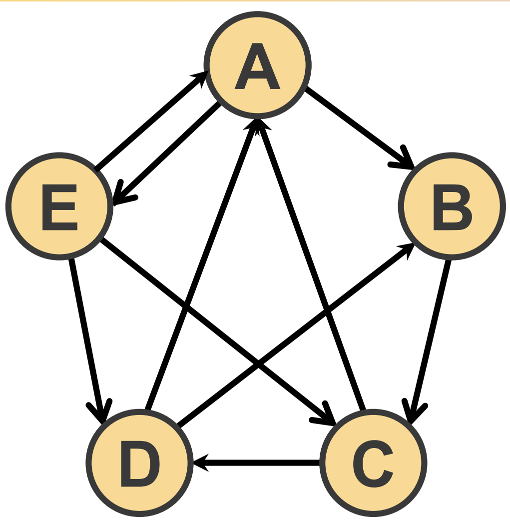
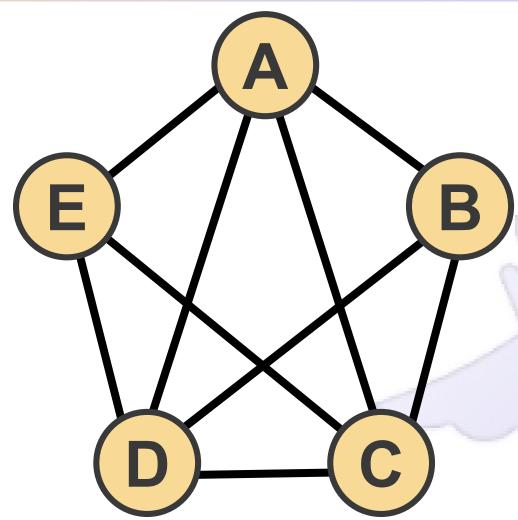
第六章 图



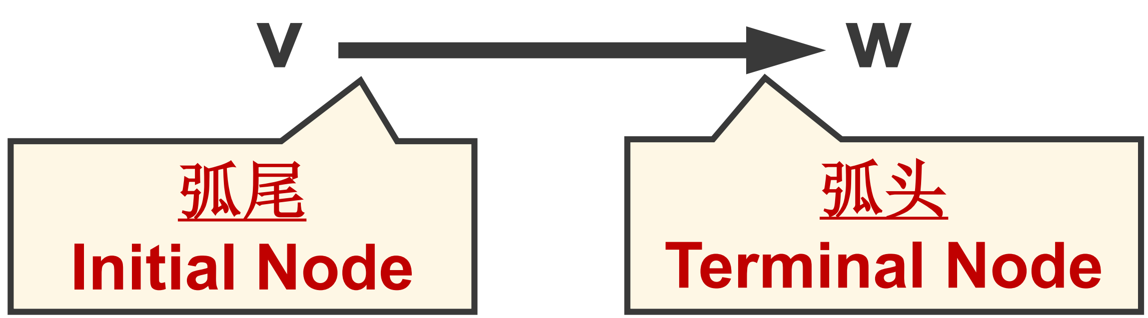


本章内容

- 6.1 抽象数据类型图的定义
- 6.2 图的存储表示
- 6.3 图的遍历
- 6.4 最小生成树
- 6.5 最短路径问题
- 6.6 拓扑排序
- 6.7 关键路径

6.1 抽象数据类型图的定义

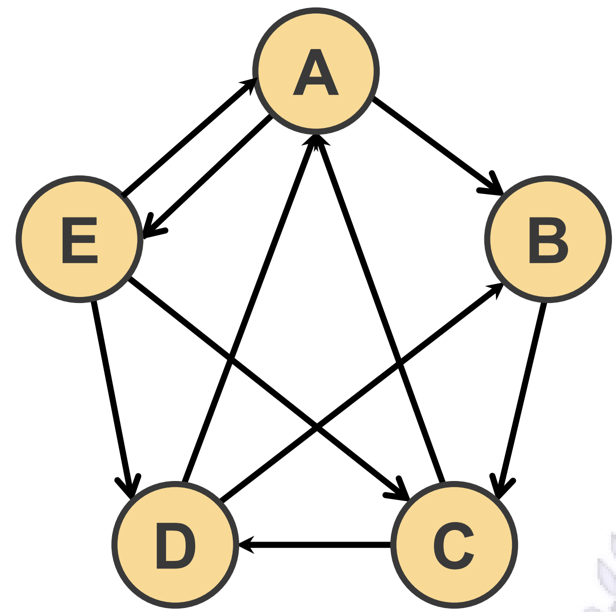
- **6.1.1 图的定义**
- 图(Graph)是由一个顶点(Vertex)集 V 和一个弧(Arc)集 R构成的数据结构。
- Graph = (V , R)
- 其中 R＝{<v, w>| v, w ∈ V 且 P(v, w)}
  - <v, w>表示从 顶点v 到顶点 w 的一条弧，并称 w 为弧头，v 为弧尾。
  - 谓词 P(v, w) 定义了弧 <v, w>的意义或信息



6.1.1 图的定义

- **有向图(Diagraph)**：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：$G_1 = (V_1, R_1)$
- V1={A, B, C, D, E}
- R1={<A,B>, <A,E>, <B,C>, <C,A>, <D,C>, <D,B>, <D,A>, <E,C> , <E,A> }



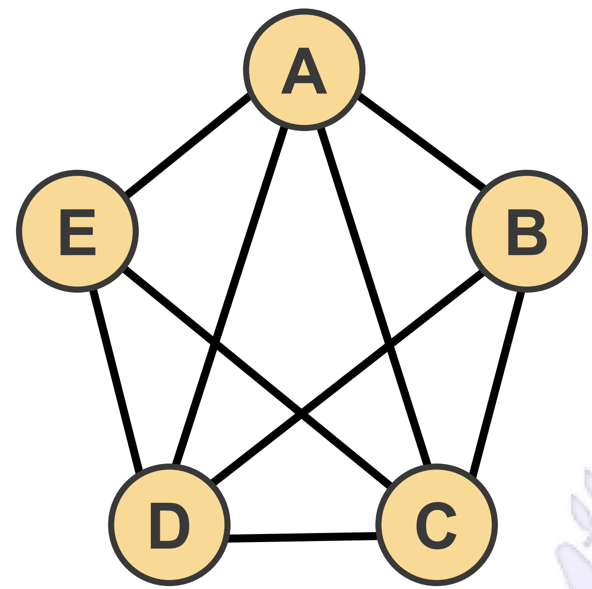
6.1.1 图的定义

- 若$<v, w> \in R$ 必有$<w, v> \in R$, 则称 (v,w) 为顶点v 和顶点 w 之间存在一条 “**边(edge)**”。
- 由顶点集和边集构成的图称作**无向图(Undigraph)**。

•例如： $G_1 = (V_1, R_1)$

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

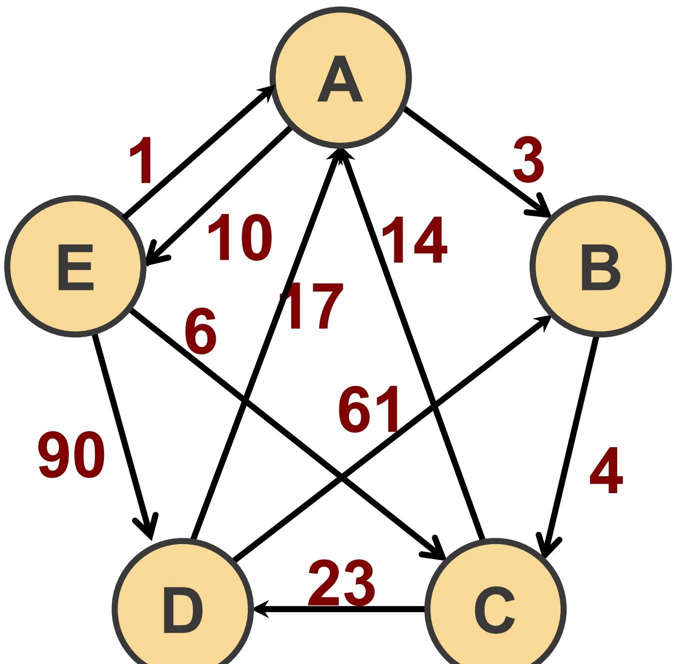
$R_1=\{(A,B), (A,E), (A,C), (B,C), (D,C), (D,B), (D,E), (D,A), (E,C)\}$

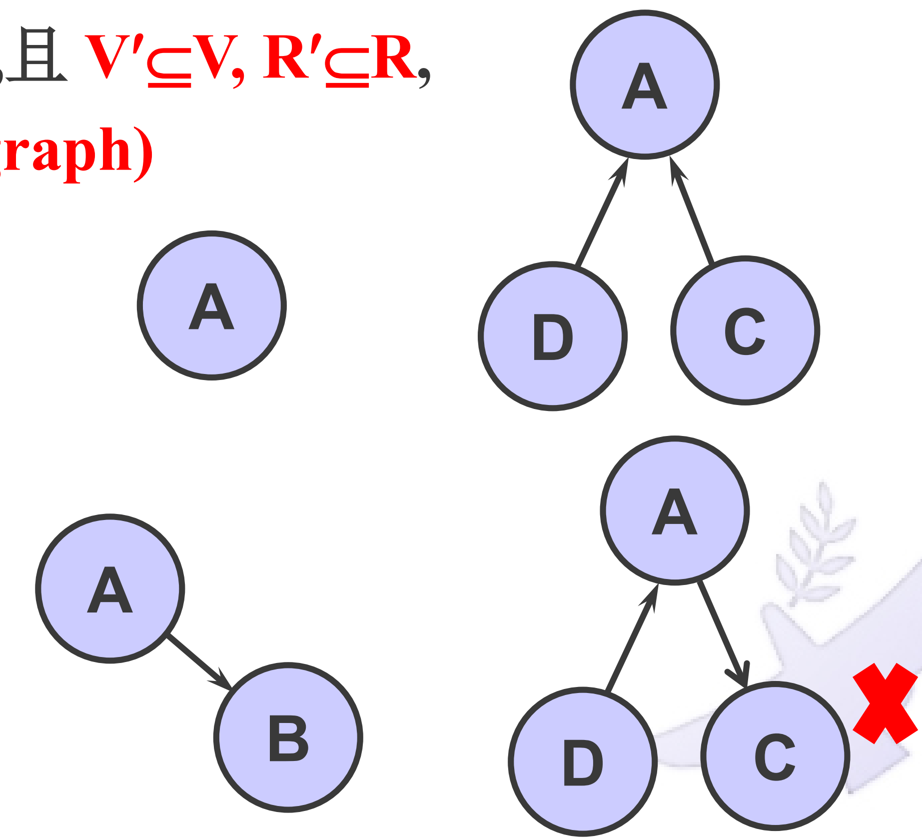


说明：本课程中不考虑环和平行边

6.1.2 图的有关术语

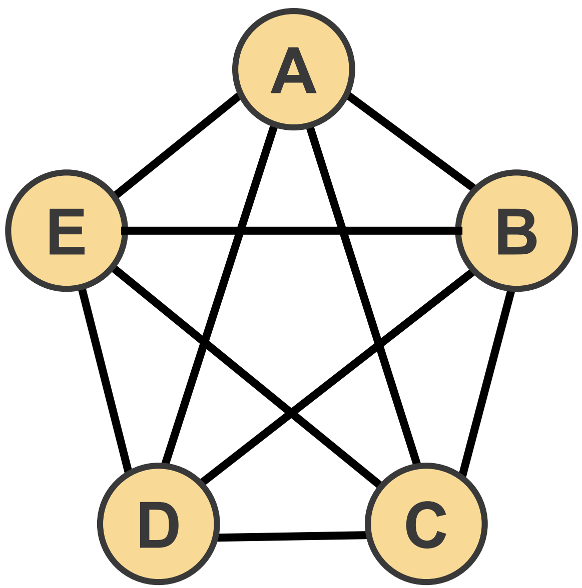
- 1)网(Network)
  - 弧或边带权(Weight)的图分别称有向网和无向网
- 2)子图
  - •G=(V , R) ， G′=(V ′, R ′),且 V′⊆V, R′⊆R,
  - •则称 G′为 G 的子图(Subgraph)





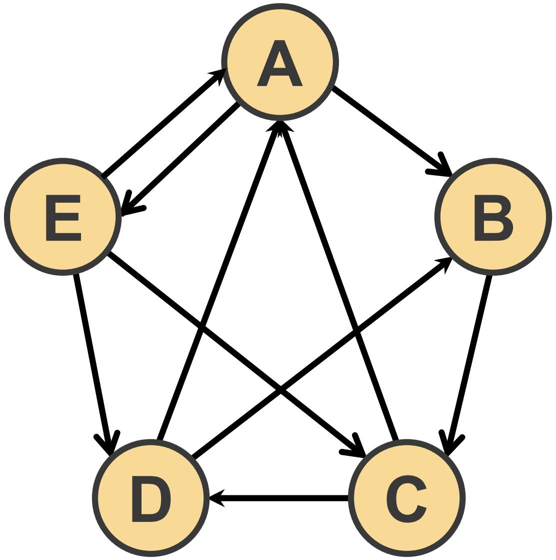
6.1.2 图的有关术语

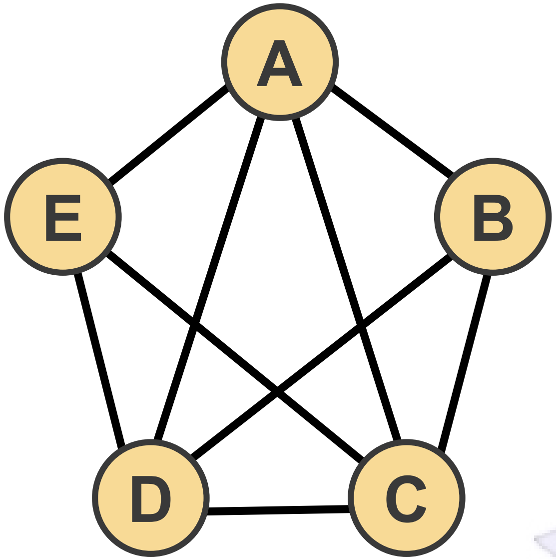
- 假设图中有 n 个顶点，e 条边，则
- 3) 含有e=n(n-1)/2 条边的无向图称作完全图(Completed graph)
- 4) 含有e=n(n-1) 条弧的有向图称作 有向完全图
- 5) 若边或弧的个数 e<nlogn，则称作稀疏图(Sparse graph)，否则称作稠密图(Dense graph).



6.1.2 图的有关术语

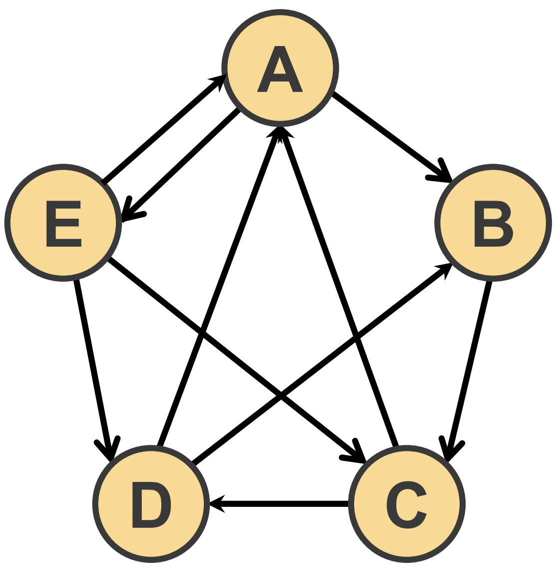
- 6）假若顶点v 和顶点w 之间存在一条边，则称顶点v 和w 互为邻接点。
- 边(v, w) 和顶点v 和w 相关联
- 7）和顶点v 关联的边的数目定义为顶点的度
- 出度、入度

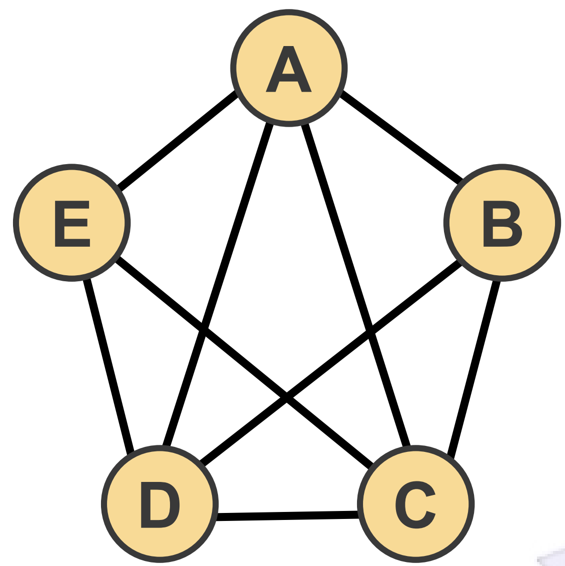




6.1.2 图的有关术语

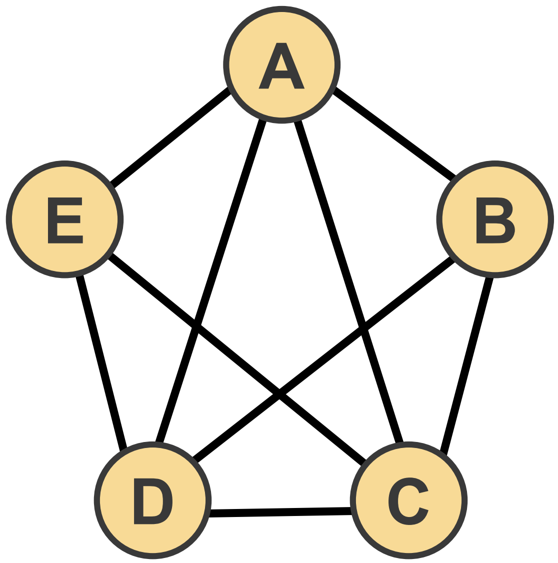
- 8) 从顶点u 到顶点w 之间存在一条路径(Path)，当 $\{ u=v_{i,0},v_{i,1}, \dots, v_{i,m}=w\}$中，$<v_{i,j-1},v_{i,j}>\in R$,$1\le j\le m$
- 路径上边的数目称作路径长度
- 简单路径:序列中顶点不重复出现的路径
- 简单回路:序列中第一个顶点和最后一个顶点相同的简单路径

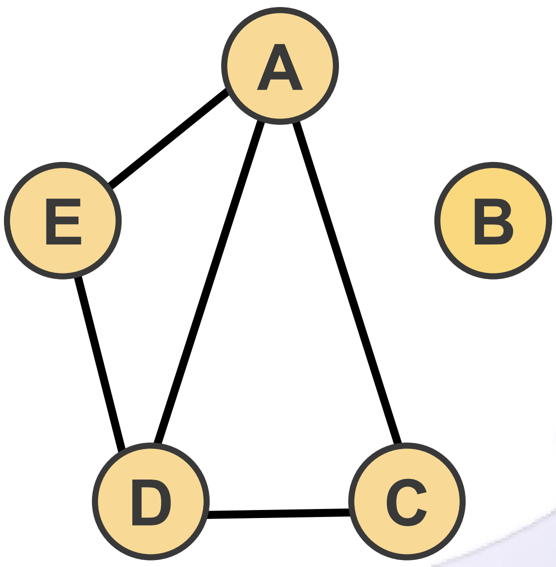




6.1.2 图的有关术语

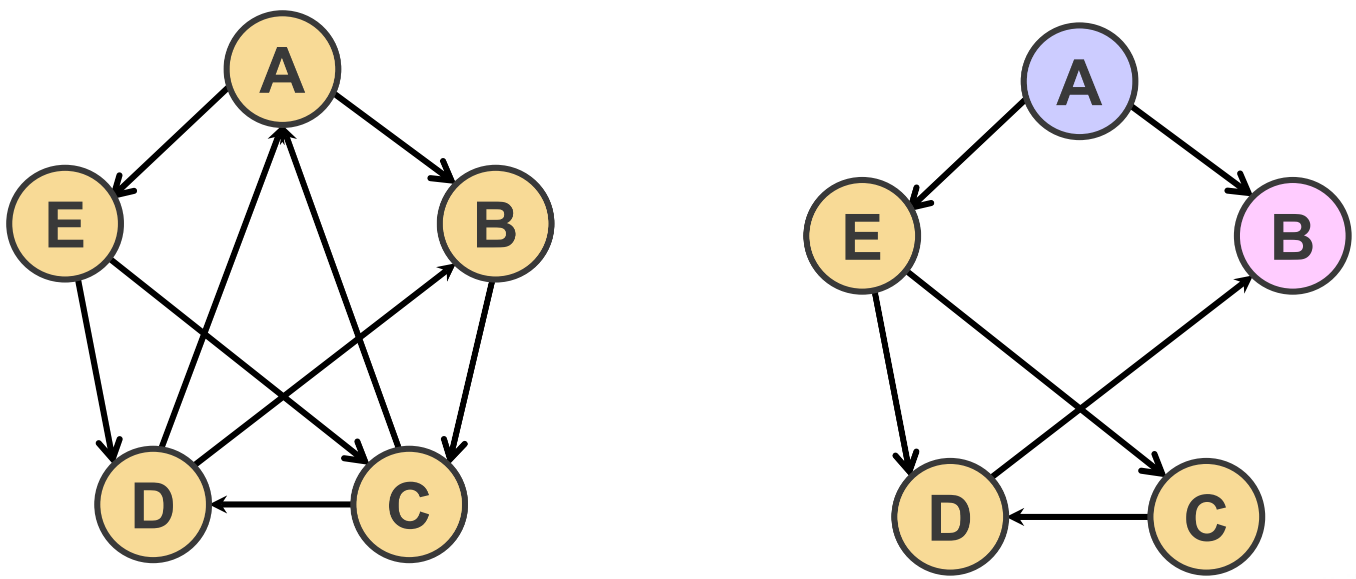
- 9)若图G中任意两个顶点之间都有路径相通，则称此图为**连通图(Connected graph)**；
- 若无向图为非连通图，则图中各个极大连通子图称作此图的**连通分量(Connected component)**；





6.1.2 图的有关术语

- 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，其各个强连通子图称作它的强连通分量



6.1.3 图的基本操作

- 结构的建立和销毁
- 对顶点的访问操作
- 对邻接点的操作
- 插入或删除顶点
- 插入和删除弧
- 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
- **GetVex(G, v);**
  - 返回 v 的值
- **PutVex(&G, v, value);**
  - 对 v 赋值value

对邻接点的操作

- **firstNeighbor(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”
- **nextNeighbor(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中插入弧<v,w>

- **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

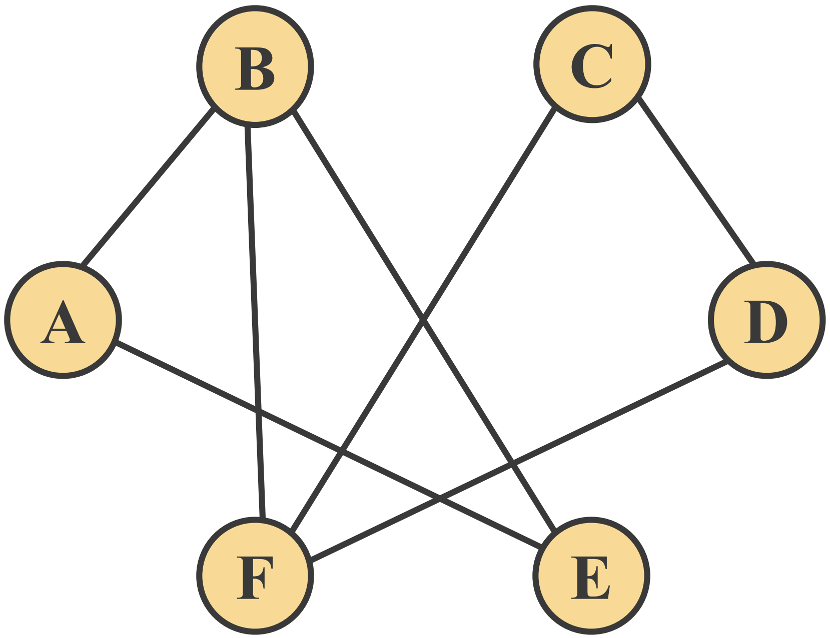
6.2 图的存储表示

- 6.2.1 图的数组(邻接矩阵)存储表示
- 6.2.2 图的邻接表存储表示
- 6.2.3 有向图的十字链表存储表示
- 6.2.4 无向图的邻接多重表存储表示

6.2.1 图的数组(邻接矩阵)存储表示

◆ 定义:矩阵的元素为

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

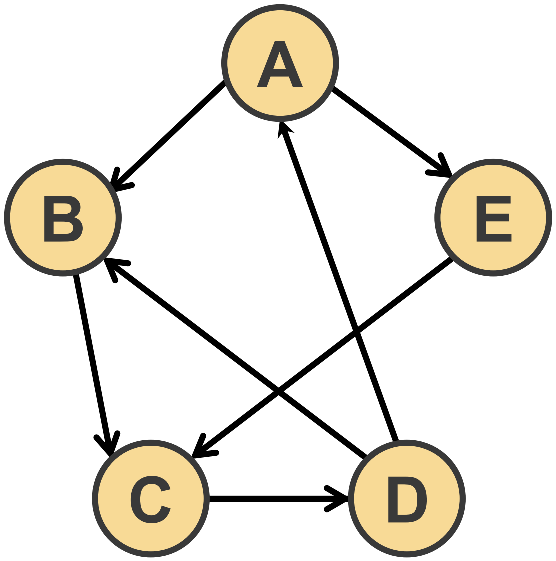


0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	1	0	1
0	0	1	0	0	1
1	1	0	0	0	0
0	1	1	1	0	0

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

顶点i 的度?
第 i 行 (列) 1 的个数。

有向图的邻接矩阵不一定是对称矩阵



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

0	A	
1	B	
2	C	
3	D	
4	E	

- 顶点 i 的出度？ 第 i 行 1 的个数
- 顶点 i 的入度？ 第 i 列 1 的个数。

6.2.1 图的数组(邻接矩阵)存储表示

弧的定义

```
typedef struct ArcCell { // 弧的定义
    VRType  adj;    // VRType是顶点关系。
            // 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]
                    [MAX_VERTEX_NUM];
```

6.2.1 图的数组(邻接矩阵)存储表示

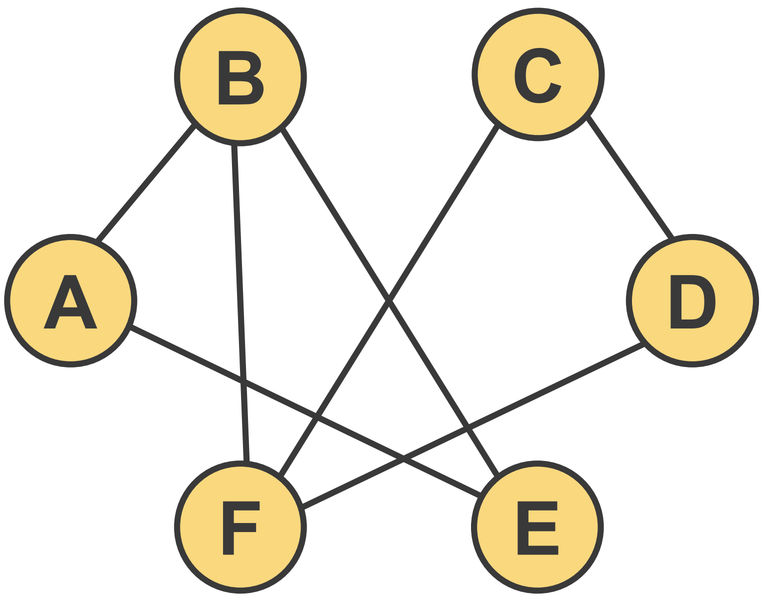
图的定义

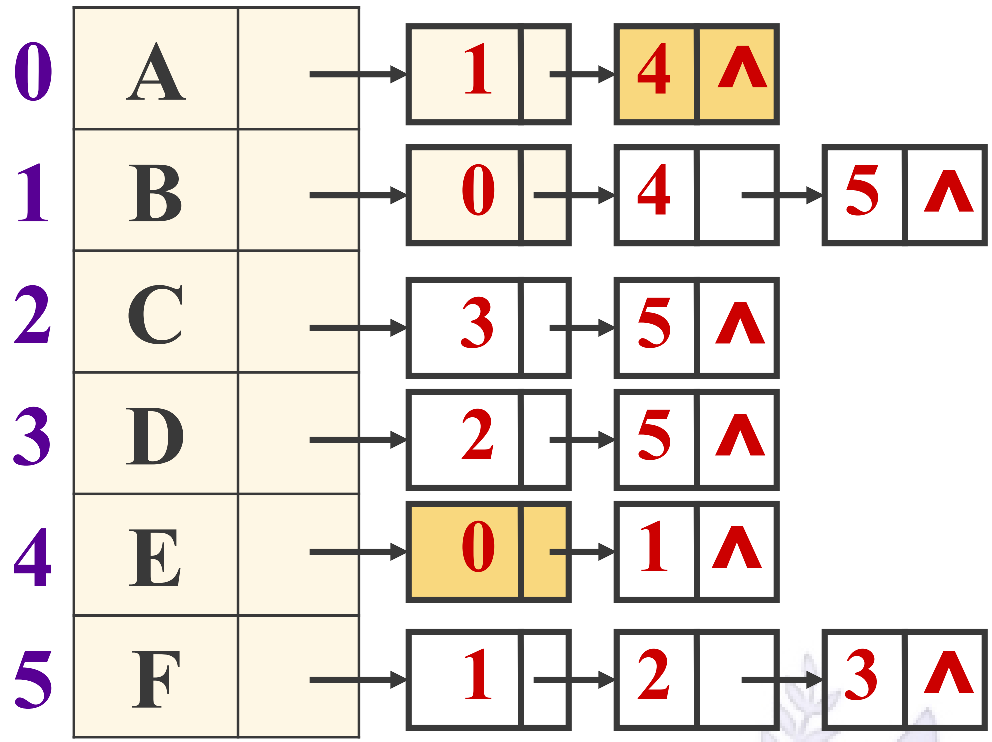
```
typedef struct { // 图的定义
    VertexType      // 顶点信息
            VerticesList [MAX_VERTEX_NUM];
    AdjMatrix   Edges;     // 弧的信息
    int   numVertices, numEdges;  // 顶点数，弧数
    GraphKind  kind;     // 图的种类标志
} MGraph;
```

```
Typedef enum {DG, DN, UDG, UDN} Graphkind;
```

