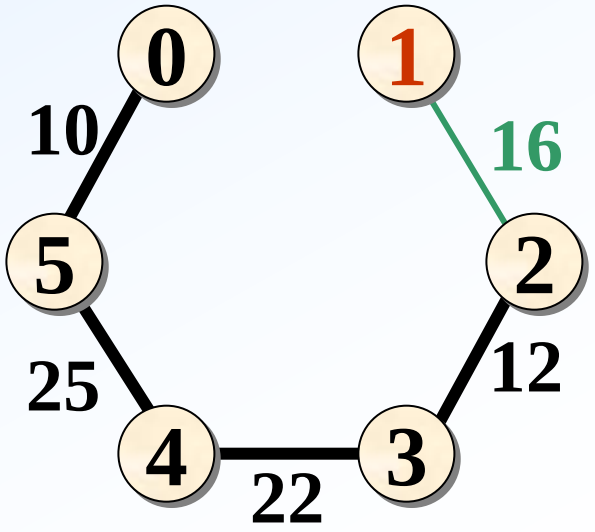
□顶点 v=2 加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	18
adjvex	−1	2	−1	−1	−1	−1	3

选 v=1　选边 (2,1)

边 (3,2,12) 加入生成树





7.5 最小生成树—Prim算法实现过程

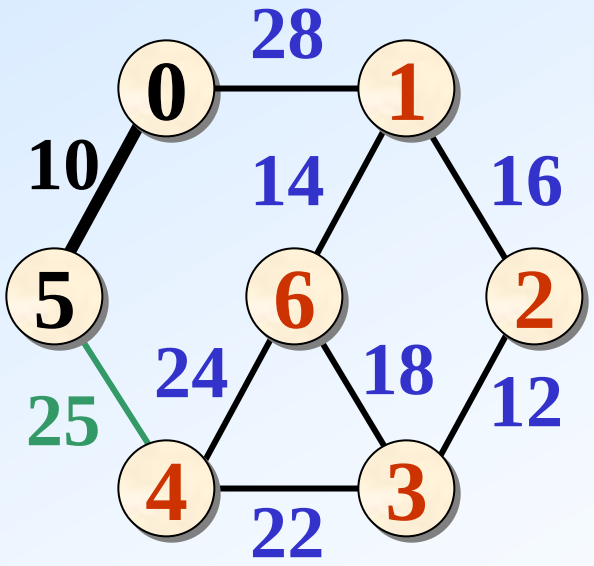
□顶点 v=1 加入生成树顶点集合：

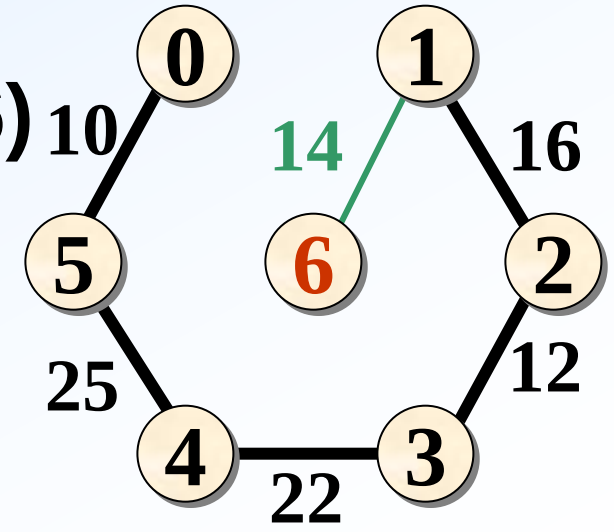
	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
adjvex	−1	−1	−1	−1	−1	−1	1

选 v=6

选边 (1,6)

边 (2,1,16) 加入生成树

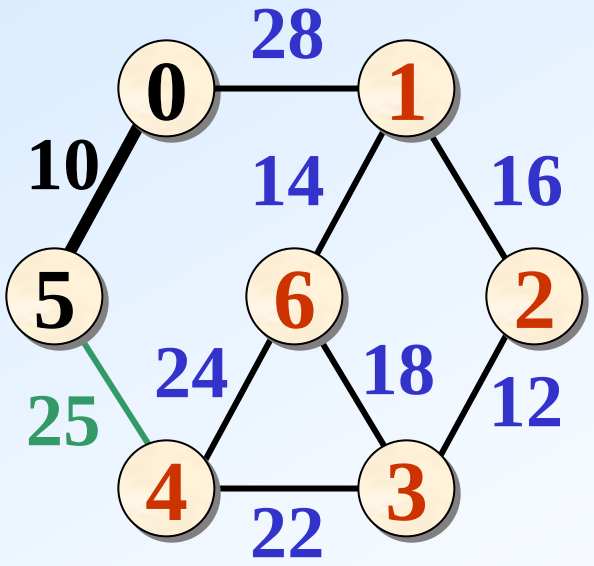




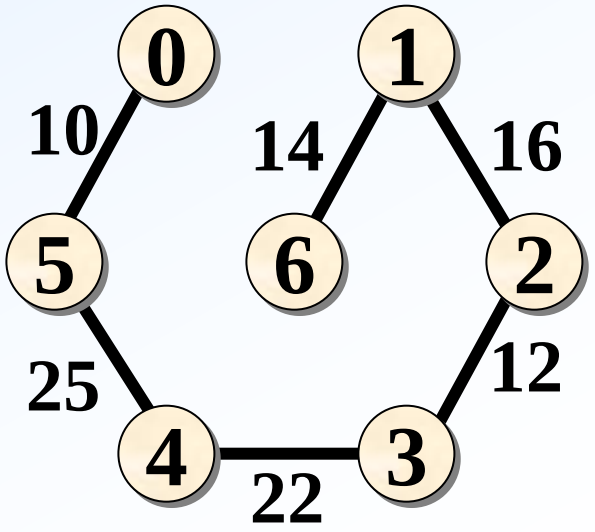
7.5 最小生成树—Prim算法实现过程

□顶点 v=6 加入生成树顶点集合：

	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
adjvex	−1	−1	−1	−1	−1	−1	-1



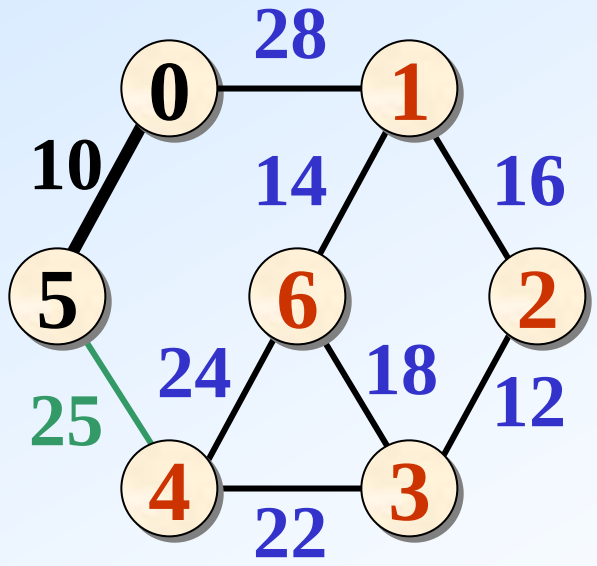
边 (1,6,14) 加入生成树



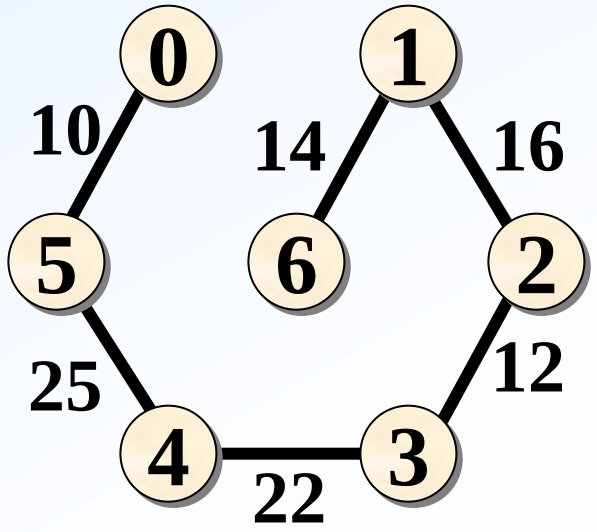
7.5 最小生成树—Prim算法实现过程

- 最后生成树中边集合里存入的各条边为：

(0, 5, 10), (5, 4, 25), (4, 3, 22),
(3, 2, 12), (2, 1, 16), (1, 6, 14)



	0	1	2	3	4	5	6
lowcost	0	16	12	22	25	10	14
adjvex	−1	−1	−1	−1	−1	−1	-1



7.5 利用Prim算法构造最小生成树

```
void Prim (Graph G, MST& T, int u) {
   float * lowcost = new float[G.n];
   int * adjvex = new int[G.n];
   for (int i = 0; i < G.n; i++) {
      lowcost[i] = G.edge[u][i];   // v_u 到各顶点代价
      adjvex[i] = u;               // 及最短带权路径
   }
   adjvex[u] = -1;                 // 加到生成树顶点集合
```

7.5 利用Prim算法构造最小生成树（续）

```
/* 查找最小lowcost */
for (i = 0; i < G.n && i != u; i++) {    // 循环n-1次，加入n-1条边
    EdgeData min = MaxValue;
    int v = 0;
/* 查找生成树外顶点到生成树内顶点具有最小权值的边，v是当前具最小权值的边 */
    for (int j = 0; j < G.n; j++)
        if (adjvex[j] != -1 && lowcost[j] < min) {  // ==-1不参选
            v = j;
            min = lowcost[j];
        } // end if
```

