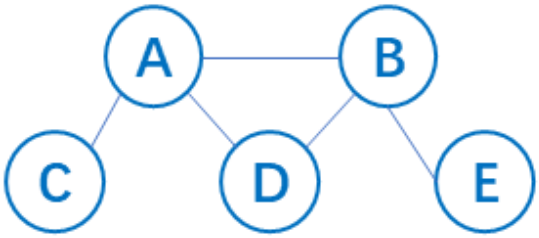
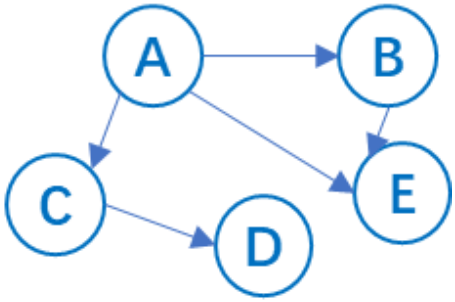
# 第六章 图

## 6.1 图的基本概念

### 6.1.1 图的定义




图由顶点集$V$和边集$E$组成，记为$G=(V,E)$，其中$V(G)$表示图$G$中顶点的有线非空集。$E(G)$表示图G中顶点间的关系（边）的集合。若

$$V=\{v_1,v_2,\dots,v_n\}$$

则使用$|V|$来表示图$G$中顶点的个数，

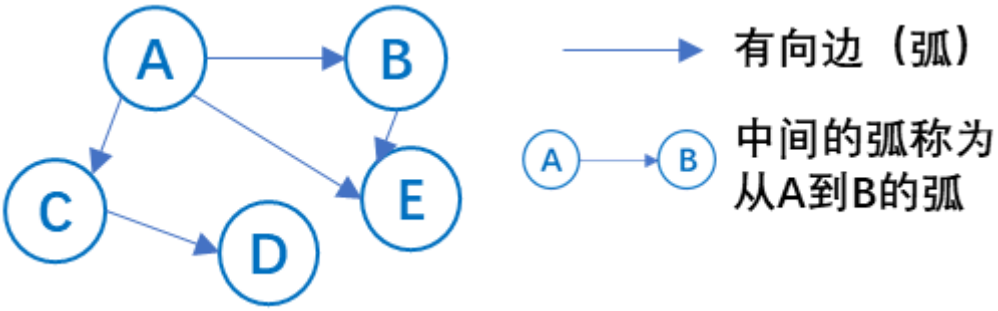
$$E=\{(u,v)|u\in V,E\in V\}$$

则使用$|E|$来表示图G中边的条数。

> 【注意】线性表可以是空表，树可以是空树，但**图不能是空图**。图的顶点集V一定非空，但是边集合E可以为空，此时图中只有顶点没有边。

**1、有向图**

若E是有向边（也称弧）的有限集合时，则图G是**有向图**。弧是顶点的有序对，记为$<v,w>$，其中v,w是顶点，v称为弧尾，w称为弧头，$<v,w>$称为从v到w的弧，也称v邻接到w。




上图所示的有向图可以表示为

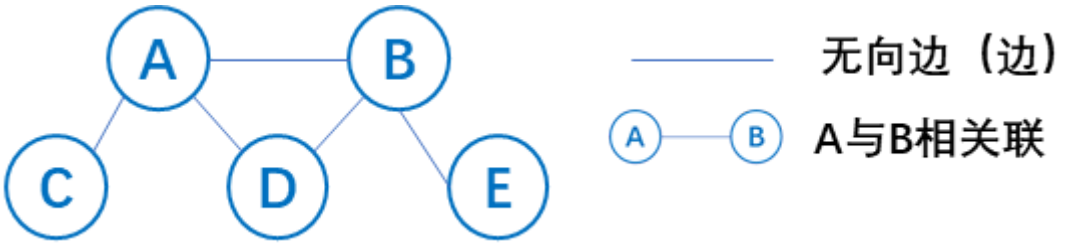
$$G=(V,E)$$

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

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

**2、无向图**

若 E 是无向边（也称边）的有限集合时，则图 G 为**无向图。**边是顶点的无序对，记为(v,w) 或(w,v)，其中 v、w 是顶点。可以说顶点 w 和顶点 v 互为邻接点，边 $(v,w)$ 依附于顶点 w 和 v；或者说边(v,w) 和顶点 v、w 相关联。




上图可以表示为：

$$G=(V,E)$$

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

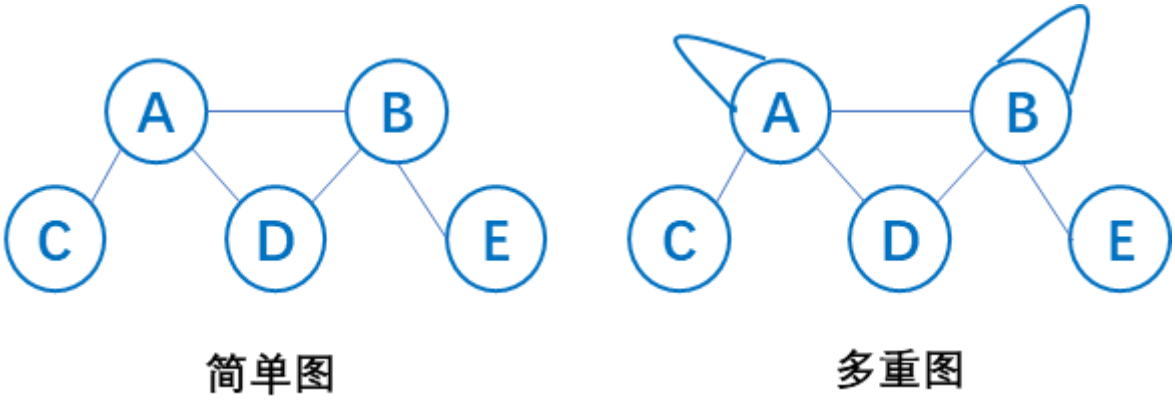
$$E=\{(A,B),(A,C),(A,D),(B,D),(B,E)\}$$

**3、简单图与多重图**

满足以下两个条件的图称为**简单图**：

1. 不存在重复边
2. 不存在顶点到自身的边

> 【注意】之后提到的图默认为简单图，数据结构中只讨论简单图。

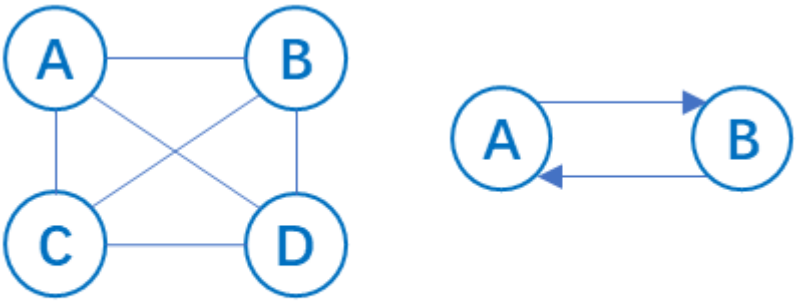



多重图与简单图是相对的两个概念。若图G中某两个顶点之间的边数大于一条，又允许顶点通过一条边与自身关联，则这样的图称为**多重图。**

**4、完全图（简单完全图）**

对于无向图，任意两个顶点间都存在边的图，这样的图称为**无向完全图。**

对于有向图，任意两个顶点间存在着方向相反的两条弧，这样的图称为**有向完全图。**



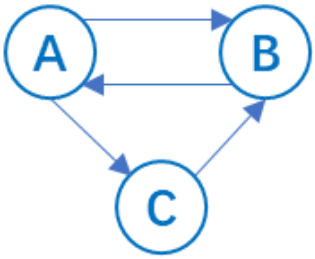

无向完全图




有向完全图

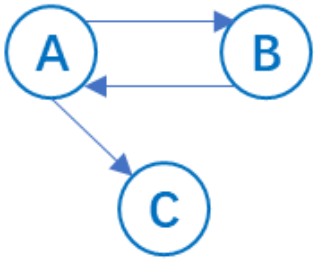
**5、子图**

设有两个图G=(V,E)和G'(V',E')，若V'是V的子集，E'是E的子集，则称G'是G的**子图**。若V(G) = V(G')则称G'是G的**生成子图**（包含原图的所有结点，可以不包含全部）。




G


G’（子图）




G’’（生成子图）

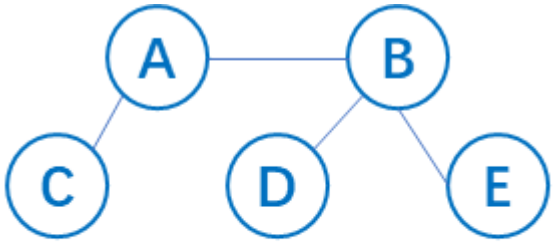
> 【注意】并非所V和E的任何子集都能构成G的子图，因为这样的子集可能**不是图**，即E的子集中的某些边关联的顶点可能不再这个V的子集中。

**6、连通、连通图和连通分量**

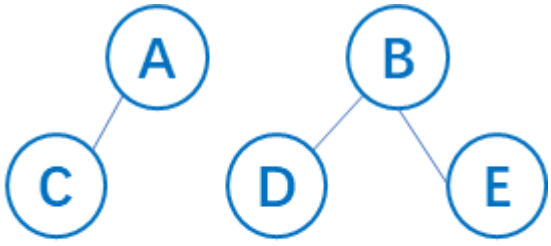
**连通**：在**无向图**中，若顶点v到顶点w有路径存在，则称v和w是连通的。

**连通图**：图中任意两点之间均至少有一条通路，否则称为非连通图。

**连通分量**：无向图中的极大连通子图称为连通分量。

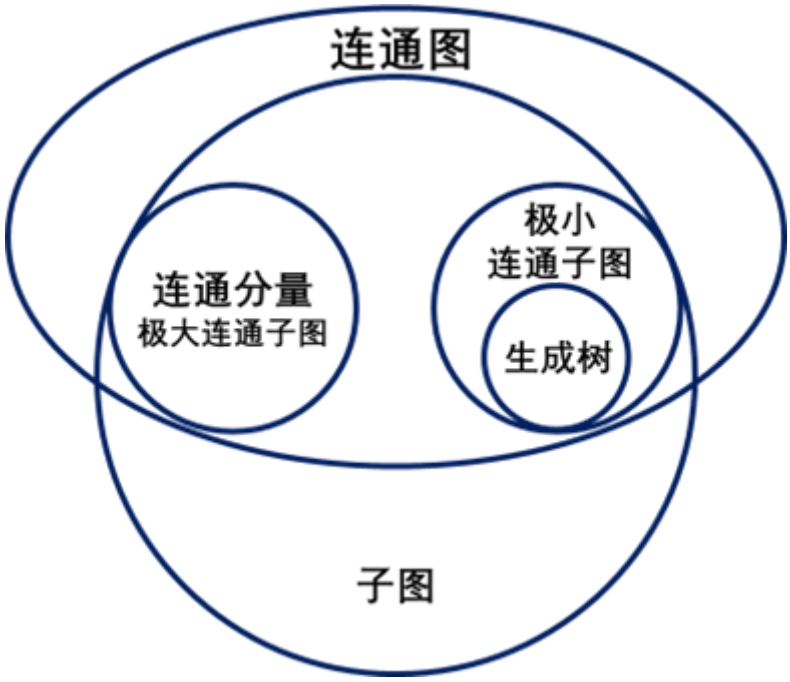



连通图




非连通图

> 若图是非连通图，则边数最多可以有$E_{max} = C_{n-1}^{2}$条。
>
> 当顶点数 − 边数 = 1时，刚好可以做到连通且无环。（注意：是可以做到，不是一定），不满足这个条件是无法连通的。
>
> 几个概念之间的关系如下：

