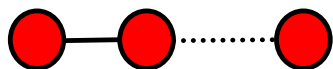


第七章 图

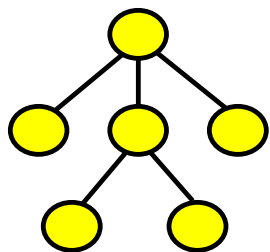
1. 图的定义和术语
2. 图的存储结构
3. 图的遍历
4. 最小生成树
5. 拓扑排序 关键路径
6. 最短路径

线性表



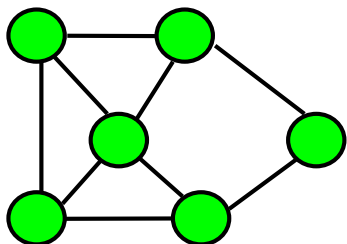
数据元素之间仅有线性关系，每个数据元素只有一个直接前驱和一个直接后继；

树形结构



数据元素之间有着明显的层次关系，并且每一层上的数据元素可能和下一层中多个元素(即其孩子结点)相关，但只能和上一层中一个元素(即其双亲结点)相关；

图形结构



结点之间的关系可以是任意的，图中任意两个数据元素之间都可能相关，是一种多对多的结构关系。

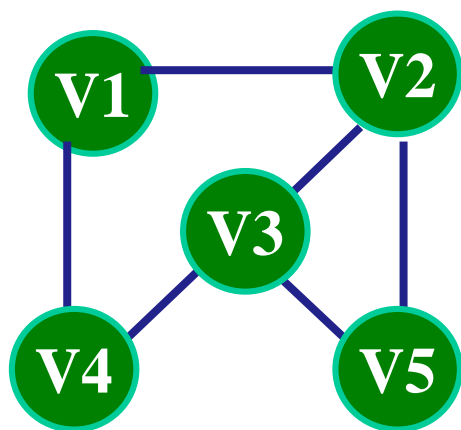
7.1 图的定义和术语

一. 图的定义

图 G (Graph)由两个集合构成, 记为 $G = (V, E)$

其中: V 是顶点(Vertex)的有穷非空集合;

E 是边(Edge)的有穷集合, 其中边是顶点的无序对或有序对。



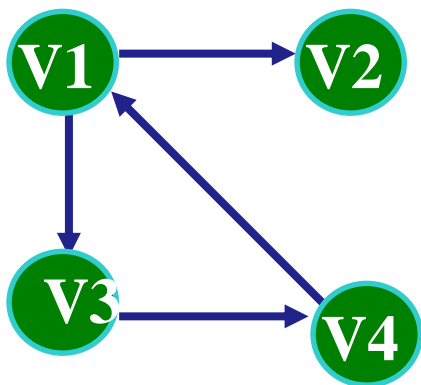
无向图 G_1

例1: $G_1 = (V_1, E_1)$

$V_1 = \{v_1, v_2, v_3, v_4, v_5\}$

$E_1 = \{ (v_1, v_2), (v_1, v_4), (v_2, v_3), (v_2, v_5), (v_3, v_4), (v_3, v_5) \}$

在无向图中, 顶点对 (v_i, v_j) 是无序的, 称为边, 用一对圆括号括起来。



有向图G2

例2 $G2=(V2, E2)$

$V2=\{v_1, v_2, v_3, v_4\}$

$E2=\{<v_1, v_2>, <v_1, v_3>, <v_3, v_4>, <v_4, v_1>\}$



在有向图中，顶点对 $<v_i, v_j>$ 是有序的，称为弧，用一对尖括号括起来。

v_i 是有向边的起点，称为弧尾，

v_j 是有向边的终点，称为弧头。

图的应用举例

例1 交通图（公路、铁路）

顶点：地点

边：连接地点的公路

交通图中的有单行道、双行道，分别用有向边、无向边表示；

例2 电路图

顶点：元件

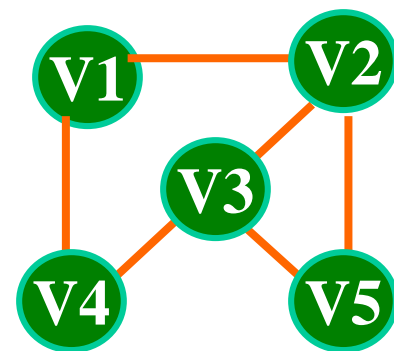
边：连接元件之间的线路

例3 各种流程图

如产品的生产流程图

顶点：工序

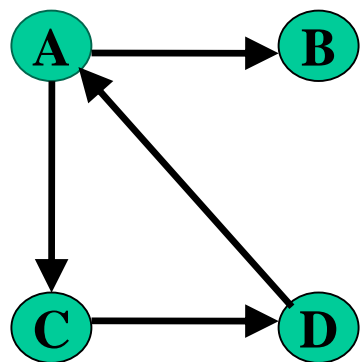
边(弧)：各道工序之间的顺序关系



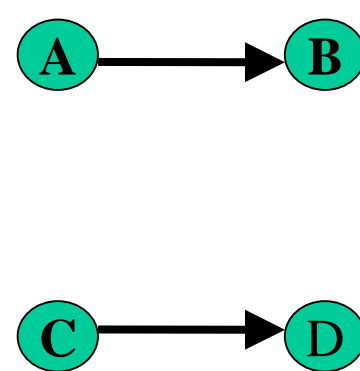
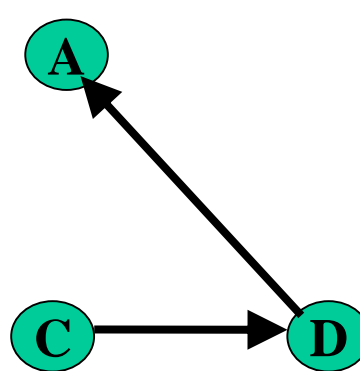
二. 图的基本术语

• 子图

设有两个图 $G = (V, E)$ 、 $G' = (V', E')$ ，若 $V' \subseteq V$ ， $E' \subseteq E$ ，则称 G' 是 G 的子图。



G_2

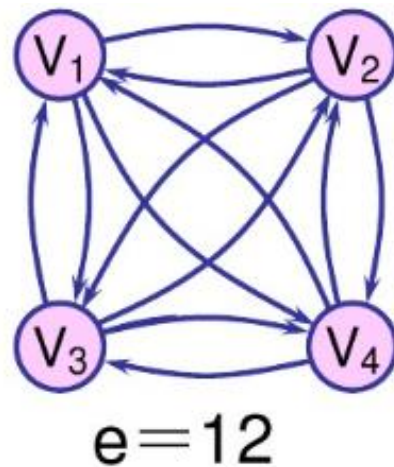
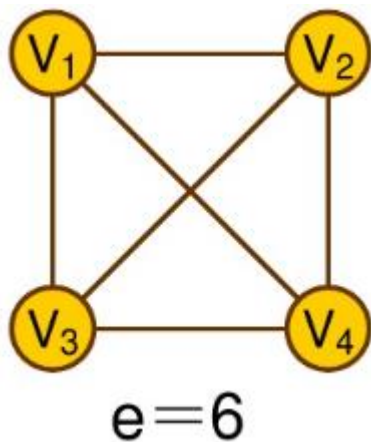


G_2 子图的一些例子

• 完全图

n 个顶点的无向图，若有 $n(n-1)/2$ 条边，则称为**无向完全图**；

n 个顶点的有向图，若有 $n(n-1)$ 条弧，则称为**有向完全图**。

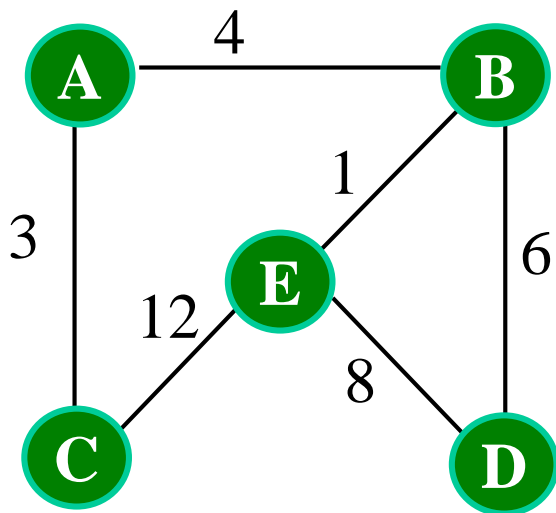


• 稀疏图、稠密图

有很少条边或弧 (如 $e < \log_2 n$) 的图称为稀疏图，反之称为稠密图。

- 权和网

在实际应用中，每条边可以标上具有某种含义的数值，该数值称为该边上的**权**。这种带权的图通常称为**网**。

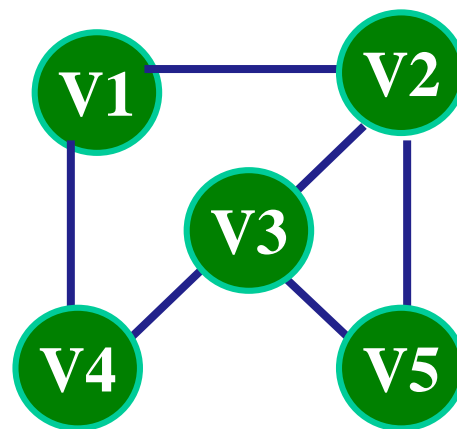


- 邻接点

对于无向图G，如果顶点v和w之间存在一条边 (v, w) ，则称v和w互为**邻接点**，这条边和顶点v、w**相关联**。

- 度、入度、出度

顶点 v 的度是指与 v 相关联的边的数目。

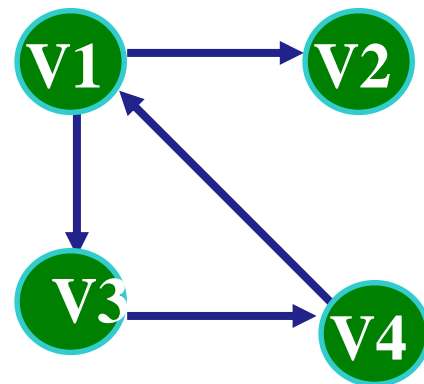


对于有向图，顶点 v 的度分为入度和出度。

入度=以顶点 v 为头的弧的数目

出度=以顶点 v 为尾的弧的数目

顶点 v 的度= v 的入度 + v 的出度



• 路径、路径长度

无向图G中，从顶点 v_1 到顶点 v_k 的**路径**是一个顶点序列 v_1, v_2, \dots, v_k ，其中 $(v_i, v_{i+1}) \in E$ ($i=1, 2, \dots, k-1$)。

如果G是有向图，则路径也是有向的，顶点序列应满足 $\langle v_i, v_{i+1} \rangle \in E$ 。

路径长度就是一条路径上经过的边或弧的数目。

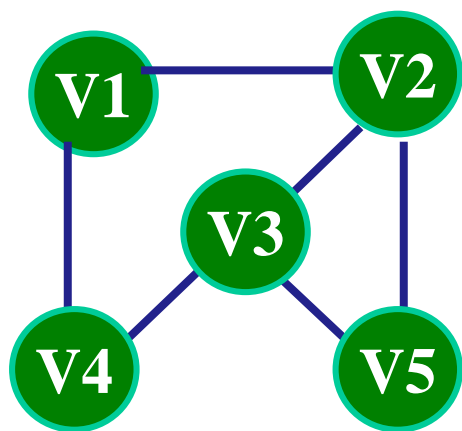


图1

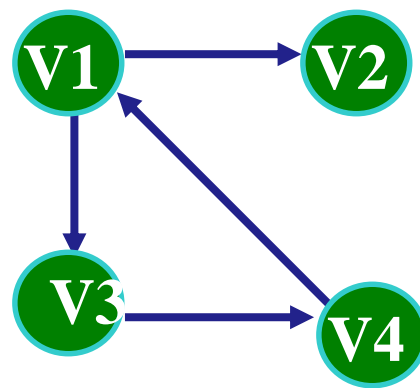


图2

图1： v_1, v_2, v_3, v_4 是 v_1 到 v_4 的路径，路径长度=3；

图2： v_1, v_3, v_4 是 v_1 到 v_4 的路径，路径长度=2；

- 回路或环

第一个顶点和最后一个顶点相同的路径称为回路或环。

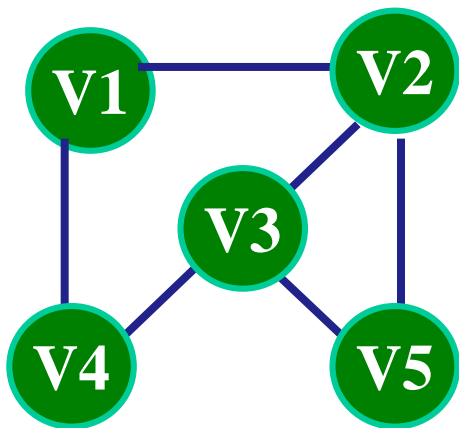


图1

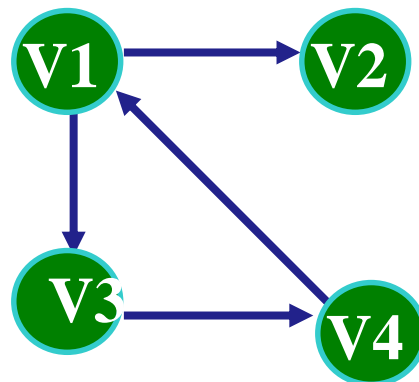


图2

图1: $V1, V2, V3, V5, V2, V1$ 是回路，路径长度=5；

图2: $V1, V3, V4, V1$ 是回路，路径长度=3；

- 简单路径、简单回路

序列中顶点不重复出现的路径称为简单路径；除了第一个顶点和最后一个顶点之外，其余顶点不重复出现的回路称简单回路。

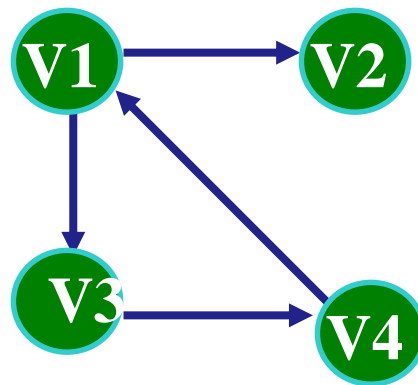
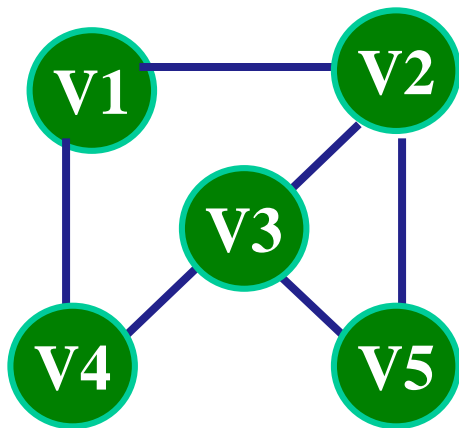
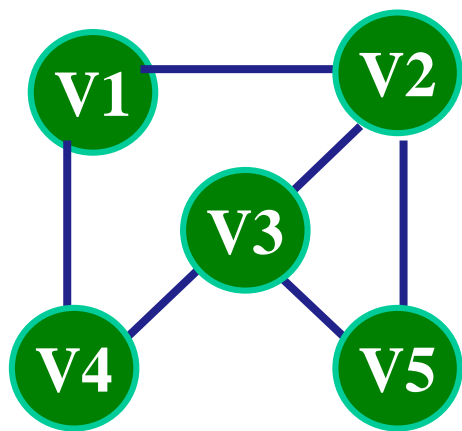


图1: $V1, V2, V3, V5, V2, V1$ 不是简单回路;

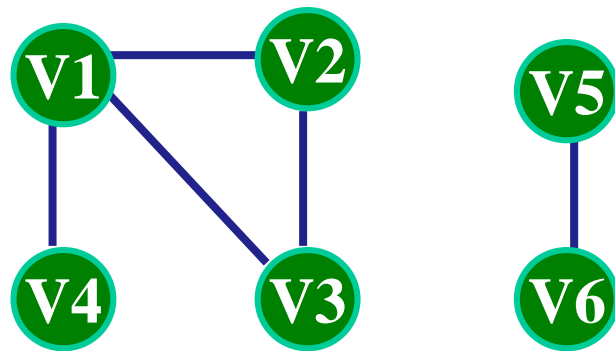
图2: $V1, V3, V4, V1$ 是简单回路。

- 连通、连通图、连通分量

在无向图 G 中，如果从顶点 v_i 到顶点 v_j 有路径，则称 v_i 和 v_j 是**连通**的。如果对于图中任意两个顶点 $v_i, v_j \in V$ ， v_i 和 v_j 都是连通的图，则称 G 是**连通图**。

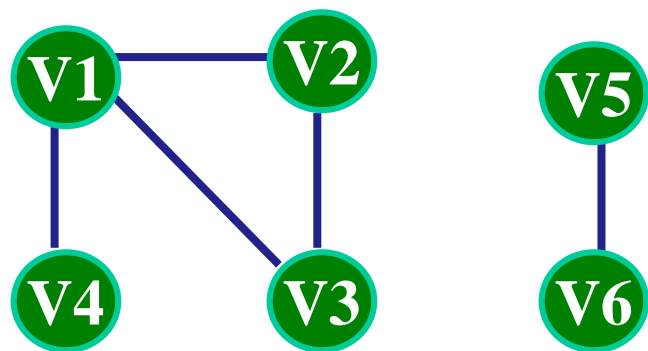


连通图 G_1

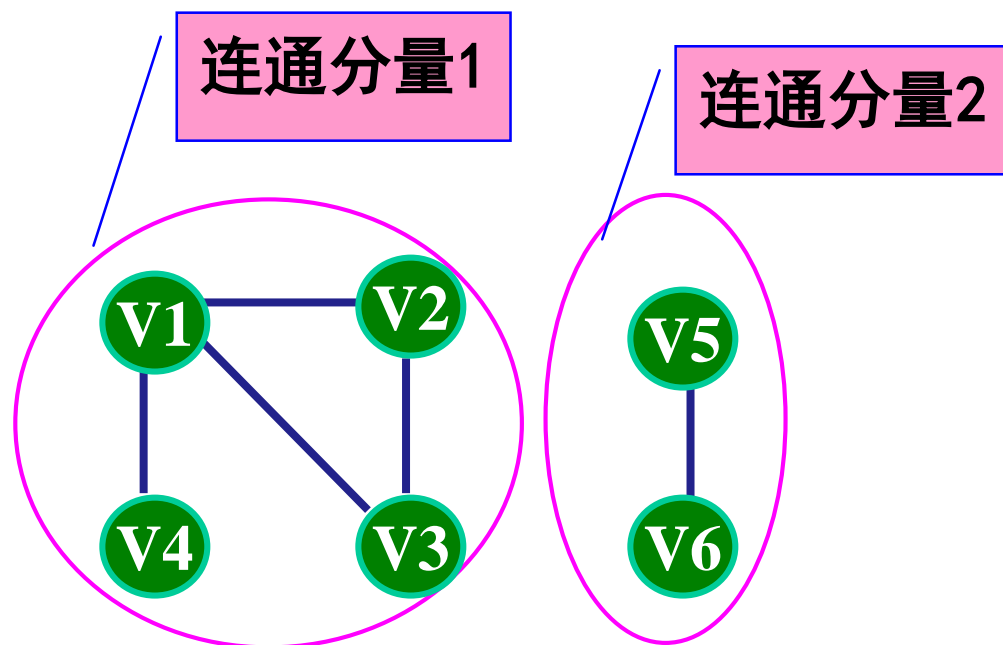


非连通图 G_3

无向图G的**极大**连通子图称为G的**连通分量**。



a. 非连通图 G_3



b. 图 G_3 的两个连通分量

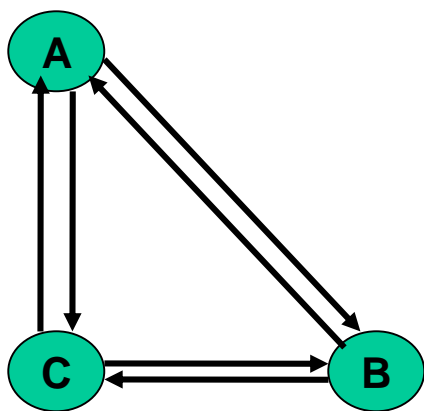
无向图及其连通分量

任何连通图的连通分量只有一个，即它本身，
非连通图的连通分量有多个。

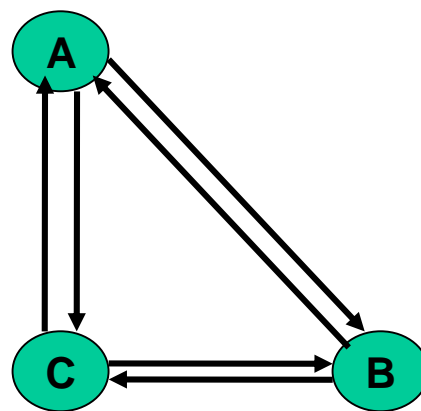
- 强连通图、强连通分量

在有向图 G 中，如果对于每一对 $v_i, v_j \in V$, $v_i \neq v_j$ ，从 v_i 到 v_j 和从 v_j 到 v_i 都存在路径，则称 G 为强连通图。