7.5 利用Prim算法构造最小生成树（续）

```
if (v != 0) {  // v=0表示再也找不到要求顶点，否则选边加入生成树
   printf ("(%d, %d, %d)", adjvex[v], v, lowcost[v]);
   adjvex[v] = -1;                          // 该边加入生成树标记
   for ( j = 0; j < G.n; j++ )              // 调整辅助数组
      if (adjvex[j] != -1 && G.edge[v][j] < lowcost[j]) {
         lowcost[j] = G.edge[v][j];  // 修改
         adjvex[j] = v;
      } // end if                                        (0, 5, 10)
   } // end if
  } // end for
} // end Prim
```

7.5 Prim算法分析

- 设连通网络有 n 个顶点，则该算法的时间复杂度为 $O(n^2)$，它适用于边稠密的网络；
- 注意
  - 当各边有相同权值时，由于选择的随意性，生成最小生成树可能不唯一；
  - 当各边的权值不相同时，产生的生成树是唯一的；
  - 未入选最小生成树的边权值有可能比最小生成树的边权值小。

7.6 活动网络

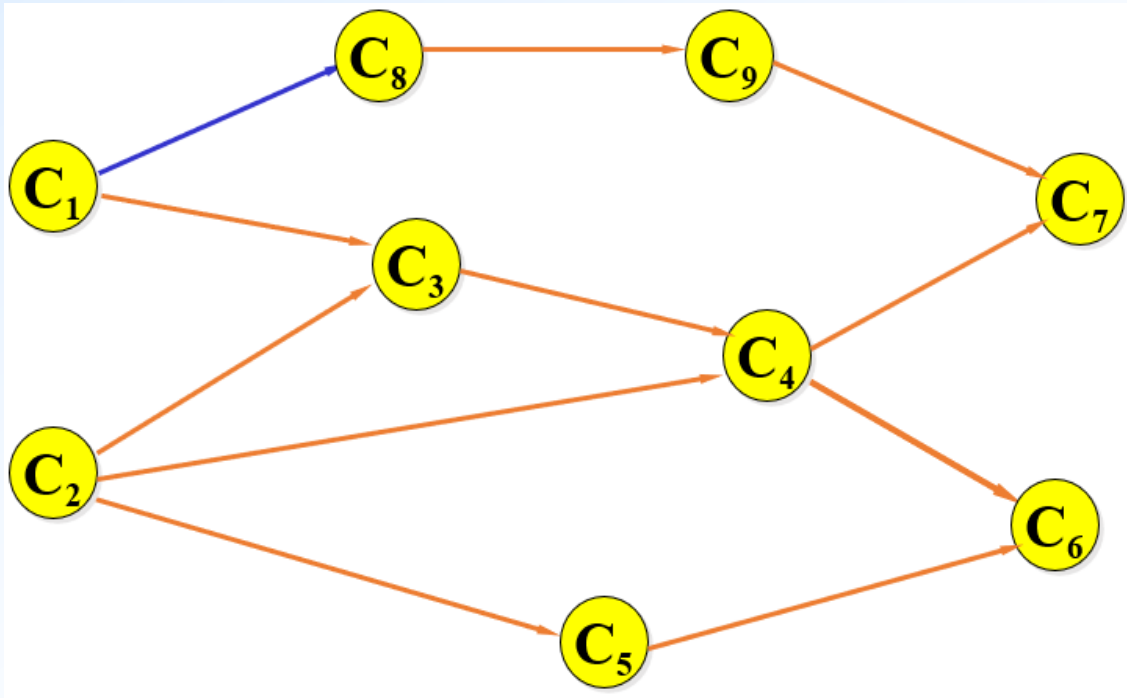
7.6.1 用顶点表示活动的网络 (AOV网络)

- 一般所有的工程可分为若干个叫做“活动”的子工程，而这些子工程之间通常存在约束，如某些子工程的开始必须在另一些子工程完成之后。
- 可以用有向图表示一个工程。在这种有向图中，用顶点表示活动，用有向边 $<V_i, V_j>$ 表示活动 V_i 必须先于活动 V_j 进行。这种有向图叫做顶点表示活动的AOV网络。

7.6.1 活动网络—AOV网络

- 例如，课程学习就是一个工程，每一门课程的学习就是一些活动，其中有些课程要求先修课程，有些则不要求。这样在有的课程之间存在前后关系，有的课程则可以并行地学习。

课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1，C_2
C_4	数据结构	C_3，C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5，C_4
C_7	操作系统	C_4，C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8



7.6.1 AOV网络—有向环判断

- 在AOV网络中**不能出现有向回路**，即有向环。否则，意味着某项活动应以自己作为先决条件这个荒谬的结论。
- 因此，**对给定的AOV网络，必须先判断它是否存在有向环。**
- 检测AOV网络无有向环的方法
  - 对AOV网络**构造它的拓扑有序序列**。即将各个顶点 (代表各个活动) 排列成一个线性有序的序列，使得AOV网络中**所有应存在的前驱和后继关系都能得到满足。**

7.6.1 AOV网络—拓扑排序

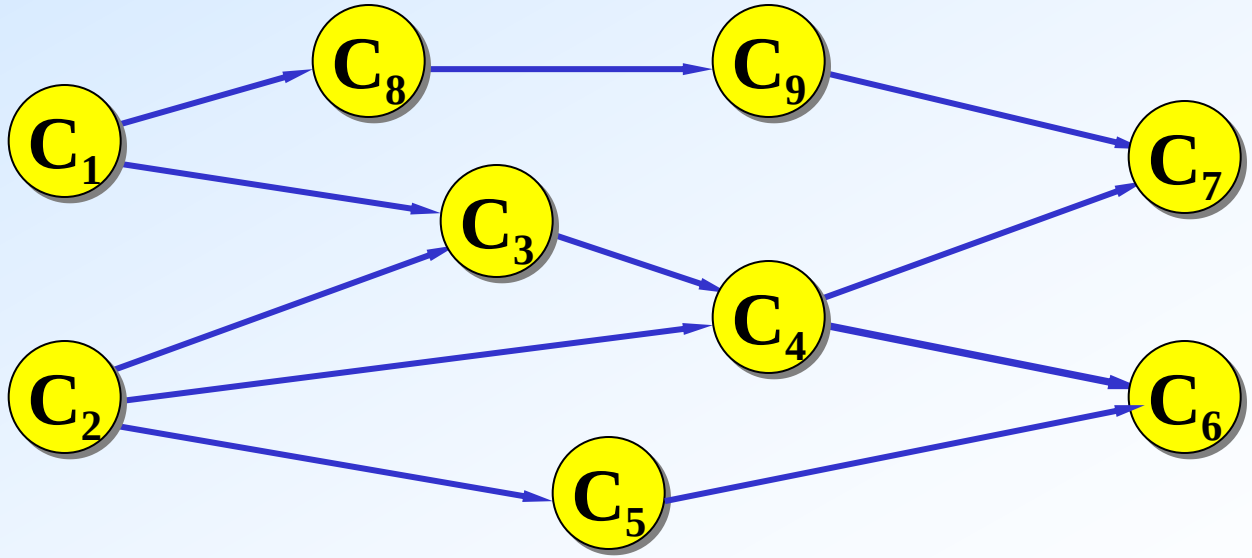
- 这种构造AOV网络全部顶点的拓扑有序序列的运算就叫做**拓扑排序**。
- **AOV网络无环条件**：如果通过拓扑排序能将AOV网络的所有顶点都排入一个拓扑有序的序列中，则该网络中必定不会出现有向环。
- 如果AOV网络中存在有向环，此AOV网络所代表的工程是不可行的。

7.6.1 AOV网络—拓扑排序

- 例如，对学生选课工程图进行拓扑排序，得到的拓扑有序序列为

$C_1, C_2, C_3, C_4, C_5, C_6, C_8, C_9, C_7$

或 $C_1, C_8, C_9, C_2, C_5, C_3, C_4, C_7, C_6$



7.6.1 AOV网络—拓扑排序方法

□拓扑排序的方法

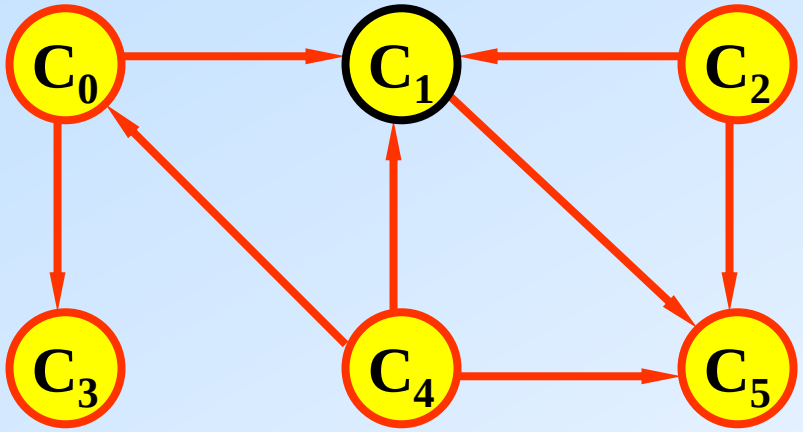
■ 输入 AOV 网络，令 n 为顶点个数。

1. 在 AOV 网络中选一个没有直接前驱的顶点，并输出之；
2. 从图中删去该顶点，同时删去所有它发出的有向边；
3. 重复以上 1、2步，直到
   - ➢ 全部顶点均已输出，拓扑排序完成，说明无有向环；

