§7.图

7.1.图的定义和术语

7.1.1.图的基本特性

元素间存在多对多的关系（网状结构）

★ 每个结点都可能有任意前驱和任意后继

★ 结点间不存在顺序关系

★ 结点的邻接点间也不存在顺序关系

7.1.2.图的形式定义(P.156-157)

Graph = (V, R);

$V = \{x \mid x \in D_0\}$

$R = \{VR\}$

$VR = \{\langle v,w\rangle \mid v,w \in V$ 且 $P(v,w)\}$

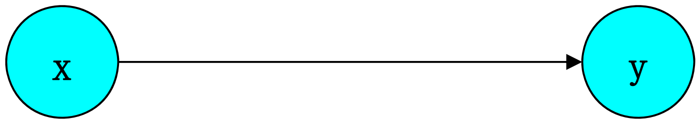
谓词P(v,w)定义了<v,w>的意义或信息

§7. 图

7.1. 图的定义和术语

7.1.3. 图的基本术语

★ 顶点：代表数据元素

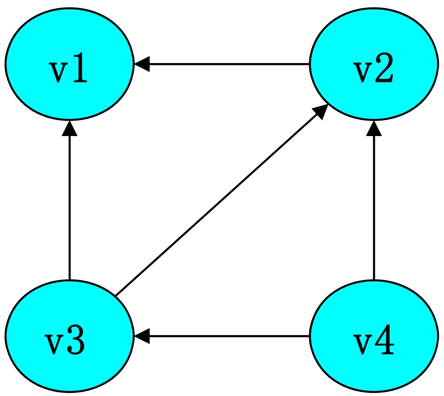


★ 弧：两个顶点间的一条有向边<x, y>

★ 弧头：有向边的终端点(y)

★ 弧尾：有向边的初始点(x)

★ 有向图：由有向边和顶点组成的图



左图的形式化表示：

G = (V, R)

V = {v1, v2, v3, v4}

R = {VR}

VR= { <v2, v1>, <v3, v1>, <v3, v2>, <v4, v2>, <v4, v3> }

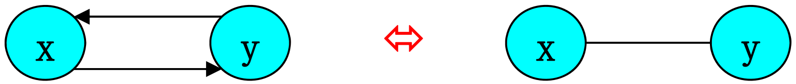
谓词P(v2, v1)表示从v2到v1的一条单向通路

§7. 图

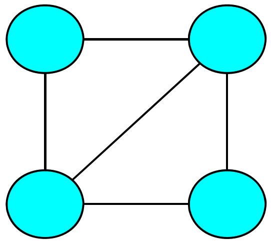
7.1. 图的定义和术语

7.1.3. 图的基本术语

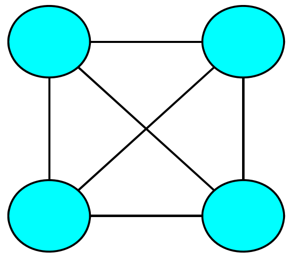
★ 边：若两个顶点之间存在两条相对的弧，即有<x, y>必有<y, x>，则表示一条无向边，记为(x, y)



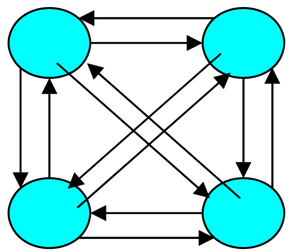
★ 无向图：由边和顶点组成的图



★ 完全图：在无向图中，如果任意两个顶点之间都有且仅有一条边相连，n个顶点有 $\frac{n(n-1)}{2}$ 条边，则称为完全图



★ 完全有向图：在有向图中，如果任意两个顶点之间都有且仅有相对的两条弧相连，n个顶点有n(n-1)条弧，则称为完全有向图



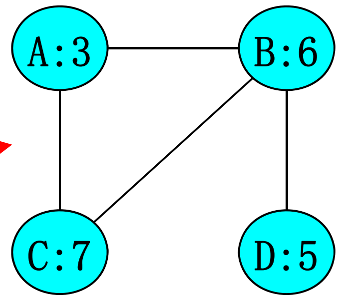
§7.图

7.1.图的定义和术语

7.1.3.图的基本术语

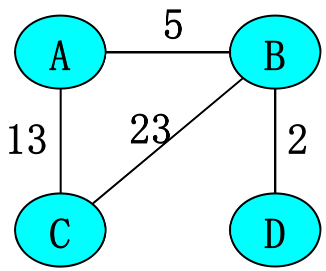
★ 稀疏图：若图G中的边/弧的数量 e<nlogn(n为顶点) 则称该图为稀疏图

★ 稠密图：反之则称为稠密图



★ 权：顶点/边相关的对应值

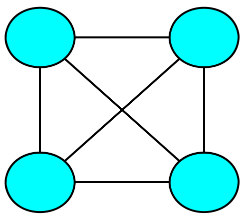
★ 网：带权的图 ⎧顶点带权
⎩边带权



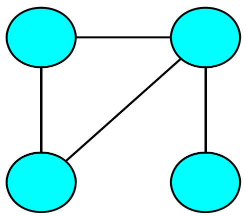
A:3 – A城市有3所高校/银行...

AC=13：两个城市间的路费/时间/距离...

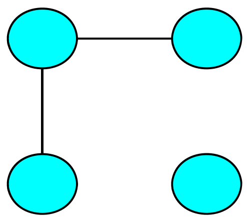
★ 子图：假设G =(V，R)，G’=(V’,R’)，若V’⊆V，R’⊆R，则称G’为G的一个子图



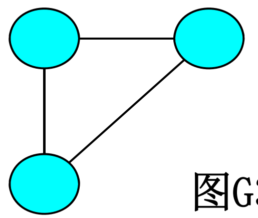
图G



图G1
是G的子图



图G2
是G的子图
是G1的子图



图G3
是G的子图
是G1的子图
不是G2的子图

§7. 图

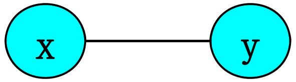
7.1. 图的定义和术语

7.1.3. 图的基本术语

无向图

★ 邻接点：若两顶点间有边,则称两顶点互为邻接点

★ 依附(相关联)：若两顶点间存在边，则称边依赖于顶点或边与顶点相关联



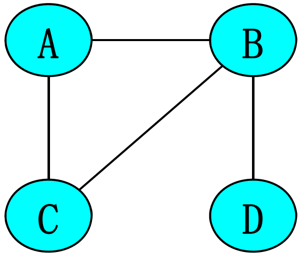
★ 顶点的度：和顶点相关联的边的数目

A的度：2

B的度：3

C的度：2

D的度：1



● 所有顶点度之和等于边数*2

§7. 图

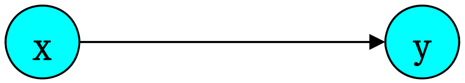
7.1. 图的定义和术语

7.1.3. 图的基本术语

有向图

★ 邻接到：若两顶点间有弧，称初始点邻接到终端点(尾邻接到头)

★ 邻接自：若两顶点间有弧，称终端点邻接自初始点(头邻接自尾)



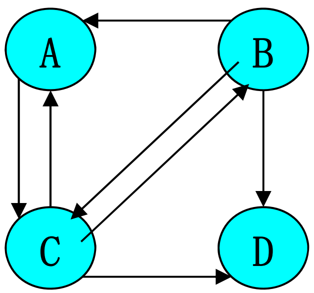
x邻接到y
y邻接自x

★ 顶点的入度：以该顶点为头的弧的数目(终端点)

★ 顶点的出度：以该顶点为尾的弧的数目(初始点)

★ 顶点的度：以该顶点为头/尾的弧的总数

	出度	入度	度
A	1	2	3
B	3	1	4
C	3	2	5
D	0	2	2

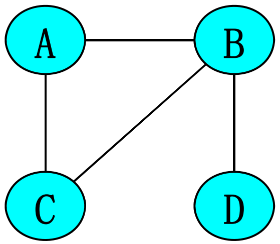


§7. 图

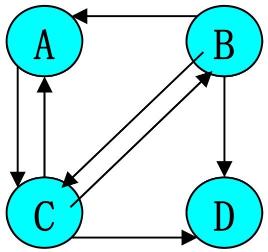
7.1. 图的定义和术语

7.1.3. 图的基本术语

★ 路径：无向图中，从一个顶点到另一个顶点所经过的顶点序列(含自身两个顶点)



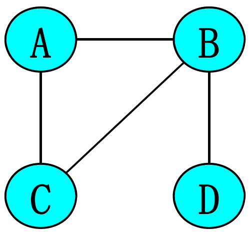
A到D的路径：
A-B-D
A-C-B-D



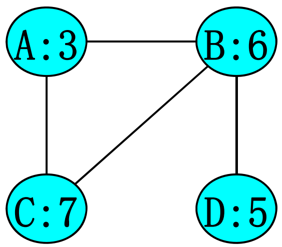
A到D的有向路径：
A-B-D 错误
A-C-B-D 正确
A-C-D 正确

★ 有向路径：有向图中，从一个顶点到另一个顶点所经过的顶点序列，且顶点间的弧同方向

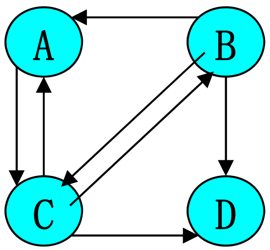
★ 路径长度：
- 路径上边/弧的数目 － 非带权图
- 路径上边/弧的权值之和 － 带权图



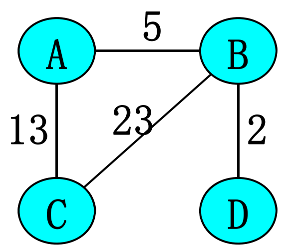
A到D的路径：
A-B-D 路径长度为2
A-C-B-D 路径长度为3



A到D的路径：
A-B-D 路径长度为14
A-C-B-D 路径长度为21



A到D的有向路径：
A-B-D 错误
A-C-B-D 正确 路径长度为3
A-C-D 正确 路径长度为2



A到D的路径：
A-B-D 路径长度7
A-C-B-D 路径长度为38

§7.图

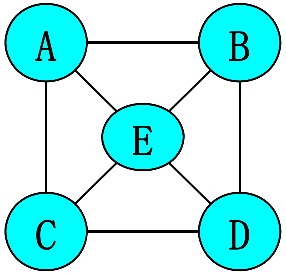
7.1.图的定义和术语

7.1.3.图的基本术语

★ 回路(环)：路径的起点与终点重合

★ 简单路径：序列中顶点不重复出现的路径

★ 简单回路：除起点/终点重合外，其余顶点不重复出现的回路



A-C-E-D　　　：简单路径

A-C-E-B-E-D　：非简单路径

A-B-E-C-A　　：简单回路

A-B-E-D-C-A　：简单回路

A-E-D-C-E-B-A：非简单回路

§7. 图

7.1. 图的定义和术语

7.1.3. 图的基本术语

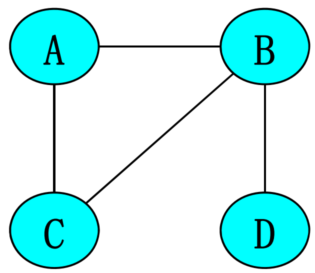
无向图

★ 连通：若两顶点间有路径，则称两点是连通的

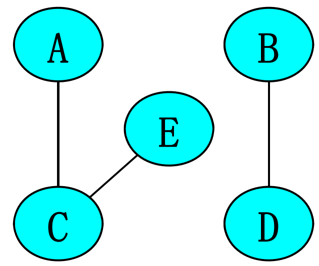
★ 连通图：图中任两个顶点间都是连通的图

★ 连通分量：无向图的极大连通子图

（Σ连通分量=无向图）



连通图



非连通图，连通分量有两个

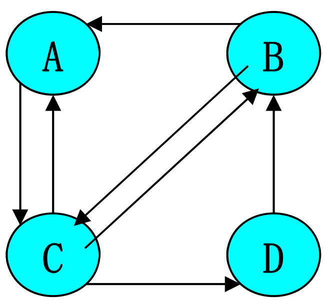
有向图

★ 强连通图：有向图中，任$V_i,V_j(i \neq j)$，都存在V_i->V_j和V_j->V_i的路径

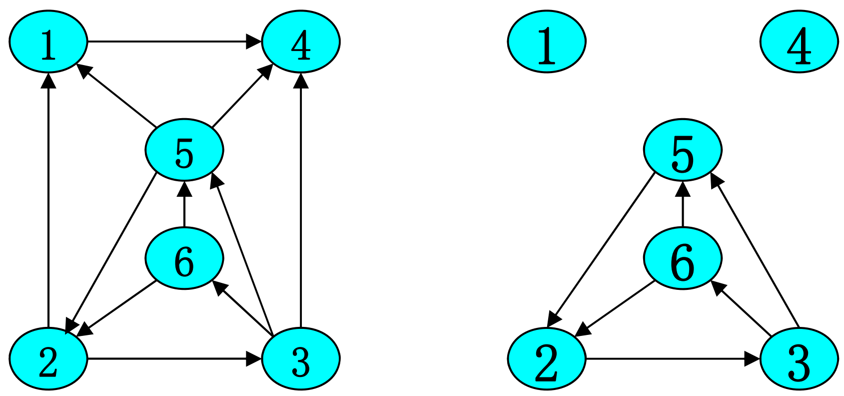
（注意：不是弧！！！）

★ 强连通分量：有向图中的极大连通子图

（Σ强连通分量≠有向图）



强连通图



非强连通图，3个强连通分量

§ 7. 图

7.1. 图的定义和术语

7.1.3. 图的基本术语

★ 连通图的生成树：n个顶点的连通图，仅含n-1条边，且所有顶点均有边相连(极小连通子图)

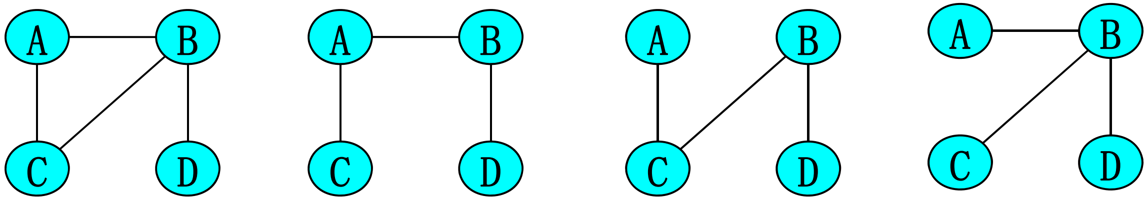
- n个顶点的连通图的生成树,有且仅有n-1条边

  (反之：n个顶点,n-1条边,不一定构成生成树)

  多于n-1条边，一定有回路

  少于n-1条边，必为非连通图

- 图的生成树不止一棵（例：图的三颗生成树）



- 无向图的生成树不严格区分树根、树枝、树叶
- 非连通图构成生成森林

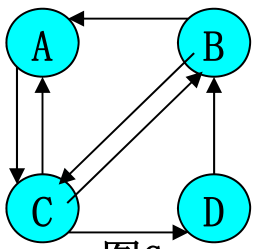
§7.图

7.1.图的定义和术语

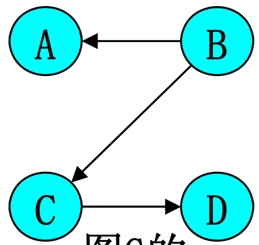
7.1.3.图的基本术语

★ 有向树：有向图中，有且仅有一个顶点的入度为0，其余顶点的入度均为1，则构成一棵有向树(树的本质是有向的)

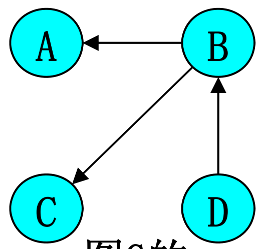
- 入度为0的顶点为树的根结点
- 有向树不是强连通图



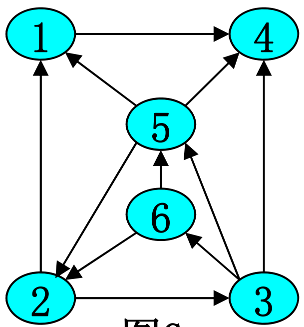
图G
强连通图



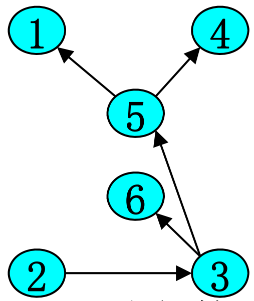
图G的
生成树1
B为根



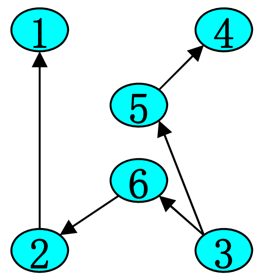
图G的
生成树2
D为根



图G
非强连通图



图G的
生成树1
2为根



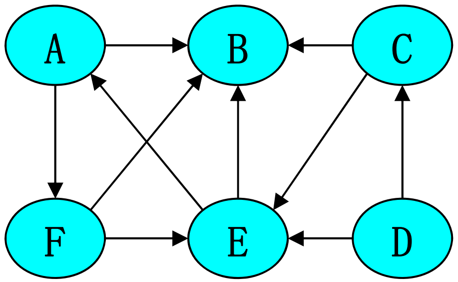
图G的
生成树2
3为根

§7.图

7.1.图的定义和术语

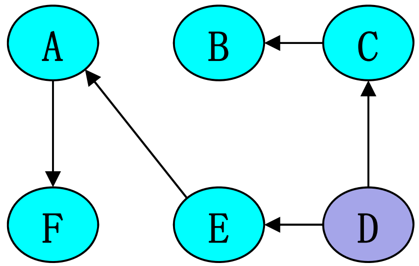
7.1.3.图的基本术语

★ 有向图的生成森林：由若干有向树构成的森林，包含全部顶点(不重复)和构成有向树的弧

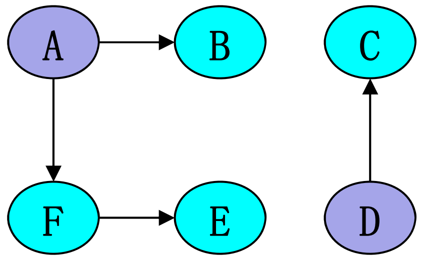


非强连通图
(因为B出度0)

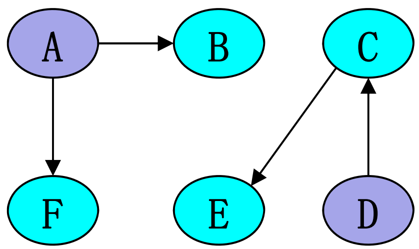
有向树，D为根



有向森林1



有向森林2

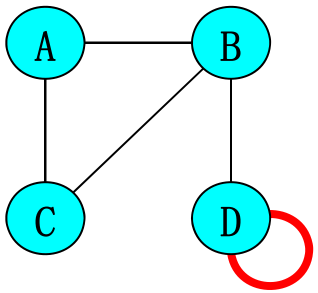


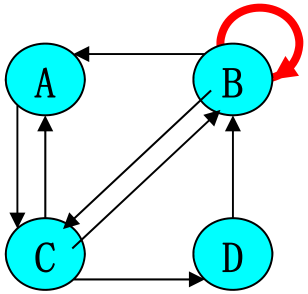
§7. 图

7.1. 图的定义和术语

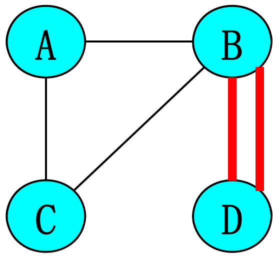
7.1.4. 本章不讨论的图

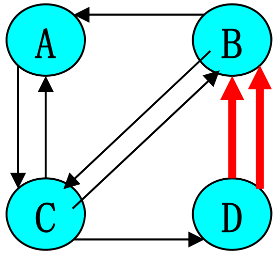
★ 带自环的图





★ 多重图





§ 7. 图

7.2. 图的存储结构

7.2.1. 存储结构的确定

★ 基本结构的存储表示

线性表：顺序/链式结构
树：顺序(完全二叉树)/链式结构
图：任意两顶点都可能有关系，无法用顺序结构因而只能用链式结构

● 顺序结构 ≠ 数组，数组也可以表示链式结构(静态链表)，二维数组也不是顺序结构

★ 多重链表表示法

● 结点同构，k=max(所有顶点出度) － 有向图
　　　　　　k=max(所有顶点度)　 － 无向图

adj_i是邻接点指针

data	adj_1	adj_2	...	adj_k

● 结点异构，degree存放该顶点的出度-有向图
　　　　　　　　　　　　　　　　度　-无向图

data	degree	adj_1	adj_2	...	adj_d

● 与P. 136 树的孩子表示法相似

§7.图

7.2.图的存储结构

7.2.1.存储结构的确定

7.2.2.数组表示法 – 邻接矩阵

7.2.2.1.邻接矩阵的定义

★ 用两个数组分别存储顶点信息和边(弧)的信息

两个数组：一维数组存放顶点的信息（n个顶点）

二维数组存放顶点间关系的信息（n*n）

[图：1 ：有关系 0：无关系

网：权值：有关系 ∞：无关系]

