第七章

图

7.1 抽象数据类型图的定义

7.2 图的存储表示

7.3 图的遍历

7.4 最小生成树

7.5 重（双）连通图和关节点

7.6 两点之间的最短路径问题

7.7 拓扑排序

7.8 关键路径

图的结构定义:

图是由一个顶点集 **V** 和一个弧集 **R**构成的数据结构。

Graph = (V , VR)

其中，VR＝{<v,w>| v,w∈V 且 P(v,w)}

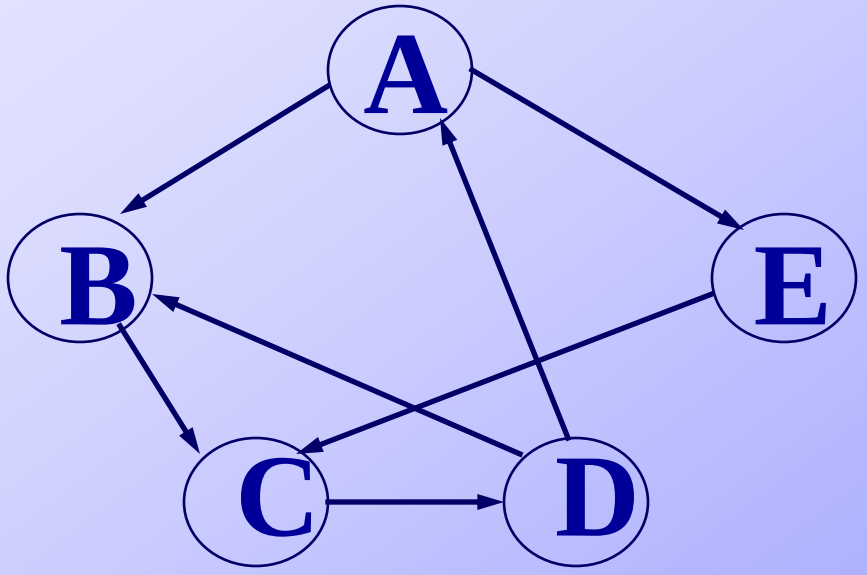
<v,w>表示从 v 到 w 的一条弧，并称 v 为**弧尾**，w 为**弧头**。//V→W

谓词 P(v,w) 定义了弧 <v,w>的意义或信息。

注：图当中的任何2个顶点都可能发生关系，不能用一句话来表示，要用谓词来表示，要用一组有序对来表示。

由于“弧”是有方向的，因此称由顶点集和弧集构成的图为**有向图**。

例如： $G_1 = (V_1, VR_1)$



其中

V_1={A, B, C, D, E}

VR_1={<A,B>, <A,E>,
<B,C>, <C,D>, <D,B>,
<D,A>, <E,C> }

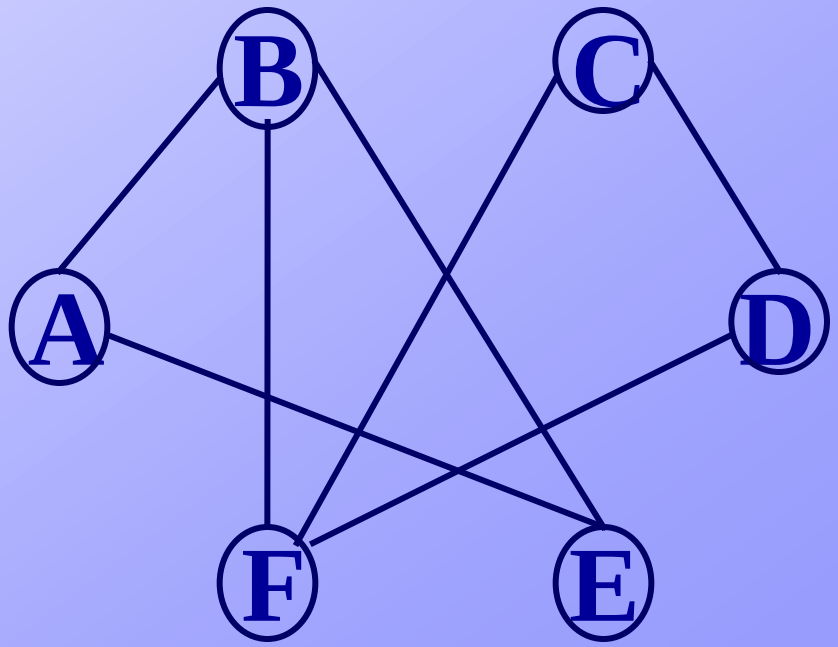
若<v, w>∈VR 必有<w, v>∈VR, 则称 (v,w) 为顶点v 和顶点 w 之间存在一条**边**。

由顶点集和边集构成的图称作**无向图**。

例如: $G_2=(V_2,VR_2)$

$V_2=\{A, B, C, D, E, F\}$

$VR_2=\{<A,B>, <A,E>,$
$<B,E>, <C,D>, <D,F>,$
$<B,F>, <C,F> \}$



名词和术语

网、子图 ➡

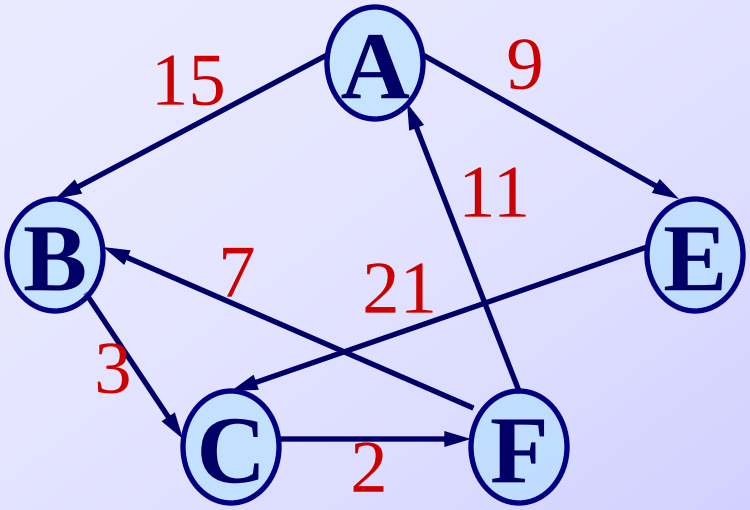
完全图、稀疏图、稠密图 ➡

邻接点、度、入度、出度 ➡

路径、路径长度、简单路径、简单回路➡

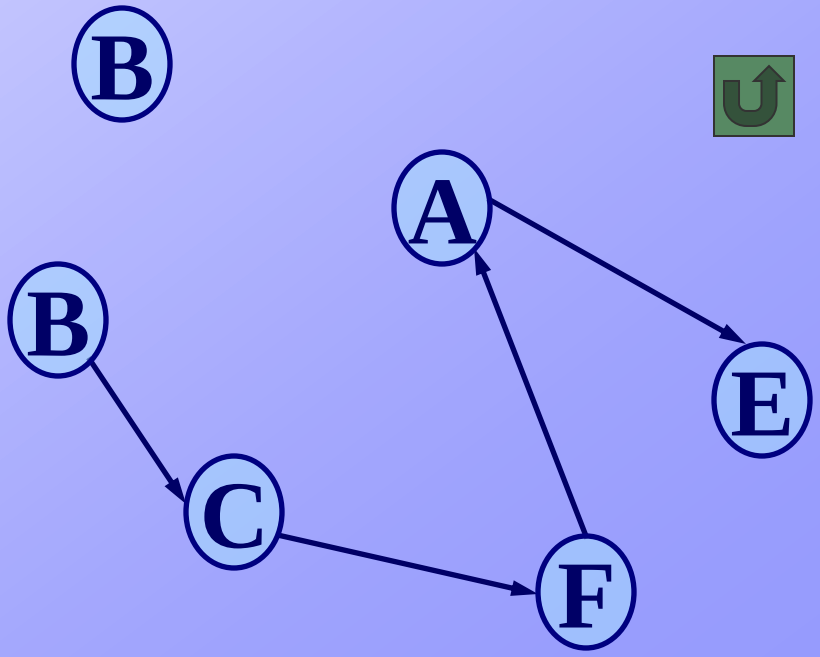
连通图、连通分量、
强连通图、强连通分量 ➡

生成树、生成森林 ➡



弧或边带权的图分别称作**有向网**或**无向网**。

设图G=(V,{VR}) 和图 G′=(V′,{VR′}), 且 V′⊆V, VR′⊆VR, 则称 G′ 为 G 的**子图**。



假设图中有 **n** 个顶点，**e** 条边，则

含有 **e=n(n-1)/2** 条边的无向图称作**完全图**；

含有 **e=n(n-1)** 条弧的有向图称作**有向完全图**；

若边或弧的个数 **e<nlogn**，则称作**稀疏图**，否则称作**稠密图**。

假若顶点**v** 和顶点**w** 之间存在一条边，则称顶点**v** 和**w** 互为**邻接点**，

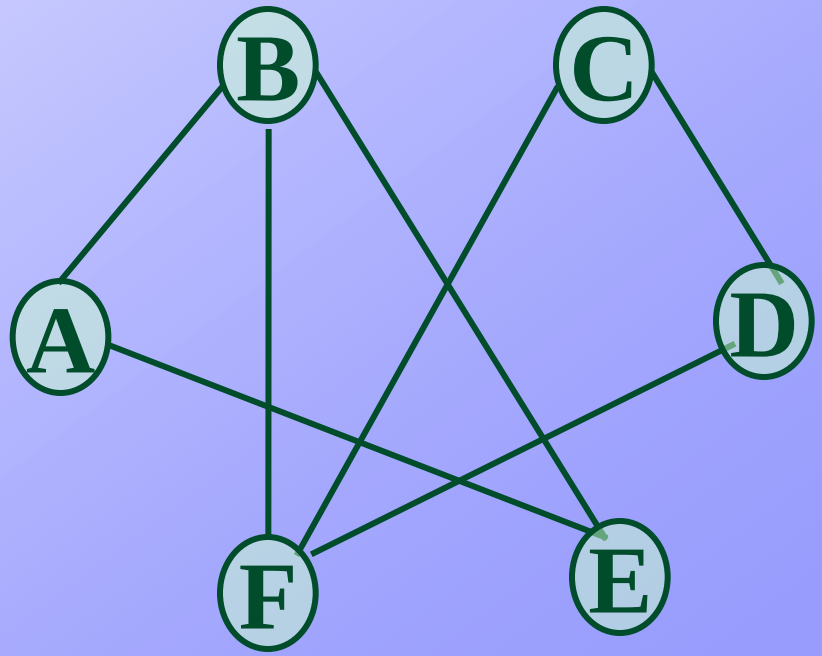
边**(v,w)** 和顶点**v** 和**w** 相**关联**。

和顶点**v** 关联的**边的数目**定义为顶点的**度**。

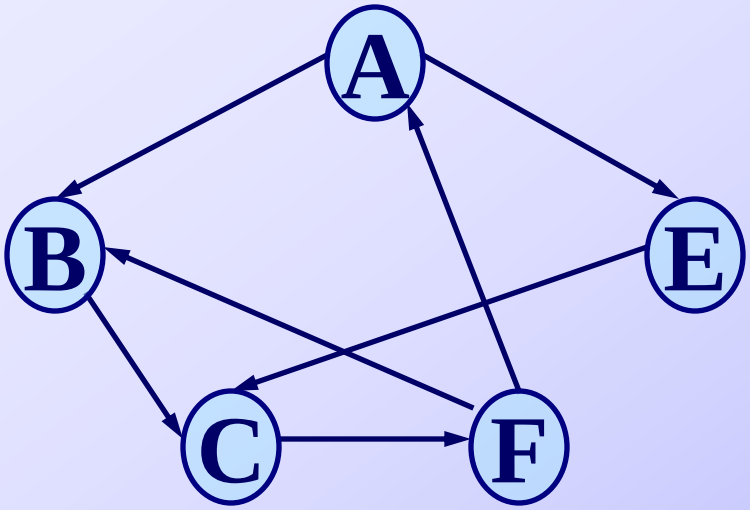
例如：

ID(B) = 3

ID(A) = 2



对有向图来说，



例如：

OD(B) = 1

ID(B) = 2

TD(B) = 3

顶点的**出度**：以顶点v为弧尾的弧的数目；

顶点的**入度**：以顶点v为弧头的弧的数目。

顶点的**度(TD)=出度(OD)+入度(ID)**

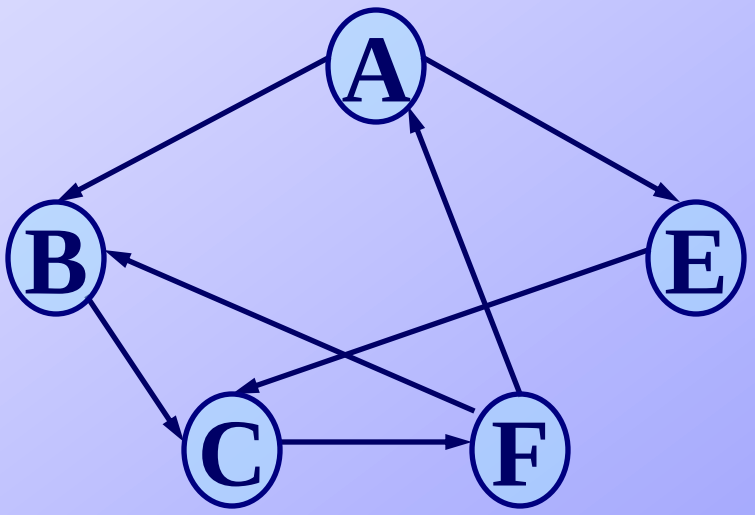
设图G=(V,{VR})中的一个顶点序列

{ $u=v_{i,0},v_{i,1}, \ldots, v_{i,m}=w$}中，$(v_{i,j-1},v_{i,j})\in VR$ $1\le j\le m$,

