数据结构

教师：姜丽红 jianglh@sjtu.edu.cn

IST 实验室 http://ist.sjtu.edu.cn

助教：芮召普 ruishaopu@qq.com 江嘉晋 IST实验室 软件大楼5号楼5314

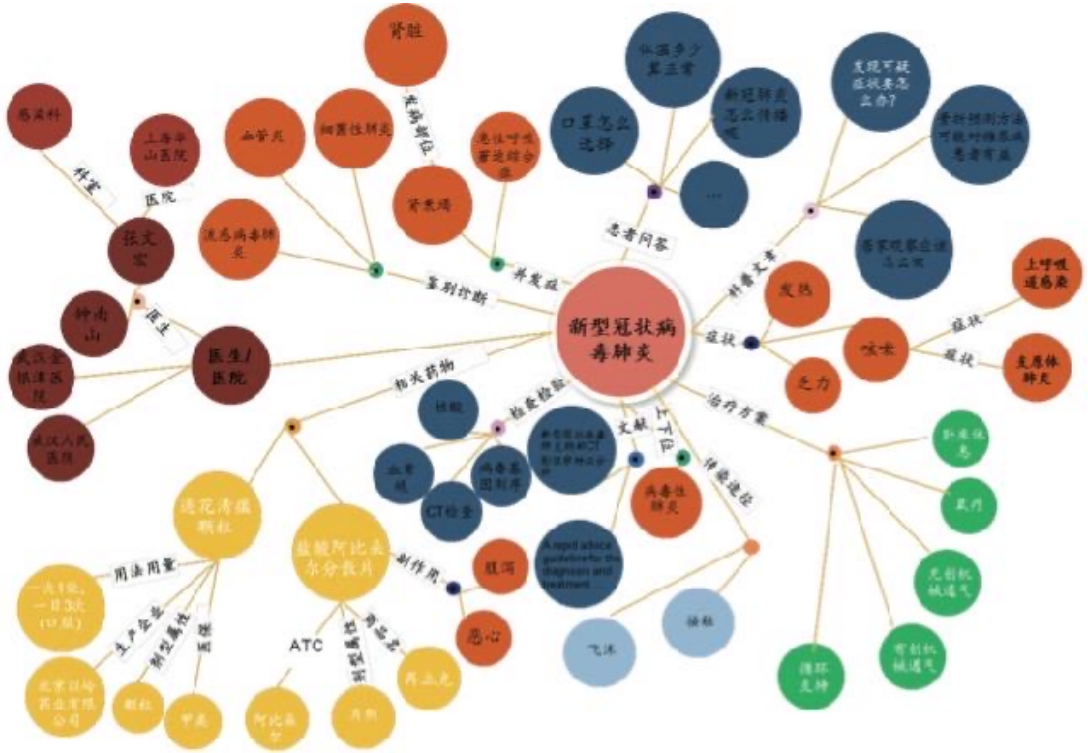
《软件基础实践》教师：杜东 IPADS 实验室 软件大楼3号楼4层

上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

第13章 图的基本概念

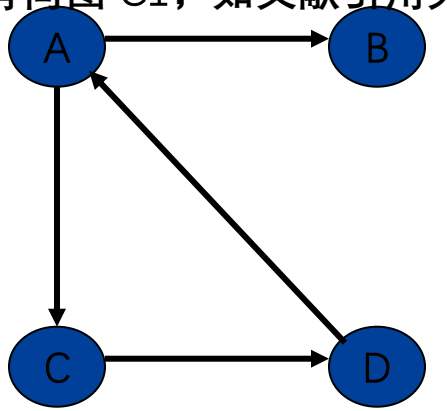
- 图的定义
- 图的术语
- 图的运算
- 图的存储
- 图的遍历
- 图遍历的应用