6.2.2 图的邻接表存储表示

无向图的邻接表



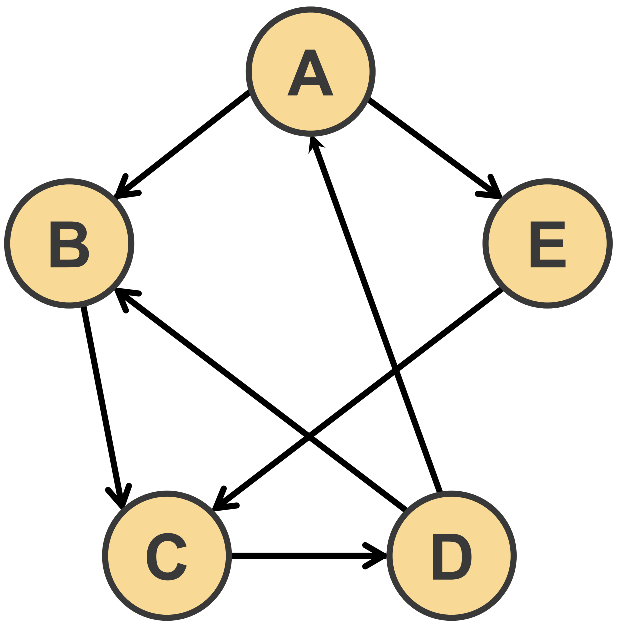


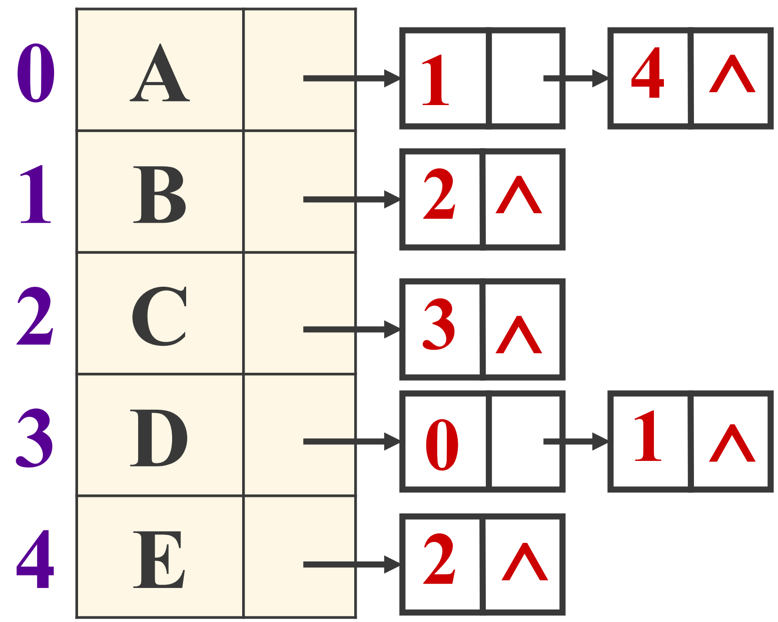
顶点i 的度？
顶点 i 边表长度

共有多少边结点？
2e。

6.2.2 图的邻接表存储表示

有向图的邻接表



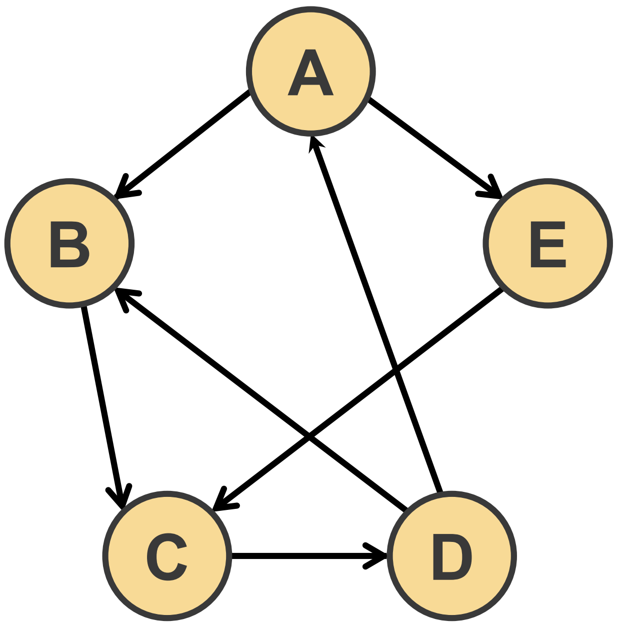


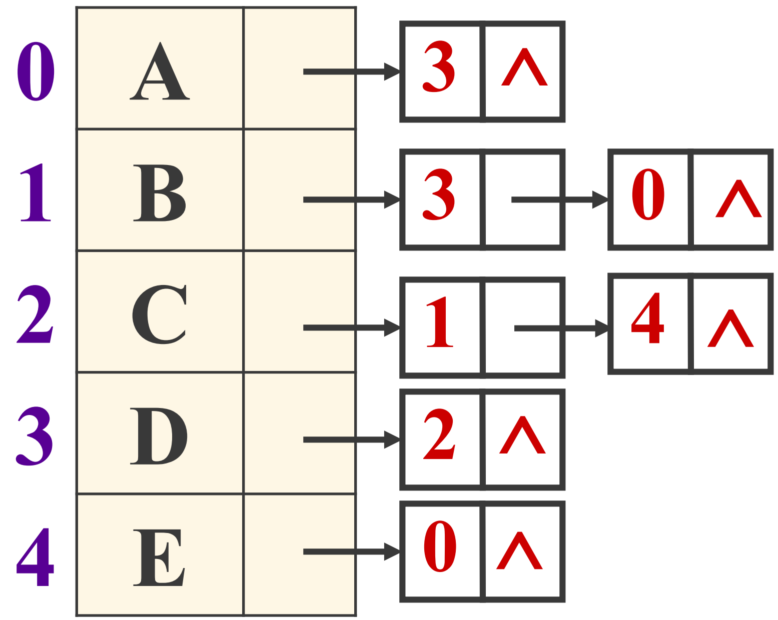
出边表

•顶点 *i* 的出度?	顶点 *i* 的出边表长度
•共有多少边结点?	e个

6.2.2 图的邻接表存储表示

有向图的逆邻接表





入边表

•顶点 *i* 的入度?	顶点 *i* 的入边表长度
•共有多少边结点?	e个

6.2.2 图的邻接表存储表示

弧的结点结构

dest	link	info

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

6.2.2 图的邻接表存储表示

顶点的结点结构

data	adj

```
typedef struct Vnode {
  VertexType  data;   // 顶点信息
  EdgeNode *adj; // 指向第一条依附该顶点的弧
  } VertexNode;
```

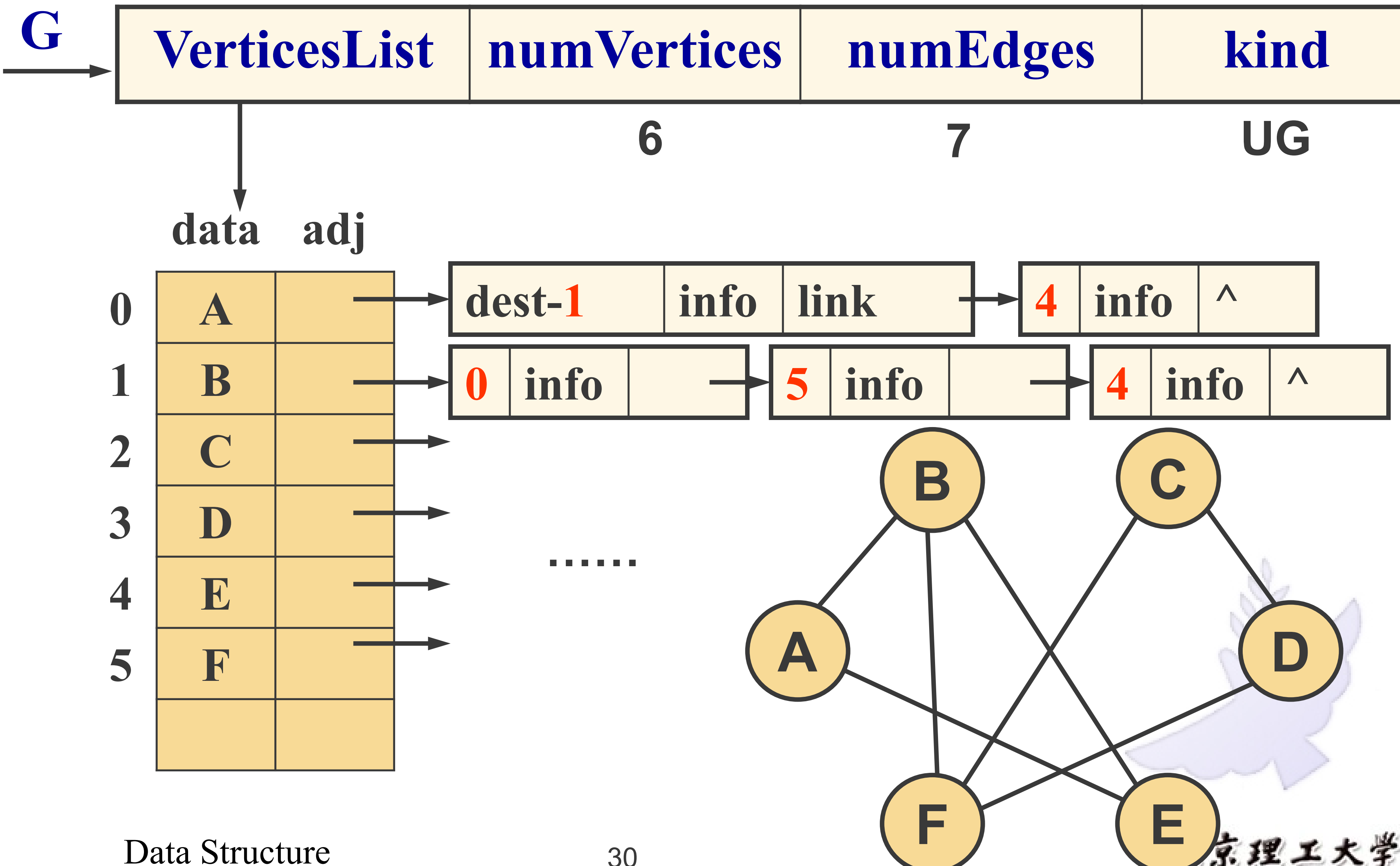
6.2.2 图的邻接表存储表示

VerticesList	numVertices	numEdges	kind

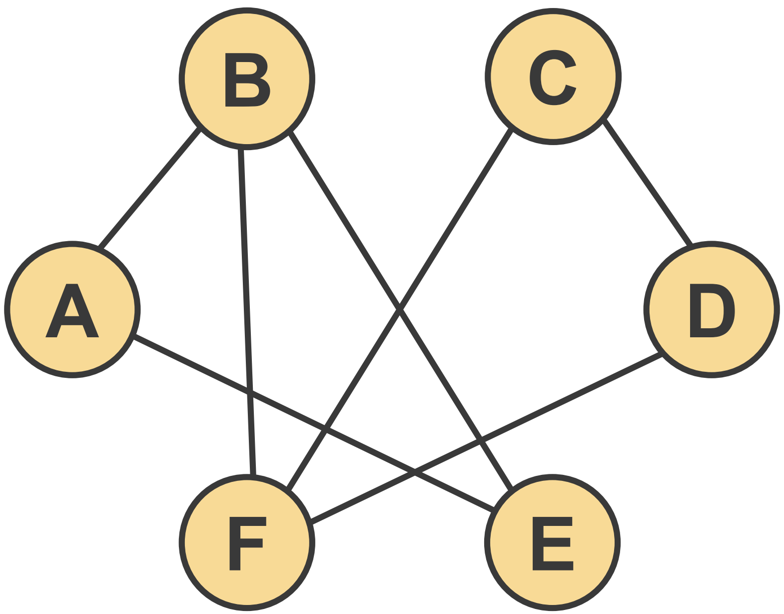
```
typedef struct { //图的结构定义
    //顶点列表
    VertexNode VerticesList[MAX_VERTEX_NUM];
    int     numVertices, numEdges; //顶点数，边数
    int     kind;          // 图的种类标志
} ALGraph;
```

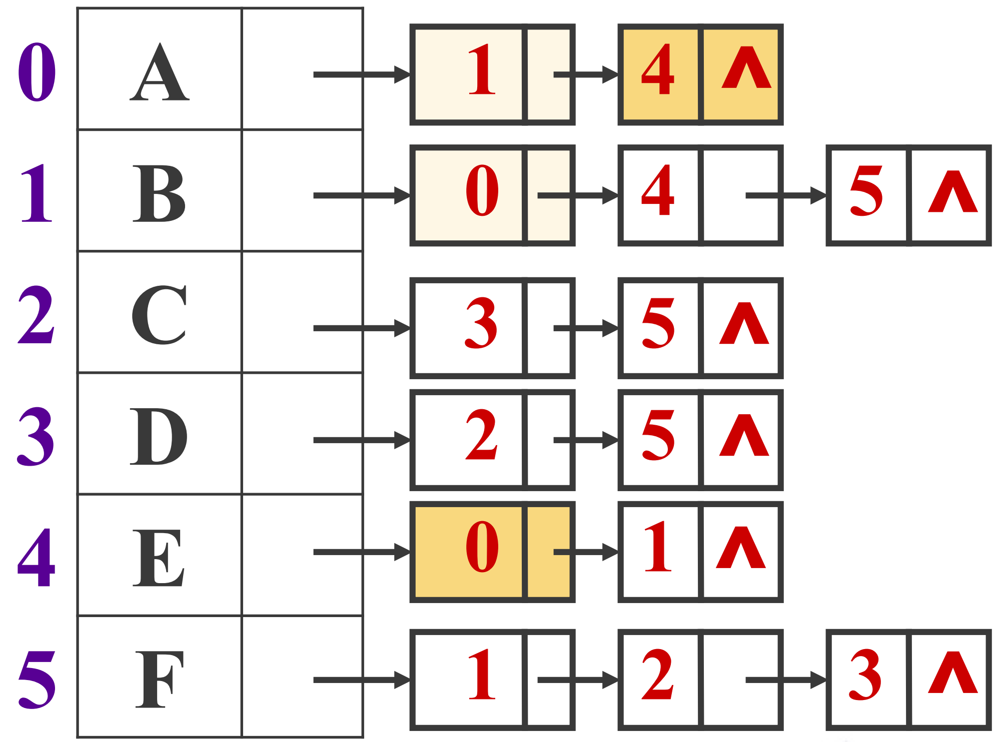
```
Typedef enum {DG, DN, UDG, UDN} Graphkind;
```

6.2.2 图的邻接表存储表示　ALGraph G;



6.2.2 图的邻接表存储表示



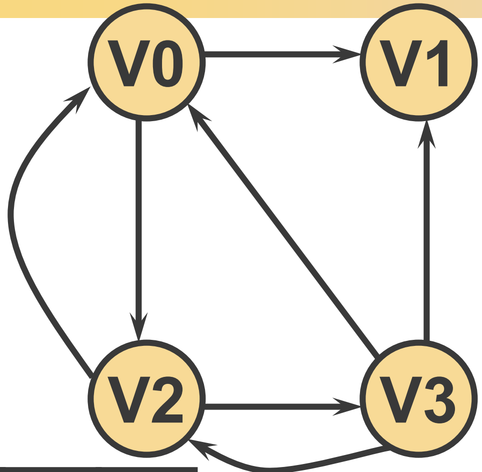


问题：输出无向图的算法复杂度是多少？
时间复杂度：O(n+e)；空间复杂度：O(1)

邻接矩阵和邻接表的比较

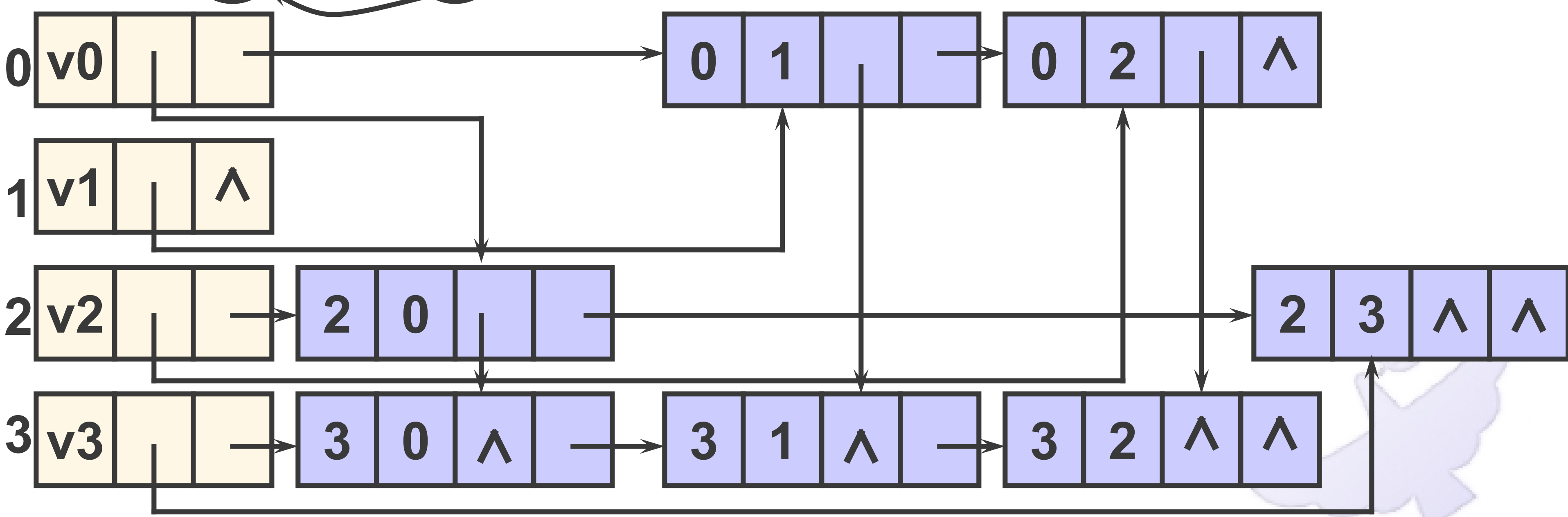
	邻接矩阵	邻接表
FirstNeighor(G, v)	最坏O(n)	O(1)
NextNeighor(G, v, w)	最坏O(n)	最坏O(e)
getWeight(G, v, w)	O(1)	最坏O(e)
printGraph(G)	$O(n^2)$	O(n+e)
空间效率	适合稠密图	适合稀疏图
时间效率	访问一条边 效率高	频繁访问邻接点 效率高

6.2.3 有向图的邻接多重表



•如何求顶点 *i* 的入度?
•如何求顶点 *i* 的出度?

十字链表



data	firstin	firstout

vertex1	vertex2	nextout	nextin	info

6.2.3 有向图的邻接多重表

顶点的结点结构

data	firstin	firstout

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

6.2.3 有向图的邻接多重表

弧的结点结构

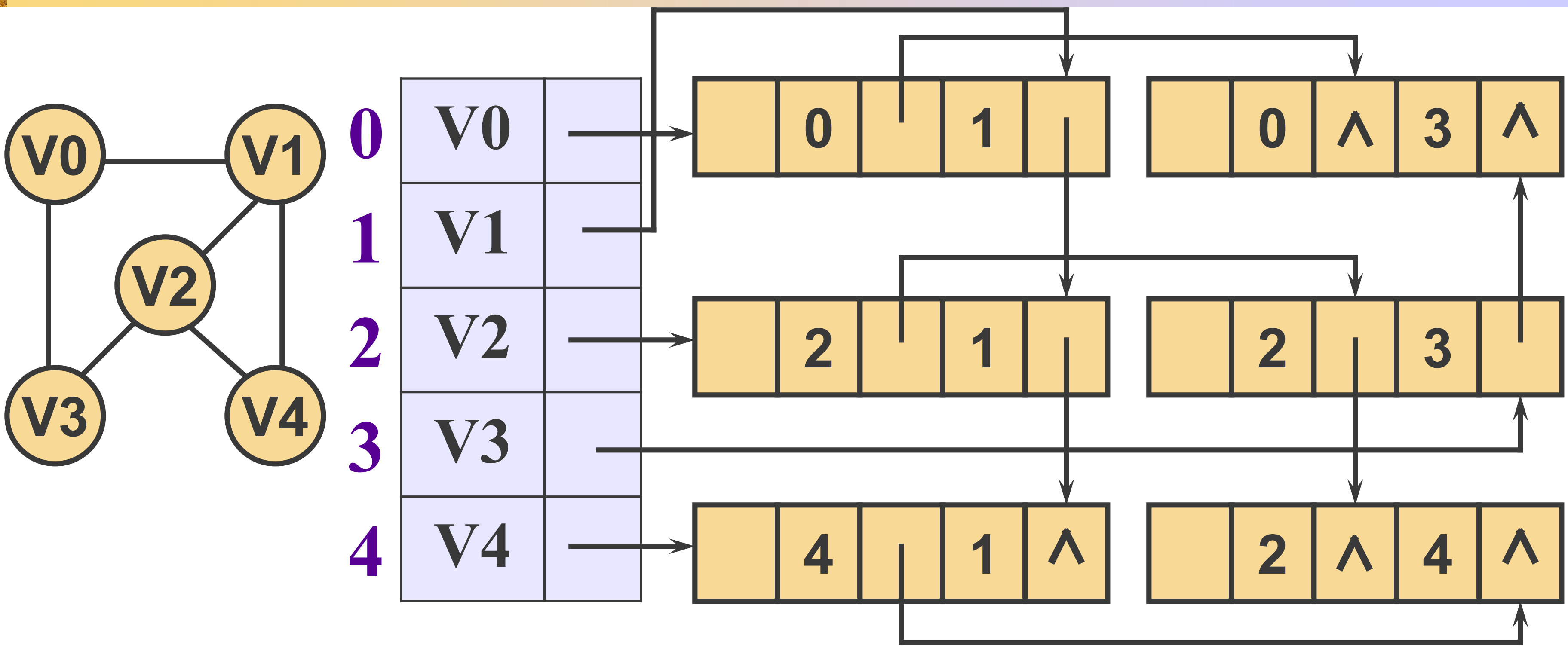
mark	vertex1	vertex2	nextout	nextin

```
typedef struct ArcBox { // 弧的结构表示
    int mark;
    int vertex1, vertex2;
    struct ArcBox  *nextout, *nextin;
    InfoType  *info;
} ArcBox ;
```

6.2.3 有向图的邻接多重表

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

6.2.4 无向图的邻接多重表存储表示



mark	vertex1	path1	vertex2	path2

•顶点 *i* 的度?

•多少边结点?

6.2.4 无向图的邻接多重表存储表示

mark	vertex1	path1	vertex2	path2

```
typedef struct Ebox { // 边的结构表示
    VisitIf      mark;       // 访问标记
    int   vertex1, vertex2;    //该边关联的两个顶点
    //分别指向关联于vertex1和vertex2的下一条边
    struct EBox  *path1, *path2;
    InfoType     *info;          // 该边信息指针
} EBox;
```

6.2.4 无向图的邻接多重表存储表示

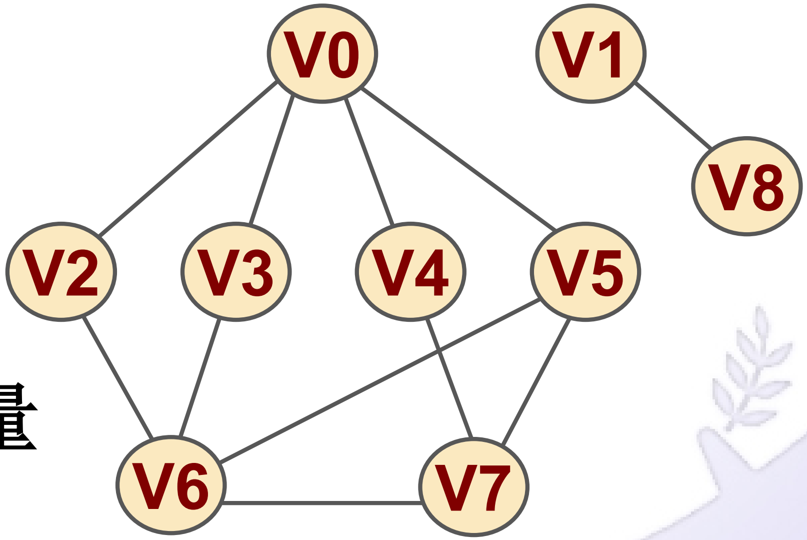
```
typedef struct VexBox {//顶点的结构
   VertexType  data;
   EBox  *Firstout; // 指向第一条关联的边
} VexBox;
```

```
typedef struct {  // 无向图的邻接多重表
   VexBox  adjmulist[MAX_VERTEX_NUM];
   int   numVertices, numEdges;
 } AMLGraph;
```

6.3 图的遍历

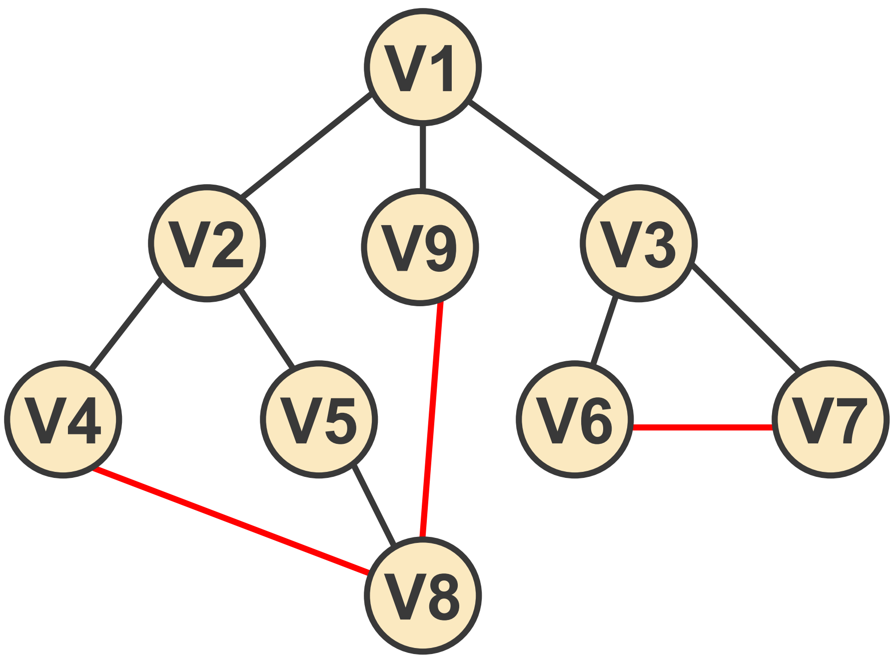
- 图的遍历：从图中某个顶点出发游历图，访遍图中其余顶点，并且使图中的每个顶点仅被访问一次的过程
- 由于图中结点可能会多次到达，所以设置数组 **visited[0,…n]**，标志结点是否被访问过

- 6.3.1 深度优先搜索
- 6.3.2 广度优先搜索
- 6.3.3 连通分量
- 6.3.4 有向图的强连通分量
- 6.3.5 双连通图



6.3.1 深度优先搜索

- **连通图的遍历**
- 深度优先遍历连通图的过程类似于树的先根遍历
- 从图中某个顶点V 出发，访问此顶点，然后依次从V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有路径相通的顶点都被访问到.



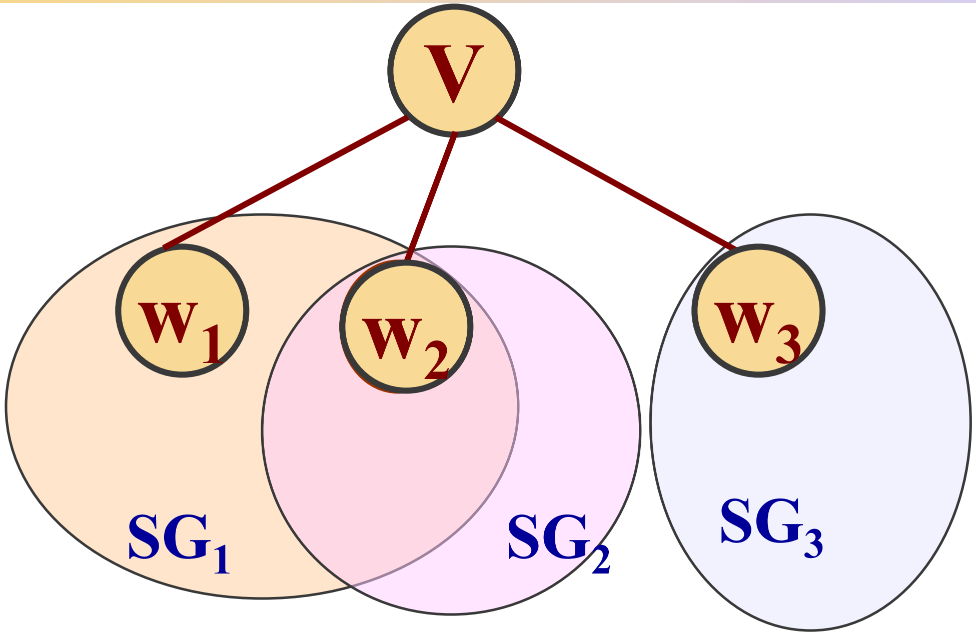
树的先根遍历：

V1,V2,V4,V5,V8,V9,V3,V6,V7

图的深度优先遍历：

V1,V2,V4,V8,V5,V9,V3,V6,V7

6.3.1 深度优先搜索



访问顶点 V ：

for (W_1、W_2、W_3)

若该邻接点Wi未被访问,

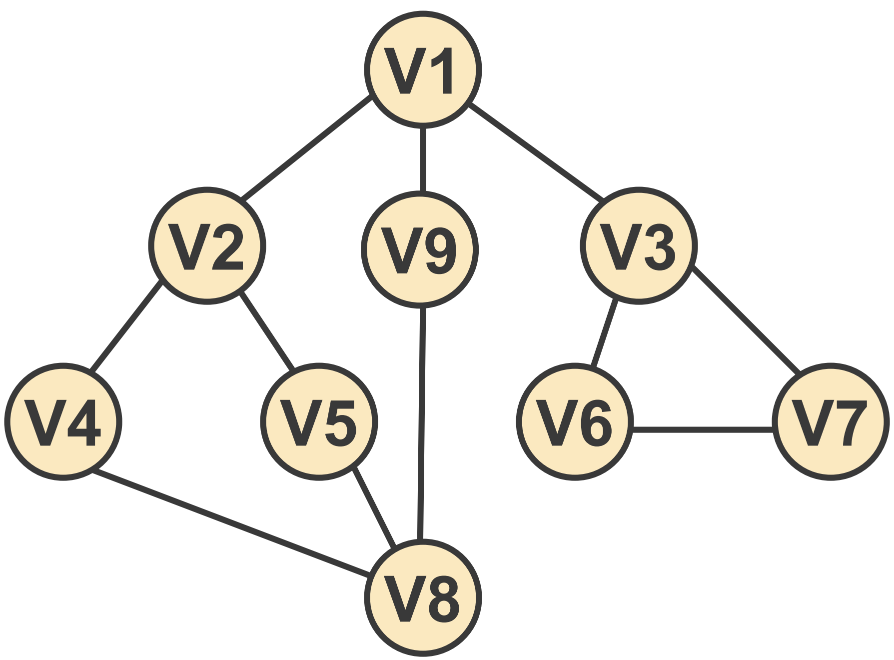
则从它出发进行深度优先搜索遍历。

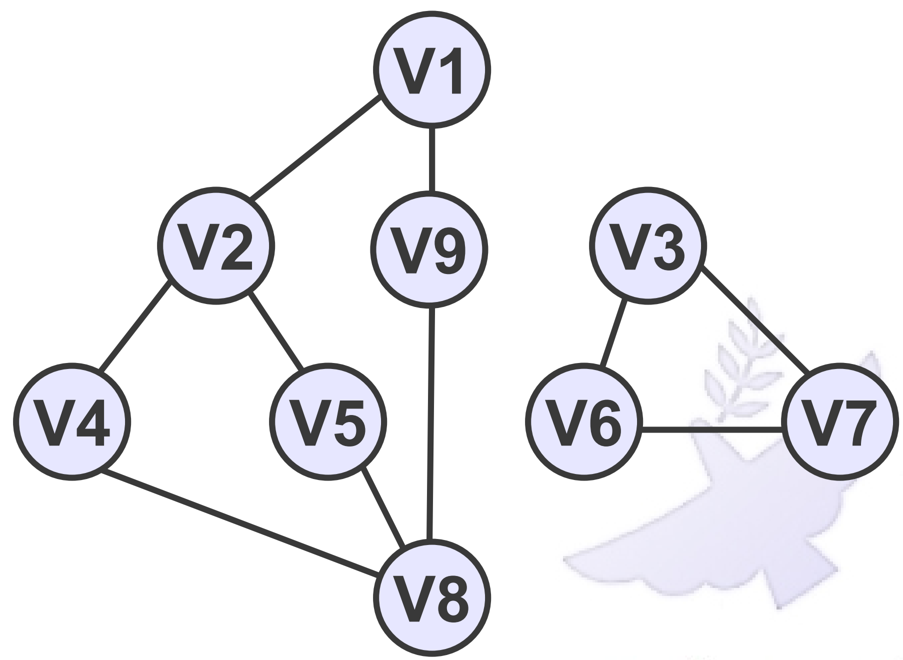
6.3.1 深度优先搜索

```
void DFS(Graph G, int v, void (* visit)(VertexType)
                , int visited[ ] ) {
  // 从顶点v出发，深度优先搜索遍历连通图 G
  visited[v] = TRUE;   visit(v);
  w = firstNeighbor (G, v);
  while(w!=-1){ //对v未访问的邻接顶点w,递归调用DFS
        if (!visited[w])  DFS(G, w, visit, visited);
        w = nextNeighbor  (G,v,w); }
} // DFS
```

6.3.1 深度优先搜索

- **非连通图的深度优先搜索遍历**
- 1) 将图中每个顶点的访问标志设为 FALSE
- 2) 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

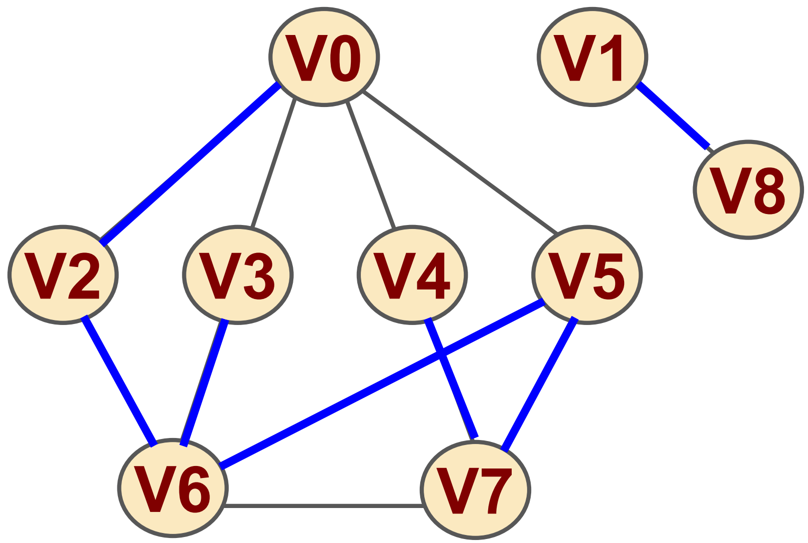




6.3.1 深度优先搜索

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

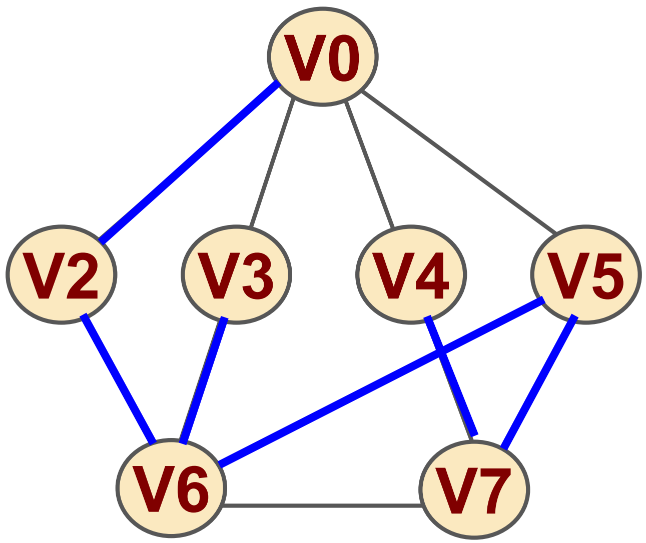
6.3.1 深度优先搜索

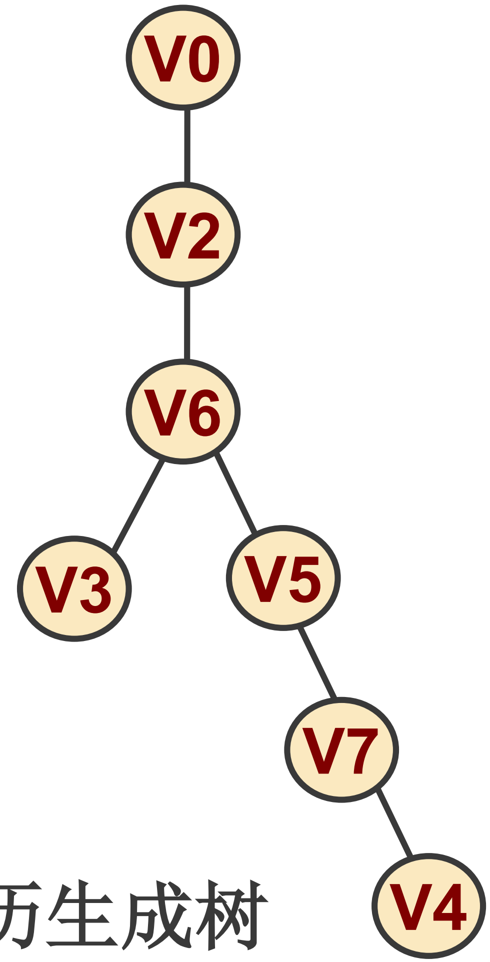


访问次序:

0	2	6	3	5	7	4	1	8

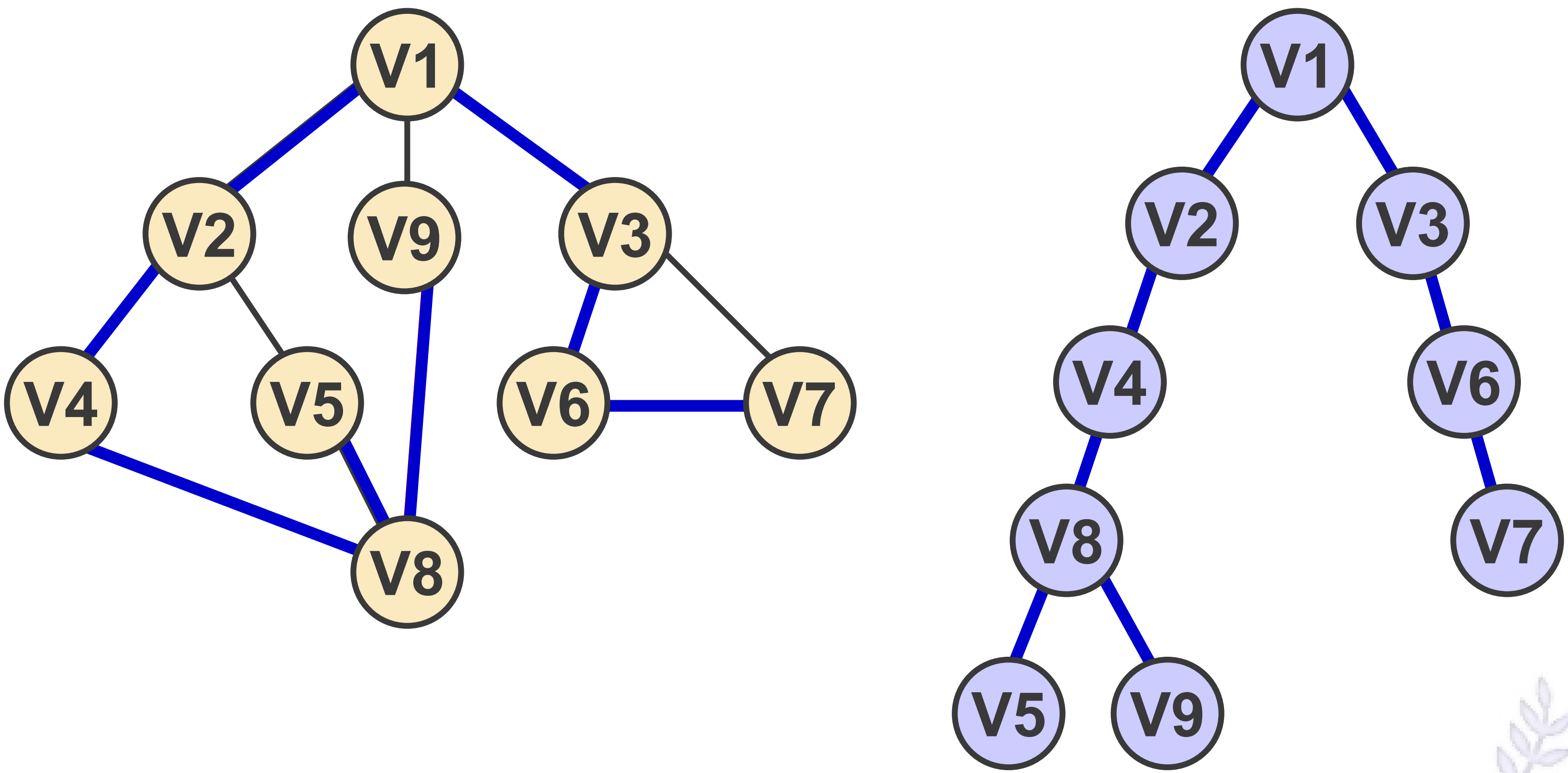
6.3.1 深度优先搜索





连通图的深度优先遍历生成树

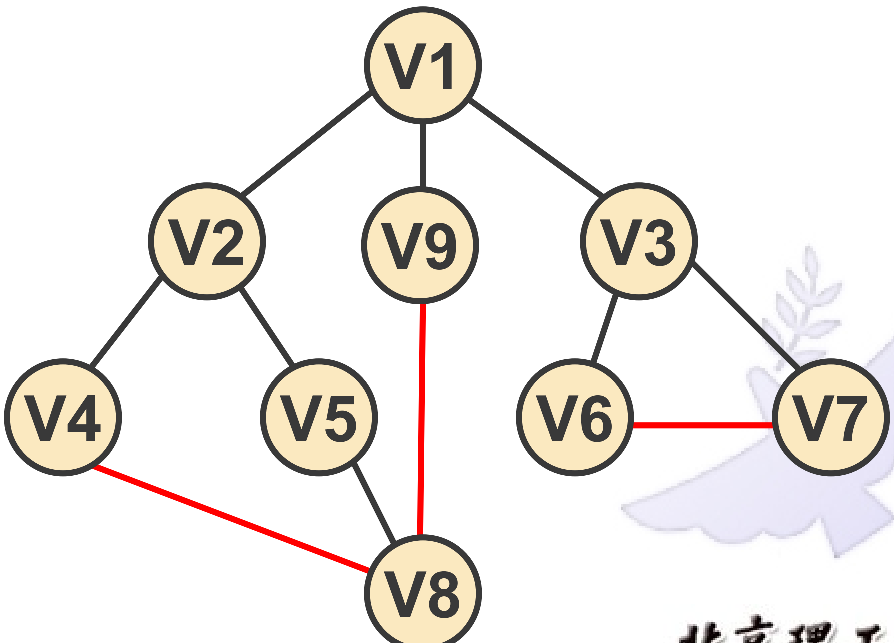
6.3.1 深度优先搜索



连通图的深度优先遍历生成树

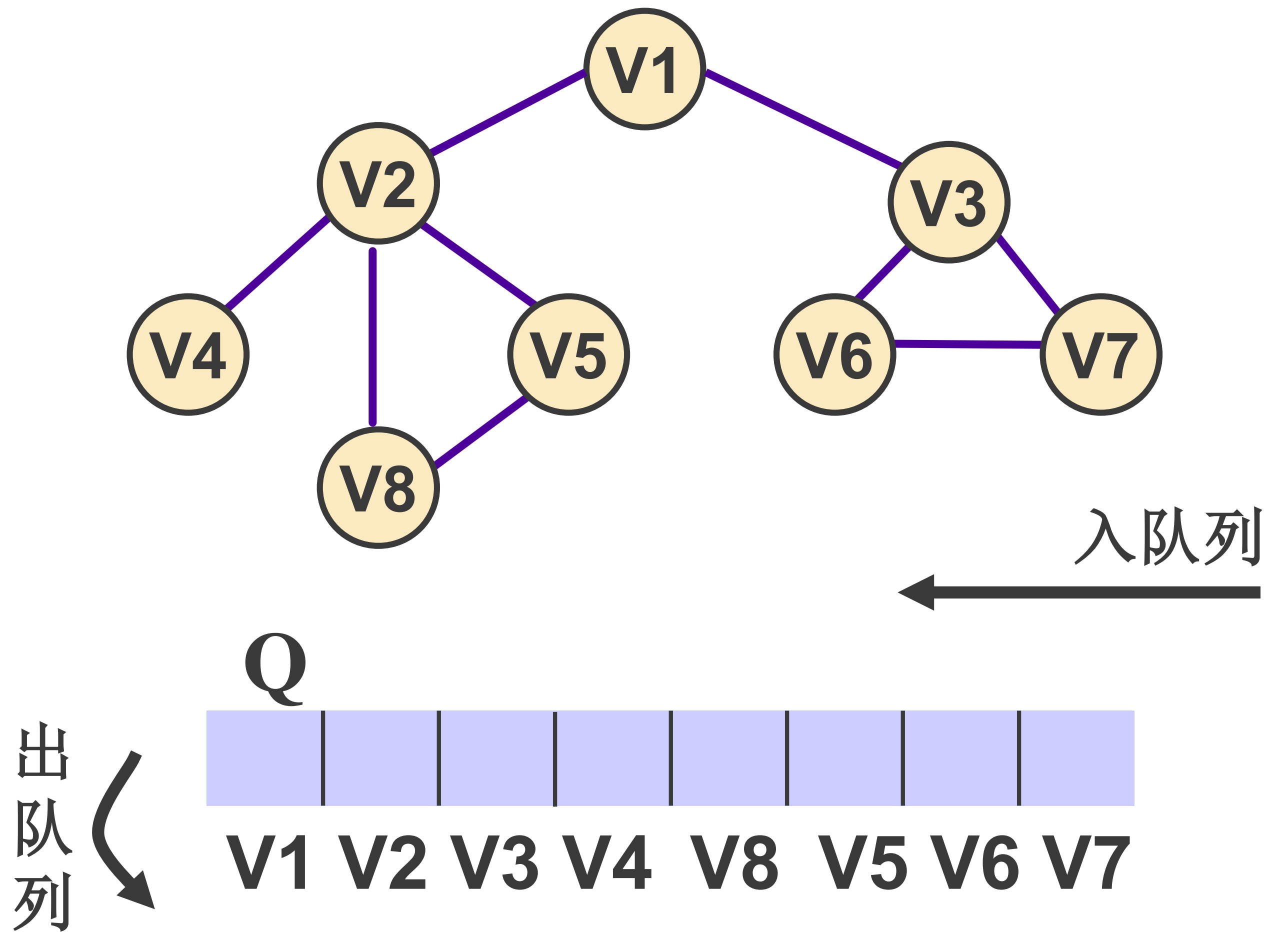
6.3.2 广度优先搜索

- 类似树的广度优先遍历
- 图中某顶点v出发： 1）访问顶点v；
- 2）访问v所有未被访问的邻接点w1, w2, …, wk;
- 3）依次从这些邻接点出发，访问其所有未被访问的邻接点。 依此类推，直到图中所有访问过的顶点的邻接点都被访问
- 借用队列暂存结点



```
void BFS(Graph G, VexIndex  v,
                        void (*visit)(VertexType), int visited[ ])
{//从第v个顶点出发，广度优先遍历G,使用辅助队列Q。
    InitQueue(Q);  //建空队列Q
    visit(v);  visited[v]=TRUE; EnQueue(Q,v) //访问v, v入队
    while(!QueueEmpty(Q)){
        DeQueue(Q,u);   //队头元素出队,并赋值给u
        w = firstNeighbor(G,u);
        while(w!=-1){ //访问u未访问的邻接顶点w，w入队
         if (!visited[w]) {
              visit(w);  visited[w]=TRUE; EnQueue(Q,w); }
         w = nextNeighbor (G,v,w); }
    }//while(!QueueEmpty(Q))
}//BFS
```

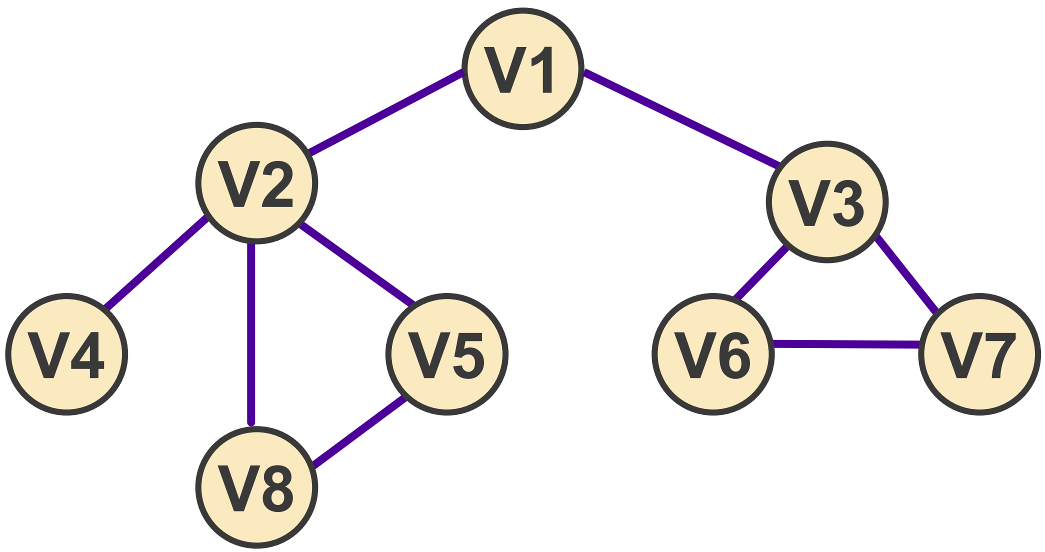
6.3.2 广度优先搜索



```
void BFSTraverse(Graph G,
                   void (*visit)(VertexType)) {
  // 对图 G 作广度优先遍历。
   for (v=0; v<G.numVertices; ++v)
       visited[v] = FALSE; // 访问标志数组初始化
   // 对尚未访问的顶点调用BFS
   for (v=0; v<G.numVertices; ++v)
       if (!visited[v])  BFS(G, v, visit, visited);
}// BFSTraverse
```

6.3.3 图遍历应用举例1

- 求一条从顶点 v到顶点 s 的简单路径
- 从v开始深度优先搜索，找到s为止
- V1→V2：Path=(V1,V2)
- V1→V8：Path=(V1,V2,V8)
- V1→V7：Path=(V1,V3, V6, V7)

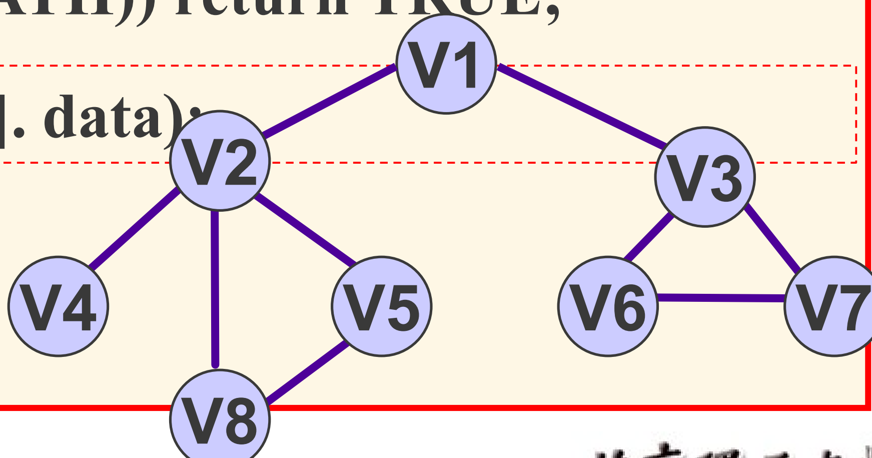


6.3.3 遍历应用举例1

◆ 求一条从顶点 v 到顶点 s 的简单路径

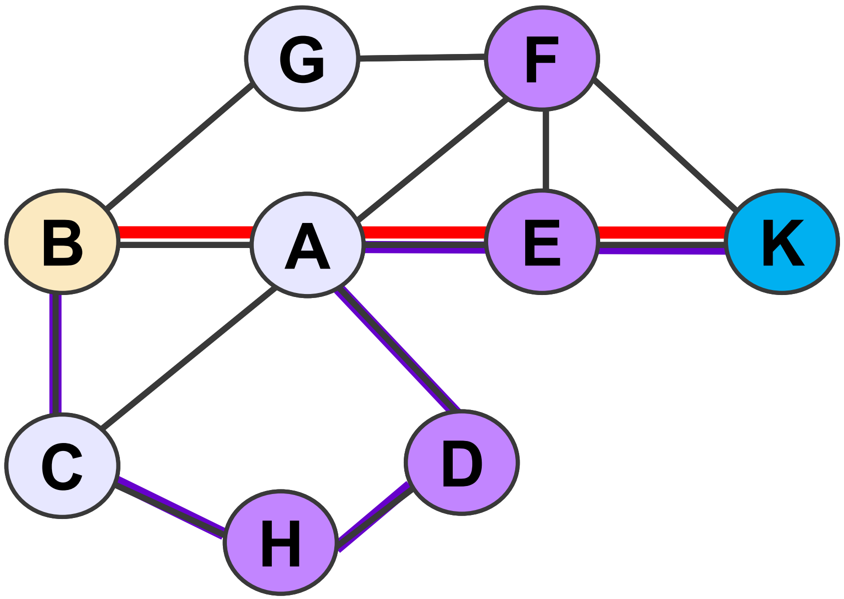
```
Status DFSearch(Graph G, VertexType v, VertexType s
        , SqList &PATH) {//从v开始深度优先搜索，找到s为止
    v1 = LocateVex(G, v); //找到v
    if ( v1 == -1) return FALSE;
    for (i=0; i<G.numVertices; ++i)
        visited[i] = FALSE; // 访问标志数组初始化
    InitList_Sq(PATH);
    return _DFSearch(G, v1, s, PATH); //从v开始深度优先搜索
}// DFSearch
```

```
Status _DFSearch(Graph G, int v, VertexType s, SqList &PATH)
{//深度优先搜索的递归程序
  visited[v] = TRUE;   // 访问第 v 个顶点
 ListAppend_Sq(PATH, G.vertices[v].data); //将点v添加到路径
 if (G.vertices[v].data == s)     return TRUE;//找到路径
  for(w = firstNeighbor(G, v); w != -1; w = nextNeighbor(G, v, w))
      if (!visited[w])
            if (_DFSearch(G, w, s, PATH)) return TRUE;
 ListDelete_Sq(PATH,G.vertices[v]. data);
 return FALSE;
}// _DFSearch
```

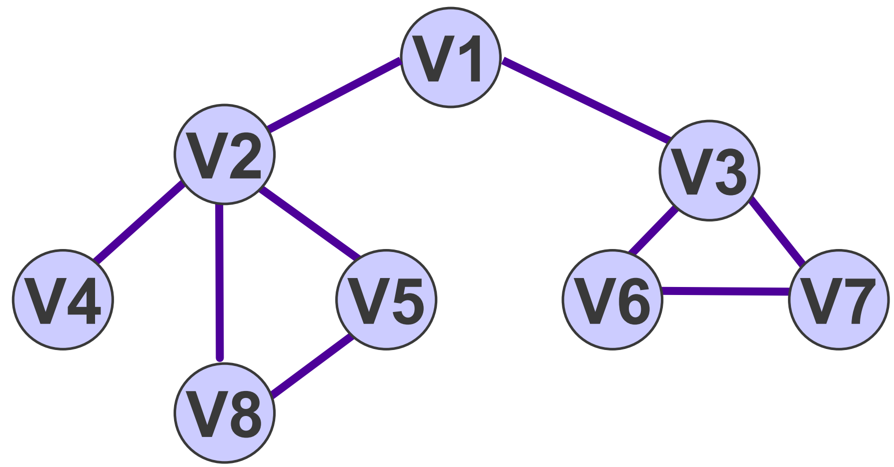


6.3.3 遍历应用举例2

- 求两个顶点之间的一条路径长度最短的路径



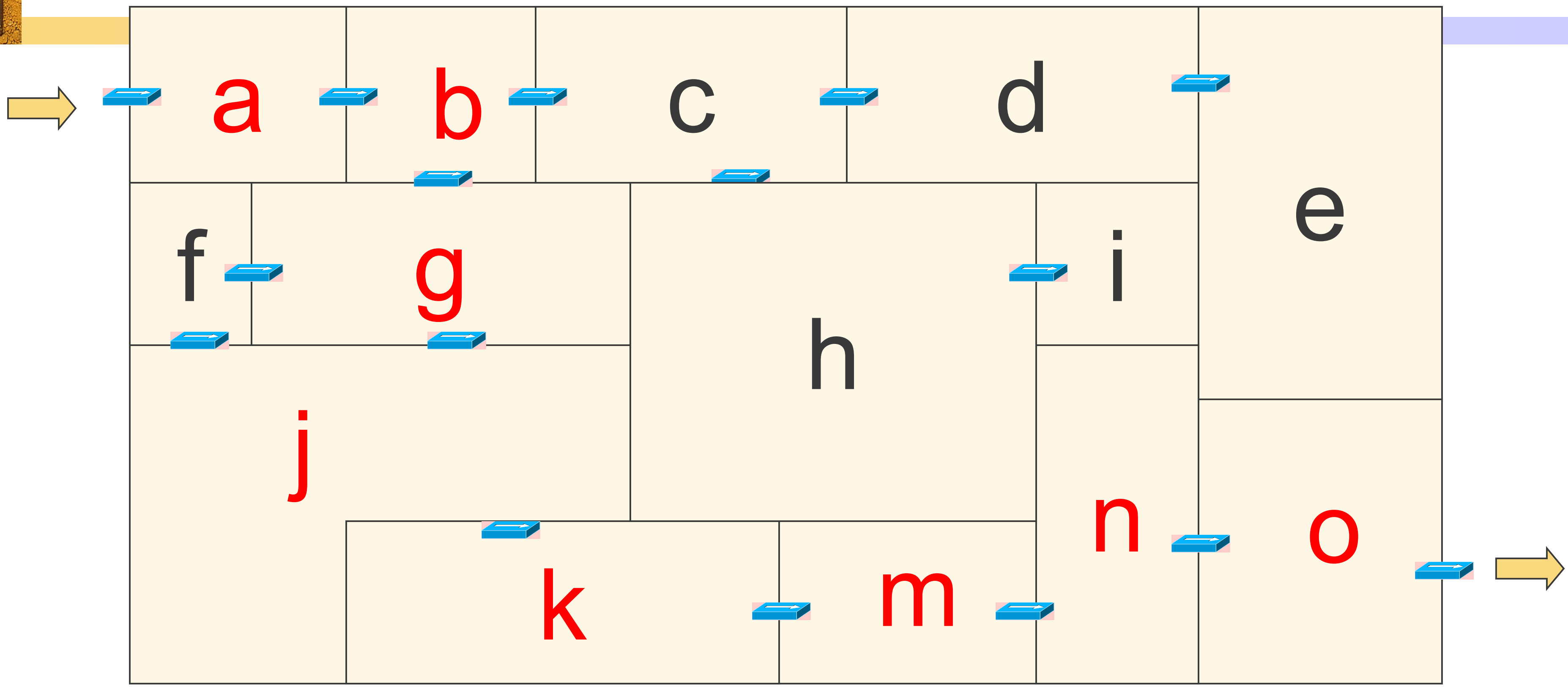
考察A到K的路径

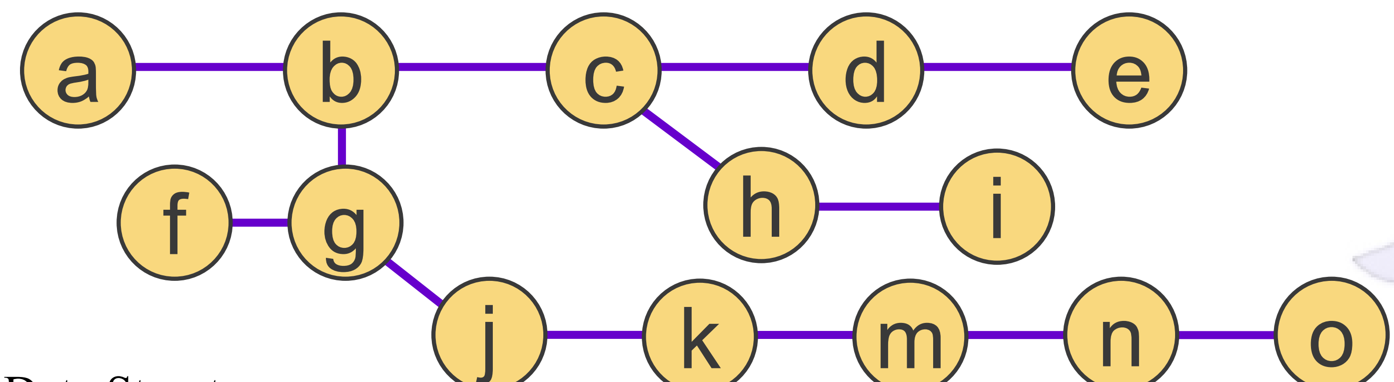


考察V1到V5的路径

广度优先遍历的次序按“路径长度”渐增的次序进行的

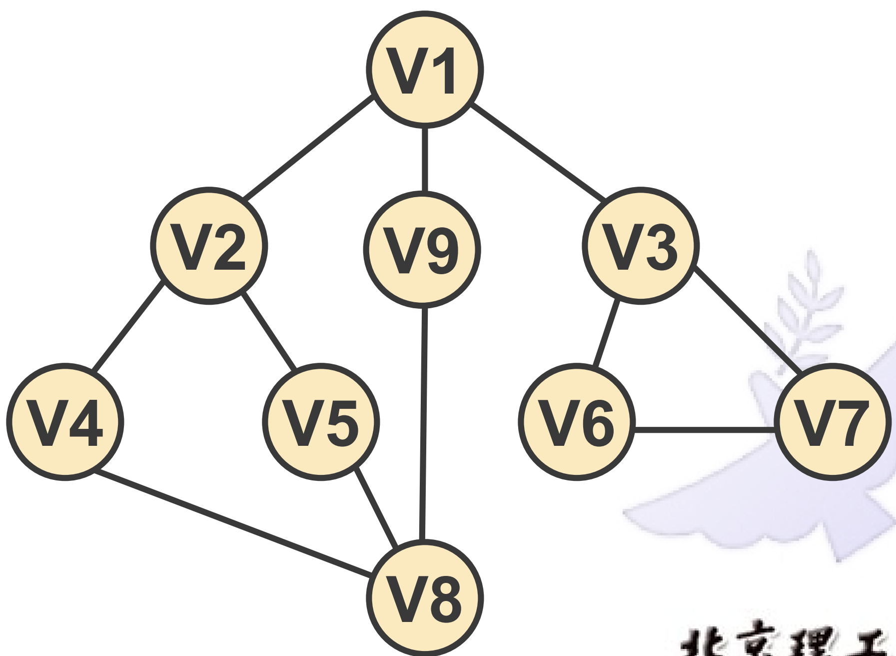
走迷宫





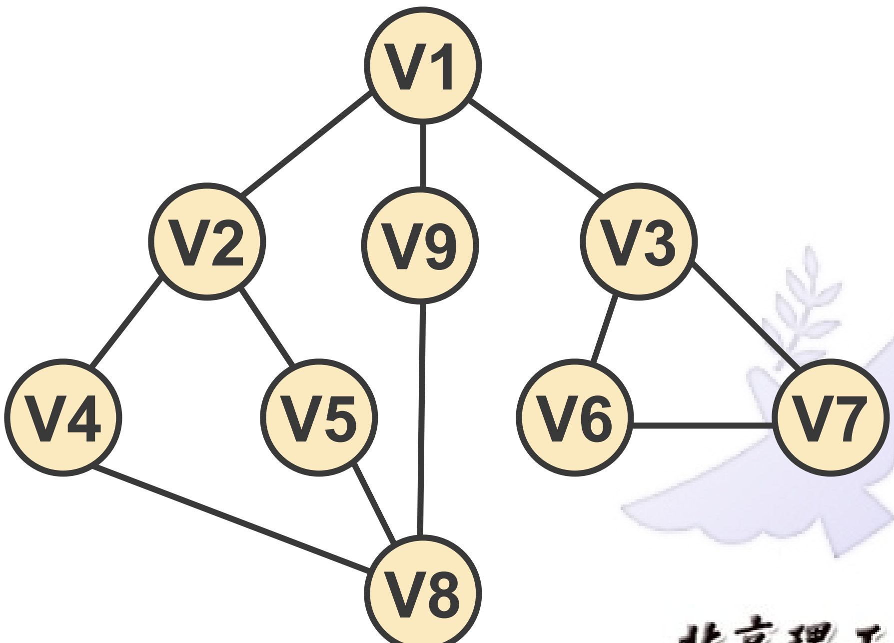
图的深度优先遍历的复杂度

- 从图中某个顶点V 出发
- 1）访问此顶点；
- 2）依次从V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有路径相通的顶点都被访问到
- 借用栈暂存结点
- 复杂度：
  - 邻接矩阵：O(n^2)
  - 邻接表O(n+e)



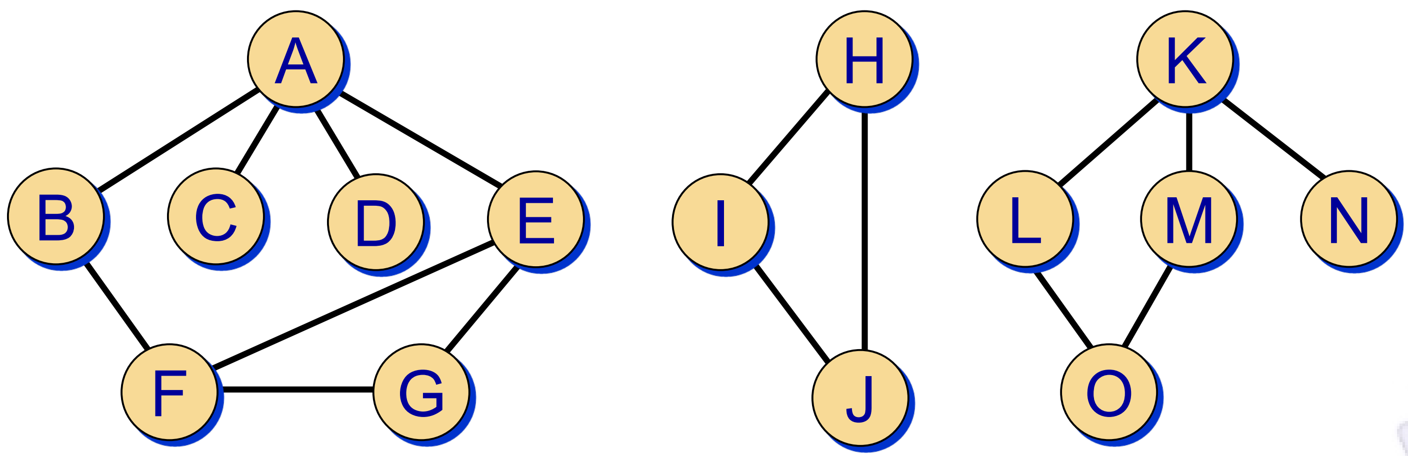
图的广度优先搜索的复杂度

- 图中某顶点v出发：
- 1）访问顶点v；
- 2）访问v所有未被访问的邻接点w1,w2,…wk;
- 3）依次从这些邻接点出发，访问其所有未被访问的邻接点。依此类推，直到图中所有访问过的顶点的邻接点都被访问
- 借用队列暂存结点
- 复杂度：
  - 邻接矩阵：$O(n^2)$
  - 邻接表$O(n+e)$