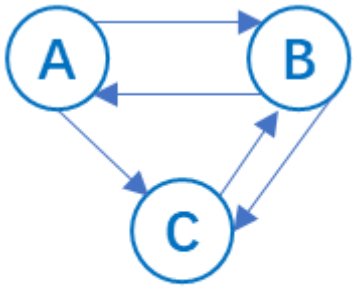



**7、强连通图、强连通分量**

**强连通**：在有向图中，如果一对顶点v和w，从v到w和从w到v之间都有路径，则称这两个顶点是**连通**的。

**强连通图**：若有向图中任意一对顶点都是强连通的，则称此图为强连通图。

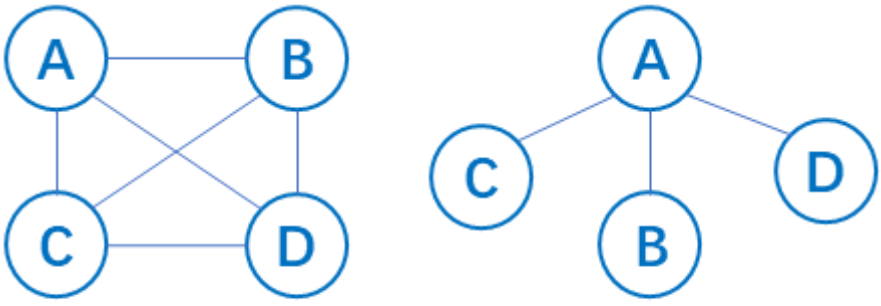
**强连通分量**：有向图中的极大强连通子图称为有向图的强连通分量。




强连通图

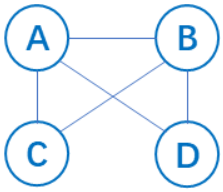
**8、生成树、生成森林**

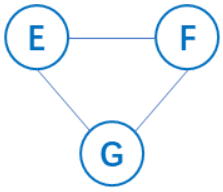
连通图的**生成树**是**包含图中所有顶点的一个极小连通子图。**



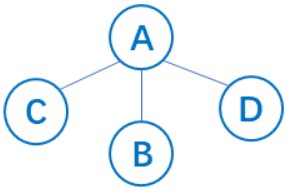

连通图　　连通图的生成树

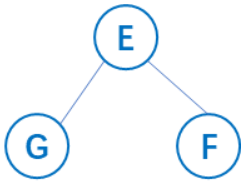
在非连通图中，连通分量的生成树构造了非连通图的**生成森林。**




非连通图

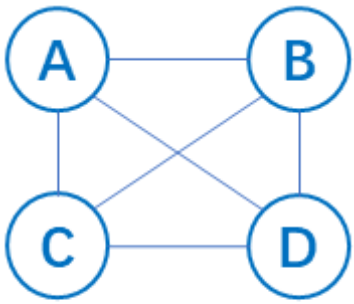



非连通图的生成森林

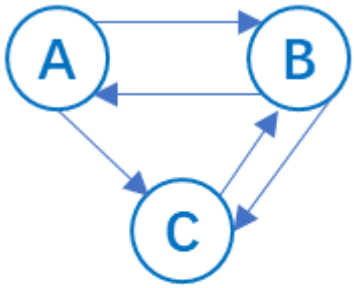
**9、顶点的度、入度和出度**

在无向图中，顶点的**度**是依附于顶点v的边的条数。




A的度为3

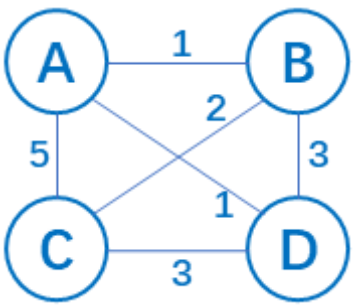
在有向图中，顶点的度分为入度和出度，**入度**是以顶点v为终点的有向边的数目，**出度**是以顶点v为起点的有向边的数目。




A的入度为1
A的出度为2

**10、边的权和网**

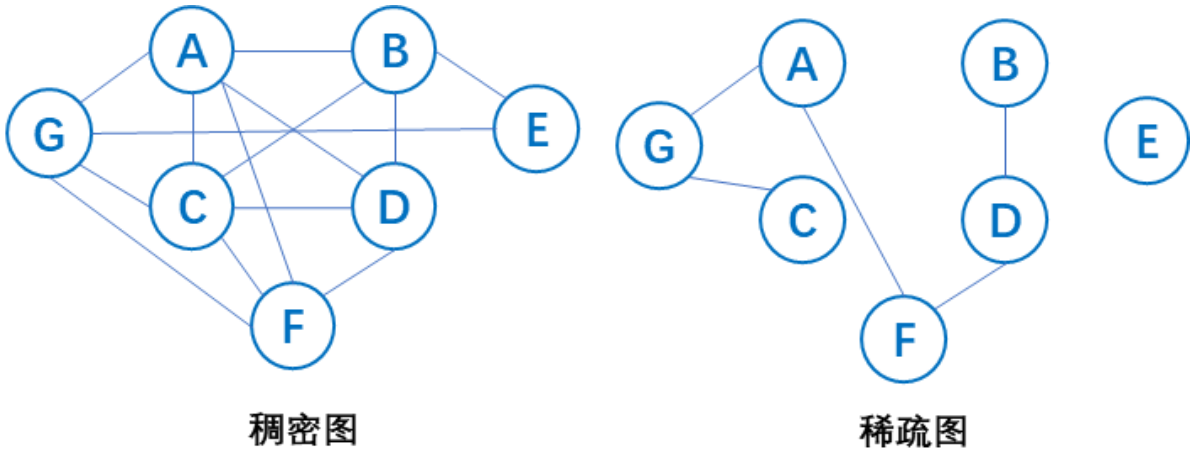
在一个图中，每条边上可以设置一些具有某些意义的数值，该数值称为边的**权**。这种边上带有权值的图称为**带权图**，也称**网**。



**11、稠密图、稀疏图**

边数很少的图称为**稀疏图**，反之称为**稠密图**。

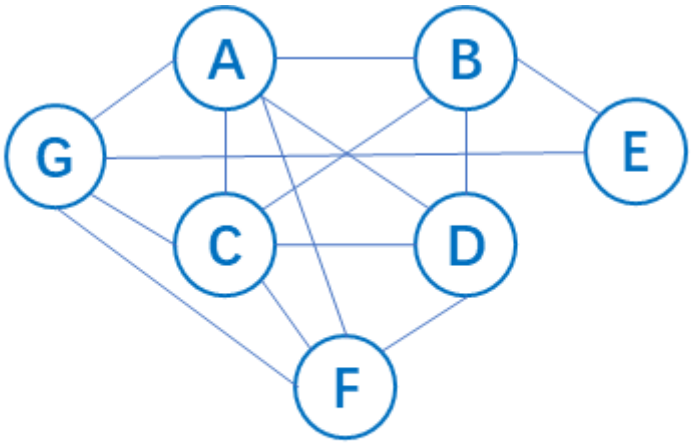
> 这两个概念本身是模糊的概念，稀疏图和稠密图是相对而言的。




**12、路径、路径长度和回路**

路径是两个顶点间访问需要经过的**结点序列**。路径上边的数目称为**路径长度**，第一个顶点和最后一个顶点相同的路径称为**回路**或**环**。

> 顶点间可能不存在路径。




E到F的一条路径为：E B A G C D F

**13、简单路径、简单回路**

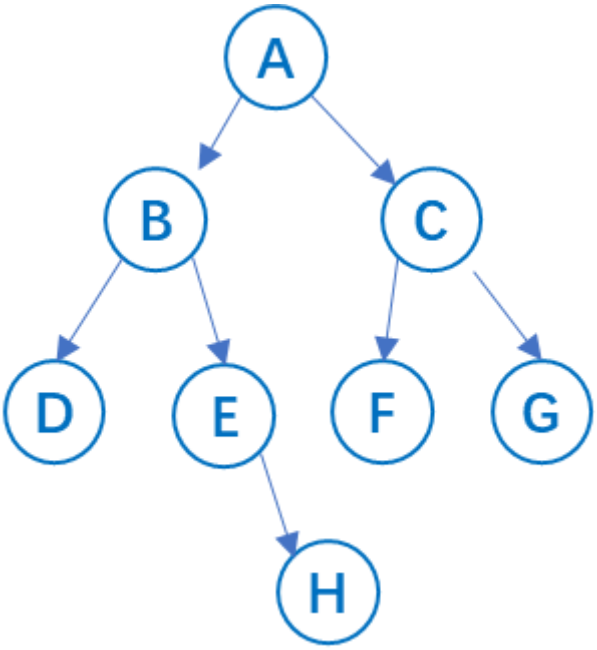
在路径序列中，顶点不重复出现的路径称为**简单路径**。除第一个顶点和最后一个顶点外不重复出现的回路称为**简单回路**。

**14、距离**

若两个顶点间的最短路径存在，则此路径的长度称为两个结点间的**距离**。若路径不存在，则距离为无穷（∞）。

**15、有向树**

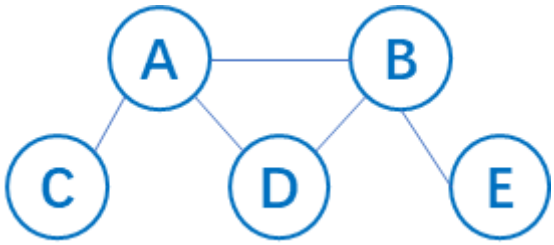
若一个顶点的入度为0，其余顶点的入度都为1的有向图，称为有向树。




## 6.2 图的存储及其基本操作

### 6.2.1 邻接矩阵法

所谓邻接矩阵存储，是使**用一个一维数组存储图中各个顶点的信息，一个二维数组存储图中边的信息**，存储结点邻接关系的二维数组称为邻接矩阵。




上图的邻接矩阵如下：

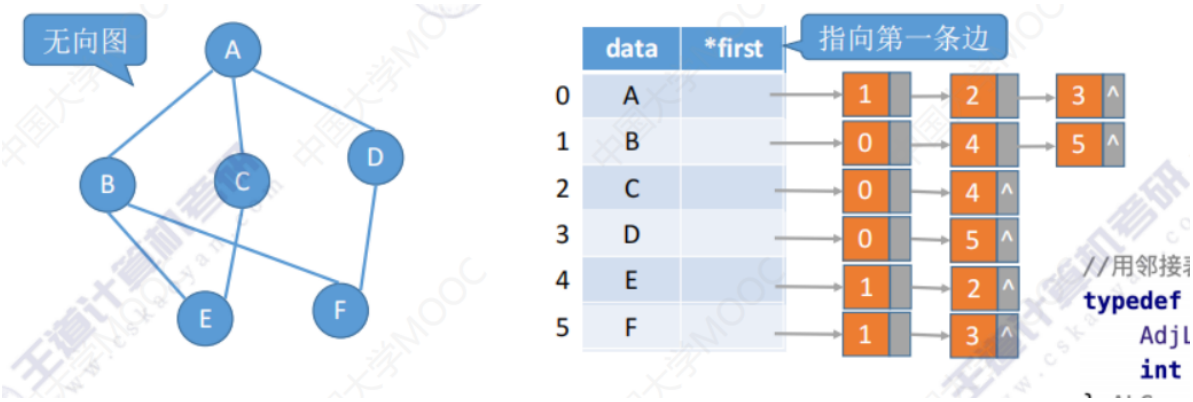
$$\begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