则称从顶点u 到顶点w 之间存在一条**路径**。

路径上边的数目称作**路径长度**。

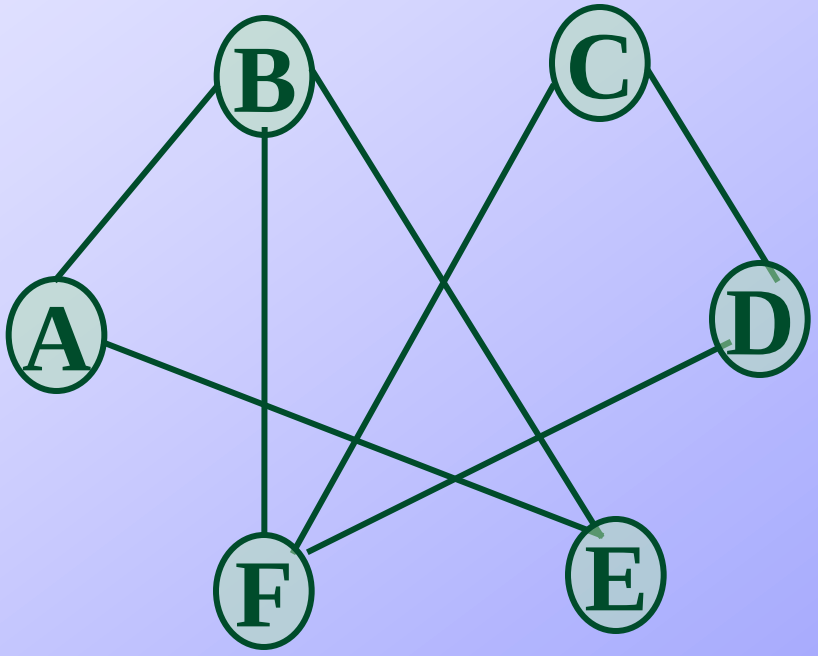
如:长度为3的路径{A,B,C,F}

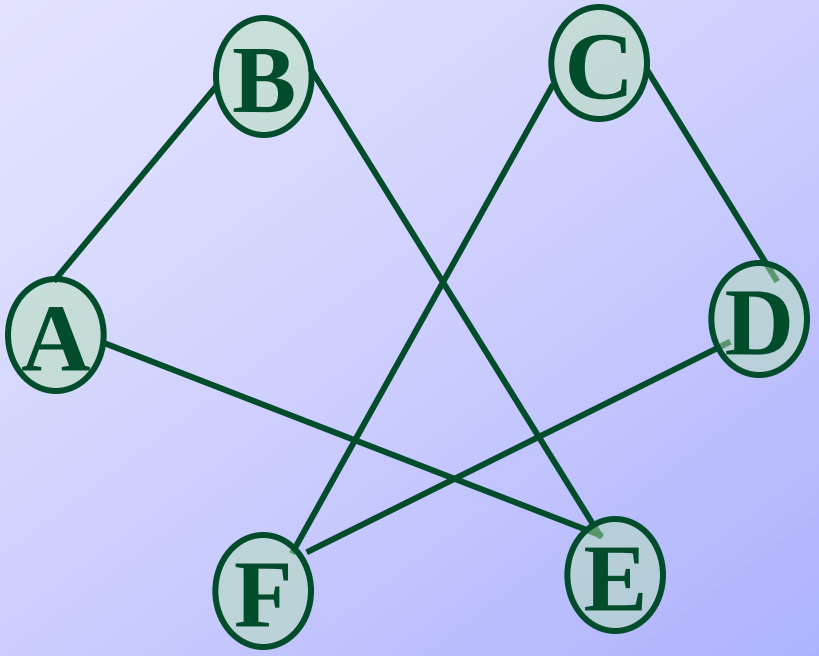


简单路径:序列中顶点不重复出现的路径。

简单回路:序列中第一个顶点和最后一个顶点相同的路径。

若图G中任意两个顶点之间都有路径相通，则称此图为**连通图**；

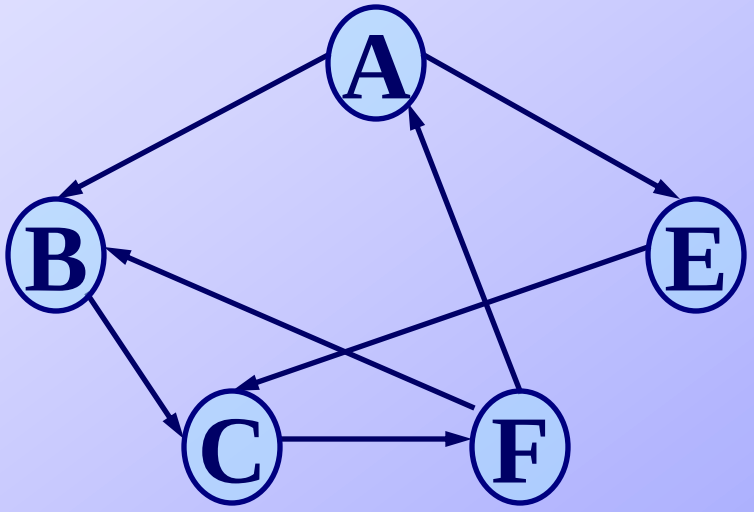


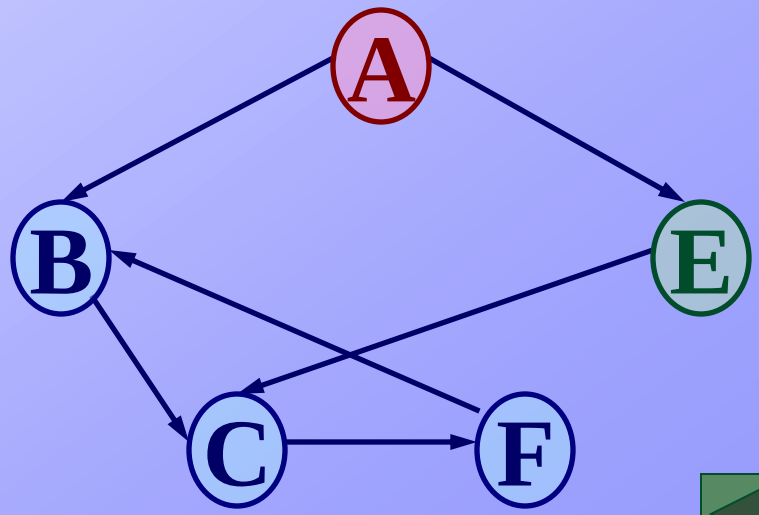


若无向图为非连通图，则图中各个极大连通子图称作此图的**连通分量**。

对有向图，若任意两个顶点之间都存在一条有向路径，则称此有向图为**强连通图**。

否则，其各个极大的强连通子图称作它的**强连通分量**。

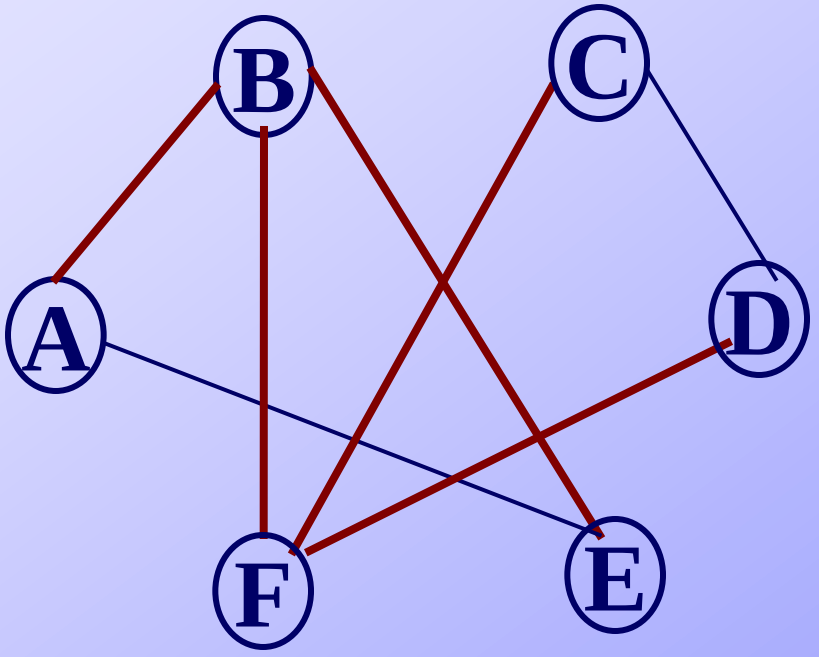




假设一个连通图有 n 个顶点和 e 条边，其中 n-1 条边和 n 个顶点构成一个极小连通子图，称该极小连通子图为此连通图的**生成树**。

注：1，加上一条边就有环，减去一条边则是非连通图。但是有n-1条边的图不一定是生成树。

2，一个连通图可以找到若干个生成树。



对非连通图，则称由各个连通分量的生成树的集合为此非连通图的**生成森林**。

基本操作

结构的建立和销毁

对顶点的访问操作

插入或删除顶点

插入和删除弧

对邻接点的操作

遍历

结构的建立和销毁

CreatGraph(&G, V, VR):

// 按定义(V, VR) 构造图

DestroyGraph(&G):

// 销毁图

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。

//顶点的位置的含义：计算机已经建好了图的话，顶点之间的
//位置已经确定了，这里就是查找顶点在图的存储结构中的位置,后面讲存储结构的时候会更明确

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”。

/*对于以某种存储结构建立的图，v有好多个邻接点，返回该结构下和v第一个相邻的邻接点。*/

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”。

/*如果w是v的第一个邻接点，则返回第二个邻接点，如果w是第二个邻接点，则返回第三个邻接点。*/

插入或删除顶点

InsertVex(&G, v);

//在图G中增添新顶点v。

//增加一个孤立的顶点

DeleteVex(&G, v);

// 删除G中顶点v及其相关的弧。

//即删除的时候还要删除相关的弧

插入和删除弧

InsertArc(&G, v, w);

// 在G中增添弧<v,w>，若G是无向的，

//则还增添对称弧<w,v>。

DeleteArc(&G, v, w);

//在G中删除弧<v,w>，若G是无向的，

//则还删除对称弧<w,v>。

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。

注：图的各种操作在相应的图的存储结构建立以后，很容易实现，不同的存储结构有不同的实现方法。

7.2 图的存储表示

一、图的数组(邻接矩阵)存储表示

二、图的邻接表存储表示

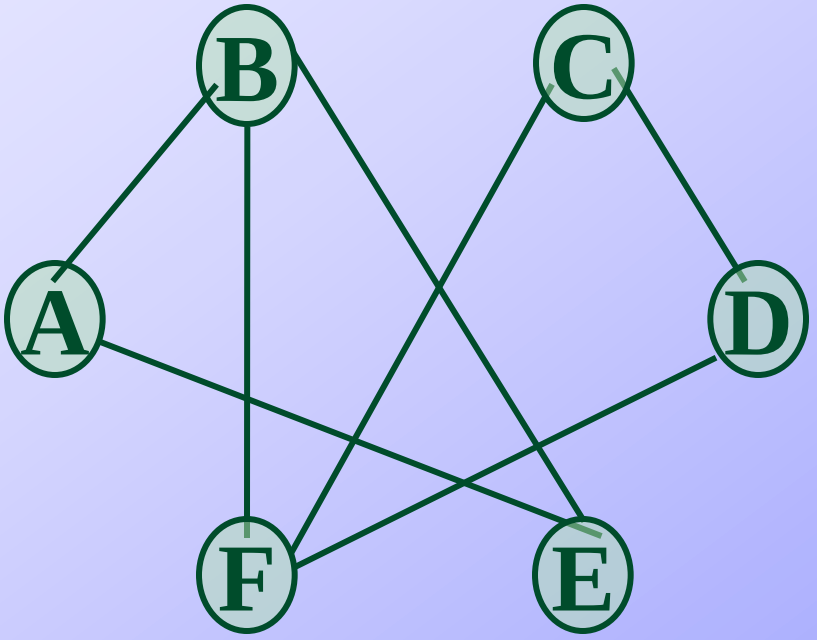
三、有向图的十字链表存储表示

四、无向图的邻接多重表存储表示

一、图的数组(邻接矩阵)存储表示

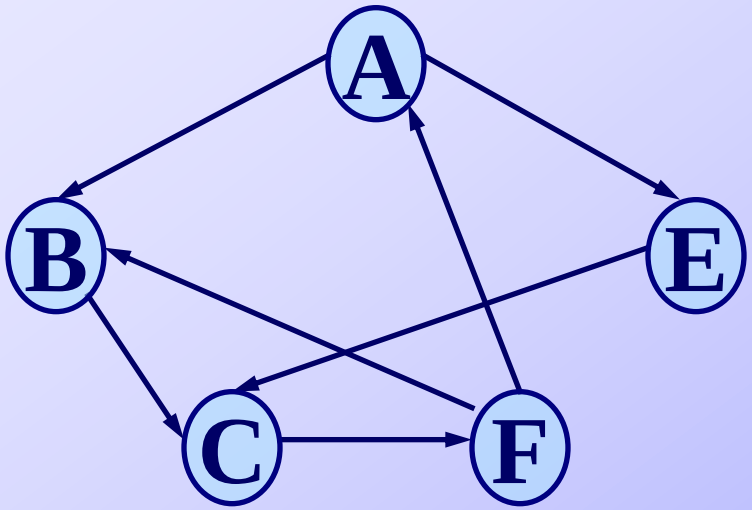
定义:矩阵的元素为

$$A_{ij}=\begin{cases}0 & (i,j)\notin VR\\ 1 & (i,j)\in VR\end{cases}$$



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

有向图的邻接矩阵为非对称矩阵

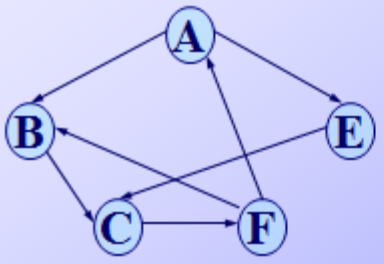


0	1	0	0	1
0	0	1	0	0
0	0	0	1	0
1	1	0	0	0
0	0	1	0	0

定义图存储结构的要素

边集　顶点集　顶点数量　边的数量　图的种类

```
typedef struct { // 图的定义
    VertexType      // 顶点信息
                    vexs[MAX_VERTEX_NUM];
    AdjMatrix   arcs;     // 弧的信息
    int   vexnum, arcnum;  // 顶点数，弧数
    GraphKind   kind;     // 图的种类标志
} MGraph;
```



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

typedef struct ArcCell { // 弧的定义

VRType adj; // VRType是顶点关系类型。

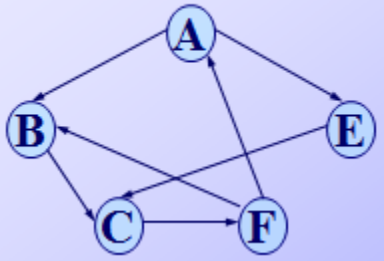
// 对无权图，用1或0表示相邻否；

// 对带权图，则为权值类型。

InfoType *info; // 该弧相关信息的指针

} ArcCell,

AdjMatrix[MAX_VERTEX_NUM]

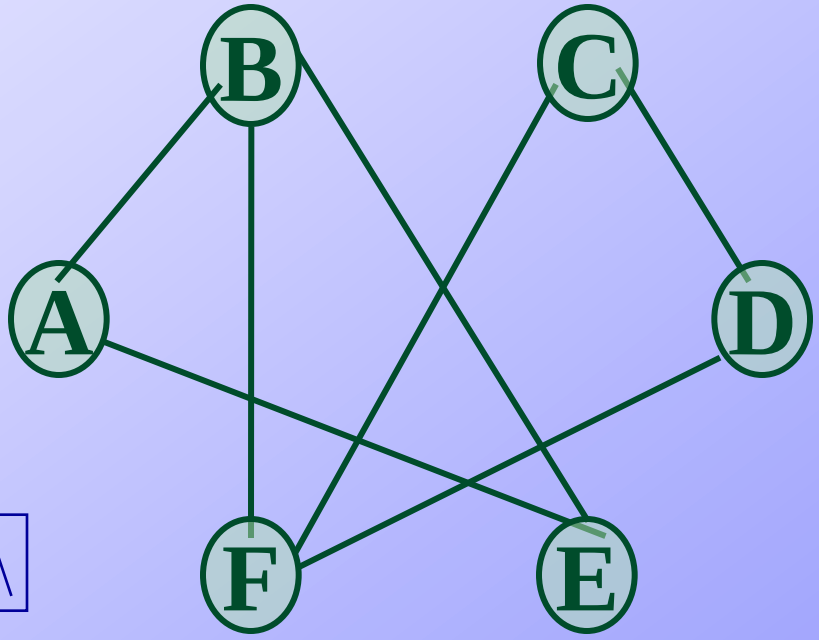


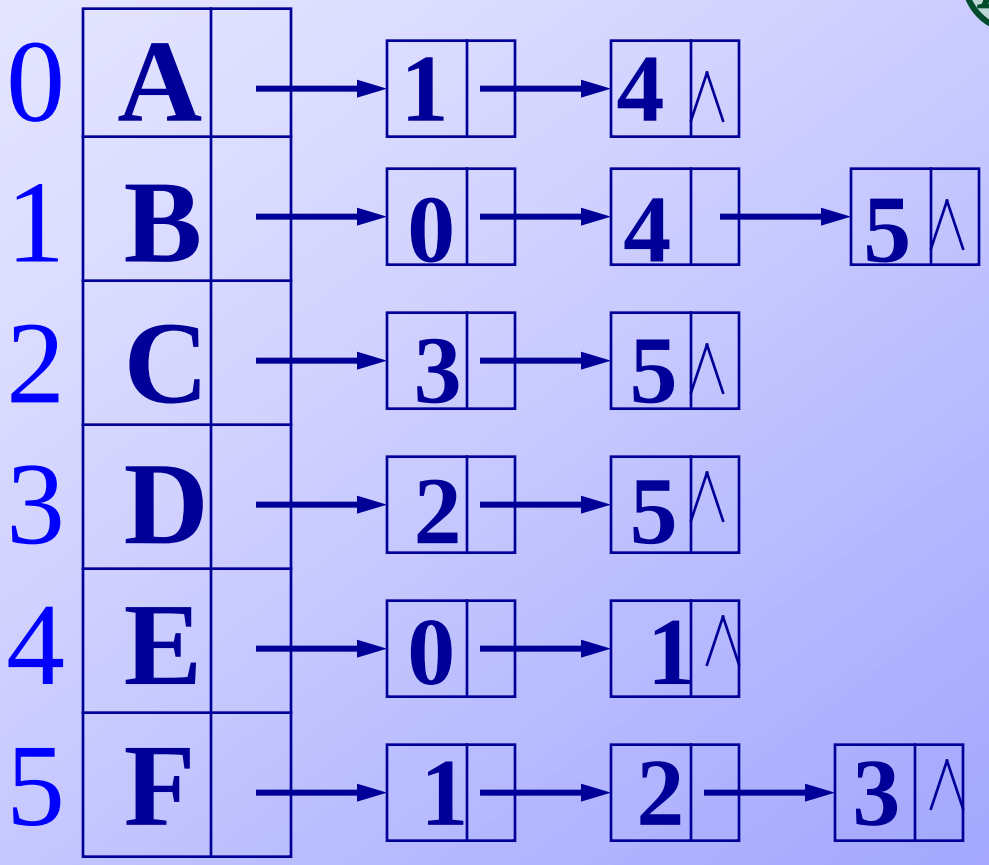
0	1	0	0	1
0	0	1	0	0
0	0	0	1	0
1	1	0	0	0
0	0	1	0	0