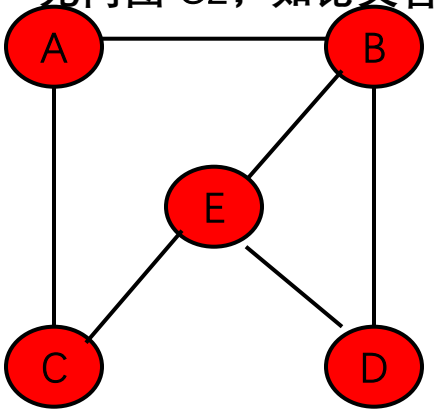
地铁网、社交网、知识图谱……

图的基本概念

有向图 G1，如文献引用关系图



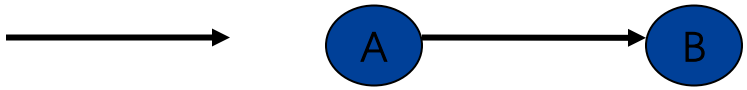
无向图 G2，如论文合作关系图



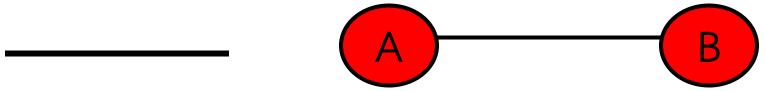
- 结点或 顶点：

- 结点或 顶点：

- 有向边（弧）、弧尾或初始结点、弧头或终止结点



- 无向边或边



- 有向图：G1 =（V1，A1）

  V1 = {A,B,C,D}

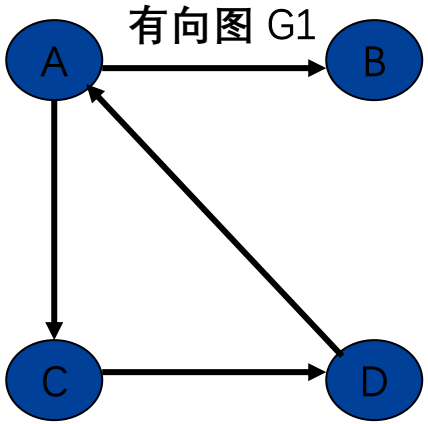
  A1 = {<A,B>, <A,C>, <C,D>, <D,A>}

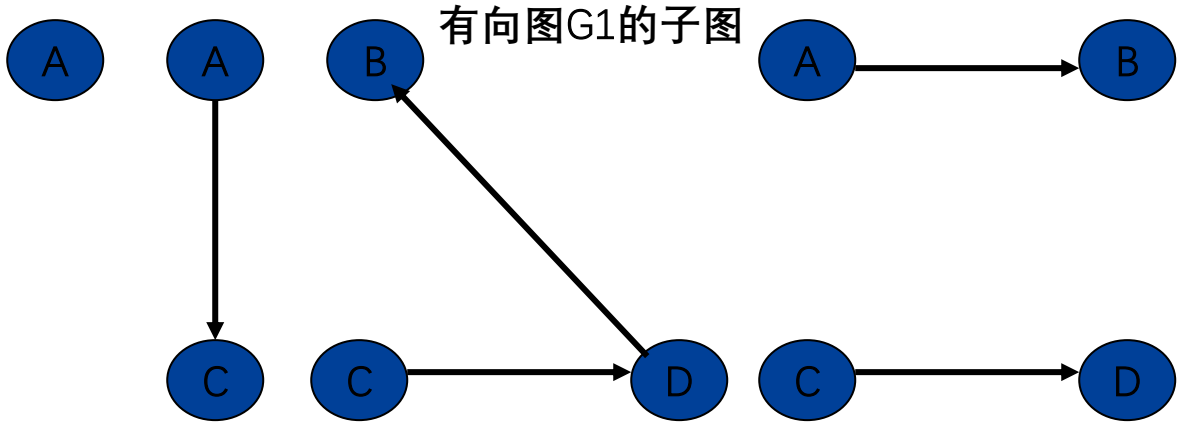
- 无向图：G2 =（V2，A2）

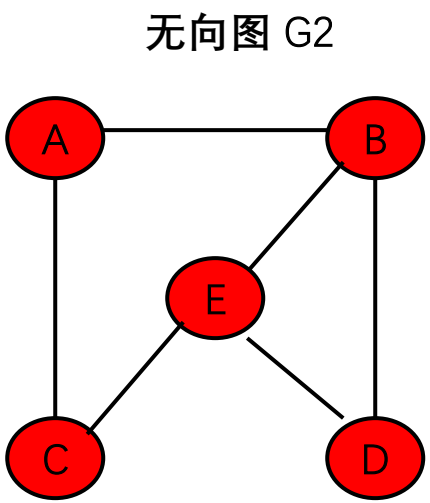
  V2 = {A,B,C,D,E}

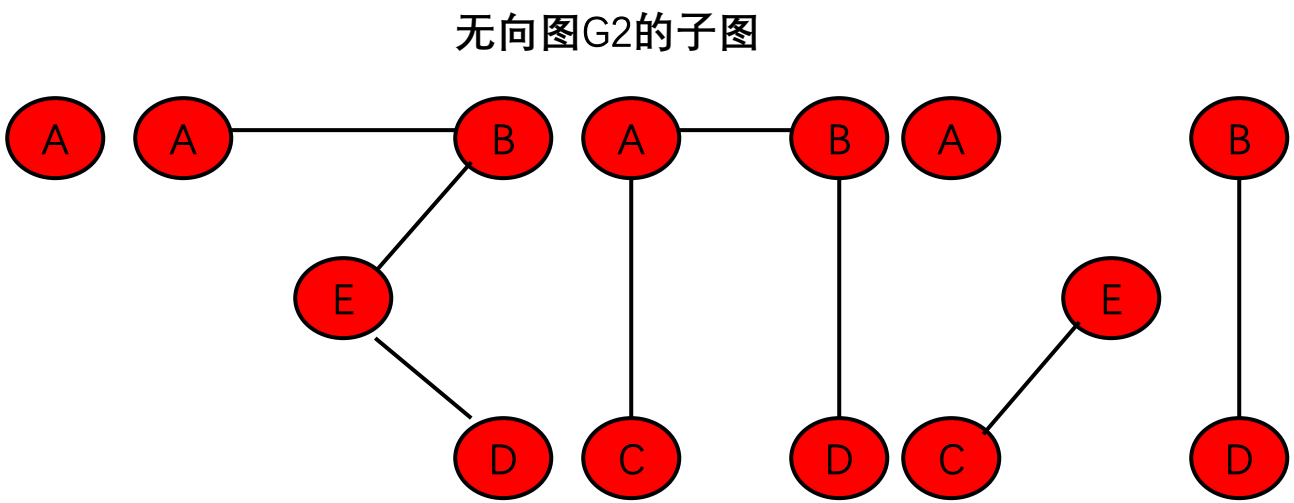
  A2 = {(A,B), (A,C),(B,D), (B,E), (C,E),(D,E)}