有向图 D 的极大强连通子图称为 D 的强连通分量。

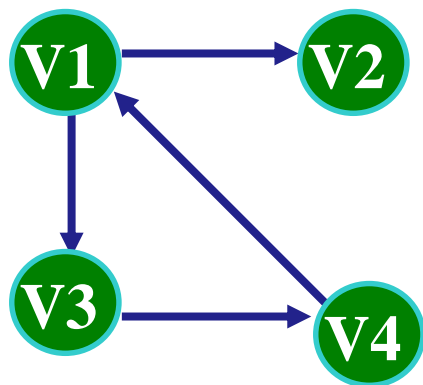


强连通图

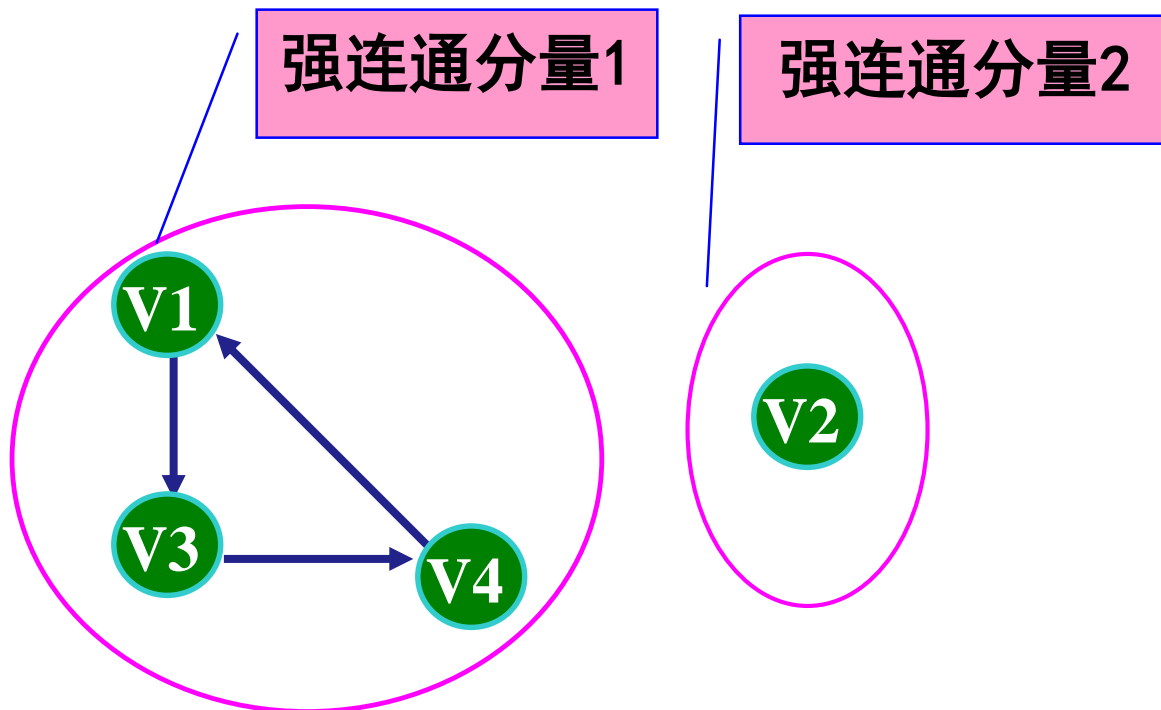


强连通分量

有向图G2不是强连通图，但它有两个强连通分量。



a. 图G2

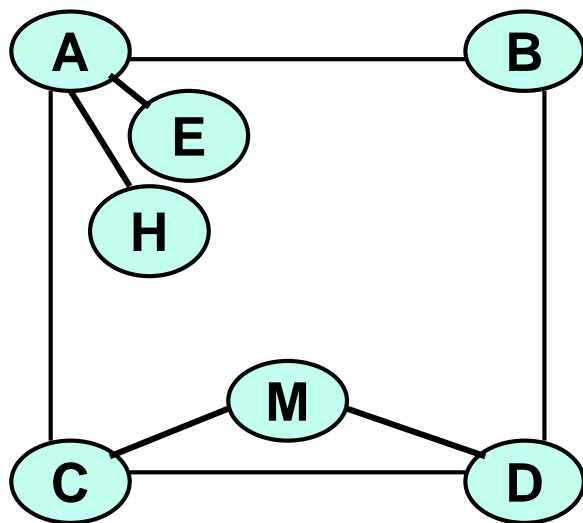


b. 图G2的两个强连通分量

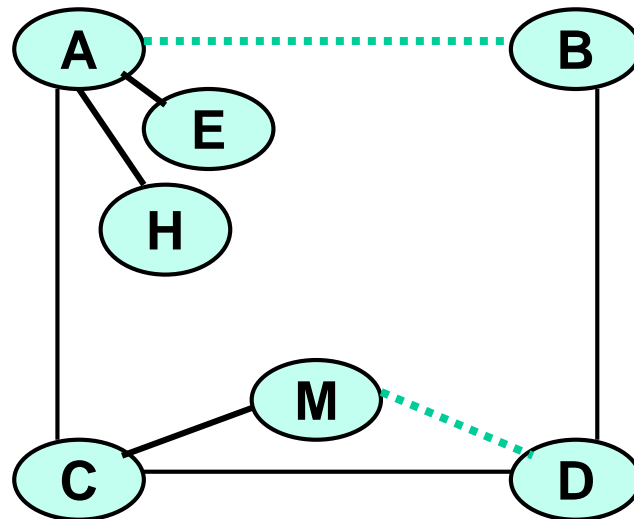
强连通图的连通分量只有一个，即它本身，
非强连通图的强连通分量有多个。

• 连通图的生成树

一个连通图的**生成树**是一个极小连通子图，它含有图中全部顶点，但只有足以构成一棵树的 $n-1$ 条边。



无向图

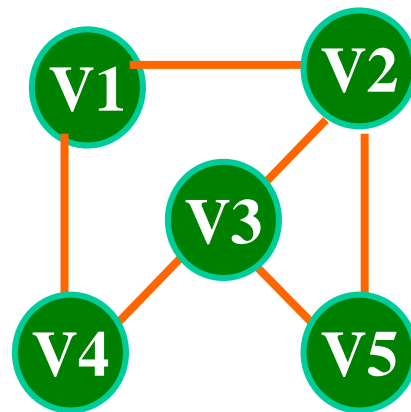


无向图的生成树

一个有 n 个顶点的连通图的生成树有且仅有 $n-1$ 条边。

- 生成树的顶点个数与图的顶点个数相同
- 一个有 n 个顶点的连通图的生成树有 $n-1$ 条边;含 n 个顶点 $n-1$ 条边的图不一定是生成树;
- 在生成树中再加一条边必然形成回路
- 一个图可以有許多棵不同的生成树;

基本操作P:



图的建立和销毁:

`CreateGraph (&G, V, VR);` // 按V和VR的定义构造图G。

`DestroyGraph (&G);` // 销毁图G。

对顶点的访问操作:

`LocateVex (G, u);` // 若G中存在顶点u, 则返回该顶点
// 在图中位置; 否则返回其它信息。

`GetVex (G, v);` // 返回v的值。

`PutVex (&G, v, value);` // 对v赋值value。

对邻接点的操作:

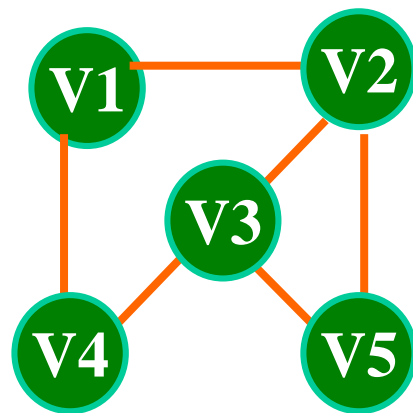
FirstAdjVex(G, v); // 返回 v 的第一个邻接点。若该顶点
//在 G 中没有邻接点, 则返回“空”。

NextAdjVex(G, v, w); //返回 v 的(相对于 w 的)下一个邻接点。
// 若 w 是 v 的最后一个邻接点, 则返回“空”。

插入或删除顶点

InsertVex(& G, v); //在图 G 中增添新顶点 v 。

DeleteVex(& G, v); //删除 G 中顶点 v 及其相关的弧。



插入和删除弧

InsertArc(& G, v, w); // 在 G 中增添弧 $\langle v, w \rangle$, 若 G 是无向的,
// 则还增添对称弧 $\langle w, v \rangle$ 。

DeleteArc(& G, v, w); //在 G 中删除弧 $\langle v, w \rangle$, 若 G 是无向的,
// 则还删除对称弧 $\langle w, v \rangle$ 。

遍历

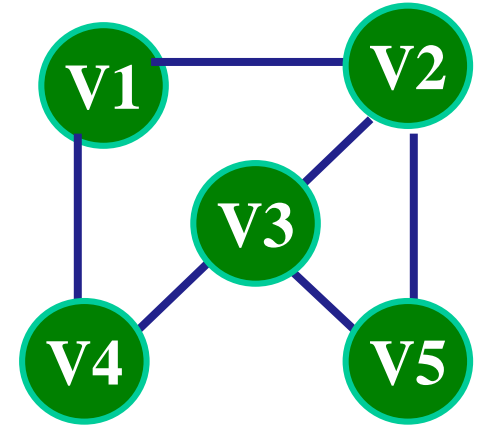
DFSTraverse(G, v, Visit()); // 从顶点v起深度优先遍历
//图G，并对每个顶点调用函数Visit一次且仅一次。

BFSTraverse(G, v, Visit()); // 从顶点v起广度优先遍历
//图G，并对每个顶点调用函数 Visit一次且仅一次。

7.2 图的存储结构

与线性结构、树结构一样，图的存储结构至少要保存**两类信息**：

- 1) 顶点的数据信息；
- 2) 顶点间的关系 (边或弧) 的信息；



图的四种常用的存储形式：

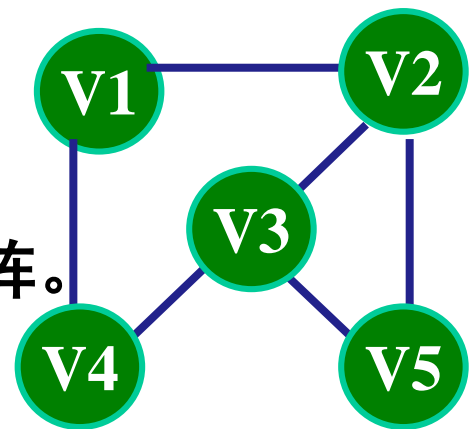
- 数组 (邻接矩阵) 表示法
- 邻接表
- 十字链表
- 邻接多重表

一. 数组(邻接矩阵)表示法

用**两个数组**分别存储顶点的信息和顶点之间的关系（边或弧）的信息。

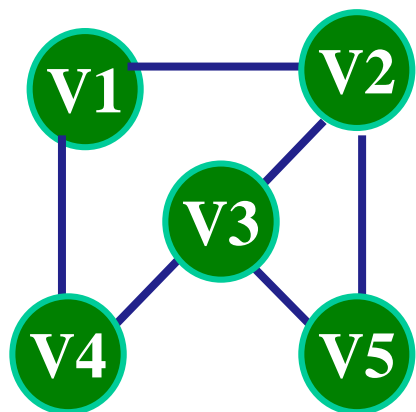
顶点信息的存储：用一维数组存储(按编号顺序)；

顶点间关系的存储：用二维数组存储图的邻接矩阵。



邻接矩阵：是表示顶点之间相邻关系的矩阵，设G是一个有 n 个顶点的图，则G的邻接矩阵是具有如下性质的 n 阶方阵：

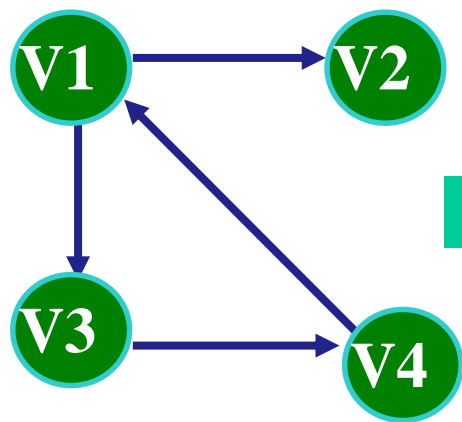
$$A[i][j] = \begin{cases} 1, & \text{若 } \langle v_i, v_j \rangle \text{ 或 } (v_i, v_j) \in E \\ 0, & \text{反之} \end{cases}$$



G1

邻接矩阵为

	v1	v2	v3	v4	v5
v1	0	1	0	1	0
v2	1	0	1	0	1
v3	0	1	0	1	1
v4	1	0	1	0	0
v5	0	1	1	0	0



G2

邻接矩阵为

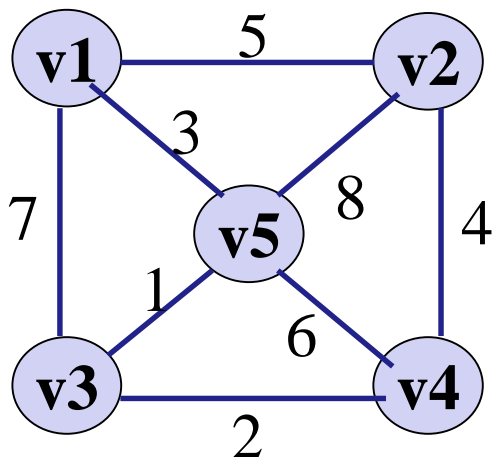
	v1	v2	v3	v4
v1	0	1	1	0
v2	0	0	0	0
v3	0	0	0	1
v4	1	0	0	0

如果G是网，则邻接矩阵可以定义为：

$$A[i][j] = \begin{cases} w_{ij}, & \text{若 } \langle v_i, v_j \rangle \text{ 或 } (v_i, v_j) \in E \\ \infty, & \text{反之} \end{cases}$$

其中， w_{ij} 表示边上的权值。

例：



网N

∞	5	7	∞	3
5	∞	∞	4	8
7	∞	∞	2	1
∞	4	2	∞	6
3	8	1	6	∞

邻接矩阵

```

// 图的数组(邻接矩阵)存储表示  教科书上
#define INFINITY INT_MAX           //表示极大值, 即 $\infty$ 
#define MaxN 20 // 最大顶点个数
typedef enum {DG, DN, AG, AN} GraphKind;
//{有向图,有向网,无向图,无向网}

typedef struct ArcCell { //边(弧)信息结构
    VRType adj; // VRType是顶点关系类型。
                // 对无权图, 用1或0表示相邻否;
                // 对带权图, 则为权值类型。
    InfoType *info; // 该弧相关信息的指针
} ArcCell, AdjMatrix[MaxN][MaxN];

typedef struct {
    VertexType vexs[MaxN]; // 存储顶点的一维数组
    AdjMatrix arcs;         // 存储邻接矩阵的二维数组
    int vexnum, arcnum; // 图的当前顶点数和弧(边)数
    GraphKind kind;       // 图的种类标志
} MGraph; //图的结构

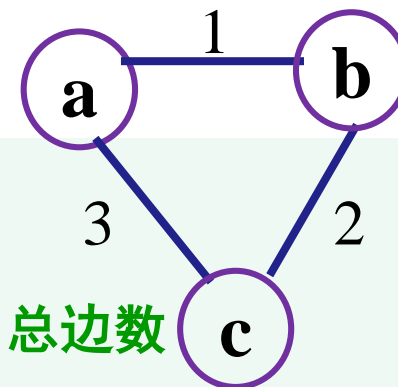
```

// 图的邻接矩阵存储表示

```
#define MaxInt INT_MAX      //表示极大值, 即 $\infty$ 
#define MaxN 20             // 最大顶点个数
typedef char VertexType;    //假设顶点的数据类型为字符型
typedef int ArcType;        //假设边的权值类型为整型

typedef struct {
    VertexType vexs[MaxN];  // 存储顶点的一维数组
    ArcType arcs[MaxN][MaxN]; // 存储邻接矩阵的二维数组
    int vexnum, arcnum; // 图的当前顶点数和弧(边)数
} MGraph; //图的结构
```

采用邻接矩阵表示法创建无向网



```
void CreatUDN(MGraph &G)
{ //采用邻接矩阵表示法, 构造无向网G
  scanf(&G.vexnum, &G.arcnum); //输入总顶点数, 总边数
  for(i=0; i<G.vexnum; ++i)
    scanf(&G.vexs[i]); //依次输入顶点的信息
  for (i=0; i<G.vexnum;++i) //初始化邻接矩阵, 边的权值均置为极大值MaxInt
    for(j=0;j<G.vexnum;++j)
      G.arcs[i][j]=MaxInt;
```

G.vexs

0	a
1	b
2	c
3	
4	
...	

G.arcs

	0	1	2
0	∞	∞	∞
1	∞	∞	∞
2	∞	∞	∞

```

for (k=0; k< G.arcnum; ++k) {
    scanf(&v1, &v2,&w);
    i=LocateVex(G, v1);
    j=LocateVex(G,v2);
    G.arcs[i][j]=w;
    G.arcs[j][i]=G.arcs[i][j];
}
}

```

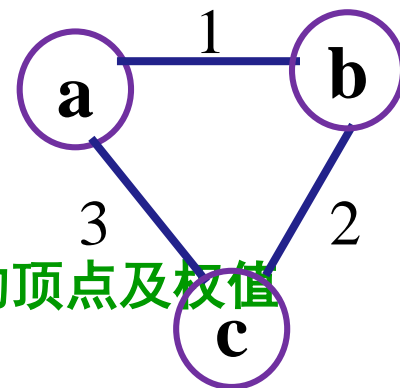
//构造邻接矩阵

//输入一条边依附的顶点及权值

//确定v1和v2在G中的位置

//边(v1,v2)的权值置为w

//置(v1,v2)的对称边(v2,v1)的权值为w



G.vexs

0	a
1	b
2	c
3	
4	
...	

G.arcs

	0	1	2
0	∞	∞	∞
1	∞	∞	∞
2	∞	∞	∞

```

for (k=0; k< G.arcnum; ++k) {
    scanf(&v1, &v2,&w);
    i=LocateVex(G, v1);
    j=LocateVex(G,v2);
    G.arcs[i][j]=w;
    G.arcs[j][i]=G.arcs[i][j];
}
}

```

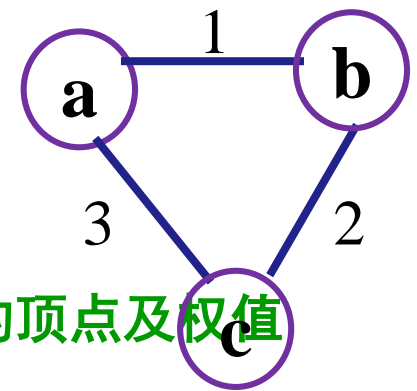
//构造邻接矩阵

//输入一条边依附的顶点及权值

//确定v1和v2在G中的位置

//边(v1,v2)的权值置为w

//置(v1,v2)的对称边(v2,v1)的权值为w



G.vexs

0	a
1	b
2	c
3	
4	
...	

G.arcs

	0	1	2
0	∞	1	3
1	1	∞	2
2	3	2	∞

采用邻接矩阵表示法创建无向网

```
void CreatUDN(MGraph &G)
{ //采用邻接矩阵表示法, 构造无向网G
  scanf(&G.vexnum, &G.arcnum); //输入总顶点数, 总边数
  for(i=0; i<G.vexnum; ++i)
    scanf(&G.vexs[i]); //依次输入顶点的信息
  for (i=0; i<G.vexnum;++i) //初始化邻接矩阵, 边的权值均置为极大值MaxInt
    for(j=0;j<G.vexnum;++j)
      G.arcs[i][j]=MaxInt;
  for (k=0; k< G.arcnum; ++k) { //构造邻接矩阵
    scanf(&v1, &v2,&w); //输入一条边依附的顶点及权值
    i=LocateVex(G, v1);
    j=LocateVex(G,v2); //确定v1和v2在G中的位置
    G.arcs[i][j]=w; //边(v1,v2)的权值置为w
    G.arcs[j][i]=G.arcs[i][j]; //置(v1,v2)的对称边(v2,v1)的权值为w
  }
}
```

无向图数组表示法特点：

1) 无向图邻接矩阵是**对称矩阵**，同一条边表示了两次；

2) **顶点v的度**：

等于二维数组对应行（或列）中1的个数；

3) **判断两顶点v、u是否为邻接点**：

只需判二维数组对应分量是否为1；

4) **顶点不变，在图中增加、删除边**：

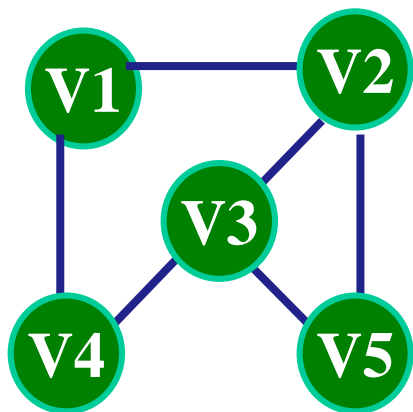
只需对二维数组对应分量赋值1或清0；

5) **设图的顶点数为m，G占用存储空间**：

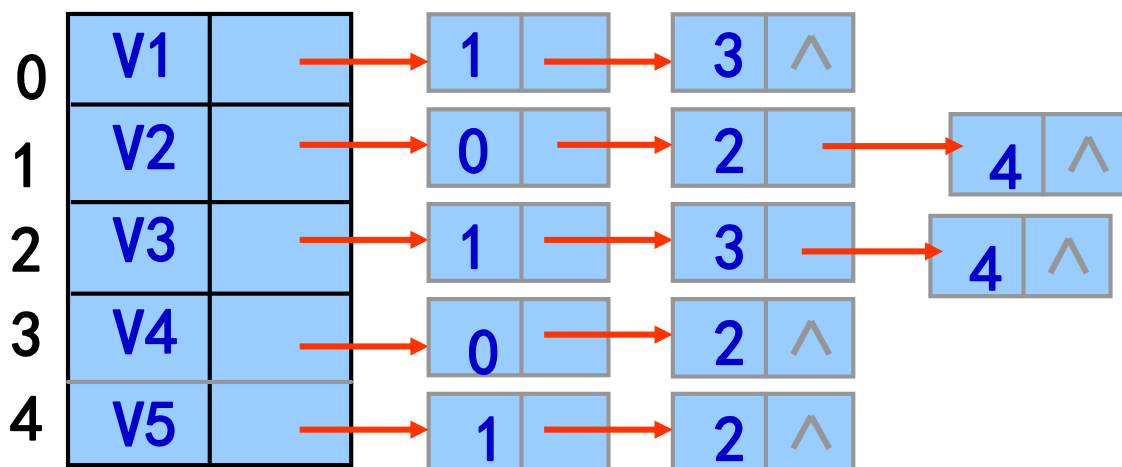
$m+m^2$ ；**G占用存储空间只与它的顶点数有关，与边数无关**；
适用于稠密图；

二. 邻接表表示法

- **顶点**：通常按编号顺序将顶点数据存储在一维数组中；
- **依附于同一顶点的边**：用线性链表存储

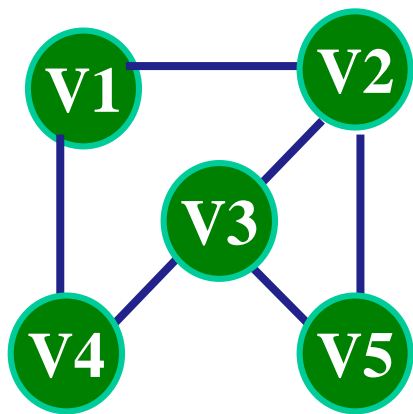


G1

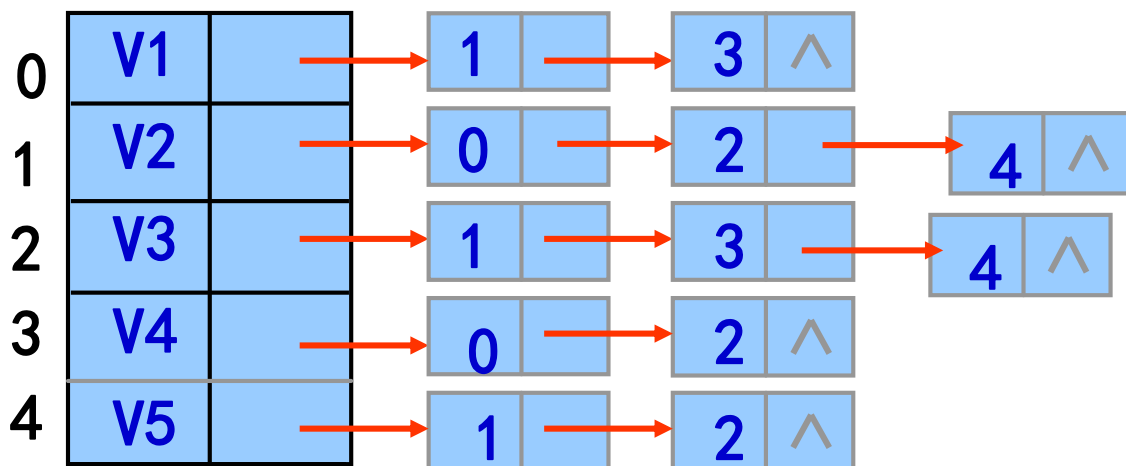


图G1的邻接表

在邻接表中，对图中每个顶点 v_i 建立一个单链表，把与 v_i 相邻接的顶点放在这个链表中。邻接表中每个单链表的第一个顶点存放有关顶点的信息，把这一结点看成链表的表头，其余结点存放有关边的信息，这样邻接表中便有两种结点：表头结点和边结点。