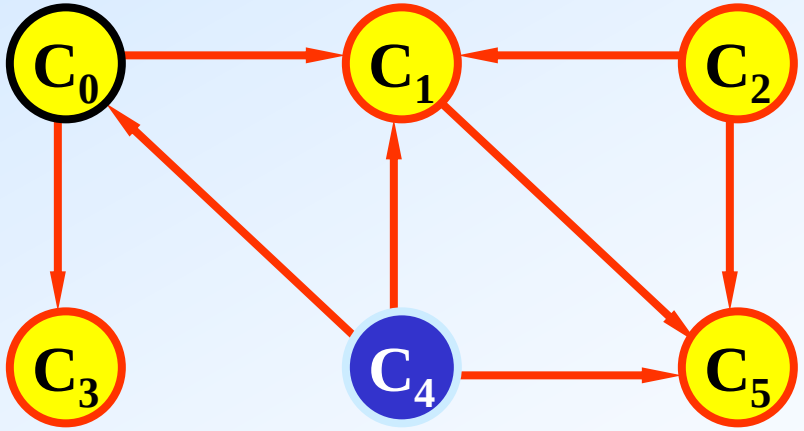
   或
   - ➢ 还有未输出的顶点，但已跳出处理循环。说明图中剩下的顶点都有直接前驱。这时网络中必存在有向环。

7.6.1 AOV网络—拓扑排序过程

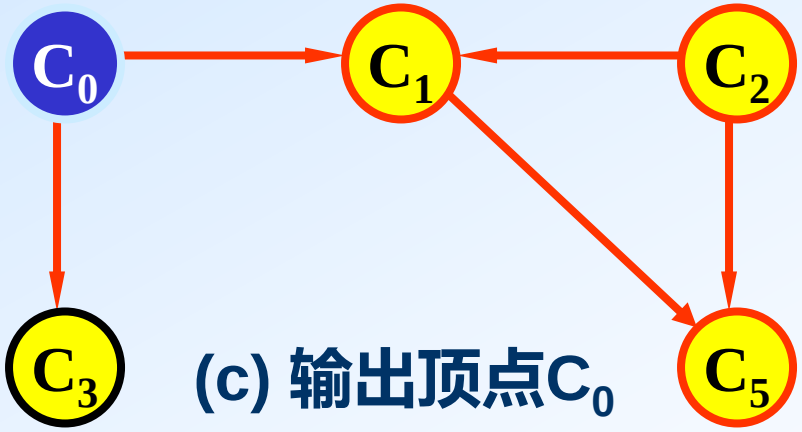
- 拓扑排序的过程



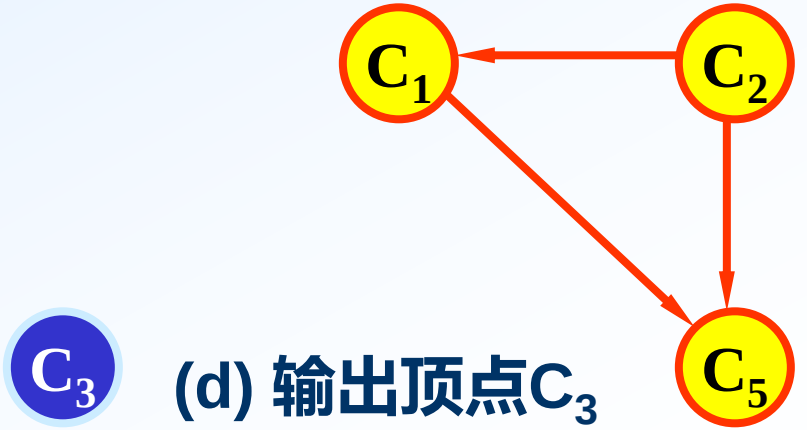
(a) 有向无环图



(b) 输出顶点C_4



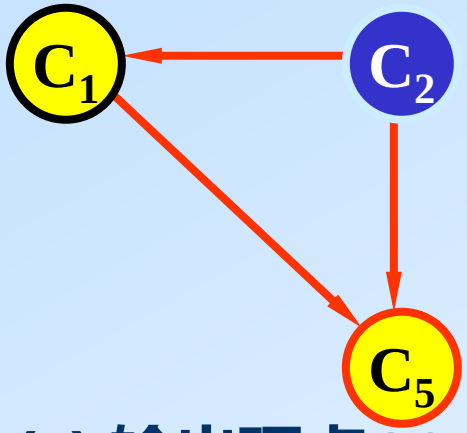
(c) 输出顶点C_0



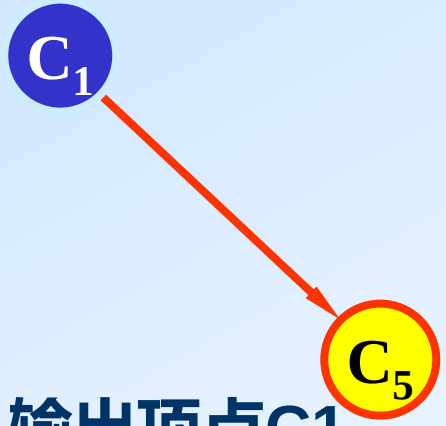
(d) 输出顶点C_3

7.6.1 AOV网络—拓扑排序过程

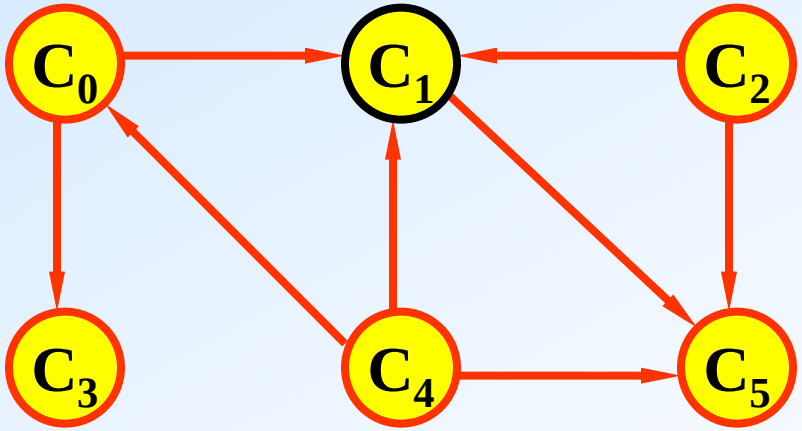
- 拓扑排序的过程（续）



(e) 输出顶点C2



(f) 输出顶点C1



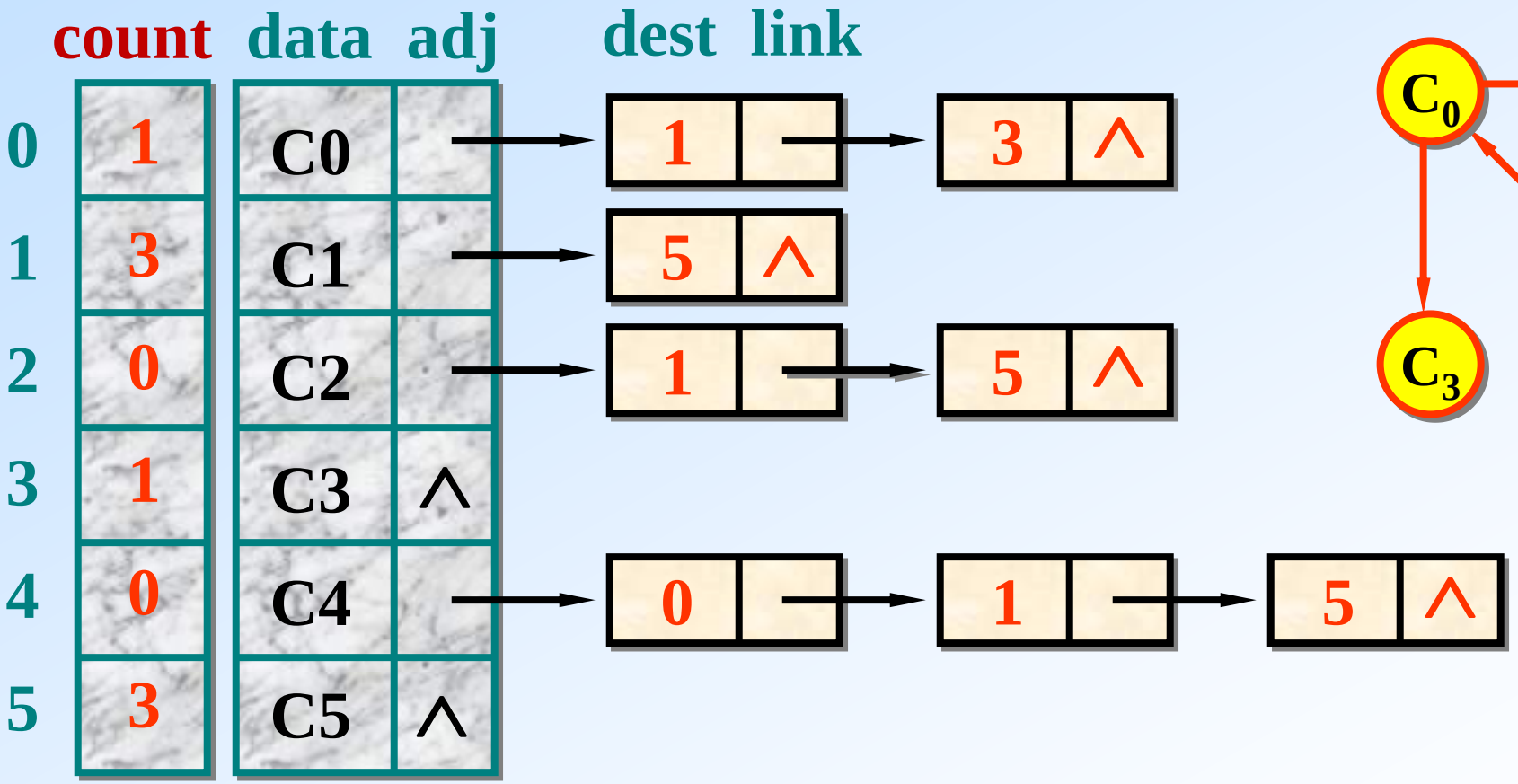
(g) 输出顶点C5

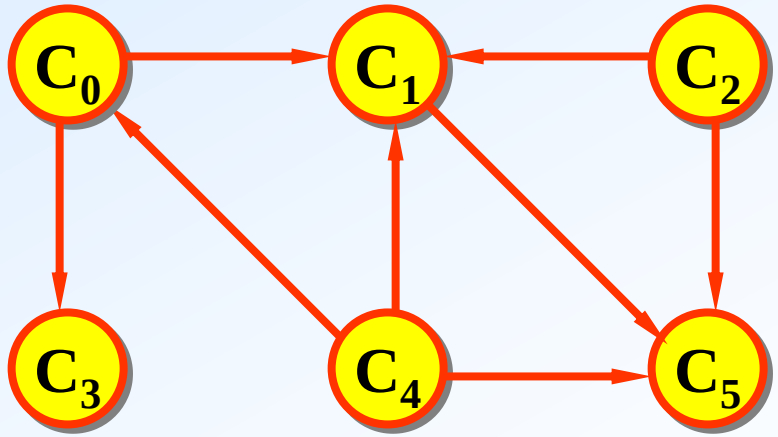
(h) 拓扑排序完成

- 最后得到的拓扑有序序列为 C_4，C_0，C_3，C_2，C_1，C_5。满足图中给出的所有前驱和后继关系；
- 对于本来没有这种关系的顶点，如C_4和C_2，也排出了先后次序关系。