图的基本概念


有向图 G1
A
B
C
D


有向图G1的子图
A
A
B
A
B
C
C
D
C
D


无向图 G2
A
B
E
C
D

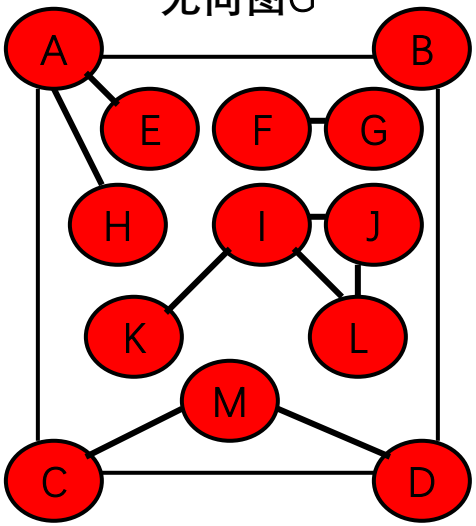

无向图G2的子图
A
A
B
A
B
A
B
E
E
D
C
D
C
D

图的基本概念

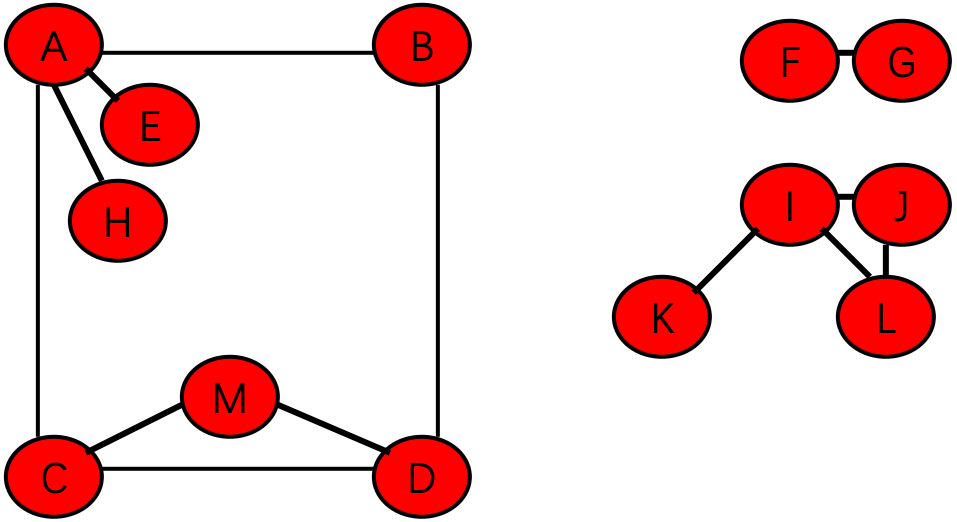
无向图的连通性

- 路径：在无向图G=(V,{E})中由顶点vi至vj 的顶点序列。
- 回路或环：第一个顶点和最后一个顶点相同的路径。
- 简单回路或简单环：除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。
- 连通：顶点vi至vj 之间有路径存在
- 连通图：无向图图 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