G1



图G1的邻接表

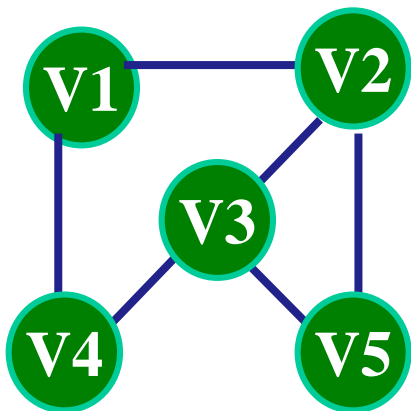
表头结点： 包括数据域 (data) 和链域 (firstarc) 两部分。



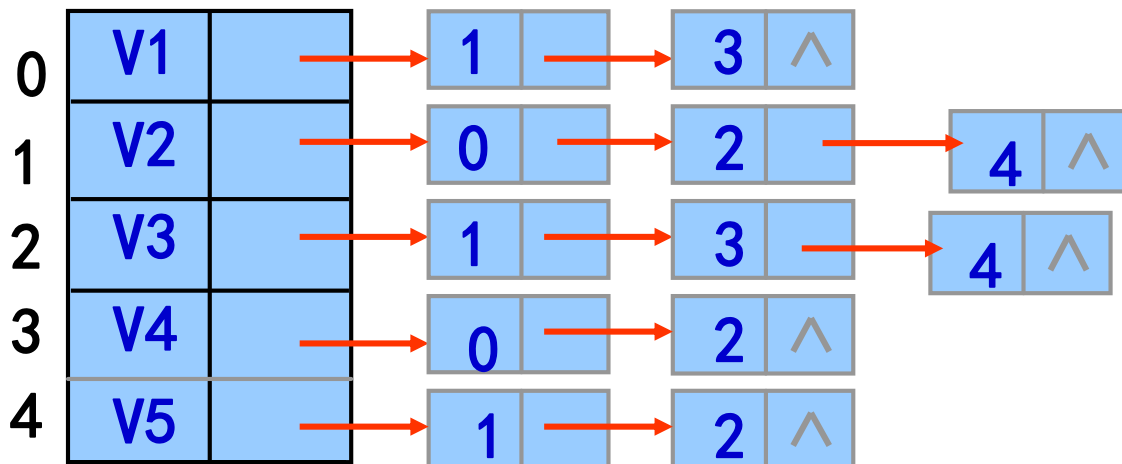
存储顶点 v_i 的名称
或其他有关信息

指向链表中第一个边结点。

所有表头结点以顺序结构的形式存储，以便可以随机访问任一顶点的链表。



G1



图G1的邻接表

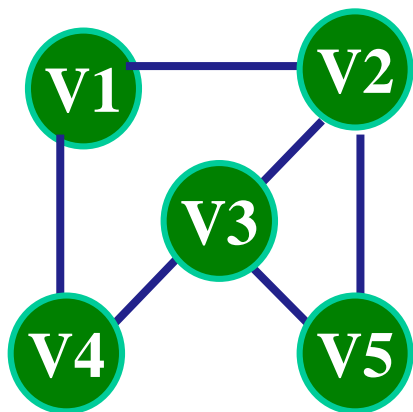
链表中的**边结点**：包括邻接点域(adjvex)、数据域(info)和链域(nextarc)。



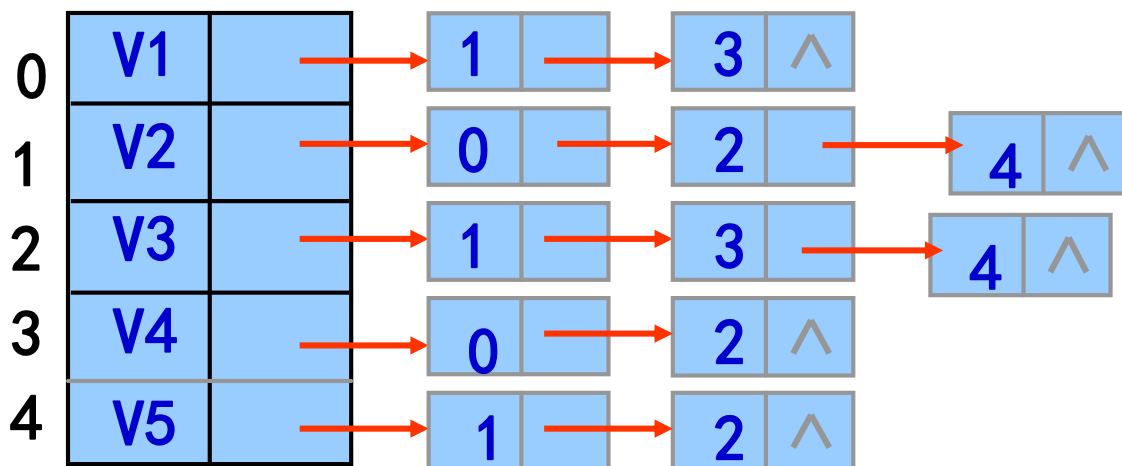
指示与顶点 v_i 邻接的
点在图中的位置

存储与边相关的
信息，如权值

指示与顶点 v_i 邻接的
下一条边的结点

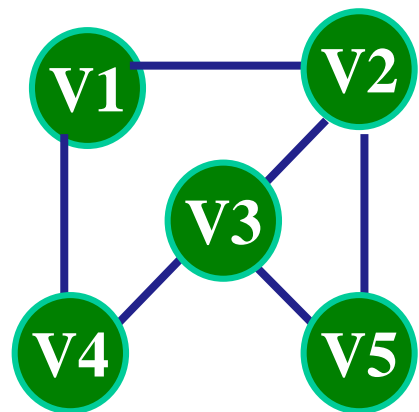


G1

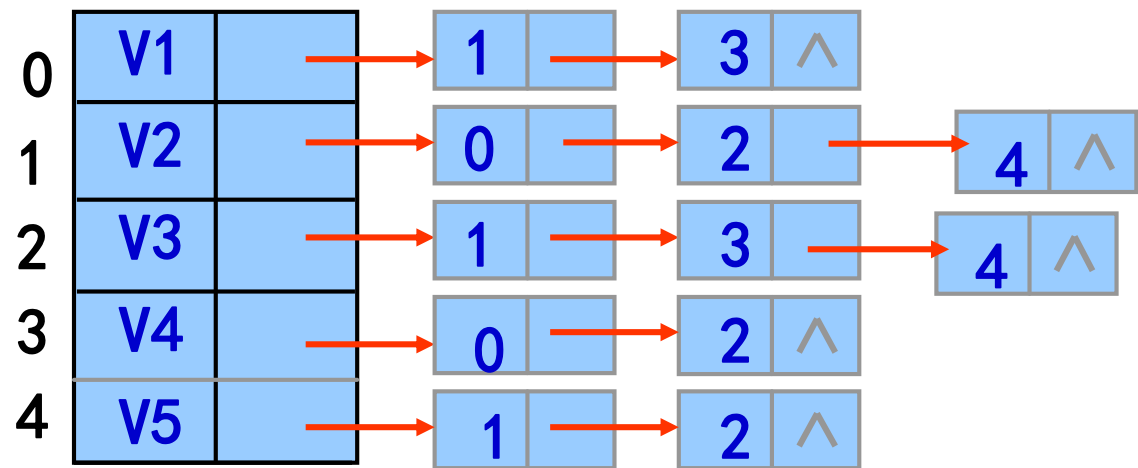


图G1的邻接表

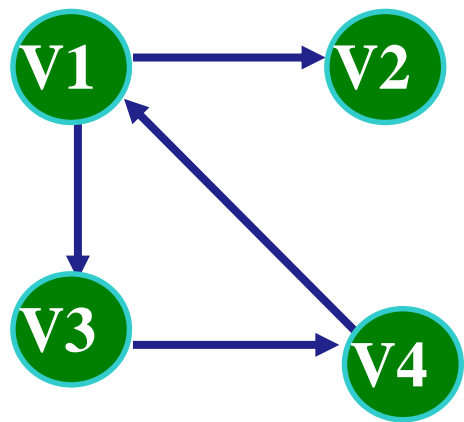
下图为图G1、G2及其邻接表：



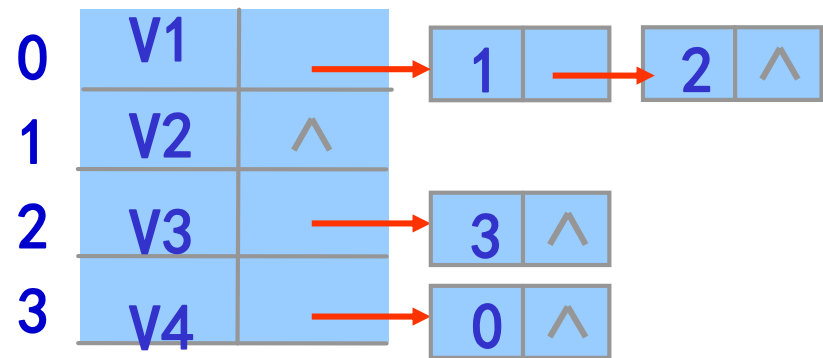
G1



图G1的邻接表

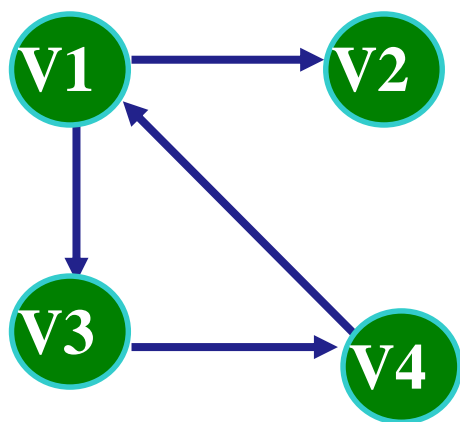


G2



图G2的邻接表

在有向图中，第 i 个链表中的结点个数只是顶点 v_i 的出度，为求入度，必须遍历整个邻接表。有时，为了便于确定顶点的入度，可以建立一个有向图的逆邻接表，即对每个顶点 v_i 建立一个链接所有进入 v_i 的边的表。下图为有向图G1和它的逆邻接表。

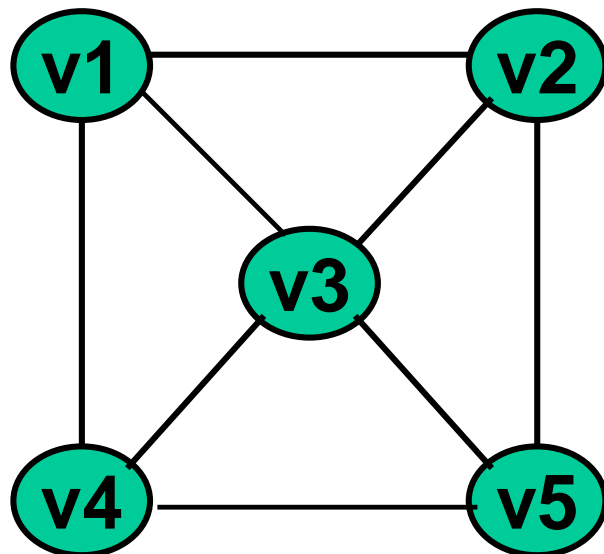


G2

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

图G2的逆邻接表

求下图所示的无向图G的邻接表：



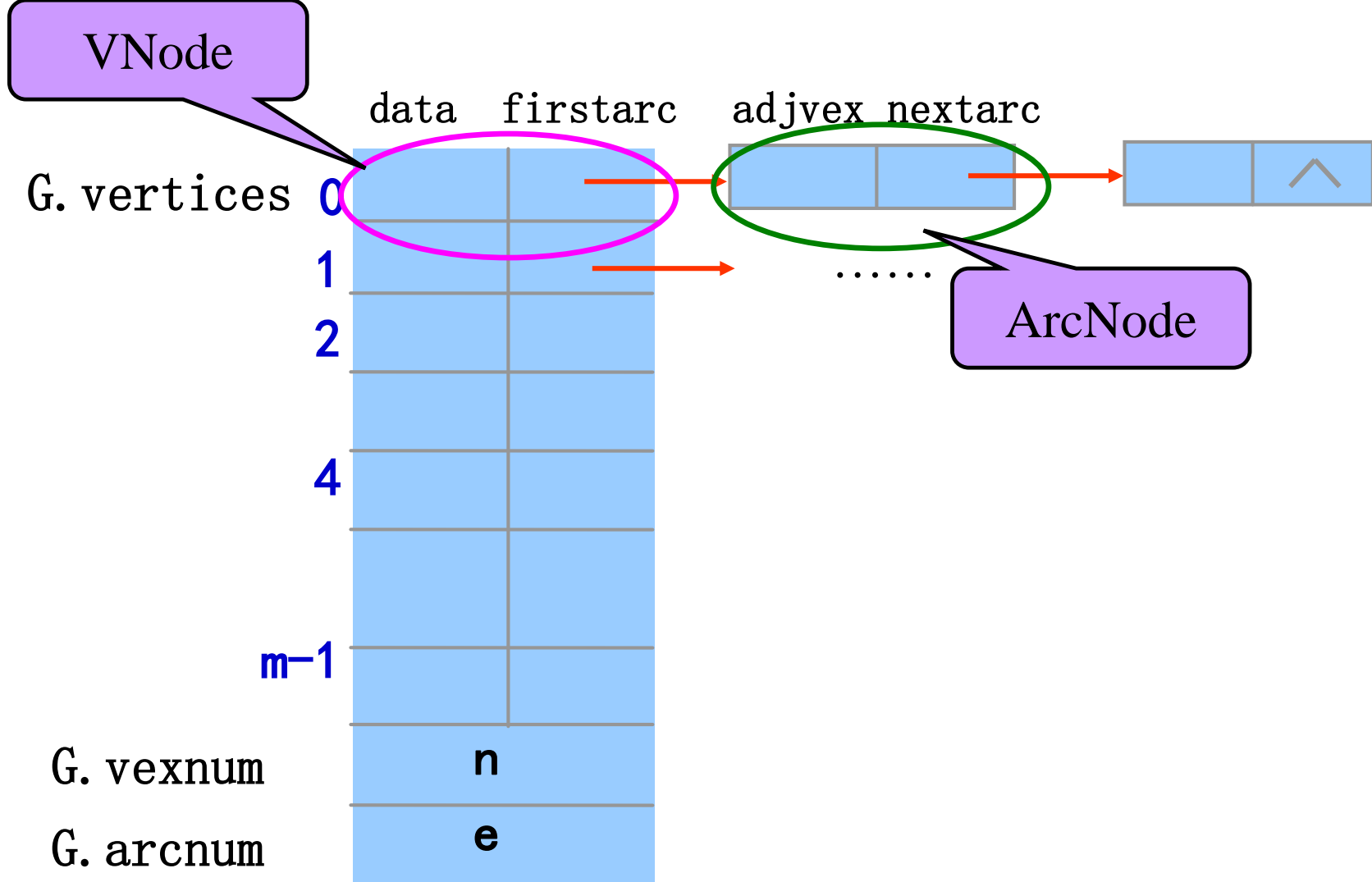
//图的邻接表存储表示

```
#define MaxN 20          // 最大顶点个数

typedef struct ArcNode {   //边结点的类型定义
    int          adjvex;   // 边的另一顶点的在数组中的位置
    struct ArcNode *nextarc; // 指向下一条边的指针
    InfoType      info;    // 该边相关信息
} ArcNode;

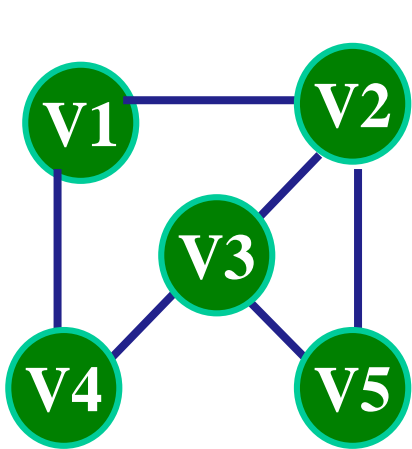
typedef struct VNode {     //顶点结点
    VertexType      data;   // 顶点信息
    ArcNode          *firstarc; //指向关联该顶点的边链表
} VNode, AdjList[MaxN];

typedef struct {
    AdjList          vertices;
    int              vexnum, arcnum; // 图的当前顶点数和边数
} ALGraph;
```

图的邻接表存储结构

设G是ALGraph 类型的变量，用于存储无向图G1，该图有5个顶点，6条边, G的邻接表图示如下：



无向图G1

G. vertices

	VNode	ArcNode			
	data	firstarc	adjvex	nextarc	
0	V1	→	1	→	3 ^
1	V2	→	0	→	2 → 4 ^
2	V3	→	1	→	3 → 4 ^
3	V4	→	0	→	2 ^
4	V5	→	1	→	2 ^

图G1的邻接表

//采用邻接表表示法创建无向图

```
void CreatGraph(ALGraph &G)
```

```
{
```

```
    scanf(&G.vexnum,&G.arcnum); //输入总顶点数，总边数
```

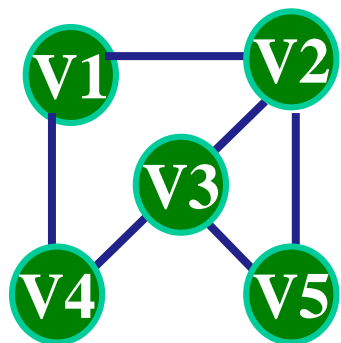
```
    for(i=0;i<G.vexnum;i++) //输入各顶点，构造表头结点
```

```
    {
```

```
        scanf(&G.vertices[i].data);
```

```
        G.vertices[i].firstarc=NULL;
```

```
    }
```



G. vertices

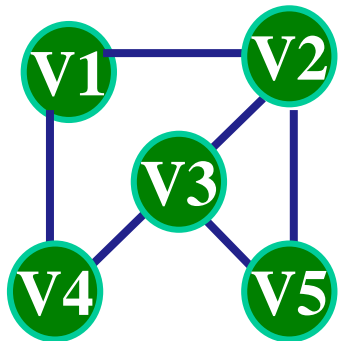
VNode

	data	firstarc
0	V1	^
1	V2	^
2	V3	^
3	V4	^
4	V5	^

```

for(k=0;k<G.arcnum;k++) //输入各边，构造邻接表
{
    scanf(&v1,&v2); //输入一条边依附的两个顶点
    i=LocateVex(G,v1);
    j=LocateVex(G,v2); //确定v1、v2在G中的位置
    p1=(ArcNode *)malloc(sizeof(ArcNode)); //生成一个新的边结点
    p1->adjvex=j; //邻接点序号为j
    p1->nextarc=G.vertices[i].firstarc;
    G.vertices[i].firstarc=p1; //将新结点插入到顶点vi的边链表头部
}

```



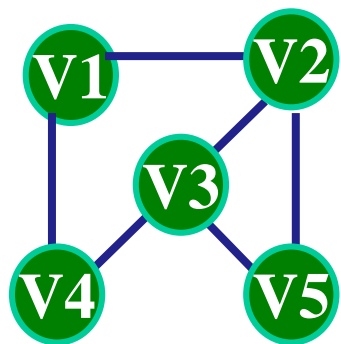
G. vertices

	VNode	
	data	firstarc
0	V1	^
1	V2	^
2	V3	^
3	V4	^
4	V5	^

```

for(k=0;k<G.arcnum;k++) //输入各边，构造邻接表
{
    scanf(&v1,&v2); //输入一条边依附的两个顶点
    i=LocateVex(G,v1);
    j=LocateVex(G,v2); //确定v1、v2在G中的位置
    p1=(ArcNode *)malloc(sizeof(ArcNode)); //生成一个新的边结点
    p1->adjvex=j; //邻接点序号为j
    p1->nextarc=G.vertices[i].firstarc;
    G.vertices[i].firstarc=p1; //将新结点插入到顶点vi的边链表头部
}

```



G. vertices

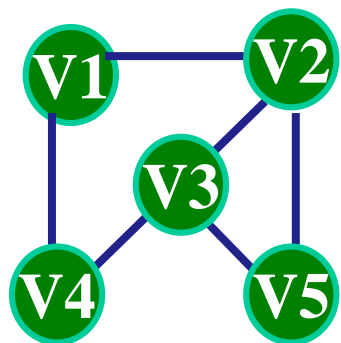
	VNode		ArcNode	
	data	firstarc	adjvex	nextarc
0	V1		1	^
1	V2	^		
2	V3	^		
3	V4	^		
4	V5	^		

```

p2=(ArcNode *)malloc(sizeof(ArcNode));
    //生成另一对称的新的边结点

p2->adjvex=i;
p2->nextarc=G.vertices[j].firstarc;
G.vertices[j].firstarc=p2; ;    //将新结点插入到顶点vj的边链表头部
}
}

```



G. vertices

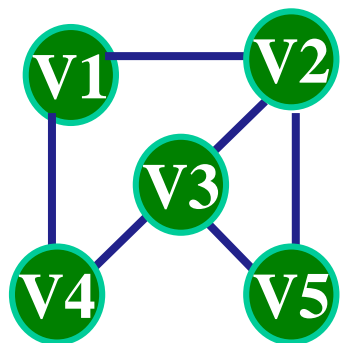
	VNode		ArcNode	
	data	firstarc	adjvex	nextarc
0	V1		1	^
1	V2	^		
2	V3	^		
3	V4	^		
4	V5	^		

```



p2=(ArcNode *)malloc(sizeof(ArcNode));
    //生成另一对称的新的边结点

p2->adjvex=i;
p2->nextarc=G.vertices[j].firstarc;
G.vertices[j].firstarc=p2; ;    //将新结点插入到顶点vj的边链表头部
}
}

```



G. vertices

	VNode		ArcNode	
	data	firstarc	adjvex	nextarc
0	V1		1	^
1	V2		0	^
2	V3	^		
3	V4	^		
4	V5	^		