[MAX_VERTEX_NUM];

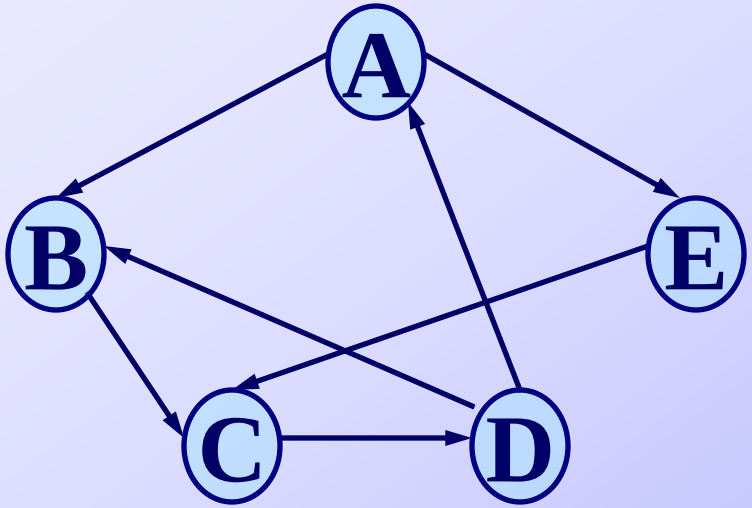
二、图的邻接表

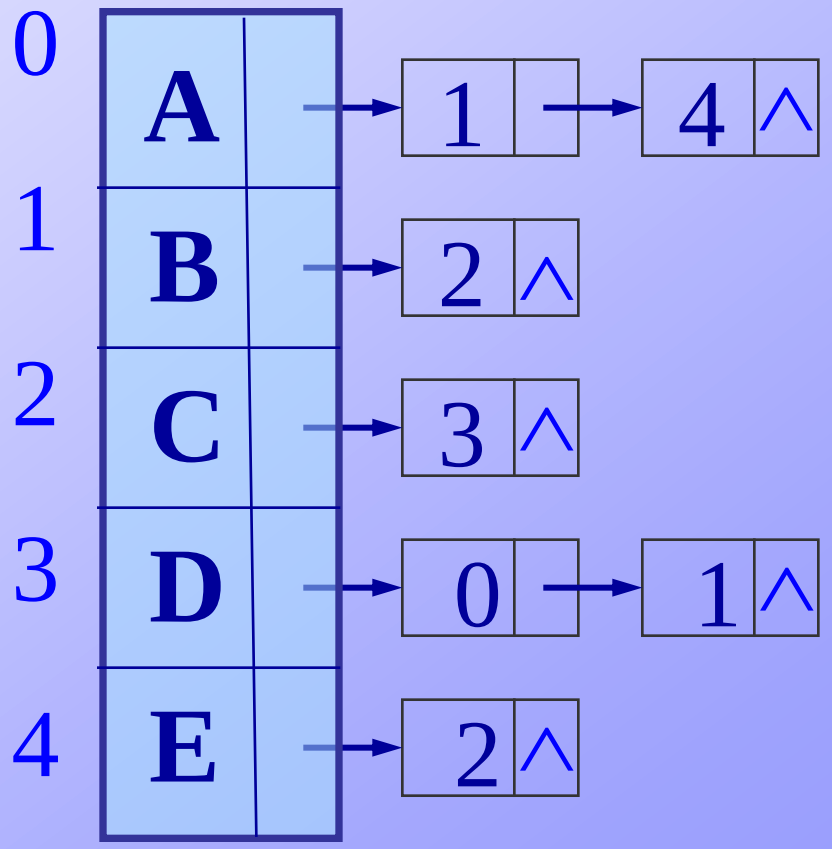
存储表示





有向图的邻接表

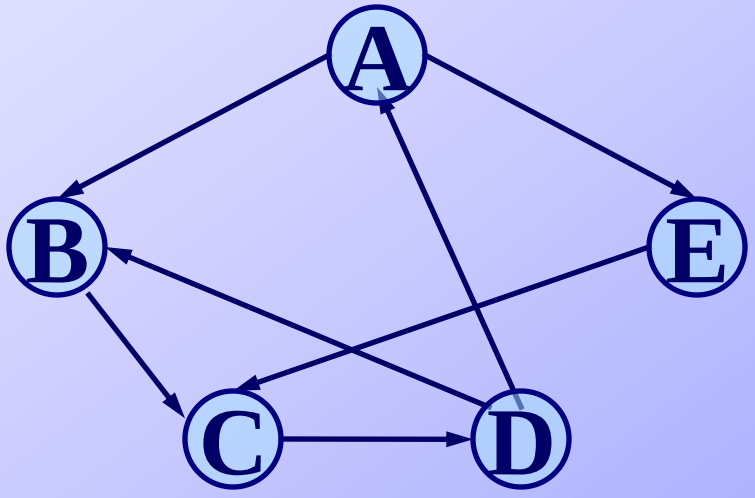




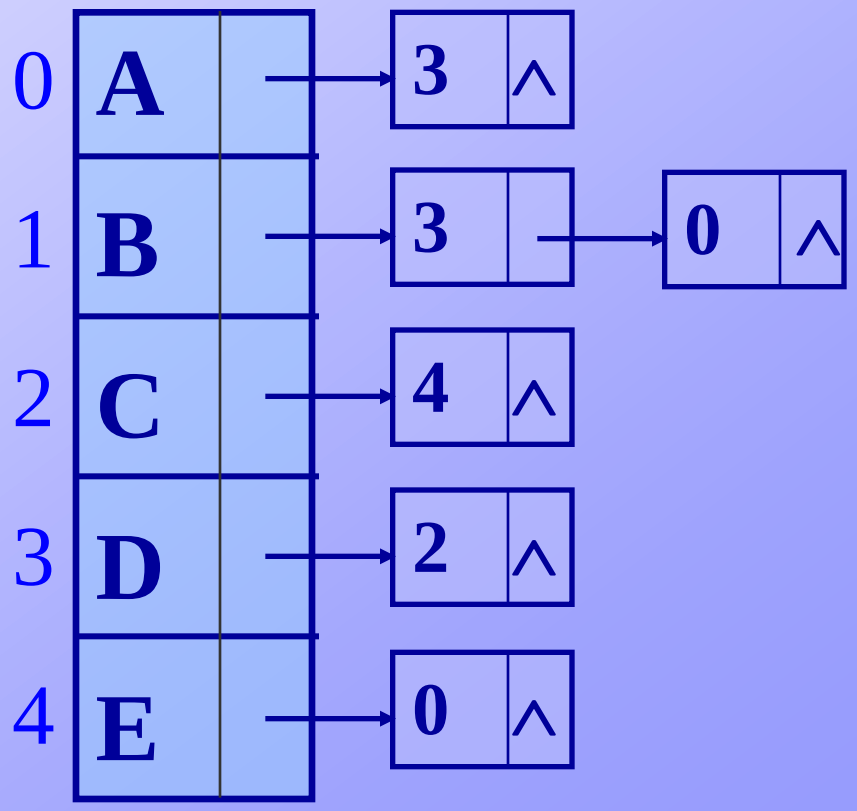
可见，在有向图的邻接表中不易找到指向该顶点的弧。

计算入度不方便

有向图的逆邻接表



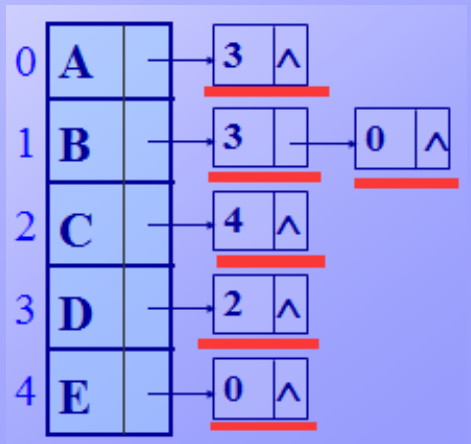
在有向图的逆邻接表中，对每个顶点，链接的是指向该顶点的弧。计算出度不方便



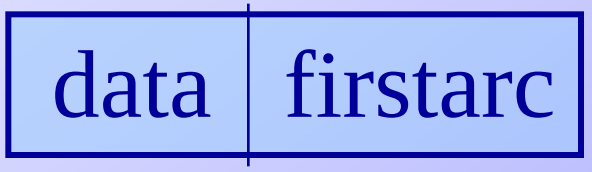
弧的结点结构

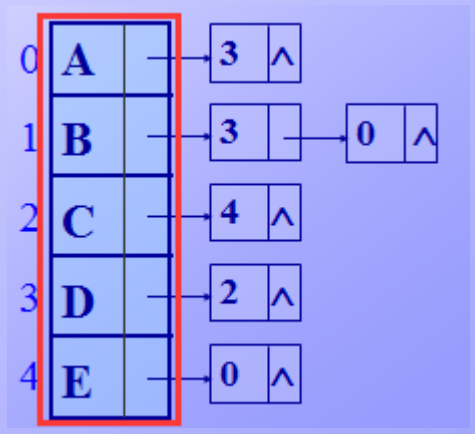
adjvex	nextarc	info

```
typedef struct ArcNode {
  int       adjvex;   // 该弧所指向的顶点的位置
  struct ArcNode  *nextarc;
                      // 指向下一条弧的指针
  InfoType   *info;   // 该弧相关信息的指针
} ArcNode;
```



顶点的结点结构





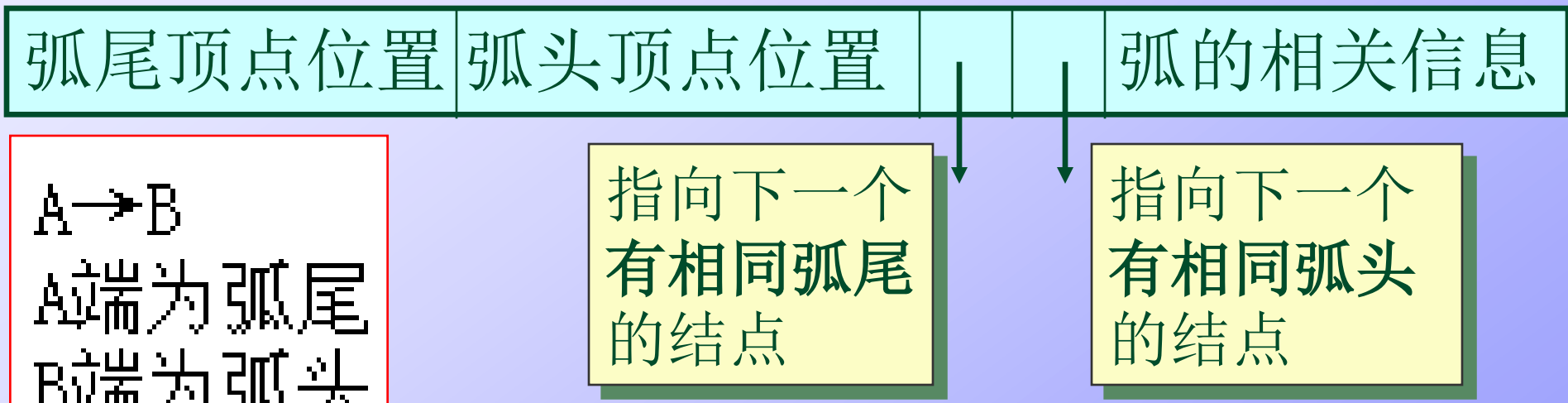

```
typedef struct VNode {
  VertexType  data;   // 顶点信息
  ArcNode  *firstarc;
                // 指向第一条依附该顶点的弧
} VNode, AdjList[MAX_VERTEX_NUM];
```

图的结构定义

```
typedef struct {
    AdjList  vertices;
    int      vexnum, arcnum;
    int      kind;          // 图的种类标志
} ALGraph;
```

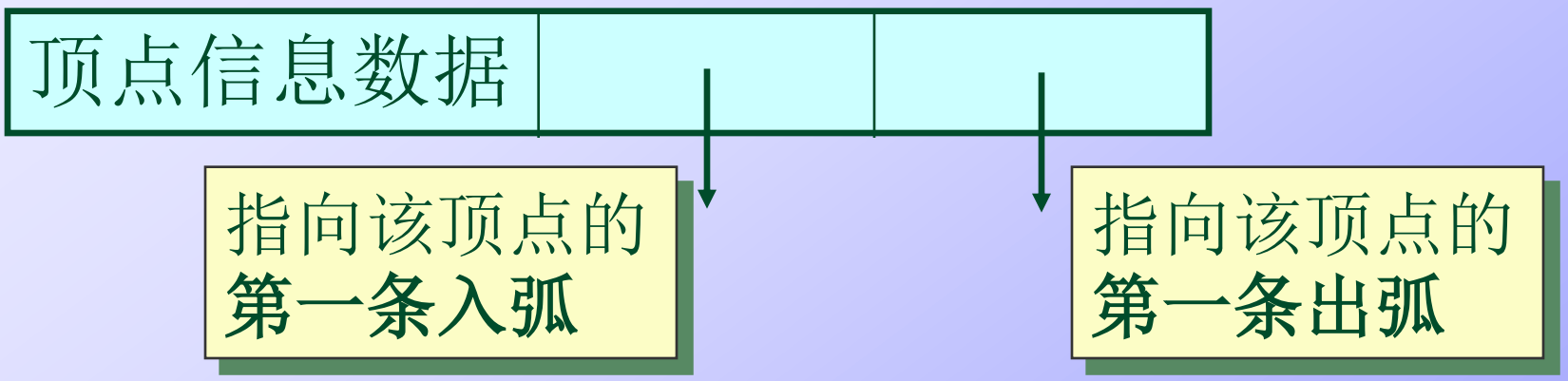
三、有向图的十字链表存储表示

弧的结点结构




```
typedef struct ArcBox { // 弧的结构表示
    int tailvex, headvex;   InfoType  *info;
    struct ArcBox  *hlink, *tlink;
    } ArcBox;
```