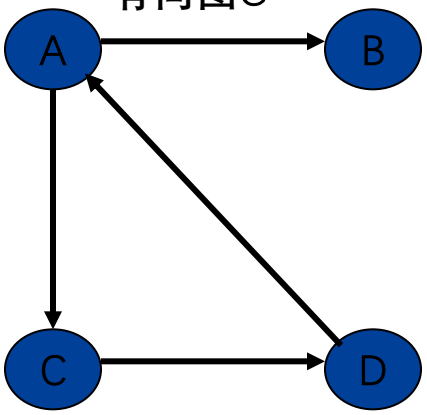


图的基本概念

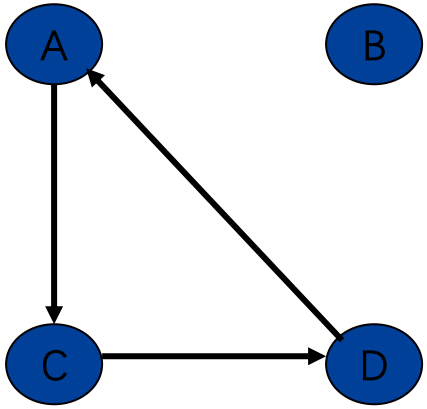
有向图的连通性

- 路径：在有向图G=(V,{E})中由顶点vi经有向边至vj的顶点序列。
- 回路或环：第一个顶点和最后一个顶点相同的路径。
- 简单回路或简单环：除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。
- 连通：顶点vi至vj之间有路径存在
- 强连通图：有向图图 G 的任意两点之间都是连通的，则称 G 是强连通图。
- 弱连通：有向图的基图（将有向边变成无向边后形成的图）是连通的。
- 强连通分量：极大连通子图

有向图G



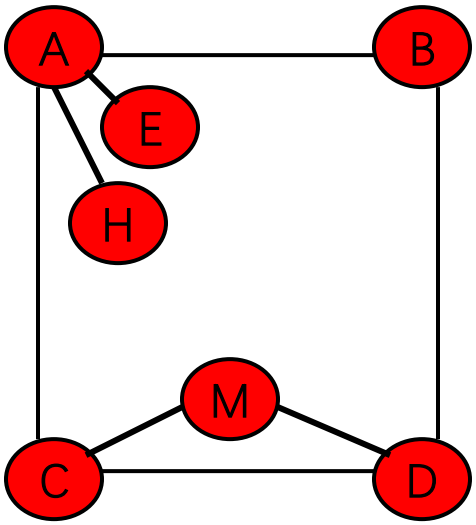
有向图G的两个强连通分量



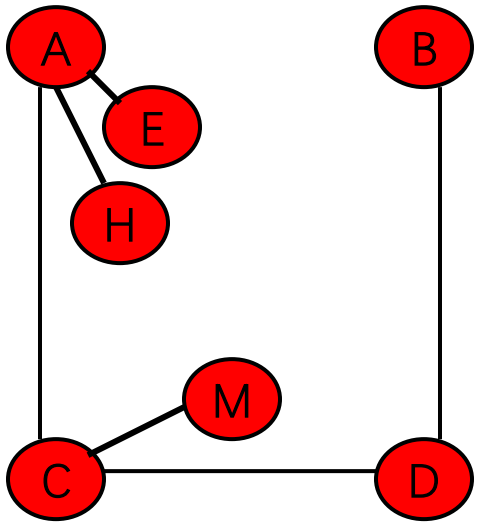
图的基本概念

•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。在生成树中添加一条边之后，必定会形成回路或环。

无向图G



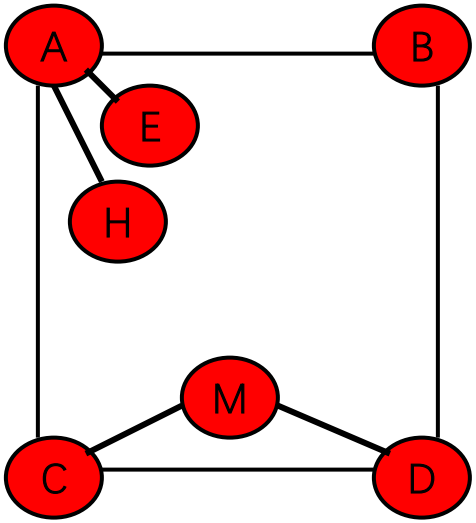
无向图G的生成树



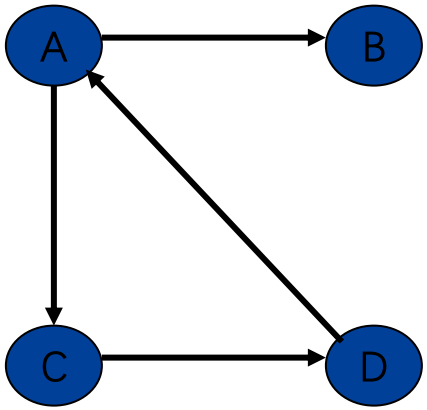
图的基本概念

- 完　全　图：有 n(n-1)/2 条边的无向图。其中 n 是结点个数。C_n^2
- 有向完全图：有 n(n-1)　条边的有向图。其中 n 是结点个数。$2C_n^2$
- 边的权值，边有权的图称之为网络。
- 邻接点
- 无向图结点的度
- 有向图结点的出度和入度