//采用邻接表表示法创建无向图

void CreatGraph(ALGraph &G)

```
{    scanf(&G.vexnum,&G.arcnum); //输入总顶点数，总边数
    for(i=0; i<G.vexnum; i++) //输入各顶点，构造表头结点
    {    scanf(&G.vertices[i].data);
        G.vertices[i].firstarc=NULL;
    }
    for(k=0; k<G.arcnum; k++) //输入各边，构造邻接表
    {    scanf(&v1,&v2); //输入一条边依附的两个顶点
        i=LocateVex(G,v1);
        j=LocateVex(G,v2); //确定v1、v2在G中的位置
        p1=(ArcNode *)malloc(sizeof(ArcNode)); //生成一个新的边结点
        p1->adjvex=j; //邻接点序号为j
        p1->nextarc=G.vertices[i].firstarc;
        G.vertices[i].firstarc=p1; //将新结点插入到顶点vi的边链表头部
        p2=(ArcNode *)malloc(sizeof(ArcNode)); //生成另一对称的新的边结点
        p2->adjvex=i;
        p2->nextarc=G.vertices[j].firstarc;
        G.vertices[j].firstarc=p2; //将新结点插入到顶点vj的边链表头部
    }
}
```


无向图的邻接表的特点：

1) 在G邻接表中，同一条边对应两个结点；

2) 顶点 v 的度：

等于 v 对应线性链表的长度；

3) 判定两顶点 v ， u 是否邻接：

要看 v 对应线性链表中有无对应的结点；

4) 在G中增减边：

要在两个单链表插入、删除结点；

5) 设图的顶点数为 m ，图的边数为 e ，G占用存储空间为：

$m+2*e$ 。G占用存储空间与G的顶点数、边数均有关；适用于边稀疏的图；

一个图的邻接矩阵是唯一的，但其邻接表不唯一。

三. 有向图的十字链表

在十字链表中，对应于有向图中每一条弧有一个结点，对应于每个顶点也有一个结点。这些结点的结构如下所示：

弧结点

tailvex	headvex	hlink	tlink	info
---------	---------	-------	-------	------

尾域(tailvex): 指示弧尾在图中的位置;
头域(headvex): 指示弧头在图中的位置;
链域hlink: 指向弧头相同的下一条弧;
链域tlink: 指向弧尾相同的下一条弧;
info域: 指向该弧的相关信息。

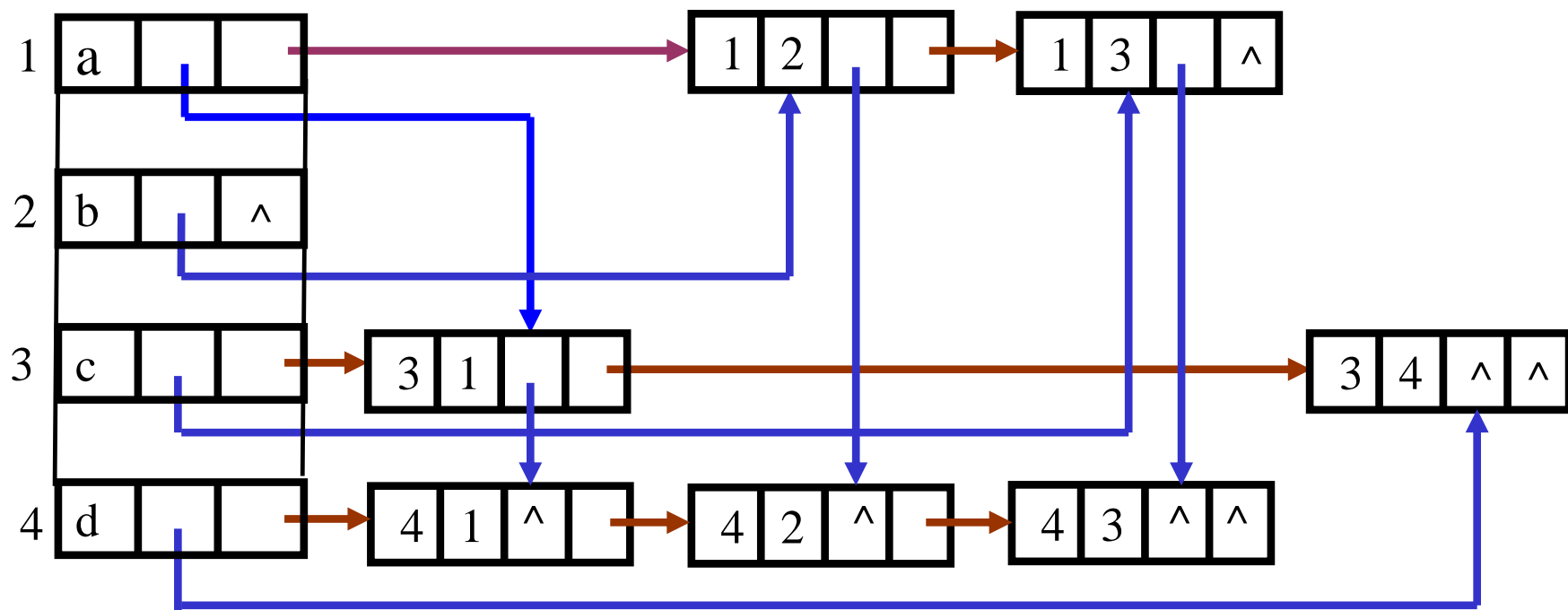
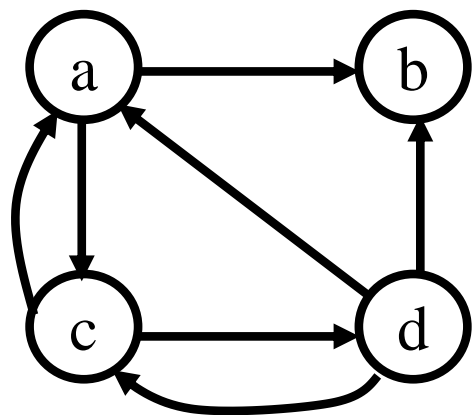
弧头相同的弧在同一链表上，弧尾相同的弧也在同一链表上。它们的头结点即为顶点结点。

顶点结点

data	firstin	firstout
------	---------	----------

data域: 存储和顶点相关的信息;
链域firstin: 指向以该顶点为弧头的第一个弧结点;
链域firstout: 指向以该顶点为弧尾的第一个弧结点。

例



4. 无向图的邻接多重表

邻接多重表的结构和十字链表类似。在邻接多重表中，每一条边用一个结点表示。

mark	ivex	ilink	ivex	jlink	info
------	------	-------	------	-------	------

mark: 为标志域，可用以标记该条边是否被搜索过；

ivex和**ivex:** 为该边依附的两个顶点在图中的位置；

ilink: 指向下一条依附于顶点ivex的边；

jlink: 指向下一条依附于顶点ivex的边，

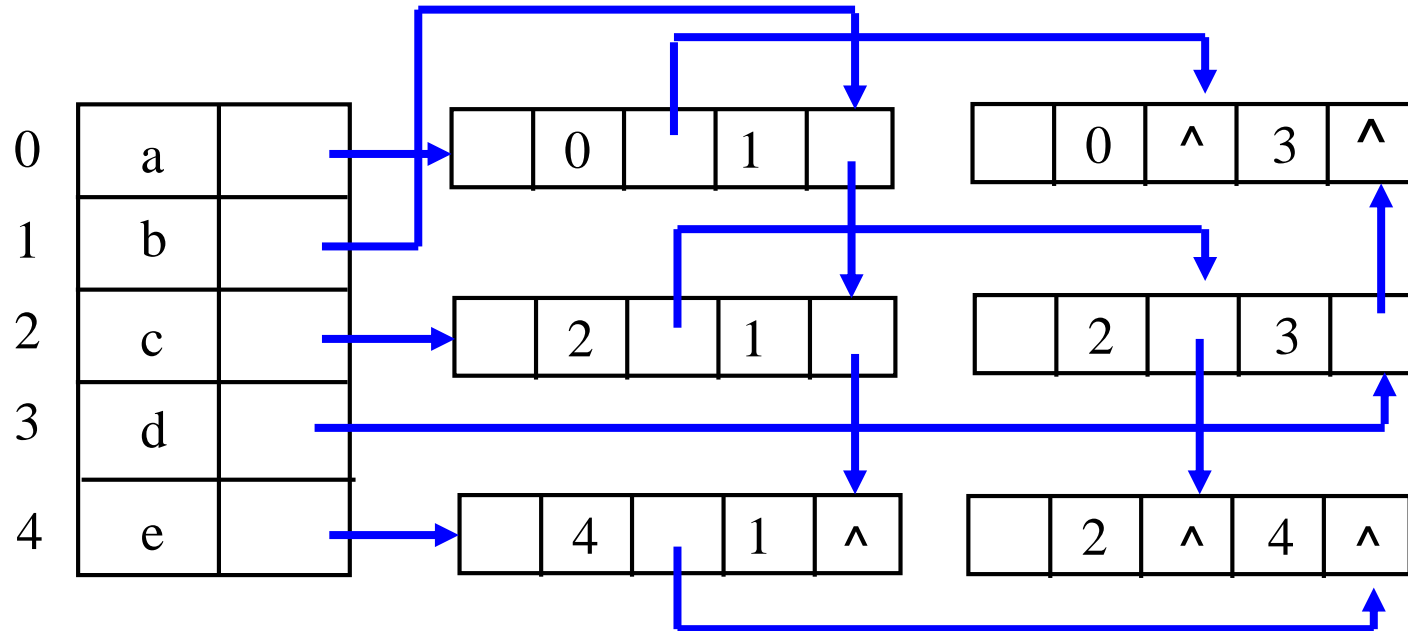
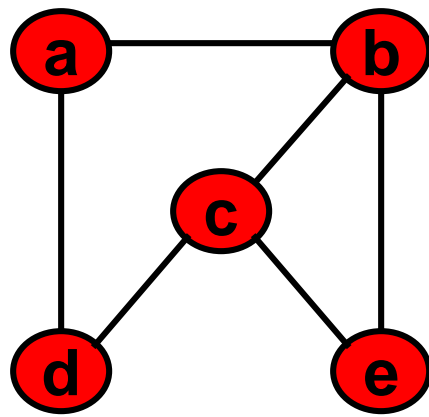
info: 为指向和边相关的各种信息的指针域。

每一个顶点也用一个结点表示：

data	firstedge
------	-----------

data域:存储和该顶点相关的信息；

firstedge域:指示第一条依附于该顶点的边。



在不同的存储结构下，实现各种操作的效率可能是不同的。所以在求解实际问题时，要根据求解问题所需操作，选择合适的存储结构。

7.3 图的遍历

图的遍历：从图的某顶点出发，访问图中所有顶点，并且每个顶点仅访问一次。

有两种遍历路径（对无向图，有向图都适用）：

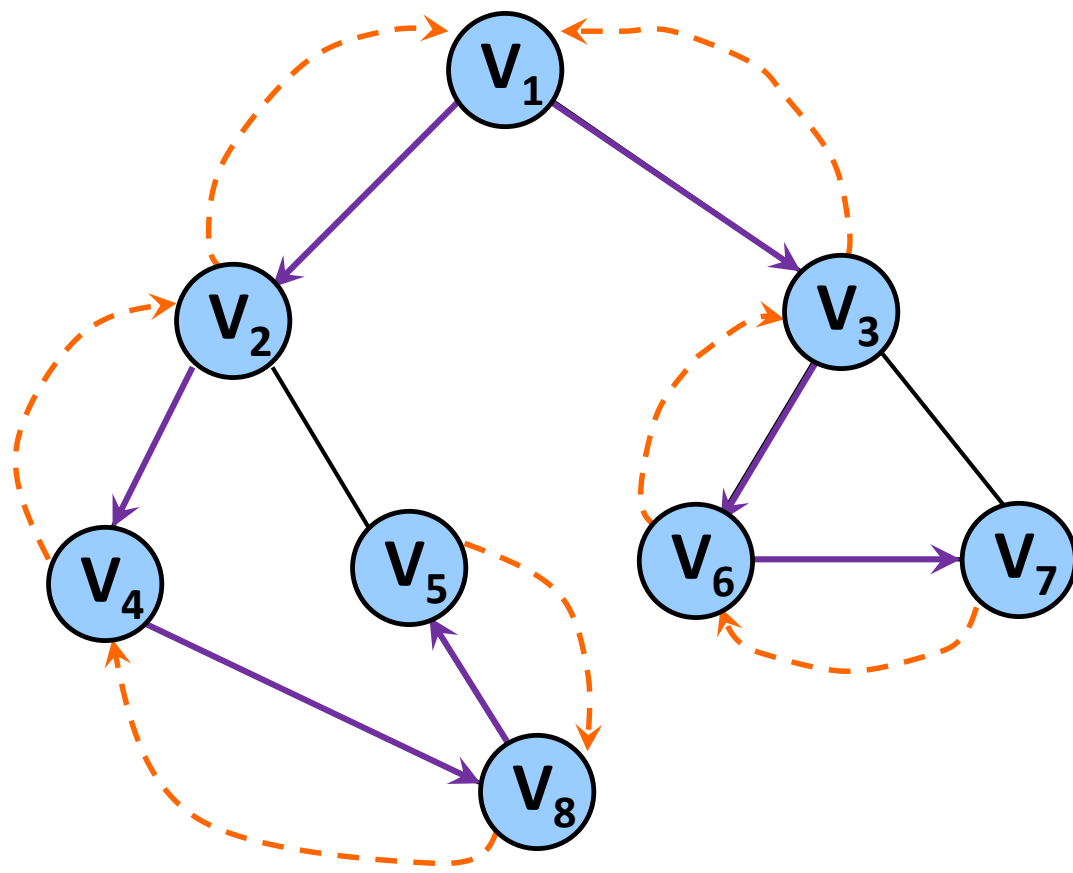
深度优先搜索

广度优先搜索

一. 深度优先搜索

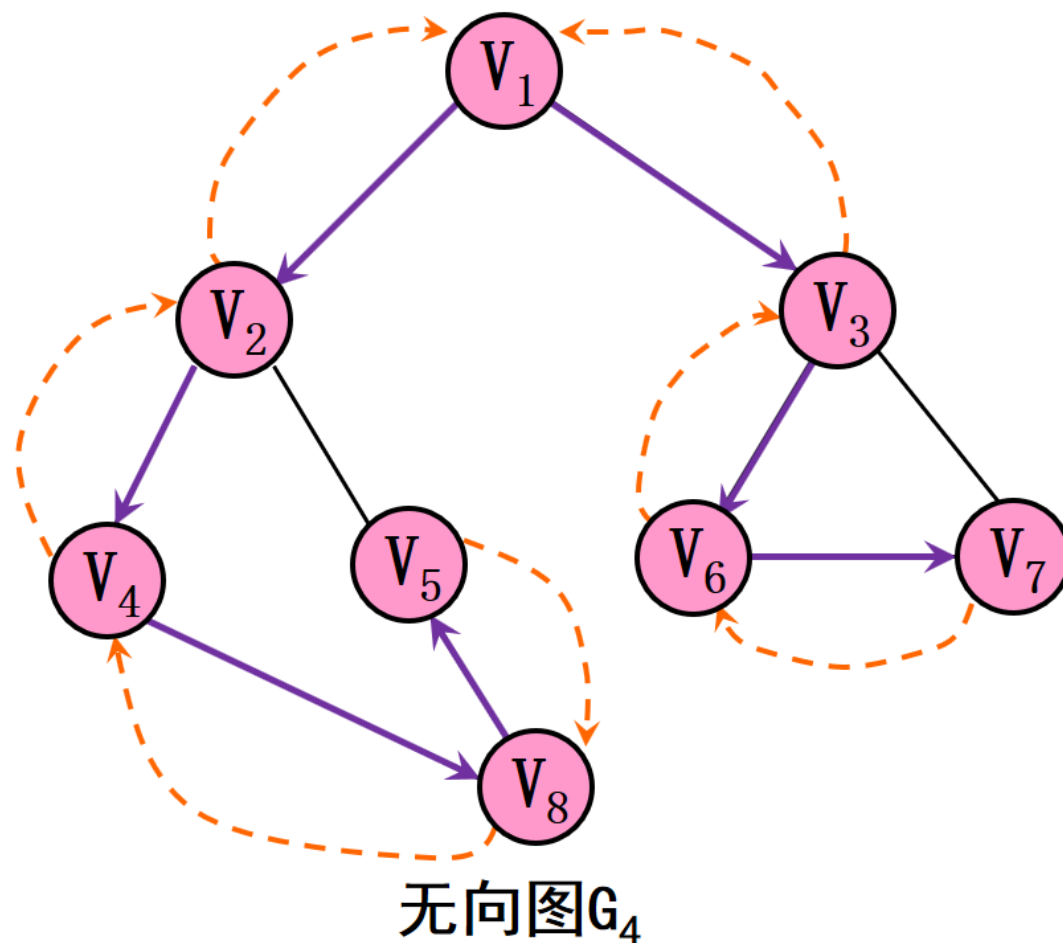
1. 深度优先搜索遍历的过程

- (1) 从图中某顶点 V 出发，访问顶点 V ；
- (2) **依次从 V 的未被访问的邻接点出发，继续对图进行深度优先遍历**，直至图中所有和 V 有路径相通的顶点都被访问到；
- (3) 若此时**图中尚有顶点未被访问**，则另选图中一个未曾被访问的顶点作起始点；
- (4) 重复上述过程，直至图中所有顶点都被访问到为止。



无向图 G_4

顶点访问序列: V_1 V_2 V_4 V_8 V_5 V_3 V_6 V_7



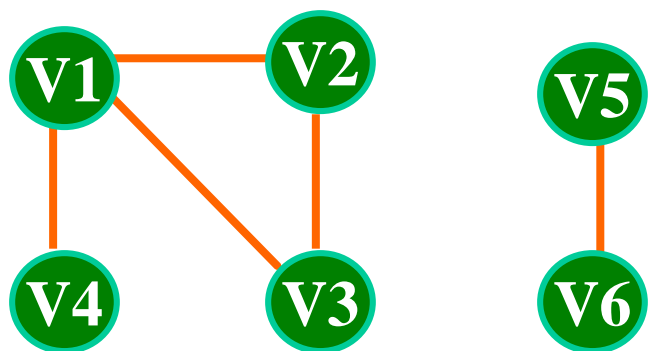
注意：

1. 结点的邻接点的次序是任意的，因此深度优先搜索的序列**可能有多种**。
2. 深度优先搜索类似于树的**先根遍历**

一. 深度优先搜索

1. 深度优先搜索遍历的过程

- (1) 从图中某顶点 V 出发，访问顶点 V ；
- (2) **依次从 V 的未被访问的邻接点出发，继续对图进行深度优先遍历**，直至图中所有和 V 有路径相通的顶点都被访问到；
- (3) 若此时**图中尚有顶点未被访问**，则另选图中一个未曾被访问的顶点作起始点；
- (4) 重复上述过程，直至图中所有顶点都被访问到为止。



非连通图

无向图G的深度优先序列:

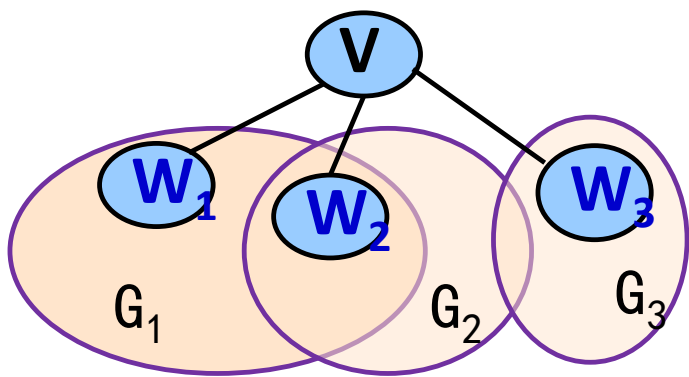
V1, V2, V3, V4, V5, V6

上面讨论的无向图的深度优先搜索遍历算法，若无向图是连通图，则从某一初始点开始能够访问到图中的所有顶点，若无向图是非连通图，需要从其它每个连通分量中选定初始点，分别进行搜索遍历，才能够访问到图中的所有顶点。

2. 深度优先搜索遍历的算法实现：

显然，深度优先搜索遍历图是一个递归的过程，为了在遍历过程中便于区分顶点是否已被访问，需附设访问数组 *visited* [*n*]，数组元素的初始值为 *false*；一旦某个顶点 *i* 被访问，则置 *visited* [*i*] 为 *true*。

<i>visited</i>	0	0
	1	0
	2	0
	3	0
	4	0
	m-1	



W_1 、 W_2 和 W_3 均为 V 的邻接点， G_1 、 G_2 和 G_3 分别为含顶点 W_1 、 W_2 和 W_3 的子图。

访问顶点 V ；

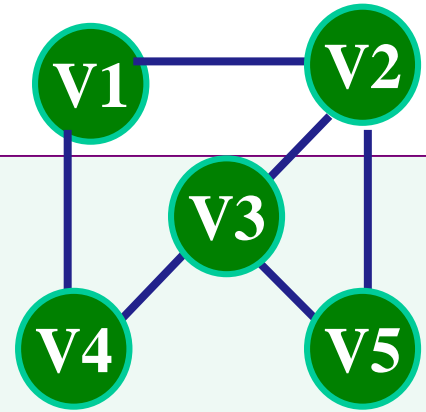
for(W_1 、 W_2 、 W_3)

若该邻接点 W_i 未被访问，

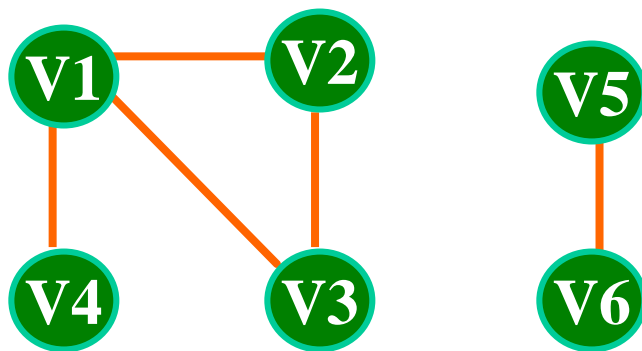
则从它出发进行深度优先搜索遍历。

深度优先搜索遍历连通图的算法描述如下：

```
void DFS(Graph G, int v)
{ //从第v个顶点出发，递归地深度优先遍历图G。
  //v是顶点在一维数组中的位置
  printf(v); //访问顶点v
  visited[v]=true;
  for(w=FirstAdjVex(G, v); w>=0; w=NextAdjVex(G, v, w))
    //依次检查V的所有邻接点w， FirstAdjVex(G, v)表示v的第一个邻接点
    // NextAdjVex(G, v, w)表示v的相对于w的下一个邻接点， w≥0表示存在邻接点
    if (!visited[w]) DFS(G, w); //对v的尚未访问的邻接点w递归调用DFS
}
```

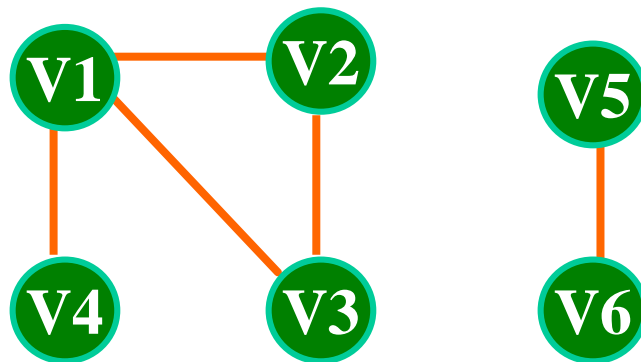


若是非连通图，上述遍历过程执行之后，图中一定还有顶点未被访问，需要从图中另选一个未被访问的顶点作为起始点，重复上述深度优先搜索过程。直到图中所有顶点均被访问过为止。



深度优先搜索遍历图的算法描述如下：

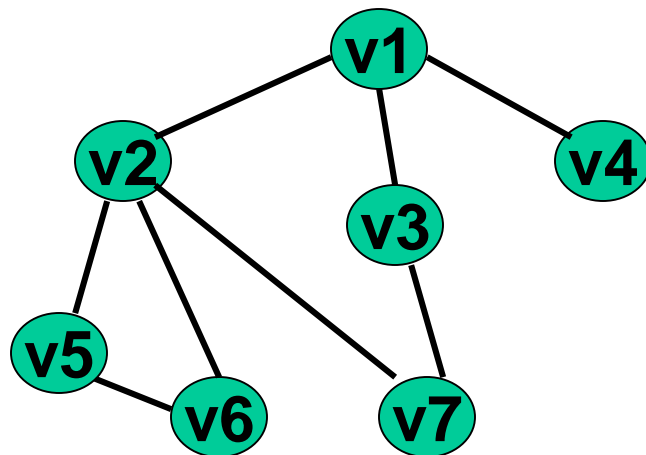
```
void DFSTraverse(Graph G)
{ // 对图G作深度优先遍历
    for (v=0; v<G.vexnum; ++v)
        visited[v] = false; // 访问标志数组初始化
    for (v=0; v<G.vexnum; ++v)
        if (!visited[v])    DFS(G, v); // 对尚未访问的顶点调用DFS
}
```



如何实现连通图G的深度优先遍历(从顶点v出发)的非递归过程?