6.3.3 无向图的连通分量

- 无向图的连通分量
  - 从无向图的每一个连通分量中的一个顶点出发进行遍历，可求得无向图的所有连通分量。

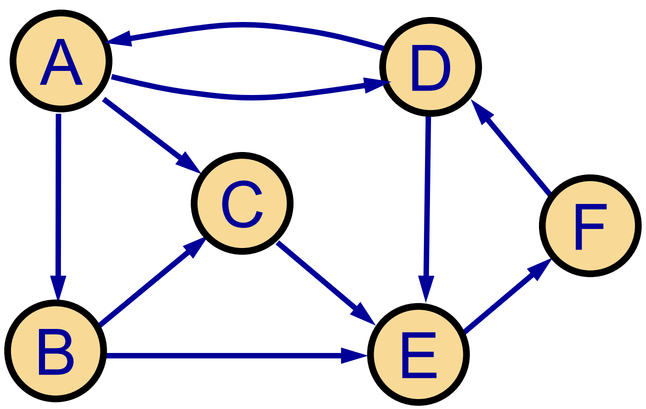


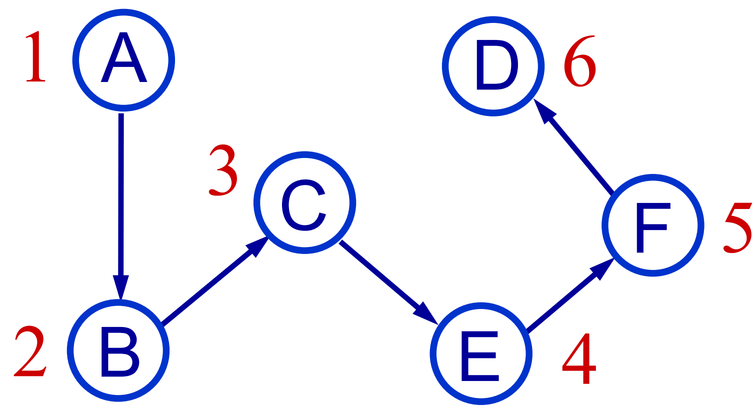
6.3.4有向图的强连通分量

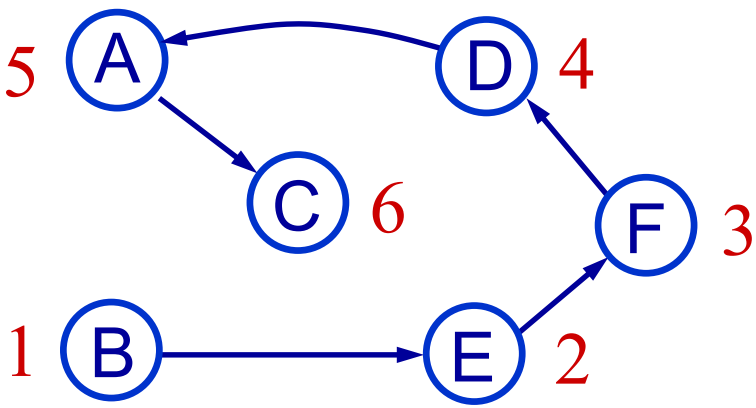
- 从不同顶点出发，对有向强连通图进行遍历，可以得到不同的生成树

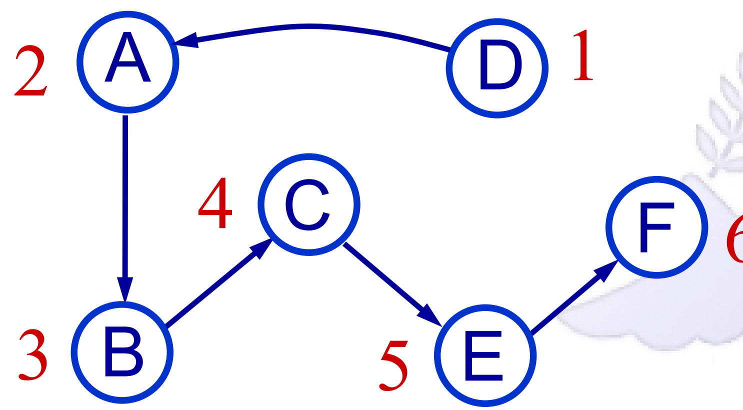
深度优先生成树

强连通图



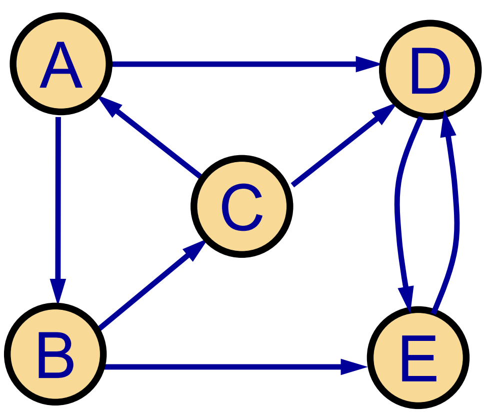




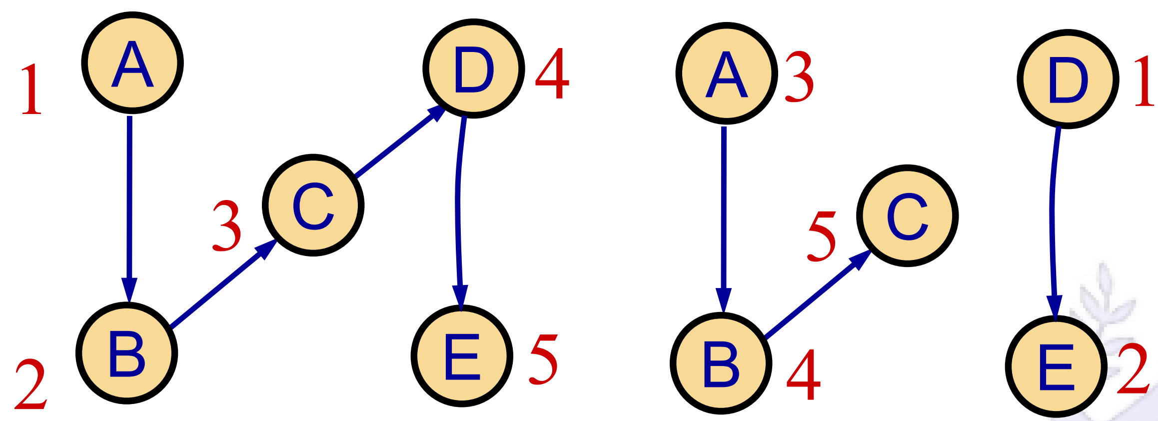


6.3.4有向图的强连通分量

- 非强连通有向图的遍历一般得到的是生成森林。
- 对于非强连通图，从某个顶点出发，只能遍访一个弱连通分量。
- 可以通过 **DFS** 遍历，求出所有强连通分量



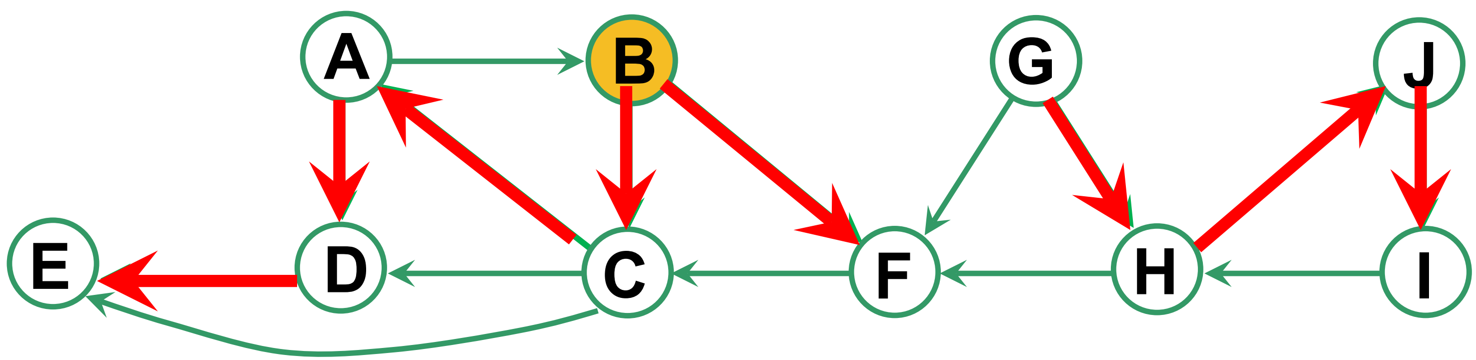
非强连通图

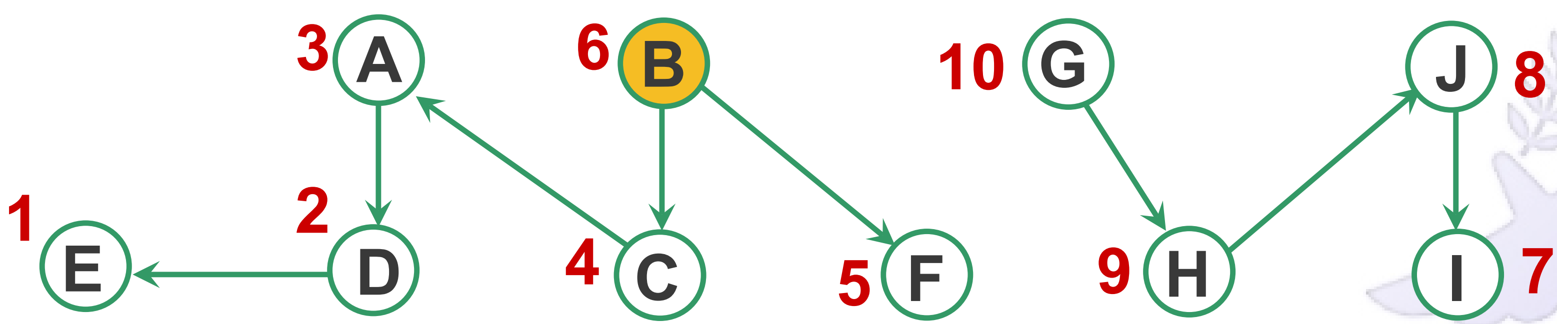


深度优先生成森林

寻找强连通分量的Kosaraju算法

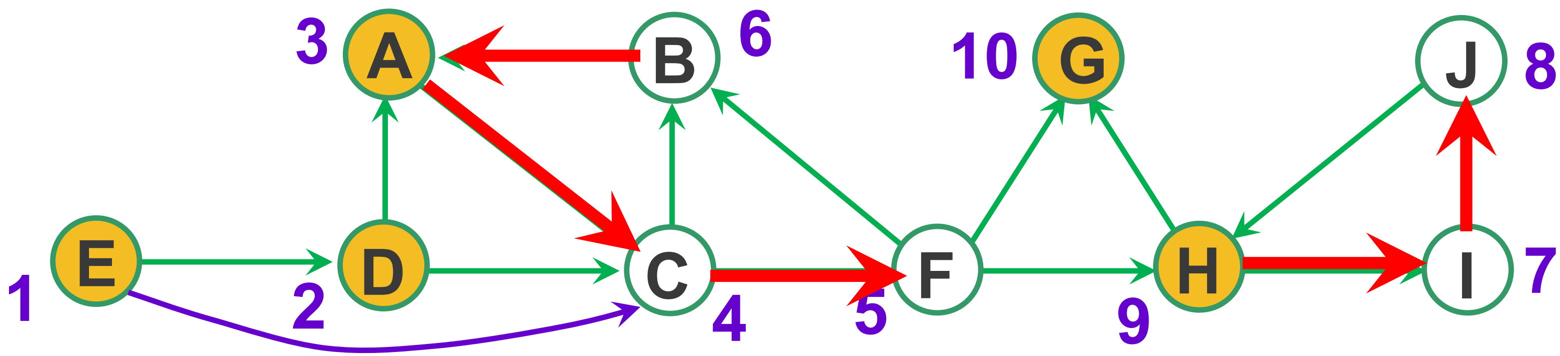
- 基本思路：
- 1、首先对图进行一次DFS，在回退时记录对结点回溯的顺序：EDACFBIJHG



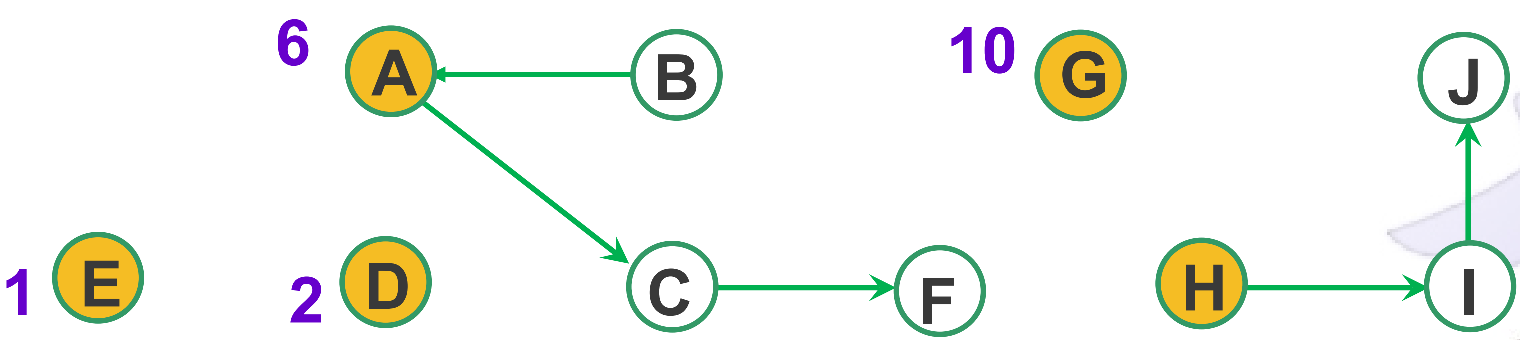


寻找强连通分量的Kosaraju算法

- 2、把图的所有的有向边逆转.

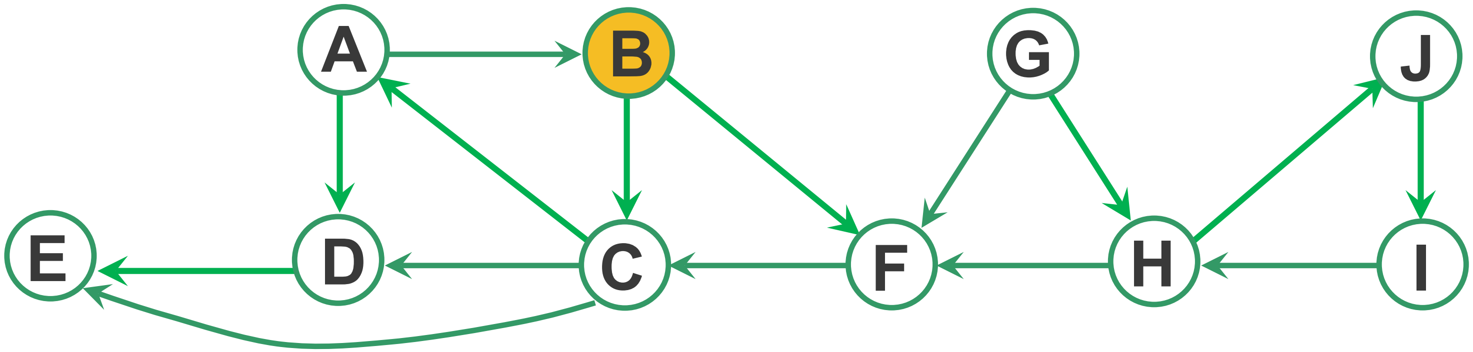


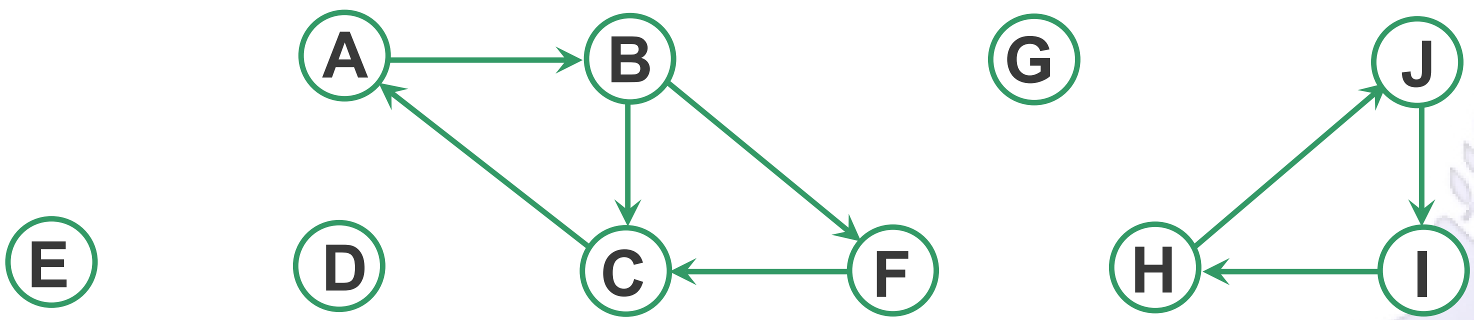
- 3、对得到的图沿回溯顺序 **EDACFBIJHG** 从编号最高的 G 开始，再进行一次 DFS，所得到的深度优先森林（树）即为强连通分量的划分



寻找强连通分量的Kosaraju算法

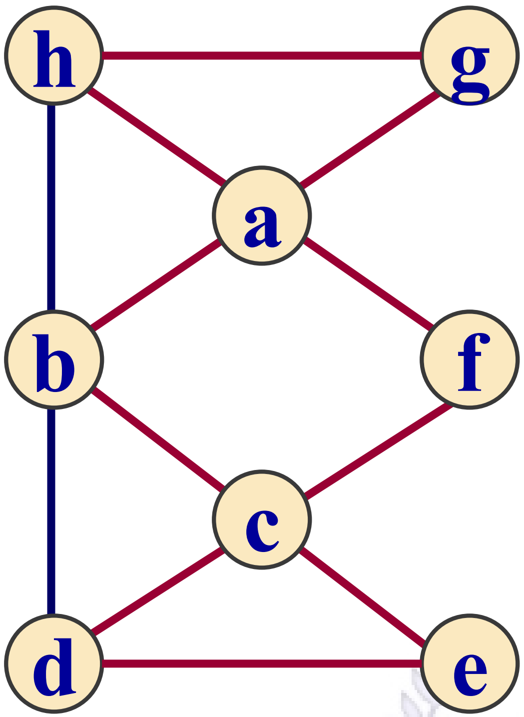
- 对应到原图，得到 5 个强连通分量





6.3.5 双连通图和关结点

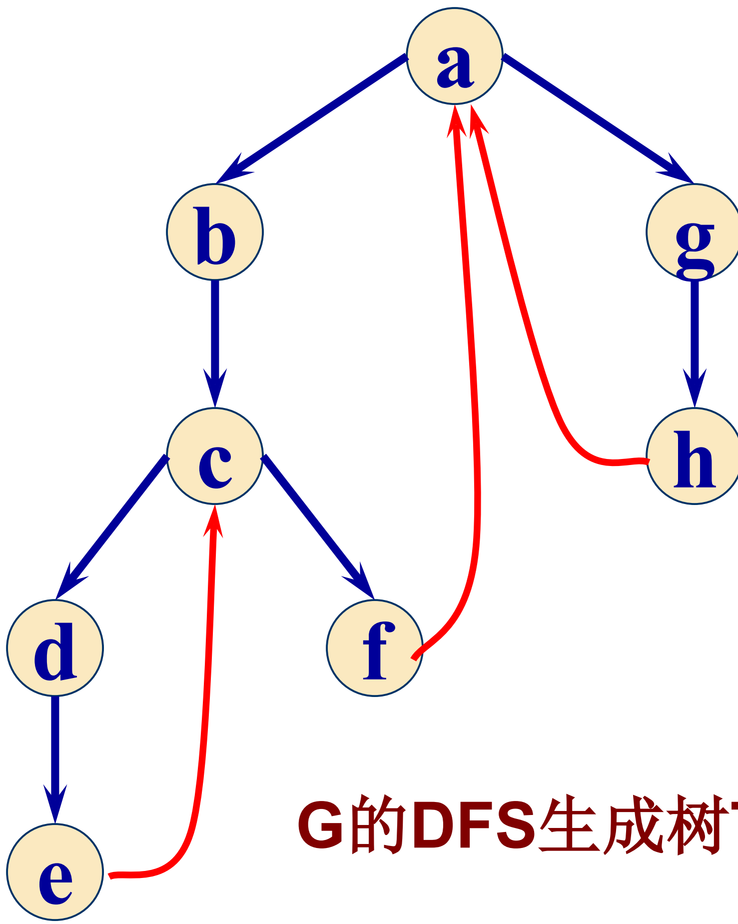
- 定义：若从一个连通图中删去一个顶点及其相关联的边，连通图成为两个或多个连通分量，则该点称为关节结点。
- 定义：若从一个连通图中删去任意一个顶点及其相关联的边，它仍为一个连通图的话，则该连通图被称为重（双）连通图。



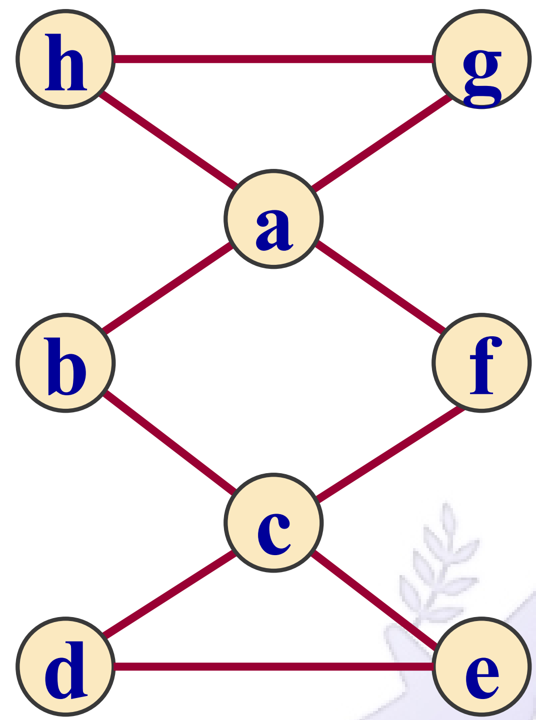
重连通图中没有关结点
没有关结点的连通图为重连通图

6.3.5 重连通图和关结点

- 对G进行深度优先遍历，得到深度优先生成树T
- 在遍历树中添加回联边：即在G中不在T中的边

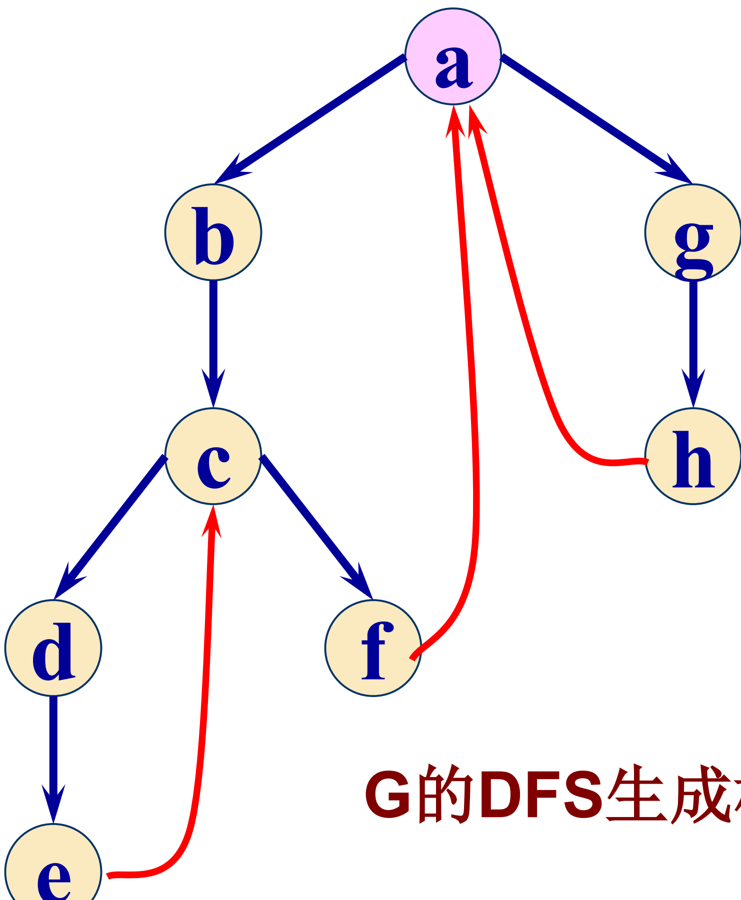


G的DFS生成树T

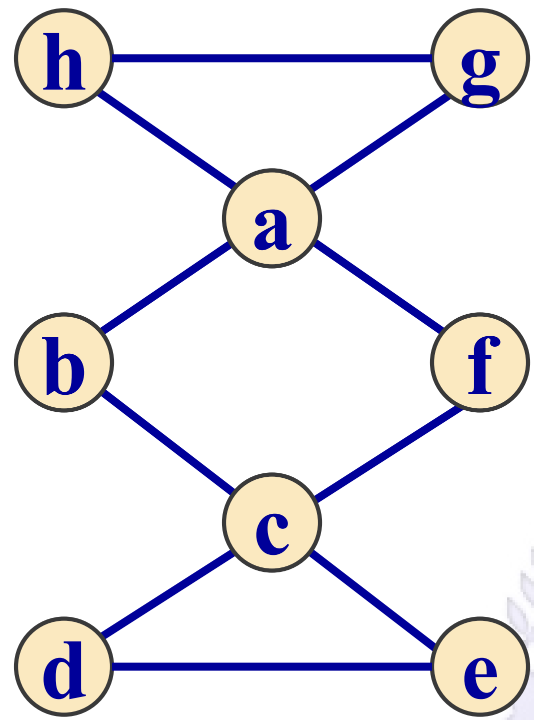


在修改的T中关结点的特征

- 特征1：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)必为关结点；

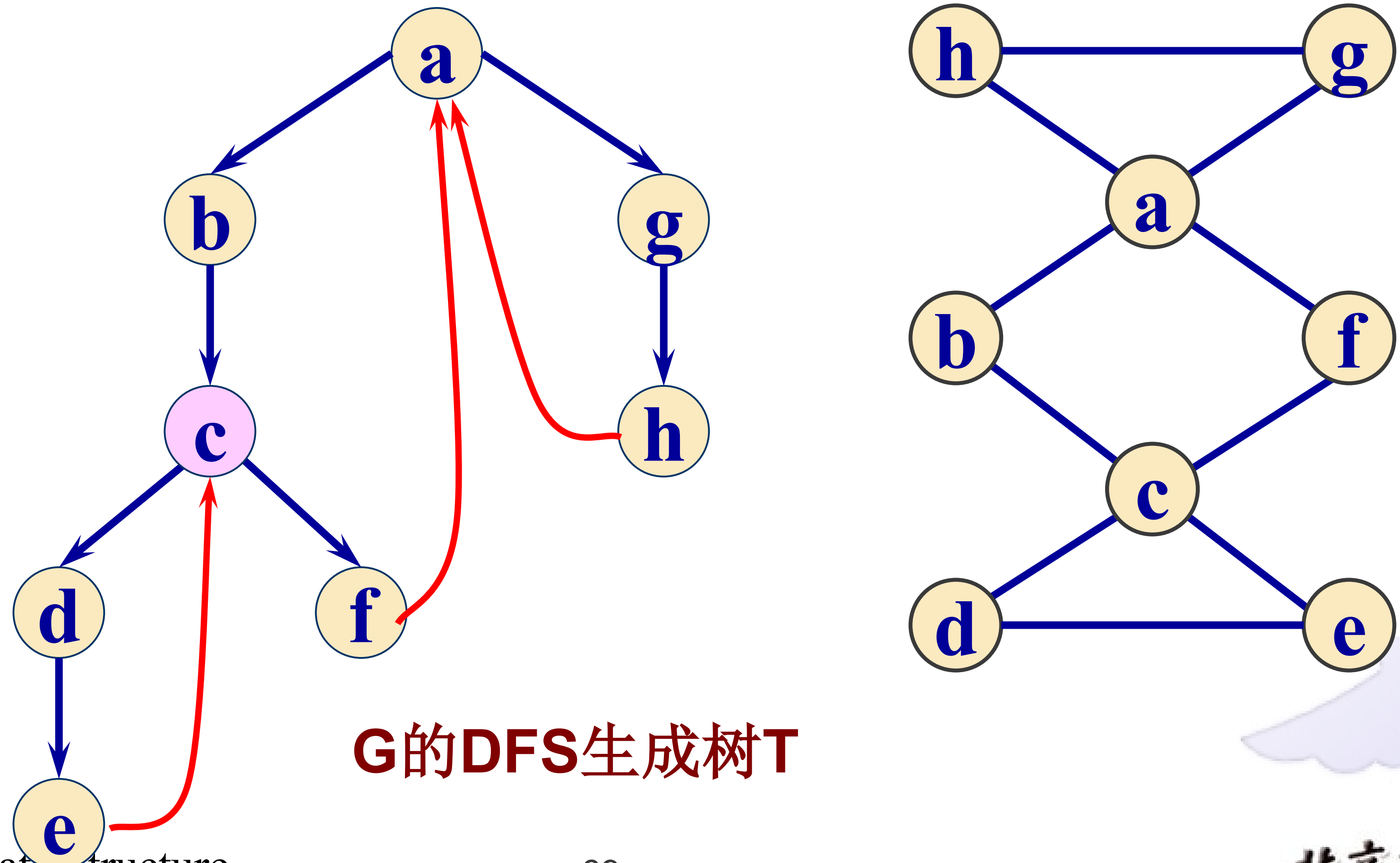


G的DFS生成树T



◆ 特征2：对生成树上除根以外任意一顶点v，若顶点v的所有子结点中存在一个子结点 wi，wi及其子孙都没有指向v祖先结点的回边，则顶点v必为关结点。

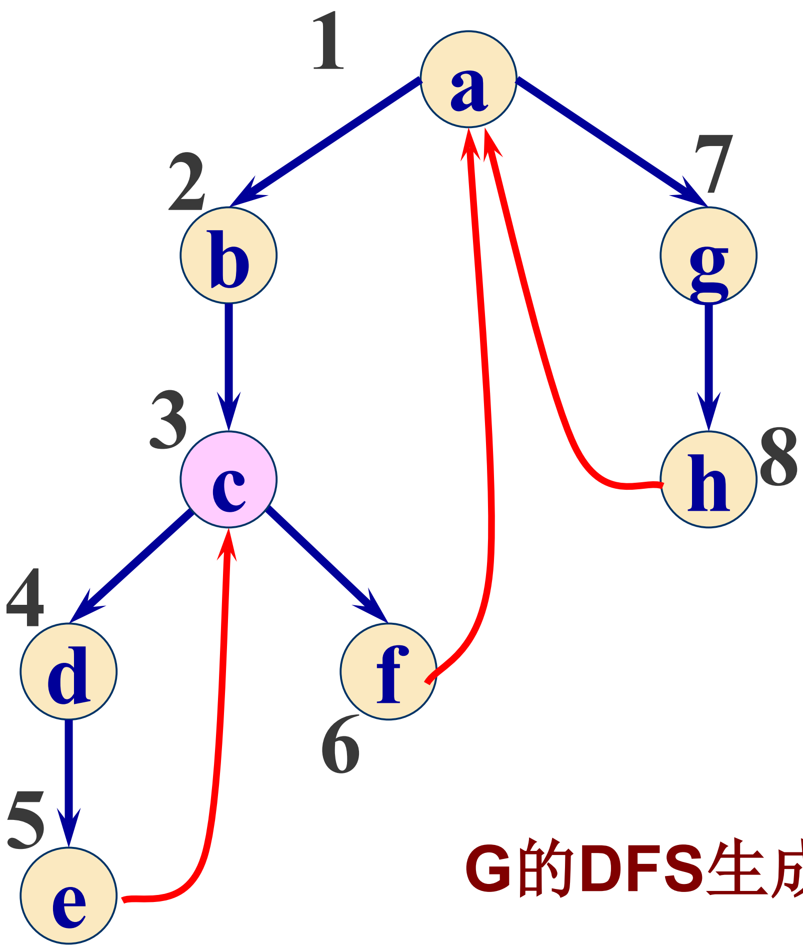
◆ 如何判断结点满足特征2？



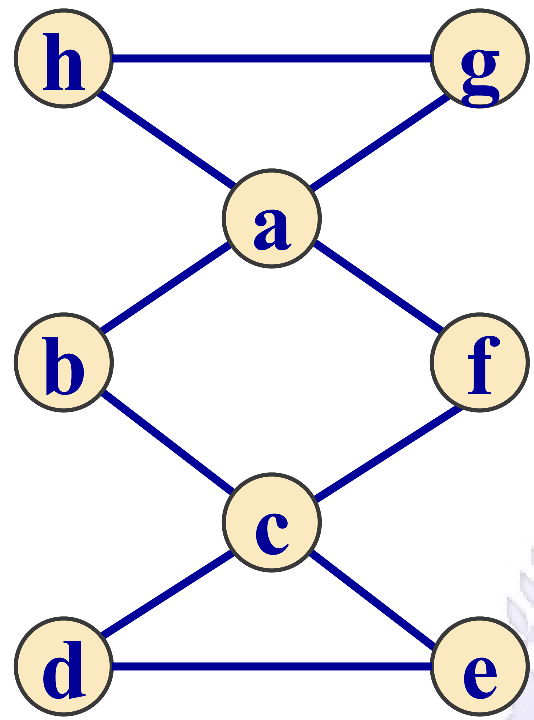
G的DFS生成树T

特征2：对生成树上除根以外任意一顶点v，若顶点v的所有子结点中存在一个子结点 wi，wi及其子孙都没有指向v祖先结点的回边，则顶点v必为关结点。

设置深度优先数visited[v]: 顶点在DFS中的序号；



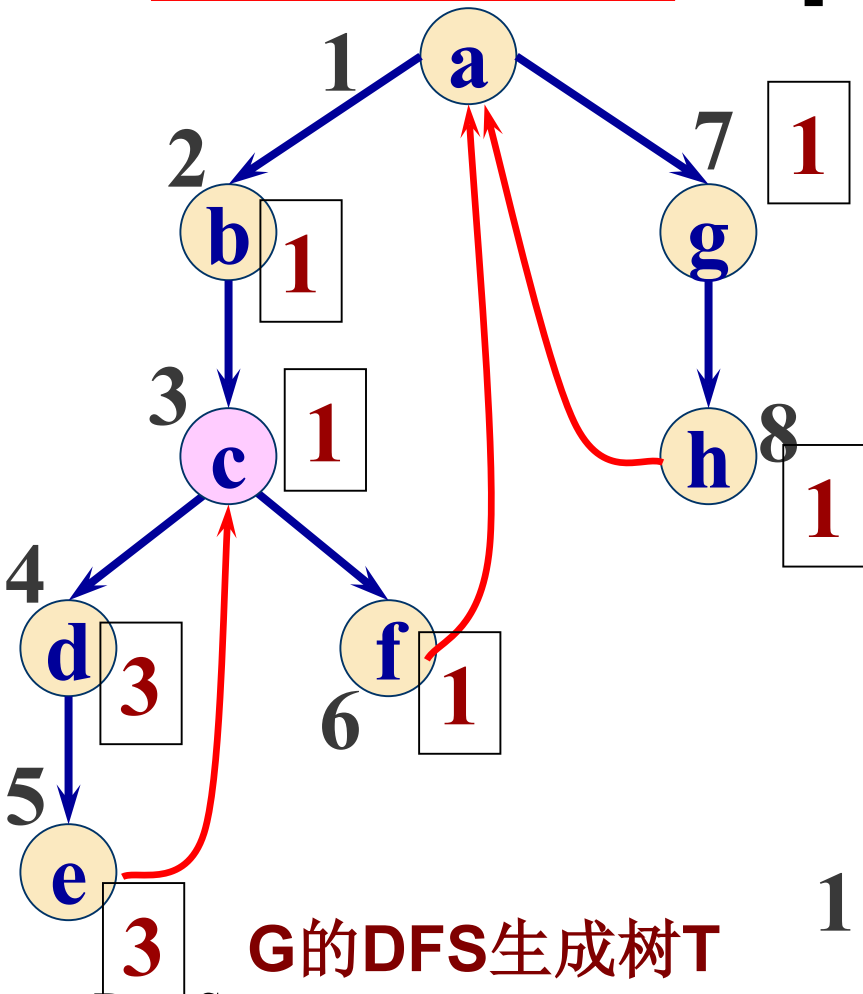
G的DFS生成树T



特征2：对生成树上除根以外任意一顶点v，若顶点v的所有子结点中存在一个子结点 wi，wi及其子孙都没有指向v祖先结点的回边，则顶点v必为关结点。

◆ 定义最低深度优先数low[v]