> 【注意】
>
> - 在简单应用中，可直接用二维数组作为图的邻接矩阵（顶点信息等均可省略）。
> - 当邻接矩阵的元素仅表示相应边是否存在时，EdgeType可采用值为0和1的枚举类型。
> - 无向图的邻接矩阵是对称矩阵，对规模特大的邻接矩阵可采用压缩存储。
> - 邻接矩阵表示法的空间复杂度为$O(N^2)$，其中n为图的顶点数|V|。

存储结构定义如下：

```
typedef char VertexType;     // 顶点的数据类型
typedef int EdgeType;        // 边的数据类型
typedef struct {
    VertexType Vex[100];
    EdgeType Edge[100][100];
    int vexNum,arcNum;       //图的当前顶点和弧数
}MGraph;
```

### 6.2.2 邻接表法(顺序+链式存储)




邻接表法是对图G中的每个顶点$v_i$建立一个单链表，第i个单链表中的结点表示依附于顶点$v_i$的边。

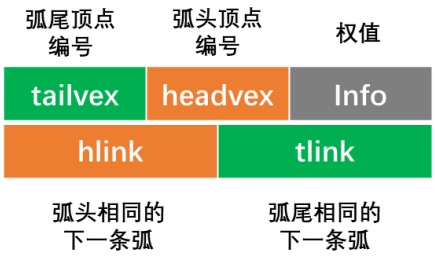
> 【注意】
>
> - 邻接表的表示方式不唯一
> - 对于无向图，邻接表的每条边会对应两条信息，删除顶点、边等操作复杂度高
> - 无向图采用邻接表存储所需要的存储空间为$O(|V|+2|E|)$（2E是因为在无向图中，每条边在邻接表中出现了两次），有向图采用邻接表存储所需的存储空间为$O(|V|=|E|)$。
> - 对于**稀疏图**，采用邻接表法表示可以节省大量存储空间。
> - 在邻接表中，找到一个顶点的邻边很容易，但是若要确定给定的两个顶点中是否存在边，在邻接矩阵中可以立刻查到，但是在邻接表中效率很低。

其存储结构表示如下：

```
#define MVNum 100                           //最大顶点数

typedef struct ArcNode {                    //边/弧
    int adjvex;                             //邻接点的位置
    struct ArcNode *next;                   //指向下一个表结点的指针
} ArcNode;

typedef struct VNode {
    char data;                              //顶点信息
    ArcNode *first;                         //第一条边/弧
} VNode, AdjList[MVNum];                    //AdjList表示邻接表类型

typedef struct {
    AdjList vertices;                       //头结点数组
    int vexnum, arcnum;                     //当前的顶点数和边数
} ALGraph;
```

### 6.2.3 十字链表（只能存储有向图）

**十字链表**是有向图的一种链式存储结构，对应于有向图中的每条弧有一个结点，对于每个**顶点**也有一个结点。结点的结构如下图所示：

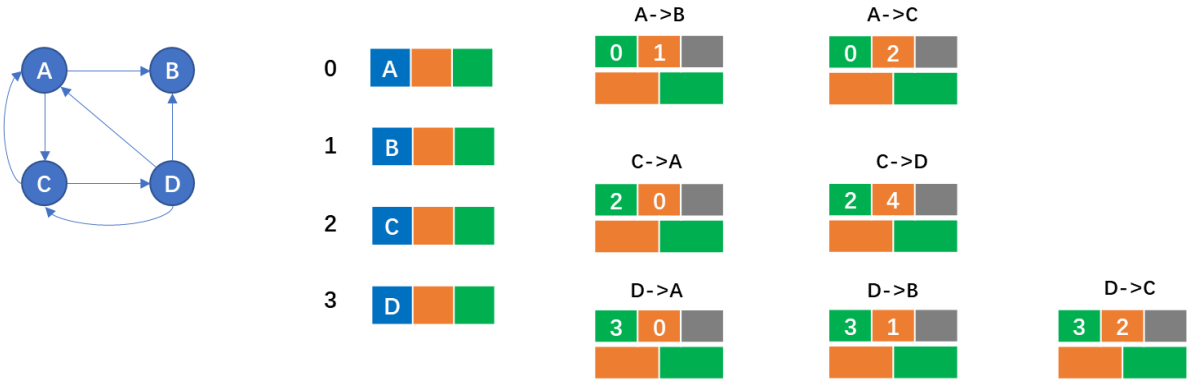



弧结点

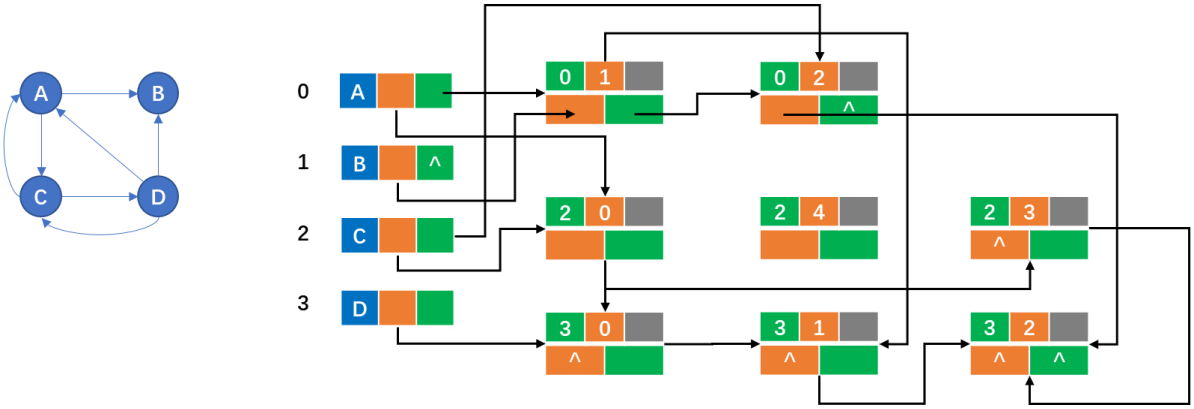

顶点结点

绘制是十字链表的过程如下：

1、将顶点和弧分别用上述两种顶点表示出来：




2、根据关系连线：



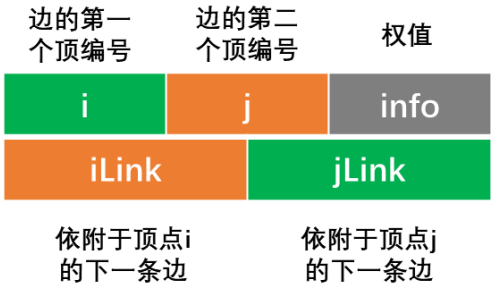

**十字链表性能分析**

空间复杂度：$O(|V|+|E|)$

想要找到十字链表指定顶点的所有入边和出边，只需要沿着某个顶点的hlink或tlink一直找即可。

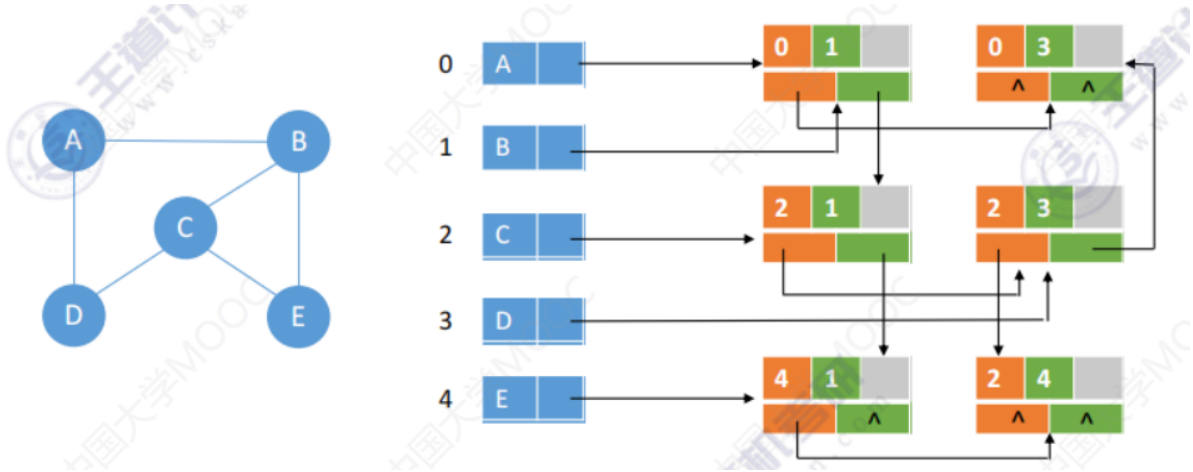
### 6.2.4 邻接多重表存储无向图

邻接多重表是存储**无向图**的另一种存储结构。




边结点


顶点结点




**性能分析：**

空间复杂度：$O(|V|+|E|)$

删除边、删除结点等操作很方便。

只适用于存储无向图。

**图的四种存储方法的比较**

| | 邻接表 | 邻接矩阵 | 十字链表 | 邻接多重表 |
|---|---|---|---|---|
| 空间复杂度 | 无向图：$O(\|V\|+2\|E\|)$，有向图：$O(\|V\|+\|E\|)$ | $O(\|V^2\|)$ | $O(\|V\|+\|E\|)$ | $O(\|V\|+\|E\|)$ |
| 适合存储 | 稀疏图 | 稠密图 | 仅有向图 | 仅无向图 |
| 表示方式 | 不唯一 | 唯一 | 不唯一 | 不唯一 |
| 计算度/出度/入度 | 计算有向图的度、入度不方便，其余很方便 | 必须遍历对应行或列 | 很方便 | 很方便 |
| 删除边或顶点 | 无向图中删除边和顶点都不方便 | 删除边很方便，删除顶点需要大量移动数据。 | 很方便 | 很方便 |
| 找邻边 | 找有向图的入边不方便，其余很方便 | 必须遍历对应行或列 | 很方便 | 很方便 |