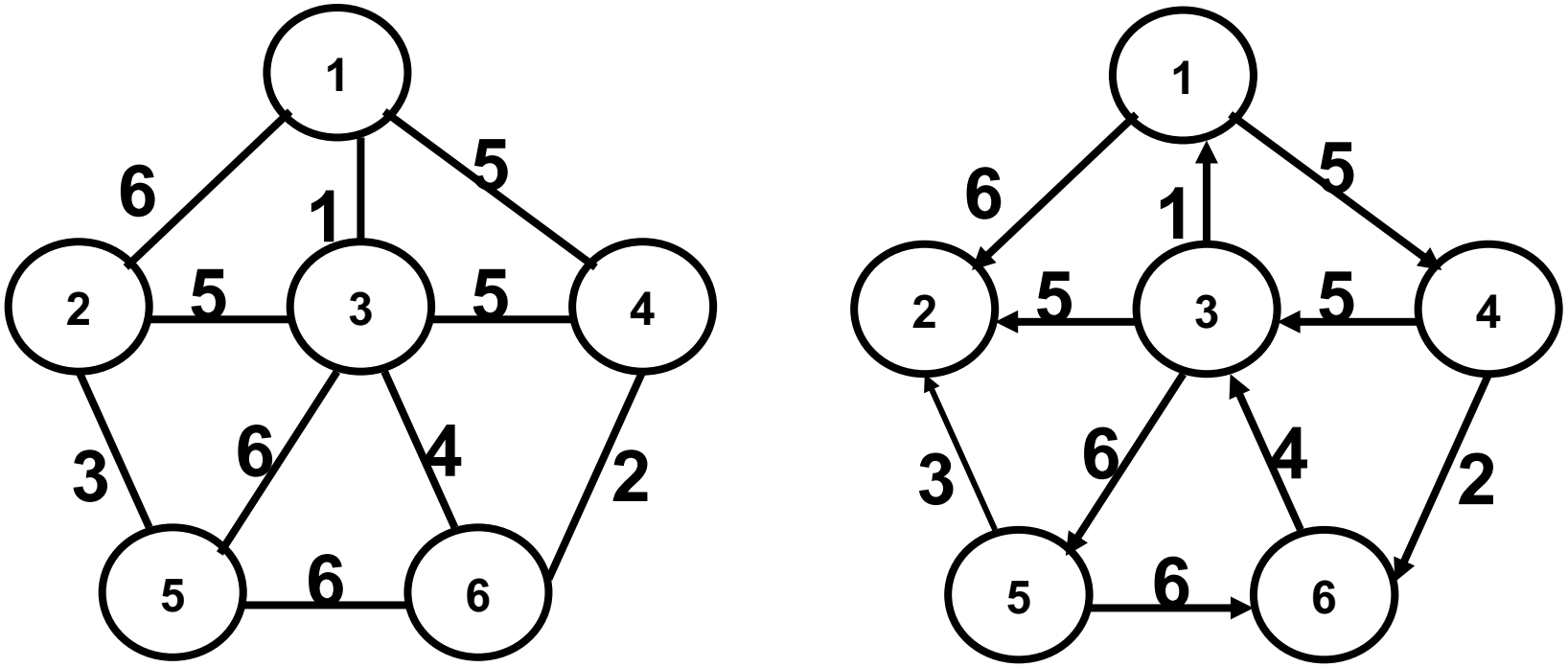
无向图G1



有向图G2



加权图



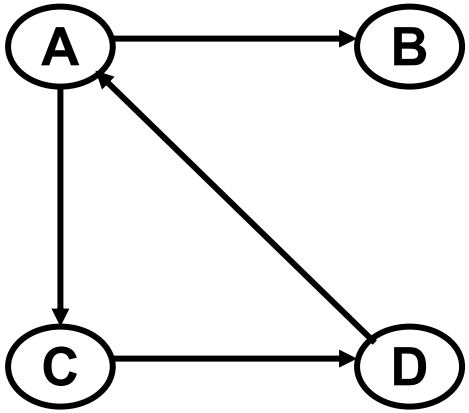
加权图中边的表示：$<V_i, V_j, W>$或(V_i, V_j, W)

图的运算

- 常规操作：
  - 构造一个由若干个结点、若干条边组成的图；
  - 判断两个结点之间是否有边存在；
  - 在图中添加或删除一条边；
  - 返回图中的结点数或边数；
  - 按某种规则遍历图中的所有结点。
- 和应用紧密结合的运算：
  - 拓扑排序 / 关键路径（本学期）
  - 找最小生成树
  - 找最短路径等。

邻接矩阵

设有向图具有 n 个结点，则用 n 行 n 列的布尔矩阵 A 表示该有向图如果i 至 j 有一条有向边, A[i,j] = 1 ,如果 i 至 j 没有一条有向边,A[i,j] = 0



表示成右图矩阵

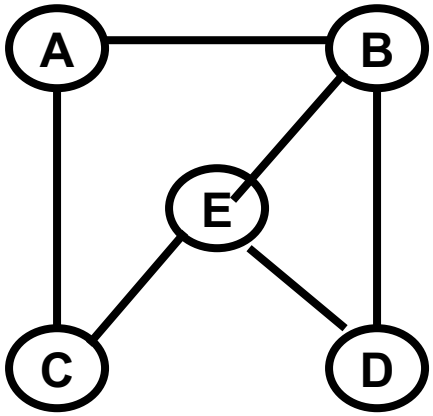
$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