解：本题的**算法思想**是：

第一步，首先访问图G的指定起始顶点v；



第二步，从v出发，访问一个与v邻接的顶点p后，再从顶点p出发，访问与p邻接未被访问的顶点q，然后从q出发，重复上述过程，直到找不到存在未访问过的邻接顶点为止；

第三步，回退到尚有未被访问过的邻接点的顶点，从该顶点出发，重复第二、三步，直到所有被访问过的顶点的邻接点都已被访问为止。

用一个**栈stack**保存被访问过的结点，以便回溯查找被访问过结点的未被访问过的邻接点。

深度优先搜索遍历算法的性能分析：

设图的顶点数为 n ，边数为 e 。

空间复杂度为 $O(n)$ 。

时间复杂度：

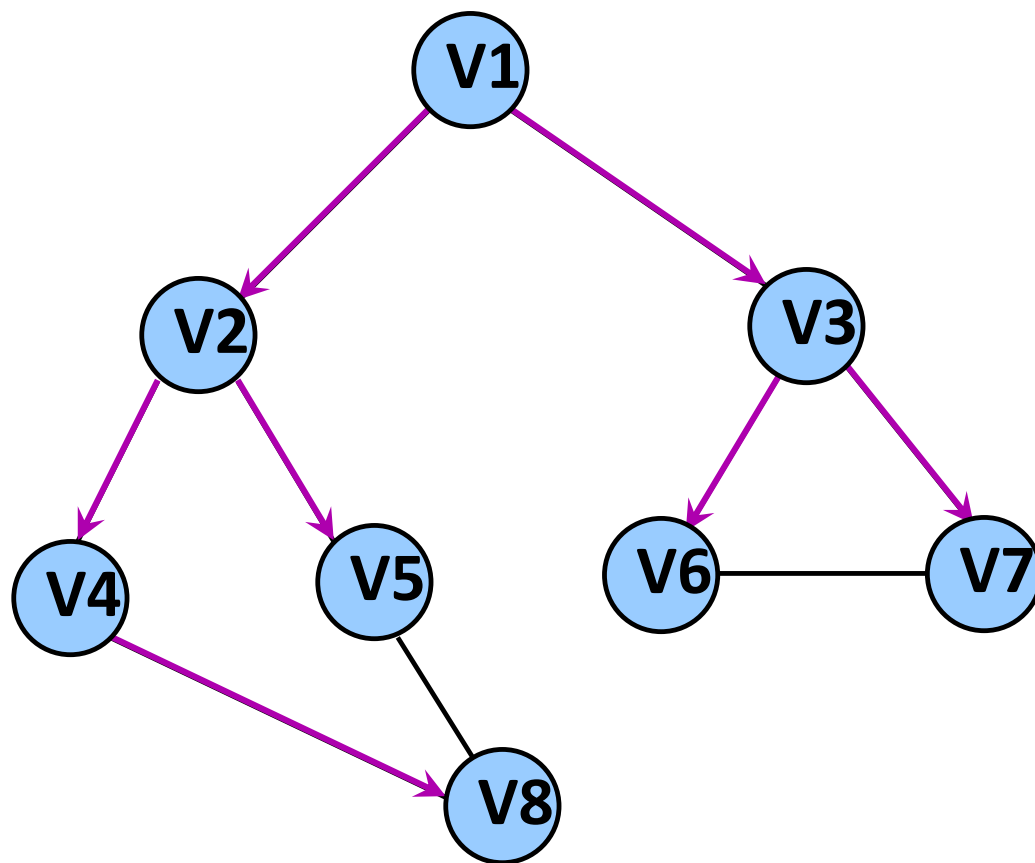
采用邻接矩阵表示时，查找每个顶点的邻接点所需时间为 $O(n)$ ，故总的时间复杂度为 $O(n^2)$ 。

采用邻接表表示时，查找所有顶点的邻接点所需时间为 $O(e)$ ，访问顶点所需时间为 $O(n)$ ，因此，总的时间复杂度为 $O(n+e)$ 。

二. 广度优先搜索

1. 广度优先搜索遍历的过程

- (1) 从图中某顶点 v 出发, 访问顶点 v ;
- (2) **依次**访问 v 的所有未被访问过的邻接点 W_1 、 W_2 , ..., W_i ;
- (3) 分别从这些邻接点出发, **依次**访问它们的邻接点, 并使“先被访问的顶点的邻接点”先于“后被访问的顶点的邻接点”被访问, 重复步骤3, 直至图中所有已被访问的顶点的邻接点都被访问到。
- (4) **若此时图中尚有顶点未被访问**, 则另选图中一个未曾被访问的顶点作起始点,
- (5) 重复上述过程, 直至图中所有顶点都被访问到为止。



无向图G4

顶点访问序列:



2. 广度优先搜索遍历的算法实现：

可以看出, 广度优先搜索遍历的特点是: **先访问的顶点其邻接点亦先被访问**。为此, 算法实现时需设置队列保存已被访问过的顶点。

广度优先搜索遍历连通图的算法描述如下：

```
void BFS (Graph G, int v)
{ // 按广度优先非递归遍历连通图G，使用辅助队列Q。
    visited[v]=True; printf(v);
    InitQueue(Q);           // 置空的辅助队列Q
    EnQueue(Q, v);          // v入队列
    while (!QueueEmpty(Q))
    {   DeQueue(Q, u);       // 队头元素出队并置为u
        for ( w=FirstAdjVex(G, u); w!=0; w=NextAdjVex(G, u, w) )
            if ( ! visited[w]) //w为u的尚未访问的邻接顶点
                { visited[w] = true; printf(w); // 访问w
                  EnQueue(Q, w);   // w入队列Q
                }
    }
}
```


广度优先搜索遍历图的算法描述如下：

```
void BFS (Graph G, int v)
{ // 按广度优先非递归遍历图G，使用辅助队列Q。
    for(v=0; v<G.vexnum; ++v) visited[v]=false;
    InitQueue(Q);                // 置空的辅助队列Q
    for(v=0; v<G.vexnum; ++v)
        if(!visited[v]){
            visited[v]=True; printf(v);
            EnQueue(Q, v);        // v入队列
            while (!QueueEmpty(Q)){
                DeQueue(Q, u);    // 队头元素出队并置为u
                for ( w=FirstAdjVex(G, u); w>=0; w=NextAdjVex(G, u, w) )
                    if ( ! visited[w] ) //w为u的尚未访问的邻接顶点
                        { visited[w] = true; printf(w); // 访问w
                          EnQueue(Q, w);    // w入队列Q
                        }
            }
        }
}
```

课堂习题：

已知无向图G的邻接矩阵如下，顶点依次为a、b、c、d、e。

0	1	0	0	1
1	0	0	1	0
0	0	0	1	1
0	1	1	0	1
1	0	1	1	0

(1) 画出该图。

(2) 根据邻接矩阵，从顶点a出发进行深度优先遍历和广度优先遍历，写出相应的遍历序列。

课堂习题：

已知无向图G的邻接矩阵如下，顶点依次为a、b、c、d、e。

0	1	0	0	1
1	0	0	1	0
0	0	0	1	1
0	1	1	0	1
1	0	1	1	0

(1) 画出该图。

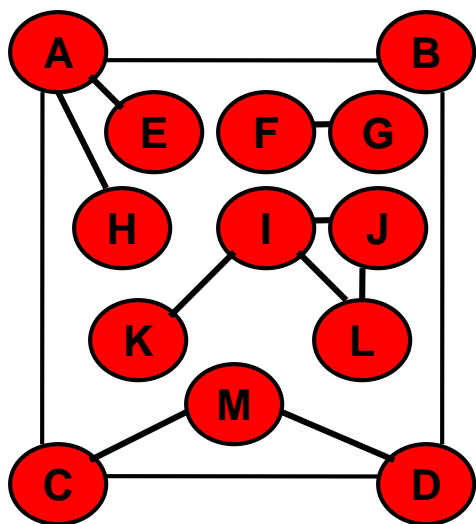
(2) 根据邻接矩阵，从顶点a出发进行深度优先遍历和广度优先遍历，写出相应的遍历序列。

7.4 最小生成树

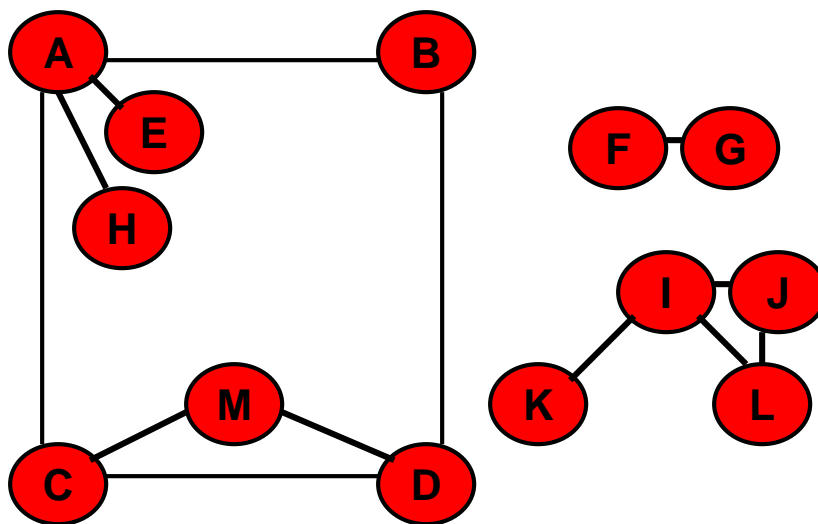
1、无向图的连通分量和生成树

- **连通**：顶点 v 至 v' 之间有路径存在
- **连通图**：无向图 G 的任意两点之间都是连通的，则称 G 是连通图。
- **连通分量**：极大连通子图

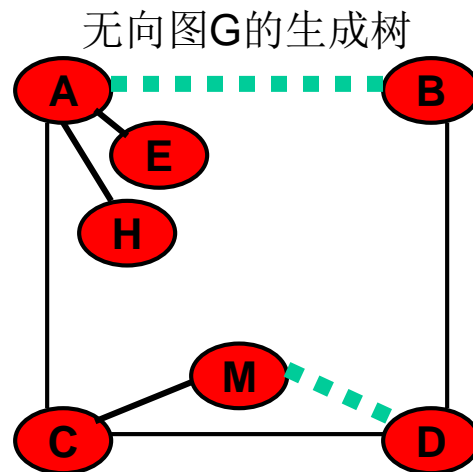
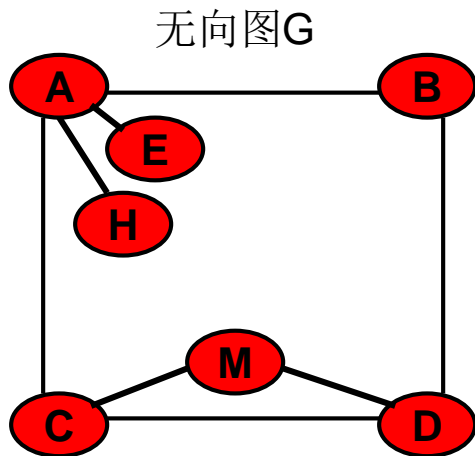
无向图 G



无向图 G 的三个连通分量



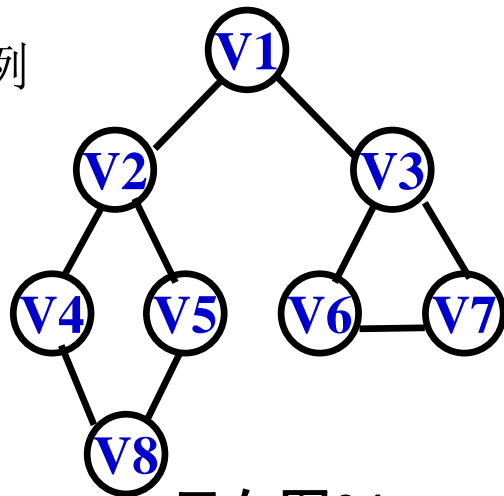
•**生成树**：极小连通子图。包含图的所有 n 个结点，但只含图的 $n-1$ 条边。



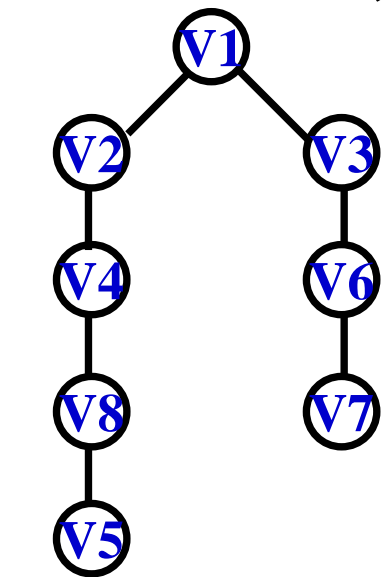
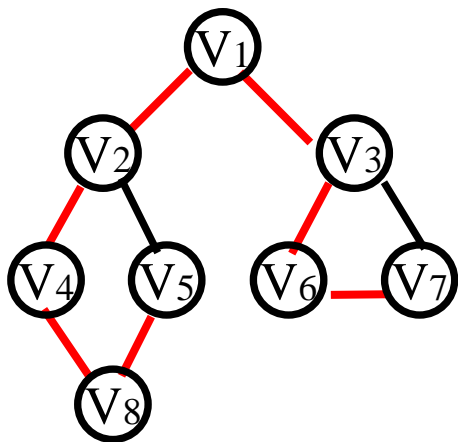
如何求得连通图的生成树？

设图 $G = (V, E)$ 是一个具有 n 个顶点的连通图，则从 G 的任一顶点出发，作一次深度优先搜索或广度优先搜索，所访问的顶点和所经过的边形成一棵生成树。

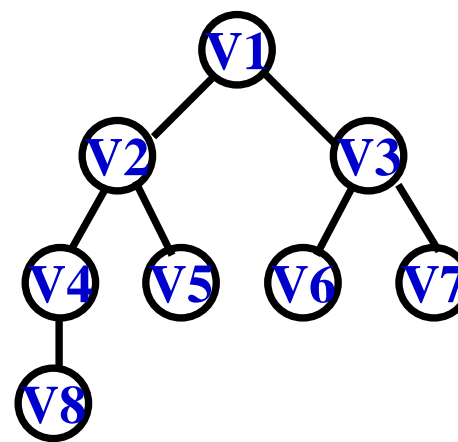
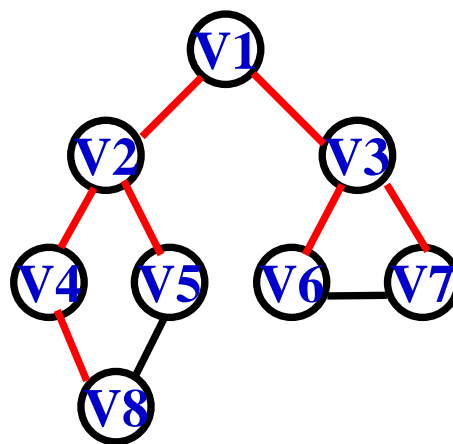
例



无向图G4



深度优先生成树



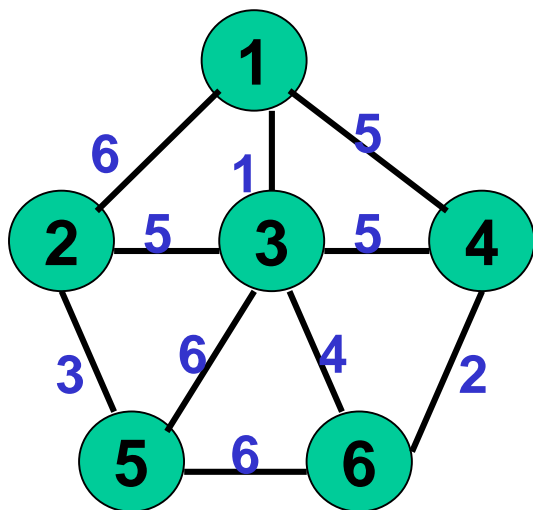
广度优先生成树

深度遍历: $V1 \Rightarrow V2 \Rightarrow V4 \Rightarrow V8 \Rightarrow V5 \Rightarrow V3 \Rightarrow V6 \Rightarrow V7$

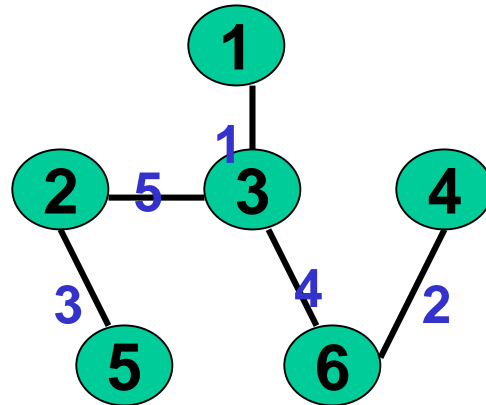
广度遍历: $V1 \Rightarrow V2 \Rightarrow V3 \Rightarrow V4 \Rightarrow V5 \Rightarrow V6 \Rightarrow V7 \Rightarrow V8$

2. 最小生成树

对于一个连通网来说，不同的生成树，其每棵树的所有边上的权值之和也可能不同。具有边上的权值之和最小的生成树称为**图的最小代价生成树** (Minimum Cost Spanning Tree)，简称为最小生成树。



a. 连通网G5



b. G5的最小生成树

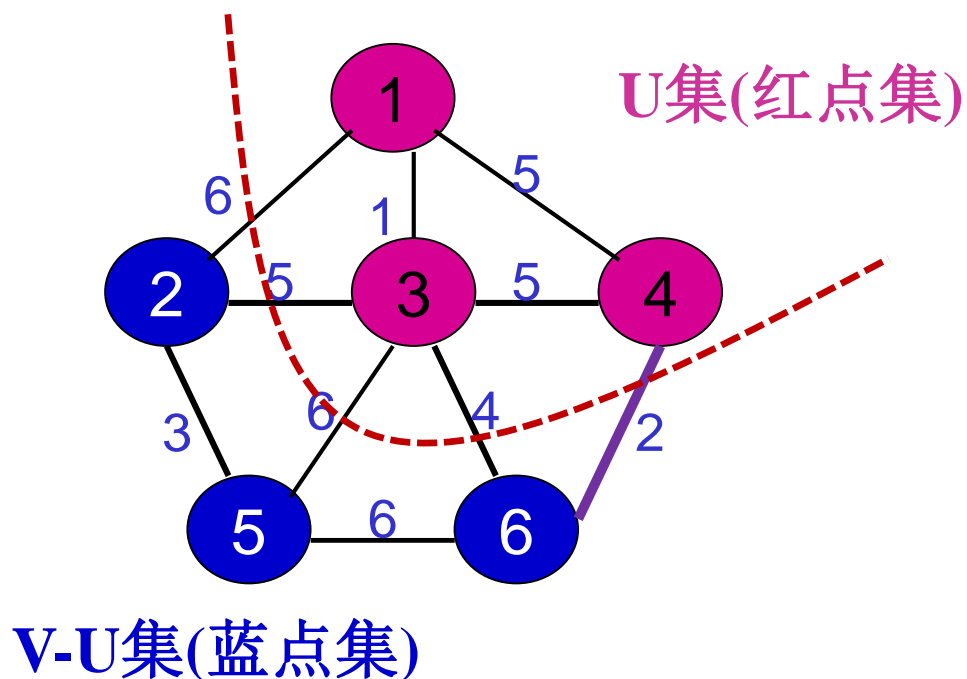
求图的最小生成树很有实际意义

例如:若一个连通网表示城市之间的通讯系统,网的顶点代表城市,网的边代表城市之间架设通讯线路的造价,各城市之间的距离不同,地理条件不同,其造价也不同,即边上的权不同,现在要求既要连通所有城市、又要使总造价最低,这就是一个求图的最小生成树的问题。

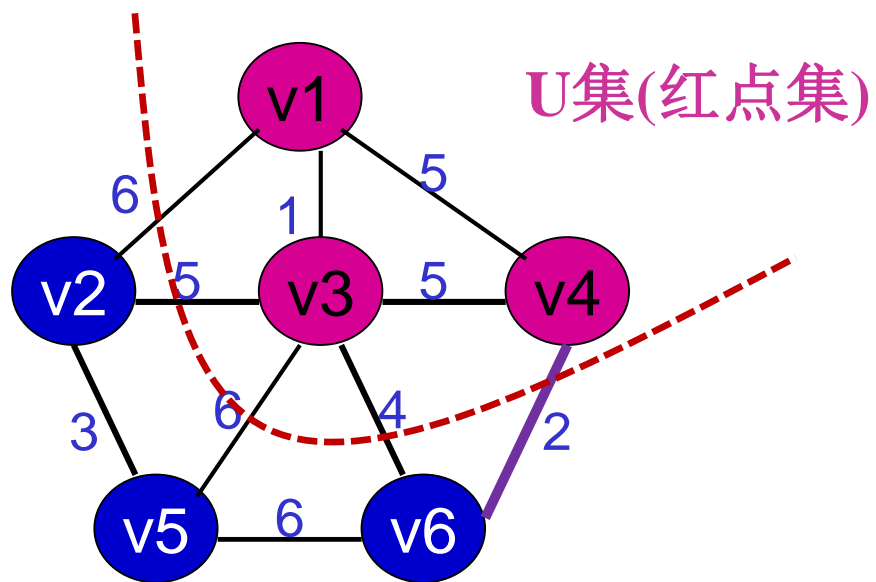
该问题等价于:构造具有 n 个顶点的网的一棵最小生成树,即:在 $n(n-1)/2$ 条带权的边中选取 $n-1$ 条(不构成回路),使“权值之和”为最小。