● 将二维数组称为邻接矩阵

§7. 图

7.2. 图的存储结构

7.2.2. 数组表示法 - 邻接矩阵

```
/* P.161 定义 */
#define INFINITY INT_MAX  /* INT_MAX是系统宏定义，表示int型的最大值，
                             用于替代无穷大，要保证权中不出现此值*/

#define MAX_VERTEX_NUM 20 /* 最大顶点个数(可按需修改) */

typedef enum {
    DG,    /* 有向图 */
    DN,    /* 有向网(带权有向图) */
    UDG,    /* 无向图 */
    UDN     /* 无向网(带权无向图) */
    } GraphKind;     /* 注意：DG、DN、UDG、UDN是整型值 */

typedef struct ArcCell {
    VRType     adj;     /* 边/弧的值(边：0/1 弧：权值) */
    InfoType *info;    /* 边/弧的附加信息(一般无) */
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef struct {
    VertexType vexs[MAX_VERTEX_NUM]; /* 顶点信息 */
    AdjMatrix arcs;      /* 邻接矩阵(二维数组表示边/弧)*/
    int vexnum, arcnum; /* 顶点数量、边/弧的数量 */
    GraphKind   kind;   /* 图的类型，4种之一 */
} MGraph;
```

算法讨论时用∞，程序实现时用-1或INT_MAX等

VRType:边/弧的类型
InfoType:附加信息类型
VertexType:顶点类型
使用时具体化即可

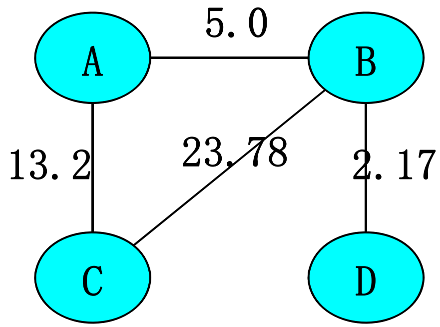
§7.图

7.2.图的存储结构

7.2.2.数组表示法 - 邻接矩阵

```
/*对右图，加以下类型声明*/
typedef double VRType;
typedef void   InfoType;
typedef char   VertexType;

/* 相当于 */
typedef struct {
    char       vexs[20];        /* 顶点信息 */
    double     arcs[20][20];    /* 邻接矩阵*/
    int        vexnum, arcnum;  /* 顶点数量、边/弧数量 */
    GraphKind  kind;            /* 图的类型，4种之一 */
} MGraph;
```

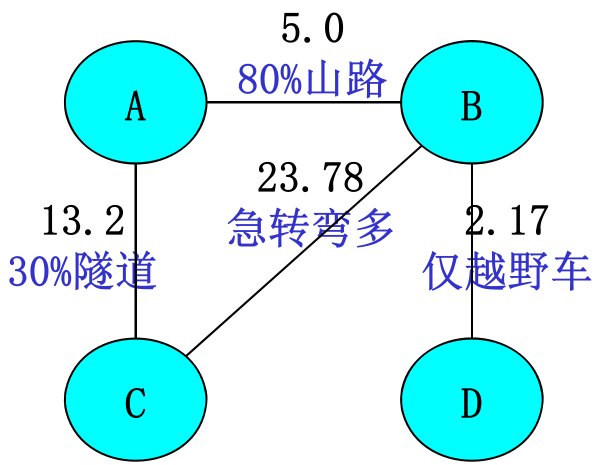


§ 7. 图

7.2. 图的存储结构

7.2.2. 数组表示法 - 邻接矩阵

```
/* 有附加信息的例子 */
typedef double VRType;
typedef char   InfoType;
typedef char   VertexType;
```

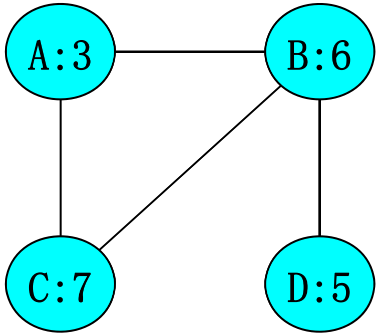



```
/* P.161 定义 */
typedef struct ArcCell {
    VRType    adj;     /* 边/弧的值(边：0/1 弧：权值) */
    InfoType *info;    /* 边/弧的附加信息(一般无) */
} ArcCell, AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
```

§7.图

7.2.图的存储结构

7.2.2.数组表示法 - 邻接矩阵




```
/* 对右图，加以下类型声明 */
struct vnode {
  char name;
  int  value;
};
typedef int  VRType;
typedef void InfoType;
typedef struct vnode VertexType;

/* 相当于 */
struct vnode {
  char name;
  int  value;
};
typedef struct {
    struct vnode vexs [20];     /* 顶点信息 */
    int          arcs[20][20];  /* 邻接矩阵，只有0、1 */
    int          vexnum, arcnum; /* 顶点数量、边/弧数量 */
    GraphKind    kind;          /* 图的类型，4种之一 */
} MGraph;
```

§7.图

7.2.图的存储结构

7.2.2.数组表示法 - 邻接矩阵

```
/* 基于P.161 定义的简化，假设
     1、假设顶点仅有编号信息（从0开始）
     2、边/弧无其它信息
   则简化为 */
typedef struct {
    VRType  arcs[MAX_VERTEX_NUM][MAX_VERTEX_NUM];//边/弧
    int vexnum, arcnum;
    GraphKind   kind;
} MGraph;
```

- 顶点的编号隐含在数组编号中

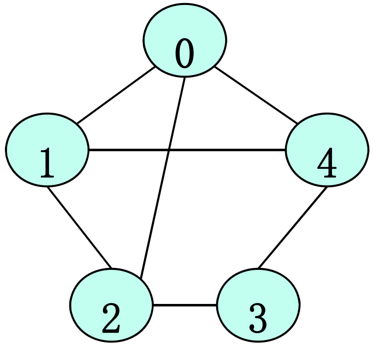
```
/* 某些简化的情况下，可直接用二维数组表示 */
int a[10][10];  /* 表示有10个顶点的图(值仅0、1) */
double b[5][5]; /* 表示权值为double的含5个顶点的网 */
```

§7.图

7.2.图的存储结构

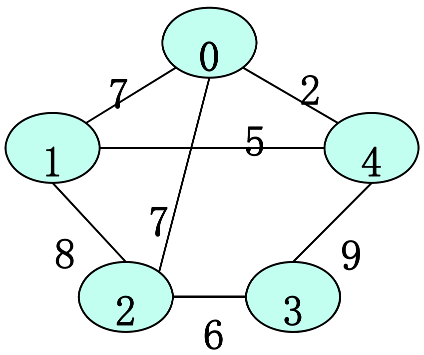
7.2.2.数组表示法 - 邻接矩阵

7.2.2.2.邻接矩阵的表示



无向图

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



无向网

$$A=\begin{pmatrix} 0 & 7 & 7 & 0 & 2 \\ 7 & 0 & 8 & 0 & 5 \\ 7 & 8 & 0 & 6 & 0 \\ 0 & 0 & 6 & 0 & 9 \\ 2 & 5 & 0 & 9 & 0 \end{pmatrix}$$

若0是可能出现的权值，
则可使用-1/INT_MAX等，
保证不是合法权值即可

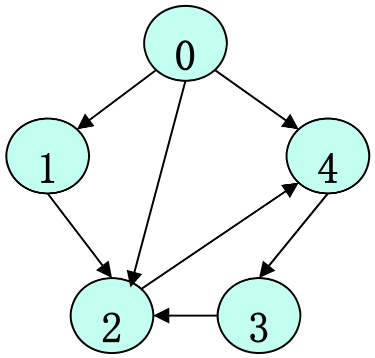
- 若(V_i, V_j)，则第i行第j列以及第j行第i列为1/权值
- 无向图(网)的主对角线为0（无自环的情况下）
- 无向图(网)的边数=邻接矩阵中值为1/权值的数量/2
- 无向图(网)的矩阵是对称矩阵(可以使用压缩存储)

§7.图

7.2.图的存储结构

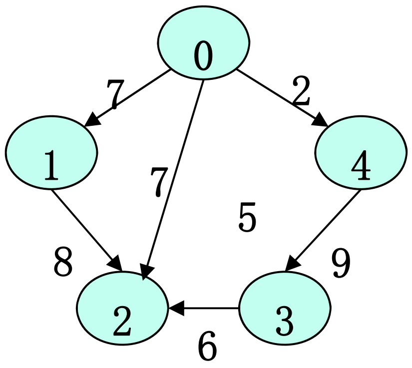
7.2.2.数组表示法 － 邻接矩阵

7.2.2.2.邻接矩阵的表示



有向图

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



有向网

$$A=\begin{pmatrix} 0 & 7 & 7 & 0 & 2 \\ 0 & 0 & 8 & 0 & 0 \\ 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 9 & 0 \end{pmatrix}$$

- 若有$\langle V_i,V_j\rangle$，则第i行第j列为1/权值
- 有向图(网)的主对角线为0（无自环的情况下）
- 有向图(网)的边数=邻接矩阵中值为非0/非∞的数量

§7.图

7.2.图的存储结构

7.2.2.数组表示法 - 邻接矩阵

7.2.2.3.邻接矩阵的运算

★ 判断i,j间是否有边/弧：A[i][j]=0/∞

★ 求顶点的度：

无向图：顶点i的度为第i行(列)为1的元素个数

有向图：顶点i的出度：第i行为1的元素个数

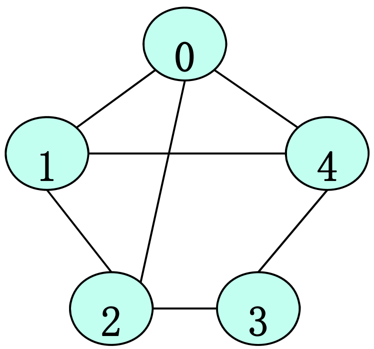
顶点i的入度：第i列为1的元素个数

★ 求顶点的邻接点：

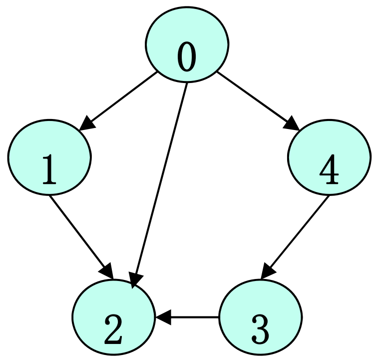
无向图：第i行(列)所有为1的元素的列(行)位置

有向图：第i行所有为1的元素的列位置为i邻接到的顶点（i为尾）

第i列所有为1的元素的行位置为i邻接自的顶点（i为头）



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



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

§7. 图

7.2. 图的存储结构

7.2.2. 数组表示法 - 邻接矩阵

7.2.2.4. 基于邻接矩阵的图的建立算法

★ 假设n个结点，e条边，则邻接矩阵方式所占空间为n^2，与e无关

（条件符合时可以压缩或稀疏矩阵存储）

★ 图建立算法的时间复杂度为 $O(n^2+e*n)$，n为顶点数量，e为边数量

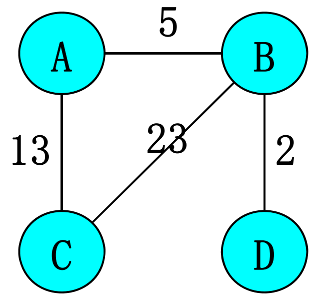
其中 n^2 为初始化邻接矩阵

e*n为读入e条边，每次LocateVex为n

★ 基于邻接矩阵的图的建立算法(P.162 算法7.1)

假设建立右侧形式的图，则类型声明为：

```
typedef int     VRType;
typedef void    InfoType;
typedef char    VertexType;
```




```
status CreateGraph(MGraph &G)
{   cin >> G.kind; //enum, 输入0-3，表示图的类型
    switch(G.kind) {
        case DG:  //有向图
            return CreateDG(G);
        case DN:  //有向网
            return createDN(G);
        case UDG: //无向图
            return CreateUDG(G);
        case UDN: //无向网
            return CreateUDN(G);
        default:
            return ERROR;
        }
     return OK;
}
```

★ 基于邻接矩阵的无向网的建立算法(P.162 算法7.2)

```
Status CreateUDN (MGraph &G)
{   cin >> G.vexnum >> G.arcnum >> IncInfo; //读入顶点数、边数、是否有附加信息的标记
    for (i=0; i<G.vexnum; i++)                          循环读入所有顶点
        cin >> G.vexs[i];                             的值(已假设为char型)
    for (i=0; i<G.vexnum; i++)                            初始化邻接矩阵
        for (j=0; j<G.vexnum; j++) {
            G.arcs[i][j].adj  = INFINITY; //INT_MAX(∞)
            G.arcs[i][j].info = NULL;     //附加信息              书上直接
        }                                 G.arcs[i][j]= {INFINITY, NULL};
    /* 循环读入所有的边 */
    for (k=0; k<G.arcnum; k++) {
        cin >> v1 >> v2 >> w; //2个顶点(假设char)+1个权
        i = LocateVex(G, v1);//找v1(例如'A')对应的下标
        j = Locatevex(G, v2);//找v2(例如'B')对应的下标
        G.arcs[i][j].adj = w;//权值赋值
        if (IncInfo) //如果需要输入附加信息，则输入              Input函数中可能有
            Input(G.arcs[i][j].info); //另行实现的函数        内存申请、赋值等操作
        G.arcs[j][i] = G.arcs[i][j]; //对称弧赋值，若有附加信息则
                                     //可能有内存申请、拷贝等操作
        }
    return OK;
}
```

★ P.162 算法7.2中LocateVex的实现

```
int LocateVex(MGraph G, VertexType  v)
{
    int i;
    for (i=0; i<G.vexnum; i++)
        if (v==G.vexs[i])
            return i; //找到则返回下标值(i≥0)

    /* 输入错误则直接退出程序 */
    cout << "Input Error" << endl;
    exit(1);
}
```

VertexType = char

若此处不是exit，而是return -1
则CreateUDN中要对LocateVex的
返回值进行判断，进行容错处理

§7. 图

7.2. 图的存储结构

7.2.3. 链式表示法 – 邻接表

7.2.3.1. 邻接表的含义及存储结构

含义：为每个顶点建立一个单链表，单链表分为头结点和表结点，头结点存放某个顶点的信息，表结点中存放该结点的邻接点(邻接到)信息，表示该顶点与其它顶点的邻接关系，称为邻接表

★ 分为头结点和表结点

data	firstarc

头结点

adjvex	nextarc	info

表结点

data　　：顶点相关信息
firstarc：指向第一个邻接点的指针

adjvex ：邻接点序号
nextarc：指向下一个邻接点的指针
info　 ：边/弧的附加信息

★ 头结点一般以顺序结构(也可以链相接)的形式存储，这样可以随机访问任一顶点的链表

★ 邻接表的存储结构(P.163)

★ 邻接表的存储结构(P.163)

```
#define MAX_VERTEX_NUM   20   //顶点的最大数量，可按需修改
```

```
typedef struct ArcNode {  //表结点
    int             adjvex;    //邻接顶点的序号
    struct ArcNode *nextarc;   //指向下一条边/弧的指针
    InfoType       *info;      //边/弧的附加信息
} ArcNode;
```

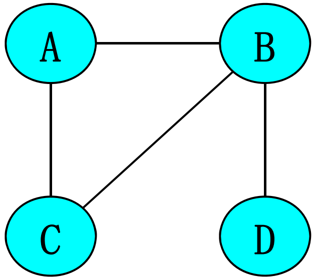
```
typedef struct VNode { //头结点
    VertexType  data;      //顶点信息
    ArcNode    *firstarc; //指向第1条边/弧的指针
} VNode, AdjList[MAX_VERTEX_NUM];
```

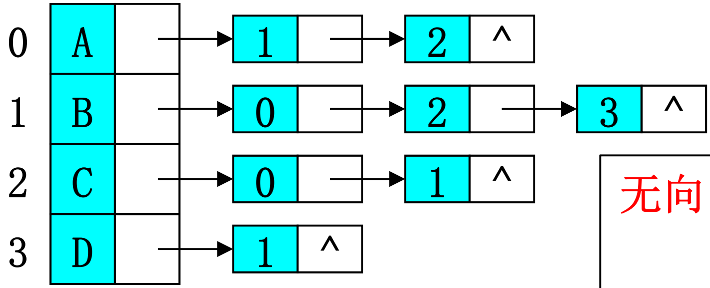
```
typedef struct { //图
    AdjList vertices;    //一维数组，存放所有头结点
    int vexnum, arcnum;  //顶点数量、边的数量
    int kind;            //图的种类标志，同P.161的enum
}ALGraph;
```

7.2. 图的存储结构

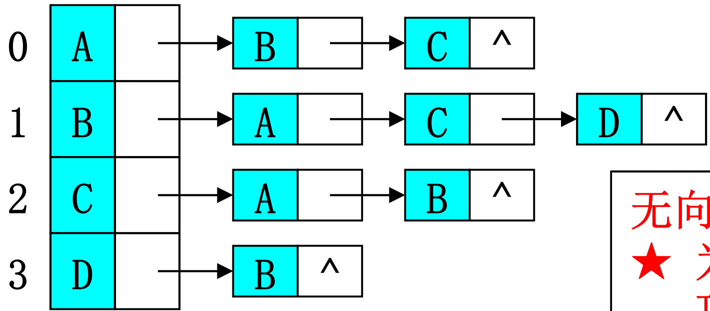
7.2.3. 链式表示法 - 邻接表

7.2.3.2. 邻接表的表示



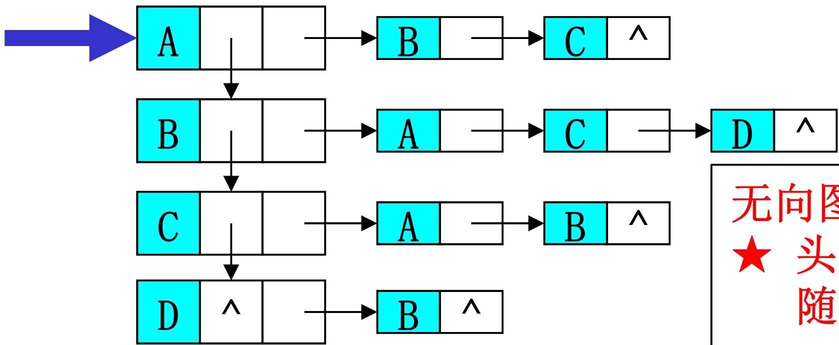


无向图 typedef char Vertextype;
typedef void InfoType;



无向图

★ 为便于理解，图示时直接标示为顶点的值而不是序号(下同)



无向图

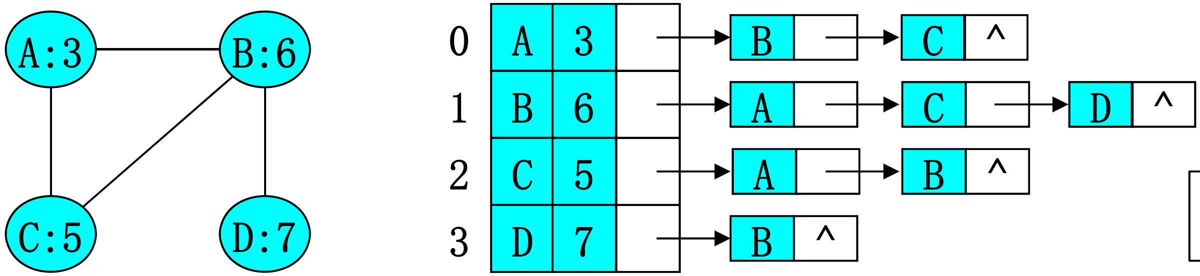
★ 头结点可以采用链式结构，但不能随机存取，故不再讨论(下同)