0	1	2	3
A	B	C	D

出度: i行之和
入度: j列之和

邻接矩阵

设无向图具有 n 个结点，则用 n 行 n 列的布尔矩阵 A 表示该无向图；并且 A[i,j] = 1，如果i 至 j 有一条无向边；A[i,j] = 0如果 i 至 j 没有一条无向边。



表示成右图矩阵

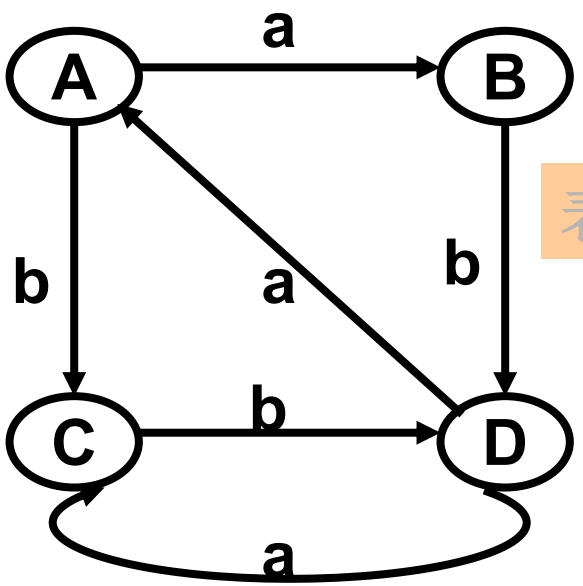
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

对称矩阵； i 结点的度为第 i 行或 i 列之和。

加权的邻接矩阵

如果i 至 j 有一条有向边且它的权值为a ，则A[i,j] = a 。

如果 i 至 j 没有一条有向边。则A[i,j] = 空 或其它标志



表示成右图矩阵

$$\begin{pmatrix} \infty & a & b & \infty \\ \infty & \infty & \infty & b \\ \infty & \infty & \infty & b \\ a & \infty & a & \infty \end{pmatrix}$$

- 优点：判断任意两点之间是否有边方便。
- 缺点：即使 $<< n^2$ 条边，也需内存 n^2 单元。

邻接矩阵类的定义

```
template <class TypeOfVer, class TypeOfEdge>
class adjMatrixGraph:public graph<TypeOfVer, TypeOfEdge> {
public:
   adjMatrixGraph(int vSize, const TypeOfVer d[],
                             const TypeOfEdge noEdgeFlag);
    void insert(TypeOfVer x, TypeOfVer y, TypeOfEdge w);
    void remove(TypeOfVer x, TypeOfVer y);
    bool exist(TypeOfVer x, TypeOfVer y) const;
    ~adjMatrixGraph() ;

private:
    TypeOfEdge **edge;          //存放邻接矩阵
    TypeOfVer *ver;             //存放结点值
    TypeOfEdge noEdge;          //邻接矩阵中的∞值
     int find(TypeOfVer v) const {
        for (int i = 0; i < Vers; ++i)    if (ver[i] == v) return i;     }
};
```

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

A	B	C	D
0	1	2	3

构造函数：数组初始化

```
template <class TypeOfVer, class TypeOfEdge>
adjMatrixGraph<TypeOfVer, TypeOfEdge>::adjMatrixGraph
      (int vSize,  const TypeOfVer d[], TypeOfEdge noEdgeFlag)
{ int i, j;
  Vers = vSize;
  Edges = 0;
  noEdge = noEdgeFlag;  // 边不存在的标记，若边有权重，可设置特殊值。

  //存放结点的数组初始化
  ver = new TypeOfVer[vSize];
  for (i=0; i<vSize;++ i) ver[i] = d[i];
//邻接矩阵初始化
  edge = new TypeOfEdge*[vSize];
    for (i=0; i<vSize; ++ i) {
       edge[i] = new TypeOfEdge[vSize];
       for (j=0; j<vSize; ++j) edge[i][j] = noEdge;
    }
}
```

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

A	B	C	D
0	1	2	3

边的插入和删除

Insert

```
template <class TypeOfVer, class TypeOfEdge>
void adjMatrixGraph<TypeOfVer, TypeOfEdge>
        ::insert(TypeOfVer x, TypeOfVer y, TypeOfEdge w)
{
      int u = find(x), v = find(y);
      edge[u][v] = w;
      ++Edges;
}
```