最小生成树的MST性质：

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



必存在一棵包含边 $(4, 6)$ 的最小生成树



U集(红点集)

V-U集(蓝点集)

求图的最小生成树的两个算法：

A. 普里姆 (Prim) 算法

B. 克鲁斯卡尔 (Kruskal) 算法。

一. 普里姆算法

普里姆算法的构造过程如下：

设 $N=(V, E)$ 是连通网， $T=(U, TE)$ 为 N 的最小生成树，其中 U 是 T 的顶点集， TE 是 T 的边集。

$U = \{ u_0 \}; TE = \{ \}; \quad // u_0 \in V$

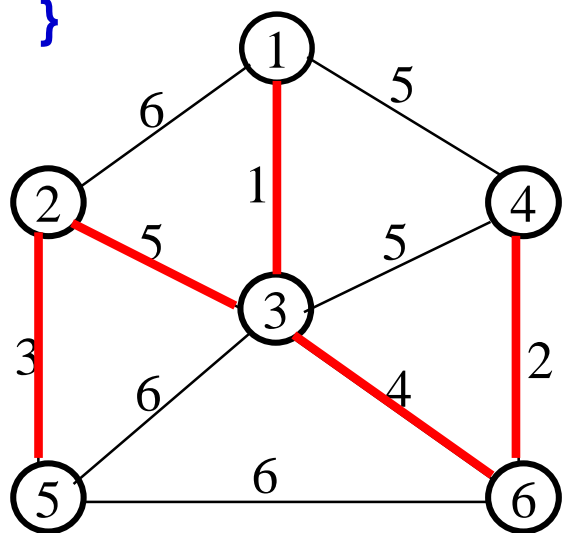
while ($U \neq V$)

{ 设 (u_0, v_0) 是一条代价最小的边，并且 $u_0 \in U, v_0 \in V - U$ 。

$TE = TE \cup (u_0, v_0);$

$U = U \cup \{ v_0 \};$

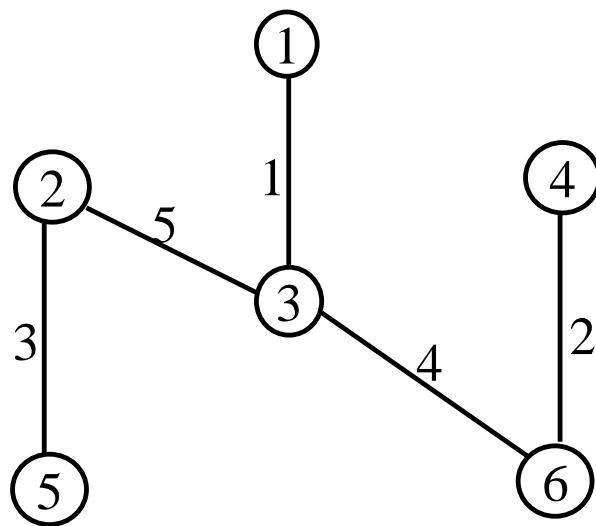
}



连通网N

普里姆算法的关键之处是：

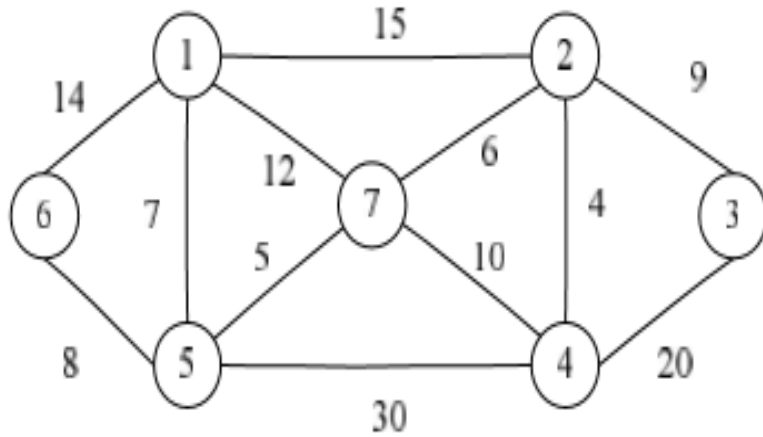
每次如何从生成树T中到T外的所有边中，找出一条代价最小的边。



最小生成树T

课堂练习：

已知如下图所示的无向带权图。 用普里姆（Prim）算法求其从顶点1开始的最小生成树（要求画出构造过程）。



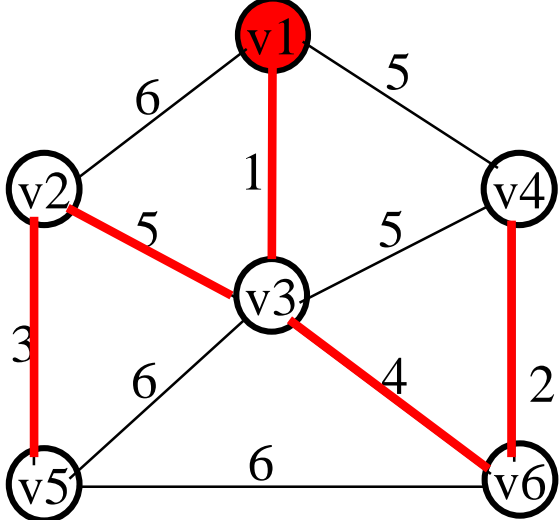
普里姆算法的实现：

为便于算法实现，设置了一个如下的辅助数组`closedge`，用来保存从顶点集 U 到 $V-U$ 的代价最小的边。对每个顶点 $v_i \in V-U$ ，在数组中存在一个相应分量`closedge[i-1]`，它包括两个域：`lowcost`和`adjvex`。其中，`lowcost`存储最小边上的权值，`adjvex`存储最小边在 U 中的那个顶点。

辅助数组的定义：

```
struct {  
    VertexType  adjvex; //边所依附于U中的顶点  
    VRType  lowcost;    //该边的权值  
} closedge[MaxN];
```

```
closedge[j].adjvex=u;  
closedge[j].lowcost =10;
```



G5

		对应顶点 V_2 V_3 V_4 V_5 V_6							
i closededge		1	2	3	4	5	U	V-U	K
adjvex	V_1	V_1	V_1				$\{v_1\}$	$\{v_2, v_3, v_4, v_5, v_6\}$	2
lowcost	6	1	5						
adjvex	v_3		v_1	v_3	v_3		$\{v_1, v_3\}$	$\{v_2, v_4, v_5, v_6\}$	5
lowcost	5	0	5	6	4				
adjvex	v_3		v_6	v_3			$\{v_1, v_3, v_6\}$	$\{v_2, v_4, v_5\}$	3
lowcost	5	0	2	6	0				
adjvex	v_3			v_3			$\{v_1, v_3, v_6, v_4\}$	$\{v_2, v_5\}$	1
lowcost	5	0	0	6	0				
adjvex				v_2			$\{v_1, v_3, v_6, v_4, v_2\}$	$\{v_5\}$	4
lowcost	0	0	0	3	0				
adjvex							$\{v_1, v_3, v_6, v_4, v_2, v_5\}$	$\{\}$	
lowcost	0	0	0	0	0				

普里姆算法描述：

```
void MiniSpanTree_PRIM (MGraph G, VertexType u)
{ //用Prim算法从顶点u 出发构造网G的最小生成树T,
  //输出T的各条边。
  k = LocateVex (G, u);           //k为顶点 u的下标
  for (j = 0; j < G.vexnum; ++ j) //辅助数组初始化
    if (j != k) closedge[j] = {u, G.arcs[k][j] };
                                     //{adjvex, lowcost}
  closedge[k].lowcost = 0;          //初始, U = {u}
```



```

for (i = 0; i < G.vexnum-1; ++ i)
{ //选择其余G.vexnum-1个顶点
    k = minimum (closedge);    //求出T的下一个结点： 顶点K
    //此时closedge[k].lowcost =
    //    MIN{ closedge[vi].lowcost | closedge[vi].lowcost > 0, vi ∈ V-U}
    printf (closedge[k].adjvex, G.vexs[k]); //输出当前的最小边
    closedge[k].lowcost = 0;                //第k顶点并入U集
    for (j = 0; j < G.vexnum; ++ j)
        if (G.arcs[k][j] < closedge[j].lowcost)
            //新顶点并入U后重新选择最小边
            closedge[j] = {G.vexs[k], G.arcs[k][j] };
    }
}

```

二. 克鲁斯卡尔算法

为使生成树上边的权值之和最小，显然，其中每一条边的权值应该尽可能地小。克鲁斯卡尔算法的做法就是：先构造一个只含 n 个顶点的子图 ST ，然后从权值最小的边开始，若它的添加不使 ST 中产生回路，则在 ST 上加上这条边，如此重复，直至加上 $n-1$ 条边为止。

克鲁斯卡尔算法的基本描述：

```
构造非连通图  $ST=(V, \{ \} )$ ;
```

```
   $k = i = 0$ ;
```

```
  while ( $k < n-1$ )
```

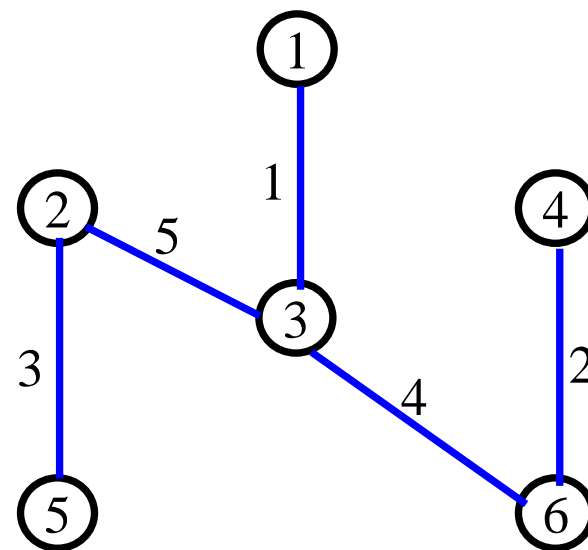
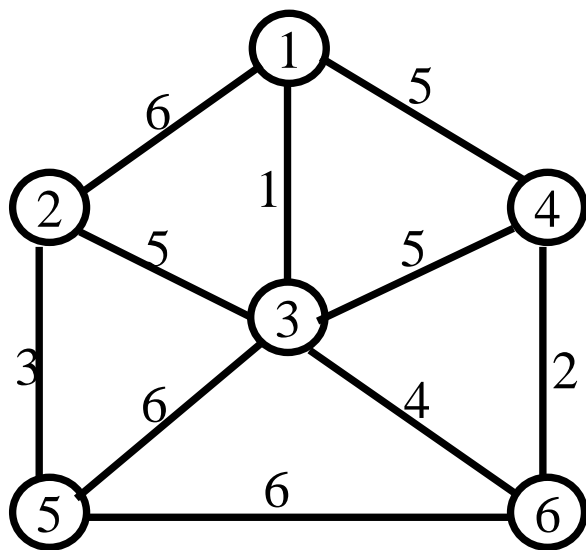
```
  {    $++i$ ;
```

```
    从边集  $E$  中选取第 $i$ 条权值最小的边 $(u, v)$ ;
```

```
    若 $(u, v)$ 加入 $ST$ 后不使 $ST$ 中产生回路,
```

```
    则将 $(u, v)$ 加入到 $ST$ 中, 且  $k++$ ;
```

```
  }
```



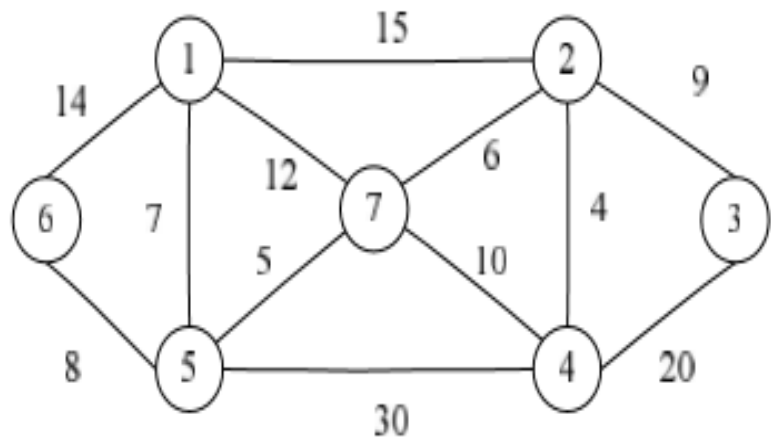
克鲁斯卡尔算法构造最小生成树的过程

算法性能：

由于普里姆算法的时间复杂度为 $O(n^2)$ ，则适于稠密图；而克鲁斯卡尔算法需对 e 条边按权值进行排序，其时间复杂度为 $O(e \log_2 e)$ ，则适于稀疏图。

课堂练习：

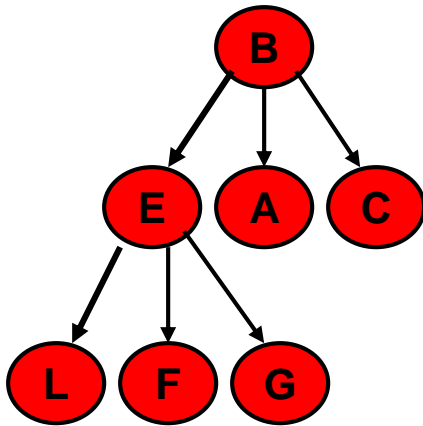
已知如下图所示的无向带权图。用克鲁斯卡尔算法求其从顶点1开始的最小生成树（要求画出构造过程）。



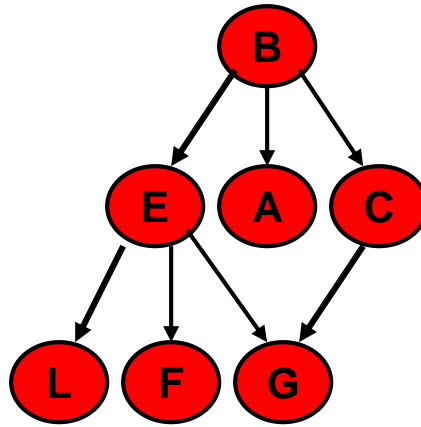
7.5 有向无环图及其应用

1、有向无环图（DAG 图）

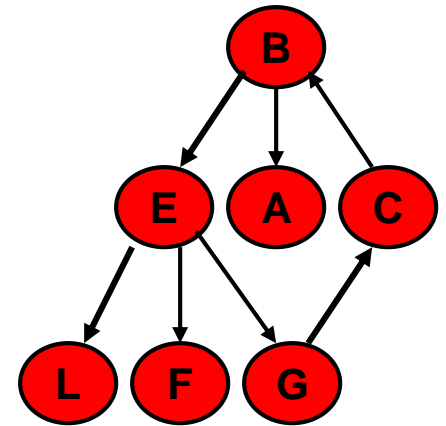
无环的有向图称为有向无环图。



有向树



有向无环图



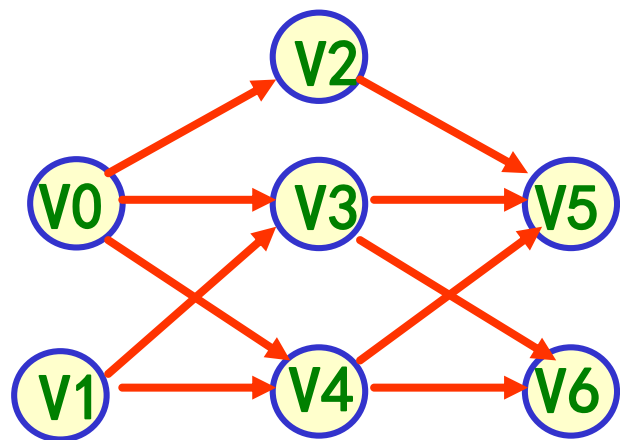
有向图（含环）

有向无环图用途：描述工程项目或系统进行过程的有效工具。许多应用问题可用有向无环图来表示和求解。

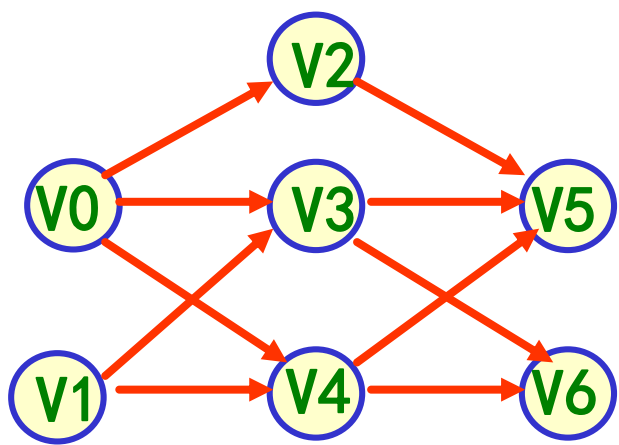
常用的两种有向图：AOV网，AOE网

AOV-网：

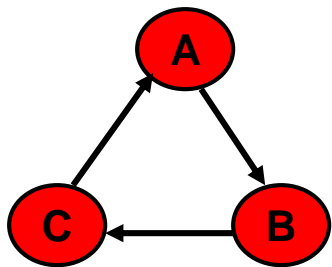
用**顶点表示活动**，用**弧表示活动间的优先关系**的有向图称为**顶点表示活动的网**(Activity On Vertex Network)，简称AOV-网。



例1 某工程可分为V0、V1、V2、V3、V4、V5、V6 7个子工程， 工程流程可用如左AOV网表示。其中
顶点：表示子工程（也称活动），
弧：表示子工程间的顺序关系。



A. 若 $\langle v_i, v_j \rangle$ 是图中有向边, 则
 v_i 是 v_j 的直接前驱;
 v_j 是 v_i 的直接后继.



有向图 (含环)

B. 在AOV-网中, 不应该出现有向环, 因为存在环意味着某项活动应以自己为先决条件。如左图所示若出现有向环, 则产生矛盾。

7.5.1 AOV网与拓扑排序

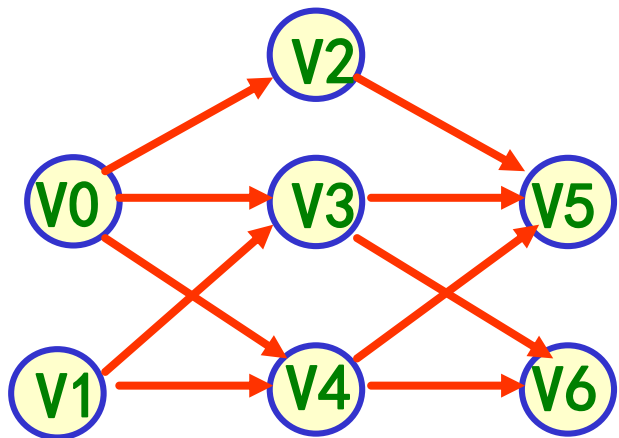
对工程问题，人们关心如下**两类问题**：

- 1) 工程能否顺序进行，即工程流程是否“合理”；
- 2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程？

拓扑排序
(AOV网应用)

求关键路径
(AOE网应用)

为求解工程流程是否“合理”，通常用AOV网来表示工程流程。



例1 某工程可分为V0、V1、V2、V3、V4、V5、V6 7个子工程，工程流程可用如左AOV网表示。其中

顶点：表示子工程（也称活动），

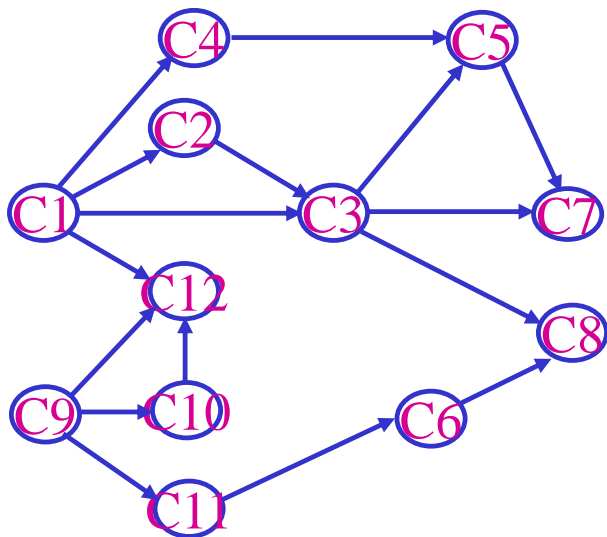
弧：表示子工程间的顺序关系。

例2 课程流程图

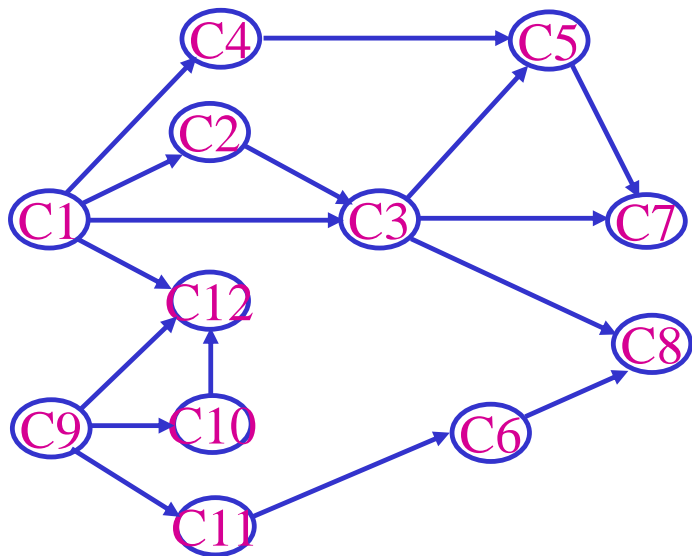
某校计算机专业课程流程可用AOV网表示。其中

顶点：表示课程（也称活动），

弧：表示课程间的优先关系, 若课程 v_i 是课程 v_j 的先修课程，则图中有弧 $\langle v_i, v_j \rangle$ ；



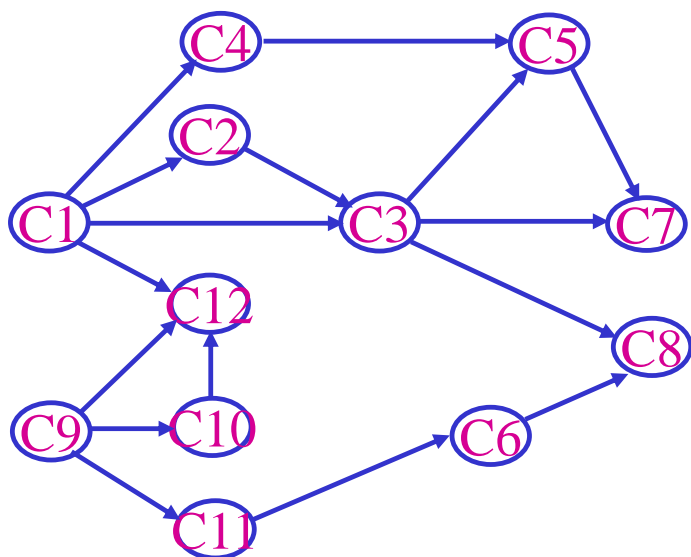
课程代号	课程名称	先修课
C1	程序设计基础	无
C2	离散数学	C1
C3	数据结构	C1,C2
C4	汇编语言	C1
C5	语言的设计和分析	C3,C4
C6	计算机原理	C11
C7	编译原理	C3.C5
C8	操作系统	C3,C6
C9	高等数学	无
C10	线性代数	C9
C11	普通物理	C9
C12	数值分析	C1,C9,C10