§7.图

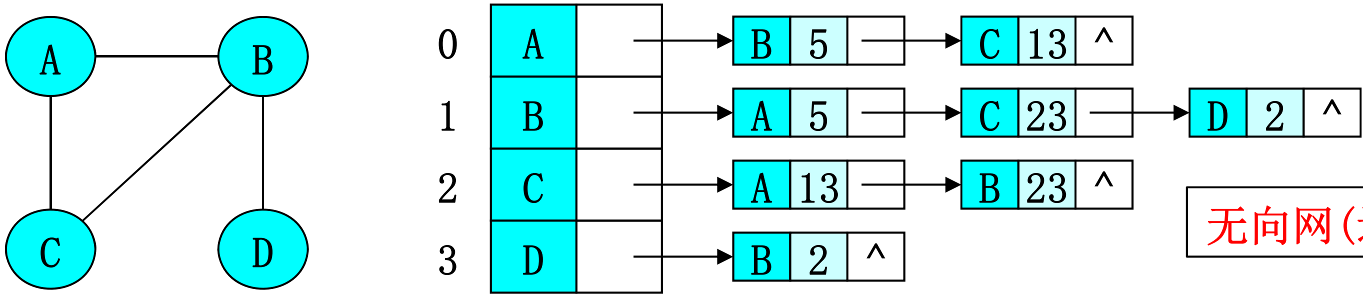
7.2.图的存储结构

7.2.3.链式表示法 – 邻接表

7.2.3.2.邻接表的表示



无向网(结点带权值)

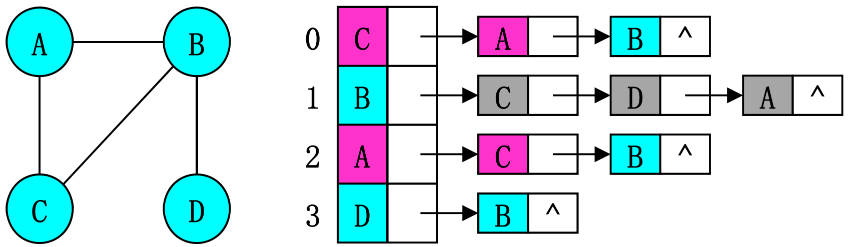


无向网(边带权值)

★ 无向图的邻接表中表结点的总数=边数*2

★ 若(V_i, V_j)，则j是以i为头结点的单链表中的表结点
　　　　　　　　i是以j为头结点的单链表中的表结点

★ 无向图邻接点的排列无顺序，可任意

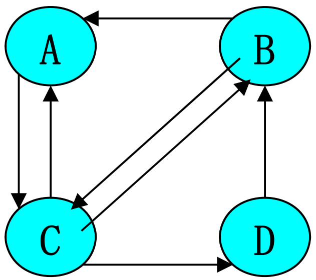


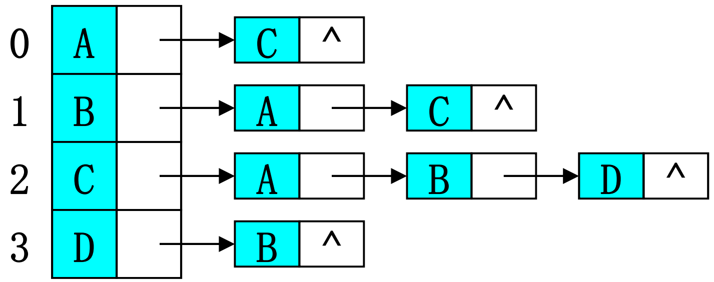
§7.图

7.2.图的存储结构

7.2.3.链式表示法 - 邻接表

7.2.3.2.邻接表的表示





有向图

★ 有向图的邻接表中表结点的总数=边数

★ 若$\langle V_i, V_j \rangle$，则j是以i为头结点的单链表中的结点

★ 有向图的邻接点排列无顺序，可任意

§7. 图

7.2. 图的存储结构

7.2.3. 链式表示法 - 邻接表

7.2.3.2. 邻接表的表示

★ 假设n个顶点，e条边，则占用n个头结点和2e个表结点(当边稀疏时，比邻接矩阵占用空间少)

★ 建立邻接表的时间复杂度为O(n*e)，即读e条边，每次LocateVex找顶点信息n，若顶点信息即为顶点的下标，则时间复杂度为O(n+e)

§7.图

7.2.图的存储结构

7.2.3.链式表示法 - 邻接表

7.2.3.3.邻接表的运算

★ 判断i,j间是否有边/弧：

无向：第i(j)个头结点的单链表中是否有j(i)出现

有向：第i个头结点的单链表中是否有j出现

★ 求顶点的度：

无向图：第i个头结点的单链表中表结点的个数

有向图：出度：第i个头结点的单链表中表结点的个数

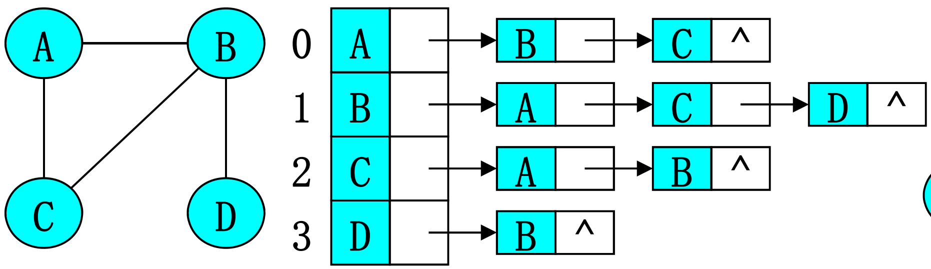
入度：所有头结点所指的单链表中结点值为i的表结点的个数(不方便)

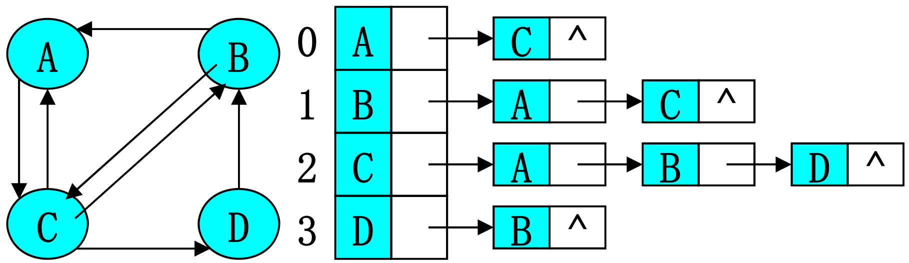
★ 求顶点的邻接点：

无向图：第i个头结点的单链表中所有的表结点

有向图：邻接到：第i个头结点的单链表中所有的表结点

邻接自：所有单链表中出现结点值为i的表结点的链表的头结点(不方便)





§7. 图

7.2. 图的存储结构

7.2.3. 链式表示法 - 邻接表

7.2.3.4. 有向图的逆邻接表

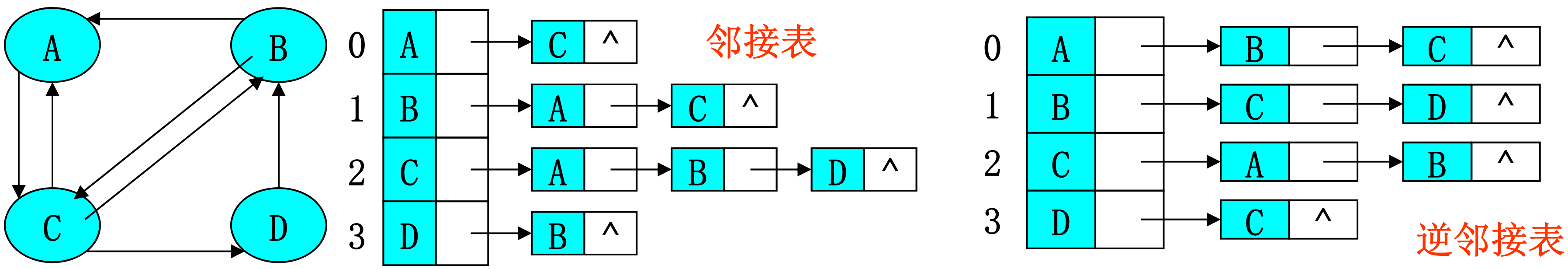
含义：形式同邻接表，结点中存放该结点邻接自的结点的信息

★ 求顶点的度：

邻接表	出度：第i个头结点的单链表中表结点的个数 入度：所有头结点所指的单链表中结点值为i的表结点的个数(不方便)
逆邻接表	出度：所有头结点所指的单链表中结点值为i的表结点的个数(不方便) 入度：第i个头结点的单链表中表结点的个数

★ 求顶点的邻接点：

邻接表	邻接到：第i个头结点的单链表中所有的表结点 邻接自：所有单链表中出现结点值为i的表结点的链表的头结点(不方便)
逆邻接表	邻接到：所有单链表中出现结点值为i的表结点的链表的头结点(不方便) 邻接自：第i个头结点的单链表中所有的表结点



§7.图

7.3.图的遍历

7.3.1.深度优先搜索(DFS)

深度优先的步骤：

① 任意选择一个未被访问的顶点(V_0)做为出发点，首先访问自身，然后进栈

② 选中V_0的一个未被访问的邻接点做为新的V_0，重复①

③ 重复①、②至无法找到未被访问的邻接点为止

④ 从栈中取一个元素，做为新的V_0，重复②，至栈空

⑤ 若尚有未访问的顶点，则任选一个，重复①-④至所有顶点均被访问过

★ 类似于树的先根遍历

★ 连通图无步骤⑤

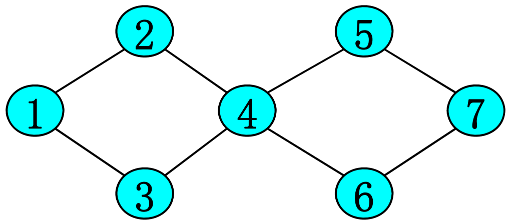
★ 深度优先搜索的时间复杂度

邻接矩阵方式： $O(n^2)$

n个顶点，每个找所有边需$O(n)$

邻接表方式： $O(n+e)$

每个顶点一次 $O(n)$ + 找所有边$O(e)$



从1出发的深度优先序列

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

★ 深度优先搜索(DFS)算法(P.169 算法7.4、7.5)

```
Boolen visited[MAX_VERTEX_NUM];    //全局标志数组
Status (*VisitFunc)(int v);        //全局函数指针

void DFSTraverse(Graph G, Status(*Visit)(int v))
{   VisitFunc = Visit;          //全局函数指针赋值
    /* 首先全部置未访问标记 */
    for(v=0; v<G.vexnum; v++)
        visited[v] = FALSE;
    /* 循环所有顶点，如未被访问过则调用DFS算法 */
    for(v=0; v<G.vexnum; v++)
        if (!visited[v])
            DFS(G, v);
}

/* 用深度优先算法从顶点v开始遍历
   注意：只能遍历整个连通图，若非连通图，则多次调用 */
void DFS(Graph G, int v)
{
    visited[v] = TRUE;  //置已访问标记
    VisitFunc(v);       //访问该顶点

    /* 从v的第1个邻接点循环到最后一个邻接点
       若该邻接点未被访问过则调用DFS算法 */
    for (w=FirstAdjvex(G, v); w>=0; w=NextAdjVex(G, v, w))
        if (!visited[w])
            DFS(G, w);
}
```

★ FirstAdjvex和NextAdjVex这两个函数找到返回下标，找不到返回-1
★ 这两个函数需另行实现，而且实现的方式因为存储结构的不同而不同

§7.图

7.3.图的遍历

7.3.1.深度优先搜索(DFS)

7.3.2.广度优先搜索(BFS)

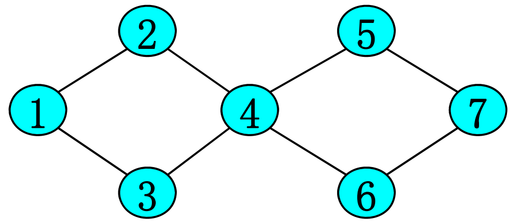
广度优先的步骤:

① 任意选择一个未被访问的顶点(V_0)做为出发点,首先访问自身,然后进队列

② 依次访问V_0的所有未被访问的邻接点并进队列

③ 从队列中取一个元素,做为新的V_0，重复②,至队空

④ 若尚有未访问的顶点，则任选一个，重复①-③至所有顶点均被访问过



从1出发的广度优先序列

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

★ 类似于树的层次遍历

★ 连通图无步骤④

★ 广度优先搜索的时间复杂度

邻接矩阵方式： $O(n^2)$

n个顶点，每个顶点循环检测是否访问过=>n*n

邻接表方式：O(e)

循环所有单链表，未置访问标记的才访问

★ 广度优先搜索(BFS)算法(P.170算法 7.6)

```
Boolen visited[MAX_VERTEX_NUM];   //全局标志数组
void BFSTraverse(Graph G, Status (*Visit)(int v))
{
    /* 初始全部置未访问标记 */
    for(v=0; v<G.vexnum; v++)
        visited[v] = FALSE;
    InitQueue(Q); /* 初始化辅助队列 */

    /* 循环所有顶点，未访问过则进行访问 */
    for(v=0; v<G.vexnum; v++)
        if (!visited[v]) {
            visited[v] = TRUE;  //置已访问标记
            (*Visit)(v);        //访问
            EnQueue(Q, v);      //顶点进队列
            while(!QueueEmpty(Q)) { //队列非空则循环
                DeQueue(Q, u); //顶点出队列
                /* 遍历改顶点的所有邻接点
                for(w=FirstAdjVex(u); w>=0; w=NextAdjVex(G,u,w))
                    if (!visited[w]) {
                        visited[w] = TRUE;  //置已访问标记
                        (*Visit)(w);        //访问
                        EnQueue(Q, w);      //顶点进队列
                    } // end of if
                } // end of for
            } // end of while
        } //end of if
    DestroyQueue(Q);  /* 销毁辅助队列(书上无) */
    return;
}
```

★ FirstAdjvex和NextAdjVex这两个函数找到返回下标，找不到返回-1
★ 这两个函数需另行实现，而且实现的方式因为存储结构的不同而不同

§7.图

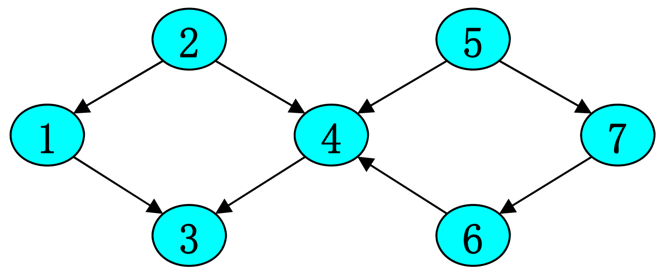
7.3.图的遍历

7.3.3.有向图的DFS和BFS

★ 规则同无向图，但是要注意弧必须是同方向的

★ 算法实现时，DFSTraverse和BFSTraverse的实现方法不变，仅按需修改FirstAdjvex以及NextAdjVex这两个函数的实现，使其找邻接点时保证弧同向即可

邻接矩阵：按行找，不需要改
　　　　　按列找，改
邻接表　：不需要改
逆邻接表：改（无向图无）



从1出发的深度优先序列
1 3 需要另找
2 4 需要另找
5 7 6 结束
=> 1 3 2 4 5 7 6 - 3次
还有很多种，都是1 3开头
1 3 7 6 4 2 5
1 3 5 7 6 4 2
...

从1出发的广度优先序列
1 3 需要另找
2 4 需要另找
5 7 6 结束
=> 1 3 2 4 5 7 6 - 3次 （和深度一样）
还有很多种，都是1 3开头
1 3 5 4 7 6 2
1 3 5 7 4 6 2
...

§7.图

7.4.图的连通性问题

7.4.1.无向图的连通分量和生成树

无向图的连通分量：

连通图：一次深度/广度优先搜索即可遍历完

非连通图：多次调用深度/广度优先搜索

★ 多次得到的顶点集合+所有依附于顶点的边=非连通图的连通分量（极大连通子图）

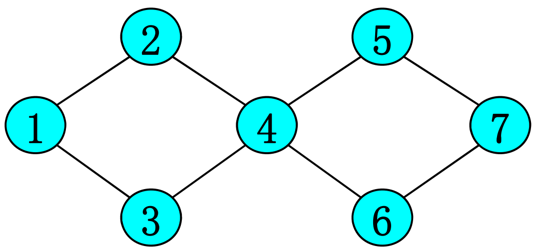
无向图的极小连通子图：

设E(G)为连通图G的所有边的集合，以某种方式遍历时，必分为两部分：

T(G)：遍历中经过的边，

B(G)：遍历中未经过的边(若有回路，B(G)必不空)

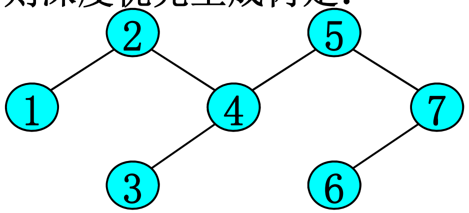
G的顶点集+T(G)=G的极小连通子图（生成树）



假设深度优先遍历序列是：

1 2 4 5 7 6 3

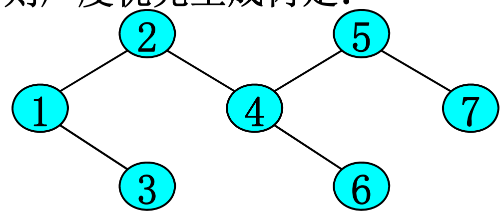
则深度优先生成树是：



假设广度优先遍历序列是：

1 2 3 4 5 6 7

则广度优先生成树是：



连通图：深度优先生成树

广度优先生成树

非连通图：深度优先生成森林

广度优先生成森林

也可以由其它形式的遍历构成生成树

§7. 图

7.4. 图的连通性问题

7.4.1. 无向图的连通分量和生成树

7.4.2. 有向图的强连通分量和有向生成树(略)

§7.图

7.4.图的连通性问题

7.4.3.无向连通网的最小生成树

7.4.3.1.基本概念

★ 最小生成树

对于无向网(带权图，假设边带权)，其生成树可有多种形式，其中各边权值之和最小的生成树称为最小代价生成树，简称为最小生成树

- 最小代价在实际生活中有很多应用
  通信网、公路网、铁路网的布局等
- 生成树结构比较脆弱，实际生活中，一条边的损坏或拥塞就会导致形成非连通图或效率降低，因此是最经济的但不是最安全的，实际生活中的应用可能需要适当甚至高度的冗余

★ MST性质

假设 N=(V, {E}) 是一个连通网，U是顶点集合V的一个非空真子集，若(u,v)是一条具有最小权值的边，且u∈U，v∈V-U，则必然存在一颗包含边(u,v)的最小生成树

证明：假设最小生成树T中不包含(u,v)

=> 将(u,v)加入T中，T中必有回路，且(u,v)存在于回路中回路中其它边的代价均比(u,v)大

=> T是生成树,故T上必存在另一条边(u',v'),u'∈U,v'∈V-U且u和u'，v和v'之间均有路径相通

=> 删去边(u',v')，则可消除回路,得到另外一颗生成树T',T'中包含(u,v)

=> (u,v)的代价小于(u',v')，故T'的代价小于T，T'是最小生成树

=> 与假设矛盾，故假设不成立

§7.图

7.4.图的连通性问题

7.4.3.无向连通网的最小生成树

7.4.3.1.基本概念

★ 最小生成树

对于无向网(带权图，假设边带权)，其生成树可有多种形式，其中各边权值之和最小的生成树称为最小代价生成树，简称为最小生成树

★ MST性质

假设 N=(V, {E}) 是一个连通网，U是顶点集合V的一个非空真子集，若(u,v)是一条具有最小权值的边，且u∈U，v∈V-U，则必然存在一颗包含边(u,v)的最小生成树