## 6.3 图的遍历

### 6.3.1 图的广度优先遍历（BFS）

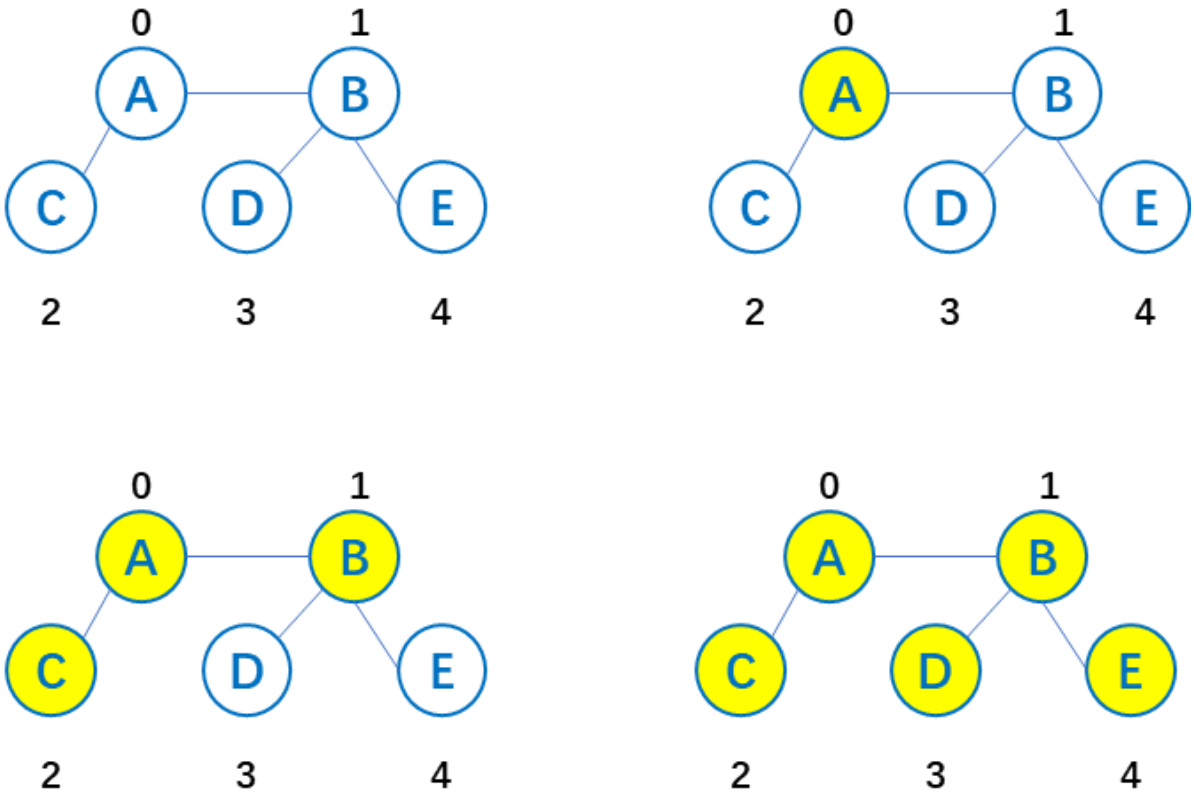
#### 1、理论原理及要点

> 【回顾】树的广度优先遍历：层序遍历

广度优先遍历指的是从图的一个未遍历的节点出发，**先遍历这个节点的相邻节点，再依次遍历每个相邻节点的相邻节点。**

> 【要点】
>
> - 找到与一个顶点相邻的所有顶点；
> - 标记哪些顶点被访问过；
> - 需要一个辅助队列。

其手动遍历方式如下：




最终得到的遍历序列为：A B C D E

#### 2、代码实现

广度优先遍历所需要的操作：

- `FirstNeighber(G,x)`：求图中顶点x的第一个邻接点，有则返回顶点号，否则返回-1；
- `NextNeighber(G,x,y)`：假设图中顶点y是顶点x的一个邻接点，则返回除了y以外，顶点x的下一个邻接点的编号，若y是x的最后一个邻接点，则返回-1；
- bool visited[MAX_VERTEX_NUM];　　// 访问标记数组

```
// 本示例为伪代码，主要是为了表现BFS的原理
#include "iostream"
```

```
#define MAX_VERTEX_NUM 10
using namespace std;

typedef int *Queue; // 定义队列类型，这里使用伪代码表示，方便代码的阅读，当做队列使用
typedef int *Graph; // 伪代码使用，作用同上
Queue Q;
bool visited[MAX_VERTEX_NUM];

void EnQueue(Queue Q, int v);   // v入队
void DeQueue(Queue Q, int v);   // v出队
bool isEmpty(Queue Q);   // v出队
void visit(int w);              // 访问结点w
int FirstNeighbor(Graph G, int v);  // 求图中顶点x的第一个邻接点
int NextNeighbor(Graph G, int v, int w);   // 求除了w外，v的下一个邻接点的编号

// 广度优先遍历
void BFS(Graph G, int v) {              // 从顶点v出发，广度优先遍历图G
    visit(v);                           // 访问起始节点v
    visited[v] = true;                  // 访问标记
    EnQueue(Q, v);                      // v结点入队
    while (!isEmpty(Q)) {               // 当队列不为空
        DeQueue(Q, v);                  // 顶点v出队
        // 检查v的所有邻接点
        for (int w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w)) {
            if (!visited[w]) {          // w是v的尚未访问的邻接点
                visit(w);               // 访问结点w
                visited[w] = true;      // 访问标记
                EnQueue(Q, w);         // 顶点w入队
            }
        }
    }
}
```

> 遍历序列的可变性：
>
> - 同一个图的**邻接矩阵**表示法**唯一**，因此广度优先遍历序列**唯一**
> - 同一个图的**邻接表**表示法**不唯一**，因此广度优先遍历序列**不唯一**

> 算法存在的问题：如果是非连通图，则上述代码无法遍历完所有结点
>
> 解决方法：遍历整个visited数组，检查**是否还有未访问过的结点。**

【结论】对于无向图，调用BFS的次数=连通分量数

### 3、性能分析

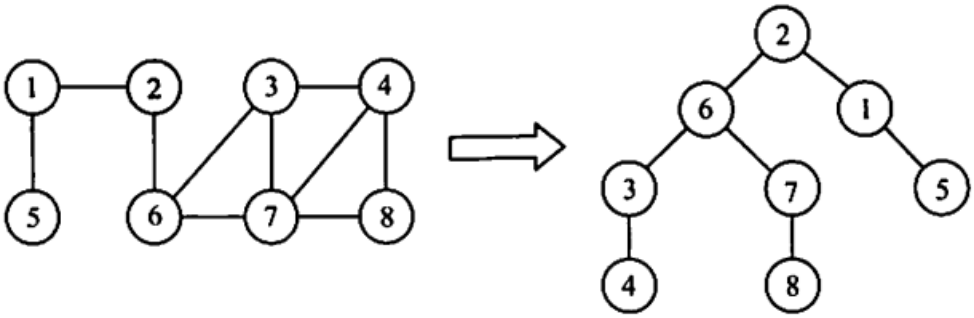
空间复杂度：最坏的情况下，辅助队列大小为$O(|V|)$

时间复杂度：

- 采用*邻接表*存储方式时，每个顶点均需要搜索一遍，故时间复杂度为$O(|V|)$，在查找某个顶点的邻接点共需要$O(|E|)$的时间，总的时间复杂度为：$O(|V|+|E|)$。
- 采用邻接矩阵存储方式时，访问V个顶点需要$O(|V|)$的时间。查找每个顶点的邻接点需要$O(|V|)$的时间，时间复杂度为$O(|V|^2)$

#### 4、广度优先生成树

在广度遍历时，我们可以得到一棵遍历树，称为广度优先生成树。




> 因为图的邻接矩阵存储表示是唯一的，所以广度优先生成树也是唯一的。
>
> 而邻接表法表示不唯一，故使用邻接表法表示的图的广度优先生成树不唯一。

### 6.3.2 深度优先遍历（DFS）

#### 1、概念

DFS的基本思想：首先访问图中的某一起始顶点v，然后从v出发，访问与v邻接且未被访问的任一顶点w1，再访问与w1邻接且未被访问的任一顶点w2.....重复上述过程，直到不能继续向下访问时，依次退回到最近被访问的顶点，若它还有邻接顶点未被访问过，则从该点继续重复上述搜索过程，直到所有顶点都被访问为止。

以下是DFS的一个示例：


#### 2、代码实现

```
// 本示例为伪代码，主要是为了表现DFS的原理
#include "iostream"

#define MAX_VERTEX_NUM 10
```

```
using namespace std;

typedef char VertexType;    // 顶点的数据类型
typedef int EdgeType;       // 边的数据类型
typedef struct {
    VertexType Vex[100];
    EdgeType Edge[100][100];
    int vexNum, arcNum;      //图的当前顶点和弧数
} Graph;
bool visited[MAX_VERTEX_NUM];
int v;

void visit(int w);               // 访问结点w
int FirstNeighbor(Graph G, int v);  // 求图中顶点x的第一个邻接点
int NextNeighbor(Graph G, int v, int w);   // 求除了w外，v的下一个邻接点的编号

// 深度优先遍历
void DFS(Graph G, int v) {  // 从顶点v出发，深度优先遍历图G
    visit(v);       // 访问顶点v
    visited[v] = true;  // 设置已访问标记
    for (int w = FirstNeighbor(G, v); w >= 0; w = NextNeighbor(G, v, w)) {
        if (!visited[w]) {  // w为v的未访问的邻接节点
            DFS(G, w);
        }
    }
}

void DFSTraverse(Graph G) {
    for (v = 0; v < G.vexNum; ++v) {
        visited[v] = false;
    }
    for (v = 0; v < G.vexNum; ++v) {
        if (!visited[v]) {
            DFS(G, v);
        }
    }
}
```

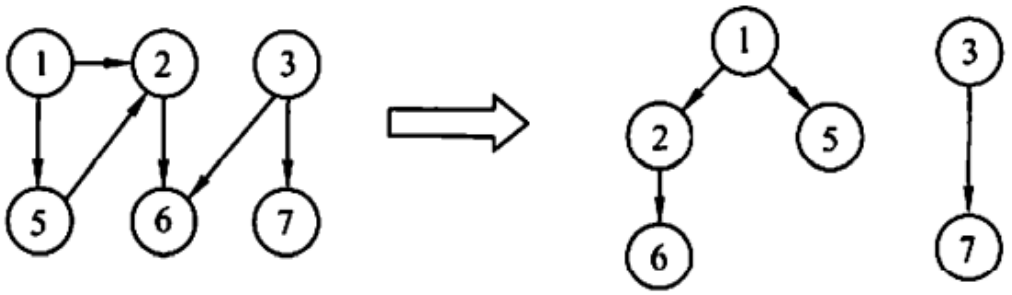
> 注意：图的邻接矩阵表示是唯一的，但对于邻接表来说，若边的输入次序不同，生成的邻接表也不同。因此，对于相同的图，基于**邻接矩阵**遍历所得到的DFS序列和BFS序列是**唯一**的，基于**邻接表**的遍历所得到的DFS和BFS序列是**不唯一**的。

### 3、效率分析

- DFS是一个**递归算法**，需要借助函数调用栈，故其空间复杂度为$O(|V|)$。
- 遍历图的过程实质上是对每个顶点查找其邻接点的过程，其耗费的时间取决于其所用的存储结构：
  - 以邻接矩阵表示时：
    - 查找每个顶点的邻接点所需的时间为$O(|V|)$
    - 总的时间复杂度为$O(|V|^2)$
  - 以邻接表表示时：
    - 查找所有顶点的邻接点所需时间为$O(|E|)$
    - 访问顶点所需的时间为$O(|V|)$
    - 总的时间复杂度为$O(|E|+|V|)$

#### 4、深度优先生成树和生成森林

与广度优先搜索一样，深度优先搜索也会产生一棵深度优先生成树。当然，这是有条件的，即对连通图调用DFS才能产生深度优先生成树，否则产生的将是深度优先生成森林。与 BFS类似，基于邻接表存储的深度优先生成树是不唯一的。




### 6.3.3 图的遍历与图的连通性

图的遍历算法可以用来判断图的连通性。

对于无向图来说：

- 若无向图是连通的，则从任一结点出发，仅需一次遍历就能够访问图中的所有顶点；
- 若无向图是非连通的，则从某一个顶点出发，一次遍历只能访问到该顶点所在连通分量的所有顶点，而对于图中其他连通分量的顶点，则无法通过这次遍历访问。