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

A	B	C	D
0	1	2	3

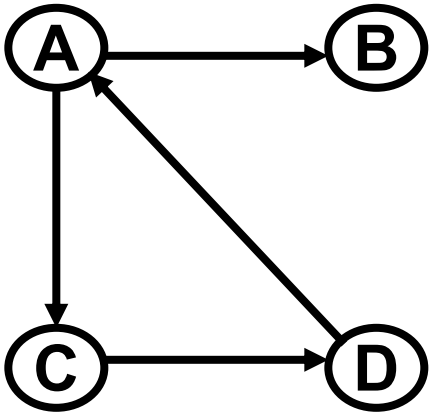
Remove：

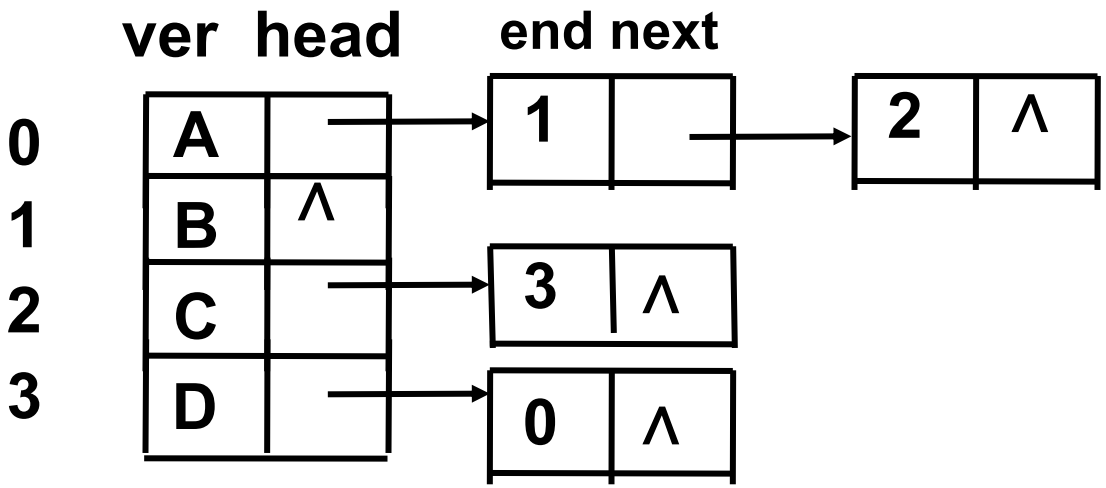
```
template <class TypeOfVer, class TypeOfEdge>
void adjMatrixGraph<TypeOfVer, TypeOfEdge>::remove(TypeOfVer x,
   TypeOfVer y)
{
      int u = find(x), v = find(y);
      edge[u][v] = noEdge;
      --Edges;
}
```