★ 构造准则

- 基于MST性质，尽可能用网络中权值最小的边
- 用且仅用n-1条边来连通n个顶点
- 不能使用产生回路的边

§7.图

7.4.图的连通性问题

7.4.3.无向连通网的最小生成树

7.4.3.1.基本概念

7.4.3.2.普利姆(Prim)算法

★ 算法描述

① 从网中任选一个顶点进入最小生成树集合中

② 找出一端顶点在最小生成树集合中，另一端不在最小生成树集合中的所有边，从中选取权值最小的边,将该边添加到最小生成树集合中

③ 重复执行②，直到所有顶点都包含在最小生成树集合中为止

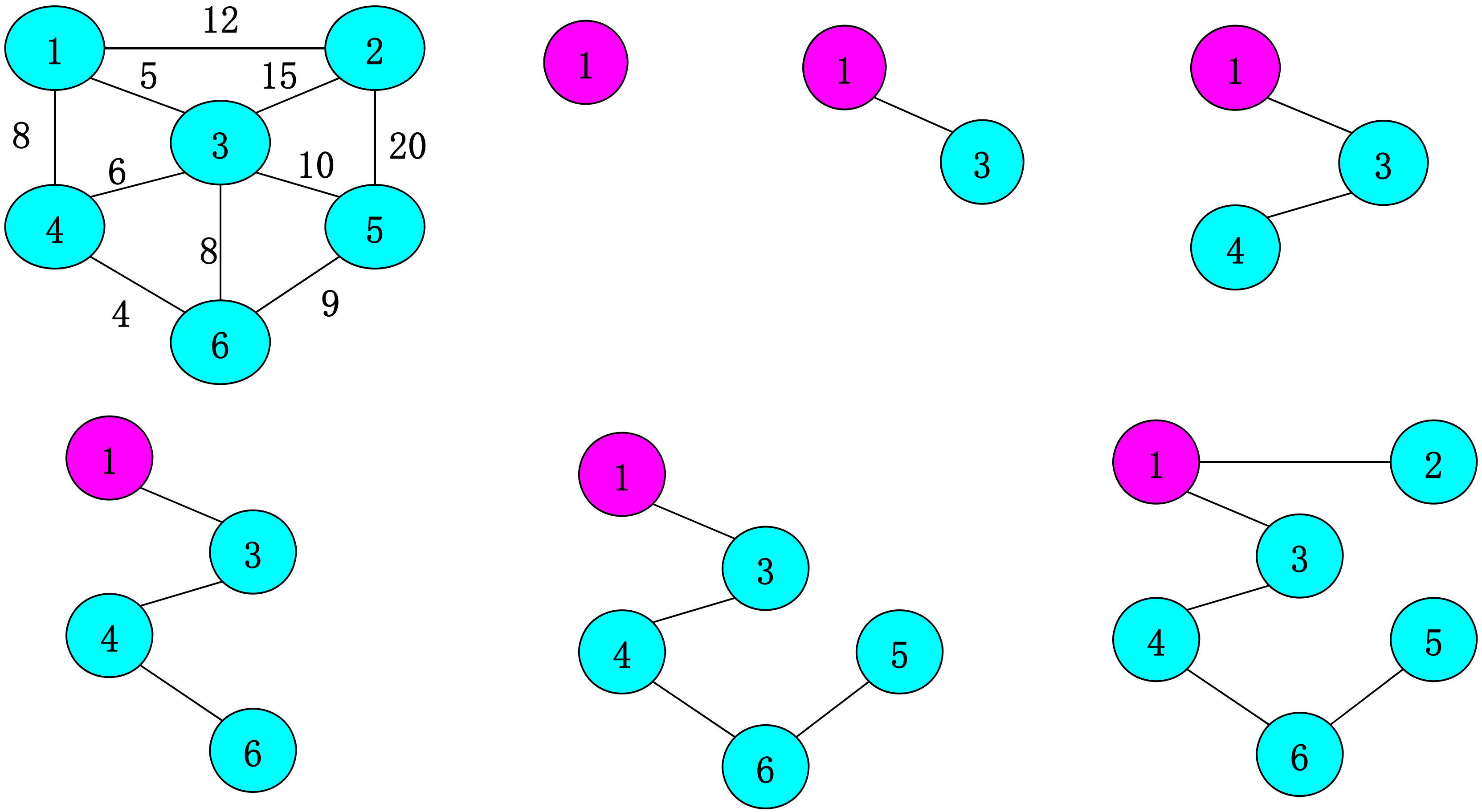
§7. 图

7.4. 图的连通性问题

7.4.3. 无向连通网的最小生成树

7.4.3.1. 基本概念

7.4.3.2. 普利姆(Prim)算法(选v1做为初始点)



最小权值=36

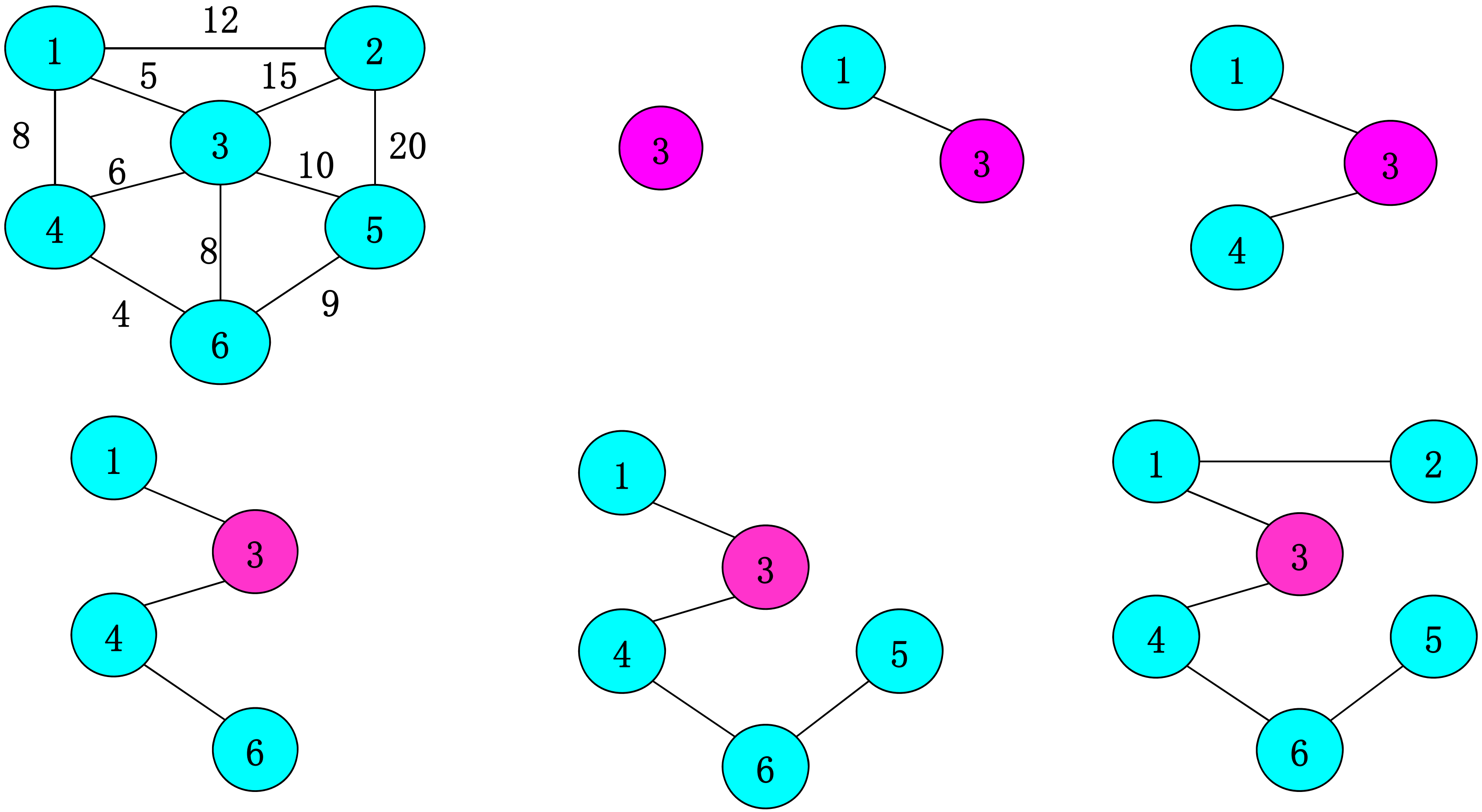
§7. 图

7.4. 图的连通性问题

7.4.3. 无向连通网的最小生成树

7.4.3.1. 基本概念

7.4.3.2. 普利姆(Prim)算法(选v3做为初始点)



最小权值=36

§7.图

7.4.图的连通性问题

7.4.3.无向连通网的最小生成树

7.4.3.1.基本概念

7.4.3.2.普利姆(Prim)算法

★ 算法描述

★ 算法实现

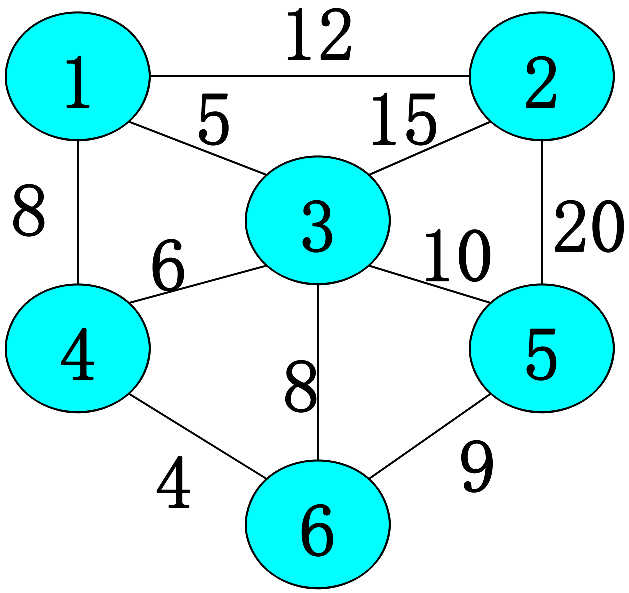
- 如何求最小代价的边?
- 如何记录U中顶点集合?

附设一个closedge数组，记录从U到V-U的具有最小代价的边，每个元素要记录权值及依附的顶点

```
/* 定义 closedge 辅助数组 (P.175) */
struct {
    VertexType adjvex;          //顶点的值
    VRType     lowcost;         //边的权值
} closedge[MAX_VERTEX_NUM];
```

§7.图

7.4.3.2.普利姆(Prim)算法



		0	1	2	3	4	5
		v1	v2	v3	v4	v5	v6
0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

左图的邻接矩阵表示

假设从v1结点开始，则初始：

U = {v1}，V-U = {v2, v3, v4, v5, v6}

k = 0 (v1的下标)

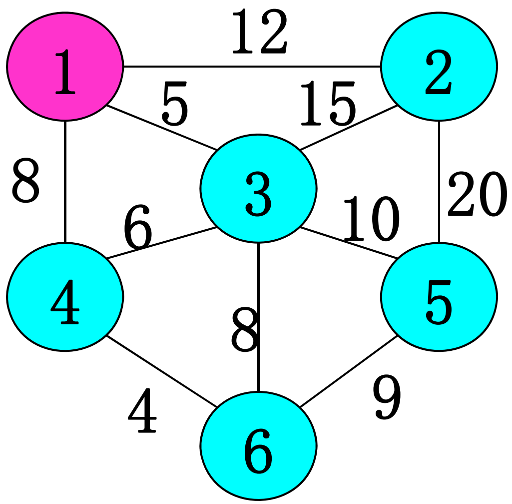
初始化closedge数组：

① adjvex全部赋起始点v1

② lowcost赋邻接矩阵中v1行

③ 仅lowcost[k]置0

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX



		v1	v2	v3	v4	v5	v6
0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

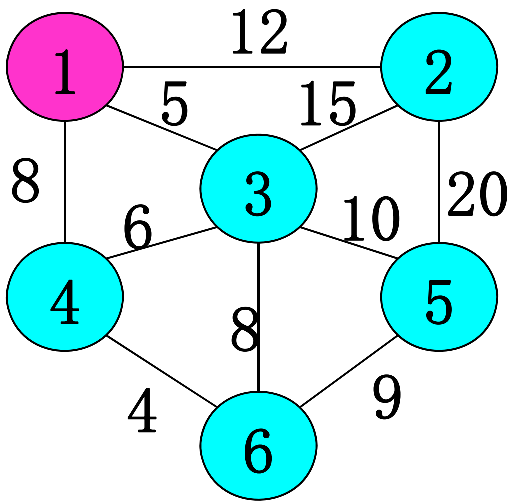
进行i=1-5 的循环(d剩余的5个顶点)，进行如下操作：

① 在closedge中选lowcost>0中最小值的下标 => k

② 输出closedge[k].adjvex和G.vexs[k]

③ 第k个顶点并入U

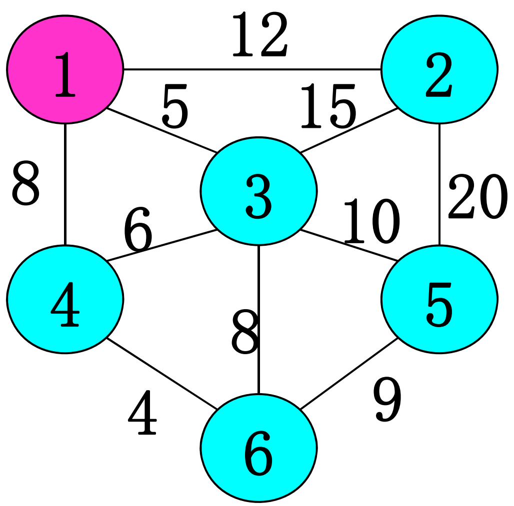
④ 循环所有顶点，若与第k个顶点的权 < 原来的权值则重新选择最小边



		0	1	2	3	4	5
		v1	v2	v3	v4	v5	v6
0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

i=1：① 选最小值，得 k=2(v3)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX
adjvex 新	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX

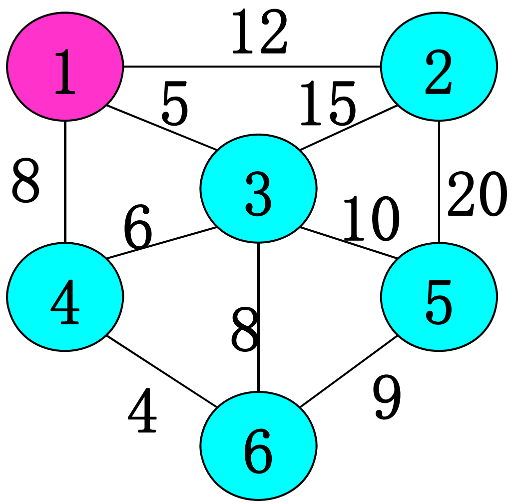


0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

i=1: ① 选最小值，得 k=2(v3)

② 输出closedge[k].adjvex和G.vexs[k](v1,v3)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX
adjvex 新	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX



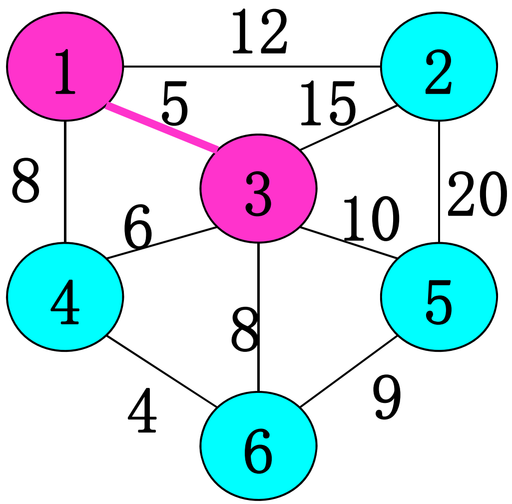
0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

i=1: ① 选最小值，得 k=2(v3)

② 输出closedge[k].adjvex和G.vexs[k](v1,v3)

③ 第k个顶点并入U closedge[k].lowcost = 0;

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX
adjvex 新	v1	v1	v1	v1	v1	v1
lowcost	0	12	0	8	INT_MAX	INT_MAX



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

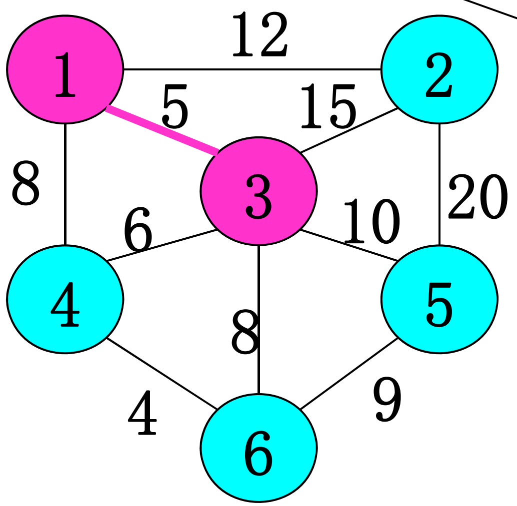
i=1: ① 选最小值，得 k=2(v3)

② 输出closedge[k].adjvex和G.vexs[k](v1,v3)

③ 第k个顶点并入U closedge[k].lowcost = 0;

④ 重选最小边(邻接矩阵的v3行与adjvex比较)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v1	v1	v1
lowcost	0	12	5	8	INT_MAX	INT_MAX
adjvex 新	v1	v1	v1	v3	v3	v3
lowcost	0	12	0	6	10	8



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

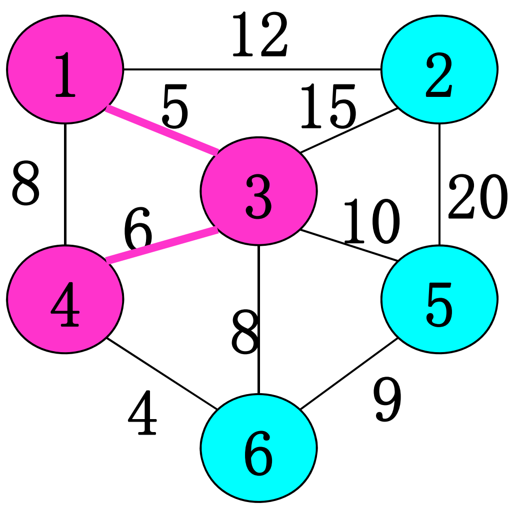
i=2: ① 选最小值，得 k=3(v4)

② 输出closedge[k].adjvex和G.vexs[k](v3,v4)

③ 第k个顶点并入U closedge[k].lowcost = 0;

④ 重选最小边(邻接矩阵的v4行与adjvex比较)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v3	v3	v3
lowcost	0	12	0	6	10	8
adjvex 新	v1	v1	v1	v3	v3	v4
lowcost	0	12	0	0	10	4



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

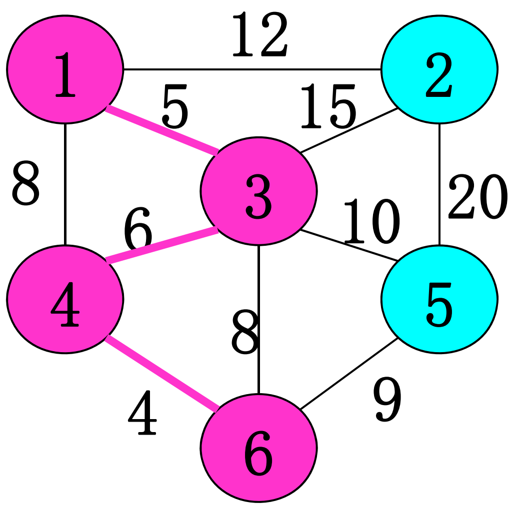
i=3：① 选最小值，得 k=5(v6)

② 输出closedge[k].adjvex和G.vexs[k](v4,v6)

③ 第k个顶点并入U closedge[k].lowcost = 0;

④ 重选最小边(邻接矩阵的v6行与adjvex比较)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v3	v3	v4
lowcost	0	12	0	0	10	4
adjvex 新	v1	v1	v1	v3	v6	v4
lowcost	0	12	0	0	9	0



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

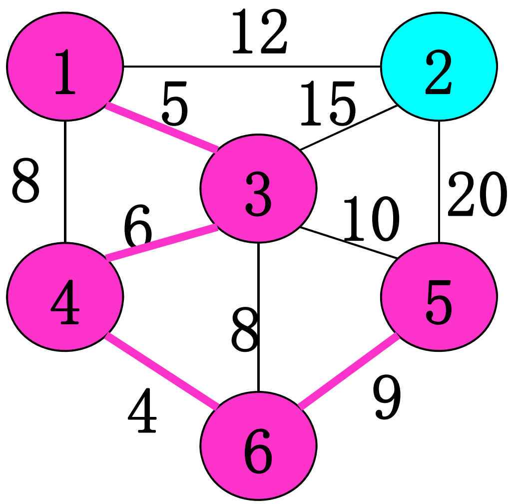
i=4: ① 选最小值，得 k=4(v5)

② 输出closedge[k].adjvex和G.vexs[k](v6,v5)

③ 第k个顶点并入U closedge[k].lowcost = 0;

④ 重选最小边(邻接矩阵的v5行与adjvex比较)

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v3	v6	v4
lowcost	0	12	0	0	9	0
adjvex 新	v1	v1	v1	v3	v6	v4
lowcost	0	12	0	0	0	0



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

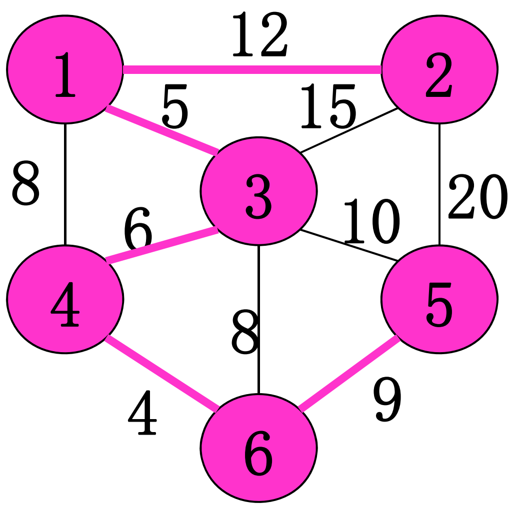
i=5: ① 选最小值，得 k=1(v2)