7.6.1 AOV网络—邻接表表示

AOV网络及其邻接表表示





7.6.1 AOV网络—邻接表构建

- 在邻接表中增设一个数组count[ ]，记录各顶点入度。入度为零的顶点即无前驱顶点。
- 在输入数据前，顶点表 VexList[ ] 和入度数组 count[ ] 全部初始化。在输入数据时，每输入一条边<j, k>，就需要建立一个边结点，并将它链入相应边链表中，统计入度信息：

```
EdgeNode * p = new EdgeNode;
p->dest = k;        // 建立边结点
p->link = G.VexList[j].firstAdj;   // 链入顶点 j 的边链表的前端
VexList[j].firstAdj = p;
count[k]++;   // 顶点 k 入度加一
```

7.6.1 AOV网络—拓扑排序算法

- 在算法中，使用一个存放入度为零的顶点的**链式栈**，供选择和输出无前驱的顶点。
- 拓扑排序算法可描述如下：
  1. 建立入度为零的顶点栈；
  2. 当入度为零的顶点栈不空时，重复执行
     - 从顶点栈中退出一个顶点，并输出之；
     - 从AOV网络中删去这个顶点和它发出的边，边的终顶点入度减一；
     - 如果边的终顶点入度减至0，则该顶点进入度为零的顶点栈；
  3. 如果输出顶点个数少于AOV网络的顶点个数，则报告网络中存在有向环。

7.6.1 拓扑排序算法

```
void TopologicalSort (AdjGraph G) {
   Stack S;  StackEmpty (S); int j; // 入度为零的顶点栈初始化
   for (int i = 0; i < G.n; i++)    // 将入度为0的顶点进栈
      if (count[i] == 0)
         Push (S, i);
   for (i = 0; i < G.n; i++)        // 期望输出 n 个顶点
      if (StackEmpty (S))           // 中途栈空，退出
         return;                    // 网络中有回路，退出
      else {              // 继续拓扑排序
         Pop (S, j);      printf ("%d \n", j);  // 输出
```

7.6.1 拓扑排序算法

```
        /* 扫描出边表，更新与j相连接的顶点入度 */
        EdgeNode * p = VexList[j].firstAdj;
        while (p != NULL) {          // 扫描出边表
            int k = p->dest;         // 获得另一顶点
            if (--count[k] == 0)     // 顶点入度减一
                Push( S, k );        // 将入度为零的顶点进栈
            p = p->link;
        } // end while
    } // end else
}
```

7.6.2 活动网络—AOE网络

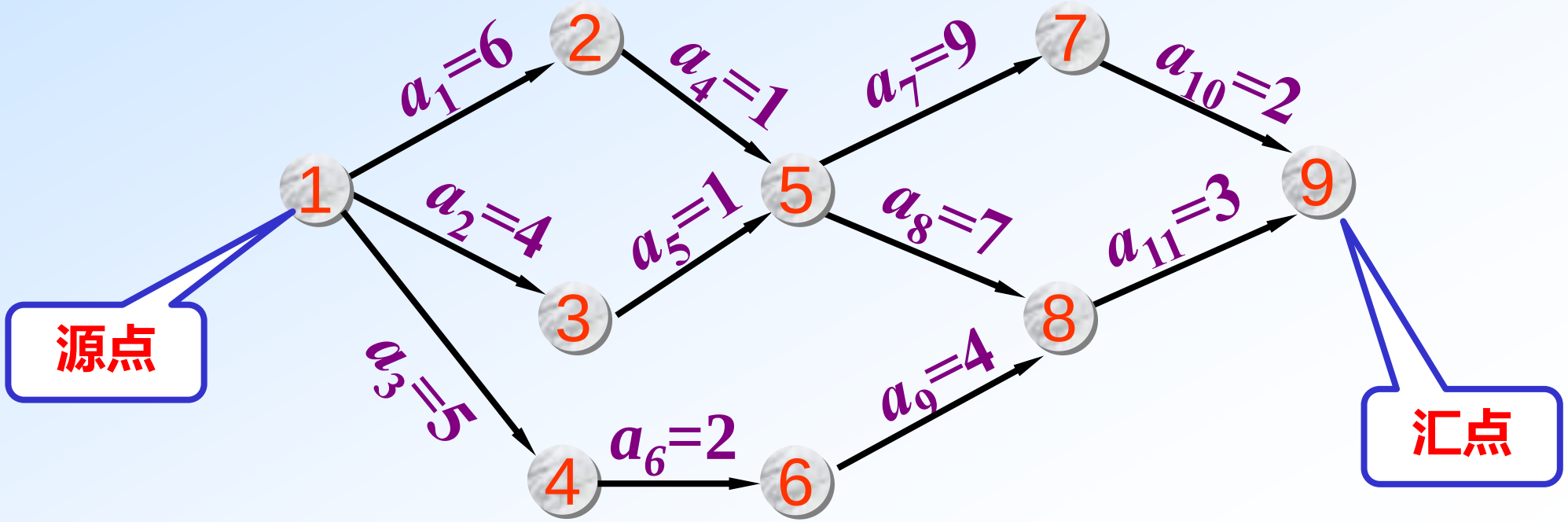
- 用边表示活动的网络(AOE网络)
- 如果在有向无环的带权图中，用有向边表示一个工程中的活动，用边上权值表示活动持续时间，用顶点表示事件，则这样的有向图叫做用边表示活动的网络，简称 AOE 网络。
- AOE网络在某些工程估算方面非常有用。例如，可以使人们了解：
  - 完成整个工程至少需要多少时间(假设网络中没有环)?
  - 为缩短完成工程所需的时间，应当加快哪些活动?

7.6.2 活动网络—AOE网络

- 从源点到各个顶点，以及从源点到汇点的有向路径可能不止一条。这些路径的长度也可能不同。完成不同路径的活动所需的时间虽然不同，但**只有各条路径上所有活动都完成了，整个工程才算完成**。
- 因此，**完成整个工程所需的时间取决于从源点到汇点的最长路径长度**，即在这条路径上所有活动的持续时间之和。这条路径长度**最长**的路径就叫做**关键路径**。

7.6.2 活动网络—AOE网络

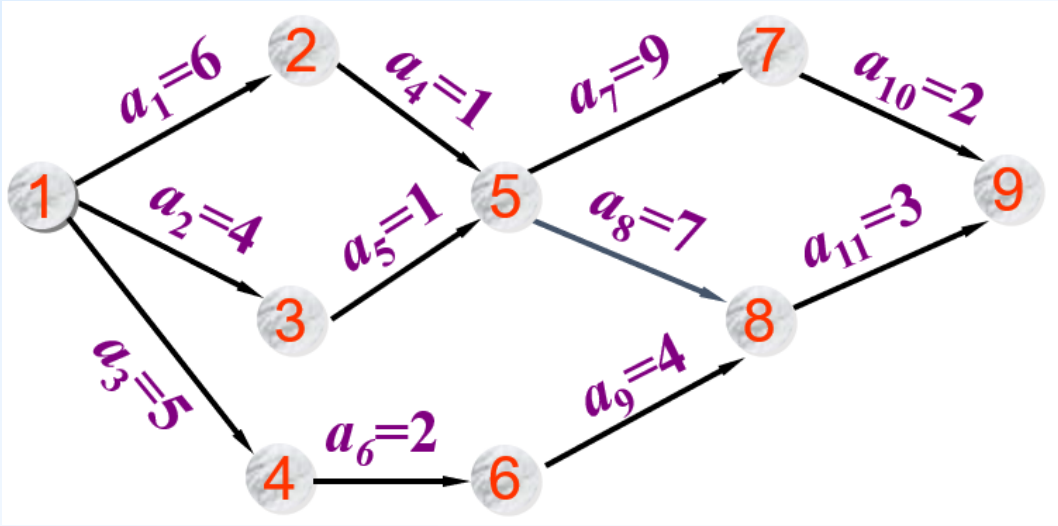
- 要找出关键路径，必须找出**关键活动**，即不按期完成就会影响整个工程完成的活动。
- 关键路径上的所有活动都是关键活动。因此，只要找到了关键活动，就可以找到关键路径。