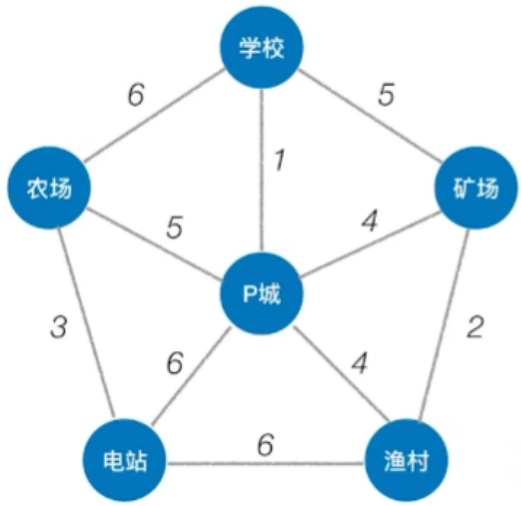
对于有向图来说：

- 若从初始点到图中的每个顶点都有路径，则能够访问到图中的所有顶点，否则不能访问到所有顶点。

> 强连通图使用一次BFS就能完成遍历。

## 6.4 图的应用

### 6.4.1 最小生成树




道路规划要求：所有地方都连通，且成本尽可能的低



对于一个带权连通无向图$G=(V,E)$，生成树不同，每棵树的权（树中所有边上的权值和）也不同，设$R$为$G$的所有生成树的集合，若$T$为$R$中权值和最小的生成树，则$T$称为$G$的最小生成树（Minimum-Spanning-Tree，MST）

> 一个图中可能存在多条相连的边,我们**一定可以从一个图中挑出一些边生成一棵树。**这仅仅是生成一棵树,还未满足最小,**当图中每条边都存在权重时,这时候我们从图中生成一棵树(n - 1 条边)时,生成这棵树的总代价就是每条边的权重相加之和。**

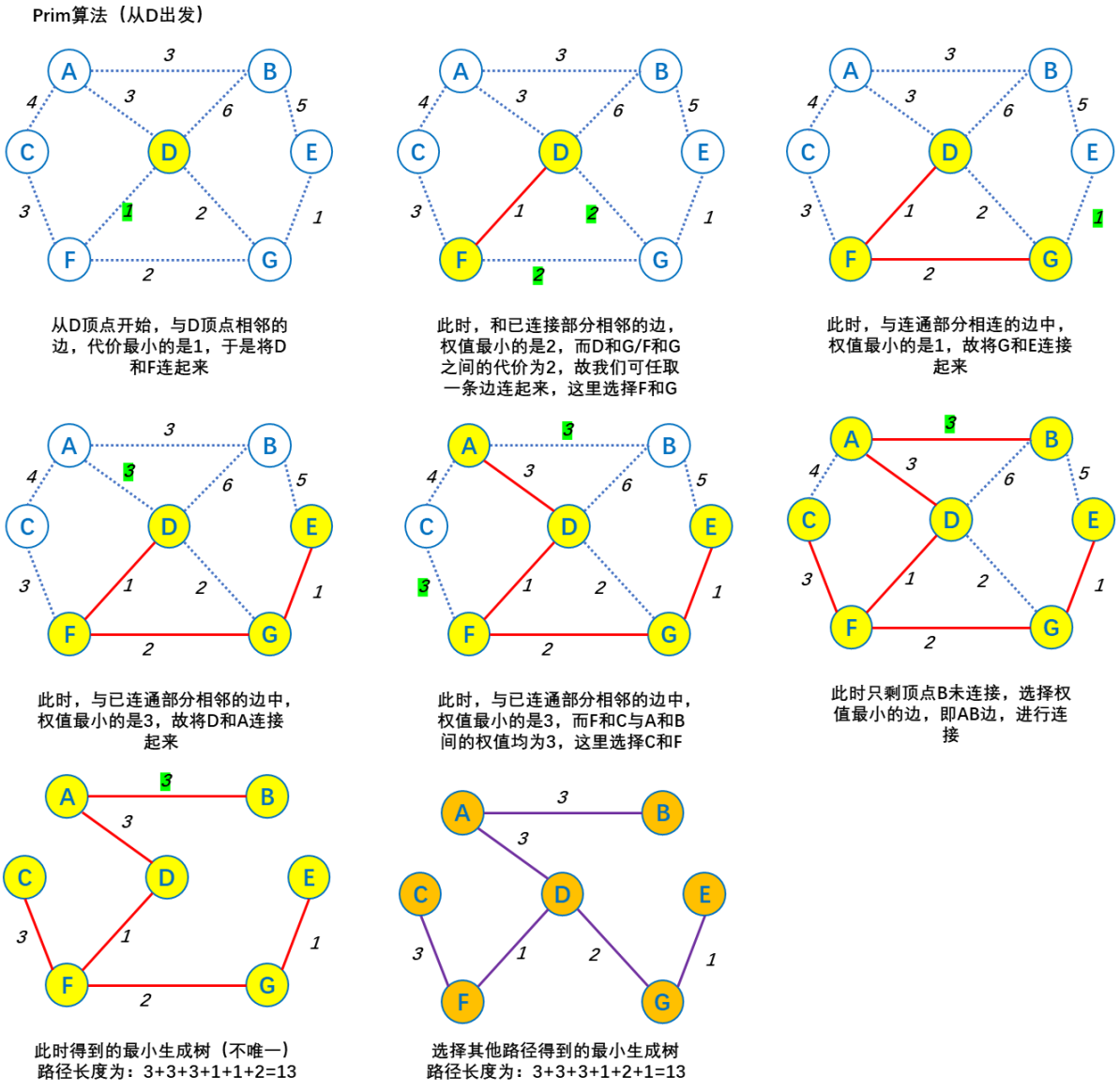
> 【注意】
>
> 1、最小生成树可能有多个，但边的权值之和总是唯一且最小的
>
> 2、最小生成树的边数=定点数-1，砍掉一条则不连通，增加一条则会出现回路
>
> 3、若一个连通图本身就是一颗树，则其最小生成树就是它本身
>
> 4、只有连通图才有生成树，非连通图只有生成森林

### 1、Prim算法

从某一个顶点（所以存在多个最小生成树）开始构建生成树，每次将代价最小的新顶点纳入生成树，直到所有顶点都纳入为止。

> 类似贪心算法

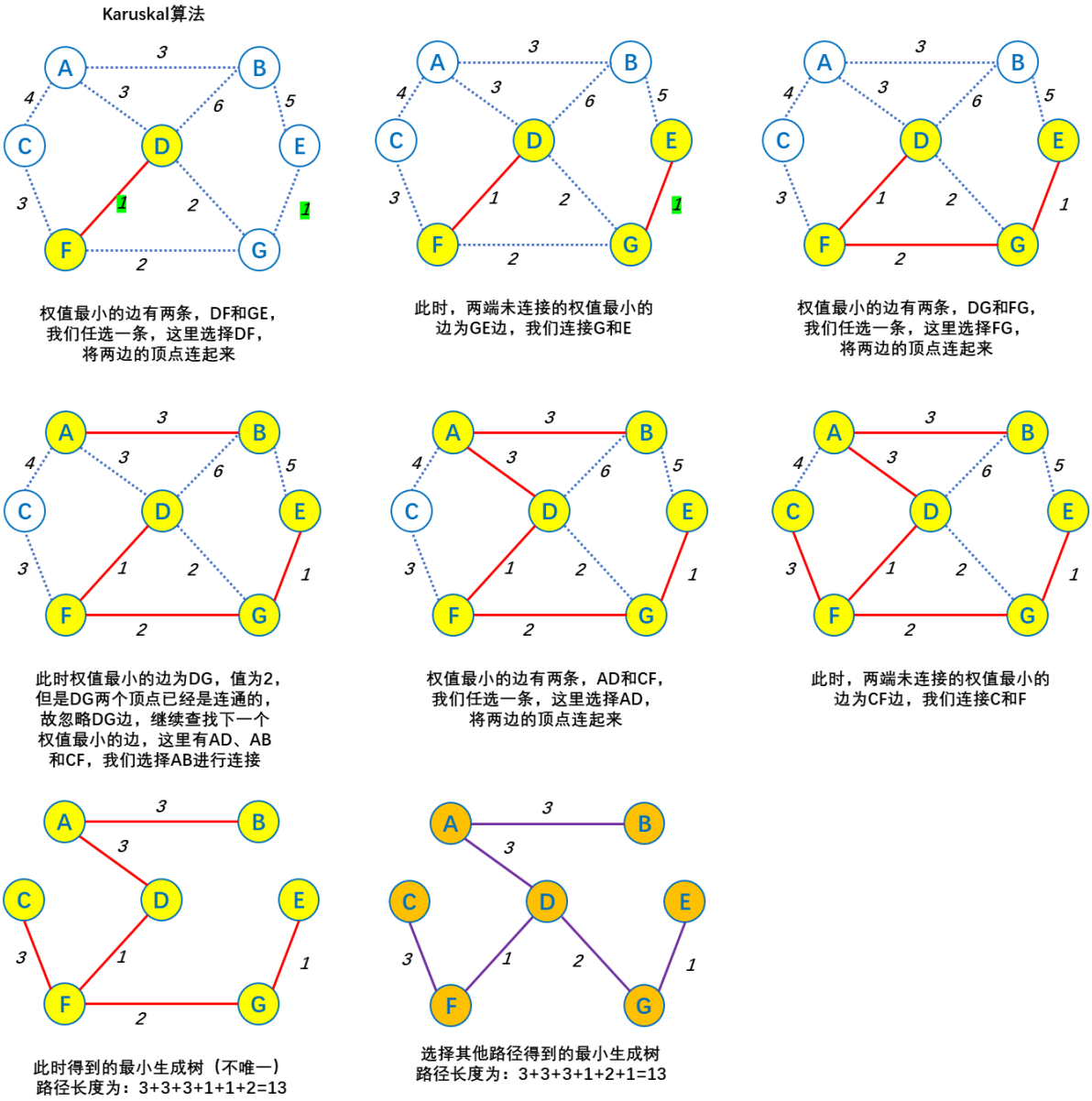
示例如下：



### 2、Kruskal算法

每次选择权值最小的边，使这条边的两头连通（原本已近连通则不选）直到所有结点都连通。

示例如下：




### 3、对比Prim和Kruskal算法

- **Prim算法**:
  - 时间复杂度:$O(|V|^2)$
  - 适合边稠密的图
- **Kruskal算法**
  - 时间复杂度：$O(|E|log_2|E|)$
  - 适用于求解边稀疏图

## 6.4.2 最短路径

最短路径算法

### 1、BFS求解最短路径

使用 BFS算法求无权图的最短路径问题，需要使用三个数组：

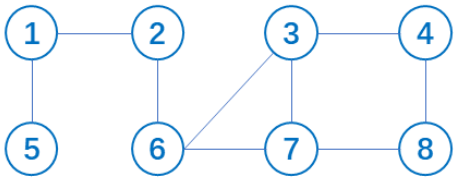
1. `d[]` 数组用于记录顶点 u 到其他顶点的最短路径。
2. `path[]` 数组用于记录最短路径从那个顶点过来。
3. `visited[]` 数组用于记录是否被访问过。

代码实现：

```
#define MAX_LENGTH 2147483647           //地图中最大距离，表示正无穷

// 求顶点u到其他顶点的最短路径
void BFS_MIN_Disrance(Graph G,int u){
    for(i=0; i<G.vexnum; i++){
        visited[i]=FALSE;               //初始化访问标记数组
        d[i]=MAX_LENGTH;                //初始化路径长度
        path[i]=-1;                     //初始化最短路径记录
    }
    InitQueue(Q);                       //初始化辅助队列
    d[u]=0;
    visites[u]=TREE;
    EnQueue(Q,u);
    while(!isEmpty[Q]){                 //BFS算法主过程
        DeQueue(Q,u);                   //队头元素出队并赋给u
        for(w=FirstNeighbor(G,u);w>=0;w=NextNeighbor(G,u,w)){
            if(!visited[w]){
                d[w]=d[u]+1;
                path[w]=u;
                visited[w]=TREE;
                EnQueue(Q,w);           //顶点w入队
            }
        }
    }
}
```