② 输出closedge[k].adjvex和G.vexs[k](v1,v2)

③ 第k个顶点并入U closedge[k].lowcost = 0;

④ 重选最小边(邻接矩阵的v2行与adjvex比较)

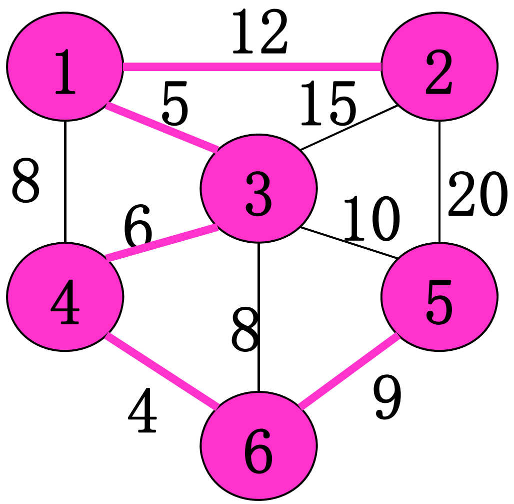
closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex 旧	v1	v1	v1	v3	v6	v4
lowcost	0	12	0	0	0	0
adjvex 新	v1	v1	v1	v3	v6	v4
lowcost	0	0	0	0	0	0



0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

循环结束，依次输出的的(v1, v3)、(v3, v4)、(v4, v6)、(v6, v5)、(v1, v2)5条边+6个顶点 = 最小生成树

closedge	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)	5(v6)
adjvex	v1	v1	v1	v3	v6	v4
lowcost	0	0	0	0	0	0



		0	1	2	3	4	5
		v1	v2	v3	v4	v5	v6
0	v1	/	12	5	8	/	/
1	v2	12	/	15	/	20	/
2	v3	5	15	/	6	10	8
3	v4	8	/	6	/	/	4
4	v5	/	20	10	/	/	9
5	v6	/	/	8	4	9	/

★ 算法实现(P.175 算法7.9 基于邻接矩阵的存储)

```
/* G是图的邻接矩阵表示，u是指定的起点 */
void MiniSpanTree_PRIM(MGraph G, VertexType u)
{
    int totalcost = 0;  //书上缺此句，用于求最小代价

    k = LocateVex(G, u);  //找出起点的下标放入k中
    /* 初始化closedge数组 */
    for (j=0; j<G.vexnum; j++)
        if (j!=k) {
            closedge[j].adjvex  = u;
            closedge[j].lowcost = G.arcs[k][j].adj;
            }
    /* 权值为0表示顶点属于U集合，初始只有u(下标k)属于U*/
    closedge[k].lowcost = 0;

    /* 剩余顶点进行循环 */
    for (i=1; i<G.vexnum; i++) {
        k = minimun(closedge, G.vexnum);
        cout << closedge[k].adjvex << G.vexs[k];//输出边
        totalcost += closedge[k].lowcost; //累加最小代价
        closedge[k].lowcost = 0; //第[k]个顶点归入U中
        for (j=0; j<G.vexnum; j++)
            if (G.arcs[k][j].adj < closedge[j].lowcost){
                closedge[j].adjvex  = G.vexs[k];
                closedge[j].lowcost = G.arcs[k][j].adj;
                }
        }
}
```

若顶点k,j之间无边，
则G.arcs[k][j]是INT_MAX

扫描所有顶点
与第k个的权值
比原来小则更新

★ minimun的实现

```
int minimun(closedge[MAX_VERTEX_NUM], vexnum)
{   int pos, minCost=INT_MAX;
    for (j=0; j<vexnum; j++) {
        if (closedge[j].lowcost < minCost &&
                        closedge[j].lowcost!=0){
            minCost=closedge[j].lowcost;//更新
                        pos=j;  //记录最小值的下标
            }
        }
    return pos; //返回最小值的下标
}
```

§7.图

7.4.图的连通性问题

7.4.3.无向连通网的最小生成树

7.4.3.1.基本概念

7.4.3.2.普利姆(Prim)算法

7.4.3.3.克鲁斯卡尔(Kruskal)算法

★ 算法描述

① 将n个顶点做为n个独立的连通分量加入到最小生成树中(不含任何边)

② 从所有边中选择权值最小的边，若该边的两个顶点分属于不同的连通分量，则将边加入最小生成树集合中；否则舍弃，再取权值次小的边进行判断

③ 重复执行②，到所有顶点属于同一个连通分量为止

§7. 图

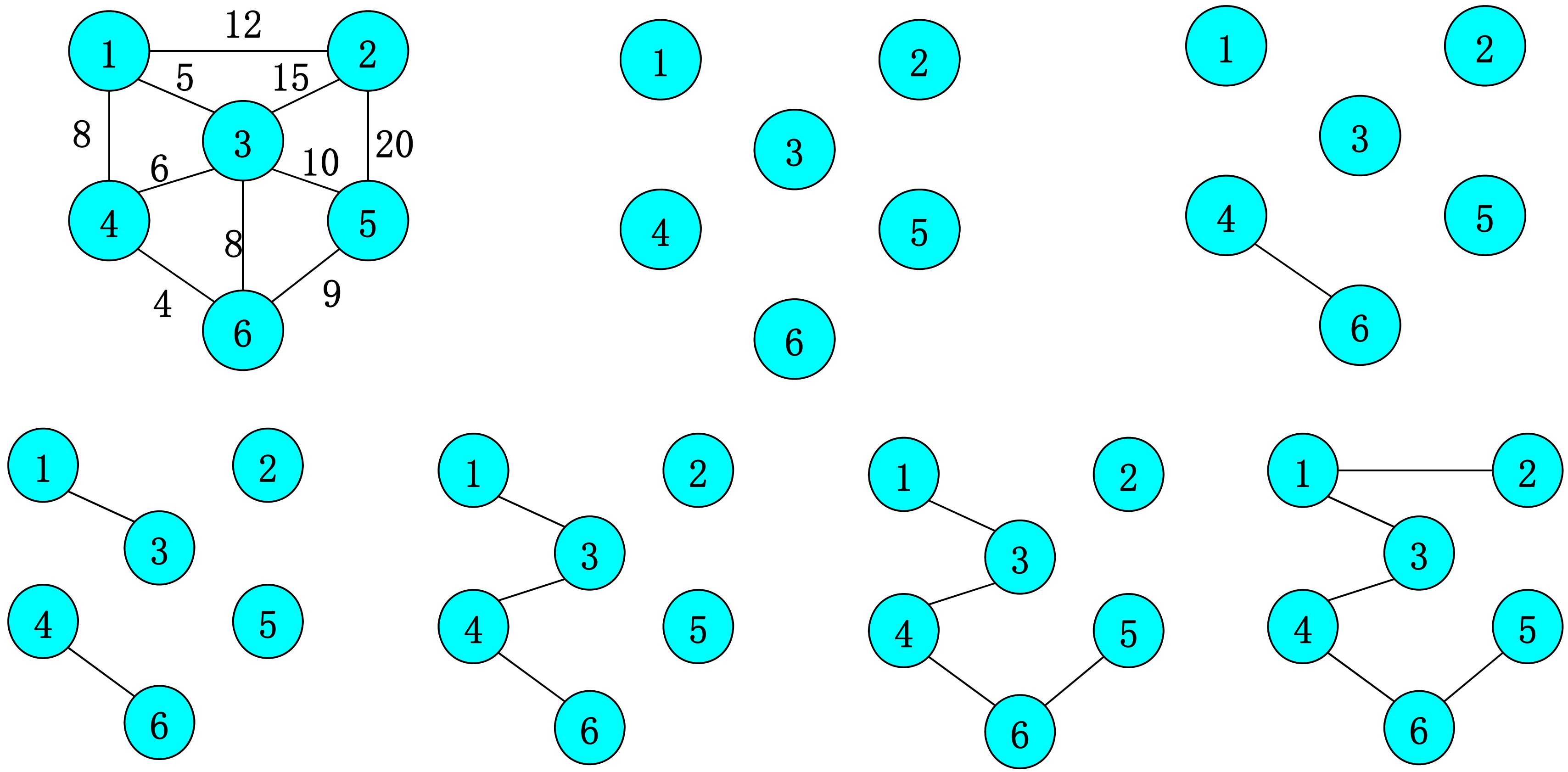
7.4. 图的连通性问题

7.4.3. 无向连通网的最小生成树

7.4.3.1. 基本概念

7.4.3.2. 普利姆(Prim)算法

7.4.3.3. 克鲁斯卡尔(Kruskal)算法



v1-v4：8舍弃
v3-v6：8舍弃

v3-v5：10舍弃
最小权值=36

§7. 图

7.4. 图的连通性问题

7.4.3. 无向连通网的最小生成树

7.4.3.1. 基本概念

7.4.3.2. 普利姆(Prim)算法

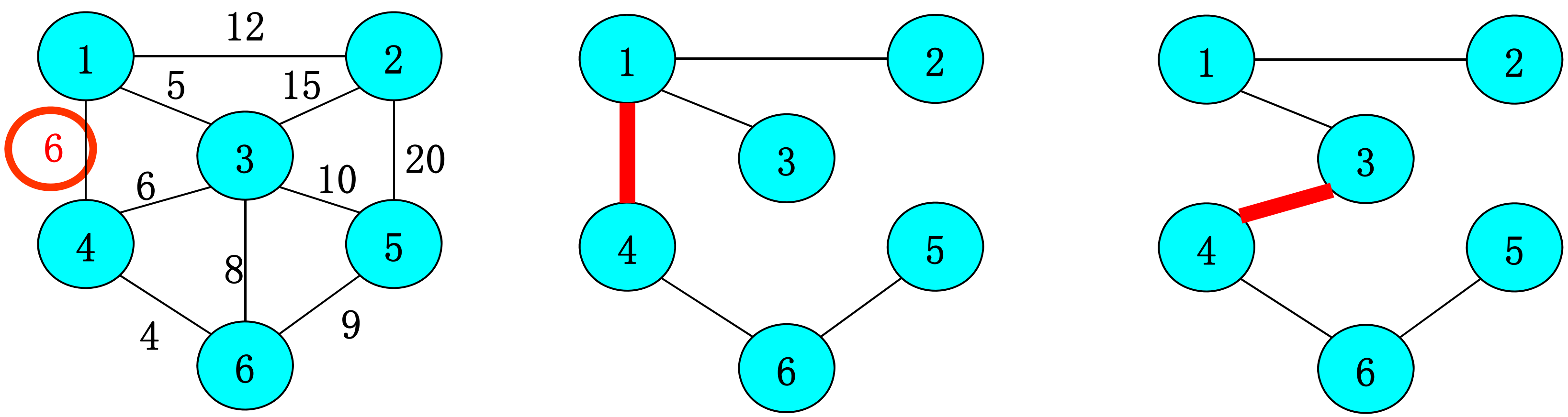
7.4.3.3. 克鲁斯卡尔(Kruskal)算法

★ 普里姆算法　　：(取决于顶点数 $O(n^2)$)

★ 克鲁斯卡尔算法：(取决于边数 O(eloge))

★ 若有最小权值相同的边，则任选一条

(最小生成树可能不唯一，但总代价应该相同，而且都是最小)



无论普利姆算法还是克鲁斯卡尔算法，都可以得到两种结果

§7. 图

7.4. 图的连通性问题

7.4.1. 无向图的连通分量和生成树

7.4.2. 有向图的强连通分量和有向生成树(略)

7.4.3. 无向连通网的最小生成树

7.4.4. 关节点和重连通分量(略)

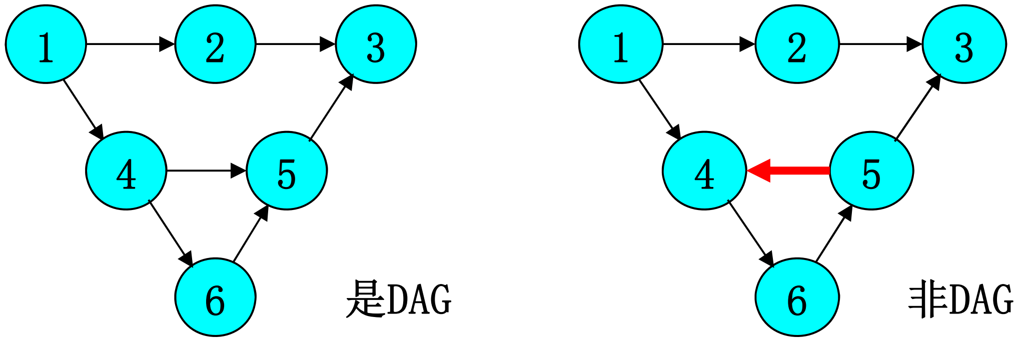
§7.图

7.5.有向无环图及其应用

7.5.1.基本概念

有向无环图：不存在任何回路的有向图称为有向无环图(DAG)

★ 不存在任一点的某个后继又是自身前驱的情况

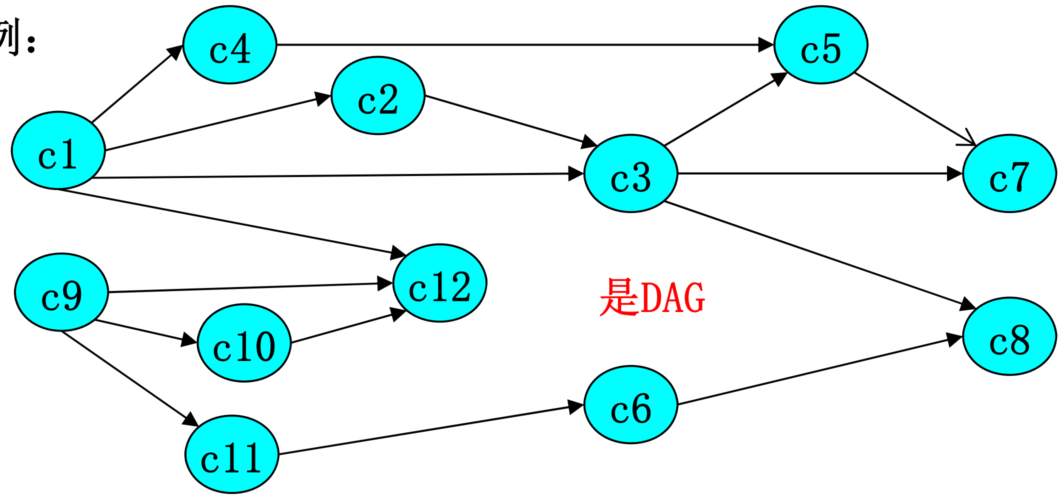


是DAG　　非DAG

AOV网：在有向图中，用顶点表示活动，用弧表示活动间的优先关系，则称该有向图是表示顶点活动的网(AOV网)

★ AOV网是有向无环图

例：



- 假设c1-c12分别表示建设一幢大楼所需要的工序（例如：c1-打地基 c5-安装门窗 ...），则该图表示工序之间的相互关系，同样必须是DAG
- 假设c1-c12分别表示申请一个科研项目所需要的步骤（例如：c1-写申请报告 c5-学校盖章 ...），则该图表示申请步骤之间的相互关系，同样必须是DAG

课程编号	课程名称	先修课程	课程编号	课程名称	先修课程
C1	程序设计基础	无	C7	编译原理	C3，C5
C2	离散数学	C1	C8	操作系统	C3，C6
C3	数据结构	C1，C2	C9	高等数学	无
C4	汇编语言	C1	C10	线性代数	C9
C5	语言的设计和分析	C3，C4	C11	普通物理	C9
C6	计算机原理	C11	C12	数值分析	C1，C9，C10

实际应用的含义：

★ 该图表达了课程之间的先后关系，排课时，有直接前驱后继关系的课程不能排在同一学期

★ 该图必须是DAG，假如是非DAG，则会出现不同课程间互为先修的错误

§7.图

7.5.有向无环图及其应用

7.5.1.基本概念

7.5.2.AOV网与拓扑排序

★ 基本概念

偏序：若集合X上的关系R符合自反、反对称、可传递，则称R是X上的偏序关系

全序：若R是X上的偏序关系，对任意x,y，均满足xRy和yRx，则成R是X上的全序关系

拓扑有序：全序关系称为拓扑有序

拓扑排序：由偏序定义得到的拓扑有序的操作称为拓扑排序

★ 拓扑排序的算法

① 从AOV网中选取一个入度为0的顶点并输出

② 删除该顶点及以之为尾的全部弧(该顶点出发)

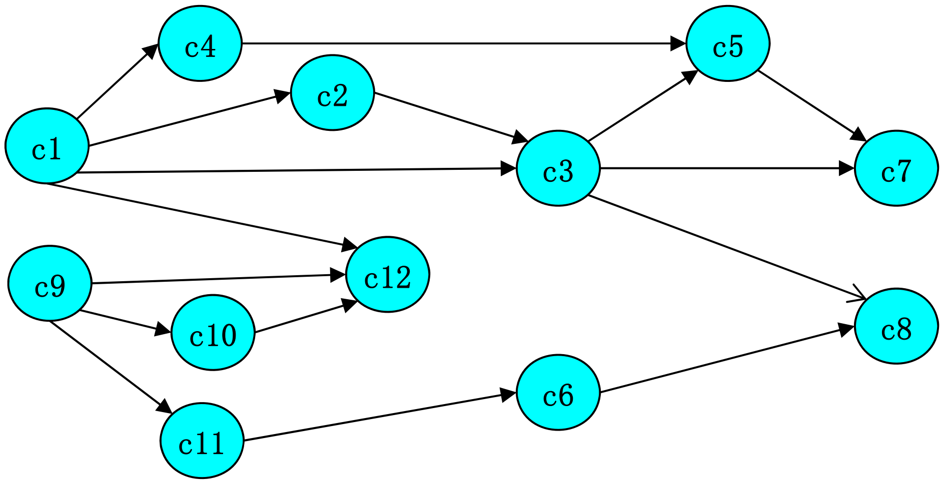
③ 重复执行①②，直到网中不存在入度为0的顶点为止，若顶点为空，则拓扑排序完成，否则出错

§7. 图

7.5. 有向无环图及其应用

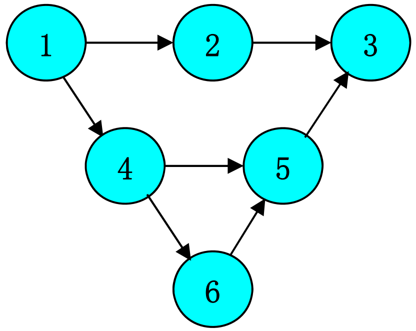
7.5.1. 基本概念

7.5.2. AOV网与拓扑排序



序列1：C1 C2 C3 C4 C5 C7 C9 C10 C11 C6 C12 C8

除此之外，还有很多拓扑有序的序列

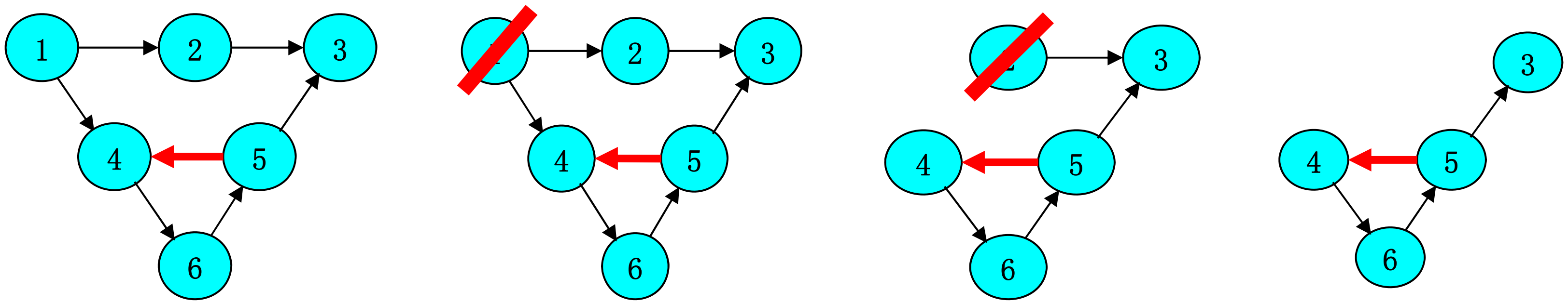


序列1：V1 V2 V4 V6 V5 V3

序列2：V1 V4 V2 V6 V5 V3

序列3：V1 V4 V6 V2 V5 V3

序列4：V1 V4 V6 V5 V2 V3



先输出：V1 V2　然后无法找到入度为0的结点，报错

§7. 图

7.5. 有向无环图及其应用

7.5.1. 基本概念

7.5.2. AOV网与拓扑排序

★ 拓扑排序算法的实现(P. 182 算法7. 12)