7.6.2 活动网络—AOE网络

- 假设从源点 v_0 至汇点 v_{n-1} 共 n 个节点。

1. 事件 v_j 的**最早可能开始时间 ve[j]** 是从源点 v_0 到顶点 v_j 的最长路径长度。
2. 事件 v_j 的**最迟允许开始时间 vl[j]** 是在保证汇点 v_{n-1} 在 ve[n-1] 时刻完成的前提下，事件 v_j 的允许的最迟开始时间。



7.6.2 活动网络—AOE网络

3. 活动 a_k 的**最早可能开始时间 e[k]**

■设活动 a_k 在边 $<v_i, v_j>$ 上，则 e[k] 是从源点 v_0 到顶点 v_i 的最长路径长度。因此，

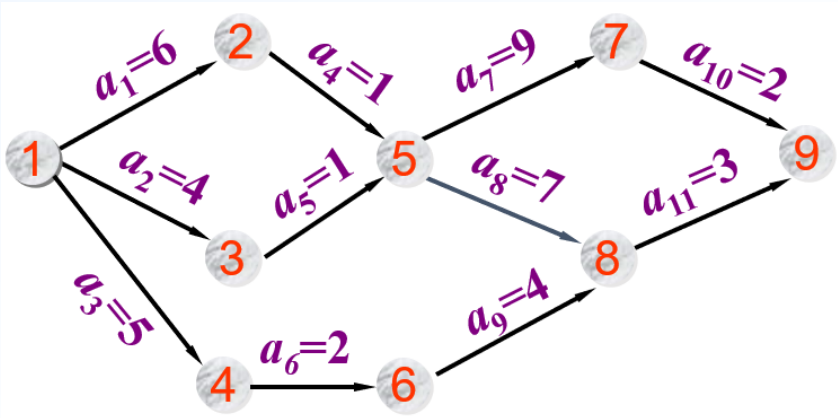
$$e[k] = ve[i]$$

4. 活动 a_k 的**最迟允许开始时间 l[k]**

■l[k] 是在不会引起时间延误的前提下，该活动允许的最迟开始时间

$$l[k] = vl[j] - dur(<i, j>)$$

■其中，dur(<i, j>)是完成 a_k 所需的时间。

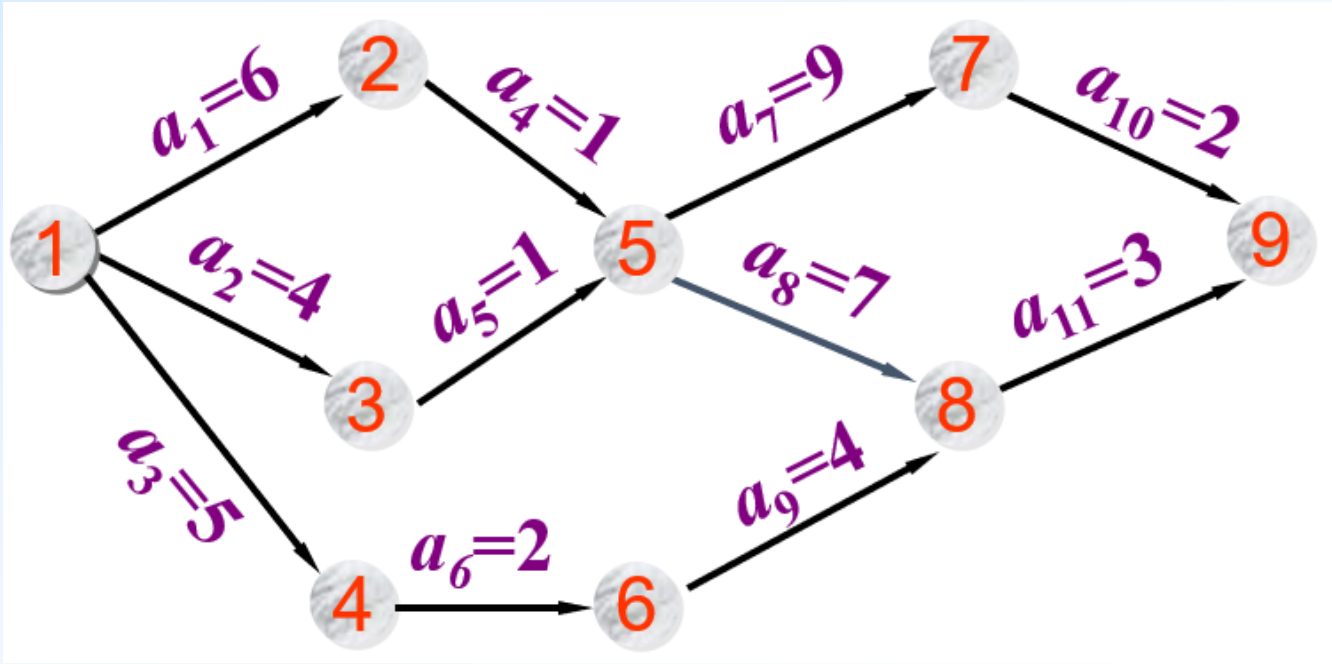


7.6.2 活动网络—AOE网络

5. 时间余量 l[k] - e[k]

- 表示活动 a_k 的最早可能开始时间和最迟允许开始时间的时间余量。l[k] == e[k] 表示活动 a_k 是没有时间余量的**关键活动**。

- 为找出关键活动，需要求各个活动的 e[k] 与 l[k]，以判别是否 l[k] == e[k]。



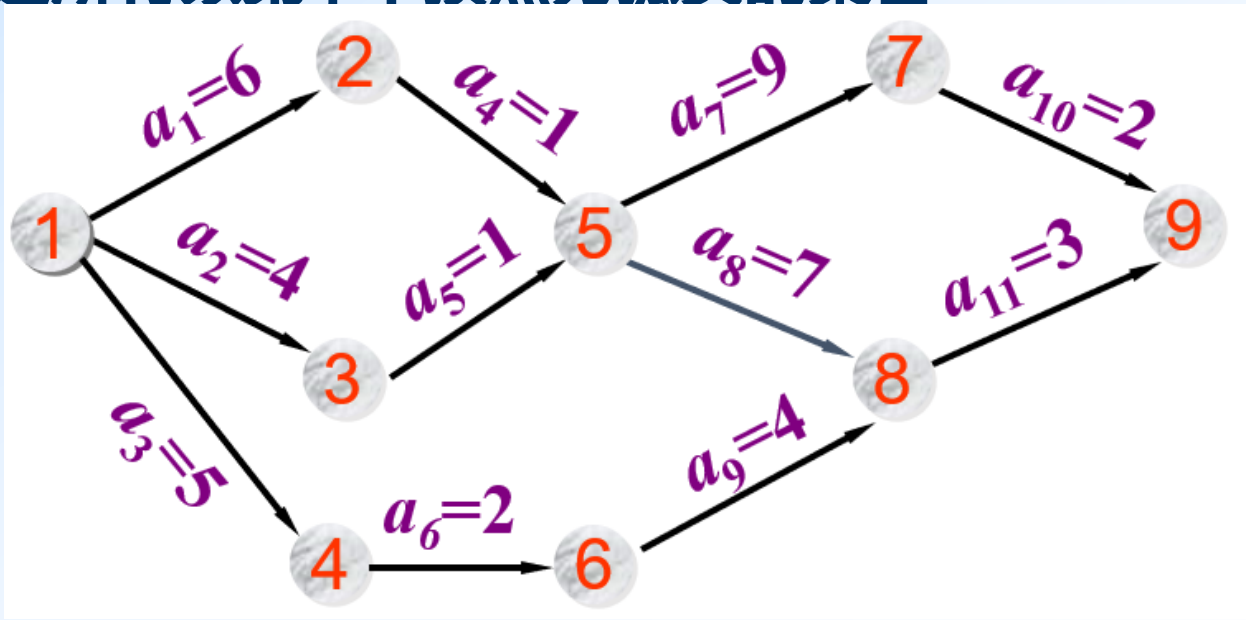
7.6.2 活动网络—AOE网络

关键路径求解方法

1. 先求从源点 v_0 到各个顶点 v_i 的 ve[j]。
   - 从源点 ve[0] = 0 开始，向前递推

$$ve[j] = \max_i\{ve[i] + dur(\langle i,j\rangle)\} \quad \langle i,j\rangle \in T, j = 1,2,\dots,n-1$$

   - 其中，T 是所有以第 i 个顶点为弧头的集合



7.6.2 活动网络—AOE网络

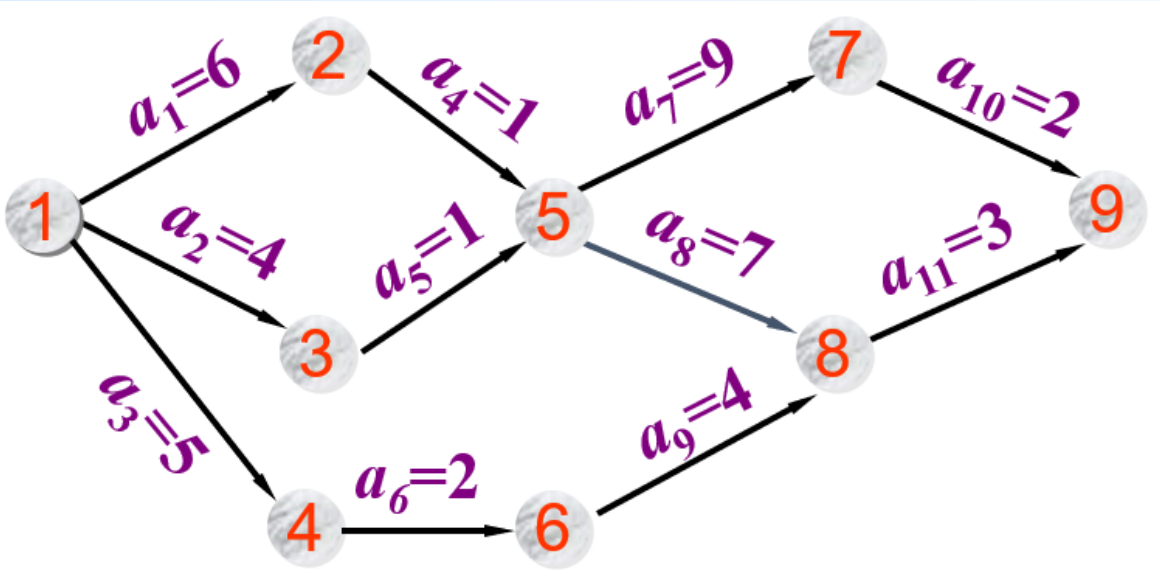
□关键路径求解方法

2. 求从汇点 v_{n-1} 到各个顶点 v_i 的 vl[i]。

➢从汇点 vl[n-1] = ve[n-1] 开始，向后递推

$$vl[i] = \min_j\{vl[j] - dur(\langle i,j\rangle)\} \quad \langle i,j\rangle \in S, i = n-2, \ldots, 0$$