$$=\mathrm{Min}\{\ visited[v],\ low[w],\ visited[k]\ \}$$



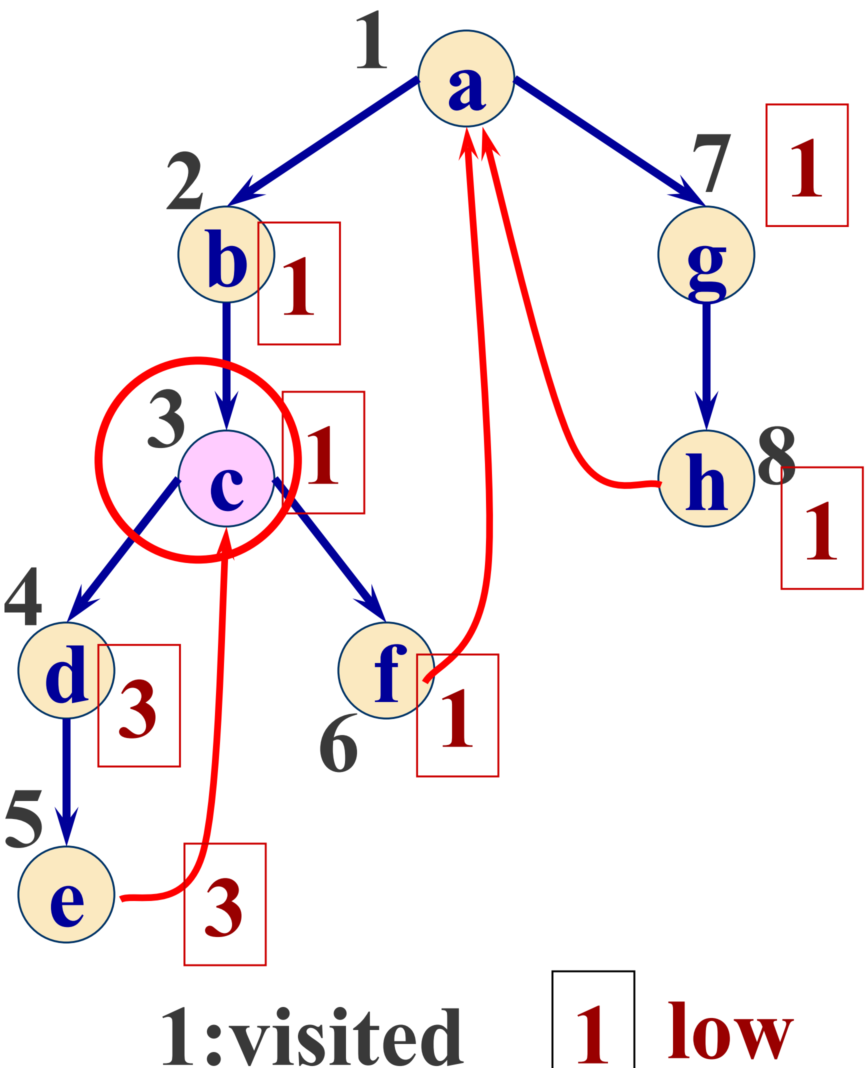
G的DFS生成树T

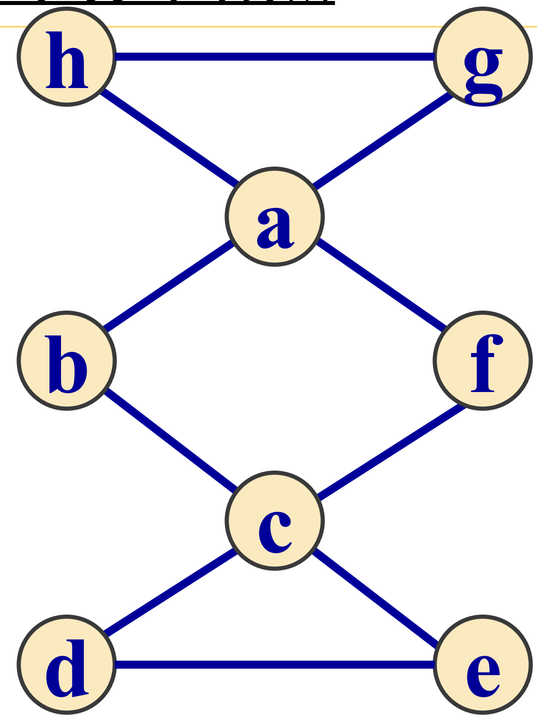
w是v在T中的子结点；
k 是v在T中回联的祖先结点；

low[v]:后序遍历T过程中求得

1:visited　　1 low

特征2：对生成树上除根以外任意一顶点v，若顶点v的所有子结点中存在一个子结点 wi，wi及其子孙都没有指向v祖先结点的回边，则顶点v必为关结点。





特征2的判断条件：

visited[v] <= low[w];

w是v在T中的子结点；

本章内容

- 6.1 抽象数据类型图的定义
- 6.2 图的存储表示
- 6.3 图的遍历
- 6.4 最小生成树
- 6.5 最短路径
- 6.6 拓扑排序
- 6.7 关键路径

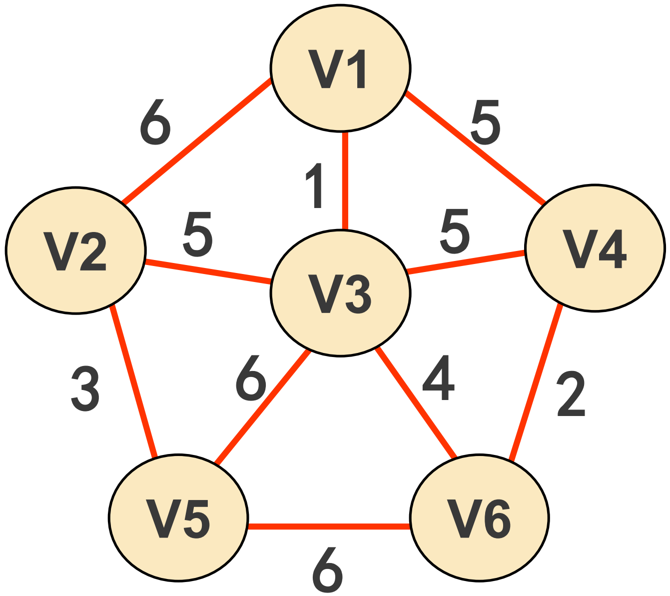
6.4 (连通网的)最小生成树

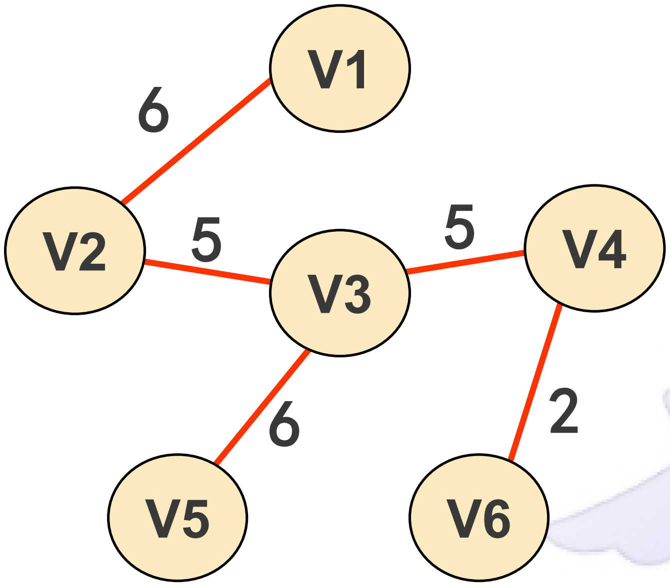
- **问题：**
  - 假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 n-1条线路，如何在最节省经费的前提下建立这个通讯网？
- **该问题等价于：**
  - 构造网的一棵最小生成树
  - 即：在 e 条带权的边中选取 n-1 条边（不构成回路），使“权值之和”为最小。

6.4 (连通网的)最小生成树

- 生成树(Spanning tree)：包含无向连通图G所有顶点的极小连通子图称为G生成树。

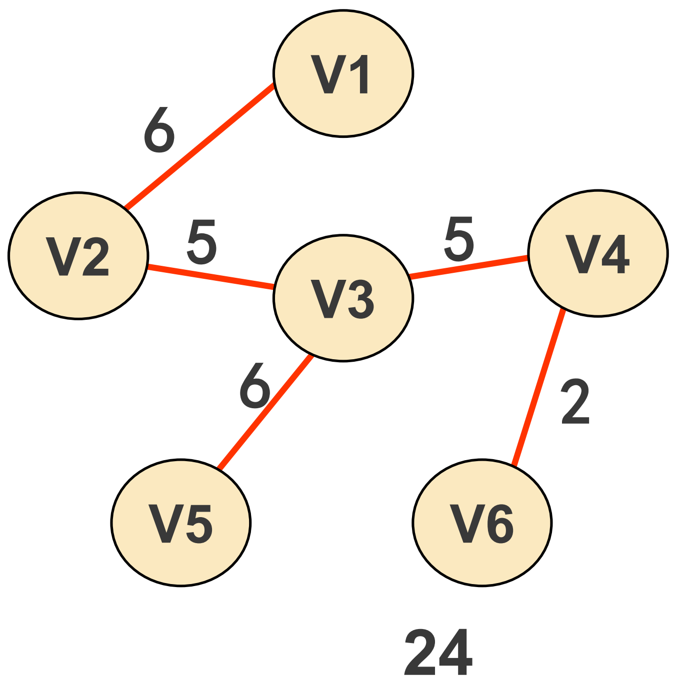
- 特点：1) T是G的连通子图 2) T包含G的所有顶点
  3) T中无回路 4) T中有n-1 条边

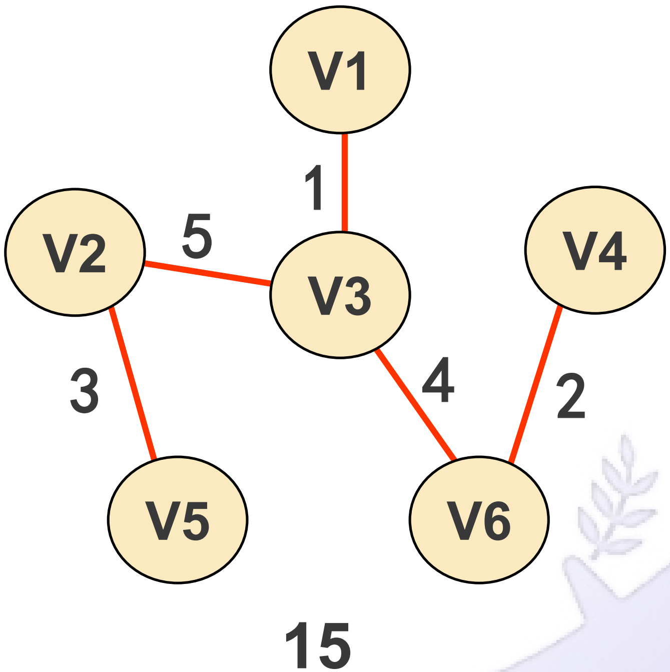




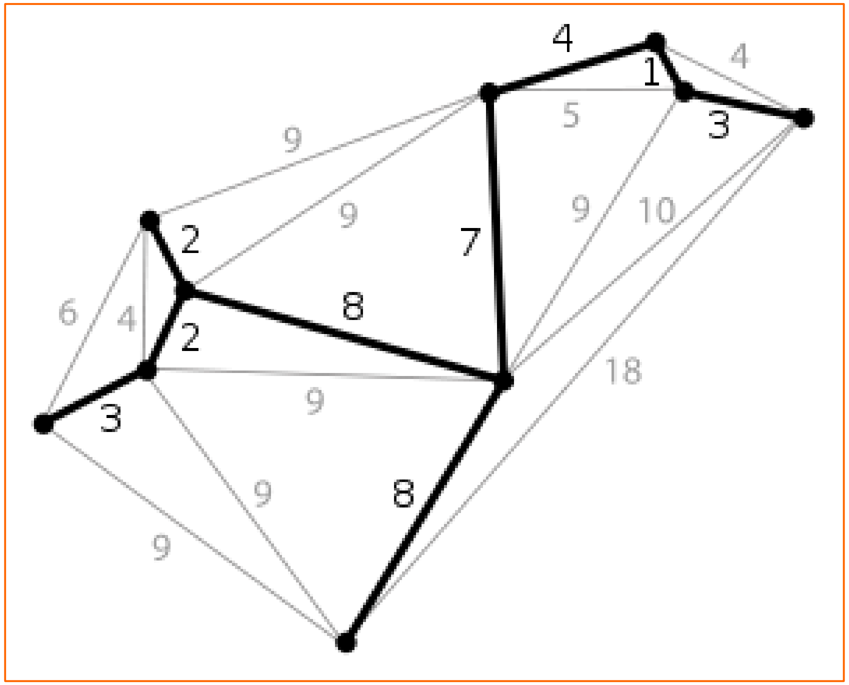
6.4 (连通网的)最小生成树

- 最小生成树(Least weighted spanning tree)
- Minimum spanning tree(MST)
- 权（之和）最小的生成树。





6.4 (连通网的)最小生成树



http://en.wikipedia.org/wiki/Minimum_spanning_tree

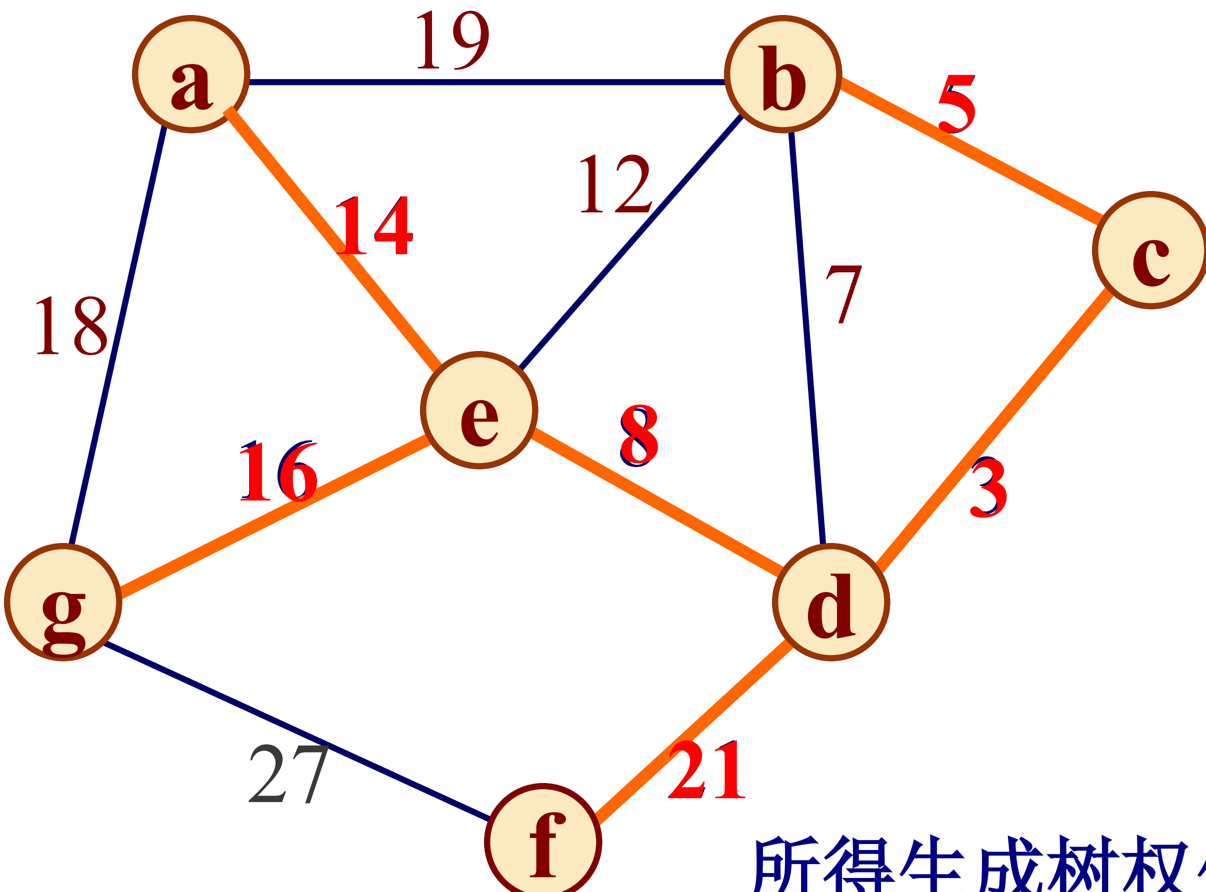
6.4 (连通网的)最小生成树

- Borůvka算法
  - 第一个算法：
  - 捷克科学家 Otakar Borůvka in 1926
- 普里姆（Prim）算法
  - 于1930年由捷克数学家沃伊捷赫·亚尔尼克（Vojtěch Jarník）发现；
  - 并在1957年由美国计算机科学家Robert C. Prim独立发现；
  - 1959年，荷兰Edsger Dijkstra再次发现了该算法。
  - 因此，在某些场合，普里姆算法又被称为DJP算法、亚尔尼克算法或普里姆－亚尔尼克算法。

6.4.1 普里姆（Prim）算法

- 基本思想：设带权连通图$N=(V, E)$.
- 设$T=(V_{mst}, E_{mst})$是当前的最小生成树。
  - 从的某一个顶点u_0开始
  - 将u_0加入当前最小生成树T的顶点集合V_{mst}。
  - 循环，每次向T中增加一个结点：
    - 从与T的结点相连的所有边中（不计结点内部的边），找到最短的边，将该边及相应结点加入T中。
    - 即从跨越$V-V_{mst}$和V_{mst}两个部分的所有边中选择权值最小边
  - 直到所有的结点都加入T中为止

例如:



所得生成树权值和

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

6.4.1 普里姆算法

- 设G=(V,E)为一个具有n个顶点的连通网络，
- T=(V_{mst}, E_{mst})为构造的生成树。
  - 1）初始时，V_{mst}={u0}，$E_{mst}=\phi$;
  - 2）在所有u∈V_{mst}、v∈V-V_{mst}的边(u, v)中选择一条权值最小的边，不妨设为(u, v);
  - 3）将v加入U；同时(u, v)加入E_{mst}，
  - 4）重复2)、3)，直到V_{mst}=V为止；

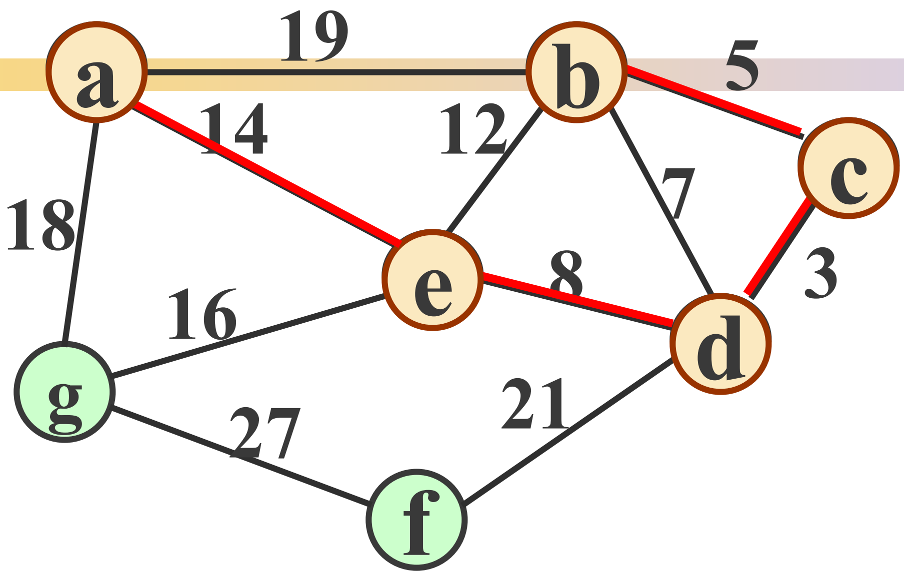
6.4.1 普里姆算法

- 设置一个两个辅助数组：对当前$V-V_{mst}$集中的每个顶点，记录顶点集V_{mst}中与该顶点相连接代价最小的边：

```
Weight lowcost[ maxVertics];// 该边的权值
int nearvex[maxVertics];//该点邻接的在Vmst中的顶点序号
```

```
typedef struct { //最小生成树边结点
        int v1, v2;                //两顶点的顶点号
        WeightType key;            //边上的权值
} MSTEdgeNode;
typedef struct {//最小生成树的定义
        MSTEdgeNode edgeValue[maxSize];   //边值数组
        int n;                            //数组当前元素个数
} MinSpanTree;
```

例如:



	a	b	c	d	e	f	g
	0	**1**	**2**	**3**	**4**	**5**	**6**
nearvex		c	d	e	a	d	e
lowcost	-1	-1	-1	-1	-1	21	16

```
void MiniSpanTree_P(MGraph G, VertexType u) {
 //用普里姆算法从顶点u构造连通网G的最小生成树
  for ( j=0; j<G.numVertices; ++j ){// 辅助数组初始化
        lowcost[j] = G.Edge[u][j];
        nearvex[j] = u;
  }
  lowcost[u]= -1;      // 初始时Emst＝{u}
  for (i=0; i<G.numVertices-1; ++i) {
        依次向生成树上添加n-1个顶点;
  }
}// MiniSpanTree_P
```

```
for (i=0; i<G.numVertices-1; ++i) {
   //依次向生成树上添加n-1个顶点;
   v = minimum(lowcost);   // 求出加入生成树的下一个顶点(v)
   lowcost[v] = -1;  // 顶点v加入生成树
   for (j=0; j<G.numVertices; ++j) //修改其它顶点的最小边
       if (G.Edge[v][j]< lowcost[j]){
           lowcost[j] = G.Edge[v][j];
           nearvex[j] = v;
       }//if
   } //for (j=0; ...
}//for (i=0;...
```

复杂度:$O(n^2)$

与边数e无关

适用于稠密图

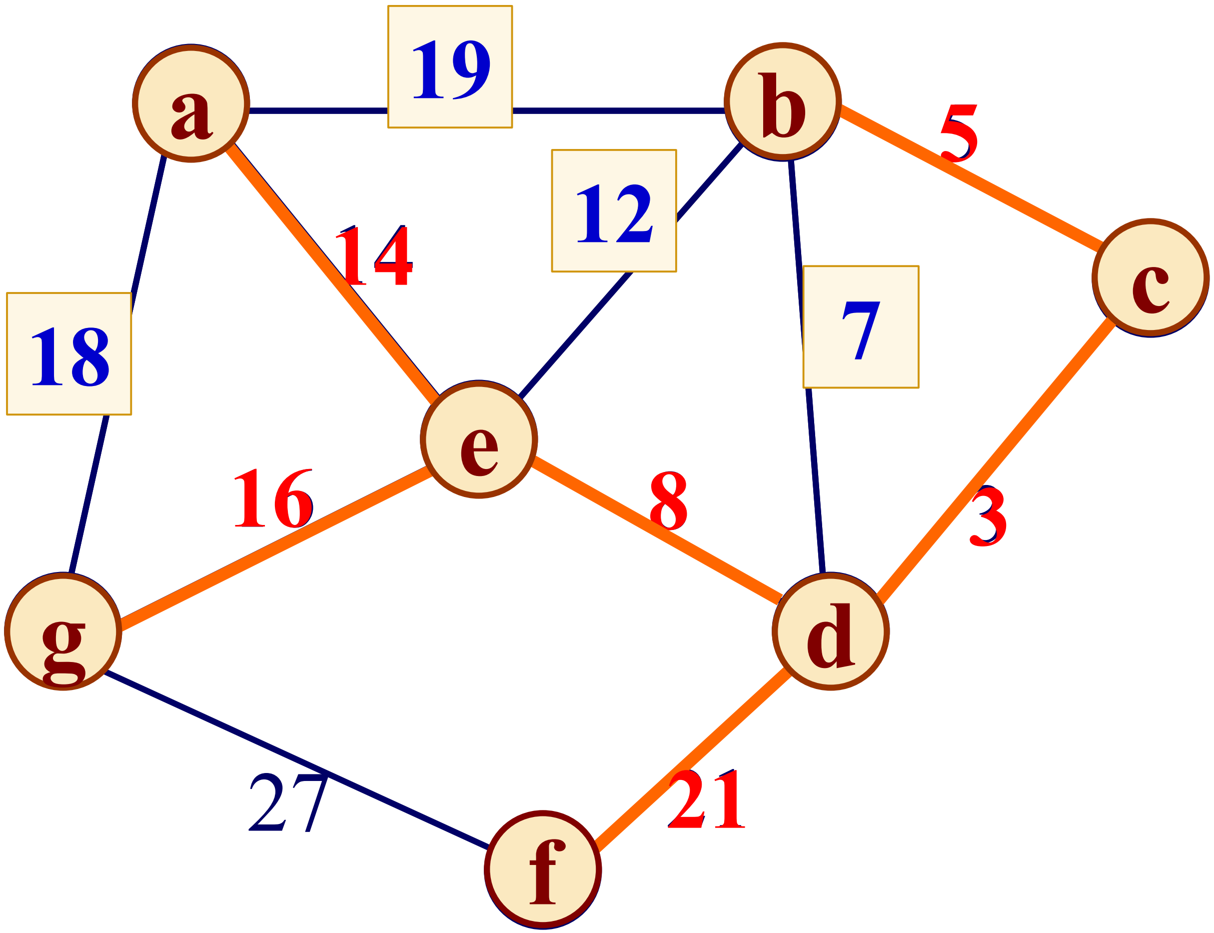
6.4.2 克鲁斯卡尔(Kruskal)算法

- 基本思想

1. 初始时最小生成树只包含图的n个顶点，每个顶点构成一个连通分量；
2. 选取权值较小且所关联的两个顶点不在同一连通分量的边，将此边加入到最小生成树中；
3. 重复第2步n-1次，即得到包含n个顶点和n-1条边的最小生成树。

6.4.2 克鲁斯卡尔(Kruskal)算法

例如:



6.4.2 克鲁斯卡尔算法

构造非连通图 ST=(V,{ });

k = 0;　// k 计选中的边数

while (k<n-1) {

　检查边集 E 中未标记边中权值最小的边(u,v);

　若(u, v)加入ST后不使ST中产生回路，则{

　　输出边(u, v); 标志(u, v)已被选择; k++; }

　否则标志(u, v)为内边;

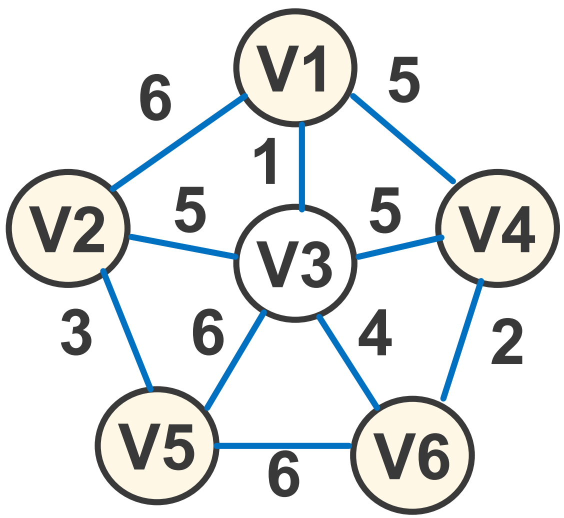
}

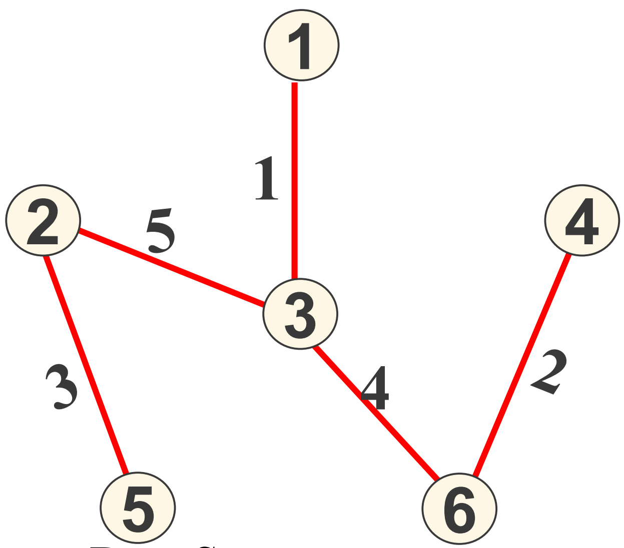
复杂度：O(eloge)	适合于稀疏图

算法至多对e条边各扫描一次。
假设选最小边用“堆”算法进行（其复杂度为loge）

6.4 图的最小生成树

克鲁斯卡尔算法的另一种实现:采用排序的边集数组保存图





	data	set
1	1	1
2	2	2 1
3	3	~~3~~ 1
4	4	~~4~~ 1
5	5	~~5~~ ~~2~~ 1
6	6	~~6~~ ~~4~~ 1

并查集

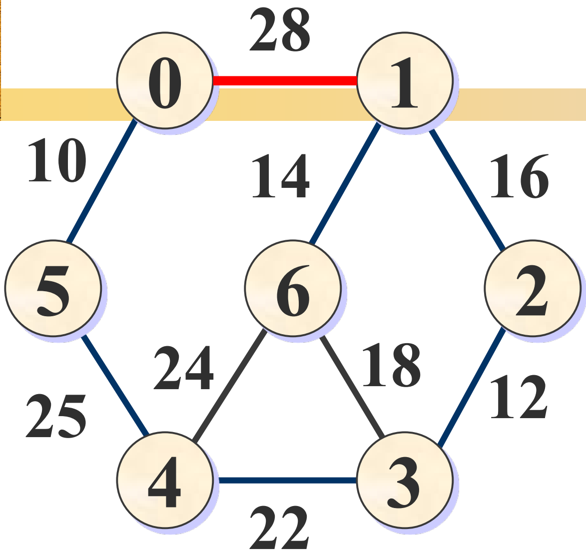
	v1	v2	w	flag
0	1	3	1	~~0~~ 1
1	4	6	2	~~0~~ 1
2	2	5	3	~~0~~ 1
3	3	6	4	~~0~~ 1
4	1	4	5	~~0~~ 2
5	2	3	5	~~0~~ 1
6	3	4	5	0
7	1	2	6	0
8	3	5	6	0
9	5	6	6	0

```
Status Kruskal_MST ( MSTEdgeNode EV[ ], int n, int e
        , MinSpanTree &T){
//从已经按权值排序的边数组中顺序取出边，建立最小生成树
    UFSets Vset; Inital(Vset); //初始化并查集
    InitMinSpanTree(T); //初始化MST
    j = 0; k = 0;//j记录加入MST中的边数; k记录当前扫描的边
    while(k < e && j < n) {
        u = Find( Vset, EV[k].v1); v = Find( Vset, EV[k].v2);
        if ( u != v ){//如果k不是内边，将k加入MST中
                T.edgeValue[T.n++] = EV[k];
                Merge( Uset, u, v);     j++;
        }//if ( u != v )
        k++;
    }//while(k < e)
    if ( j < n-1 ) return -1; else return 1;
}//Kruskal_MST
```

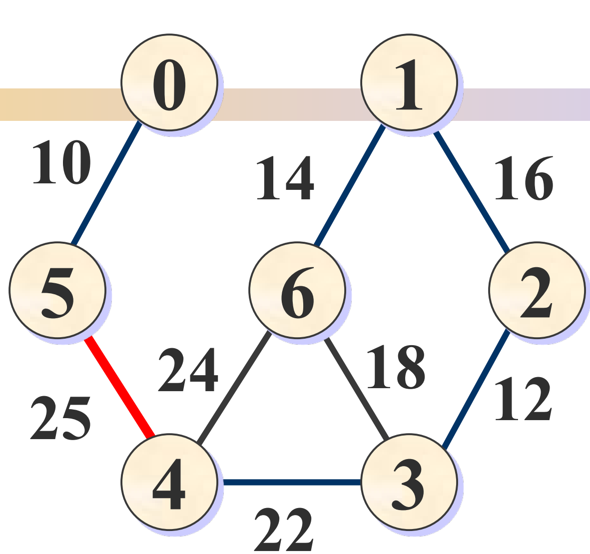
排序边数组，并查集复杂度：O(n+e)