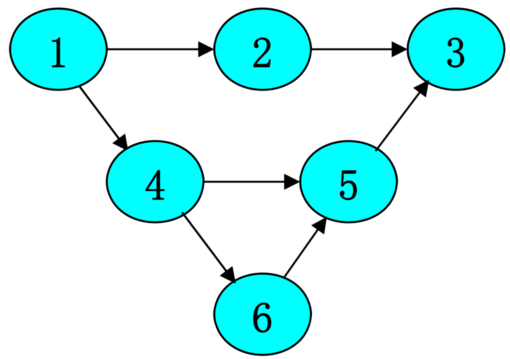
基本思路：

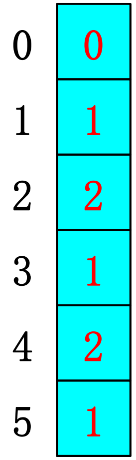
1、采用邻接表方式存储

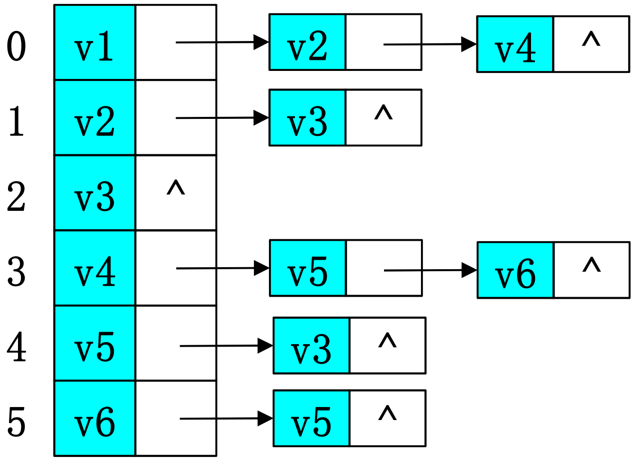
2、增加indegree数组，存放所有顶点的入度

3、反复查找入度为0的结点，输出该结点并更新其它顶点的入度

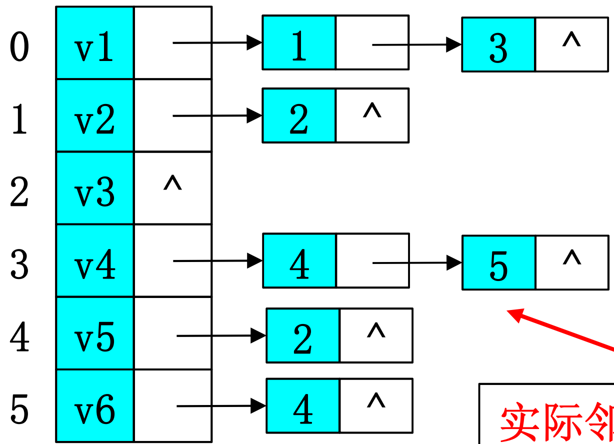
4、若找不到入度为0的顶点，则结束，再判断是否正确







邻接表的图示



实际邻接表的内容
当时为方便，表结点标示为顶点的值，实际是序号

★ 拓扑排序算法的实现(P.182 算法7.12)

思考：若换成队列，能得什么结果？

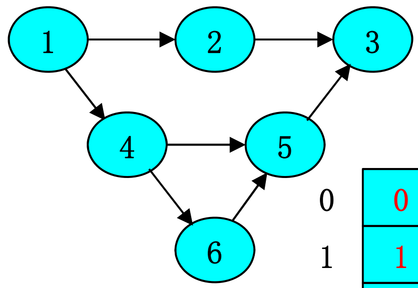
```
Status TopologicalSort(ALGraph G)
{
    FindInDegree(G, indegree);//计算头结点入度(另行实现)
    InitStack(S); //初始化辅助栈
    for (i=0; i<G.vexnum; i++)
        if (!indegree[i])
            Push(S, i);
    count = 0;  //输出顶点计数器
    while(!StackEmpty(S)) {
        Pop(S, i);
        cout << i << G.vertices[i].data; //输出顶点信息
        count ++;
        for(p=G.vertices[i].firstarc; p; p=p->nextarc) {
            k = p->adjvex;  //k为邻接点的下标
            if (!(--indegree[k]))
                Push(S, k);
        }
    }

    DestroyStack(S); //书上缺
    if (count < G.vexnum)
        return ERROR;  //输出顶点数<图的顶点数表示有错
    else
        return OK;
}
```

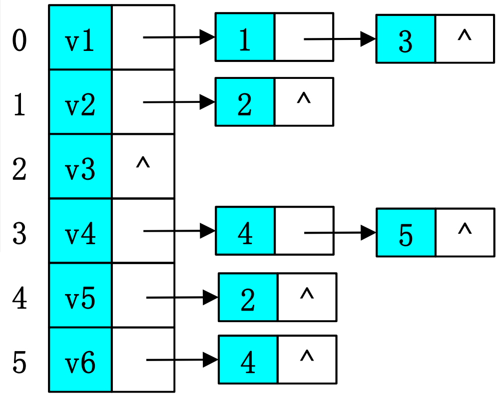
所有入度为0的顶点入栈
(引入栈的目的是提高效率，
不必反复检测入度为0的顶点)

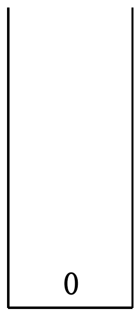
循环至栈空
(表示无入度为0的顶点)

循环该输出顶点的单链表
(对应该顶点的所有邻接点)，
相应的indegree数组-1，为0的入栈



0	0
1	1
2	2
3	1
4	2
5	1

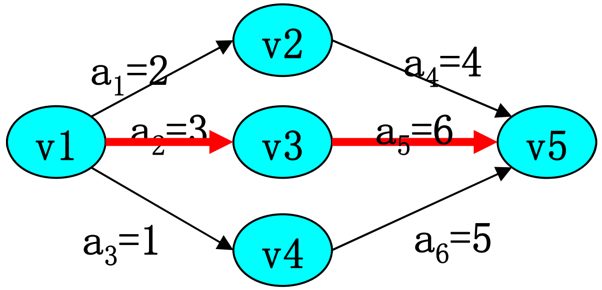




while执行前的状态
栈S中1个元素
count=0

给出程序执行过程中的输出、
栈及indegree数组的变化，
加深对算法的理解

§7. 图



7.5. 有向无环图及其应用

7.5.3. 关键路径

★ AOE网的含义

如果在带权有向无环图中用有向弧表示一个工程中的各项活动(或子工程)，用边上的权值表示活动的持续时间，用顶点表示事件，则把这样的有向图称为边表示活动的网络，简称AOE网络

假设活动如下：
a1：订购CPU所需时间
a2：订购内存所需时间
a3：订购硬盘所需时间
a4：测试并安装CPU时间
a5：测试并安装内存时间
a6：测试并安装硬盘时间

- 整个工程只有一个开始点和一个完成点
- 源点：AOE网中入度为0的点（开始点）
- 汇点：AOE网中出度为0的点（完成点）
- 各项活动可能顺序进行，也可能并行进行
- 从源点到汇点的有向路径可能不止一条，这些路径的长度(完成活动所需的时间)也可能不同,但只有各条路径上的所有活动都完成了，整个工程才算完成
- 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称为关键路径　(a2->a5是关键路径)
- 不按期完成就会影响整个工程完成的活动称为关键活动　(a2，a5是关键活动)
- 关键路径上的所有活动都是关键活动，只要找到关键活动，就能找到关键路径

★ AOE网的功能

AOE网络可用来估算工程的完成时间，例如，可以使人们了解：

(1) 完成整个工程至少需要多少时间？　(9天)

(2) 在整个工程的所有活动(子工程)中，哪些活动是影响工程进度的关键？　(a2, a5)

(3) 为了缩短整个工程所需的完成时间，应当加快哪些活动？　(a2, a5)

§7. 图

7.5. 有向无环图及其应用

7.5.3. 关键路径

★ AOE网的求解

(1) 如何求关键活动

e(i)：表示活动a_i的最早开始时间

l(i)：表示活动a_i的最晚开始时间

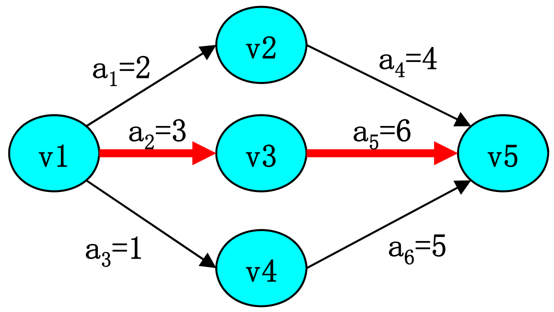
d(i)=l(i)-e(i)：完成活动a_i的时间余量

d(i)==0表示ai为关键活动

(即l(i)==e(i))

例：
e(a1)=0　l(a1)=3　d(a1)=3
e(a2)=0　l(a2)=0　d(a2)=0
a(a3)=0　l(a3)=3　d(a3)=3
e(a4)=2　l(a4)=5　d(a4)=3
e(a5)=3　l(a5)=3　d(a5)=0
e(a6)=1　l(a6)=4　d(a6)=3

a2,a5是关键活动，不能推迟
a1,a4不是关键活动，可推迟三天
a3,a6不是关键活动，可推迟三天



(2) 如何求e(i)、l(i)

假设活动ai由弧<j,k>(即事件j,k)表示，则将活动持续时间记为dut(<j, k>)

ve(j)：表示事件j的最早发生时间

vl(j)：表示事件j的最迟发生时间

则：e(i)=ve(j)：ai的最早时间=前驱的最早时间

l(i)=vl(k)-dut(<j,k>) ：ai的最晚时间=后继的最迟时间-持续时间

(3) 如何求ve(j)

从ve(源点)=0开始正向递推

ve(j) = MAX{ve(i)+dut(<i,j>)} <i,j>∈T，T是所有以j为头的弧的集合

例：
ve(v1)=0

ve(v2)=ve(v1)+2=2
ve(v3)=ve(v1)+3=3
ve(v4)=ve(v1)+1=1

ve(v5)是
　ve(v2)+4=6
　ve(v3)+6=9
　ve(v4)+5=6
中最大的一个 (9)

(4) 如何求vl(i)

从vl(汇点)开始逆向递推

vl(i) = MIN{vl(j)-dut(<i,j>)} <i,j>∈S，S是所有以i为尾的弧的集合

例：
vl(v5)=ve(v5)=9

vl(v4)=vl(v5)-5=4
vl(v3)=vl(v5)-6=3
vl(v2)=vl(v5)-4=5

vl(v1)是
　vl(v4)-1=3
　vl(v3)-3=0
　vl(v2)-2=3
中最小的一个 (0)

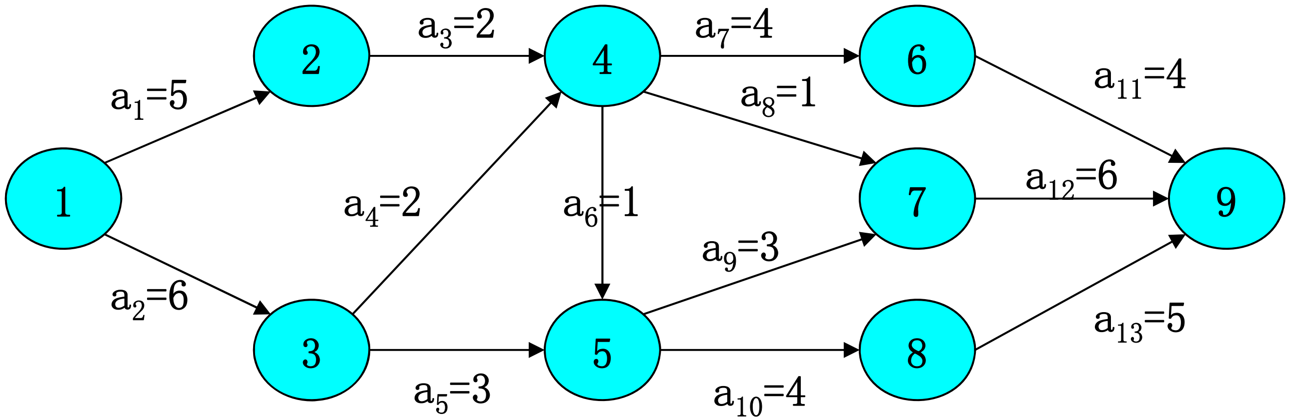
● ve(i)和vl(i)必须在拓扑有序及逆拓扑有序的情况下才能求解

例：如图所示AOE网，其中顶点代表工序，边的权值代表工序的完成时间，问：

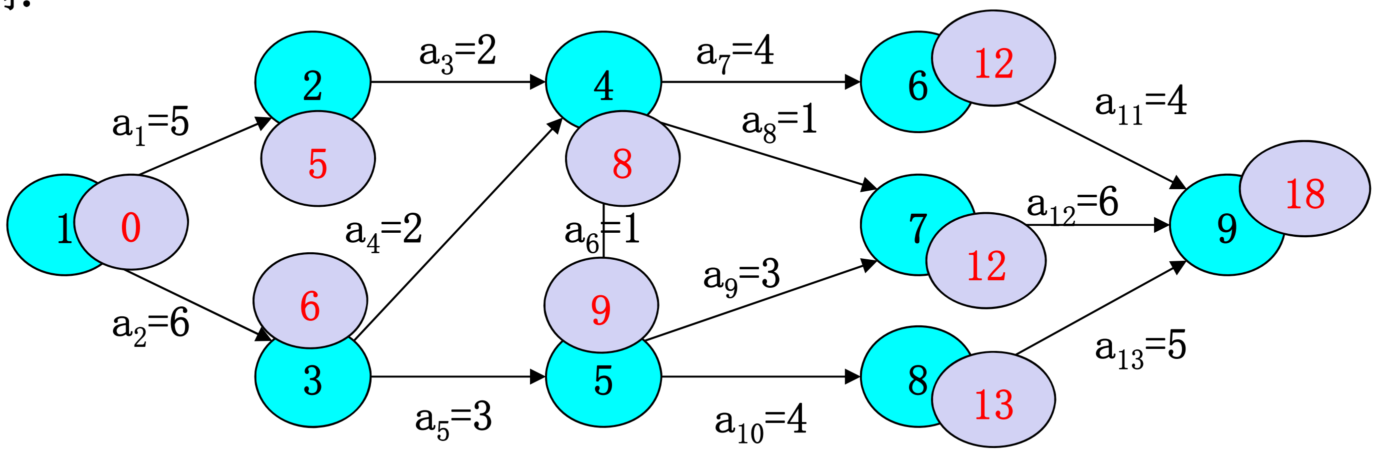
(1) 工程完成的最短时间是多少？

(2) 给出关键路径

(3) 哪些工序提高速度后能使整个工程缩短工期？



例：



首先计算各事件的ve

ve(1) = 0

ve(2) = ve(1)+a_1 = 5

ve(3) = ve(1)+a_2 = 6

ve(4) = MAX{ve(2)+a_3, ve(3)+a_4} = MAX{5+2, 6+2} = 8

ve(5) = MAX{ve(3)+a_5, ve(4)+a_6} = MAX{6+3, 8+1} = 9

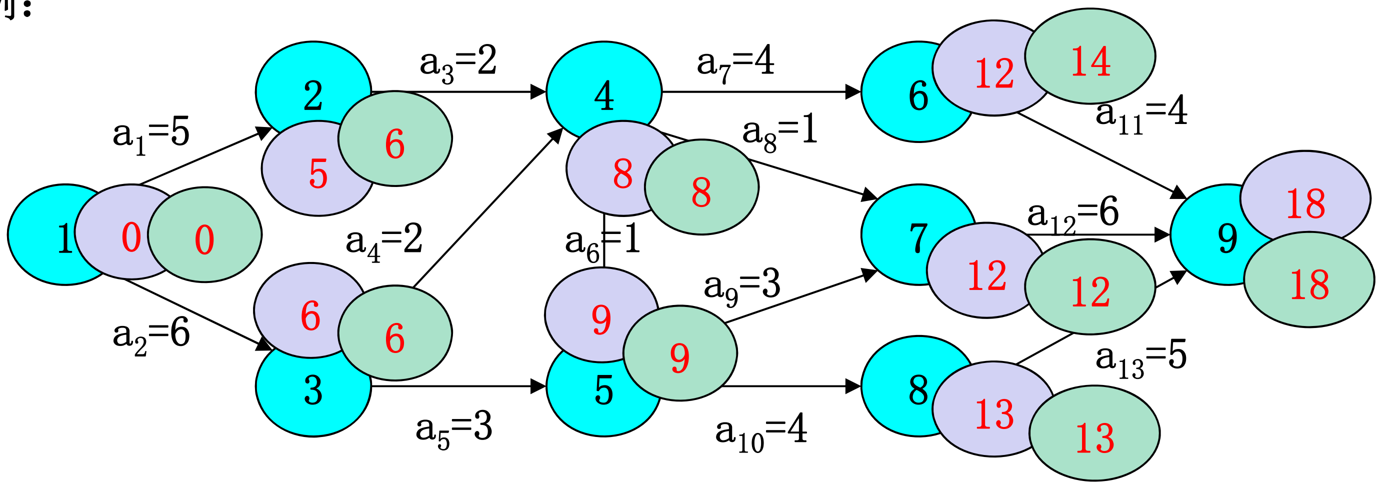
ve(6) = ve(4)+a_7 = 12

ve(7) = MAX{ve(4)+a_8, ve(5)+a_9} = MAX{8+1, 9+3} = 12

ve(8) = ve(5)+a_{10} = 13

ve(9) = MAX{ve(6)+a_{11}, ve(7)+a_{12}, ve(8)+a_{13}}=MAX{12+4, 12+6, 13+5} = 18

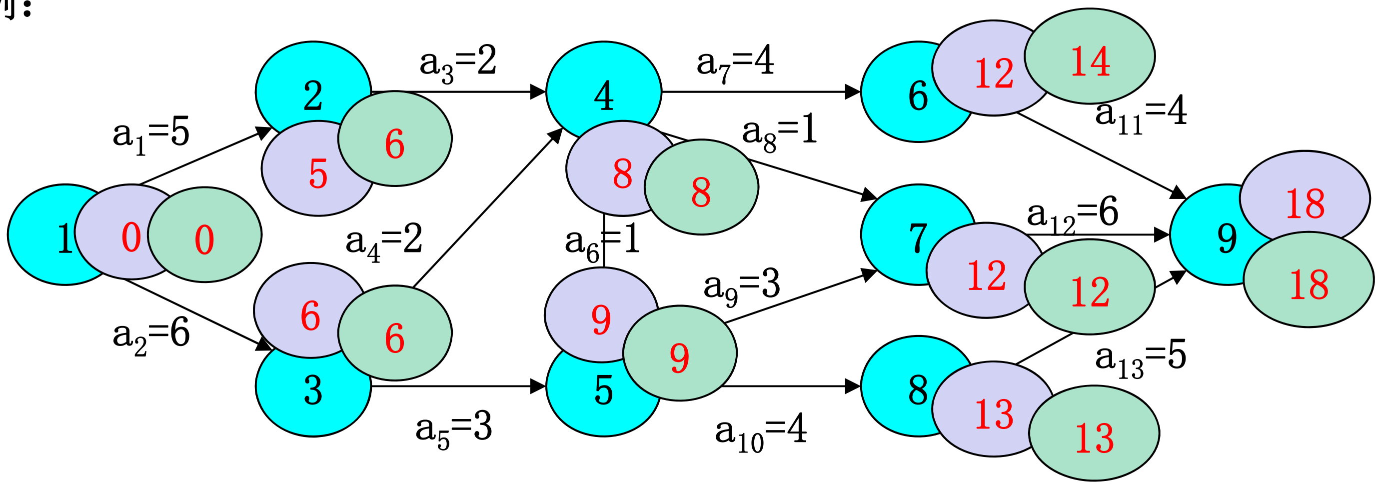
例：



其次计算各事件的vl

vl(9) = 18
vl(8) = vl(9)−a_{13} = 18−5 = 13
vl(7) = vl(9)−a_{12} = 18−6 = 12
vl(6) = vl(9)−a_{11} = 18−4 = 14
vl(5) = MIN{vl(8)−a_{10}, vl(7)−a_9} = MIN{13−4, 12−3} = 9
vl(4) = MIN{vl(7)−a_8, vl(6)−a_7, vl(5)−a_6} = MIN{12−1, 14−4, 9−1} = 8
vl(3) = MIN{vl(5)−a_5, vl(4)−a_4} = MIN{9−3, 8−2} = 6
vl(2) = vl(4)−a_3 = 8−2 = 6
vl(1) = MIN{vl(3)−a_2, vl(2)−a_1} = MIN{6−6, 6−5} = 0