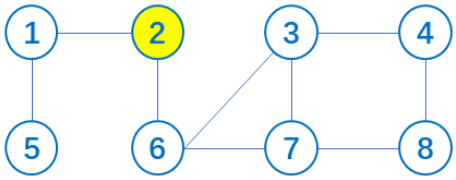
以下为手算示例：

求顶点2到其他顶点的最短路径（BFS）




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | F | F | F | F | F | F | F | F |
| d | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| path | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

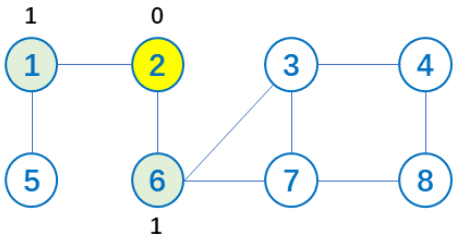
初始状态，visited数组表示顶点是否访问过，d数组表示初始顶点到某个顶点间的距离，-1表示无穷大，path表示到达该结点的上一个结点的编号，-1表示该结点为起始节点。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | F | T | F | F | F | F | F | F |
| d | -1 | 0 | -1 | -1 | -1 | -1 | -1 | -1 |
| path | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |

2

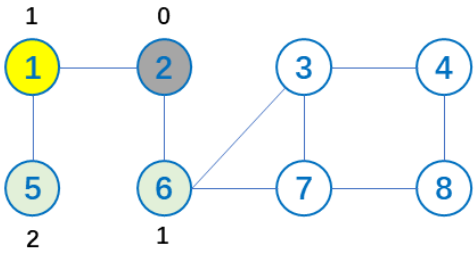
执行BFS操作，顶点2入队。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | F | F | F | T | F | F |
| d | 1 | 0 | -1 | -1 | -1 | 1 | -1 | -1 |
| path | 2 | -1 | -1 | -1 | -1 | 2 | -1 | -1 |

1 6

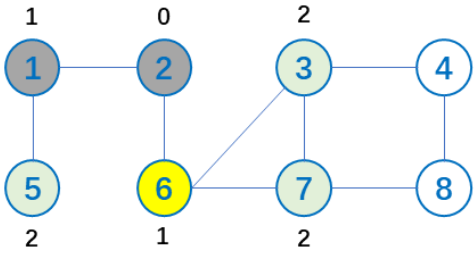
此时队列不为空，2号顶点出队，开始访问与2号顶点相邻的顶点，并修改1号和6号顶点的d值和path值，并将访问标记置为TRUE。1、6号顶点入队。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | F | F | T | T | F | F |
| d | 1 | 0 | -1 | -1 | 2 | 1 | -1 | -1 |
| path | 2 | -1 | -1 | -1 | 1 | 2 | -1 | -1 |

6 5

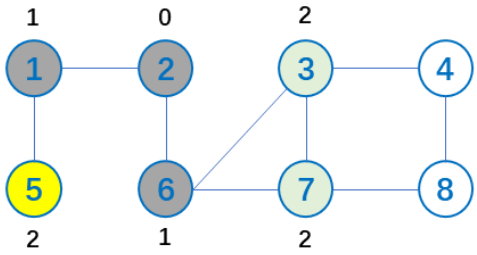
1号顶点出队，开始访问1号顶点相邻的顶点，发现2号顶点已被访问，故跳过2号顶点，访问5号顶点。对5号顶点进行相应操作。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | T | F | T | T | T | F |
| d | 1 | 0 | 2 | -1 | 2 | 1 | 2 | -1 |
| path | 2 | -1 | 6 | -1 | 1 | 2 | 6 | -1 |

5 3 7

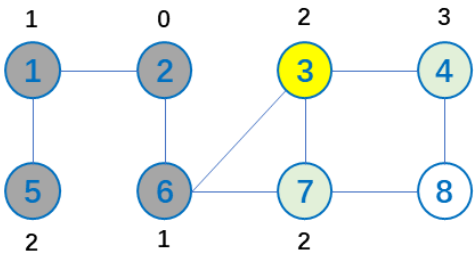
6号结点出队，访问其相邻的结点，并执行相应操作。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | T | F | T | T | T | F |
| d | 1 | 0 | 2 | -1 | 2 | 1 | 2 | -1 |
| path | 2 | -1 | 6 | -1 | 1 | 2 | 6 | -1 |

3 7

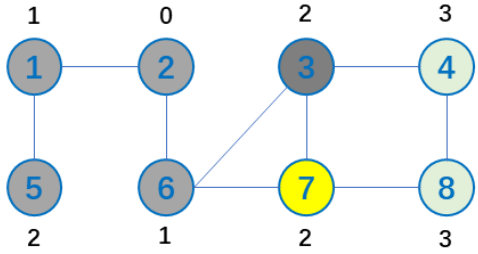
5号结点出队，发现没有可以访问的结点，故跳过。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | T | T | T | T | T | F |
| d | 1 | 0 | 2 | 3 | 2 | 1 | 2 | -1 |
| path | 2 | -1 | 6 | 3 | 1 | 2 | 6 | -1 |

7 4

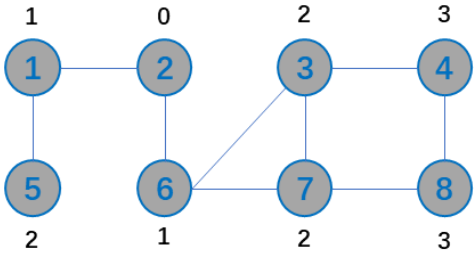
3号结点出队，访问其相邻的结点，并执行相应操作。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | T | T | T | T | T | T |
| d | 1 | 0 | 2 | 3 | 2 | 1 | 2 | 3 |
| path | 2 | -1 | 6 | 3 | 1 | 2 | 6 | 7 |

4 8

7号结点出队，访问其相邻的结点，并执行相应操作。




| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| visted | T | T | T | T | T | T | T | T |
| d | 1 | 0 | 2 | 3 | 2 | 1 | 2 | 3 |
| path | 2 | -1 | 6 | 3 | 1 | 2 | 6 | 7 |

剩余两个结点均没有其他结点可以访问，此时就得到了最终的数据，可以直接根据数组的值来获取最短路径，如从2到8的最短路径为2 6 7 8

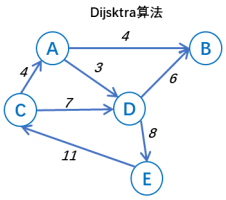
## 2、Dijkstra算法求解单源最短路径问题

BFS算法的局限性：BFS算法求单源最短路径只适用于无权图，或所有边的权值都相同的图。

Dijkstra算法需要用到三个数组：

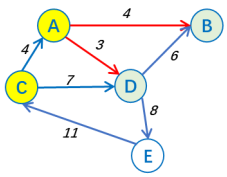
- `final[]`：标记各个顶点是否已经找到最短路径
- `dist[]`：最短路径长度
- `path[]`：路径上的前驱

以下是示例执行过程：



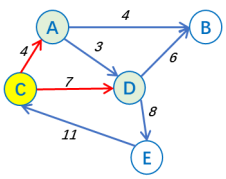

| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | F | F | T | F | F |
| dist | INF | INF | 0 | INF | INF |
| path | -1 | -1 | -1 | -1 | -1 |

初始状态，final表示顶点是否找到了最短路径，dist表示最短路径长度，path表示最短路径的前驱。从C点出发，到达C点的最短路径为0，设置其已经找到最短路径



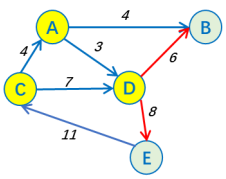

| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | T | F | T | F | F |
| dist | 4 | 8 | 0 | 7 | INF |
| path | C | A | -1 | C | -1 |

此时计算B点和D点的距离，发现D点的dist值不变，故只需更改B点的信息。



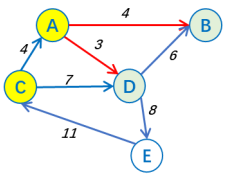

| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | F | F | T | F | F |
| dist | 4 | INF | 0 | 7 | INF |
| path | C | -1 | -1 | C | -1 |

检查由C点出发的顶点路径，发现A和D点能够被访问，这时修改A和D点的数组信息。此时，一轮操作已经完成。



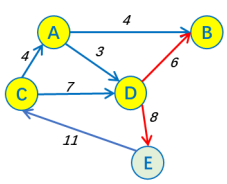

| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | T | F | T | T | F |
| dist | 4 | 8 | 0 | 7 | 15 |
| path | C | A | -1 | C | D |

继续下一趟操作，检查final数组为False的顶点中，dist值最小的，并将其置为True，继续检查从D顶点出发的边。此时计算B点和E点的距离，发现B点的dist值比之前更大，故只需更改E点的信息。




| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | T | F | T | F | F |
| dist | 4 | INF | 0 | 7 | INF |
| path | C | -1 | -1 | C | -1 |

这时检查final数组为False的顶点中，dist值最小的，并将其置为True，继续检查从A顶点出发的边。




| | A | B | C | D | E |
|---|---|---|---|---|---|
| final | T | T | T | T | T |
| dist | 4 | 8 | 0 | 7 | 15 |
| path | C | A | -1 | C | D |

继续检查final数组为False的顶点中，dist值最小的，并将其置为True，发现B点出度为0，继续下一步。继续检查final数组为FALSE的顶点中，dist值最小的，发现只剩一个顶点，将其的final值置为true，dijsktra算法结束。