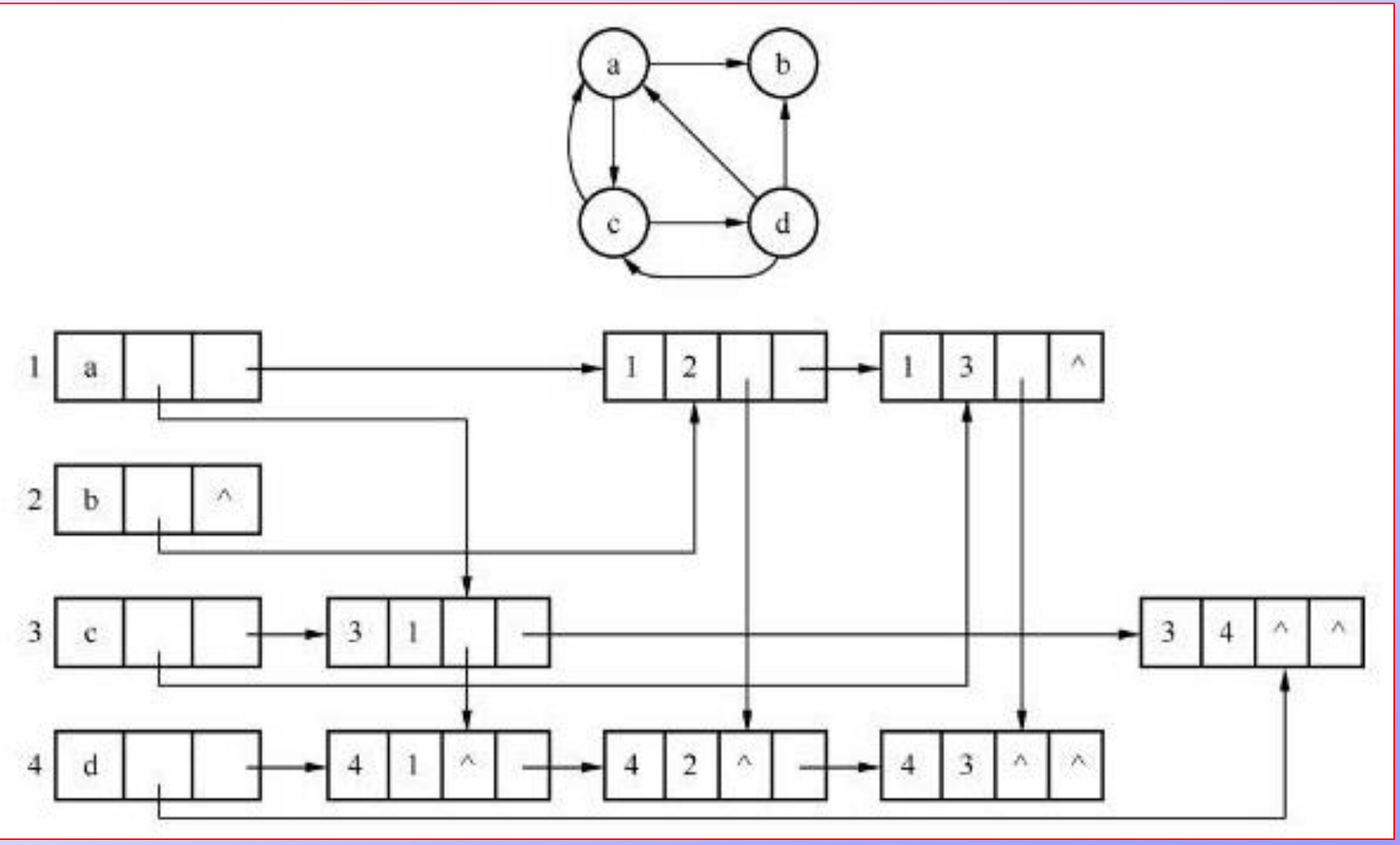
顶点的结点结构




```
typedef struct VexNode { // 顶点的结构表示
    VertexType  data;
    ArcBox  *firstin, *firstout;
} VexNode;
```

有向图的结构表示(十字链表)

```
typedef struct {
   VexNode  xlist[MAX_VERTEX_NUM];
            // 顶点结点(表头向量)
   int   vexnum, arcnum;
            //有向图的当前顶点数和弧数
} OLGraph;
```



有向图的十字链表存储结构计算出度和入度都很方便

四、无向图的邻接多重表存储表示

边的结构表示

```
typedef struct Ebox {
    VisitIf      mark;      // 访问标记
    int      ivex, jvex;
                //该边依附的两个顶点的位置
    struct EBox  *ilink, *jlink;
    InfoType     *info;          // 该边信息指针
} EBox;
```

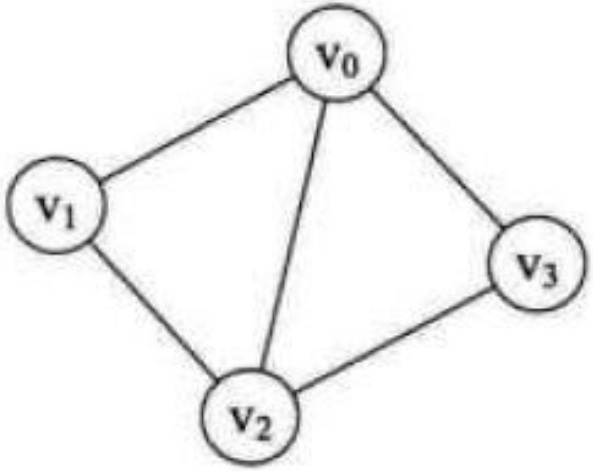
把表示同一条边的2个结点合起来，这样对于需要对边操作的算法比较简单了。
ilink：和i相关的下一条边
jlink：和j相关的下一条边

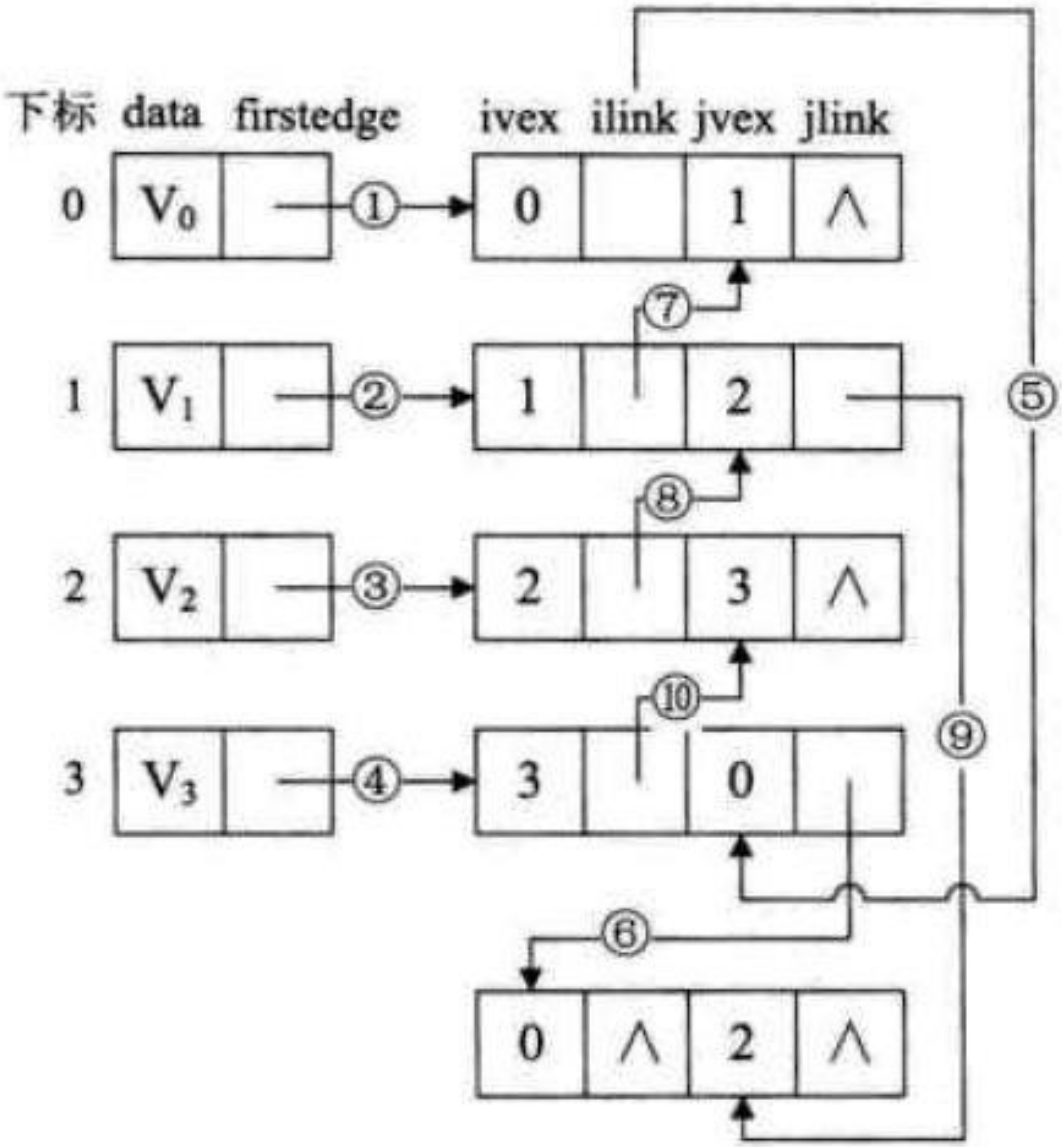
顶点的结构表示

```
typedef struct VexBox {
   VertexType  data;
   EBox  *firstedge; // 指向第一条依附该顶点的边
} VexBox;
```

无向图的结构表示

```
typedef struct {  // 邻接多重表
    VexBox  adjmulist[MAX_VERTEX_NUM];
    int   vexnum, edgenum;
} AMLGraph;
```





无向图的邻接多重表对边的处理比邻接矩阵方便，
邻接矩阵表示无向图时候，对边的处理需要对称处理2次

7.3 图的遍历

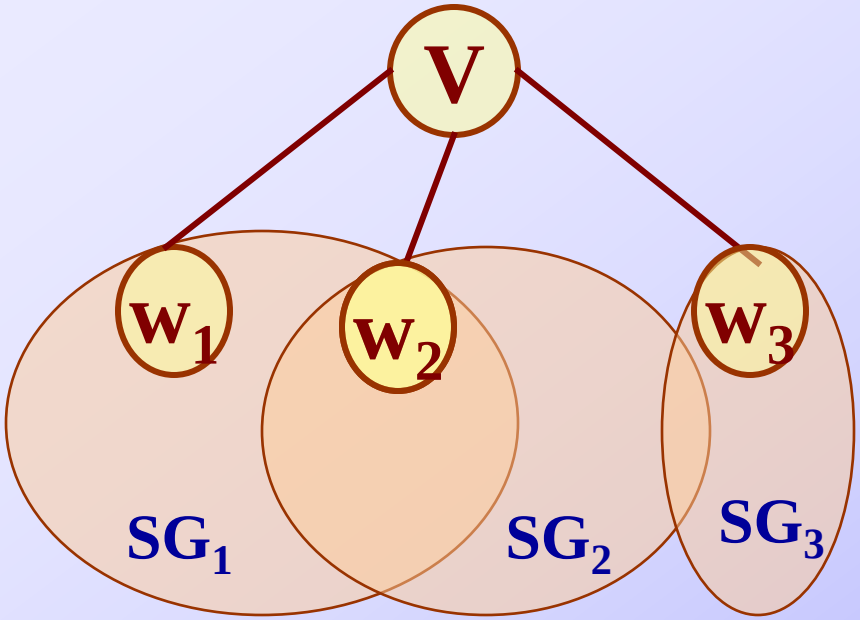
从图中某个顶点出发游历图，访遍图中其余顶点，并且使图中的每个顶点仅被访问一次的过程。

- 深度优先搜索
- 广度优先搜索
- 遍历应用举例

一、深度优先搜索遍历图

连通图的深度优先搜索遍历

从图中某个顶点V_0出发，访问此顶点，然后**依次从V_0的各个未被访问的邻接点出发深度优先搜索遍历图**，直至图中所有和V_0有路径相通的顶点都被访问到。



W_1、W_2和W_3 均为 V 的邻接点，SG_1、SG_2 和 SG_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未被访问，

则从它出发进行深度优先搜索遍历。

从上页的图解可见：

1. 从深度优先搜索遍历连通图的过程类似于树的先根遍历；

2. 如何判别V的邻接点是否被访问？

解决的办法是：为每个顶点设立一个 “访问标志 visited[w]”。

```
void DFS(Graph G, int v) {
  // 从顶点v出发，深度优先搜索遍历连通图 G
   visited[v] = TRUE;   VisitFunc(v);
   for(w=FirstAdjVex(G, v);
          w!=0; w=NextAdjVex(G,v,w))
      if (!visited[w])  DFS(G, w);
          // 对v的尚未访问的邻接顶点w
          // 递归调用DFS
} // DFS
```

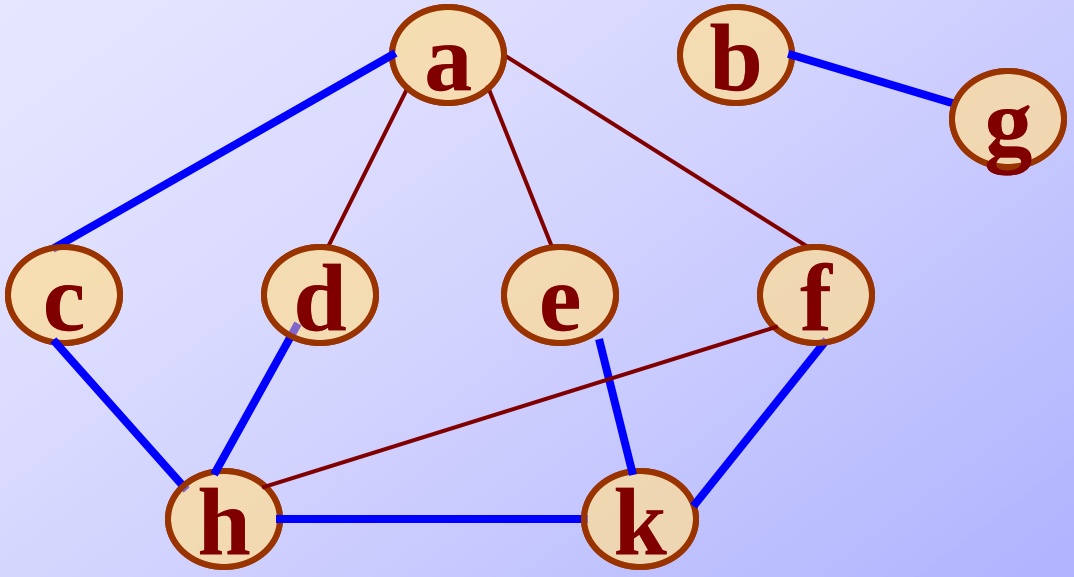
非连通图的深度优先搜索遍历

首先将图中每个顶点的访问标志设为 FALSE，之后搜索图中每个顶点，如果未被访问，则以该顶点为起始点，进行深度优先搜索遍历，否则继续检查下一顶点。

```
void DFSTraverse(Graph G,
                 Status (*Visit)(int v)) {
  // 对图 G 作深度优先遍历。
  VisitFunc = Visit;
  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
}
```