➢其中，S 是所有以第 i 个顶点为弧尾的集合



7.6.2 AOE网络—关键路径的求法

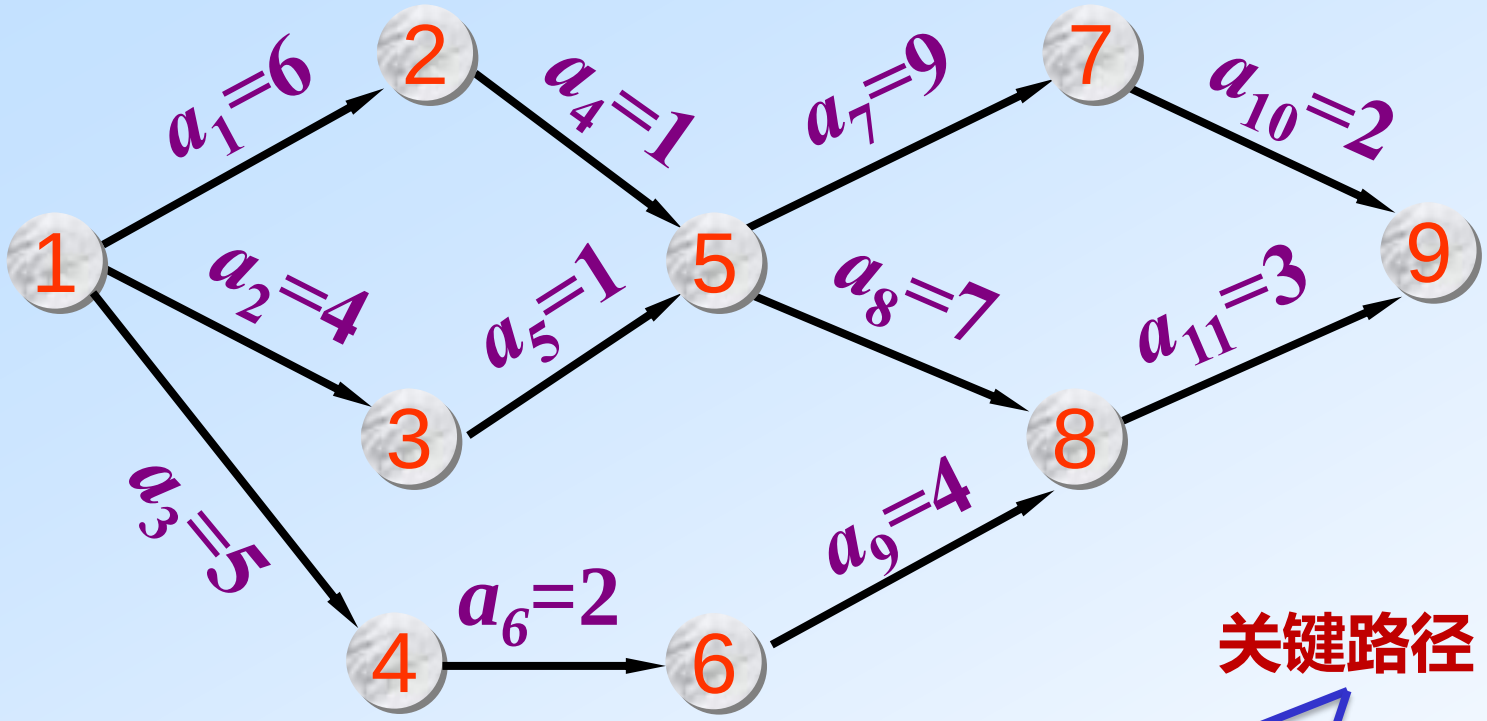
- 上述两个递推公式的计算必须分别在拓扑有序及逆拓扑有序的前提下进行。

3. 求活动 a_k (k=1,2,…,e) 的 e[k] 和 l[k]

$$e[k] = ve[i]$$

$$l[k] = vl[j] - dur(<i, j>)$$

4. 判断关键路径
   - 若e[k]=l[k]，则活动k为关键活动。



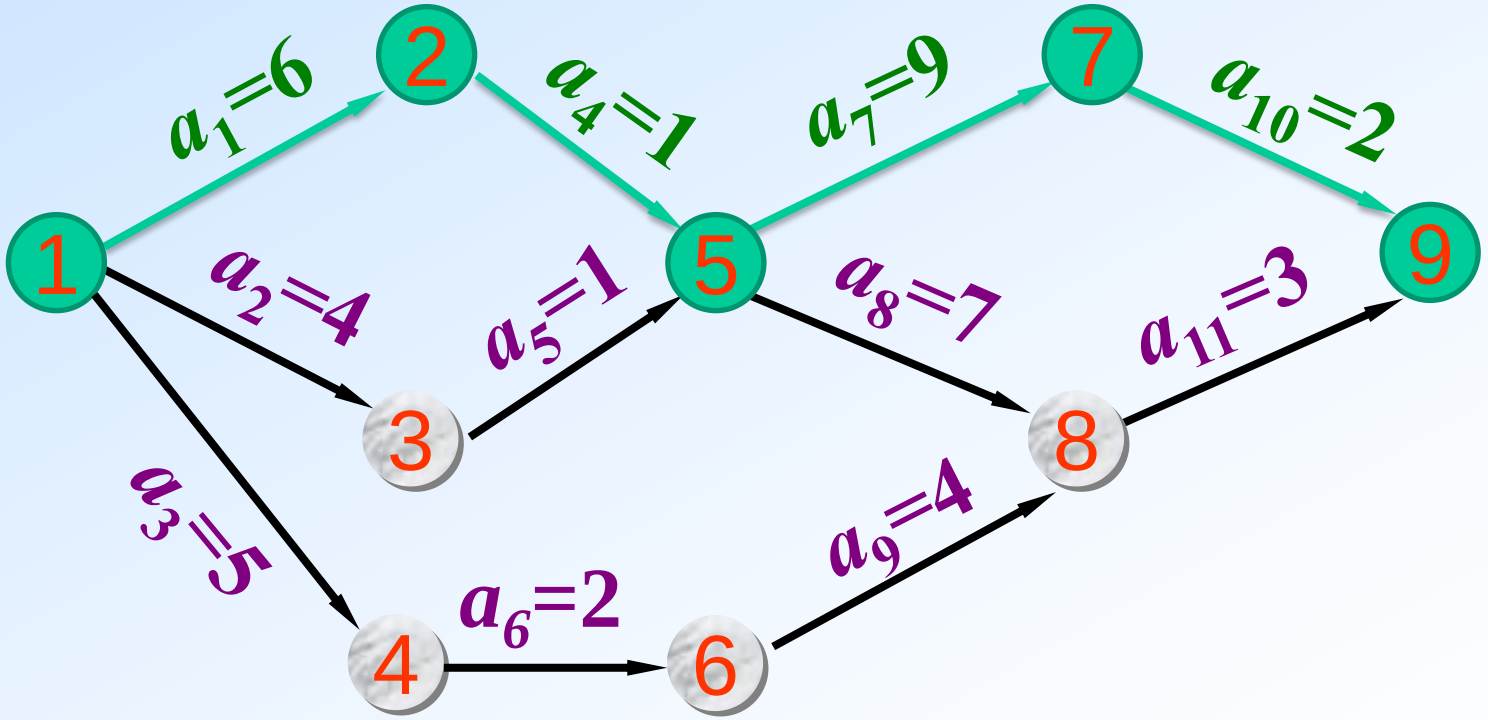
关键路径

	1	2	3	4	5	6	7	8	9
ve	0	6	4	5	7	7	16	14	18
vl	0	6	6	9	7	11	16	15	18

	1	2	3	4	5	6	7	8	9	10	11
e	0	0	0	6	4	5	7	7	7	16	14
l	0	2	4	6	6	9	7	8	11	16	15

7.6.2 AOE网络—关键路径

- 关键活动：a_1, a_4, a_7, a_{10}
- 关键路径：(1, 2, 5, 7, 9)



7.6.2 AOE网络—关键路径算法

```
void CriticalPath (AdjGraph G) {
   int i, k;  int e, l;  int ve[G.n]; int vl[G.n];
   for (i = 0; i < G.n; i++) ve[i] = 0;        // 初始化
   for (i = 0; i < G.n; i++) {                 // 顺向计算事件最早允许开始时间
      EdgeNode *p = G.VexList[i].adj;          // 边链表链头指针p
      while (p != NULL) {                      // 找所有后继邻接顶点
         k = p→dest;                           // i 的后继邻接顶点k
         if (ve[i] + p→cost > ve[k]) ve[k] = ve[i] + p→cost;
         p = p→link;                           // 找下一个后继
      } // end while
   }    // end for
```