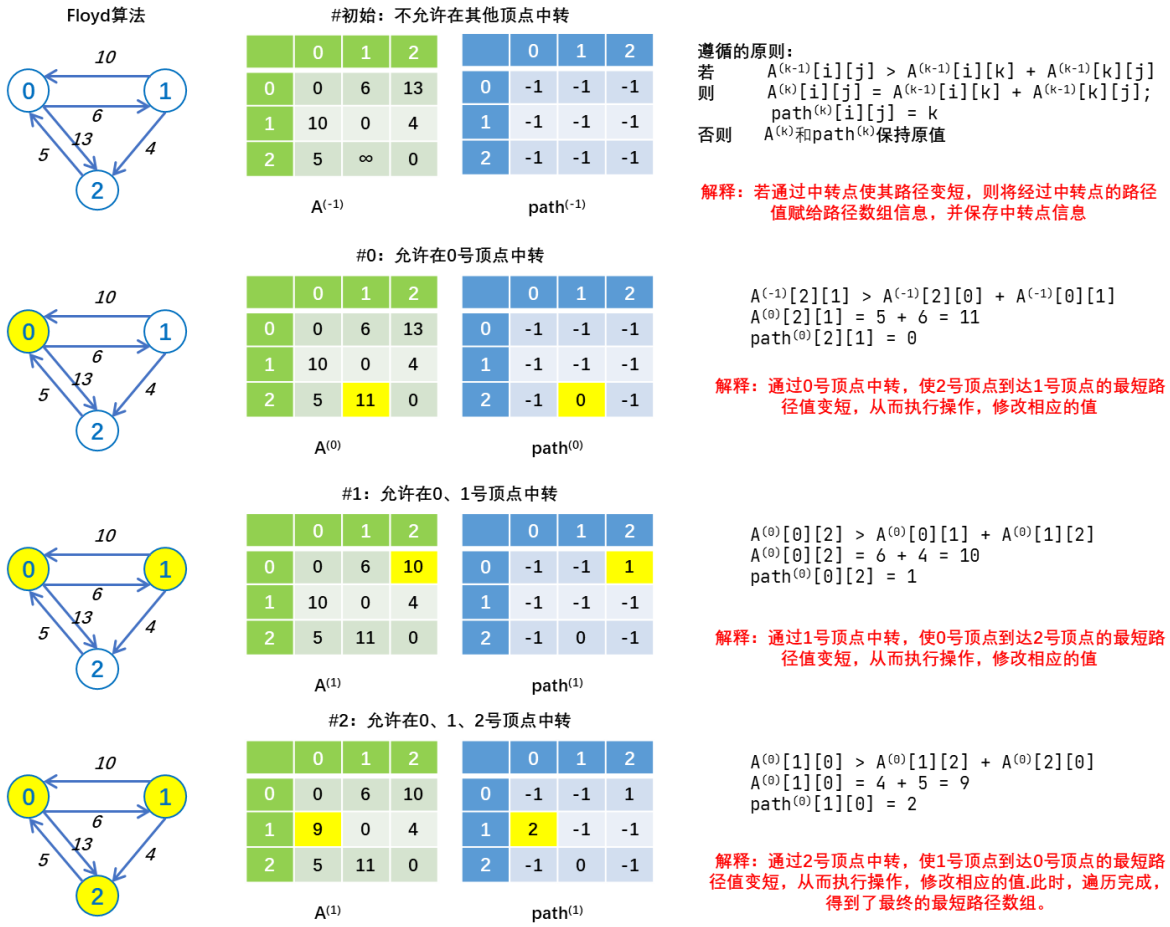
### 3、Floyd算法

Floyd算法用于求出**每一对顶点之间的最短路径**，使用**动态规划**思想，将问题的求解分为多个阶段。

Floyd算法使用到两个矩阵：

- `dist[][]`：目前各顶点间的最短路径。
- `path[][]`：两个顶点之间的中转点。

以下是执行过程：




代码实现如下：

```
int dist[MaxVertexNum][MaxVertexNum];
int path[MaxVertexNum][MaxVertexNum];

void Floyd(MGraph G){
    int i,j,k;
    // 初始化部分
    for(i=0;i<G.vexnum;i++){
        for(j=0;j<G.vexnum;j++){
            dist[i][j]=G.Edge[i][j];
            path[i][j]=-1;
        }
    }
    // 算法核心部分
    for(k=0;k<G.vexnum;k++){
        for(i=0;i<G.vexnum;i++){
            for(j=0;j<G.vexnum;j++){
                if(dist[i][j]>dist[i][k]+dist[k][j]){
                    dist[i][j]=dist[i][k]+dist[k][j];
```

```
                    path[i][j]=k;
                }
            }
        }
    }
}
```

Floyd算法的时间复杂度为$O(|V|^3)$，但由于其代码很紧凑，不包含其他复杂的数据结构，其隐含的常数系数是很小的，在处理中等规模的问题上，还是很有效的。

> Floyd算法可以用于负权值带权图，但是不能解决带有“负权回路”的图（有负权值的边组成回路），这种图有可能没有最短路径。

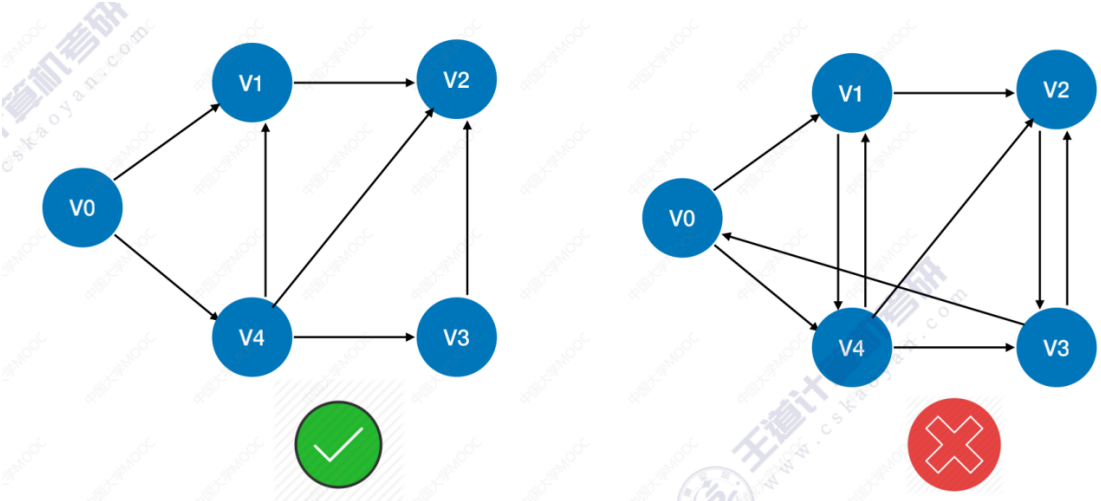
#### 4、最短路径算法比较

| BFS算法 | Dijkstra算法 | Floyd算法 | BFS |
| --- | --- | --- | --- |
| 无权图 | ✔ | ✔ | ✔ |
| 带权图 | ✗ | ✔ | ✔ |
| 带负权值的图 | ✗ | ✗ | ✔ |
| 带负权回路的图 | ✗ | ✗ | ✗ |
| 通常用于 | 求无权图的单源最短路径 | 求带权图的单源最短路径 | 求带权图中各顶点间的最短路径 |

### 6.4.3 有向无环图描述表达式

#### 1、有向无环图

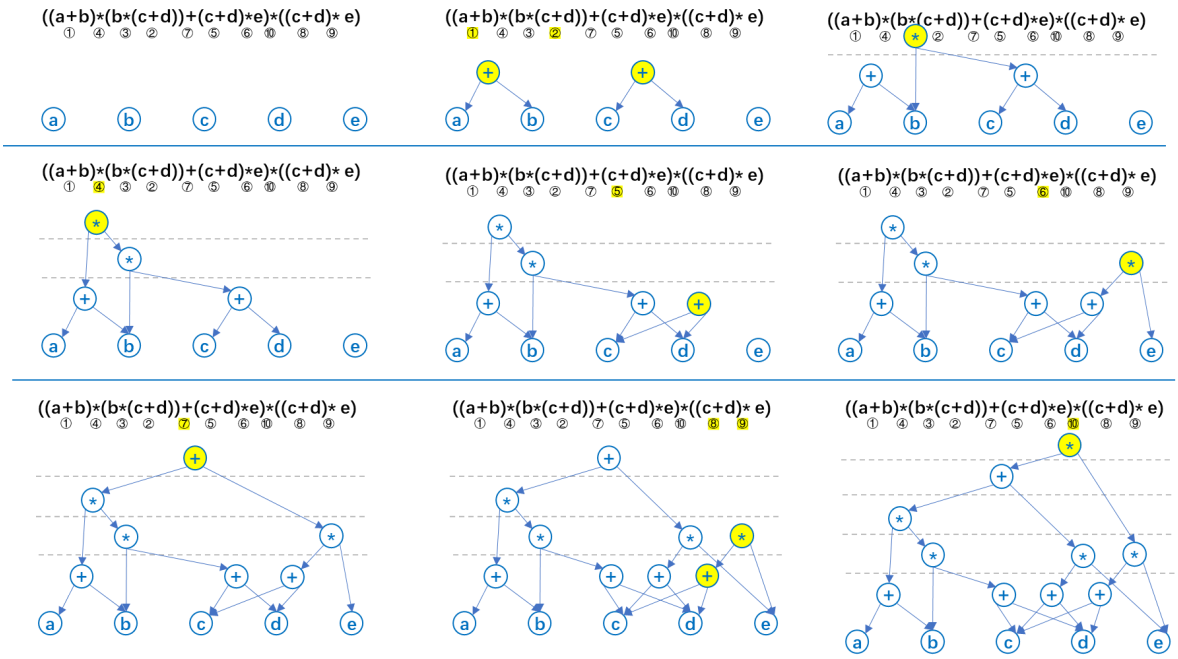
若一个有向图中不存在环，则称为有向无环图，简称 DAG图（Directed Acyclic Graph）。




手动构造有向无环图的步骤：

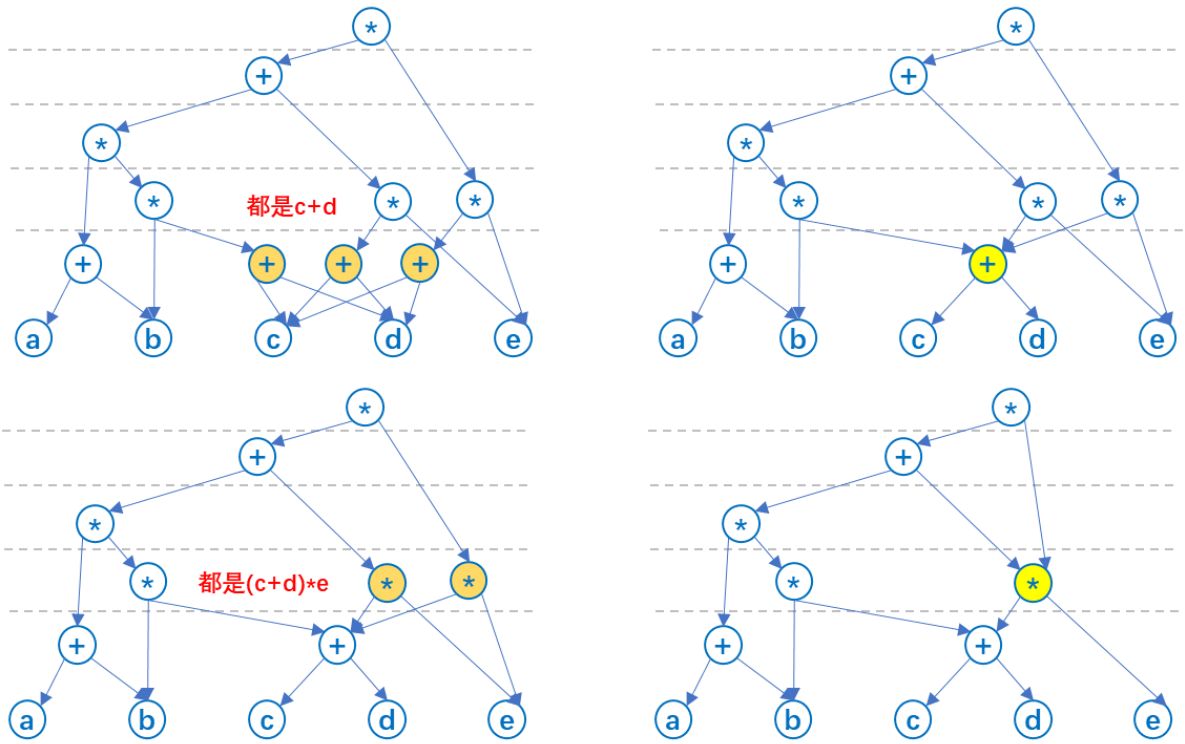
1. 把各个操作数不重复地排成一排
2. 标出各个运算符的生效顺序（不完全唯一）
3. 按顺序加入运算符，注意“分层”
4. 从底向上逐层检查同层的运算符是否可以合体。