演示只是逻辑演示
每种存储对应一种遍历方式。

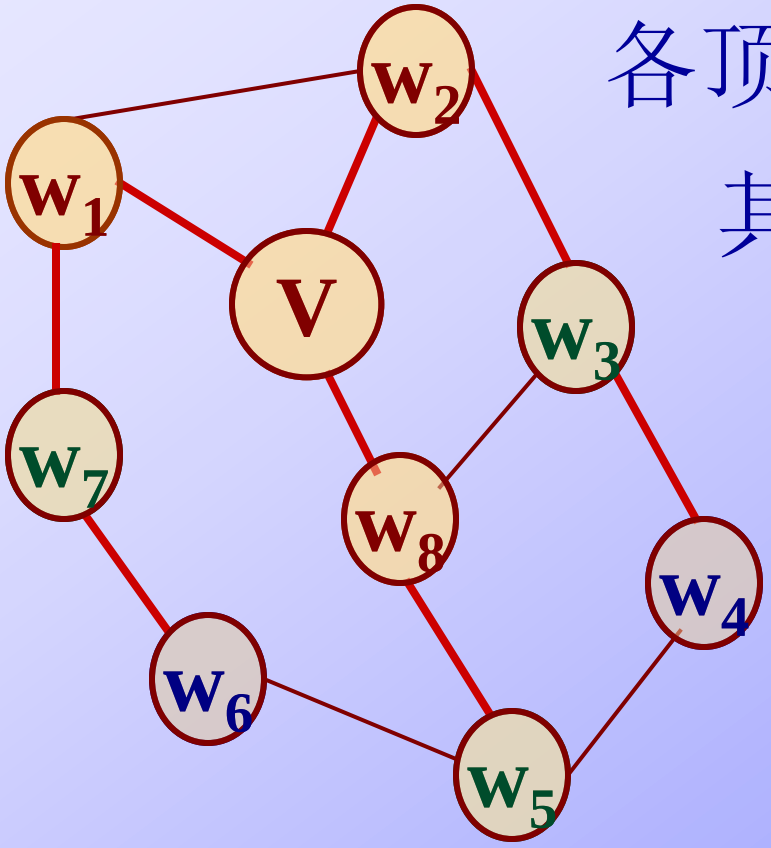
例如:



	0	1	2	3	4	5	6	7	8
访问标志:	T	T	T	T	T	T	T	T	T

访问次序: a c h d k f e b g

二、广度优先搜索遍历图



对连通图，从起始点V到其余各顶点必定存在路径。

其中，$V->w_1$, $V->w_2$, $V->w_8$ 的路径长度为1；

$V->w_7$, $V->w_3$, $V->w_5$ 的路径长度为2；

$V->w_6$, $V->w_4$ 的路径长度为3。

从图中的某个顶点V_0出发，并在访问此顶点之后依次访问V_0的所有**未被访问**过的邻接点，之后**按这些顶点被访问的先后次序依次访问它们的邻接点**，直至图中所有和V_0有路径相通的顶点都被访问到。

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，重复上述过程，直至图中所有顶点都被访问到为止。

```
void BFSTraverse(Graph G,
                 Status (*Visit)(int v)){
  for (v=0; v<G.vexnum; ++v)
     visited[v] = FALSE;  //初始化访问标志
  InitQueue(Q);        // 置空的辅助队列Q
  for ( v=0;  v<G.vexnum;  ++v )
    if ( !visited[v]) {          // v 尚未访问
       … …
   }
} // BFSTraverse
```

```
visited[v] = TRUE;  Visit(v);    // 访问u
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])  {
            visited[w] = TRUE;  Visit(w);    // 访问w
            EnQueue(Q, w); // 未访问的顶点w入队
      } // if
} // while
```

三、遍历应用举例

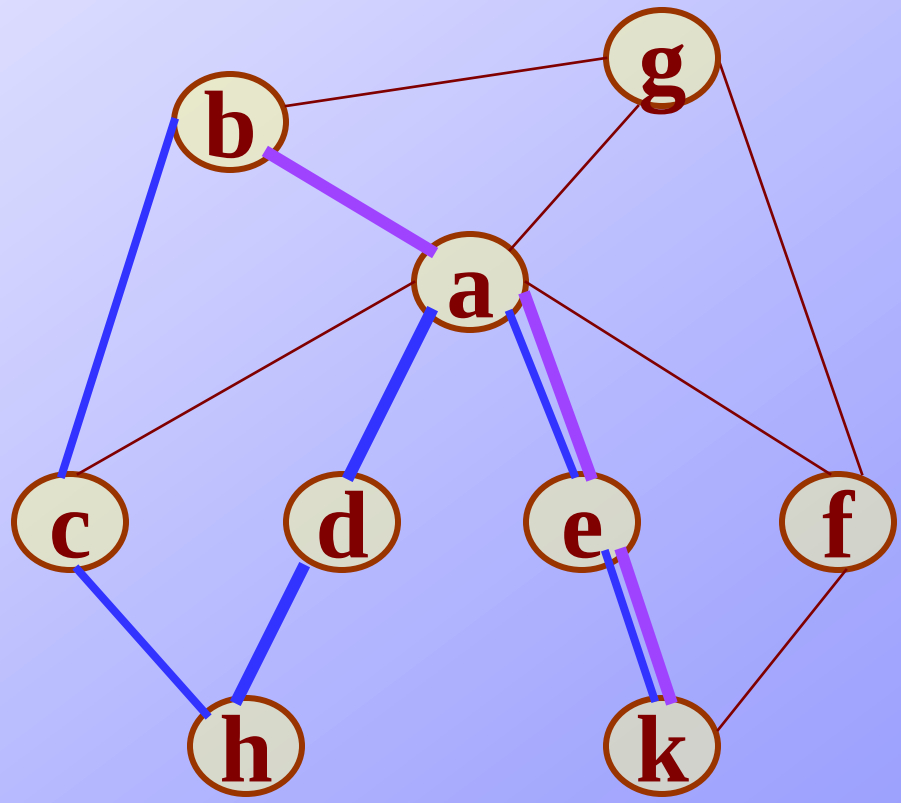
1. 求一条从顶点 i 到顶点 s 的简单路径

2. 求两个顶点之间的一条路径长度最短的路径

1. 求一条从顶点 i 到顶点 s 的简单路径

例如：求从顶点 b 到顶点 k 的一条简单路径。

从顶点 b 出发进行深度优先搜索遍历。



假设找到的第一个邻接点是c，则得到的结点访问序列为：

b c h d a e **k** f g,

假设找到的第一个邻接点是a，则得到的结点访问序列为：**b** a d h c e **k** f g。

结论:

1. 从顶点 i 到顶点 s，若存在路径，则从顶点 i 出发进行深度优先搜索，必能搜索到顶点 s。

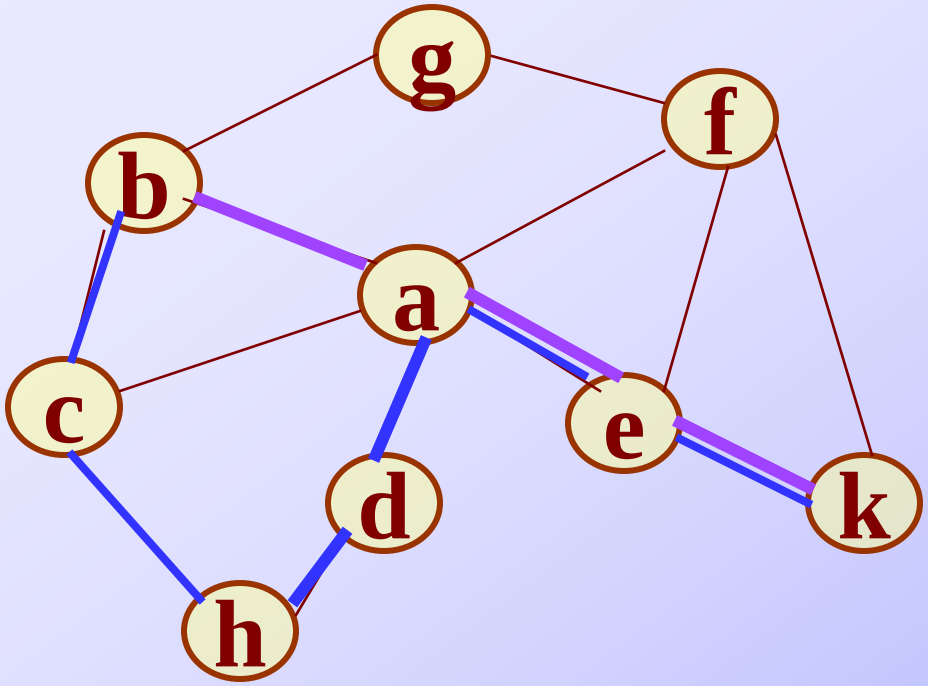
2. 遍历过程中搜索到的顶点不一定是路径上的顶点。

3. 由它出发进行的深度优先遍历已经完成的顶点不是路径上的顶点。

```
void DFSearch( int v, int s, char *PATH) {
 // 从第v个顶点出发递归地深度优先遍历图G，
 // 求得一条从v到s的简单路径，并记录在PATH中
  visited[v] = TRUE;  // 访问第 v 个顶点
  Append(PATH, getVertex(v));  // 第v个顶点加入路径
  for (w=FirstAdjVex(v);  w!=0&&!found;
                          w=NextAdjVex(v) )
     if (w=s) { found = TRUE;  Append(PATH, w); }
     else  if (!visited[w])  DFSearch(w, s, PATH);
  if (!found)  Delete (PATH); // 从路径上删除顶点 v
}
```

2. 求两个顶点之间的一条路径长度最短的路径

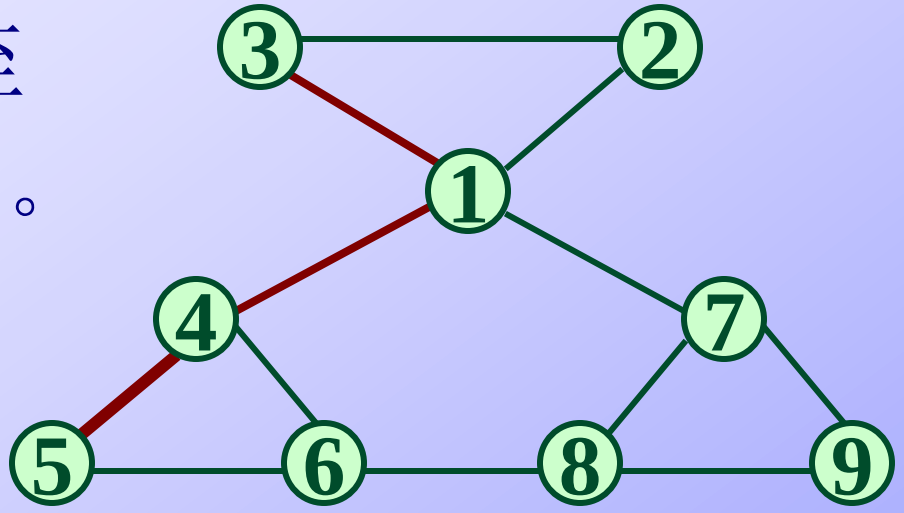
若两个顶点之间存在多条路径，则其中必有一条路径长度最短的路径。如何求得这条路径?



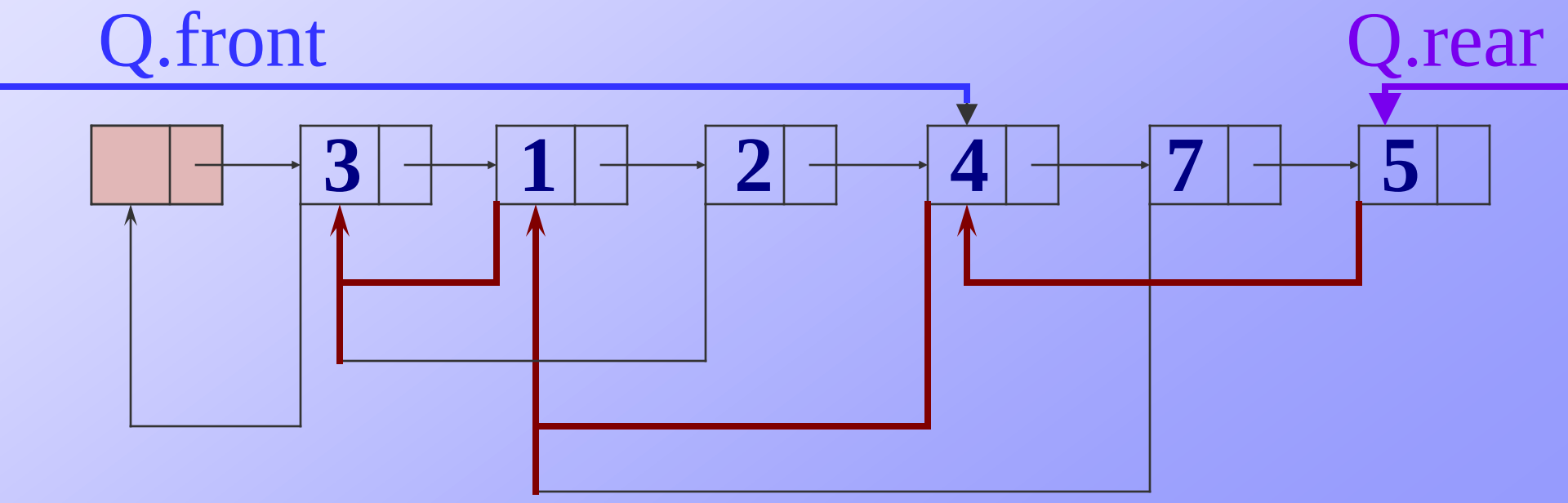
深度优先搜索访问顶点的**次序**取决于图的**存储结构**，而广度优先搜索访问顶点的次序是按“路径长度”渐增的次序。

因此，**求路径长度最短的路径可以基于广度优先搜索遍历进行**，但需要**修改链队列的结点结构及其入队列和出队列的算法**。

例如:求下图中顶点 3 至顶点 5 的一条最短路径。



链队列的状态如下所示:



1) **将链队列的结点改为“双链”结点**。即结点中包含next 和priou两个指针；

2) **修改入队列的操作**。插入新的队尾结点时，令其priou域的指针指向刚刚出队列的结点，即当前的队头指针所指结点；

3) **修改出队列的操作**。出队列时，仅移动队头指针，而不将队头结点从链表中删除。

```
typedef  DuLinkList QueuePtr;
void InitQueue(LinkQueue& Q) {
  Q.front=Q.rear=(QueuePtr)malloc(sizeof(QNode));
  Q.front->next = Q.rear->next = NULL
}
void EnQueue( LinkQueue& Q, QelemType e ) {
  p = (QueuePtr) malloc (sizeof(QNode));
  p->data = e;  p->next = NULL;
  p->priou = Q.front
  Q.rear->next = p;  Q.rear = p;
}
void DeQueue( LinkQueue& Q, QelemType& e ) {
  Q.front = Q.front->next;  e = Q.front->data
}
```

7.4 (连通网的)最小生成树

问题：

假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 $n-1$ 条线路，**如何在最节省经费的前提下建立这个通讯网？**