例：



再计算各活动的e、l、d

e(a1)=ve(1)=0	l(a1)=vl(2)-a1=6-5=1	d(a1)=1	
e(a2)=ve(1)=0	l(a2)=vl(3)-a2=6-6=0	d(a2)=0	✓
e(a3)=ve(2)=5	l(a3)=vl(4)-a3=8-2=6	d(a3)=1	
e(a4)=ve(3)=6	l(a4)=vl(4)-a4=8-2=6	d(a4)=0	✓
e(a5)=ve(3)=6	l(a5)=vl(5)-a5=9-3=6	d(a5)=0	✓
e(a6)=ve(4)=8	l(a6)=vl(5)-a6=9-1=8	d(a6)=0	✓
e(a7)=ve(4)=8	l(a7)=vl(6)-a7=14-4=10	d(a7)=2	
e(a8)=ve(4)=8	l(a8)=vl(7)-a8=12-1=11	d(a8)=3	
e(a9)=ve(5)=9	l(a9)=vl(7)-a9=12-3=9	d(a9)=0	✓
e(a10)=ve(5)=9	l(a10)=vl(8)-a10=13-4=9	d(a10)=0	✓
e(a11)=ve(6)=12	l(a11)=vl(9)-a11=18-4=14	d(a11)=2	
e(a12)=ve(7)=12	l(a12)=vl(9)-a12=18-6=12	d(a12)=0	✓
e(a13)=ve(8)=13	l(a13)=vl(9)-a13=18-5=13	d(a13)=0	✓

例：如图所示AOE网，其中顶点代表工序，边的权值代表工序的完成时间，问：

（1）工程完成的最短时间是多少？ （18）

（2）给出关键路径

关键活动 a2 a4 a5 a6 a9 a10 a12 a13

关键路径 a2->a5->a9->a12

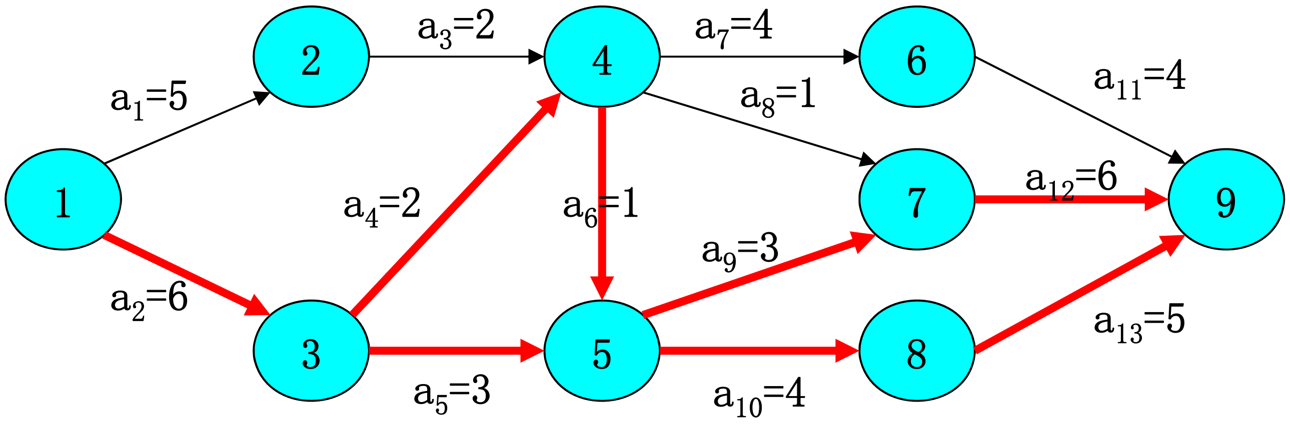
a2->a5->a10->a13

a2->a4->a6->a9->a12

a2->a4->a6->a10->a13

（3）哪些工序提高速度后能使整个工程缩短工期？

所有关键路径中，共同关键活动是a2，通过提高a2的速度来缩短整个工程的周期



§7.图

7.5.有向无环图及其应用

7.5.3.关键路径

★ AOE网的含义

★ AOE网的功能

★ AOE网的求解

★ 求关键路径的算法

★ 求关键路径的算法(P.185 算法7.13)

```
Status TopologicalOrder(ALGraph G, stack &T)
{   FindInDegree(G, indegree); //求各顶点的入度
    InitStack(S); //建零入度栈(同算法7.12)
    for (i=0; i<G.vexnum; i++)
        if (!indegree[i])
            Push(S, i);
    InitStack(T);  //初始化用于存放拓扑排序序列的栈
    count = 0;
    for (i=0; i<G.vexnum; i++)      书上：ve[0..G.vexnum-1] = 0;
        ve[i] = 0;
    while(!StackEmpty(S)) {
        Pop(S, j);  //从零入度栈出栈
        Push(T, j); //进拓扑排序结果栈
        count ++;
        for(p=G.vertices[j].firstarc; p; p=p->nextarc) {
            k = p->adjvex;
            if (--indegree[k]==0)
                Push(S, k);
            if (ve[j]+ *(p->info) > ve[k])
                ve[k] = ve[j] + *(p->info);
        }
    }                                    黑色部分同算法7.12
    /* 最后部分同算法7.12 */
    DestroyStack(S); //书上缺
    if (count < G.vexnum)
        return ERROR;
    else
        return OK;
}
```

1、p->info中存储了边的权值 (参见图存储)

2、<j,k>正好是弧

3、循环后，ve就是max

★ 求关键路径的算法(P.185 算法7.14)

```
Status CriticalPath(ALGraph G)
{   /* 调用算法7.13       已求得ve ： 假设ve是全局变量中
                          已求得拓扑有序序列：在T中，T的出栈顺序为逆拓扑 */
    if (!TopologicalOrder(G, T)
        return ERROR;
    /* 先给vl赋初值，书上形式为 vl[0..G.vexnum-1] = ve[G.vexnum-1]; */
    for (i=0; i<G.vexnum; i++)
        vl[i] = ve[G.vexnum-1];

    while(!StackEmpty(T)) //while无括号, T出栈顺序逆拓扑
        for(Pop(T, j),p=G.vertices[j].firstarc; p; p=p->nextarc) {
            k = p->adjvex;
            dut = *(p->info);  //取边的权值
            if (vl[k]-dut < vl[j])
                vl[j] = vl[k]-dut;
            }
    for (j=0; j<G.vexnum; j++) //外层循环无括号
        for(p=G.vertices[j].firstarc; p; p=p->nextarc) {
            k = p->adjvex;
            dut = *(p->info);
            ee = ve[j];         //e(j)
            el = vl[k] - dut;   //l(j)
            tag = (ee==el) ? '*':''; //判断是否关键活动
            /* 输出每条边(每个活动)，关键活动加*标记 */
            cout << j << k << dut << ee << el << tag;
            }
    DestroyStack(T); //书上缺，栈T在TopologicalOrder中初始化
    return OK; //书上无
}
```

§7.图

7.6.有向网的最短路径

7.6.1.基本概念

边带权的有向图，从源点到终点权值最小的路径

7.6.2.迪杰斯特拉(dijstra)算法

求单源最短路径，即从给定的某个源点出发到其它顶点的最短路径

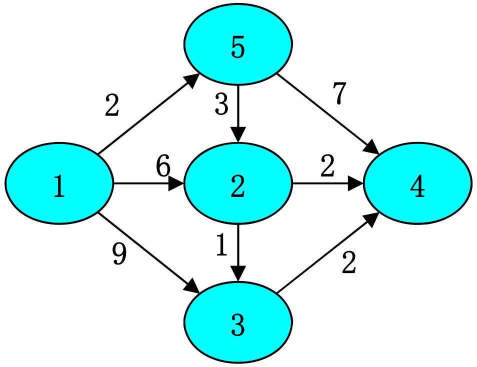
★ 算法描述

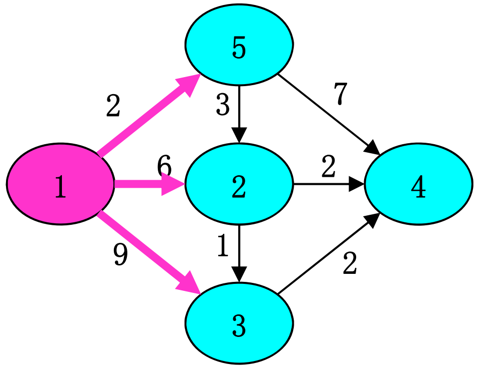
① 将顶点分为两个集合，已确定最短路径的集合S及未确定最短路径的集合V，初始状态源点在S中，其余在V中

② 计算源点到V中各顶点的最短路径，从V中选取与源点路径最短的顶点，加入S中

③ 重复执行②，直到全部顶点加入S中为止

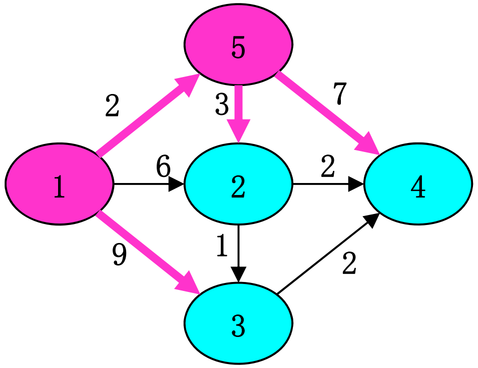
例：求V1到其它各点的最短路径



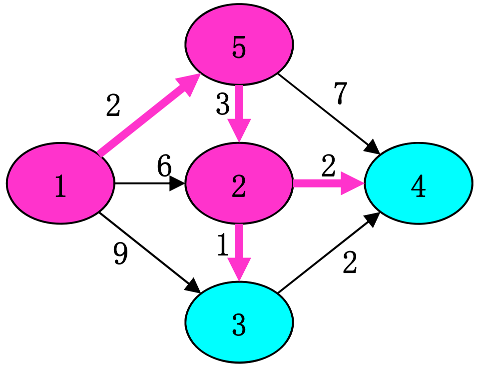


初始 S={v1}
V={v2, v3, v4, v5}
求源点到各点的路径：
v1-v2：6 (1,2)
v1-v3：9 (1,3)
v1-v4：/ (暂无路径)
v1-v5：2 (1,5) (选)

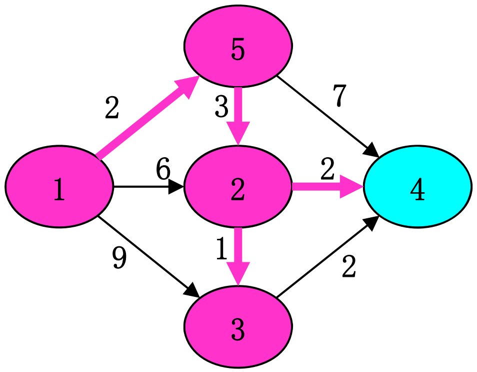
V中4个顶点，循环4次



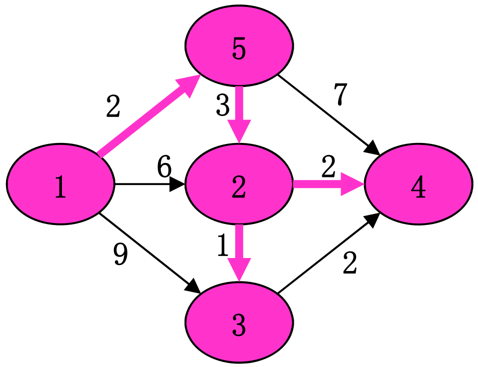
变化 S={v1, v5}
V={v2, v3, v4}
求源点到各点的路径：
原 新
v1-v2：6 5 (1,5,2)(选)
v1-v3：9 9 (1,3)
v1-v4：/ 9 (1,5,4)
v1-v5：2 (1,5)(不参与)



变化 S={v1, v2, v5}
V={v3, v4}
求源点到各点的路径：
原 新
v1-v2：5 (1,5,2)(不参与)
v1-v3：9 6 (1,5,2,3)(选)
v1-v4：9 7 (1,5,2,4)
v1-v5：2 (1,5)(不参与)



变化 S={v1, v2, v3, v5}
V={v4}
求源点到各点的路径：
原 新
v1-v2：5 (1,5,2)(不参与)
v1-v3：6 (1,5,2,3)(不参与)
v1-v4：7 7 (1,5,2,4)(选)
v1-v5：2 (1,5)(不参与)



变化 S={v1,v2,v3,v4,v5}
V={} //结束
求源点到各点的路径：
原 新
v1-v2：5 (1,5,2)(不参与)
v1-v3：6 (1,5,2,3)(不参与)
v1-v4：7 (1,5,2,4)
v1-v5：2 (1,5)(不参与)

§7. 图

7.6.有向网的最短路径

7.6.1.基本概念

7.6.2.迪杰斯特拉(dijstra)算法

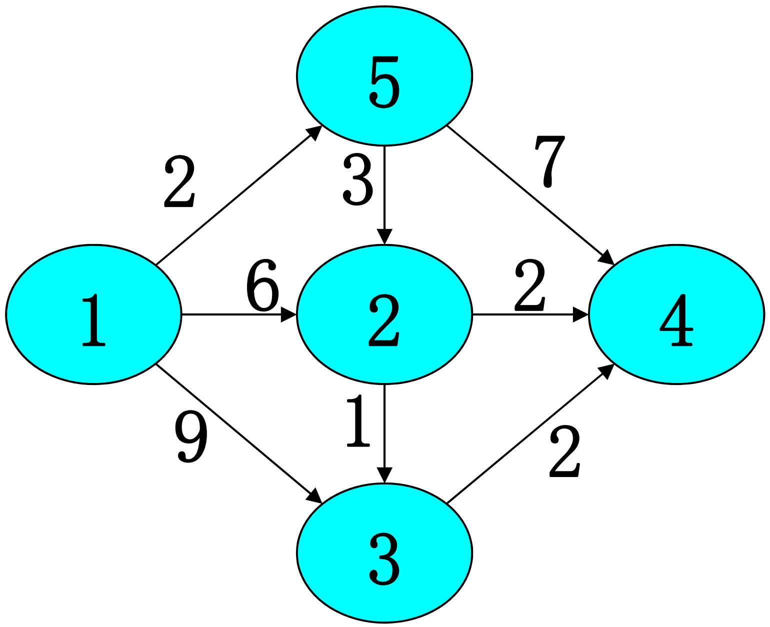
★ 算法描述

★ 算法实现

- 引进一维数组D，每个D[i]表示当前找到的从起始点v到每个终点vi的最短路径长度
- 引进一维数组final，每个元素初始为FALSE，若求完最短路径则置为TRUE
- 引进二维数组P，记录最短路径对应的顶点序列

§7.图

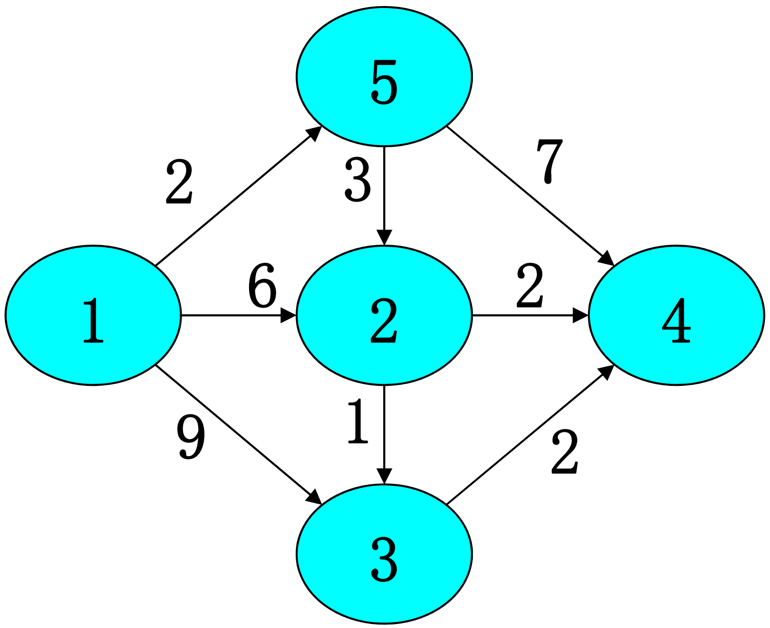
例：求V1到其它各点的最短路径



		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	/	6	9	/	2
1	v2	/	/	1	2	/
2	v3	/	/	/	2	/
3	v4	/	/	/	/	/
4	v5	/	3	/	7	/

源点为v1，初始化操作如下：

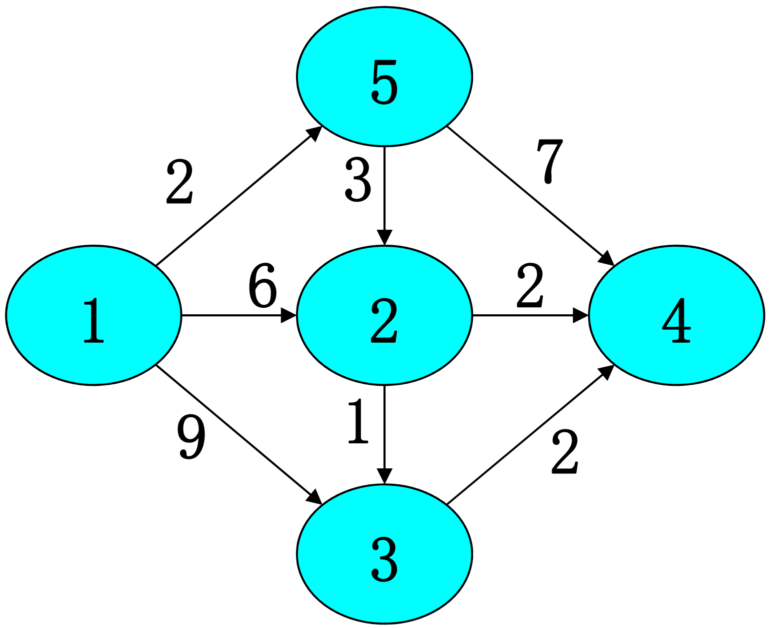
	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组					
D数组					



		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	/	6	9	/	2
1	v2	/	/	1	2	/
2	v3	/	/	/	2	/
3	v4	/	/	/	/	/
4	v5	/	3	/	7	/

源点为v1，初始化操作如下：
① final全部为FALSE(v1对应的final为TRUE)

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组					



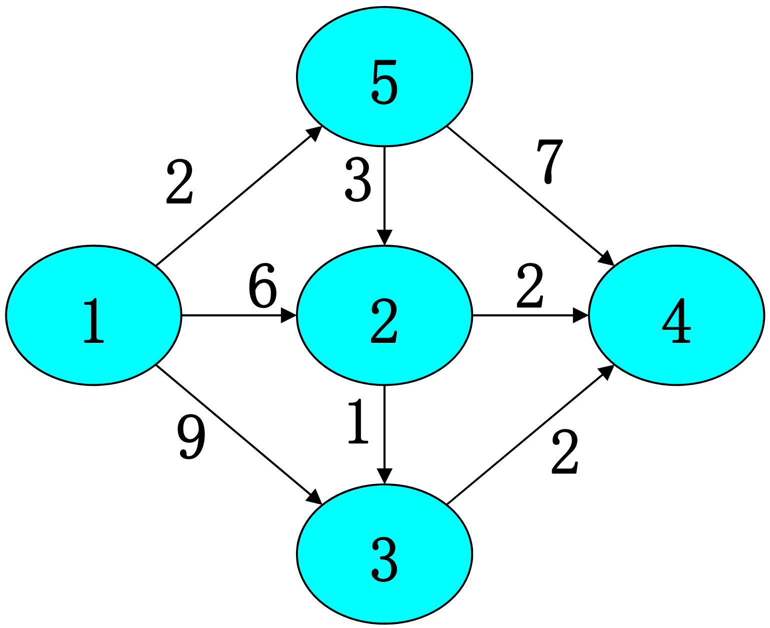
		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	/	6	9	/	2
1	v2	/	/	1	2	/
2	v3	/	/	/	2	/
3	v4	/	/	/	/	/
4	v5	/	3	/	7	/

源点为v1，初始化操作如下：

① final全部为FALSE(v1对应的final为TRUE)

② D为邻接矩阵第v1行的对应内容(v1对应的D为0)