问题提出： 教学计划的制定——**学生选修课程安排问题**：

哪些课程是必须**先修**的，哪些课程是可以**并行**学习的。
学生应按怎样的顺序学习这些课程，才能有步骤、顺利地完
成学业？

——AOV网的**拓扑排序**



一个可行的教学计划为：

C1, C9, C4, C2, C10, C11, C12, C3, C6, C5, C7, C8

可行的计划的特点：若在图中顶点 v 是顶点 u 的前趋，则在计划序列中顶点 v 也是 u 的前趋。

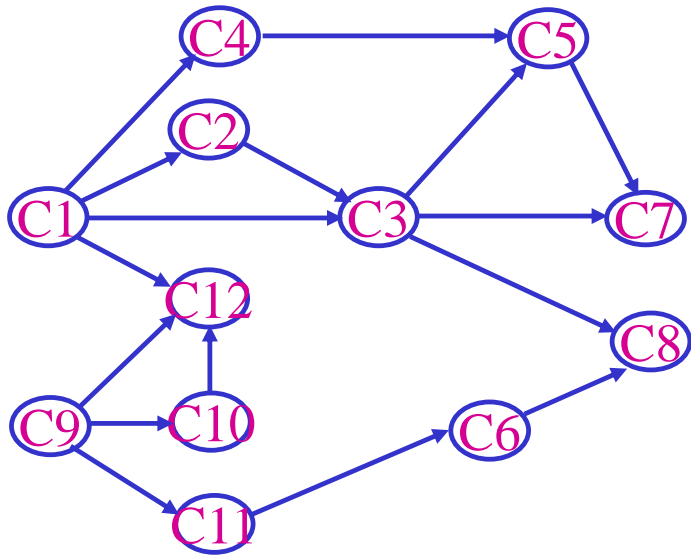
拓扑序列 (Topological order) :

有向图D的一个顶点序列称作一个拓扑序列，如果该序列中任两顶点 v 、 u ，若在D中 v 是 u 前趋，则在序列中 v 也是 u 前趋。

拓扑排序 (Topological Sort) :

就是将有序向图中顶点排成拓扑序列。

AOV网的拓扑序列不是唯一的，满足定义的任一线性序列都称作它的拓扑序列。

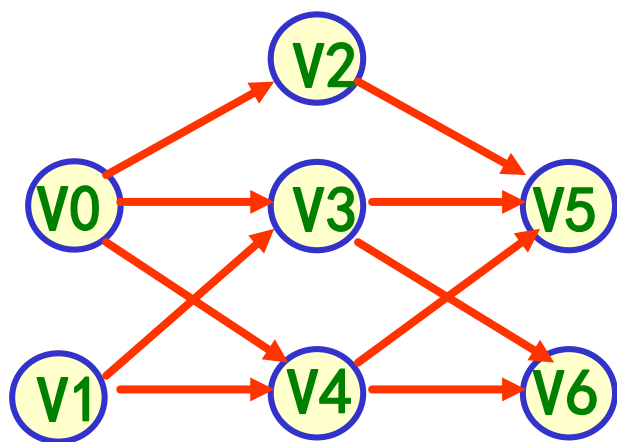


例如，下面的三个序列都是左图的拓扑序列，当然还可以写出许多。

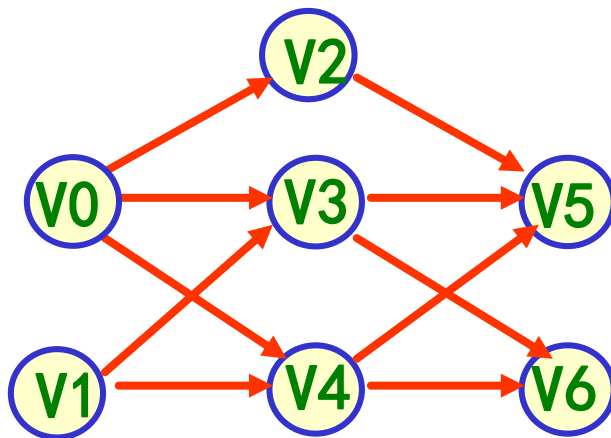
C1, C9, C4, C2, C10, C11, C12, C3, C6, C5, C7, C8
C1, C2, C3, C4, C5, C7, C9, C10, C11, C6, C12, C8
C9, C10, C11, C6, C1, C12, C4, C2, C3, C5, C7, C8

2 拓扑排序方法：

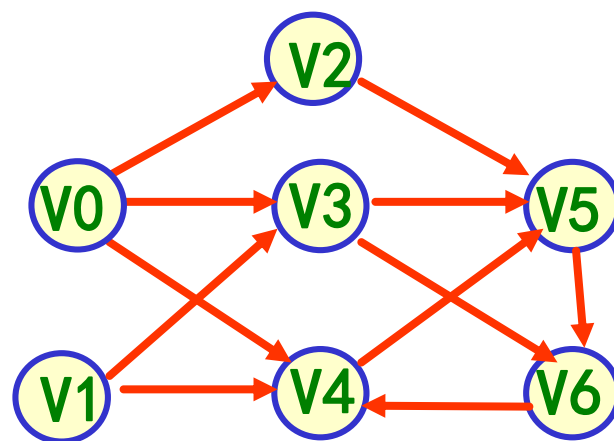
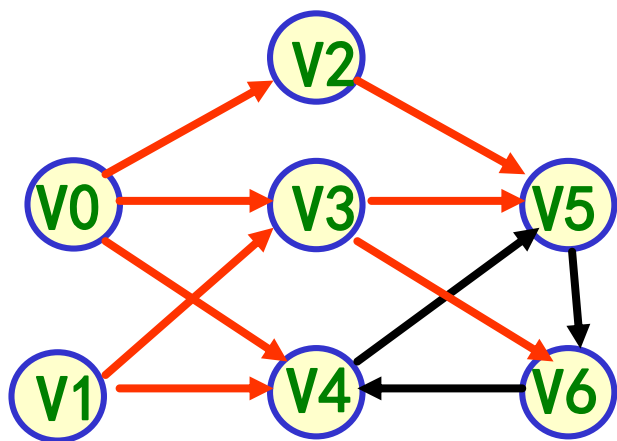
- 1) 在有向图选一无前趋的顶点 v ，输出；
- 2) 从有向图中删除 v 及以 v 为尾的孤；
- 3) 重复1、2、直接输出全部顶点或有向图中不存在无前趋顶点；



AOV网



拓扑序列： V0 V1 V2 V3 V4 V5 V6



拓扑序列: V0 V1 V2 V3

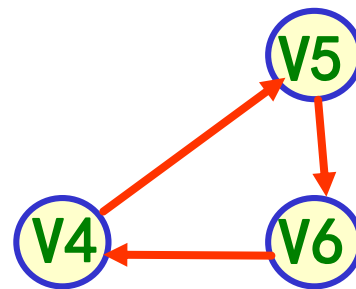
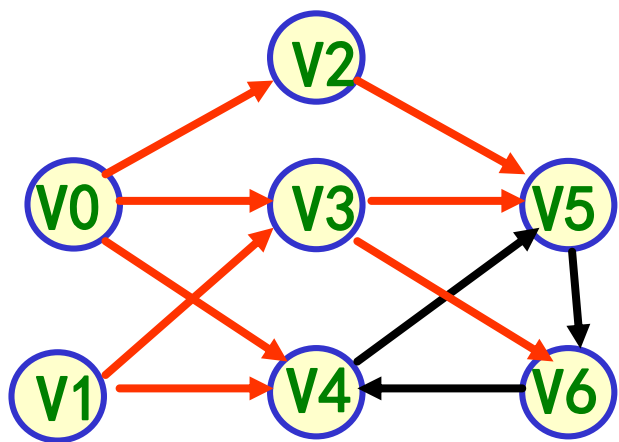
拓扑排序的应用

例 判断工程流程是否合理

工程流程是否合理？至少要看表示工程流程AOV网是否存在“死循环”，用图的术语就是AOV网是否存在回路。若存在回路，显然工程流程不合理。

如何判断AOV网是否存在回路？

对有向图构造其顶点的拓扑有序序列，若图中所有顶点都在它的拓扑有序序列中，则该AOV网必定不存在环。



拓扑序列: V0 V1 V2 V3

如何在计算机上实现对有向图的拓扑排序？

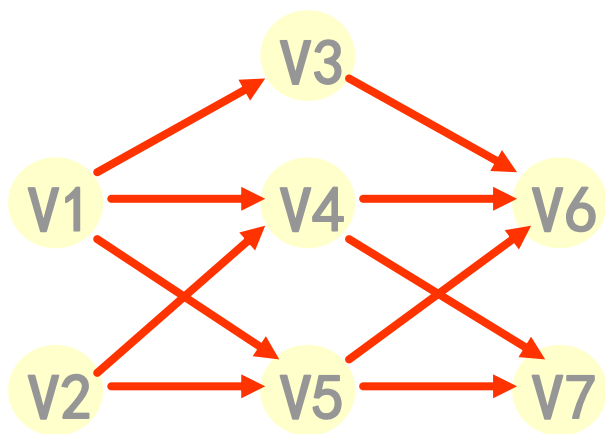
3 拓扑排序算法

为实现有向图的拓扑排序，我们给出拓扑排序方法的另一种描述：

- (1) 选择一入度为0(无前趋)的顶点 v ，输出；
- (2) 弧头顶点的入度减1(删除顶点及以它为尾的弧)；
- (3) 重复1、2 直到输出全部顶点或有向图没有入度为0的顶点；

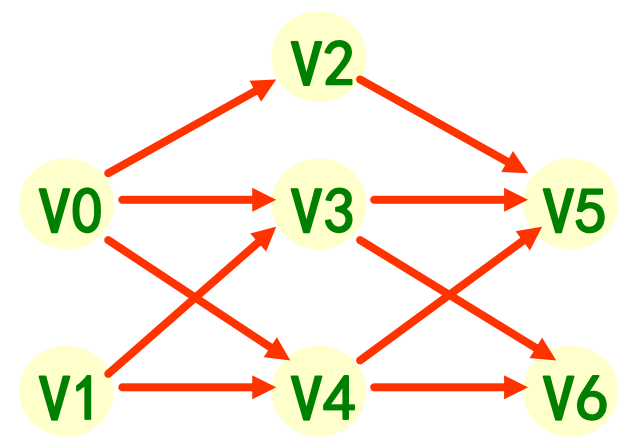
为算法的有关数据选择设计存储结构

(1)ALGraph G 邻接表：存储有向图



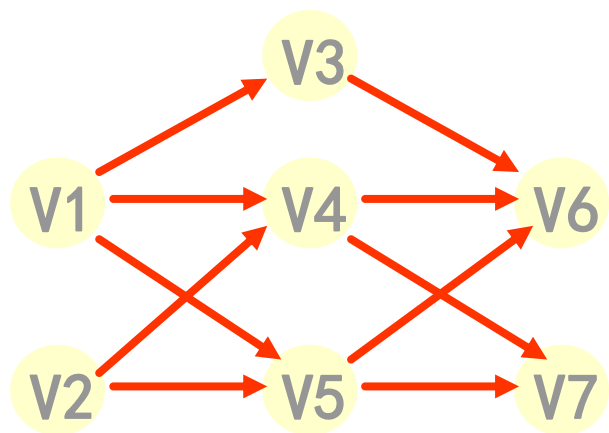
D. vertices	0	V1		→	2	→	3	→	4	^
	1	V2		→	3	→	4	^		
	2	V3		→	5	^				
	3	V4		→	5	→	6	^		
	4	V5		→	5	→	6	^		
	5	V6	^							
	6	V7	^							
D. vexnum		n								
D. arcnu		e								
D. kind		DG								

(2) indegree[] 一维数组：存储顶点的入度：



indegree	0	0
1	0	
2	1	
3	2	
4	2	
5	3	
6	2	

存储顶点的入度的
数组



D. vertices	0	V1	0	→	2	→	3	→	4	^
	1	V2	0	→	3	→	4	→	4	^
	2	V3	1	→	5	→	5	→	5	^
	3	V4	2	→	5	→	6	→	6	^
	4	V5	2	→	5	→	6	→	6	^
	5	V6	3							
	6	V7	2							
<div>初始时，只有v1, v2 两顶点的入度为0</div>										
D. vexnum			n							
D. arcnu			e							
D. kind			DG							

	6
	5
	4
	3
S. top →	2
	1
S. base →	0

(3) 栈S：用于存储入度为0的顶点的编号（也是顶点在邻接表顶点数组中的位置）

算法的思想：

- 1、用一个数组记录每个顶点的入度。将入度为零的顶点进栈。
- 2、将栈中入度为零的顶点 V 输出。
- 3、根据邻接表找到顶点 V 的所有的邻接顶点，并将这些邻接顶点的入度减一。如果某一顶点的入度变为零，则进栈。
- 4、反复执行 2、3；直至栈空为止。

.....

执行结束时，如果输出顶点数等于图的顶点总数，则有向图无环，否则有向图有环。

4) 拓扑排序算法

Status TopologicalSort(ALGraph G) {

//有向图G采用邻接表存储结构。

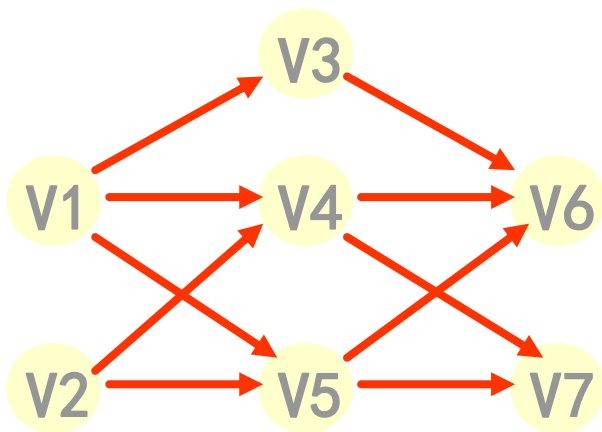
//若G无回路，则输出G的顶点的一个拓扑序列并返回OK，否则ERROR。

FindInDegree(G, indegree); //求各顶点入度indegree[0..vernum-1]

InitStack(S); //建入度为0的顶点栈S

For(i=0; i<G.vexnum; ++i)

if (!Indegree[i]) Push(S, i); //入度为0顶点的编号进栈



indegree 0	0
1	0
2	1
3	2
4	2
5	3
6	2


```
count=0;    //对输出顶点计数
while(! StackEmpty(S)){
    Pop(S, i);    //将栈顶顶点i出栈
    printf(i, G. vertices[i].date); ++count; //输出i号顶点的数据, 并计数
    for (p=G. vertices[i]. firstarc; p; p=p->nextarc) {
        k=p->adjvex;
        if (!(- indegree[k])) Push(S, k);
        //对顶点i邻接到的每个顶点的入度减1, 若入度变为0, 则入栈
    }
}
if (count<G.vexnum) return  ERROR; //该有向图有回路
else return OK;
}
```

7.5.2 AOE网与关键路径

对工程人们关心两类问题：

- 1) 工程能否顺序进行，即工程流程是否“合理”
- 2) 完成整项工程至少需要多少时间，哪些子工程是影响工程进度的关键子工程？

为解决第二类问题，通常可用称为AOE网的有向图表示工程流程

AOE网 (activity on edge net)

用弧表示**活动**，**顶点**表示**事件**的有向图称为AOE网。 **事件发生**表示以该事件为起点的活动可以开始，以该事件为终点的活动已经结束。

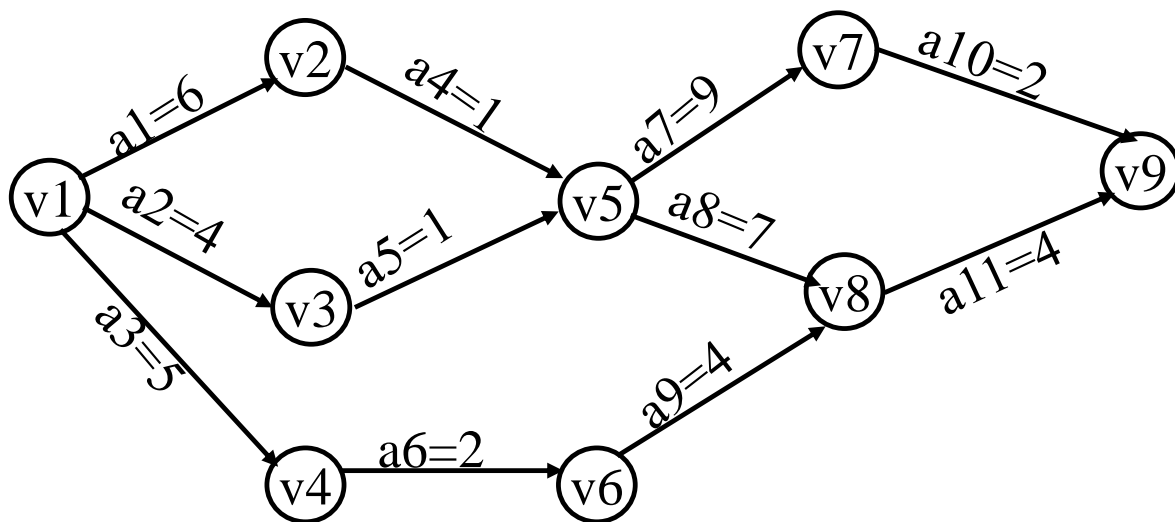
例 设一个工程有**11项活动**，**9个事件**

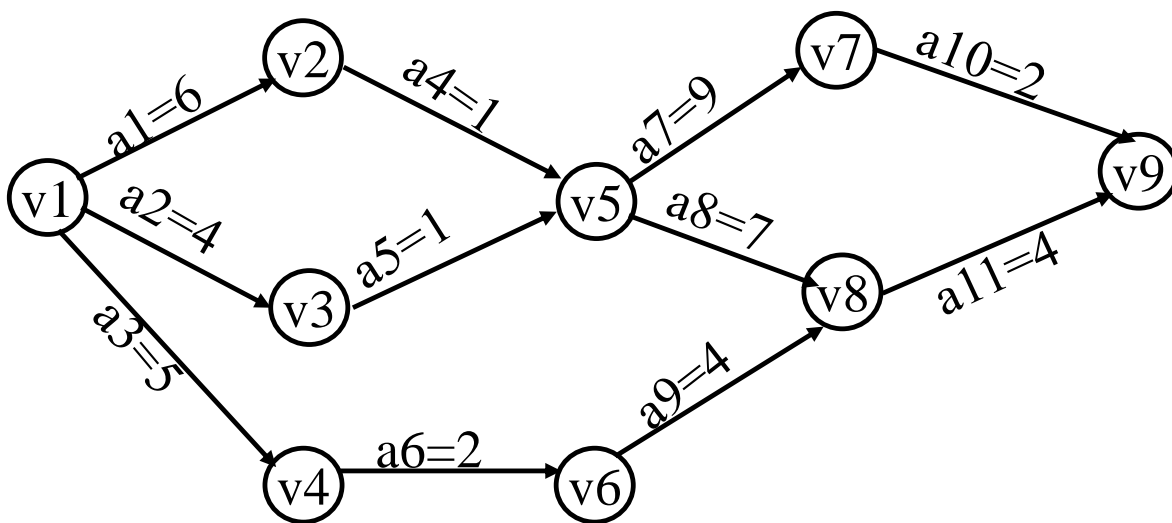
每个事件表示在它之前的活动已经完成，在它之后的活动可以开始。

事件v1 (**源点**)——表示整个工程开始

事件v9 (**汇点**)——表示整个工程结束

弧的权**值**——表示该活动**持续时间**





在AOE网中，有些活动可以并行地进行，所以完成工程的最短时间是从开始点到完成点的最长路径的长度（指路径上各**活动持续时间之和，而非弧数目**）。路径长度最长的路径**叫做关键路径**。

从V1到V9的最长路径是 (v1, v2, v5, v8, v9)，路径长度是18。

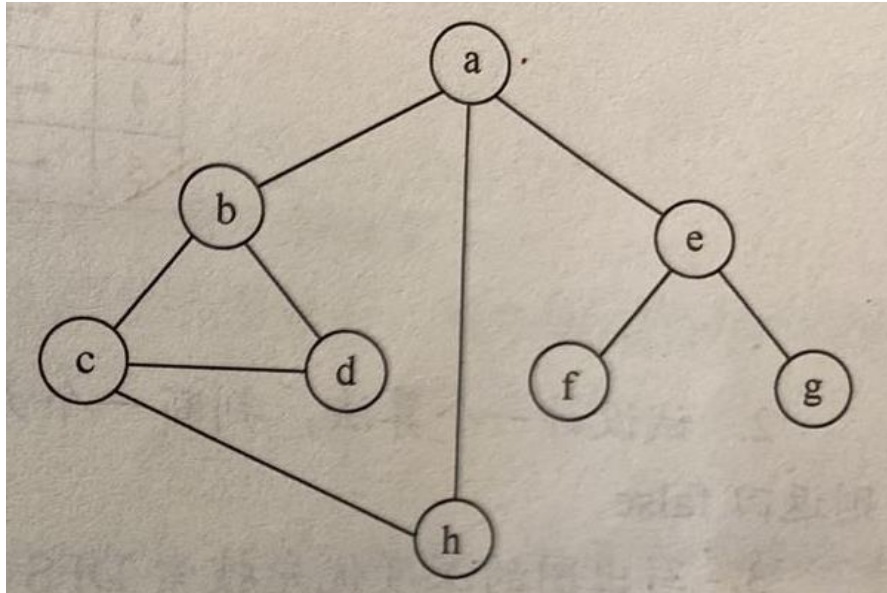
AOV网，AOE网，都能表示工程各子工程的流程，一个用顶点表示活动，一个用边表示活动，但AOV网侧重表示活动的前后次序，AOE网除表示活动先后次序，还表示了活动的持续时间等，因此可利用AOE网解决工程所需最短时间及哪些子工程拖延会影响整个工程按时完成等问题。