该问题等价于：

构造网的一棵最小生成树，即：

在 e 条带权的边中选取 n-1 条边（不构成回路），使“**权值之和**”为最小。

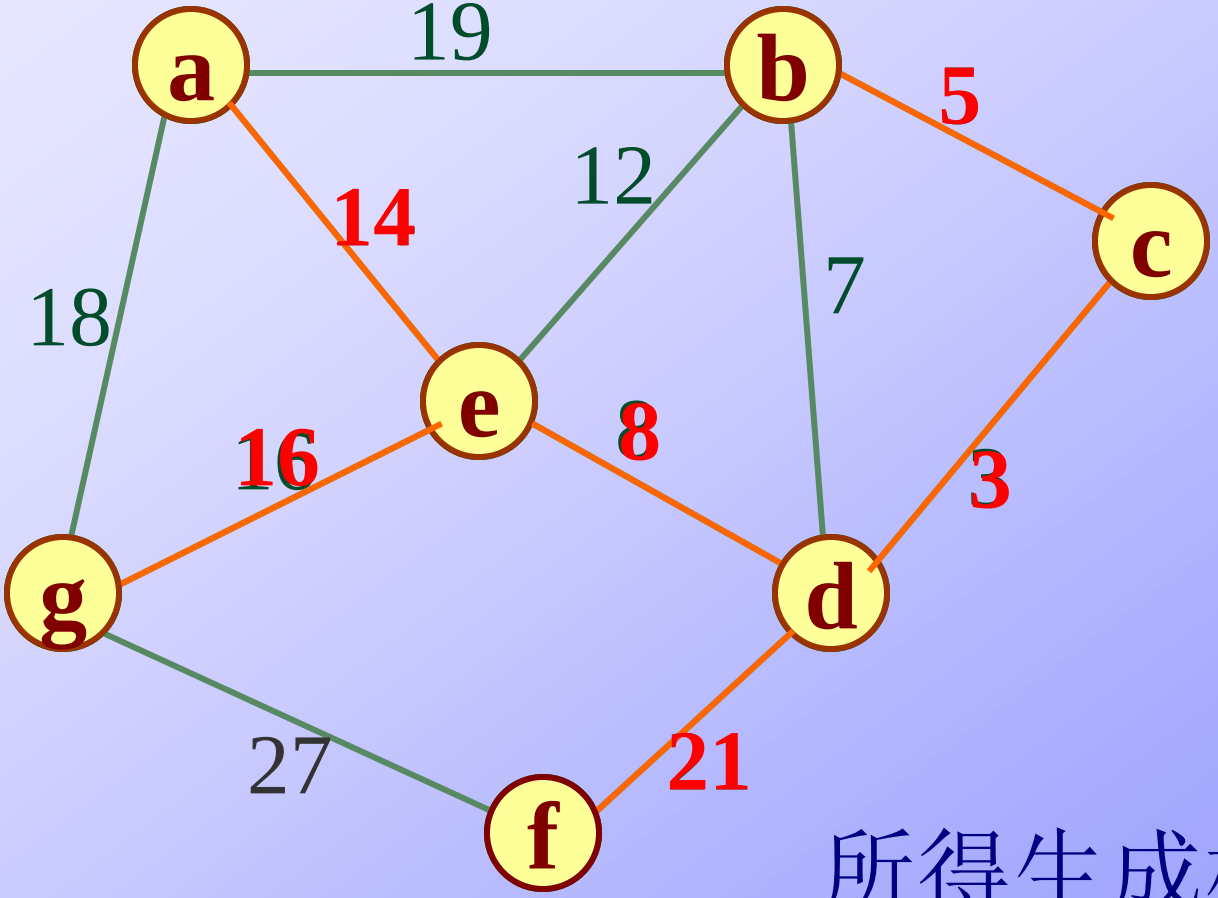
算法一：（普里姆算法）

算法二：（克鲁斯卡尔算法）

普里姆算法的基本思想：

取图中任意一个顶点 v 作为生成树的根，之后往生成树上添加新的顶点 w。**在添加的顶点 w 和已经在生成树上的顶点v 之间必定存在一条边，并且该边的权值在所有连通顶点 v 和 w 之间的边中取值最小。**之后继续往生成树上添加顶点，直至生成树上含有 n-1 个顶点为止。

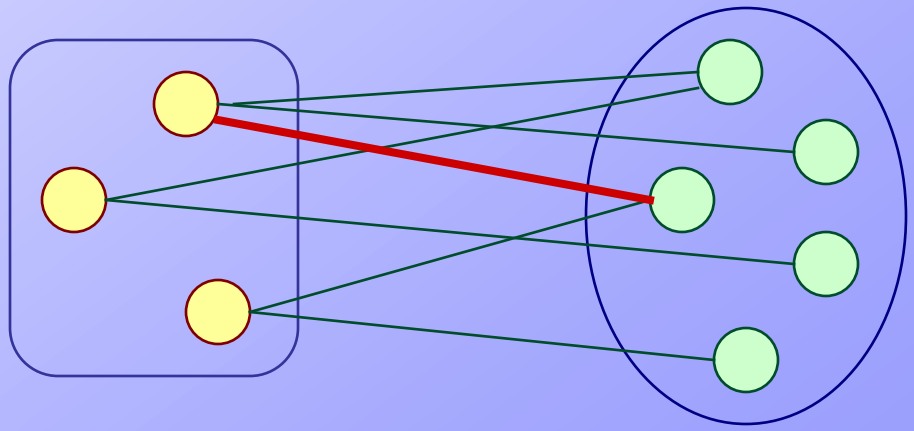
例如:



所得生成树权值和

= 14+8+3+5+16+21 = 67

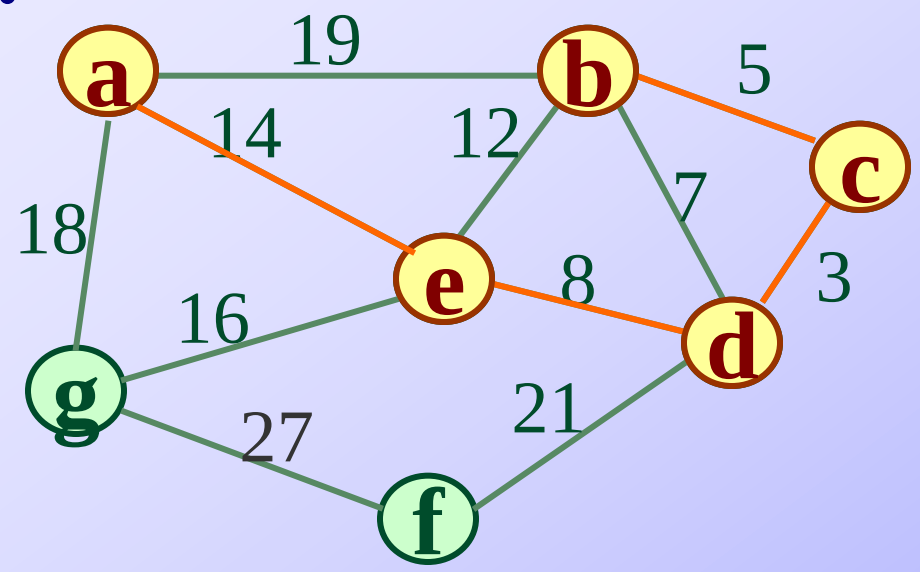
一般情况下所添加的顶点应满足下列条件：在生成树的构造过程中，图中 n 个顶点分属两个集合：已落在生成树上的顶点集 **U** 和尚未落在生成树上的顶点集 **V-U**，则应**在所有连通U中顶点和V-U中顶点的边中选取权值最小的边。**



设置一个辅助数组，对当前V－U集中的每个顶点，记录和顶点集U中顶点相连接的代价最小的边：

```
struct {
    VertexType  adjvex;  // U集中的顶点值
    VRType      lowcost;  // 边的权值
} closedge[MAX_VERTEX_NUM];
```

例如:



closedge	0	1	2	3	4	5	6
Adjvex		c	d	e	a	d	e
Lowcost		**5**	**3**	**8**	**14**	21	16

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

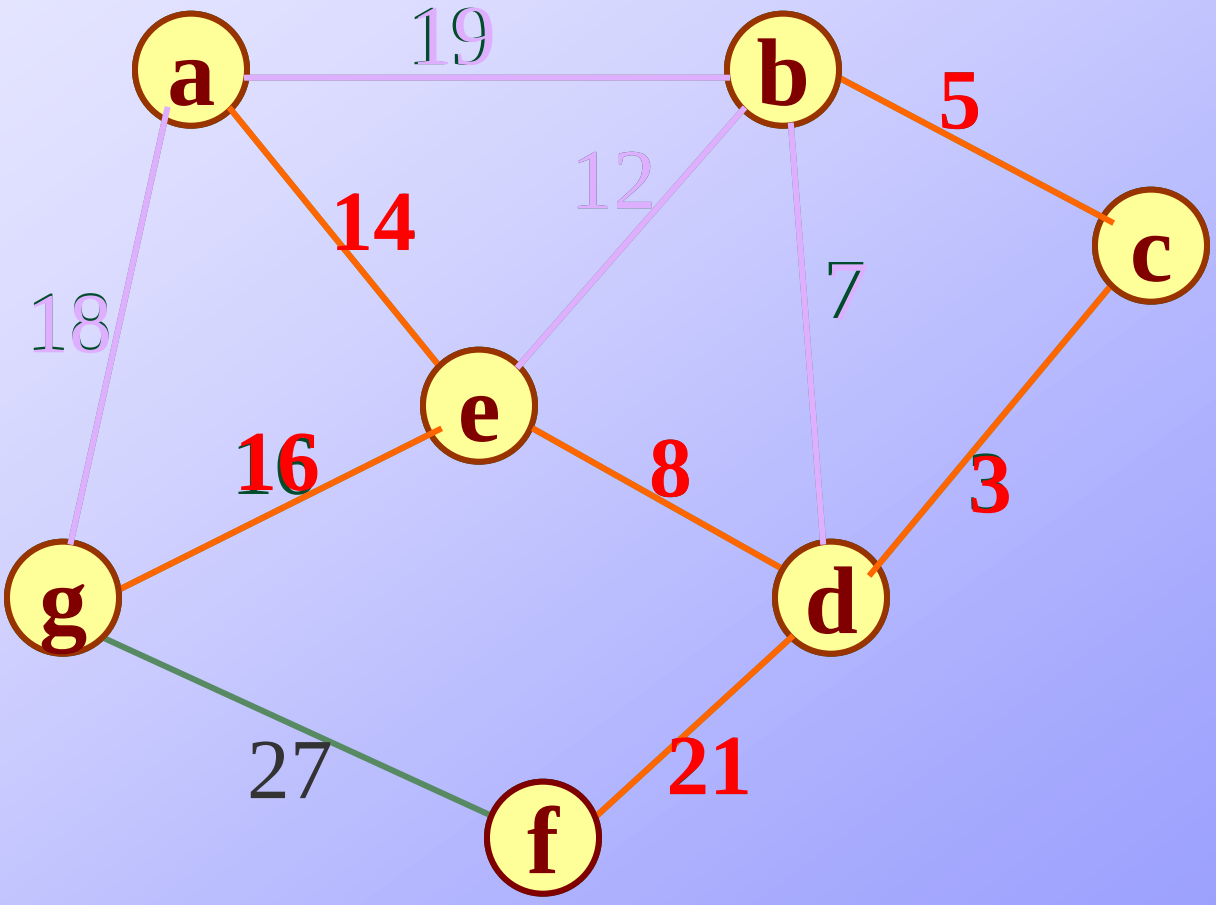
```
k = minimum(closedge);
                    // 求出加入生成树的下一个顶点(k)
printf(closedge[k].adjvex, G.vexs[k]);
                    // 输出生成树上一条边
closedge[k].lowcost = 0;    // 第k顶点并入U集
for (j=0; j<G.vexnum; ++j)
      //修改其它顶点的最小边
  if (G.arcs[k][j].adj < closedge[j].lowcost)
    closedge[j] = { G.vexs[k],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

克鲁斯卡尔算法的基本思想：

- **考虑问题的出发点：**为使生成树上边的权值之和达到最小，则应使生成树中每一条边的权值尽可能地小。

- **具体做法：**先构造一个只含 n 个顶点的子图SG，然后从权值最小的边开始，若它的添加不使SG 中产生回路，则在 SG 上加上这条边，如此重复，直至加上 n-1 条边为止。

例如:



算法描述:

构造非连通图 ST=(V,{ });

k = i = 0;　// k 计选中的边数

while (k<n-1) **{**

++i;

检查边集 E 中**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　输出边(u,v);　且　k++;

}

比较两种算法

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

7.5 重（双）连通图和关节点

问题：

若从一个连通图中删去任何一个顶点及其相关联的边，它仍为一个连通图的话，则该连通图被称为重（双）连通图。

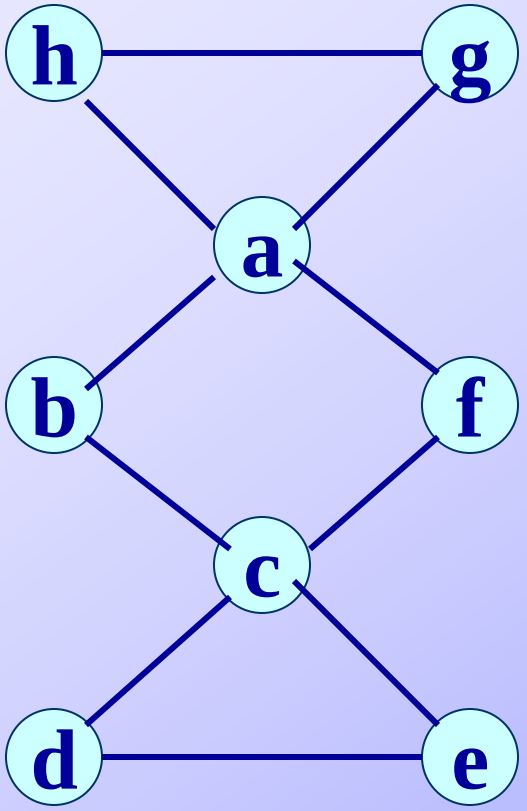
如何判别给定的连通图是否是双连通图？

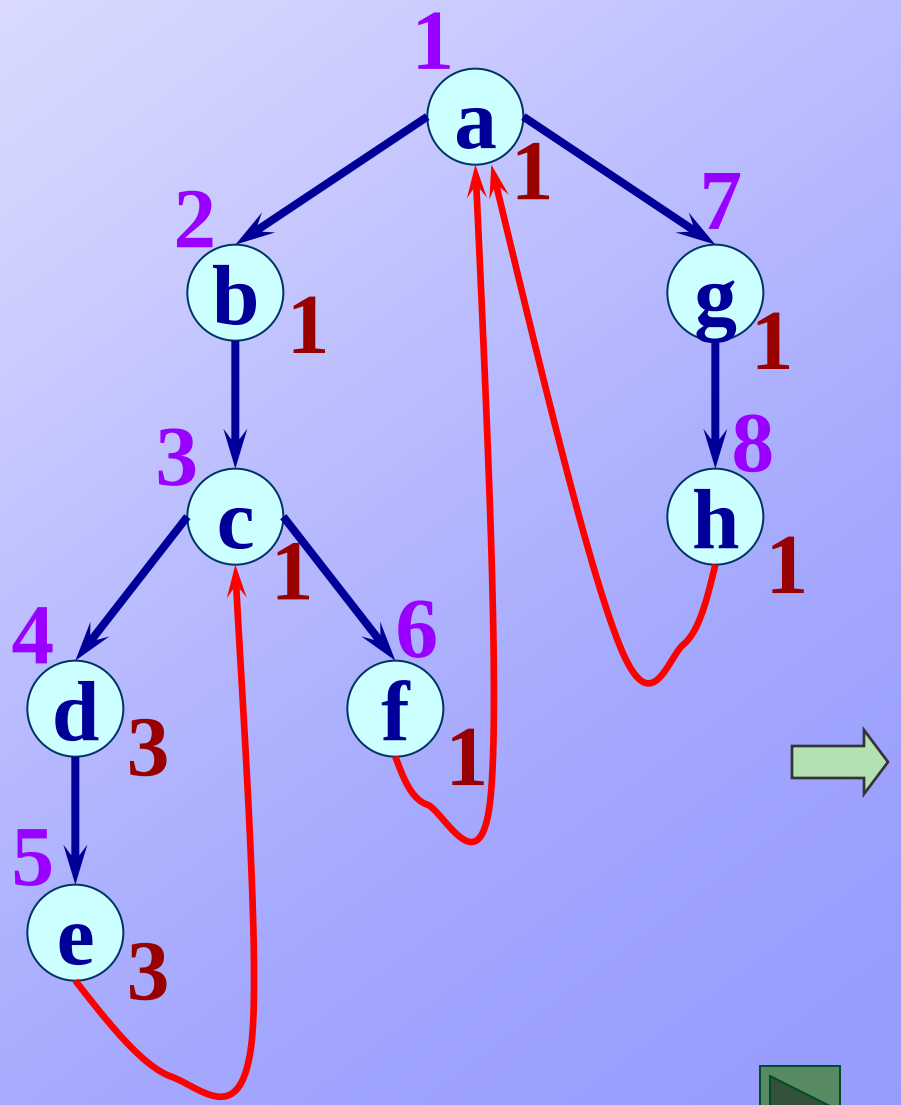
若连通图中的某个顶点和其相关联的边被删去之后，该连通图被分割成两个或两个以上的连通分量，则称此顶点为**关节点**。

没有关节点的连通图为双连通图。

例如:下列连通图中，

深度优先生成树





顶点 a 和顶点 c 是关节点

关节点有何特征？

需借助图的深度优先生成树来分析。

假设从某个顶点V_0出发对连通图进行深度优先搜索遍历，则可得到一棵**深度优先生成树**，树上包含图的所有顶点。

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

§ 对生成树上的任意一个“顶点”，若其某棵子树的根或子树中的其它“顶点”没有和其祖先相通的回边，则该“顶点”必为关节点。

对上述两个判定准则在算法中如何实现？

1) 设V_0为深度优先遍历的出发点

```
p = G.vertices[0].firstarc;   v = p->adjvex;
DFSArticul(G, v);
    // 从第v顶点出发深度优先搜索
if (count < G.vexnum) {
    // 生成树的根有至少两棵子树
   printf (0, G.vertices[0].data);
    // 根是关节点
```

2) 定义函数:

low(v) = Min{visited[v], low[w], visited[k] }

其中: 顶点**w** 是生成树上顶点**v** 的**孩子**;

顶点**k** 是生成树上和顶点**v**有**回边**相通的**祖先**;

visited记录深度优先遍历时的访问次序

若对顶点v，在生成树上**存在**一个**子树根w，**

且 low[w] ≥ visited[v]

则顶点v为关节点。

对深度优先遍历算法作如下修改：

1．visited[v]的值改为遍历过程中顶点的访问次序count值；

2．遍历过程中求得
low[v]=**Min**{visited[v],low[w],visited[k]}

3．从子树遍历返回时，
判别low[w]≥visited[v]?