手动构造有向无环图的示例如下：



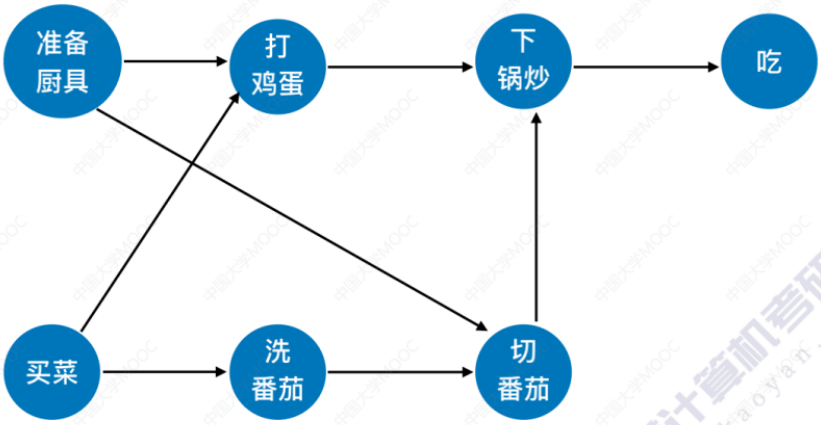

合并运算符

**((a+b)*(b*(c+d))+(c+d)*e)*((c+d)* e)**

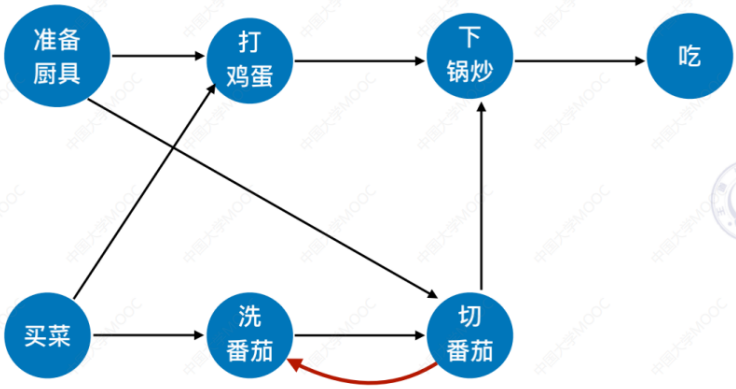



## 6.3.4 拓扑排序

AOV网：用顶点表示活动的网



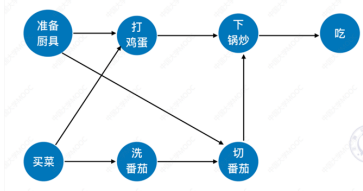

表示“番茄炒蛋工程”的AOV网

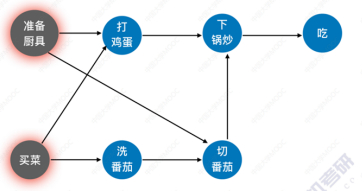


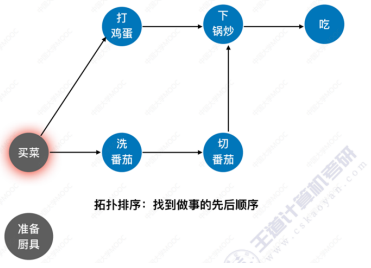

不是AOV网

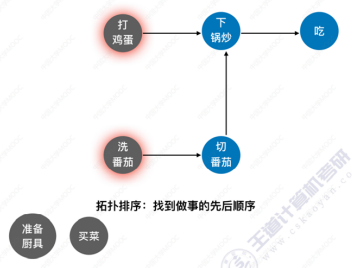
> 拓扑排序：找到**做事的先后顺序**

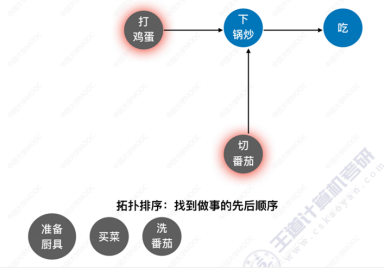
拓扑排序的一个过程如下：

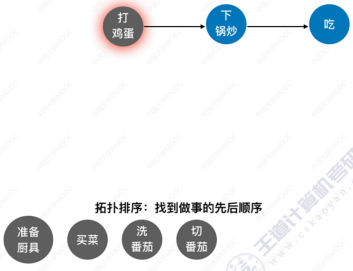



拓扑排序:在图论中，由一个有向无环图的顶点组成的序列，当且仅当满足下列条件时，称为该图的一个拓扑排序:

①每个顶点出现且只出现一次。

②若顶点A在序列中排在顶点B的前面，则在图中不存在从顶点B到顶点A的路径。

或定义为:

拓扑排序是对有向无环图的顶点的一种排序，它使得若存在一条从顶点A到顶点B的路径，则在排序中顶点B出现在顶点A的后面。每个AOV网都有一个或多个拓扑排序序列。

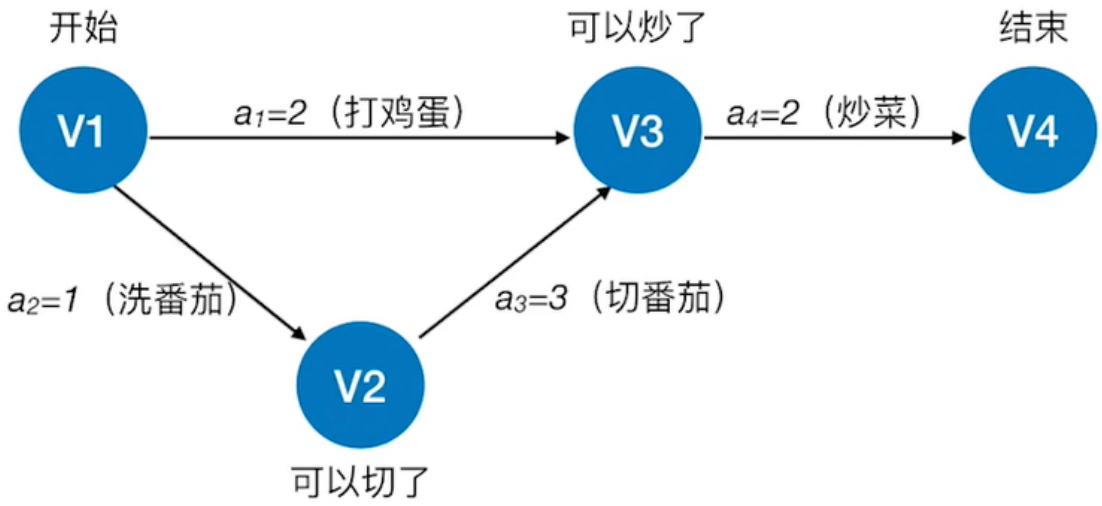
> 代码实现的文字描述如下：
>
> 1. 从 AOV 网中选择一个没有前驱（入度为0）的顶点并输出。
> 2. 从网中删除该顶点和所有以它为起点的有向边。
> 3. 重复 ① 和 ② 直到当前的 AOV 网为空或当前网中不存在无前驱的顶点为止。

```
#define MaxVertexNum 100              //图中顶点数目最大值

typedef struct ArcNode{               //边表结点
    int adjvex;                       //该弧所指向的顶点位置
    struct ArcNode *nextarc;          //指向下一条弧的指针
}ArcNode;

typedef struct VNode{                 //顶点表结点
    VertexType data;                  //顶点信息
    ArcNode *firstarc;                //指向第一条依附该顶点的弧的指针
}VNode,AdjList[MaxVertexNum];

typedef struct{
    AdjList vertices;                 //邻接表
    int vexnum,arcnum;                //图的顶点数和弧数
}Graph;                               //Graph是以邻接表存储的图类型

// 对图G进行拓扑排序
bool TopologicalSort(Graph G){
    InitStack(S);                     //初始化栈，存储入度为0的顶点
    for(int i=0;i<g.vexnum;i++){
        if(indegree[i]==0)
            Push(S,i);                //将所有入度为0的顶点进栈
    }
    int count=0;                      //计数，记录当前已经输出的顶点数
    while(!IsEmpty(S)){               //栈不空，则存入
        Pop(S,i);                     //栈顶元素出栈
        print[count++]=i;             //输出顶点i
        for(p=G.vertices[i].firstarc;p;p=p=->nextarc){
            //将所有i指向的顶点的入度减1，并将入度为0的顶点压入栈
```

```
            v=p->adjvex;
            if(!(--indegree[v]))
                Push(S,v);              //入度为0，则入栈
        }
    }
    if(count<G.vexnum)
        return false;                   //排序失败
    else
        return true;                    //排序成功
}
```

### 6.4.5 关键路径

AOE网：在带权有向图中，以顶点表示事件，以有向边表示活动，以边上的权值表示完成该活动的开销(如完成活动所需的时间)，称之为用边表示活动的网络，简称AOE网(Activity On Edge NetWork)




- AOE网具有以下两个性质：
  - 只有**在某顶点所代表的事件发生后**，从该顶点**出发**的各有向边所代表的活动才能开始；
  - 只有在**进入某顶点的各有向边所代表的活动都已结束时**，该顶点所代表的事件才能发生。
  - 另外，有些活动是可以**并行**进行的。
- 在AOE网中仅有一个入度为0的顶点，称为开始顶点（源点），它表示整个工程的开始；也仅有一个出度为0的顶点，称为结束顶点（汇点），它表示整个工程的结束。
- 从源点到汇点的有向路径可能有多条，所有路径中，具有最大路径长度的路径称为**关键路径**，而把关键路径上的活动称为**关键活动**
- **完成整个工程的最短时间就是关键路径的长度，若关键活动不能按时完成，则整个工程的完成时间就会延长**
  - 事件$v_k$的最早发生时间$ve(k)$：决定了所有从$v_k$开始的活动能够开工的最早时间。
  - 事件$v_k$的最迟发生时间$vl(k)$：它是指在不推迟整个工程完成的前提下，该事件最迟必须发生的时间。
  - 活动$a_i$的最早开始时间$e(i)$：指该活动弧的起点所表示的事件的最早发生时间。
  - 活动$a_i$的最迟开始时间$l(i)$：它是指该活动弧的终点所表示事件的最迟发生时间与该活动所需时间之差。
  - 活动$a_i$的时间余量$d(i) = l(i) - e(i)$：表示在不增加完成整个工程所需总时间的情况下，活动$a_i$可以拖延的时间。若一个活动的时间余量为0，则说明该活动必须要如期完成，否则会拖延整个工程的进度，所以称$l(i) - e(i) = 0$即$l(i) = e(i)$的活动为**关键活动**。由关键活动组成的路径称为**关键路径**。

1. 求所有事件的最早发生时间 $ve()$：按拓扑排序序列，依次求各个顶点的 $ve(k)$，$ve(源点)=0$，$ve(k)=Max\{ve(j)+Weight(v_j,v_k)\}$，$v_j$ 为 $v_k$ 的任意前驱。
2. 求所有事件的最迟发生时间 $vl()$：按逆拓扑排序序列，依次求各个顶点的 $vl(k)$，$vl(汇点)=ve(汇点)$，$vl(k)=Min\{vl(j)-Weight(v_k,v_j)\}$，$v_j$ 为 $v_k$ 的任意后继。
3. 求所有活动的最早发生时间 $e()$：若边 $<v_k,v_j>$ 表示活动 $a_i$，则有 $e(i)=ve(k)$。
4. 求所有活动的最迟发生时间 $l()$：若边 $<v_k,v_j>$ 表示活动 $a_i$，则有 $l(i)=vl(j)-Weight(v_k,v_j)$。
5. 求所有活动的时间余量 $d()$：$d(i)=l(i)-e(i)$。

- 关键活动、关键路径的特性：
  - 若关键活动耗时增加，则整个工程的工期将增长。
  - 缩短关键活动的时间，可以缩短整个工程的工期。
  - 当缩短到一定程度时，关键活动可能会变成非关键活动。
  - 可能有多条关键路径，只提高一条关键路径上的关键活动速度并不能缩短整个工程的工期，只有加快那些包括在所有关键路径上的关键活动才能达到缩短工期的目的。