7.6.2 AOE网络—关键路径算法

```
// 逆向计算事件的最迟开始时间
for (i = 0; i < G.n; i++) vl[i] = ve[G.n-1];   // 初始化
for (i = G.n-2; i > 0; i--) {                  // 逆向计算事件的最迟开始时间
    p = G.VexList[i].adj;                      // 该顶点边链表链头指针p
    while (p != NULL) {
        k = p→dest;                            // i的后继邻接顶点k
        if ( vl[k] - p→cost < vl[i])
            vl[i] = vl[k] - p→cost;
        p = p→link;                            // 找下一个后继
    } // end while
}     // end for
```

7.6.2 AOE网络—关键路径算法

```
// 逐个顶点求各活动的e[k]和l[k]
    for ( i = 0; i < G.n; i++ ) {
        p = G.VexList[i].adj;     // 该顶点边链表链头指针p
        while ( p != NULL ) {
            k = p→dest;           // k是i的后继邻接顶点
            e = ve[i];    l = vl[k] - p→cost;
            if (l == e) printf ("< %d , %d >\n",  i, k);      // 输出关键活动
            p = p→link;           // 找下一个后继
        } // end while
    } // end for
}
```

7.6.2 AOE网络—关键路径算法分析

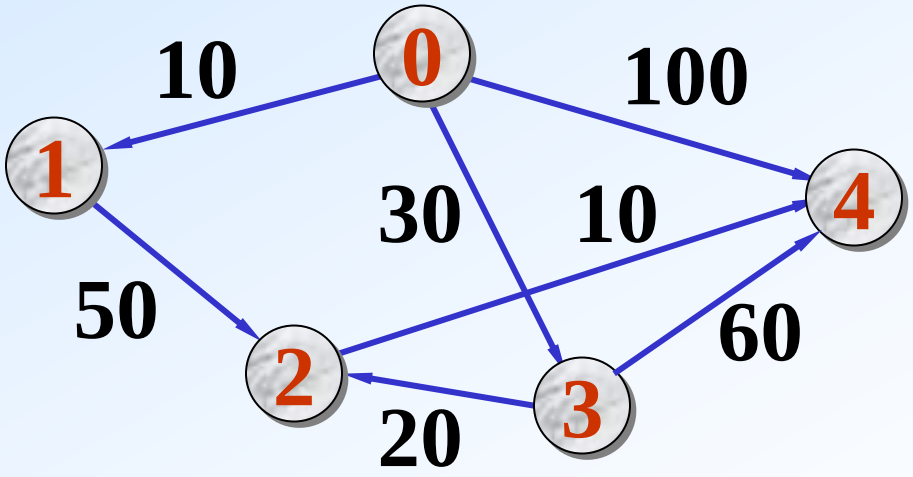
- 时间复杂度：在拓扑排序求 ve[i] 和逆拓扑有序求 vl[i] 时，所需为O(n+e)，求各个活动的 e[k] 和 l[k] 时所需时间为O(e)，总共花费时间仍然是 O(n+e)；
- 存在多条关键路径的图，任一关键活动加速不能使整个工程提前；想使整个工程提前，要提高各个关键路径上所有关键活动。

7.7 最短路径

- **最短路径问题：如果从图中某一顶点(称为源点)到达另一顶点(称为终点)的路径可能不止一条，如何找到一条路径使得沿此路径上各边上的权值总和达到最小。**
- **问题解法**
  - **边上权值非负情形的单源最短路径问题**
    - **Dijkstra算法**
  - **边上权值为任意值的单源最短路径问题**
    - **Bellman和Ford算法**
  - **所有顶点之间的最短路径**
    - **Floyd算法**

7.7 最短路径

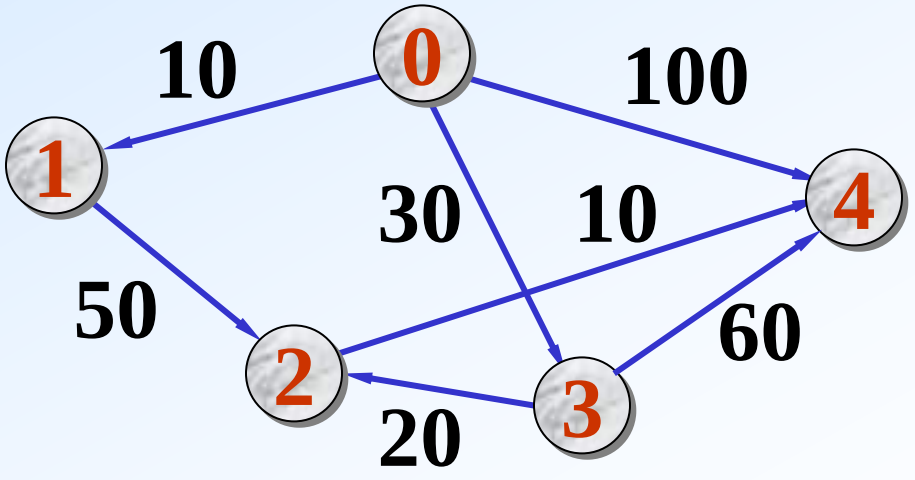
- 边上权值非负情形的单源最短路径问题
- 问题的提法： 给定一个带权有向图 G 与源点 v，求从 v 到 G 中其它顶点的最短路径。限定各边上的权值大于或等于 0。



7.7 最短路径—Dijkstra算法

- Dijkstra算法：**按路径长度的递增次序，逐步产生最短路径。**
  1. 首先求出长度最短的一条最短路径；
  2. 再参照它求出长度次短的一条最短路径；

  依次类推……
  3. 直到从顶点 v 到其它各顶点的最短路径全部求出为止。



7.7 最短路径—Dijkstra算法基本思想

1. 引入辅助数组 dist[]。它的每一个分量 dist[i] 表示当前找到的从源点 v_0 到终点 v_i 的最短路径的长度。
2. 初始状态：
   - 若从源点 v_0 到顶点 v_i 有边，则dist[i]为该边上的权值；
   - 若从源点 v_0 到顶点 v_i 无边，则dist[i]为∞ 。
3. 长度为$\mathrm{dist}[j] = \min_i\{\mathrm{dist}[i] | v_i \in V\}$的路径是从源点 v_0 出发的**长度最短的最短路径**，其路径为 (v_0, v_j) 的最短路径的长度。

7.7 最短路径—Dijkstra算法基本思想

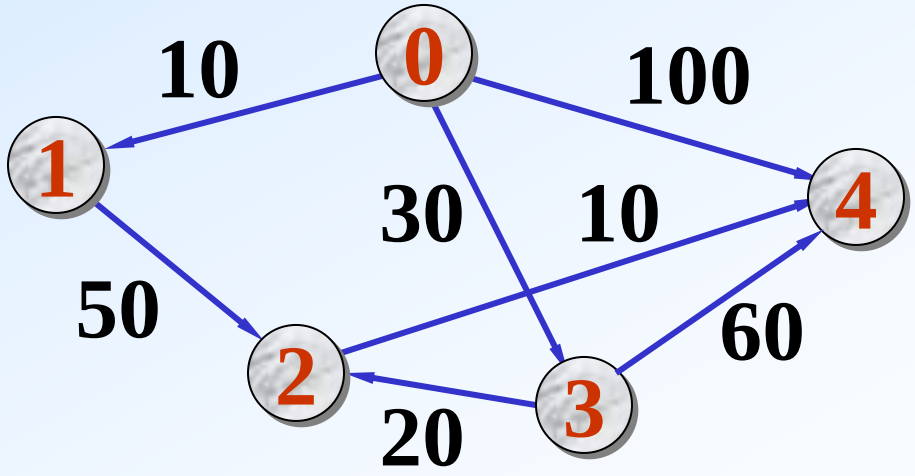
4. 求解次短路径
   - 假设次短路径终点为v_k，则这条最短路径或为(v_0, v_k)，或为(v_0, v_j, v_k)
   - 次短路径长度为 edge[0][k] 或 dist[j] + edge[j][k]

- 假设 S 是已求得的最短路径的终点的集合，则可证明：
  - 下一条最短路径必然是从 v_0 出发，中间只经过 S 中的顶点便可到达的那些顶点 v_x ($v_x \in V - S$) 的路径。

5. 每次求得一条最短路径后，其终点 v_k 加入集合 S，然后对所有的 $v_i \in V - S$，修改其 dist[i] 值。

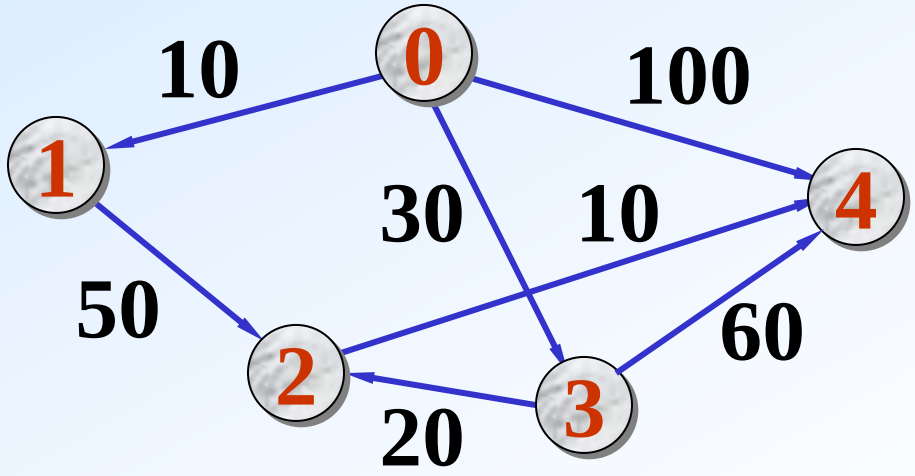
7.7 最短路径—Dijkstra算法求解过程(1/5)