```
void  DFSArticul(ALGraph G, int v0) {
  // 从第v0个顶点出发深度优先遍历图 G,
  // 查找并输出关节点
  min =visited[v0] = ++count;
   // v0是第count个访问的顶点, 并设low[v0]的初值为min
   for(p=G.vertices[v0].firstarc;   p;
                                  p=p->nextarc) {
     // 检查v0的每个邻接点
   }
   low[v0] = min;
} // DFSArticul
```

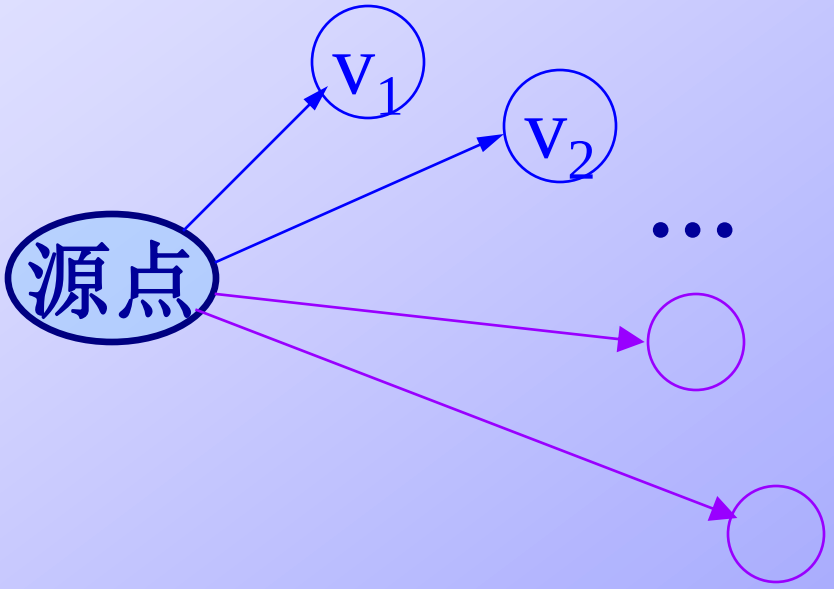
```
w = p->adjvex;       // w为v0的邻接顶点
if (visited[w] == 0) {       // w未曾被访问
  DFSArticul(G, w);  // 返回前求得low[w]
  if (low[w] < min)   min = low[w];
  if (low[w]>=visited[v0])
    printf(v0, G.vertices[v0].data); //输出关节点
}
else     // w是回边上的顶点
    if (visited[w] < min)   min = visited[w];
```

7.6 两点之间的最短路径问题

- 求从某个源点到其余各点的最短路径（单源最短路径）
- 每一对顶点之间的最短路径

求从源点到其余各点的最短路径的算法的基本思想:

依最短路径的长度递增的次序求得各条路径



其中，从源点到顶点v的最短路径是所有最短路径中长度最短者。

- 路径长度最短的**最短路径**的特点：

  在这条路径上，必定只含一条弧，并且这条弧的权值最小。

- 下一条路径长度次短的**最短路径**的特点：

  它只可能有两种情况：或者是直接从源点到该点(只含一条弧)； 或者是从源点经过顶点v_1，再到达该顶点(由两条弧组成)。

- 再下一条路径长度次短的**最短路径**的特点:

  它可能有三种情况：或者是直接从源点到该点(只含一条弧)； 或者是从源点经过顶点v_1，再到达该顶点(由两条弧组成)；或者是从源点经过顶点v_2，再到达该顶点。

- 其余最短路径的特点：

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

求最短路径的迪杰斯特拉算法：

设置辅助数组Dist，其中每个分量Dist[k]表示 当前所求得的从源点到其余各顶点 k 的最短路径。

一般情况下，

Dist[k] = <源点到顶点 k 的弧上的权值>

或者 = <源点到其它顶点的路径长度>

+ <其它顶点到顶点 k 的弧上的权值>。

1）在所有从源点出发的弧中选取一条权值最小的弧，即为第一条最短路径。

$$Dist[k]=\begin{cases} G.arcs[v0][k] & \text{V0和k之间存在弧} \\ INFINITY & \text{V0和k之间不存在弧} \end{cases}$$

其中的最小值即为最短路径的长度。

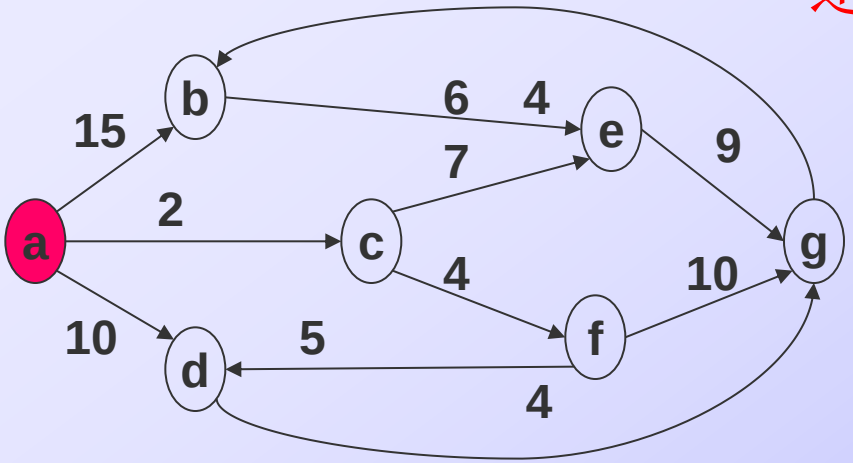
2）修改其它各顶点的*Dist*[*k*]值。

假设求得最短路径的顶点为u，

若 ***Dist*[*u*]+*G.arcs*[*u*][*k*]<*Dist*[*k*]**

则将 *Dist*[*k*] 改为 *Dist*[*u*]+*G.arcs*[*u*][*k*]。

迪杰斯特拉算法模拟



终点	b	c	d	e	f	g
Dist						
Path						

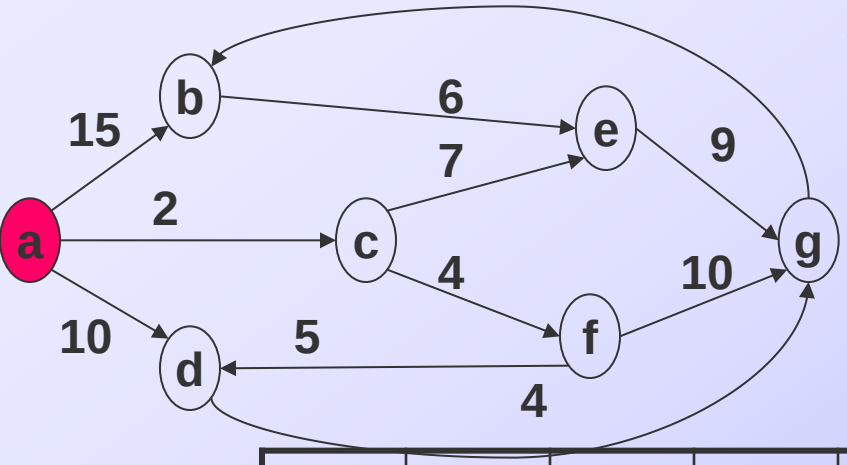
终点	b	c	d	e	f	g
Dist	15	2	10	∞	∞	∞
Path	ab	ac	ad			

从a出发到达b、c、d有边ac最小

终点	b	c	d	e	f	g
Dist	15	2	10	9	6	16
Path	ab	ac	ad	ace	acf	acfg

根据ac最小
修正 ace=9< ∞,acf=6< ∞
选出最小acf
修正 acfg=16<∞ (因为acfd和为11<10 不需修正)

迪杰斯特拉算法模拟



终点	b	c	d	e	f	g
Dist	15	2	10	9	6	16
Path	ab	ac	ad	ace	acf	acfg

选出最小**ace**
修正（无需修正 因为**aceg=18>acfg=16**）

终点	b	c	d	e	f	g
Dist	15	2	10	9	6	14
Path	ab	ac	ad	ace	acf	adg

选出最小**ad**
修正 **adg=14<acfg=16**

终点	b	c	d	e	f	g
Dist	15	2	10	9	6	14
Path	ab	ac	ad	ace	acf	adg

选出最小**adg**
修正 ,没有可修正
选出最小**ab**，也没有可修正的，完毕

求每一对顶点之间的最短路径

弗洛伊德算法的基本思想是：

从 v_i 到 v_j 的所有可能存在的路径中，选出一条长度最短的路径。

若<v_i,v_j>存在，则存在路径{v_i,v_j}

// 路径中不含其它顶点

若<v_i,v_1>,<v_1,v_j>存在，则存在路径{v_i,v_1,v_j}

// 路径中所含顶点序号不大于1

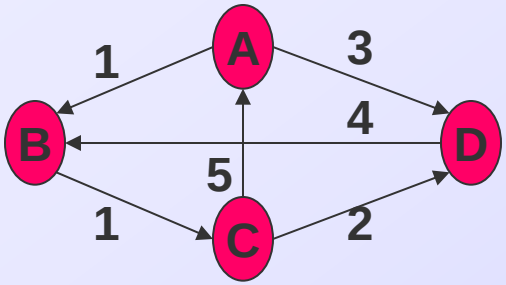
若{v_i,…,v_2}, {v_2,…,v_j}存在，

则存在一条路径{v_i, …, v_2, …v_j}

// 路径中所含顶点序号不大于2

…

依次类推，则 vi 至 vj 的最短路径应是上述这些路径中，路径长度最小者。



弗洛伊德举例

	A	B	C	D
A	0	1		3
B		0	1	
C	5		0	2
D		4		0

路径矩阵

	AB		AD
		BC	
CA			CD
	DB		

路径长度矩阵和路径矩阵初始值为当前存在的弧，自己到自己的路径为0

之后在v和w直间加u（u从A，B，C，D）条件如下：

If(D[v][u]+D[u][w])<D[v][w])

D[v][w]=D[v][u]+D[u][w]

	A	B	C	D
A	0	1		3
B		0	1	
C	5		0	2
D		4		0

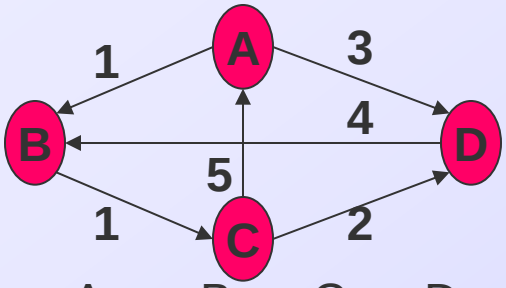
路径矩阵

	AB		AD
		BC	
CA			CD
	DB		

加A

对于第一行是从A出发的AB,AC,AD,加上AAB，AAC，AAD是一样的所以不检查了

对于第二行，本来B到A没有路径，现在还是一样的，我们看B到C，如果BA，AC有路径，但是没有路径，所以就不用考虑这一行了



弗洛伊德举例

	A	B	C	D
A	0	1		3
B		0	1	
C	5	6	0	2
D		4		0

路径矩阵

	AB		AD
		BC	
CA	CAB		CD
	DB		

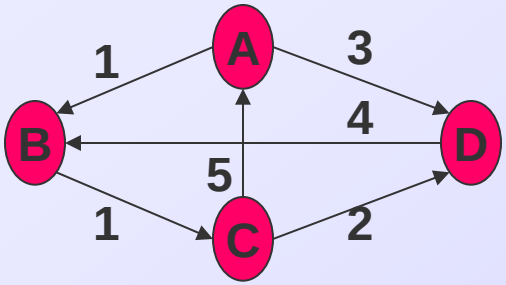
加A，
对于第三行，CA有路径为5,原来C到B是没有弧的，现在CA=5，AB=1，所以存在路径CAB，长度是6 C到D本有一条弧是2，现在CA+AD=8>2所以不修正
对于第四行，因为DA没有路径，所以也不需要改变
加B
对于第一行，AC原来没有路径，现在AB，BC 都通所以AC路径长度为2，路径为ABC；AD路径长度是3，BD没有路径，所以就不需改变
对于第二行不变
对于第三、四行采用上面同样的方法，最终结果如左

	A	B	C	D
A	0	1	2	3
B		0	1	
C	5	6	0	2
D		4	5	0

路径矩阵

	AB	ABC	AD
		BC	
CA	CAB		CD
	DB	DBC	

之后在v和w直接加u（u从A，B，C，D）条件如下：
If(D[v][u]+D[u][w])<D[v][w])　　D[v][w]=D[v][u]+D[u][w]



弗洛伊德举例

	A	B	C	D
A	0	1	2	3
B	6	0	1	3
C	5	6	0	2
D	10	4	5	0

路径矩阵

	AB	ABC	AD
BCA		BC	BCD
CA	CAB		CD
DBCA	DB	DBC	

加C，

第一行：AC=2，CB=6 和大于AB=1，所以不修正AB。

AC=2，CD=2大于目前AD=3，也需不要修正

第二行：BC=1，CA=5，所以BA=6路径为BCA

BC=1，CD=2，所以BD=3，路径为BCD

第三行：不考虑

第四行：DC=5，CA=5，所以DA=10，路径为DCA

DC=5，B=6>DB=4，所以不修正

加D，

如左，略方法同上，公式都采用最下面的。

	A	B	C	D
A	0	1	2	3
B	6	0	1	3
C	5	6	0	2
D	10	4	5	0

路径矩阵

	AB	ABC	AD
BCA		BC	BCD
CA	CAB		CD
DBCA	DB	DBC	

之后在v和w直接加u（u从A，B，C，D）条件如下：

If(D[v][u]+D[u][w])<D[v][w])　　D[v][w]=D[v][u]+D[u][w]

7.7 拓扑排序

问题:

假设以有向图表示一个工程的施工图或程序的数据流图，则图中不允许出现回路。

检查有向图中是否存在回路的方法之一，是对有向图进行**拓扑排序**。

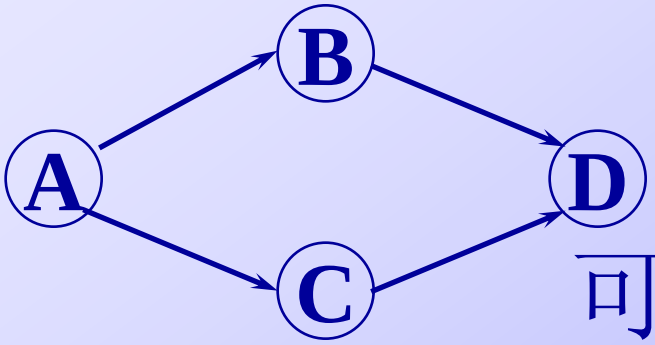
何谓“拓扑排序”？

对有向图进行如下操作：

按照有向图给出的次序关系，将图中顶点排成一个线性序列，对于有向图中没有限定次序关系的顶点，则可以人为加上任意的次序关系。

由此所得顶点的线性序列称之为拓扑有序序列

例如：对于下列有向图

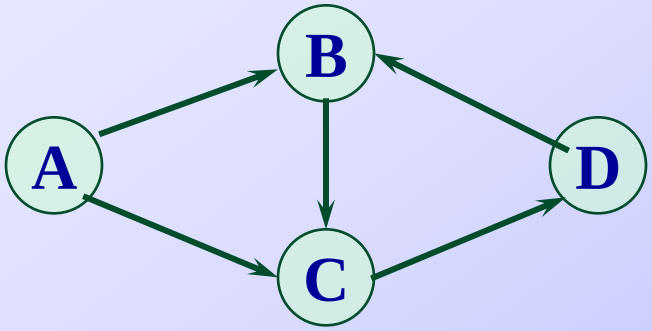


可求得拓扑有序序列：

A B C D　或　A C B D

A领先于BCD
BC领先于D
BC之间在有向图中没有约束所以BC CB都可以，所以得到两个拓扑有序序列。
能够得到拓扑序列则说明该图中没有回路

反之，对于下列有向图



不能求得它的拓扑有序序列。

因为图中存在一个回路 {B, C, D}

A领先于BCD
但BCD互相之间是一个领先的关系，所以没办法写出B在前，C在前还是D在前
对这个有向图进行排序，得不出有序序列，那么就说明存在了一个回路
所以可以通过拓扑排序判断有向图是不是有回路

如何进行拓扑排序？

一、从有向图中选取一个**没有前驱**的顶点，并输出之；

二、从有向图中**删去此顶点以及所有以它为尾的弧**；

重复上述两步，直至图空，或者图不空但找不到无前驱的顶点为止。

a b h c d g f e

在算法中需要用定量的描述替代定性的概念

没有前驱的顶点 == 入度为零的顶点

删除顶点及以它为尾的弧 == 弧头顶点的入度减1

算法描述