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组	0	6	9	INT_MAX	2



$$
\begin{array}{cc|ccccc}
 & & 0 & 1 & 2 & 3 & 4 \\
 & & v1 & v2 & v3 & v4 & v5 \\
\hline
0 & v1 & / & 6 & 9 & / & 2 \\
1 & v2 & / & / & 1 & 2 & / \\
2 & v3 & / & / & / & 2 & / \\
3 & v4 & / & / & / & / & / \\
4 & v5 & / & 3 & / & 7 & /
\end{array}
$$

源点为v1，初始化操作如下：

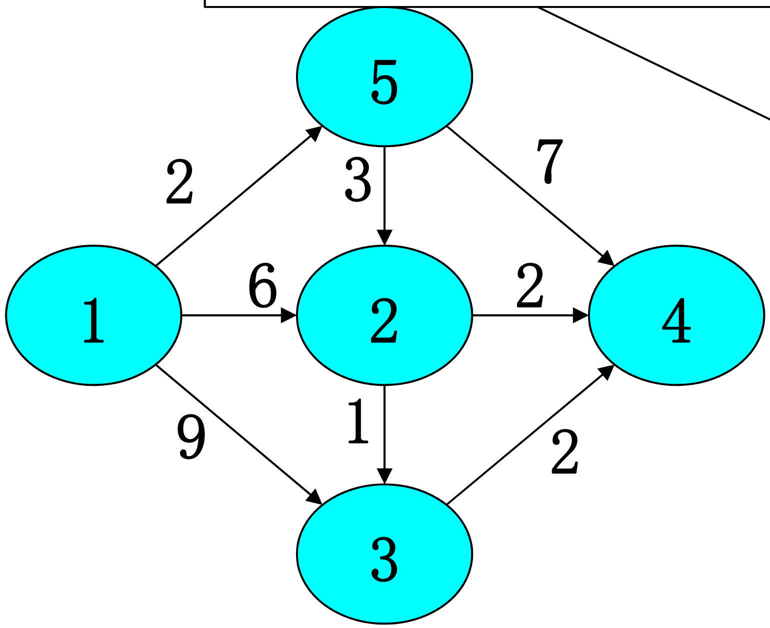
① final全部为FALSE(v1对应的final为TRUE)

② D为邻接矩阵第v1行的对应内容(v1对应的D为0)

③ P中v1及被v1指向的行,其v1列及主对角线为TRUE,其余FALSE

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组	0	6	9	INT_MAX	2

为方便看清，F全为/



P=

		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

源点为v1，初始化操作如下：

① final全部为FALSE(v1对应的final为TRUE)

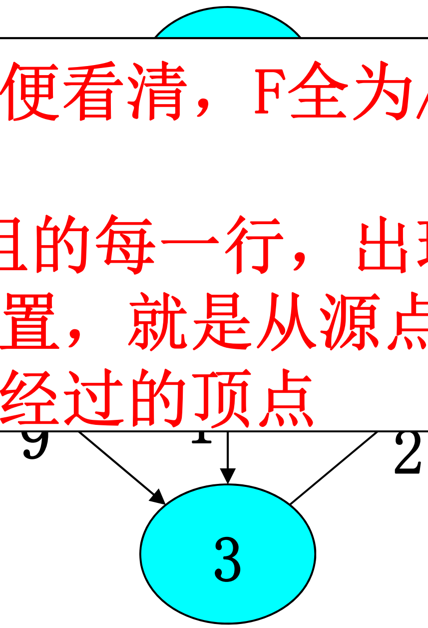
② D为邻接矩阵第v1行的对应内容(v1对应的D为0)

③ P中v1及被v1指向的行，其v1列及主对角线为TRUE，其余FALSE

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组	0	6	9	INT_MAX	2

为方便看清，F全为/

P数组的每一行，出现TRUE的位置，就是从源点到该点所经过的顶点



P=

		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

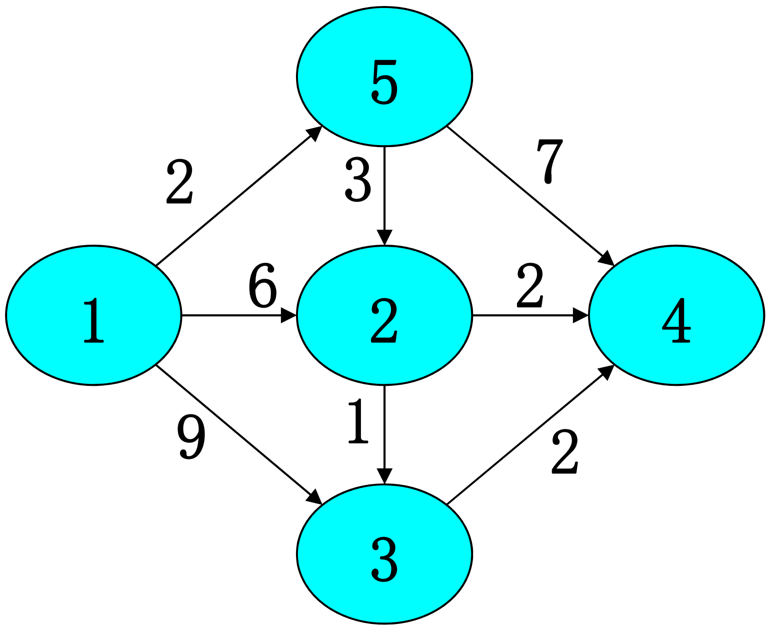
进行i=1-4的循环，每次

① 在D中找最小值min(只找对应final位置为FALSE的)及下标v

② 置对应final为TRUE(下次不再查找)

③ 循环D数组w=0-4，若对应位置final为FLASE且
(D[w] < min+G.arcs[v][w]) 则更新D、P数组

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组	0	6	9	INT_MAX	2

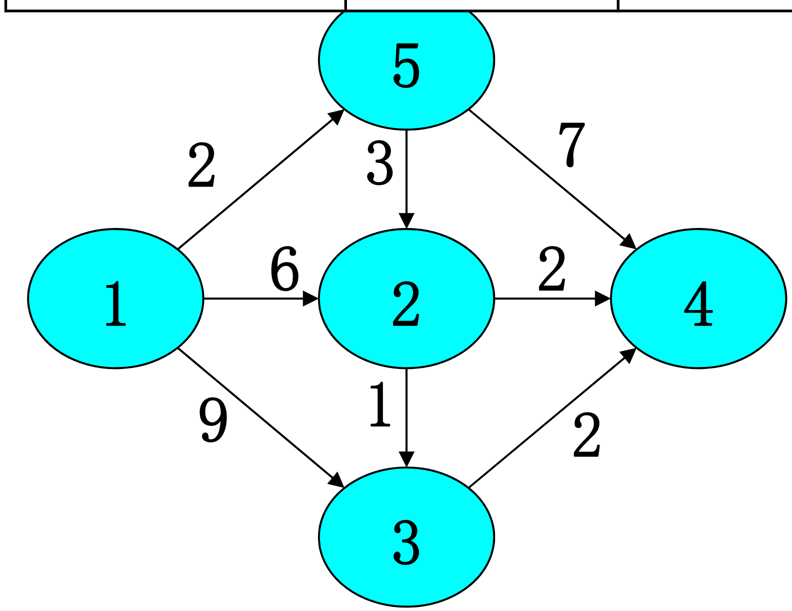


P=

		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

i=1: ① 在D的([1][2][3][4])中找最小值，得 v=4(v5) min=2

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
旧 D数组	0	6	9	INT_MAX	2
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
新 D数组	0	6	9	INT_MAX	2



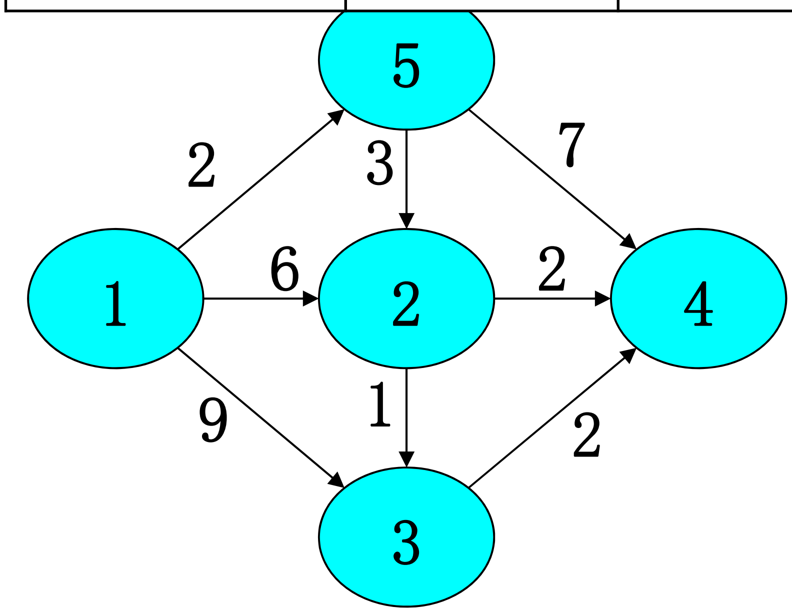
P=

		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

i=1: ① 在D的([1][2][3][4])中找最小值，得 v=4(v5) min=2

② final[4] = TRUE

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组 旧	0	6	9	INT_MAX	2
final数组	TRUE	FALSE	FALSE	FALSE	TRUE
D数组 新	0	6	9	INT_MAX	2



P=

		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

i=1: ③ 循环D(w=[1][2][3])，min+第v行对应位置权值<D[w]
则更新D、P数组

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组 旧	0	6	9	INT_MAX	2
final数组	TRUE	FALSE	FALSE	FALSE	TRUE
D数组 新	0	2+3	9	2+7	2

		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	/	6	9	/	2
1	v2	/	/	1	2	/
2	v3	/	/	/	2	/
3	v4	/	/	/	/	/
4	v5	/	3	/	7	/

P=

		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	/
2	v3	T	/	T	/	/
3	v4	/	/	/	/	/
4	v5	T	/	/	/	T

i=1：③ 循环D(w=[1][2][3])，min+第v行对应位置权值<D[w]
则更新D、P数组

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	FALSE
D数组 旧	0	6	9	INT_MAX	2
final数组	TRUE	FALSE	FALSE	FALSE	TRUE
D数组 新	0	2+3	9	2+7	2

P数组的更新方法：
将第v行复制到第w行
(本例:v5行复制到v2、v4)
第w行的P[w][w]置TRUE
(本例：P[1][1]、P[3][3])

P=

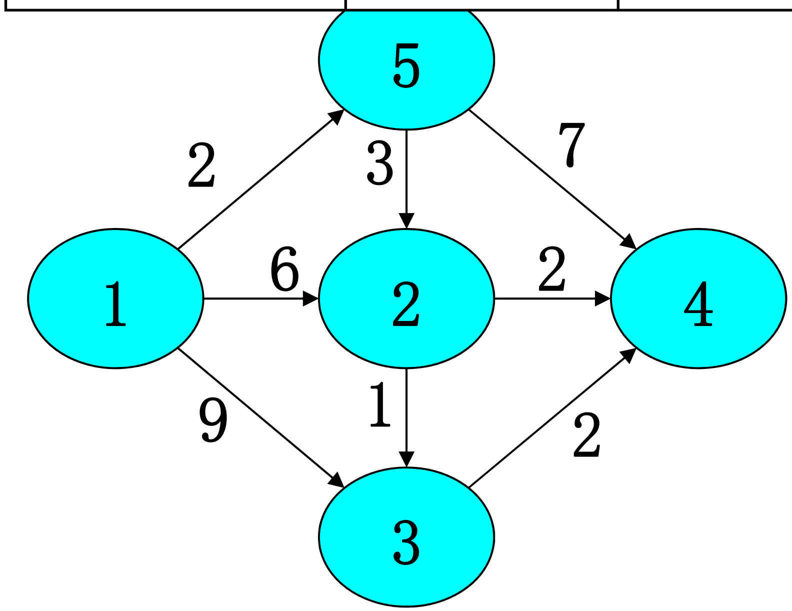
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	T
2	v3	T	/	T	/	/
3	v4	T	/	/	T	T
4	v5	T	/	/	/	T

i=2: ① 在D的([1][2][3])中找最小值，得 v=1(v2) min=5

② final[1] = TRUE

③ 循环D(w=[2][3]),min+第v行对应位置权值<D[w]则更新

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	FALSE	FALSE	FALSE	TRUE
D数组 旧	0	5	9	9	2
final数组	TRUE	TRUE	FALSE	FALSE	TRUE
D数组 新	0	5	5+1	5+2	2



P=

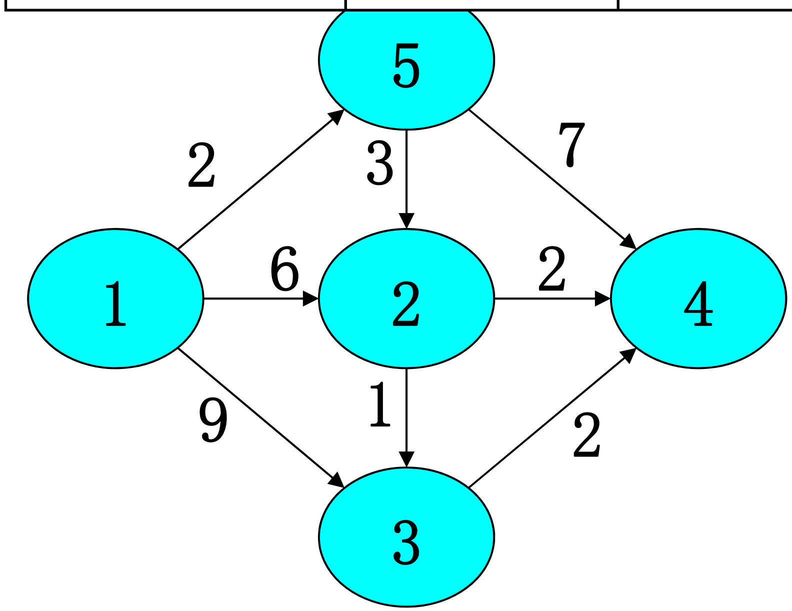
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	T
2	v3	T	T	T	/	T
3	v4	T	T	/	T	T
4	v5	T	/	/	/	T

i=3: ① 在D的([2][3])中找最小值，得 v=2(v3) min=6

② final[2] = TRUE

③ 循环D(w=[3]),min+第v行对应位置权值<D[w]则更新(无)

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	TRUE	FALSE	FALSE	TRUE
旧 D数组	0	5	6	7	2
final数组	TRUE	TRUE	TRUE	FALSE	TRUE
新 D数组	0	5	6	7	2



P=

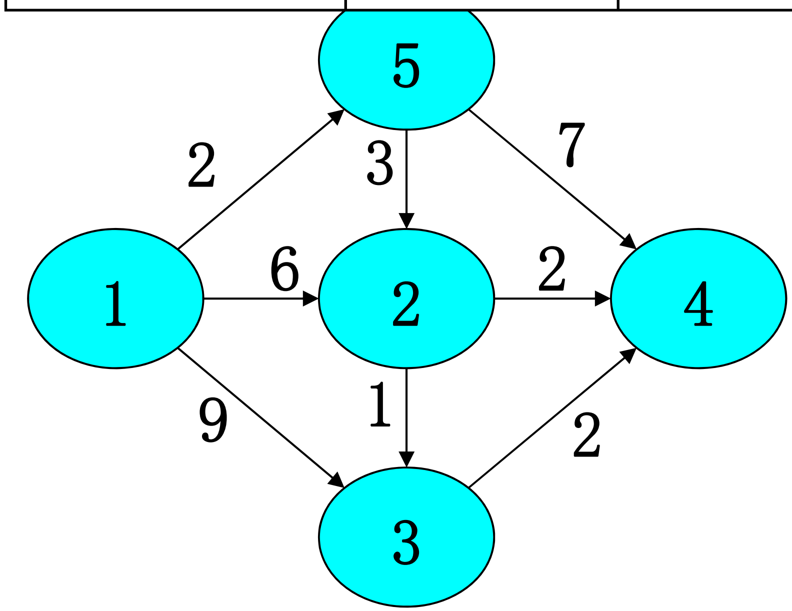
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	T
2	v3	T	T	T	/	T
3	v4	T	T	/	T	T
4	v5	T	/	/	/	T

i=4: ① 在D的([3])中找最小值，得 v=3(v4) min=7

② final[3] = TRUE

③ 循环D(w=空),min+第v行对应位置权值<D[w]则更新(无)

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	TRUE	TRUE	FALSE	TRUE
D数组 旧	0	5	6	7	2
final数组	TRUE	TRUE	TRUE	TRUE	TRUE
D数组 新	0	5	6	7	2

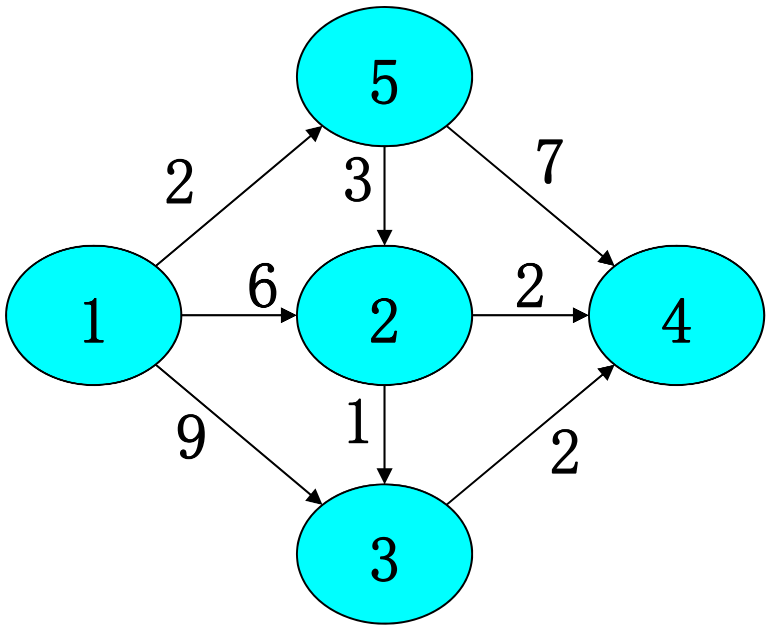


P=

		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	T
2	v3	T	T	T	/	T
3	v4	T	T	/	T	T
4	v5	T	/	/	/	T

循环结束，D数组的终值就是源点到各点的最短路径
P数组的终值就是源点到各点经过的顶点集合

	0(v1)	1(v2)	2(v3)	3(v4)	4(v5)
final数组	TRUE	TRUE	TRUE	TRUE	TRUE
D数组	0	5	6	7	2



P=

		0	1	2	3	4
		v1	v2	v3	v4	v5
0	v1	T	/	/	/	/
1	v2	T	T	/	/	T
2	v3	T	T	T	/	T
3	v4	T	T	/	T	T
4	v5	T	/	/	/	T

★ 迪杰斯特拉算法的实现(P. 189 算法7. 15 邻接矩阵)

```
void ShortestPath_DIJ(MGraph G, int v0, PathMatrix &P, shortPathTable &D)
{   /* 所有顶点进行初始化 */
    for(v=0; v<G.vexnum; v++) {
        final[v] = FALSE;   //标记数组置为未访问
        D[v]=G.arcs[v0][v]; //D的初值就是从v0到各点的权
        for (w=0; w<G.vexnum; w++)
            P[v][w] = FALSE; //P数组第v行全部FALSE
        if (D[v] < INFINITY) {
            P[v][v0] = TRUE;            若v0到v有边，则
            P[v][v]  = TRUE;            P数组的第v行v0列
            }                           及第v行第v列置TRUE
        }
    /* 源点v0做特殊处理 */
    D[v0]     = 0;      //最短路径=0
    final[v0] = TRUE;   //已计算出最短路径
    for (i=1; i<G.vexnum; i++) {  //循环剩余的顶点
        min = INFINITY;
        for (w=0; w<G.vexnum; w++)
            if (!final[w])
                if (D[w]<min) {             D数组找最小值
                    v = w;                  要求对应final为FALSE
                    min = D[w];             内循环for(w)结束后
                    }                       v是找到的下标
                                            min是找到的最小值
        final[v] = TRUE;
        /* 循环所有顶点，看final为FALSE的是否要更新*/
        for(w=0; w<G.vexnum; w++)
            if (!final[w] && (min+G.arcs[v][w]<D[w])) {
                D[w] = min + G.arcs[v][w]; //更新D[w]
                P[w] = P[v];      //整行复制（此处示例，实际应该循环）
                P[w][w] = TRUE;   //把自己(w行w列)加进去
                }
        }
}
```