	0	1	2	3	4
dist	0	10	∞	30	100
path	-	(0,1)	-	(0,3)	(0,4)
S	√				



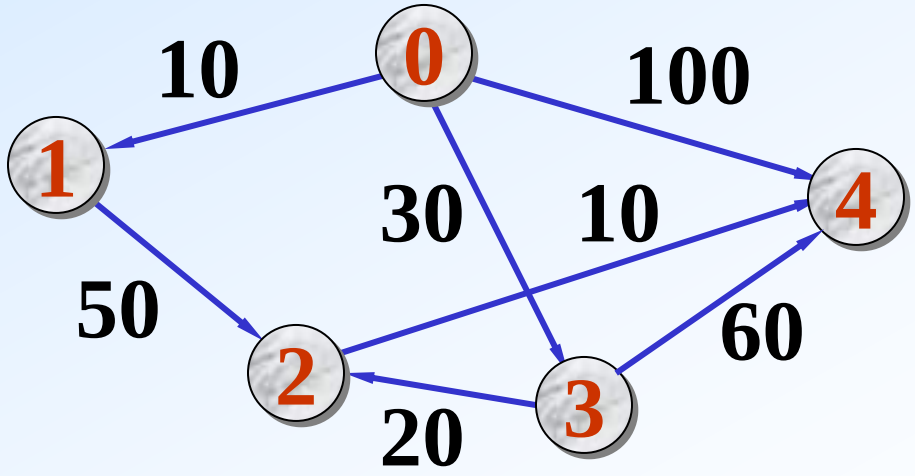
7.7 最短路径—Dijkstra算法求解过程(2/5)

	0	1	2	3	4
dist	0	**10**	60	30	100
path	-	**(0,1)**	(0,1,2)	(0,3)	(0,4)
S	√	√			



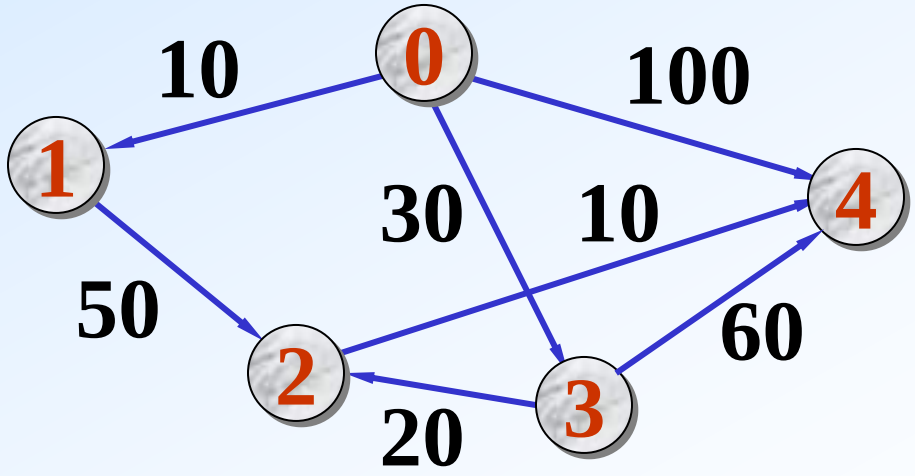
7.7 最短路径—Dijkstra算法求解过程(3/5)

	0	1	2	3	4
dist	0	10	50	**30**	90
path	-	(0,1)	(0,3,2)	**(0,3)**	(0,3,4)
S	√	√		√	



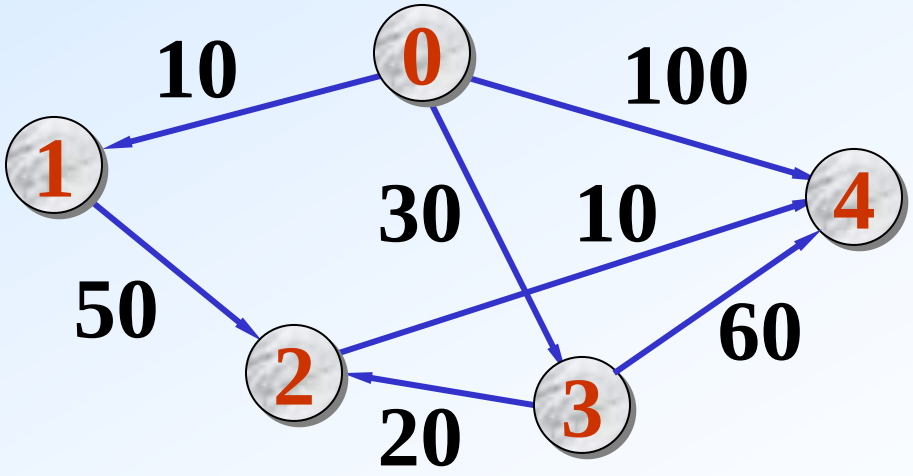
7.7 最短路径—Dijkstra算法求解过程(4/5)

	0	1	2	3	4
dist	0	10	**50**	30	60
path	-	(0,1)	**(0,3,2)**	(0,3)	(0,3,2,4)
S	√	√	**√**	√	



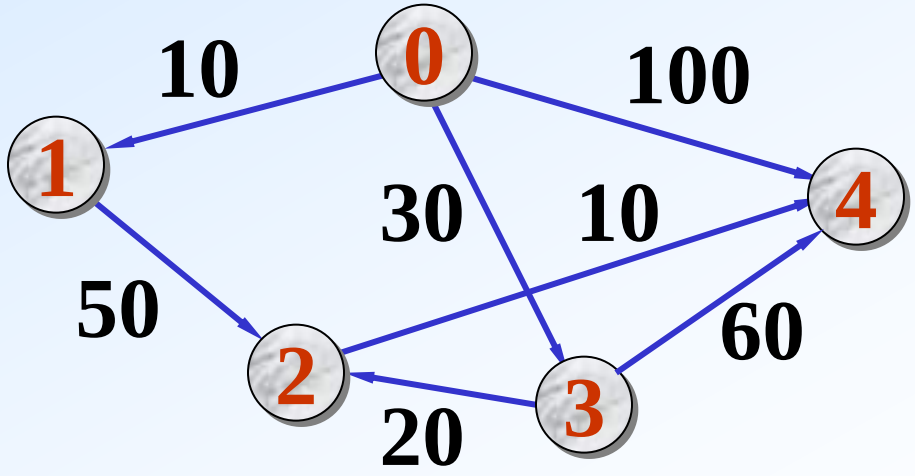
7.7 最短路径—Dijkstra算法求解过程(5/5)

	0	1	2	3	4
dist	0	10	50	30	**60**
path	-	(0,1)	(0,3,2)	(0,3)	**(0,3,2,4)**
S	√	√	√	√	**√**



7.7 最短路径—Dijkstra算法求解过程(5/5)

	0	1	2	3	4
dist	0	10	50	30	60
path	-	(0,1)	(0,3,2)	(0,3)	(0,3,2,4)
	-	**0**	**3**	**0**	**2**
S	√	√	√	√	√



7.7 最短路径—Dijkstra算法

- 采用邻接矩阵存储方式

1. 初始化：
   - $S \leftarrow \{v_0\}$；$dist[j] \leftarrow edge[0][j]$, $j = 1, 2, \ldots, n-1$;
2. 求出最短路径的长度：
   - $dist[k] \leftarrow \min\{dist[i]\}$, $i \in V- S$；$S \leftarrow S \cup \{k\}$;
3. 调整：
   - $dist[i] \leftarrow \min\{dist[i], dist[k] + edge[k][i]\}$, $i \in V- S$;
4. 判断：
   - 若 $S = V$，则算法结束，否则转第 2 步。

7.7 最短路径—Dijkstra算法

```
void ShortestPath (MTGraph G, int v) {
    EdgeData dist[G.n];        // 最短路径长度数组
    int path[G.n];             // 最短路径数组
    int S[G.n];                // 最短路径顶点集
    for (int i = 0; i < n; i++) {
        dist[i] = G.edge[v][i];   // dist数组初始化
        S[i] = 0;                 // 集合S初始化
        if (dist[i] < MaxValue)
            path[i] = v;
        else path[i] = -1;        // path数组初始化
    }
```

7.7 最短路径—Dijkstra算法

```
S[v] = 1;                          // 顶点v加入顶点集合
// 从顶点 v 确定 n-1 条路径
for (i = 0; i < G.n - 1; i++) {
    float min = MaxValue;
    int u = v;
    // 选当前不在集合 S 中具有最短路径的顶点 u
    for (int j = 0; j < G.n; j++)
        if (!S[j] && dist[j] < min) {
            u = j;      min = dist[j];
        }
```

7.7 最短路径—Dijkstra算法

```
S[u] = 1;                          // 将顶点 u 加入集合 S
// 修改可经过顶点 u 变短的路径值
for (int w = 0; w < G.n; w++)
   if (!S[w] && G.edge[u][w] < MaxValue
   && dist[u] + G.edge[u][w] < dist[w] ) {
      // 顶点 w 未加入 S，且经过 u 可以缩短路径值
      dist[w] = dist[u] + G.edge[u][w];
      path[w] = u;     // 修改到 w 的最短路径
   }
}     // 选定各顶点到顶点 v 的最短路径
```

7.7 最短路径—Dijkstra算法

```
for ( i = 0; i < G.n; i++ ) {   // 打印各顶点的最短路径: 路径为逆向输出
    printf ("\n");    printf ("Distance: %d; Path: %d", dist[i], i);
    // 输出终点的最短路径长度和终点
    int pre = path[i];    // 取终点的直接前驱
    while ( pre != v ) {    // 沿路径上溯输出
        printf ("<-- %d ", pre);
        if (pre == -1) break;    // 无法从 v 到达该结点
        pre = path[pre];
    } // end while
} // end for
} // end ShortestPath
```