```
取入度为零的顶点v;
while (v<>0) {//v<>0 说明取到了一个入度为0的顶点v
    printf(v);  ++m;//m为取到的顶点个数
    w=FirstAdj(v);
    while (w<>0) {
      inDegree[w]--;
      w=nextAdj(v,w);
    }
    取下一个入度为零的顶点v;
}
if m<n  printf("图中有回路" );
```

算法关键：如何确定入度为0的顶点?

//m与实际个数n相比,m<n 就没取够，表示有回路，否则没有回路

为避免每次都要搜索入度为零的顶点，在算法中设置一个“**栈**”，以保存“入度为零”的顶点。

```
CountInDegree(G,indegree);
    //对各顶点求入度
InitStack(S);
for ( i=0; i<G.vexnum; ++i)
  if (!indegree[i])  Push(S, i);
    //入度为零的顶点入栈
```

开始对已经建好存储结构的图，对它的各个顶点求入度。
然后，把所有入度为0 的顶点入栈
这样，每次去栈里头取就行了

```
count=0;          //对输出顶点计数
while (!EmptyStack(S)) {
  Pop(S, v);  ++count;  printf(v);
  for (w=FirstAdj(v); w;  w=NextAdj(G,v,w)){
    --indegree(w);  // 弧头顶点的入度减一
    if (!indegree[w])  Push(S, w);
        //新产生的入度为零的顶点入栈
  }
}
```

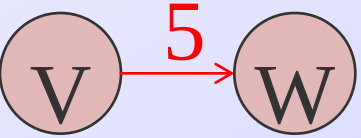
/*每次去栈里头取的时候，要把所有它的邻接点的入度-1
如果对于邻接点的入度-1后，变成0度了，我们也把其入栈。
因此整个算法进行到栈空为止。
到栈空的时候看看整个输出的顶点个数够不够，够的话就没回路，拓扑排序成功，否则有回路*/

```
if (count<G.vexnum) printf("图中有回路" )
```

7.8 关键路径

拓扑的施工流图 表示的是施工顺序，拓扑不考虑期限

现在进一步，我们要讨论的是整个工程的完成期限是多少，就是效益问题，完成工程期间，有没有某个工序是比较关键的---关键工程，关键工程的拖期，提前将影响整个工程



弧<v,w>代表活动或子工程，权值代表工期，

顶点表示事件，即为一个活动的开始或结束事件

问题：

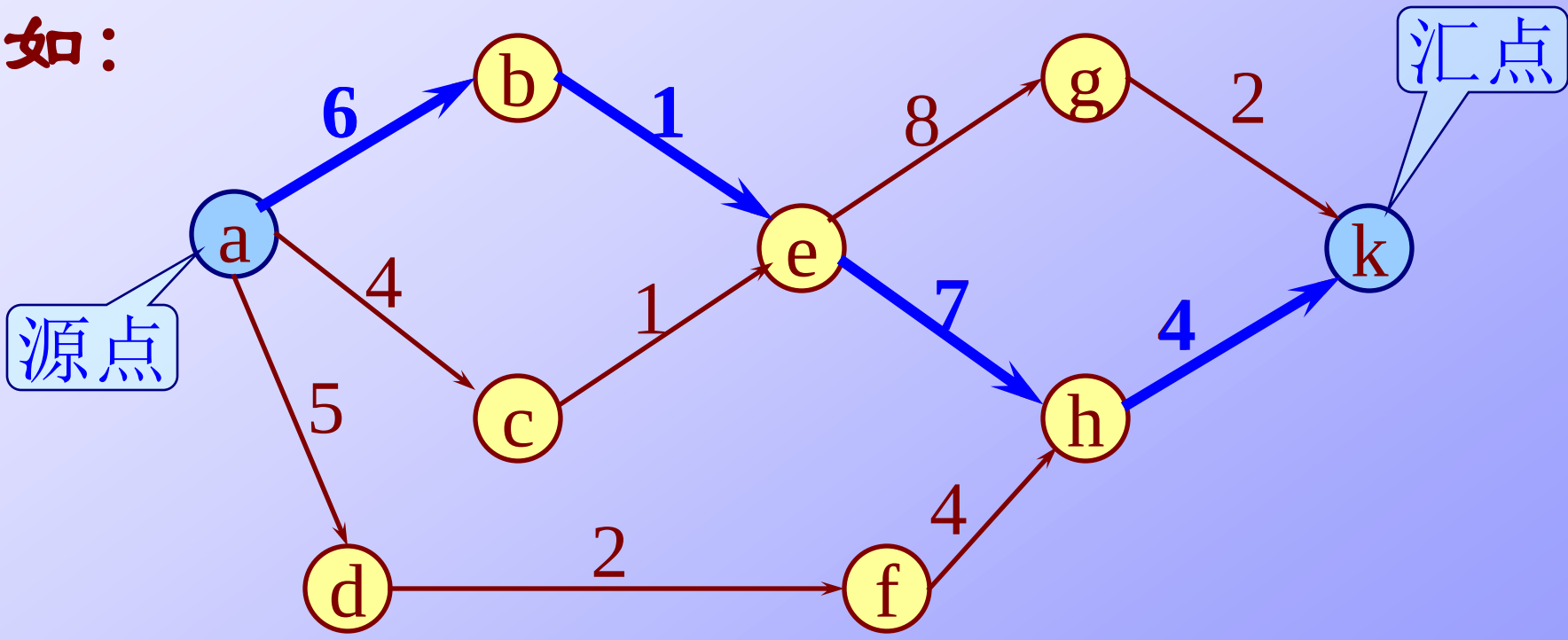
假设以有向网表示一个施工流图，弧上的权值表示完成该项子工程所需时间。

问：哪些子工程项是“关键工程”？

即：哪些子工程项将影响整个工程的完成期限的。

整个工程完成的时间为：从有向图的**源点**到**汇点**的最长路径。

例如：



“关键活动”指的是：该弧上的**权值增加**将使有向图上的**最长路径的长度增加。**

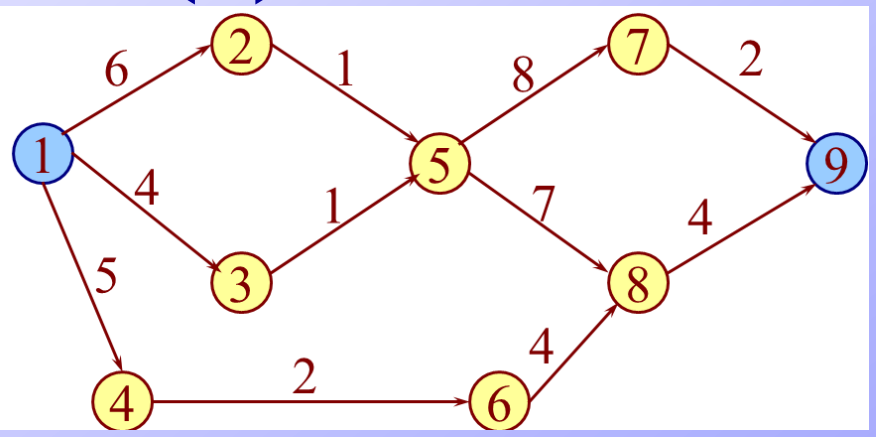
如何求关键活动？

“事件(顶点)” 的 最早发生时间 *ve*(j)

ve(j) = 从源点到顶点j的最长路径长度；

“事件(顶点)” 的 最迟发生时间 *vl*(k)

vl(k) = 从顶点k到汇点的最短路径长度。



	1	2	3	4	5	6	7	8	9
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18

Ve(j):为了保证后序进行的发生时间,用拓扑的正序来求
Vl(k):为了保证完工的最晚开始时间,用拓扑的逆序来求

假设第 i 条弧为 <j, k>

则 对第 i 项活动言

“活动(弧)”的 最早开始时间 *ee*(i)

ee(i) = ve(j);

“活动(弧)”的 最迟开始时间 *el*(i)

el(i) = vl(k) – dut(<j,k>);

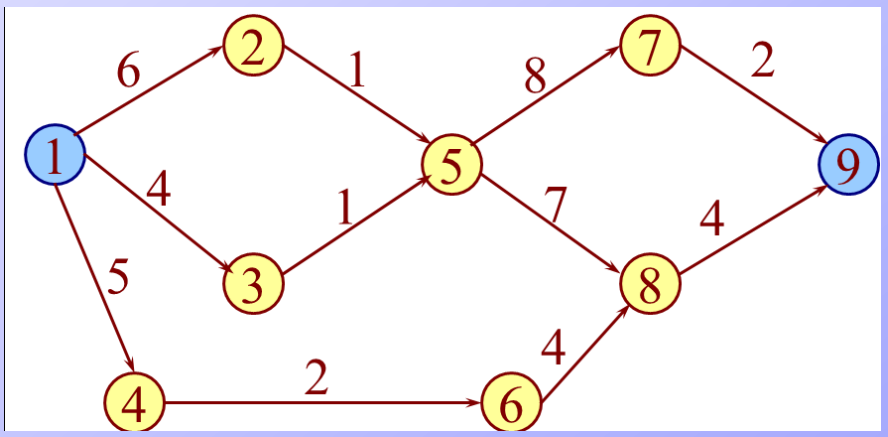
事件发生时间的计算公式：

ve(源点) = 0；

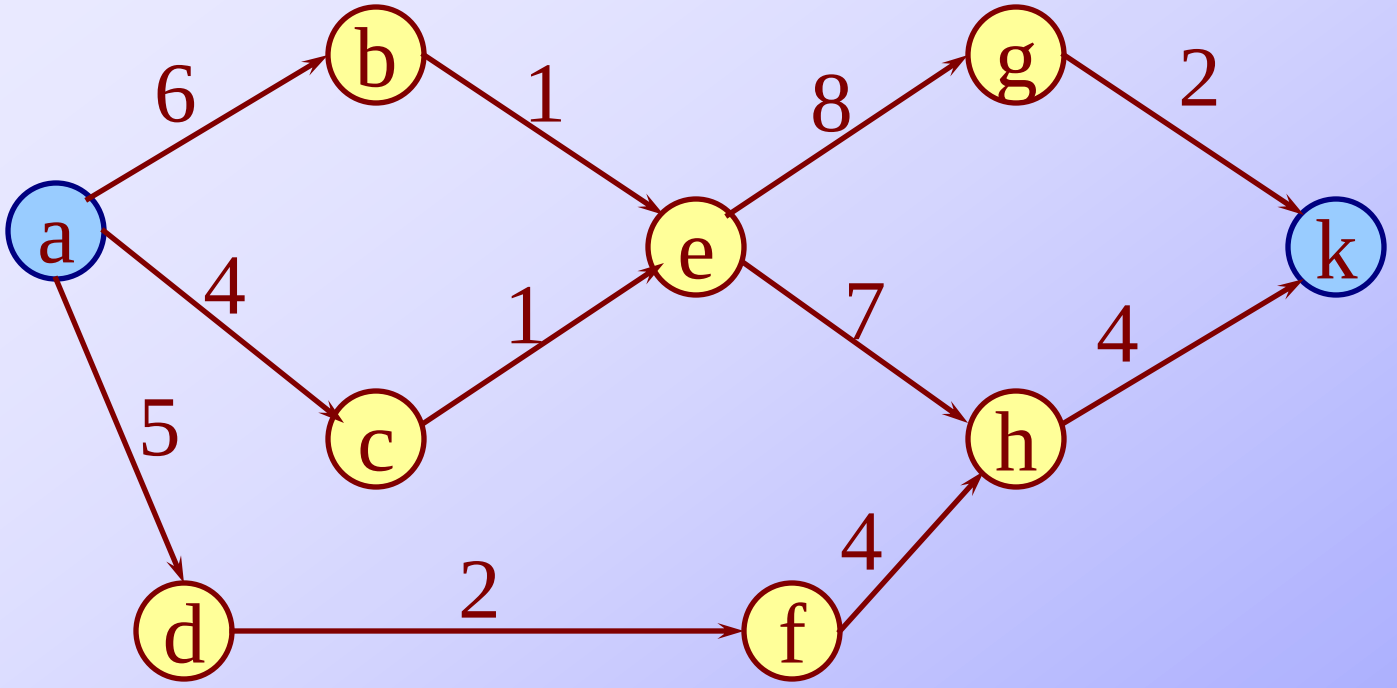
ve(k) = Max{ve(j) + dut(<j, k>)}

vl(汇点) = ve(汇点)；

vl(j) = Min{vl(k) – dut(<j, k>)}



	1	2	3	4	5	6	7	8	9
ve	0	6	4	5	7	7	15	14	18
vl	0	6	6	8	7	10	16	14	18



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

拓扑有序序列：a - d - f - c - b - e - h - g - k

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

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

算法的实现要点:

显然，求**ve**的顺序应该是按**拓扑有序**的次序；

而　求**vl**的顺序应该是按**拓扑逆序**的次序；

因为　拓扑逆序序列即为拓扑有序序列的**逆序列**，

因此　应该在拓扑排序的过程中，

另设一个“**栈**”记下拓扑有序序列。

1. 熟悉图的各种存储结构及其构造算法，了解实际问题的求解效率与采用何种存储结构和算法有密切联系。

2. 熟练掌握图的两种搜索路径的遍历：遍历的逻辑定义、深度优先搜索和广度优先搜索的算法。

在学习中应注意图的遍历算法与树的遍历算法之间的类似和差异。

3. 应用图的遍历算法求解各种简单路径问题。

4. 理解教科书中讨论的各种图的算法。