其他方法1：破圈法

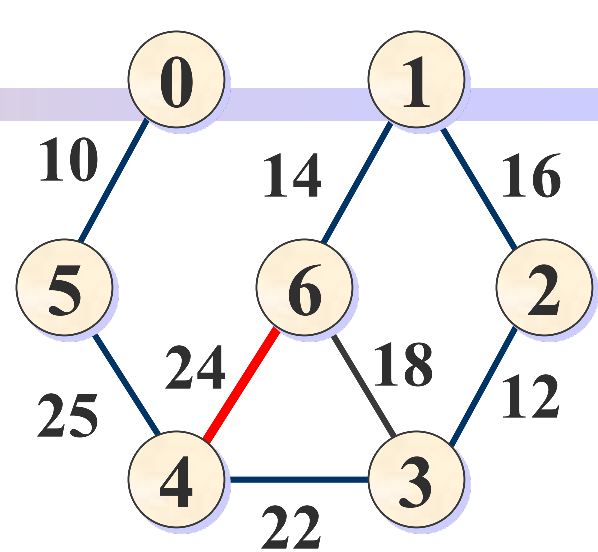
- **基本思想**：对一个有 n 个顶点的连通带权图，按其权值**从大到小顺序逐个删除各边，直到剩下n-1 条边**。
- **删除的原则是**：删除该边后各个顶点之间还是连通的。
- **判断图连通性的方法**：每删除一条权值最大的边，就调用 DFS 遍历图，如果能够访遍图中所有顶点则表明删除此边没有破坏图的连通性，此边可删，否则撤销删除，恢复此边。
- 若采用邻接矩阵存储图，DFS的时间代价$O(n^2)$，若采用邻接表，DFS的时间代价$O(n+e)$。



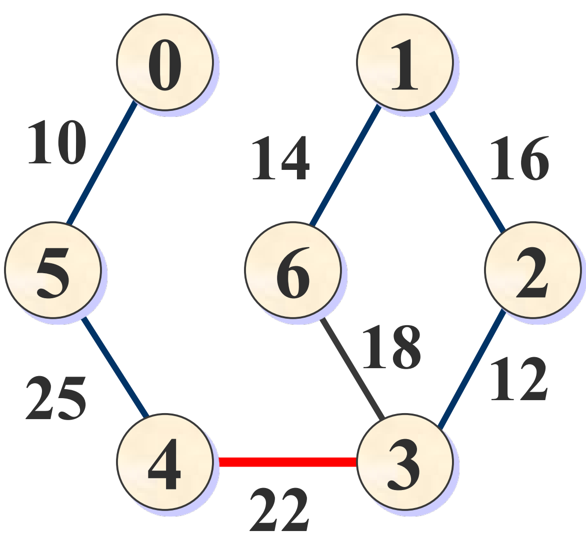
原图



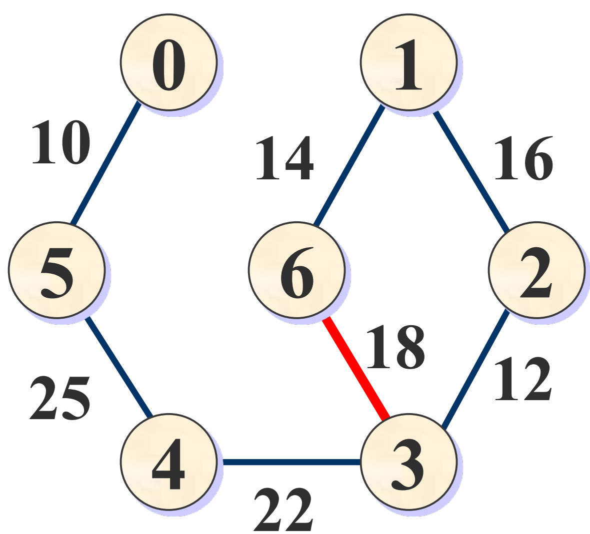
删除 (0, 1) 28



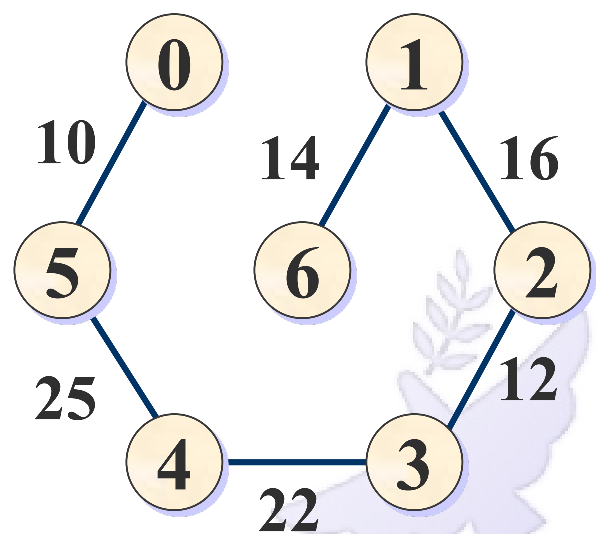
保留 (5, 4)24



删除 (6, 4) 24



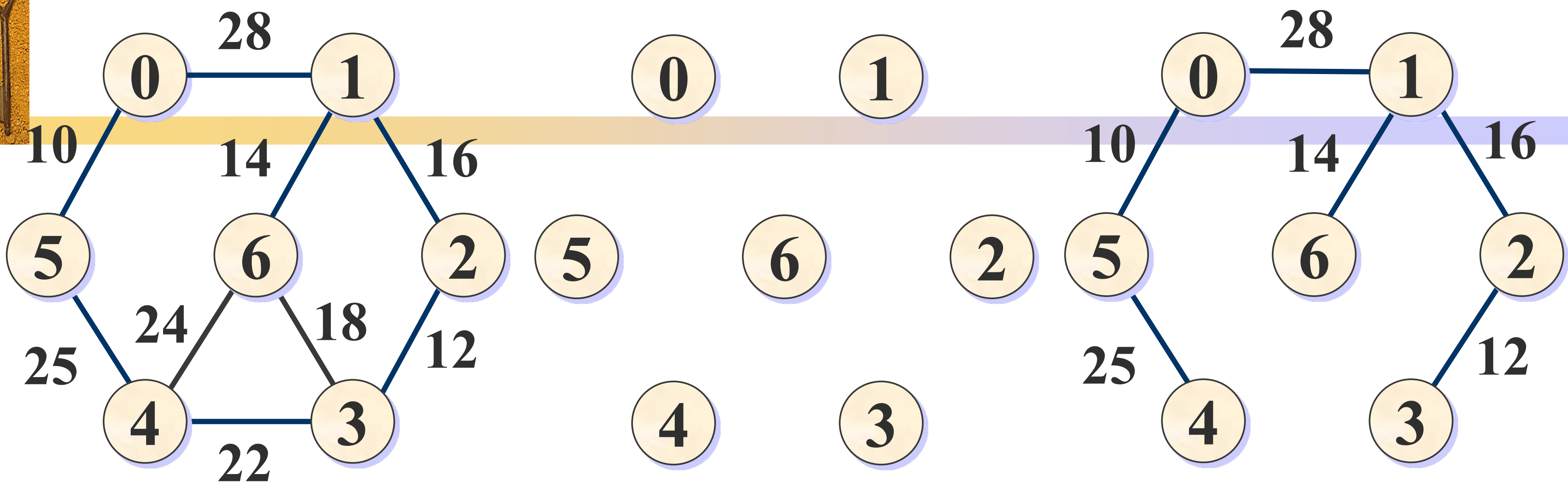
保留 (4, 3) 22



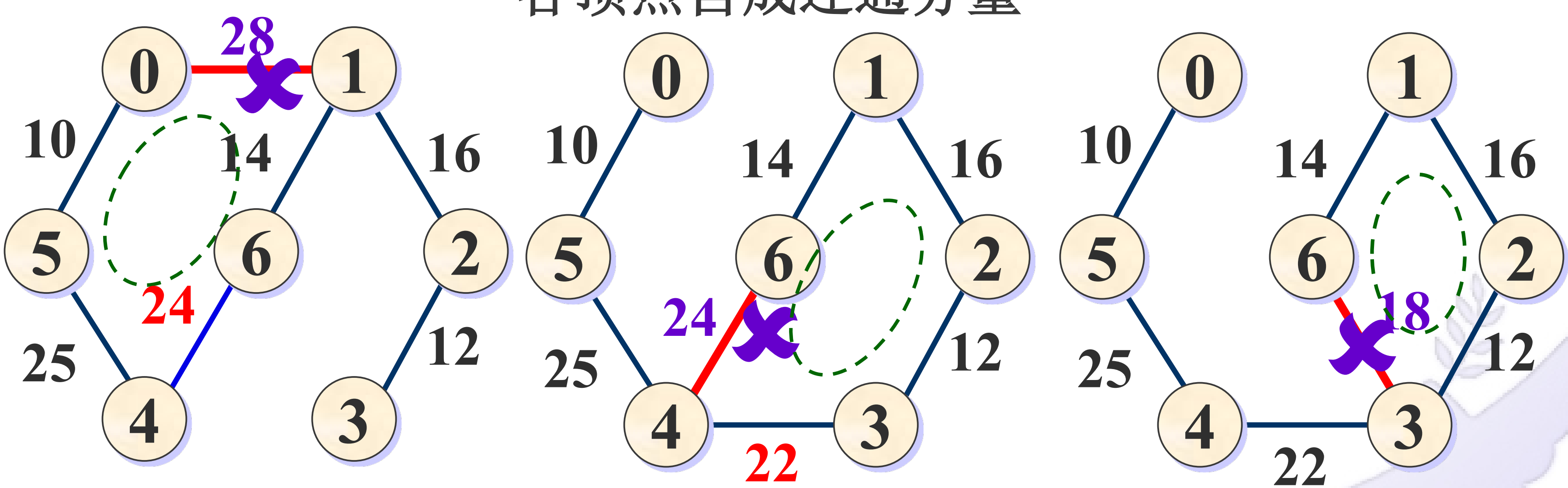
删除 (6, 3)18

其他方法2：迪杰斯特拉(Dijkstra)算法

- 基本思想：
- 1、将带权图中每个顶点视为一个独立的连通分量；
- 2、然后逐个检查各条边，直到所有边都检查完为止：
  - 2.1 如果该边的两个端点在不同的连通分量上，直接把它加入生成树；
  - 2.2 如果该边的两个端点在同一个连通分量上，加入它后会形成一个环，把该环上权值最大的边删除。
- 在此算法中，边的检查顺序没有限制，只要出现圈就破圈。为此又用到了并查集。



各顶点自成连通分量



加入(6,4)后,删(0,1)　加入(4,3)后,删(6,4)　加入(6,3)后,删(6,3)

6.5 最短路径问题

- 问题描述：
- 在有向网络上，规划从起点到终点的“最短”路径。
- 规划目标（“最短”）：
  - 距离最短
  - 费用最少
  - 油耗最少
  - 安全性最高
  - ……

6.5 最短路径问题

- 迪杰斯特拉(Dijkstra)算法（非负权值）
  - 求从某个源点到其余各顶点的最短路径
  - 即，对已知图 G = (V, E)，给定源顶点 s∈V，找出 s 到图中其它各顶点的最短路径。
- Bellman-Ford算法（允许负权值）
  - 求从某个源点到其余各顶点的最短路径
- 弗洛伊德(Floyd)算法（允许负权值，不允许负回路）
  - 求每一对顶点之间的最短路径
  - 即，对已知图 G = (V, E)，任意的顶点 Vi，Vj∈V，找出从 Vi 到 Vj 的最短路径。

6.5.1 迪杰斯特拉(Dijkstra)算法

- 荷兰科学家Edsger Dijkstra 1956提出并在1959发表。
- 解决非负权图中的最短路径问题
- 基本思想：
  - 图的广度优先搜索
  - 贪心算法

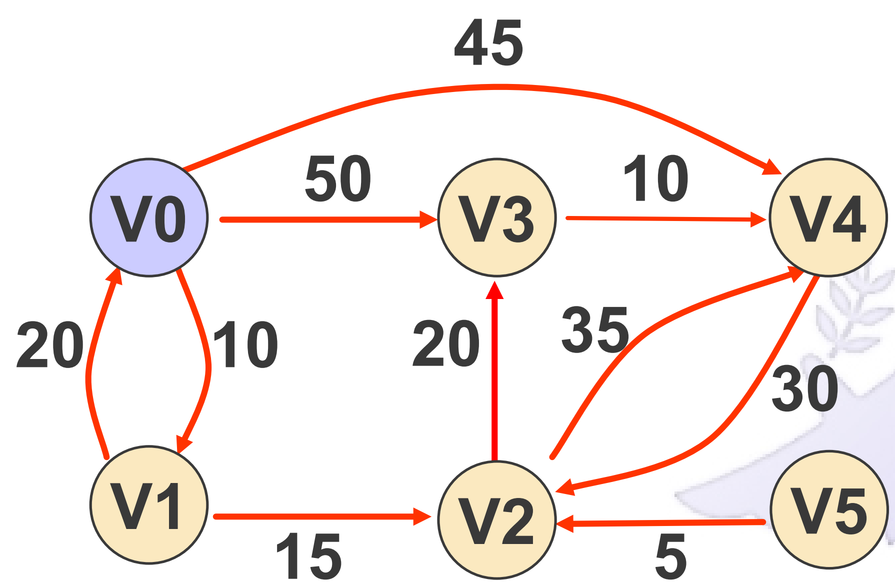
6.5.1 迪杰斯特拉(Dijkstra)算法

- **问题**：求从某个源点到其余各点的最短路径
- **基本思想**：
  - 依最短路径的长度递增的次序求得各条路径
- 设置辅助数组dist[n-1]
  - dist[k] 表示从源点V0到顶点Vk最短路径的长度

当前

dist[n-1]

V1	V2	V3	V4	V5
10	∞	50	45	∞

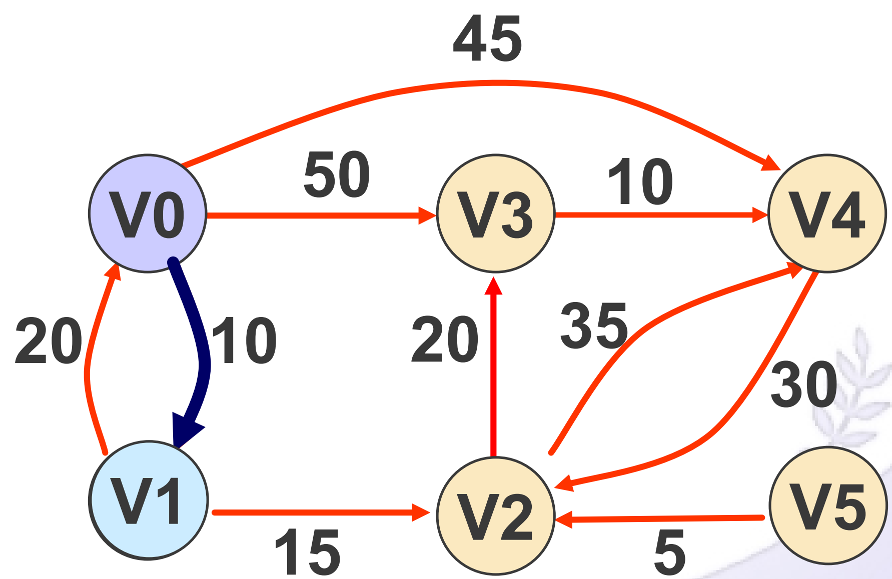


6.5.1 迪杰斯特拉(Dijkstra)算法

- 在dist[n-1]中，长度最短的路径的特点：
  - 必定只含一条弧，且这条弧是始于V0的弧中权值最小的，设最短路径的终点为V1

dist[n-1]

V1	V2	V3	V4	V5
10	∞	50	45	∞

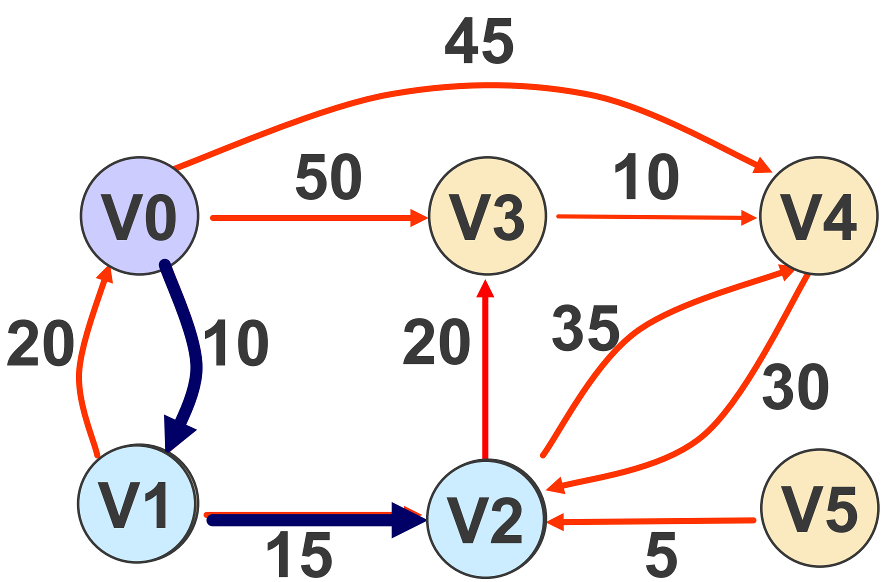


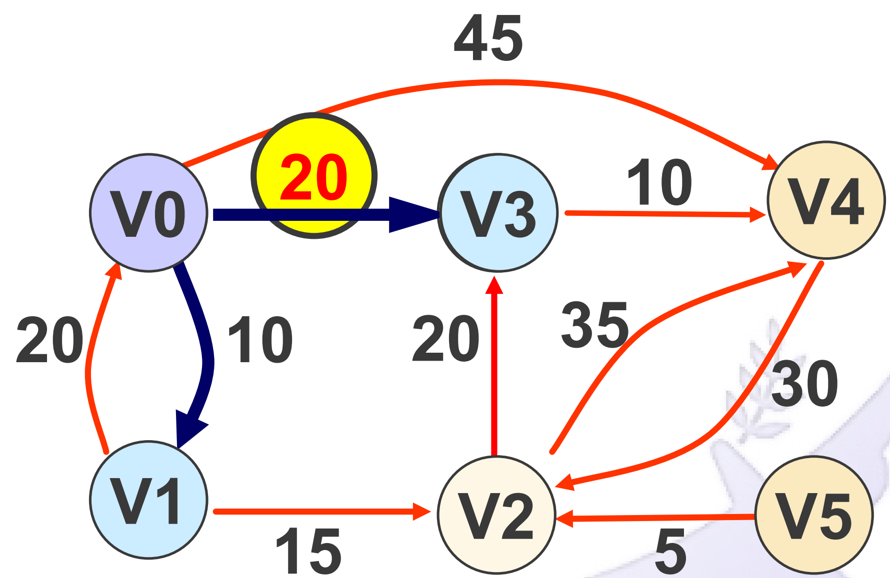
6.5.1 迪杰斯特拉(Dijkstra)算法

- 长度次短的最短路径的特点

—1）是直接从源点到该点（记为V2） (只含一条弧);

—2）从源点经过顶点v1，再到达该顶点 (由两条弧组成)





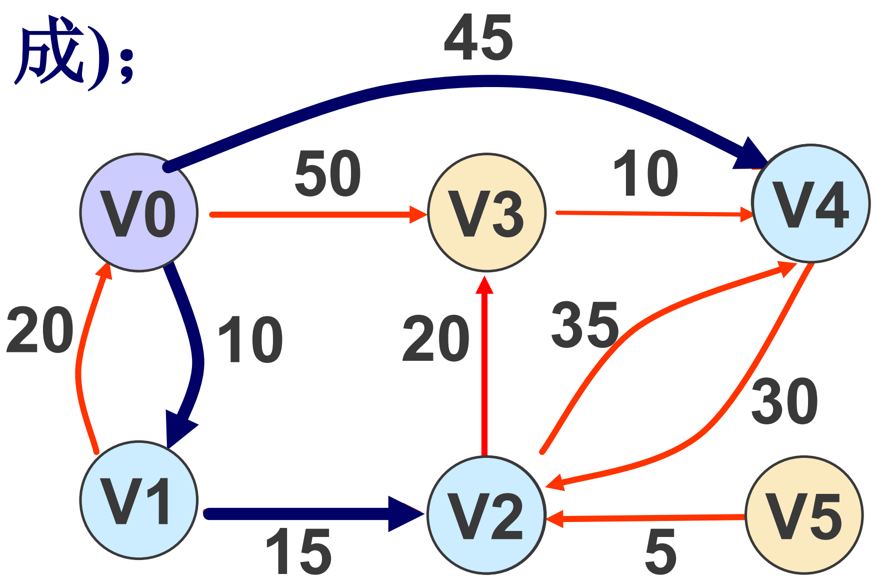
6.5.1 迪杰斯特拉(Dijkstra)算法

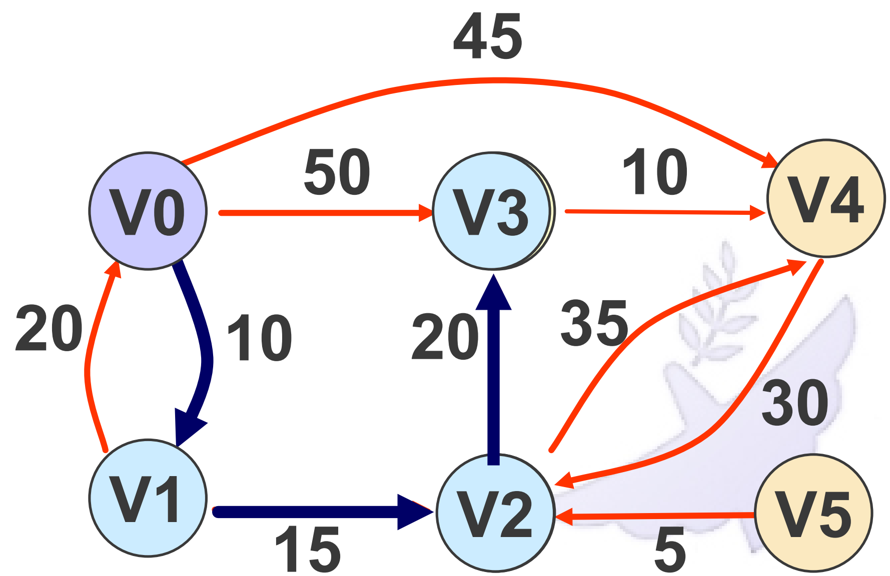
- ♦ 再下一条路径长度次短的最短路径的特点

➢1）直接从源点到该点（记为V3）(只含一条弧);

➢2）从源点经过顶点v1，再到达该顶点(由两条弧组成);

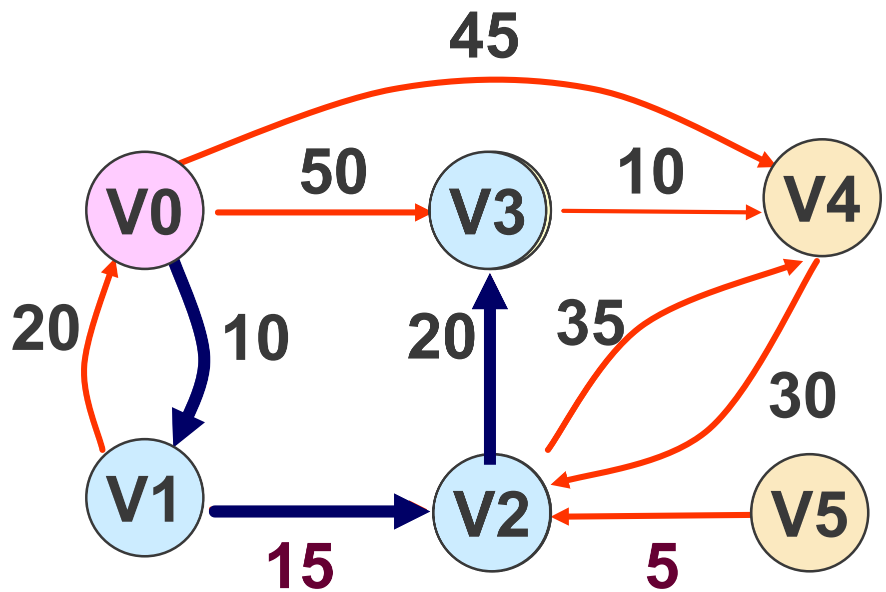
➢3）从源点经过顶点v2，再到达该顶点, (由三条弧组成);





6.5.1 迪杰斯特拉(Dijkstra)算法

- 其余最短路径的特点
  - 1）直接从源点到该点(只含一条弧)；
  - 2）从源点经过已求得最短路径的顶点，再到达该顶点



6.5.1 迪杰斯特拉算法

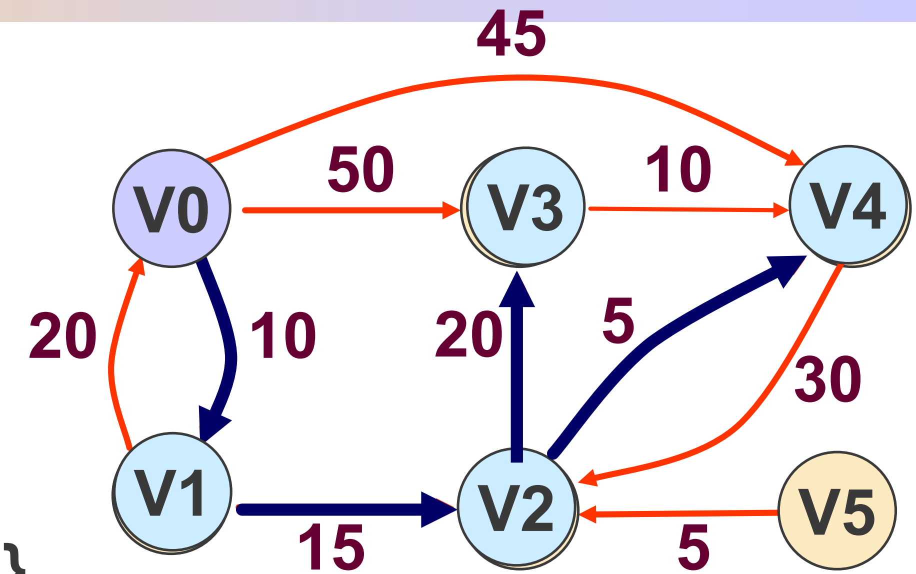
- 辅助集合S：
  - 当前已经得到最短路径的顶点集合
  - 初始时，S={V0}
- 辅助数组dist[]：表示“当前”所求得的从源点到顶点 k 的最短路径
  - dist[k] = <源点到顶点 k 的弧上的权值>
  - 或者dist[k] = <源点到顶点j的路径长度> +

    \+ <顶点j到顶点 k 的弧上的权值>
- 辅助数组Path[k]: 顶点k当前最短路径的前驱结点

6.5.1迪杰斯特拉算法

- 1. 初始化数组dist[ ]，
  - V0和k之间存在弧： dist[k]＝G.Edge[0][k]
  - V0和k之间不存在弧： dist[k]＝无穷
- 2. 循环
- 2.1 在dist[ ]中选取一条权值最小的弧，即得到一条最短路径。
- 2.2 依次修改其它未得到最短路径顶点的dist[k]值。
  - 假设最近求得最短路径的顶点为u，
  - 则 dist[k] ＝min(dist[k], dist[u] + G.Edge[u][k])

6.5.1迪杰斯特拉算法



Path[k]

0	0	1	2	2	-1

V0	V1	V2	V3	V4	V5	S
0	∞	∞	∞	∞	∞	{V0}
0	10	∞	50	45	∞	{V0,V1}
0	10	25	50	45	∞	{V0,V1,V2}
0	10	25	45	30	∞	{V0,V1,V2,V4}
0	10	25	45	30	∞	{V0,V1,V2, V4,V3}

dist[n-1]　　　　S

6.5.1迪杰斯特拉算法

```
void ShortestPath ( MGraph& G, int v,  Weight dist[ ], int path[ ] )
{ //求带权有向图G中从源点v到其他顶点的最短路径
   int S[maxVertices];           //最短路径顶点集
   n = G.verticeNum;
   for ( i = 0; i < n; i++ ) {    //初始化数组S、path和dist
      dist[i] = G.Edge[v][i];  S[i] = 0;
      if ( dist[i] < maxWeight) path[i] = v;
      else path[i] = -1; } //for
   path[v] = v;   S[v] = 1;  dist[v] = 0;        //顶点v加入S集合
   for ( i = 0; i < n-1; i++ ) {  //求到其他点最短路径
         求到其他点最短路径
   }//for ( i = 0; ...)
}//ShortestPath
```

6.5.1迪杰斯特拉算法

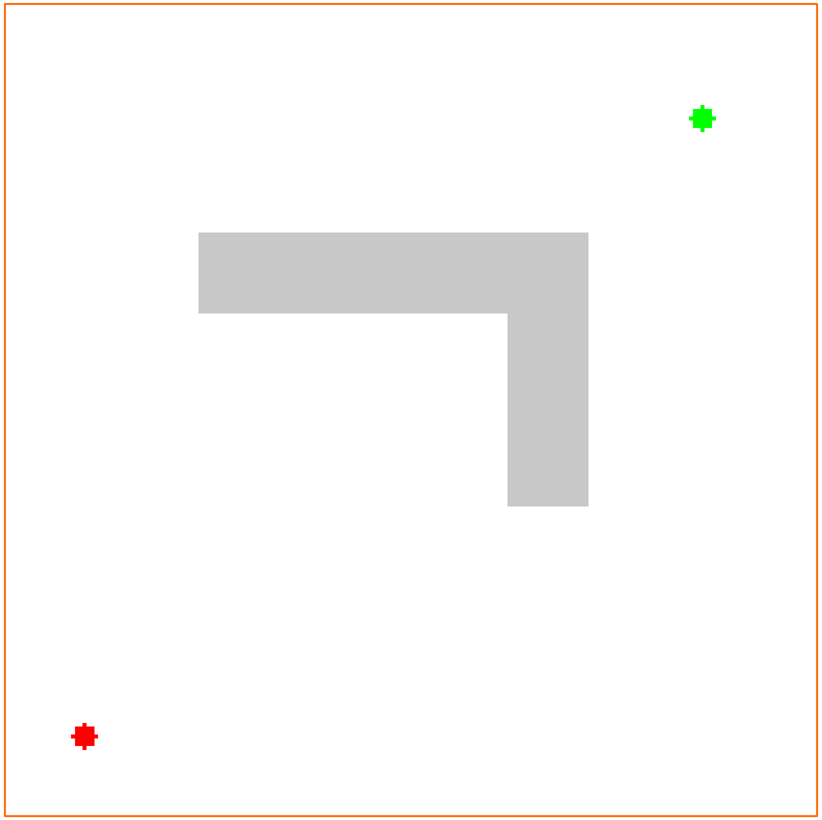
```
for ( i = 0; i < n−1; i++ ) {//求到其他点最短路径
    u = SelectMin(dist); //选不在S中具有最短路径顶点u
    S[u] = 1;              //将顶点u加入集合S
    for ( k = 0; k < n; k++ ) {//修改其他顶点的路径长度，松弛
        if ( !S[k] &&  dist[u] + G.Edge[u][k] < dist[k] ) {
                //顶点k未加入S, 且v到k经过u的路径更短
            dist[k] = dist[u] + G.Edge[u][k];
            path[k] = u;          //修改到k的最短路径
        }   //if ( !S[k] &&  ...
    }  //for ( k = 0; ...
}//for ( i = 0; ...)
```