实际应用中采用哪一种图，取决于要求解的问题。

练习题

单项选择题：

1. 若对如下无向图进行遍历，则下列选项中，不是广度优先遍历序列的是（ ）。

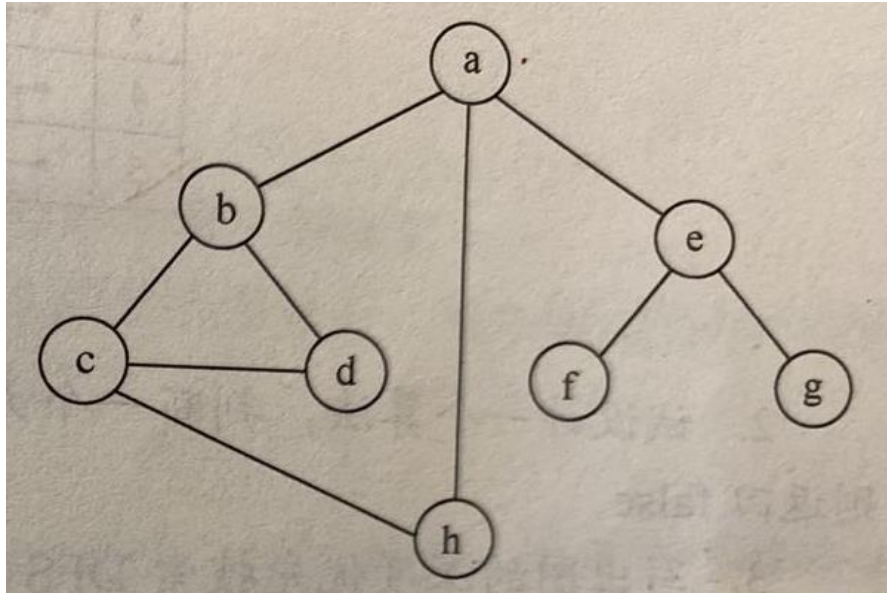


- A. h, c, a, b, d, e, g, f
- B. e, a, f, g, b, h, c, d
- C. d, b, c, a, h, e, f, g
- D. a, b, c, d, h, e, f, g

练习题

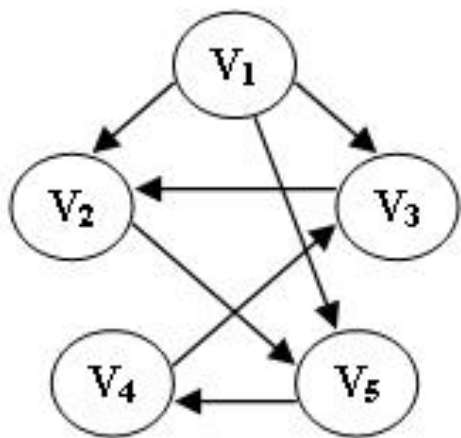
单项选择题：

1. 若对如下无向图进行遍历，则下列选项中，不是广度优先遍历序列的是（ ）。



- A. h, c, a, b, d, e, g, f
- B. e, a, f, g, b, h, c, d
- C. d, b, c, a, h, e, f, g
- D. a, b, c, d, h, e, f, g

2. 下列选项中，不是下图深度优先搜索序列的是()。



- A. V1, V5, V4, V3, V2
- B. V1, V3, V2, V5, V4
- C. V1, V2, V5, V4, V3
- D. V1, V2, V3, V4, V5

3. 下列关于最小生成树的叙述中，正确的是()。

- I. 最小生成树的代价唯一
- II. 权值最小的边一定会出现在所有的最小生成树中
- III. 使用普里姆(Prim)算法从不同顶点开始得到的最小生成树一定相同
- IV. 使用普里姆算法和克鲁斯卡尔(Kruskal)算法得到的最小生成树总不相同

- A. 仅 I
- B. 仅 II
- C. 仅 I、III
- D. 仅 II、IV

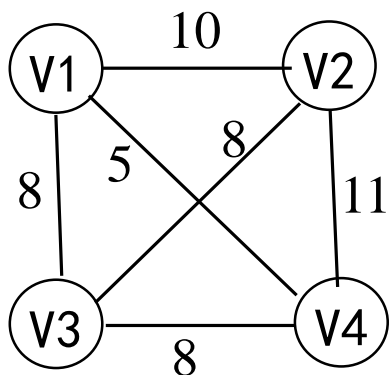
4. 若一个有向图中的顶点不能排成一个拓扑序列，则可断定该有向图()。

- A. 是个有根的有向图
- B. 是个强连通图
- C. 含有多个入度为0的顶点
- D. 含有顶点数目大于1的强连通分量

5. 下列关于图的说法中，正确的是()。

- I. 有向图中顶点V的度等于其邻接矩阵中第V行中的1的个数
 - II. 无向图的邻接矩阵一定是对称矩阵，有向图的邻接矩阵一定是非对称矩阵
 - III. 在图G的最小生成树，可能会有某条边的权值超过未选边的权值
 - IV. 如果有向无环图的拓扑序列唯一，则可以唯一确定该图
- A. I、II和III B. III和IV C. III D. IV

6. 求下面带权图的最小(代价)生成树时, 可能是克鲁斯卡尔(Kruskal)算法第二次选中但不是普里姆(Prim)算法(从V4开始)第2次选中的边是()。



- A. (v1,v3) B. (V1,V4) C.(v2,v3) D.(v3,v4)

7.6 最 短 路 径

假设用带权的有向图表示一个**交通运输网**，图中：

顶点—表示城市

边— 表示城市间的交通联系

权— 表示此线路的长度或沿此线路运输所花的时间或费用等

在城市A（**源点**）到达城市B（**终点**）的多条路径中，寻找一条**各边权值之和最小**的路径，即**最短路径**。

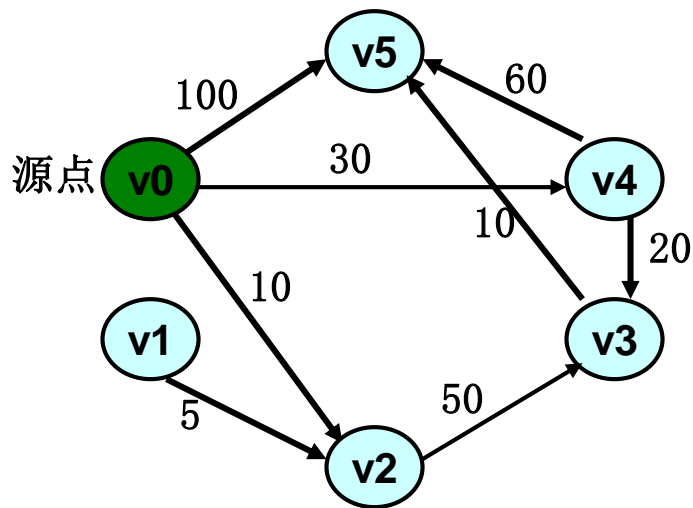
两种常见的最短路径问题：

1、从某个源点到其余各顶点的最短路径

Dijkstra (迪杰斯特拉) 算法

2、任意两顶点之间的最短路径

Floyd (弗洛伊德) 算法

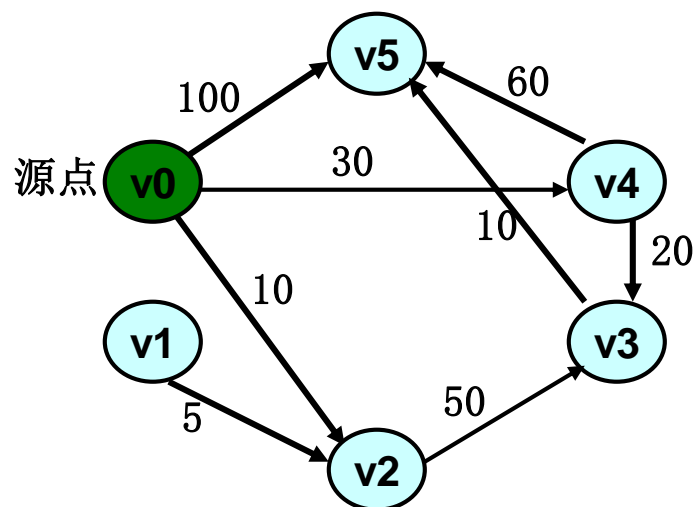


有向网G6

- 从某个源点到其余各顶点的最短路径 -----Dijkstra算法

Dijkstra提出了一个按路径长度递增的次序产生各顶点的最短路径的算法，称为Dijkstra算法。

例如下图所示的有向网G6，假定以顶点v0为源点，则源点到其余各顶点的最短路径如表6. 2所示。



有向网G6

表6. 2

有向网G6中从v0到其余各点的最短路径：

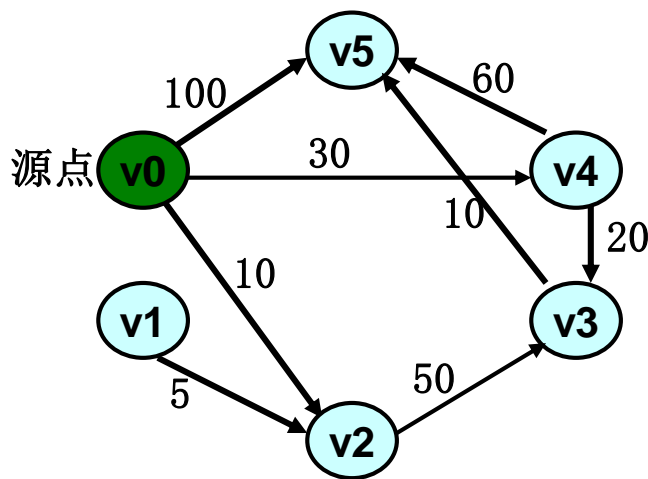
源点	终点	最短路径	路径长度
v0	v2	(v0,v2)	10
	v4	(v0,v4)	30
	v3	(v0,v4,v3)	50
	v5	(v0, v4, v3, v5)	60
	v1	无	

1. Dijkstra算法的求解过程

首先，采用一个辅助向量D，它的每个分量D[i]表示当前所找到的从源点v0到每个终点vi的最短路径的长度。它的初态为：若存在 $\langle v_0, v_i \rangle$ ，则D[i]为弧上的权值；否则置D[i]为 ∞ 。

$$\text{长度为 } D[j] = \min_i \{ D[i] \mid v_i \in V \}$$

的路径就是从v0出发的长度最短的一条最短路径。此路径为(v0, vj)。

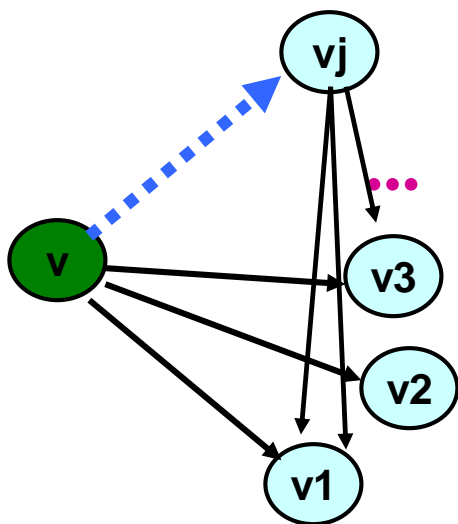


有向网G6

	v0	v1	v2	v3	v4	v5
v0	∞	∞	10	∞	30	100
v1	∞	∞	5	∞	∞	∞
v2	∞	∞	∞	50	∞	∞
v3	∞	∞	∞	∞	∞	10
v4	∞	∞	∞	20	∞	60
v5	∞	∞	∞	∞	∞	∞

数组D

0	∞
1	∞
2	10
3	∞
4	30
5	100



如左图所示， v_j 是所有从源点出发的弧中权值最小的弧的弧头。

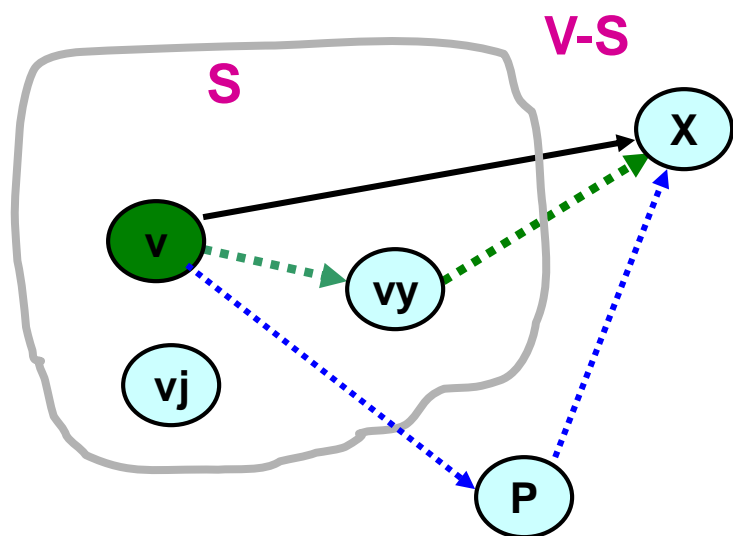
下一条长度次短的最短路径是哪一条呢？

• 长度次短的路径可能有两种情况：

它可能是从源点直接到该点的路径；

也可能是，从源点到 v_j ，再从 v_j 到该点

Dijkstra 算法: 正确性证明



假设 S 为已求得最短路径的终点的集合, 则可证明: 下一条最短路径 (设其终点为 x) 或者是弧 (v, x) , 或者是中间只经过 S 中的顶点而最后到达顶点 x 的路径。

用反证法来证明:

假设此路径上有一个顶点不在 S 中, 则说明存在一条终点不在 S 而长度比此路径短的路径。但这假设是不可能的。因为我们是按路径长度递增的次序来产生各最短路径的, 故长度比此路径短的所有路径均已产生, 它们的终点必定在 S 中, 即假设不成立。

因此，下一条长度次短的最短路径的长度必是

$$D[j] = \underset{i}{\text{Min}} \{ D[i] \mid v_i \in V - S \}$$

其中， $D[i]$ 或者是弧 $\langle v, v_i \rangle$ 上的权值，或者是 $D[k]$ ($v_k \in S$)和弧 $\langle v_k, v_i \rangle$ 上的权值之和。

Dijkstra 算法的步骤:

(1). 假设用带权的邻接矩阵arcs来表示带权有向图, arcs[i][j]表示弧 $\langle v_i, v_j \rangle$ 上的权值。若 $\langle v_i, v_j \rangle$ 不存在, 则置arcs[i][j]为 ∞ 。**S为已找到从v出发的最短路径的终点集合**, 它的初始状态为空集。那么, 从v出发到图上其余各顶点(终点) v_i 可能达到的最短路径长度的初值为:

$$D[i] = \text{arcs}[\text{Locate Vex}(G, v)][i] \quad v_i \in V$$

(2). 选择 v_j , 使得

$$D[j] = \text{Min}\{ D[i] \mid v_i \in V-S \}$$

v_j 就是当前求得的一条从v出发的最短路径的终点。令

$$S = S \cup \{ j \}$$

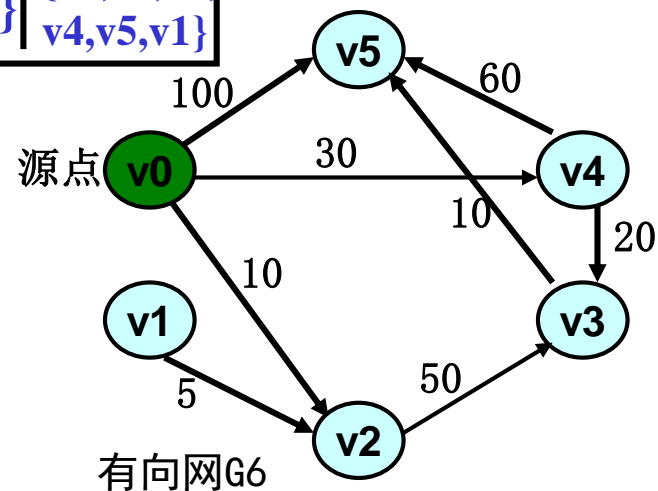
(3). 修改从v出发到集合V-S上任一顶点 v_k 可达的最短路径长度。如果

$$D[j] + \text{arcs}[j][k] < D[k]$$

则修改D[k]为 $D[k] = D[j] + \text{arcs}[j][k]$

(4). 重复操作(2)、(3)共n-1次。由此求得从v到图上其余各顶点的最短路径是依路径长度递增的序列。

终点	从v0到各终点的D值和最短路径的求解过程				
	第1趟	第2趟	第3趟	第4趟	第5趟
v1	∞	∞	∞	∞	∞
v2	10 (v0,v2)				
v3	∞	60 (v0,v2,v3)	50 (v0,v4,v3)		
v4	30 (v0,v4)	30 (v0,v4)			
v5	100 (v0,v5)	100 (v0,v5)	90 (v0,v4,v5)	60 (v0,v4,v3,v5)	
vj	v2	v4	v3	v5	
S	{v0,v2}	{v0,v2,v4}	{v0,v2,v3,v4}	{v0,v2,v3,v4,v5}	{v0,v2,v3,v4,v5,v1}



2. Dijkstra算法的实现

假设用带权的邻接矩阵arcs来表示带权有向图G, $G.arcs[i][j]$ 表示弧 $\langle v_i, v_j \rangle$ 上的权值。若 $\langle v_i, v_j \rangle$ 不存在, 则置 $G.arcs[i][j]$ 为 ∞ , 源点为 v_0 。

算法的实现要引入以下辅助的数据结构:

- (1) **一维数组S[i]**: 记录从源点 v_0 到终点 v_i 是否已被确定最短路径长度, true表示确定, false表示尚未确定。

(2) **一维数组Path[i]**: 记录从源点 v_0 到终点 v_i 的当前最短路径上 v_i 的直接前驱顶点序号。其初值为: 如果 v_0 到 v_i 有弧, 则Path[i]为 v_0 , 否则为-1。

(3) **一维数组D[i]**: 记录从源点 v_0 到终点 v_i 的当前最短路径长度。其初值为: 如果 v_0 到 v_i 有弧, 则D[i]为弧上的权值, 否则为 ∞ 。

Dijkstra算法描述如下：

```
void ShortestPath_DIJ (MGraph G, int v0)
{ //用Dijkstra算法求有向网G的v0顶点到其余顶点v的最短路径

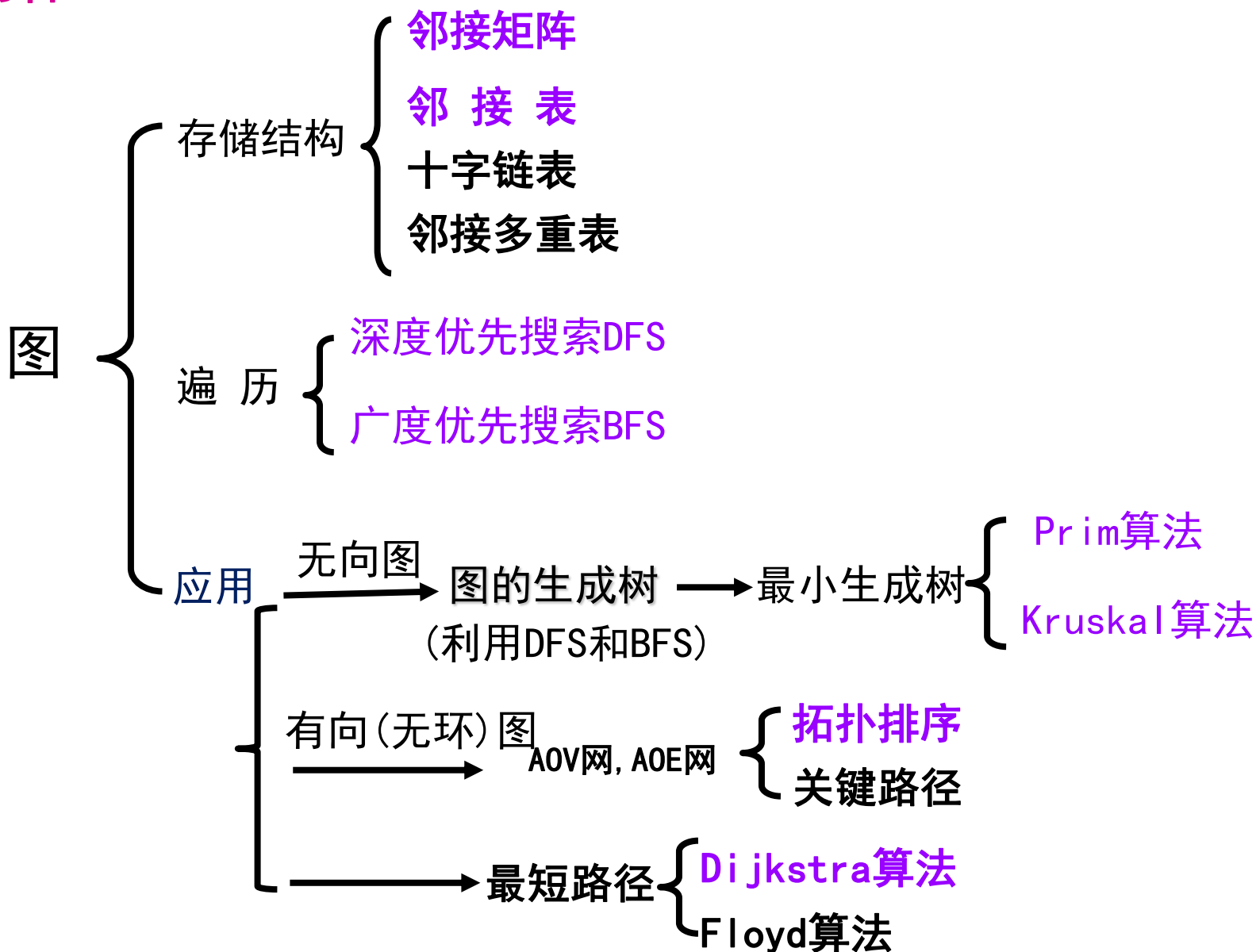
    n=G.vexnum;
    for (v = 0; v < n; ++ v)
    {
        S[v] = false;
        D[v] = G.arcs[v0][v];
        //将v0到各个终点的最终路径长度初始化为权值
        if (D[v]< MaxInt) Path[v]= 0;
        else Path[v]=-1;
    }
    D[v0] = 0;    S[v0] = true;    //初始化， v0顶点加入S集
    /*---初始化结束---*/
}
```

```

//开始主循环，每次求得v0到某个顶点v的最短路径，将v加到S集
for (i = 1; i < n; ++ i) {           //其余n-1个顶点
    min = MaxInt;
    for (w = 0; w < n; ++ w)
        if (!S[w] && D[w]< min)
            { v = w;  min = D[v];} //选择一条当前的最短路径，终点为v
    S[v] = true;           //将v加入S
    for (w = 0; w < n; ++ w)
        //更新从v0出发到集合V-S上所有顶点的最短路径长度
        if (!S[w] & & (min + G.arcs[v][w] < D[w]))
            { // w不属于S集且v0→v→w的距离<目前v0→w的距离
                D[w] = min + G.arcs[v][w]; //修改D[w], w ∈ V-S
                Path[w] = v;
            }
    }
}

```

基本内容：



“巧渡河”问题

- 问题：人、狼、羊、菜用一条只能同时载两位的小船渡河，“狼羊”、“羊菜”不能在无人在场时共处，当然只有人能架船。



问题：

你认为针对这个问题，合适的抽象应该是什么样的？即什么作为“点”，什么作为“边”？

“巧渡河” 问题

- 图模型：
 - 顶点表示“原岸的状态”；
 - 两点之间有边当且仅当一次合理的渡河“操作”能够实现该状态的转变；
- 起始状态是“人狼羊菜”，结束状态是“空”。
- 问题的解：找到一条从起始状态到结束状态的尽可能短的通路。

“巧渡河”问题的解

- 注意：在“人狼羊菜”的16种组合中允许出现的只有10种。