时间复杂度？

邻接矩阵，扫描法：

$O(n^2)$

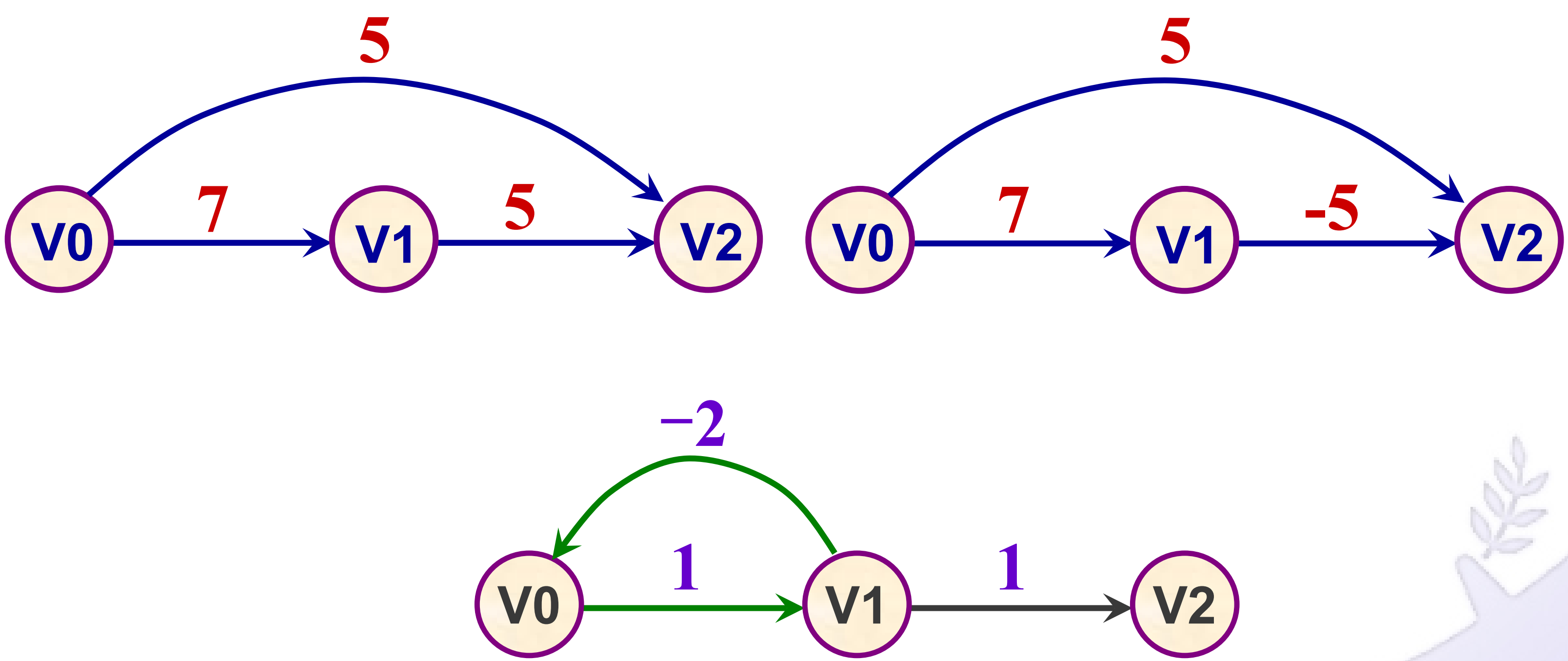
6.5.1迪杰斯特拉算法



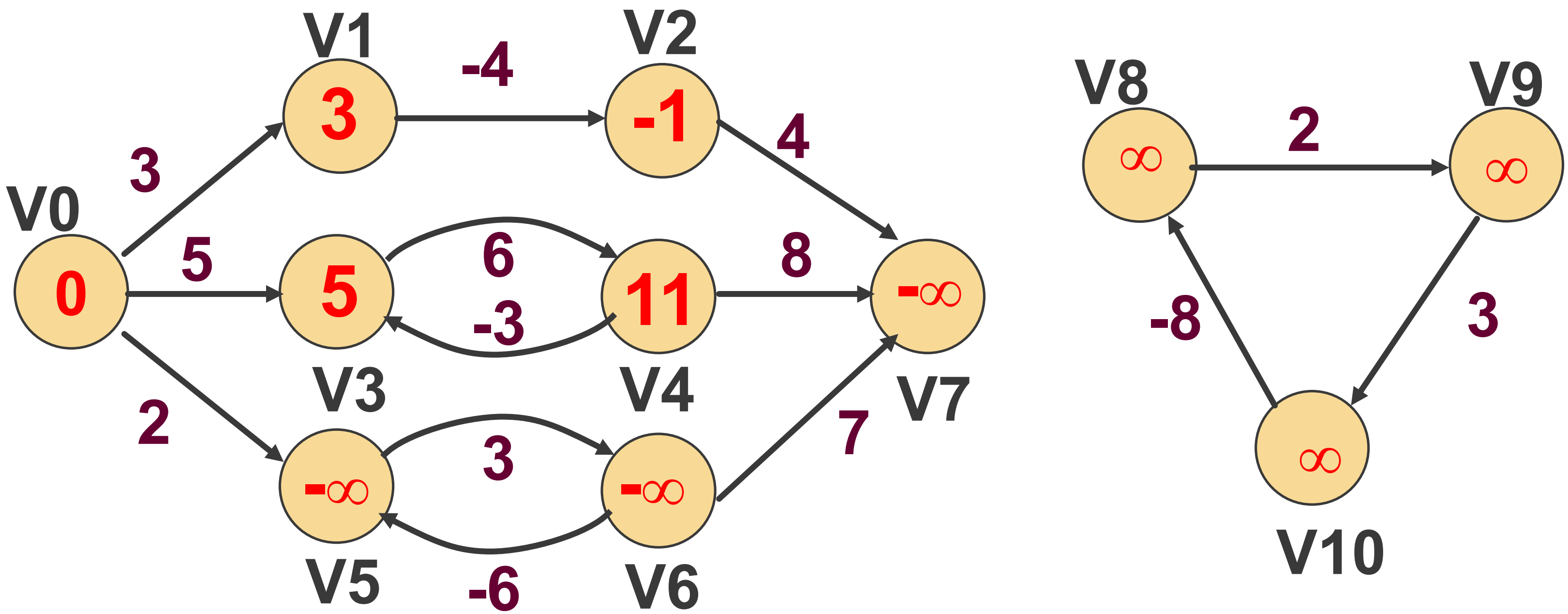
http://zh.wikipedia.org/wiki/%E8%BF%AA%E7%A7%91%E6%96%AF%E5%BD%BB%E7%AE%97%E6%B3%95

有负权重边的情况

- 带权有向图的某几条边或所有边的长度可能为负值。用**Dijkstra**算法不一定能得到正确的结果。



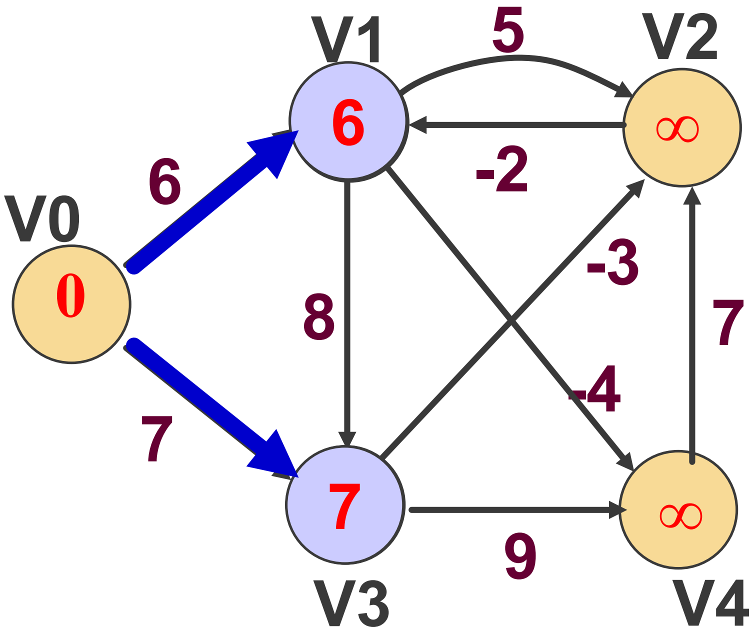
有负权重边的情况



- 最短路径上包括回路吗?
  - 负回路—最短路径无定义
  - 正回路—不可以
  - 零回路—可以找到不包括回路的简单路径

6.5.2 Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法



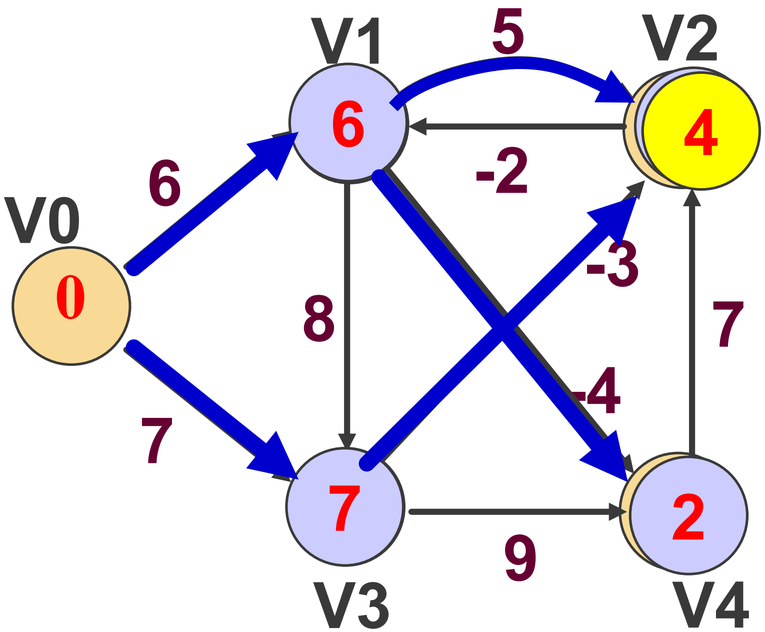
V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞

第1次考察边

V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

6.5.2 Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法



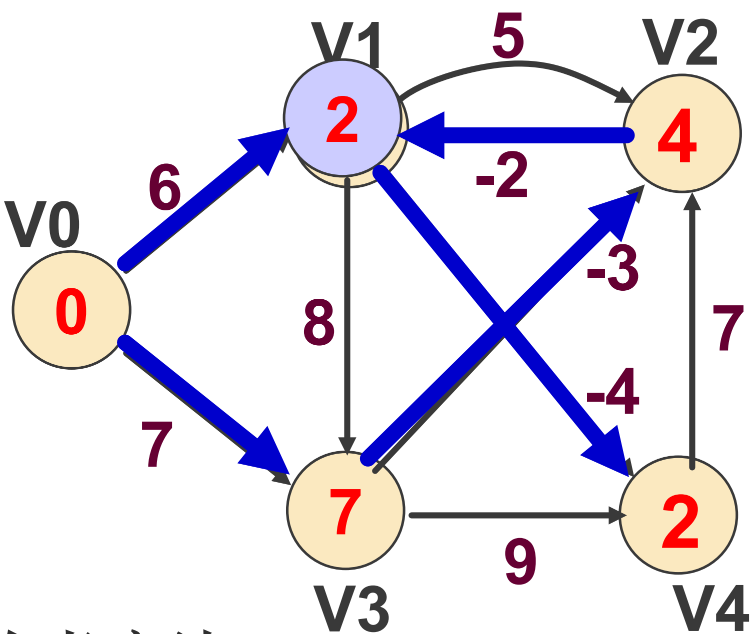
V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2

第2次考察边

V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

6.5.2 Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法



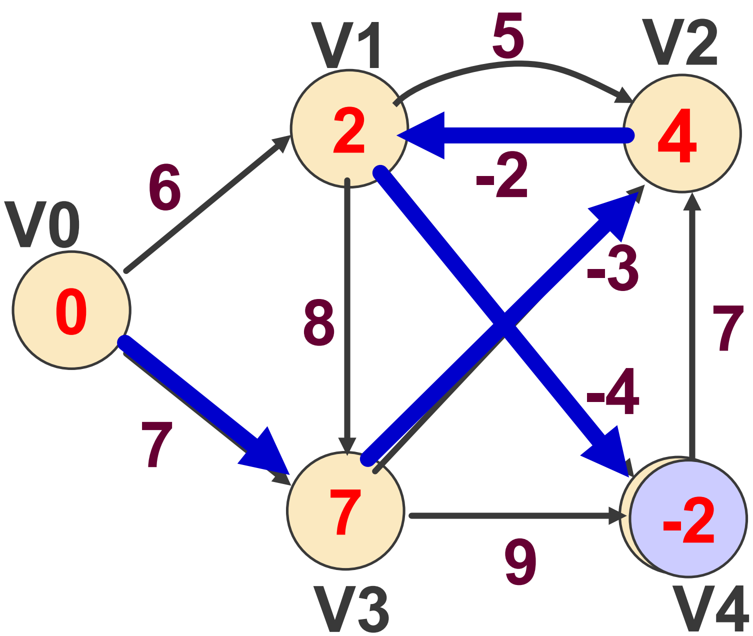
V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2

第3次考察边

V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

6.5.2 Bellman-Ford算法

- 单源最短路径问题，顶点V0到其它顶点间最短路径
- 基本思想：逐条边试探法　　　　动态规划

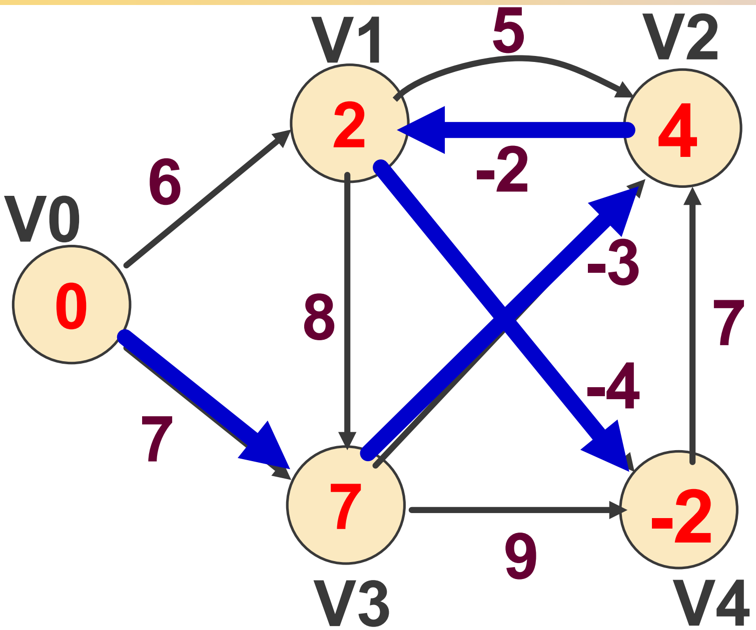


V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

第4次考察边

V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

6.5.2 Bellman-Ford算法（简单证明）



V0	V1	V2	V3	V4
0	∞	∞	∞	∞
0	6	∞	7	∞
0	6	4	7	2
0	2	4	7	2
0	2	4	7	-2

1：V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

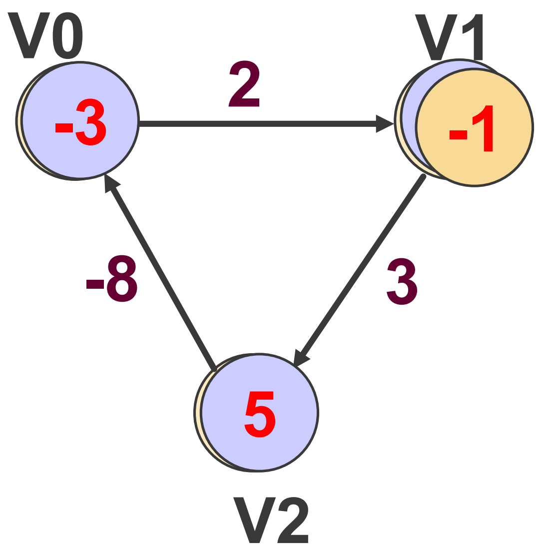
2：V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

3：V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

4：V1V2, V1V3, V1V4, V2V1, V3V2, V3V4, V4V2, V0V1, V0V3

6.5.2 Bellman-Ford算法

- 图中有负回路的情况



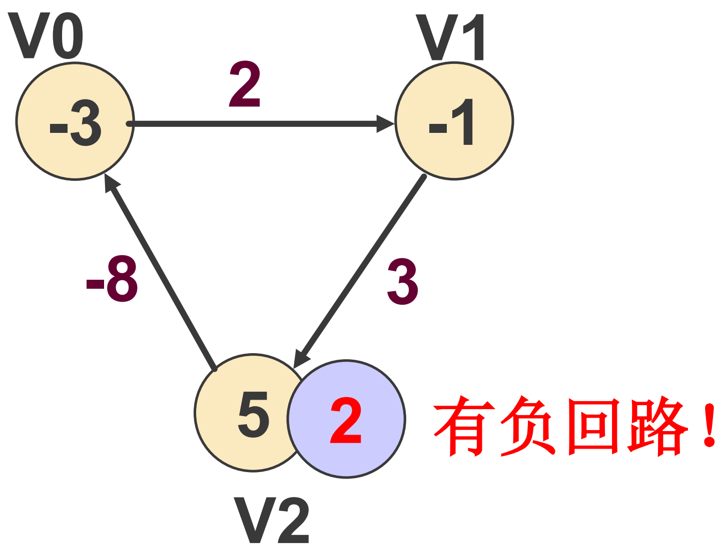
V0, V1, V2
0, ∞, ∞
0, 2, ∞
-3, -1, 5

第1次考察边：V1V2, V2V0, V0V1

第2次考察边：V1V2, V2V0, V0V1

6.5.2 Bellman-Ford算法

- 图中有负回路的情况
- 如何判断图中有负回路?



V0, V1, V2
0, ∞, ∞
0, 2, ∞
-3, -1, 5

规划完成后再次考察边集**V1V2**, V2V0, V0V1

若存在某个顶点，其距离再次减小，则有负回路。

6.5.2 Bellman-Ford算法

- 算法过程：
- 1. 初始化所有结点到源点距离为∞;
- 2. for(i=1; i<n; i++)
- 对每一条边<u, v>依次进行下列判断（松弛操作）
  - 如果 dist[v] > dist[u] + w(u, v)
  - 则 dist[v] = dist[u] + w(u, v) ;
- 3. 判断图中是否有负回路：对每一条边<u, v>依次进行下列判断：
  - 如果dist[v] > dist[u] + w(u, v)
  - 则有负回路，return false;

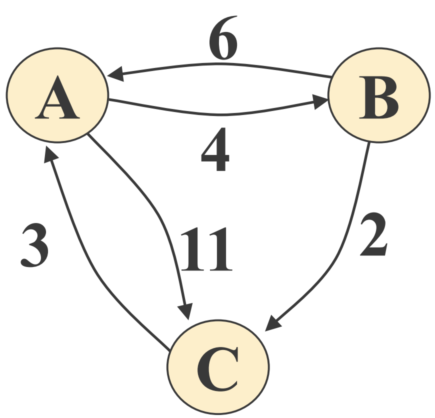
时间复杂度？

边集数组

O(n×E)

6.6.3 弗洛伊德算法

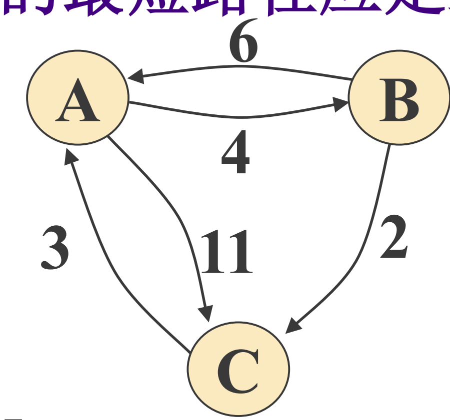
- 求每一对顶点之间的最短路径
- 条件：可以有负权重，不能有负回路
- 基本思想：逐个顶点试探法
- 1、定义一个n阶方阵*A*[n][n]，记录当前任意两点间的距离.
  - 初始时，若弧<vi, vj>存在，则(vi, vj)间最短路径记为{vi, vj}
- 2、依次尝试在路径中加入新顶点能够得到更短的路径



	A	B	C
A	0	4	11
B	6	0	2
C	3	∞	0

6.6.3 弗洛伊德算法

- 2、从v0开始，依次尝试在路径中加入新顶点能够得到更短的路径。
- 即要找出从 vi 到 vj的最短路径，从vi 到 vj所有可能的路径中，选出一条长度最短的路径
  - 考察v0：若弧<vi,v0>,<v0,vj>存在，则存在路径{vi,v0,vj}，若该路径更短，则更新(vi, vj)间最短路径
  - 考察v1：{vi,…,v1}, {v1,…,vj}存在，则存在一条路径{vi, …, v1, …vj}，即中间序号不大于1的路径
  - 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



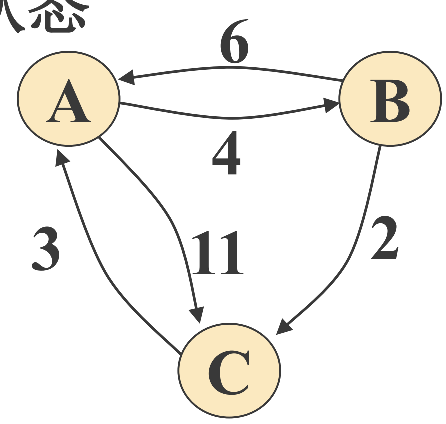
6.6.3 弗洛伊德算法

- 求最短路径步骤:
- 初始时设置一个 n 阶方阵，令其对角线元素为 0，若存在弧 <Vi, Vj>，则对应元素为权值；否则为∞。

> 逐步试着在原直接路径中增加中间顶点，若加入中间点后路径变短，则修改之；否则，维持原值。

> 所有顶点试探完毕，算法结束。

初始状态



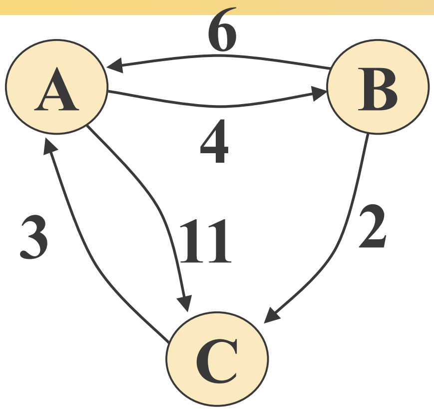
	A	B	C
A	0	4	11
B	6	0	2
C	3	∞	0

对应路径:

	A	B	C
A		AB	AC
B	BA		BC
C	CA		

6.6.3 弗洛伊德算法

初始状态



	A	B	C
A	0	4	11
B	6	0	2
C	3	∞	0

对应路径：

	AB	AC
BA		BC
CA		

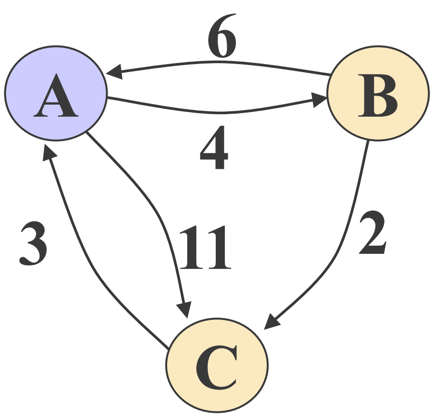
第一步：加入A点

考察：<B, C> = 2

<B, A> <A,C> = 6+11=17

考察： <C, B> = ∞

<C, A> <A,B> = 3+4=7

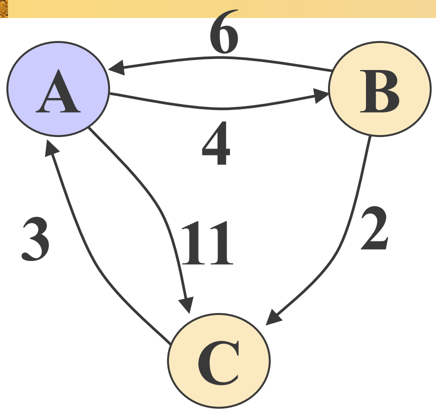


	A	B	C
A	0	4	11
B	6	0	2
C	3	7	0

对应路径：

	AB	AC
BA		BC
CA	CAB	

第1步：加入A点

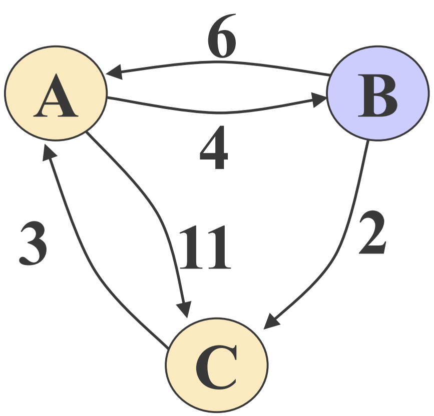


	A	B	C
A	0	4	11
B	6	0	2
C	3	7	0

对应路径：

	AB	AC
BA		BC
CA	CAB	

第2步：加入B点



考察：<A, C> = 11

<A, B> <B, C> = 4+2=6

考察： <C, A> = 3

<C, B> <B, A> = 7+6=13

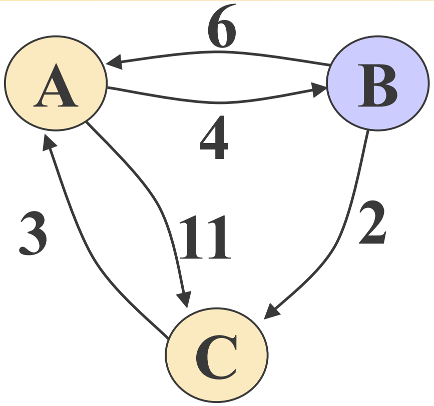
	A	B	C
A	0	4	6
B	6	0	2
C	3	7	0

对应路径：

	AB	ABC
BA		BC
CA	CAB	

6.6.3 弗洛伊德算法

第2步：加入B点



	A	B	C
A	0	4	6
B	6	0	2
C	3	7	0

对应路径：

	AB	ABC
BA		BC
CA	CAB	

第3步：加入C点

考察：<A, B> = 4

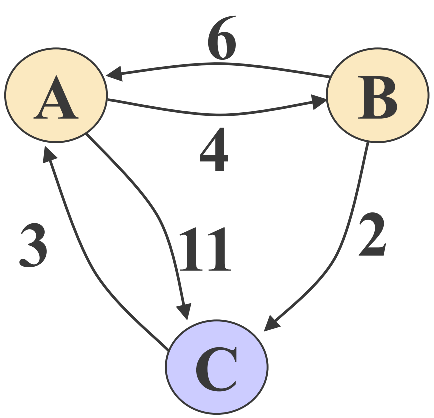
<A, C> <C, B> = 6+7=13

考察： <B, A> = 6

<B, C> <C, A> = 2+3=5

复杂度:

$O(n^3)$



	A	B	C
A	0	4	6
B	5	0	2
C	3	7	0

对应路径：

	AB	ABC
BCA		BC
CA	CAB	

6.5.3 弗洛伊德算法

- **Floyd算法的基本思想**
- 定义一个n阶方阵序列：

$$A^{(0)},\ A^{(1)},\ \ldots,\ A^{(n)}.$$

- 其中 $A^{(0)}[i][j] = \text{G.Edge}[i][j]$

$$A^{(k)}[i][j] = \min\{ A^{(k\text{-}1)}[i][j],$$
$$A^{(k\text{-}1)}[i][k] + A^{(k\text{-}1)}[k][j] \},\ k = 1, 2, \ldots, n$$

- $A^{(1)}[i][j]$ 是从顶点 v_i 到 v_j，中间可能绕过顶点 v_1 的最短路径长度;
- $A^{(k)}[i][j]$ 是从顶点 v_i 到 v_j，中间可能绕过顶点 v_1, v_2, …, v_k 的最短路径的长度;
- $A^{(n)}[i][j]$是从顶点v_i到v_j的最短路径长度。

```
void Floyd ( MGraph& G, Weight a[ ][maxVertices],
      int path[ ][maxVertices] ) {
//算法计算每一对顶点间最短路径及最短路径长度。
//a[i][j] 是顶点 i 和 j 间的最短路径长度。
//path[i][j] 是相应路径上顶点 j 的前一顶点的顶点号。
      int i, j, k, n = G.numVertices; // 步骤1：初始化
      for ( i = 0; i < n; i++ )    //初始化矩阵a与path
           for ( j = 0; j < n; j++ ) {
                 a[i][j] = G.Edge[i][j];
                 if ( i == j ) path[i][j] = 0; //对角线置零
                 else if ( a[i][j] < maxWeight ) path[i][j] = i;
                 else path[i][j] = -1;
           }
     //步骤2：依次尝试经过各顶点是否能得到更短路径
} //Floyd
```

6.5.3 弗洛伊德算法

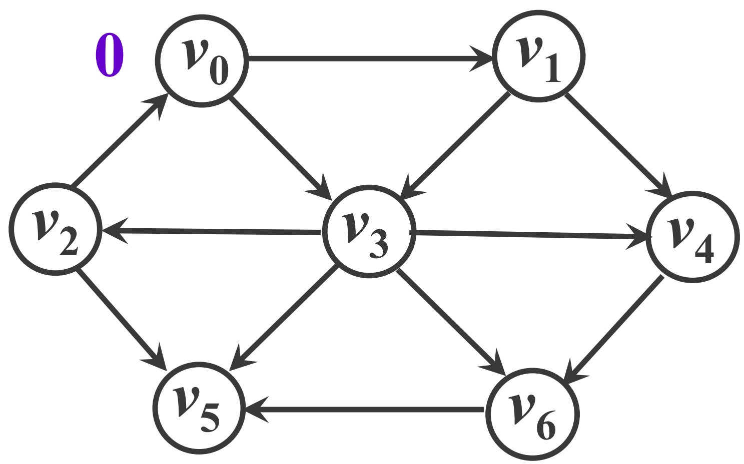
```
//步骤2：依次尝试经过各顶点是否能得到更短路径
for ( k = 0; k < n; k++ ) { //对每一个 顶点k 计算
    for ( i = 0; i < n; i++ ) //更新矩阵a和path
        for ( j = 0; j < n; j++ )
            if ( a[i][k] + a[k][j] < a[i][j] ) {   //经过k 到 j
                a[i][j] = a[i][k] + a[k][j];
                path[i][j] = path[k][j];    //缩短路径长度
            }//for ( j = 0
    } //for ( i = 0
} //for ( k = 0
```

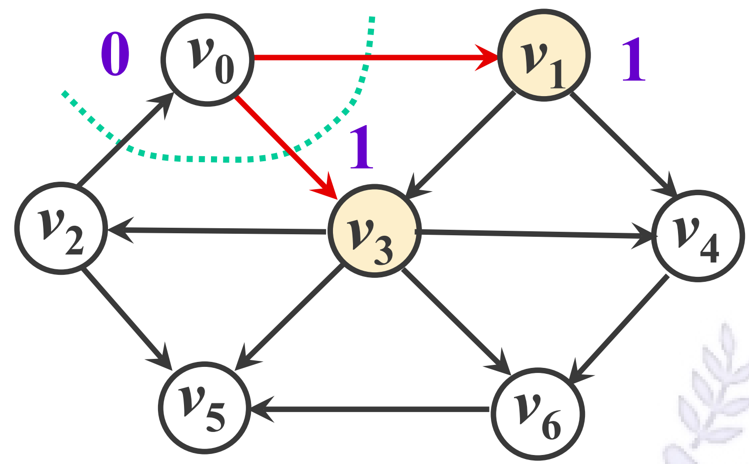
复杂度:

$O(n^3)$

6.5.4 非带权图的最短路径问题

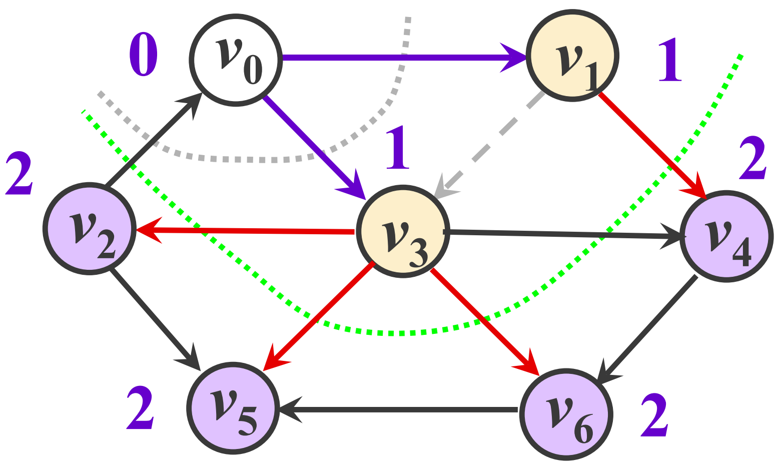
- 问题：在非带权图中以某个顶点 v 作源点，求出它到其他各个顶点的最短路径
- 方法：图的广度优先搜索BFS算法

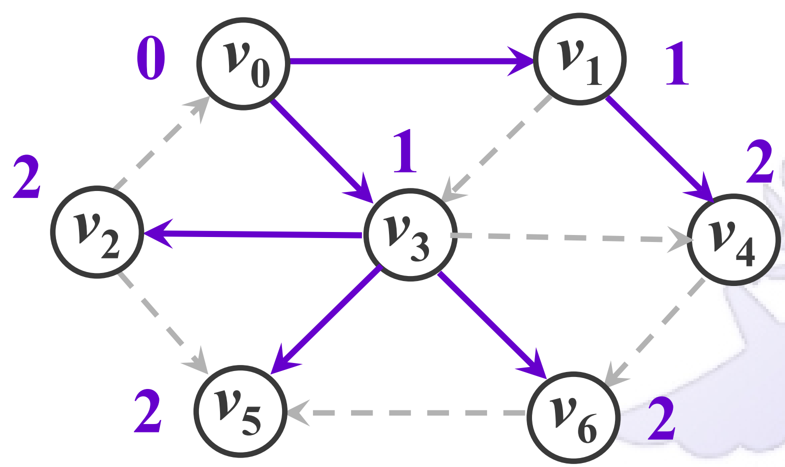




6.5.4 非带权图的最短路径问题

- 在BFS过程中记录路径的长度和结点的前驱
  - int dist[ ];//v0到各个顶点的路径长度
  - int path[ ];//各个顶点的最短路径的前驱结点
- 时间复杂度：邻接表O(n+e)；邻接矩阵O(n²)
- 空间复杂度：O(n)

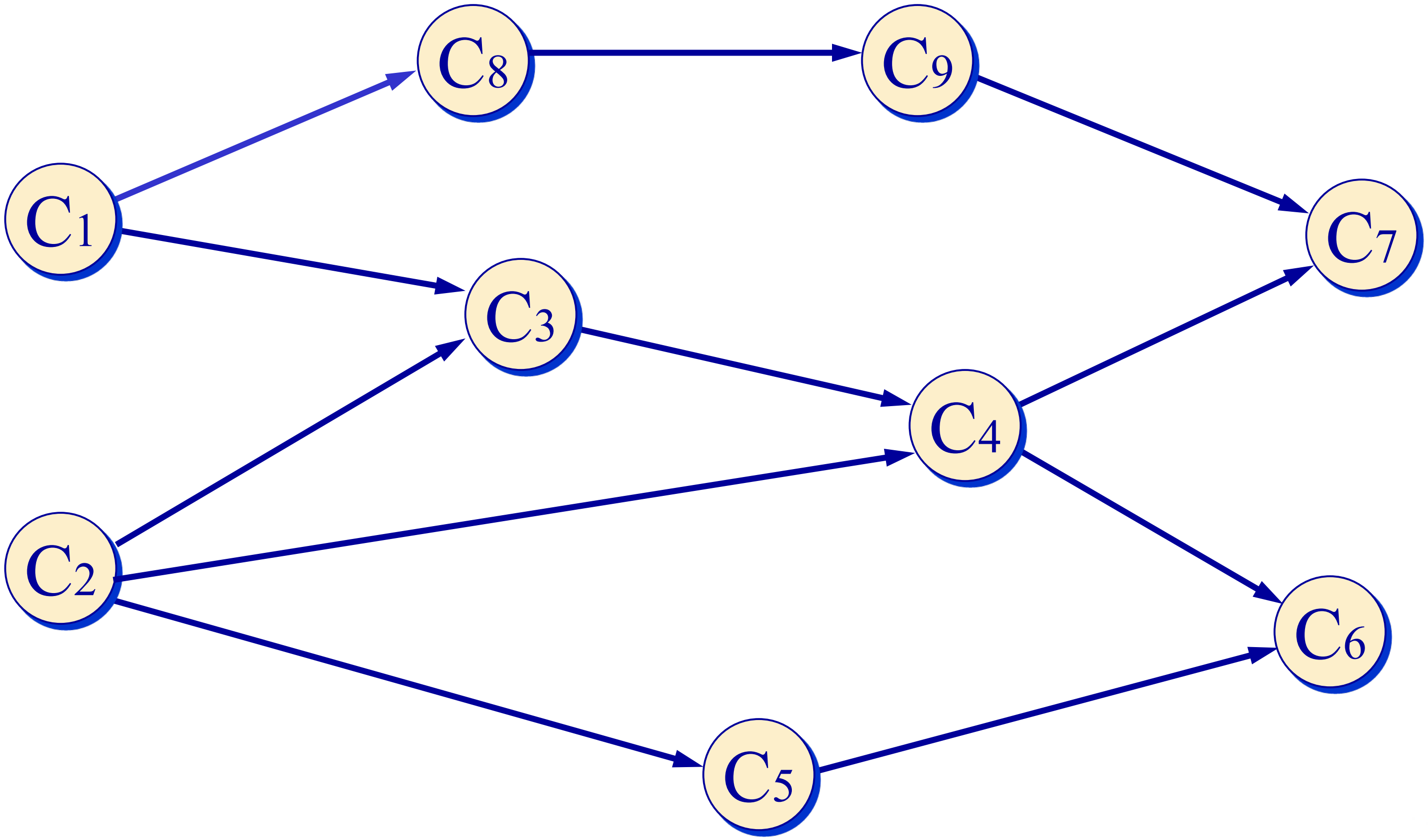




6.6 活动网络

- 可以用有向图表示一个工程的施工图或流程图.
  - 顶点表示活动
  - 有向边<Vi, Vj>表示活动Vi 必须先于活动Vj 进行。
- 这种有向图叫做顶点表示活动的**AOV网络 (Activity On Vertices)**

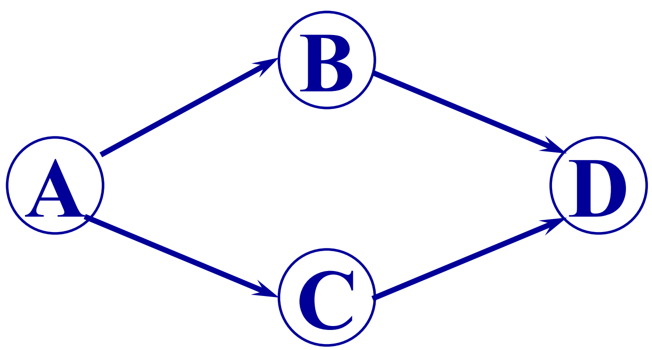
课程代号	课程名称	先修课程
C_1	高等数学	
C_2	程序设计基础	
C_3	离散数学	C_1，C_2
C_4	数据结构	C_3，C_2
C_5	高级语言程序设计	C_2
C_6	编译方法	C_5，C_4
C_7	操作系统	C_4，C_9
C_8	普通物理	C_1
C_9	计算机原理	C_8

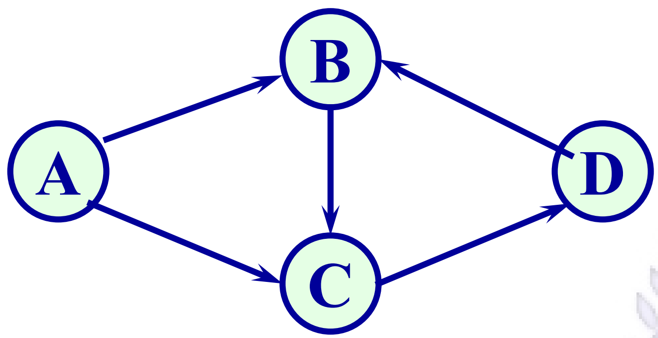


学生课程学习工程图

6.6 活动网络

- 活动网络中不允许出现回路.
- 这样的图称为**有向无环图——Directed Acycline Graph)**



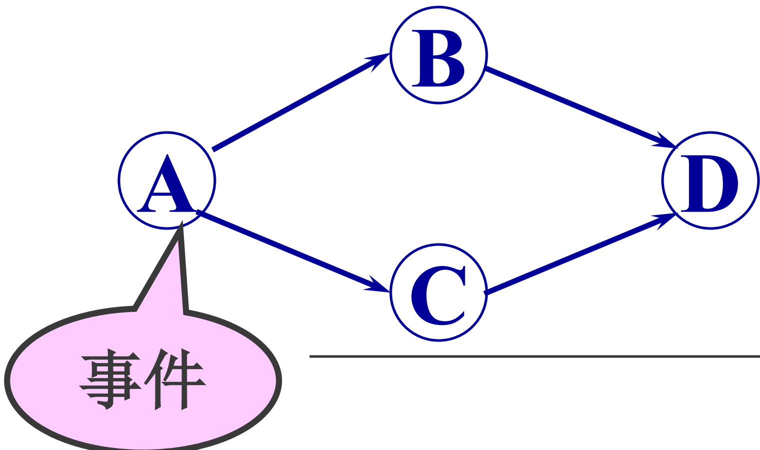


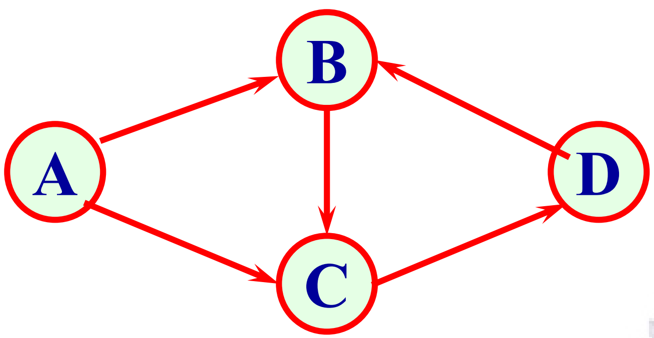
6.6 活动网络

- 工程图中的问题
  - 有向图中是否存在回路?
  - 如何安排施工计划?
- 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
- 安排施工计划的方法是找关键路径

6.6.1 拓扑排序

- 拓扑排序：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





A B C D

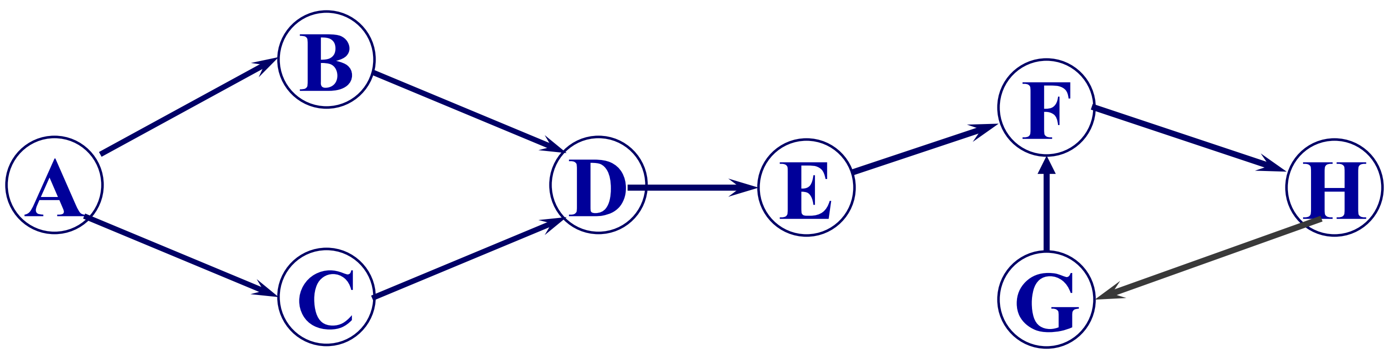
A C B D

AOV网络
Activity on vertex network

有回路
无拓扑排序

如何进行拓扑排序？

1. 从有向图中选取一个没有前驱的顶点，并输出之；
2. 从有向图中删去此顶点以及所有以它为尾的弧；
3. 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。

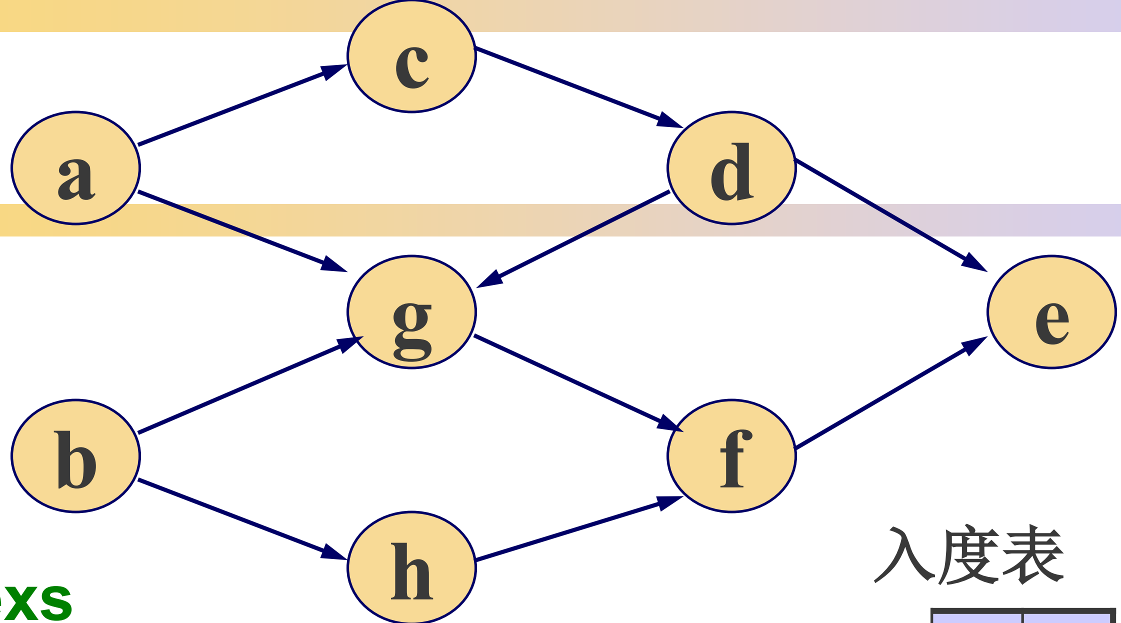


a　b　h　c　d　g　f　e

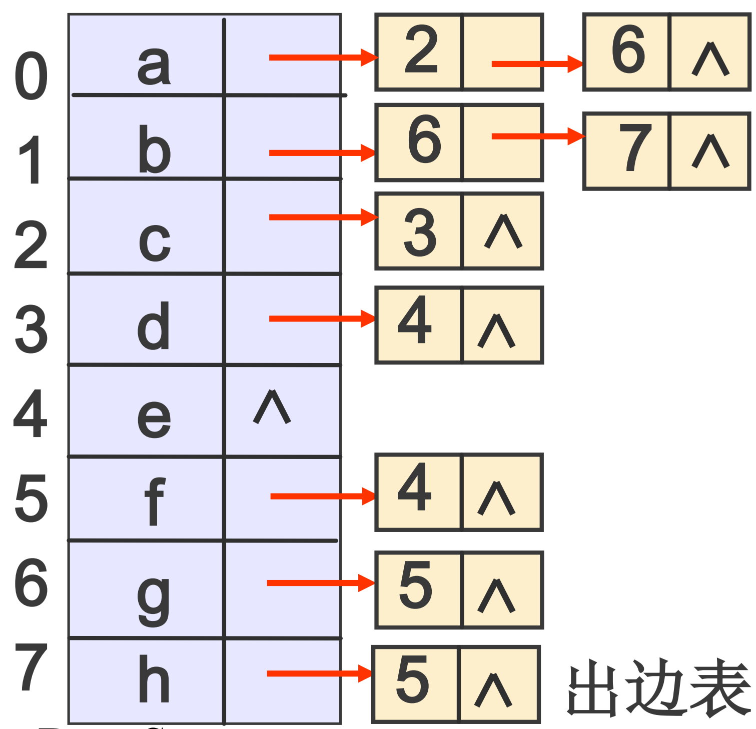
什么顶点是没有前驱的?	== 入度为零的顶点
如何删除顶点及以它为尾的弧?	== 弧头顶点的入度减1

拓扑排序算法实现

- 数据结构
  - 用邻接表作存储结构；
  - 用一个数组记录顶点当前的入度
  - 用栈记录入度为0的顶点
- 算法过程：
  - 1）把邻接表中所有入度为0的顶点进栈。
  - 2）栈非空时
    - 弹出栈顶元素 Vj 并输出；
    - 在邻接表中查找 Vj 的直接后继 Vk，把 Vk 的入度减1；
    - 若 Vk 的入度为 0 则进栈。
  - 3）重复上述操作直至栈空为止。
  - 4）若栈空时输出的顶点个数小于 n，则有向图有回路；否则，拓扑排序完毕。



G.vexs

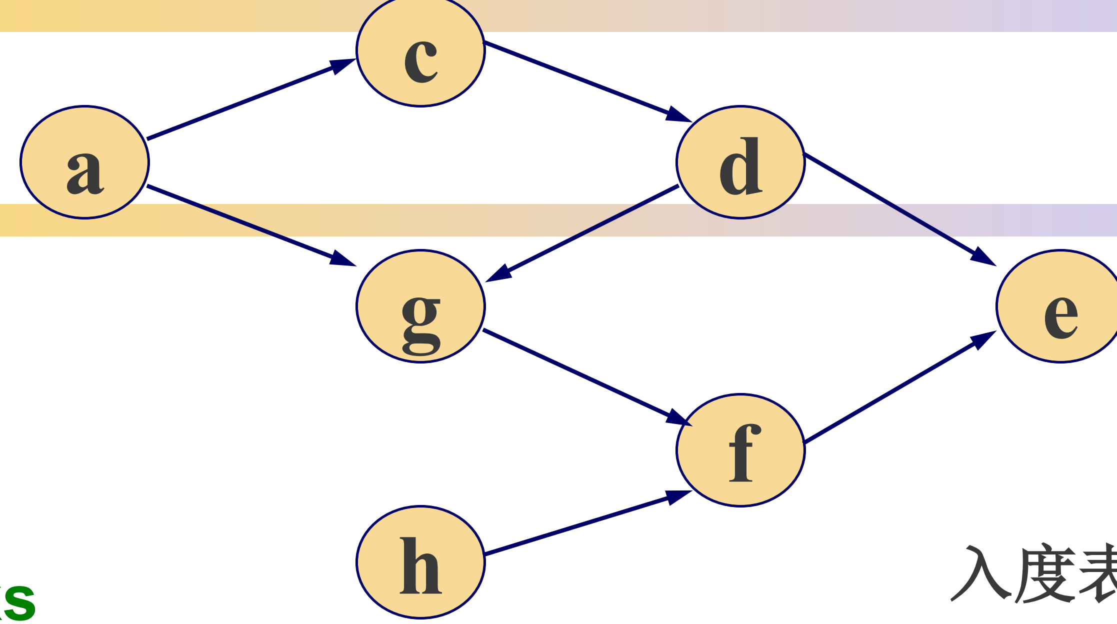


入度表

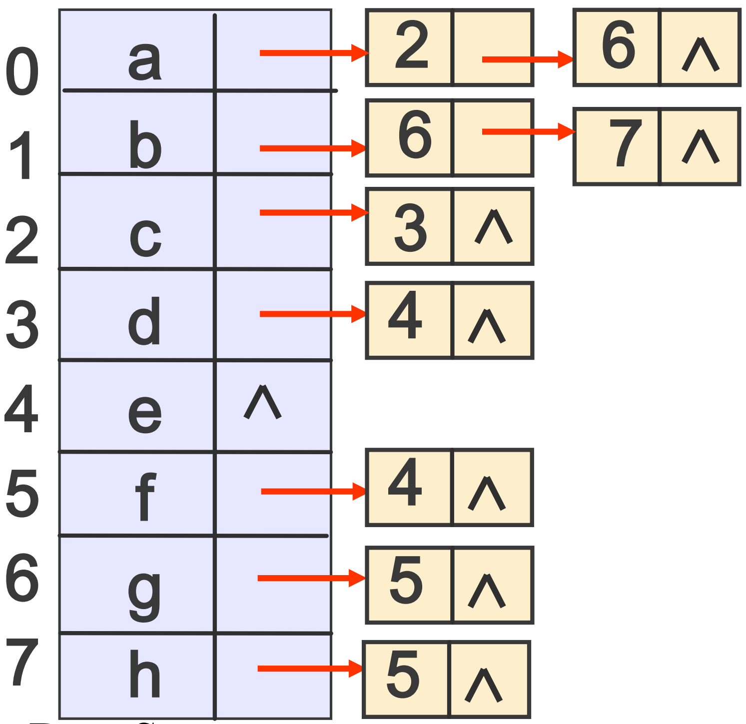
0	a	0
1	b	0
2	c	1
3	d	1
4	e	2
5	f	2
6	g	1
7	h	0

堆栈

8	
7	
6	
5	
4	
3	
2	h
1	a



G.vexs



入度表

0	a	0
1	b	0
2	c	1
3	d	1
4	e	2
5	f	1
6	g	1
7	h	0

栈

8	
7	
6	
5	
4	
3	
2	
1	a

拓扑排序算法

- 在算法中设置一个“栈”，以保存“入度为零”的顶点.

```
bool TopologicalSort ( ALGraph& G, int topoArray[ ], int& k )
{// 拓扑排序，图G采用邻接表存储
    CountInDegree(G, indegree);  //对各顶点求入度
    InitStack(S);
    //入度为零的顶点入栈
    for ( i=0; i<G.numVertices; ++i)
        if (!indegree[i])  Push(S, i);//若入度为0则进栈
    拓扑排序
}//TopologicalSort
```

拓扑排序

时间复杂度： O(n+e)

空间复杂度： O(n)

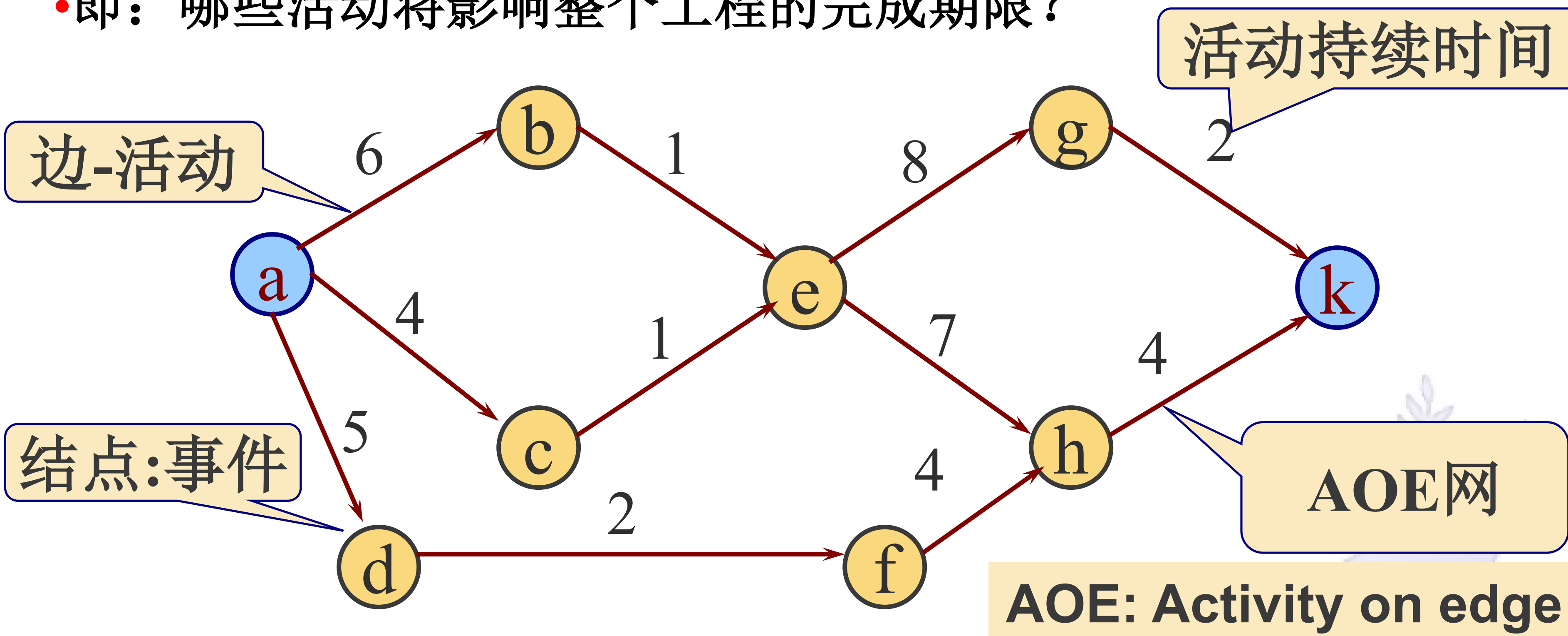
```
k=0;          //对输出顶点计数
while (!EmptyStack(S)) {
    Pop(S, j);  topoArray[k++] = j;
    w = firstNeighbor(G, j);
    while(w != -1) {
            --indegree[w];  // 弧头顶点的入度减一
            if (indegree[w] == 0)  Push(S, w); //零入度点入栈
            w = nextNeighbor (G, j, w))
    }//while(w!=-1)
}//while (!EmptyStack(S))
if (k<G.numVertices) printf("图中有回路" )
```

6.5.2 关键路径

- 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某活动所需时间。

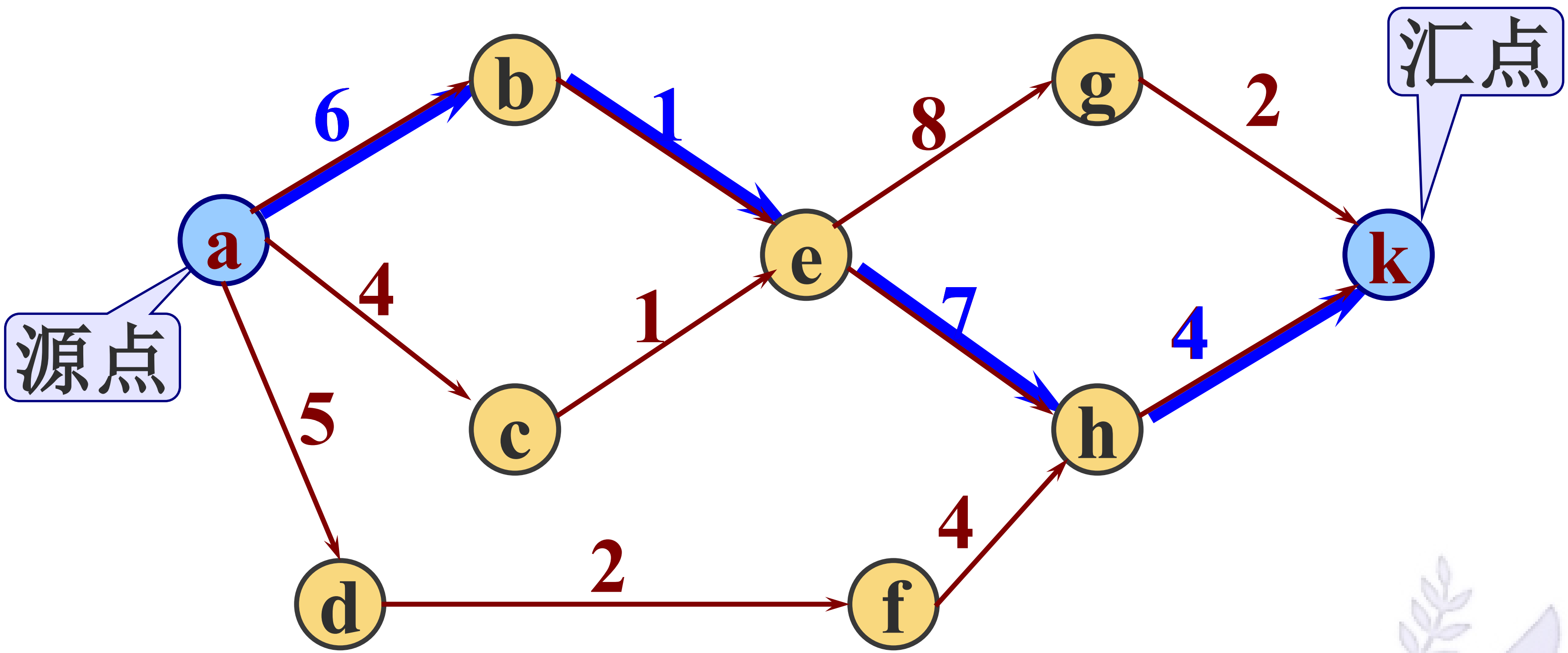
•问：哪些活动是“关键活动”？

•即：哪些活动将影响整个工程的完成期限？



6.5.2 关键路径

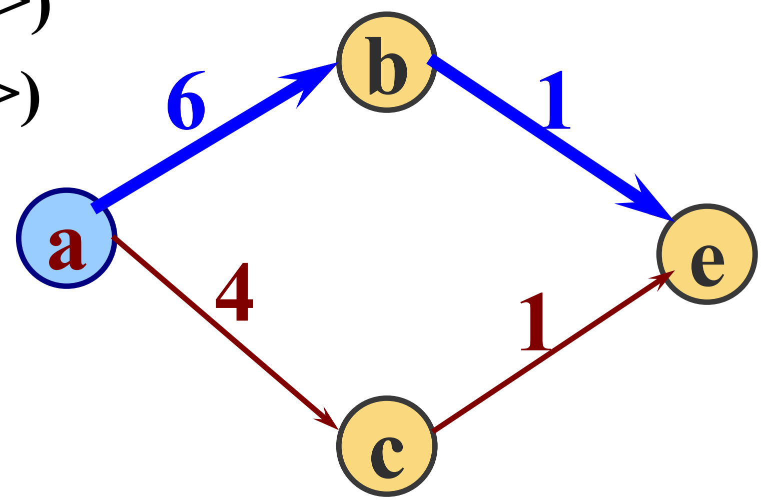
整个工程的完成时间：从有向图源点到汇点的最长路径。



"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？

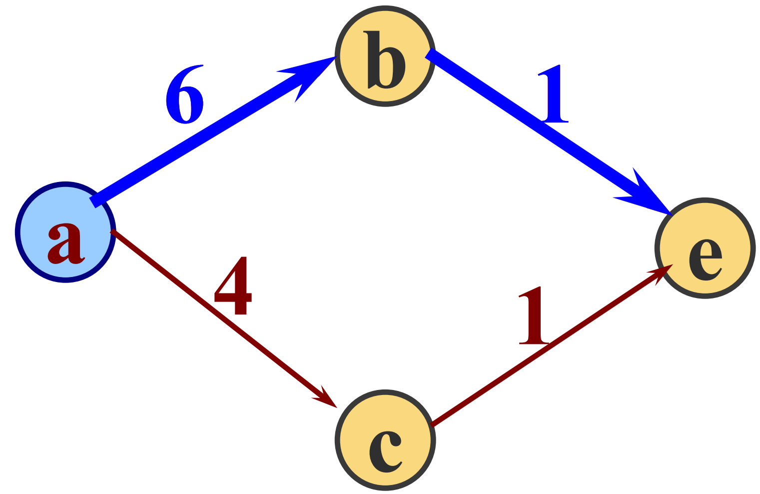
- “活动(弧)”的 最早开始时间 Ae(<j, k>)
- “活动(弧)”的 最迟开始时间 Al(<j, k>)
- 关键活动Ae(<j, k>) = Al(<j, k>)
- “事件(顶点)” 的最早发生时间 Ve(j)
- “事件(顶点)” 的最迟发生时间 Vl(k)

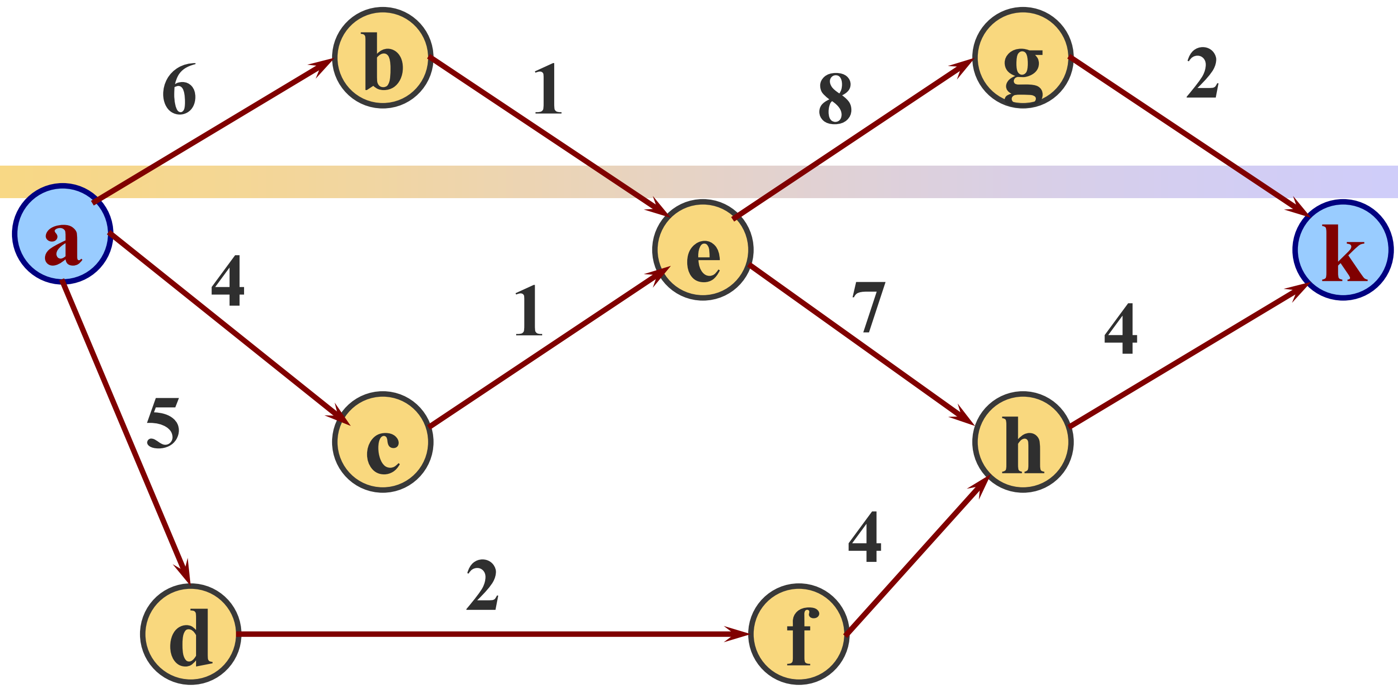


- 活动(弧)发生时间的计算公式
- 假某条弧为 <j, k>， 则 对活动<j, k> 而言
- Ae(<j, k>) = Ve(j);
- Al(<j, k>) = Vl(k) – dut(<j, k>);

如何求关键活动？

- 事件(顶点)发生时间的计算公式
- 最早开始时间：
  - $Ve(源点) = 0$；
  - $Ve(k) = Max\{Ve(j) + dut(<j, k>)\}$
  - 从前向后计算，按照拓扑排序次序计算
- 最迟开始时间：
  - $Vl(汇点) = Ve(汇点)$；
  - $Vl(j) = Min\{Vl(k) - dut(<j, k>)\}$
  - 从后向前计算，按照逆拓扑排序次序计算





	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 - b - c - d - f - e - h - g - k

如何求关键活动？

	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	af	be	ce	df	eg	eh	fh	gk	hk
权	6	4	5	1	1	2	8	7	4	2	4
Ae	0	0	0	6	4	5	7	7	7	15	14
Al	0	2	5	6	6	8	8	7	10	16	14
	✓			✓				✓			✓

- e(<j, k>) = ve(j);
- l(<j, k>) = vl(k) – dut(<j,k>);

算法的实现要点

- 算法步骤：

1. 计算一个拓扑排序；
2. 按拓扑序列的顺序，求顶点的最早发生时间Ve;
3. 按逆拓扑序列的顺序，求顶点的最迟发生时间Vl;
4. 由Ve、Vl, 计算每个活动的Ae[k]和Al[k]；
5. 找出Ae[k]==Al[k]的关键活动

- 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。

END

本章要点

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。
3. 注意图的遍历算法与树的遍历算法之间的类似和差异。
4. 应用图的遍历算法求解各种简单路径问题
5. 理解教科书中讨论的各种图的算法