邻接表

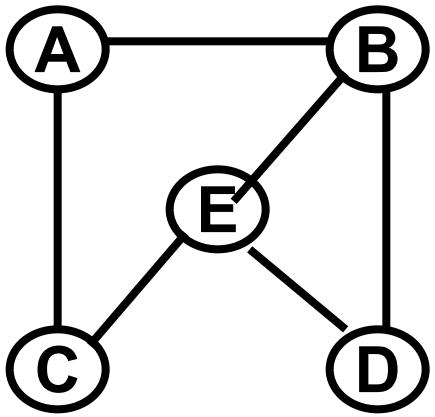
- 空间 = 结点数 + 边数，时间O(|V|+|E|)。

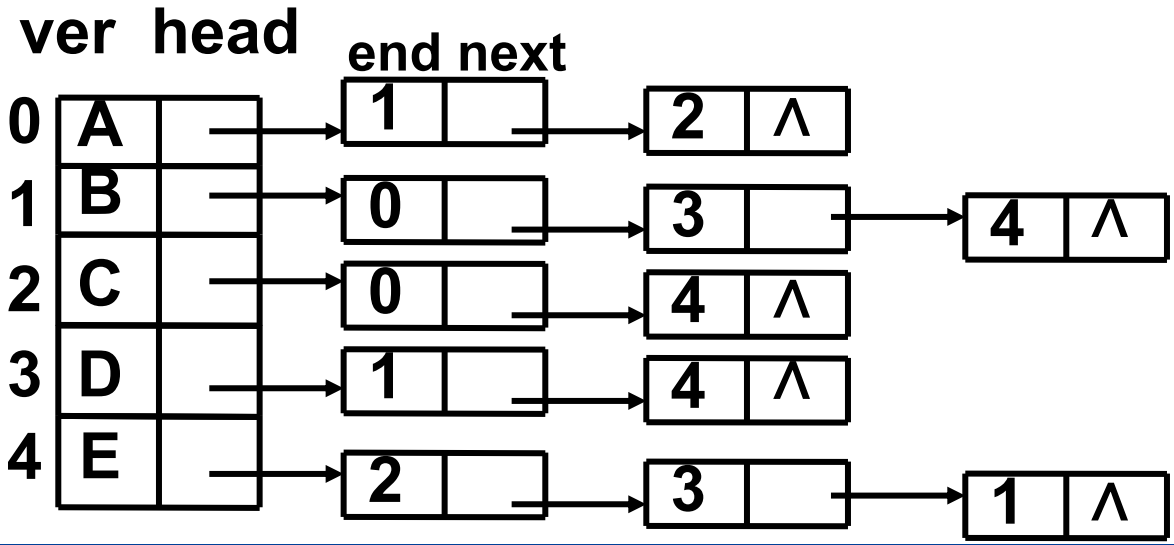
有向图 G1





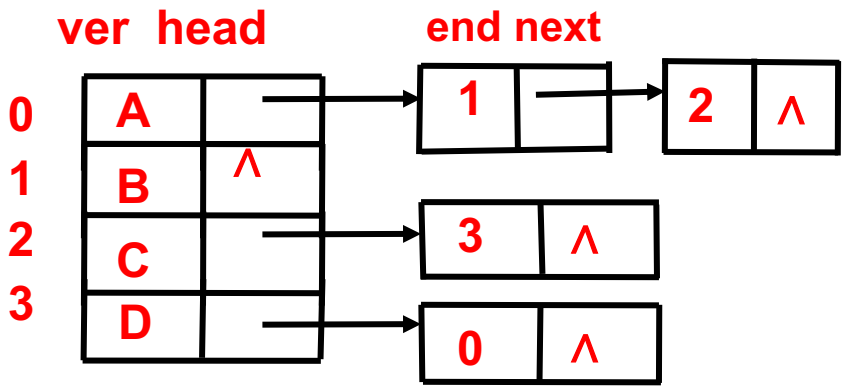
无向图 G2





邻接表类的定义

```
template <class TypeOfVer, class TypeOfEdge>
class adjListGraph:public graph<TypeOfVer,
                                TypeOfEdge> {
  public:
  adjListGraph(int vSize, const TypeOfVer d[]);
    void insert(TypeOfVer x, TypeOfVer y, TypeOfEdge w);
    void remove(TypeOfVer x, TypeOfVer y);
    bool exist(TypeOfVer x, TypeOfVer y) const;
    ~adjListGraph() ;
```

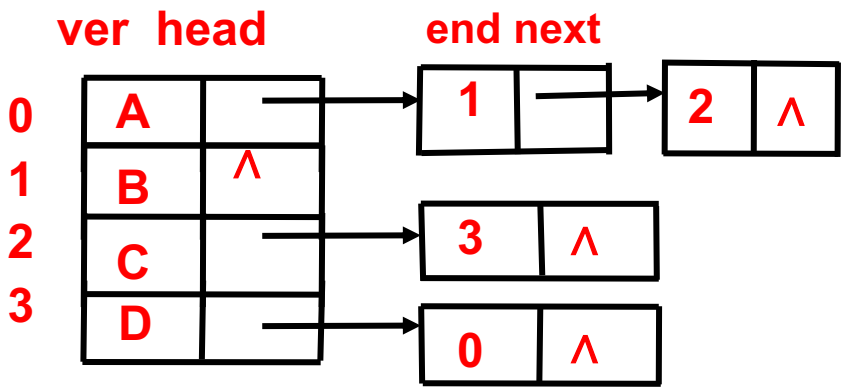


```
private:
  struct edgeNode {                  //邻接表中存储边的结点类
            int end;                    //终点存储下标
           TypeOfEdge weight;      //边的权值
           edgeNode *next;

           edgeNode(int e, TypeOfEdge w, edgeNode *n = NULL)
                    { end = e; weight = w; next = n;}
     };
  struct verNode{                    //保存顶点的数据元素类型
       TypeOfVer ver;          //顶点值
        edgeNode *head;       //对应的单链表的头指针

           verNode( edgeNode *h = NULL) { head = h;}
            };

 verNode *verList;
     int find(TypeOfVer v) const {
            for (int i = 0; i < Vers; ++i)
                    if (verList[i].ver == v) return i;
     }
};
```

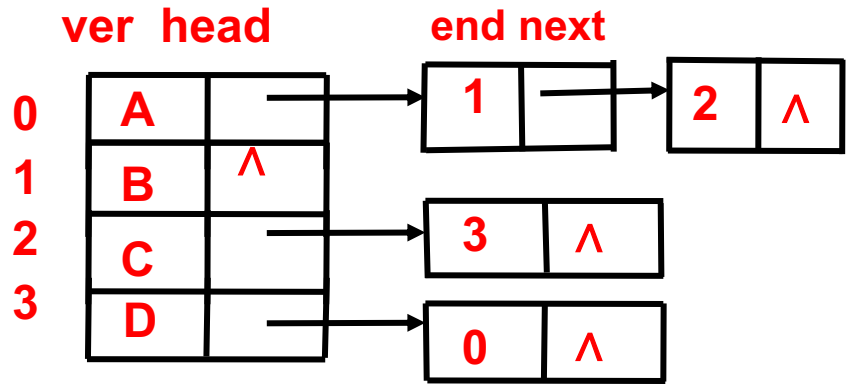


构造函数

顶点集合用顺序表存储，边用单链表存储

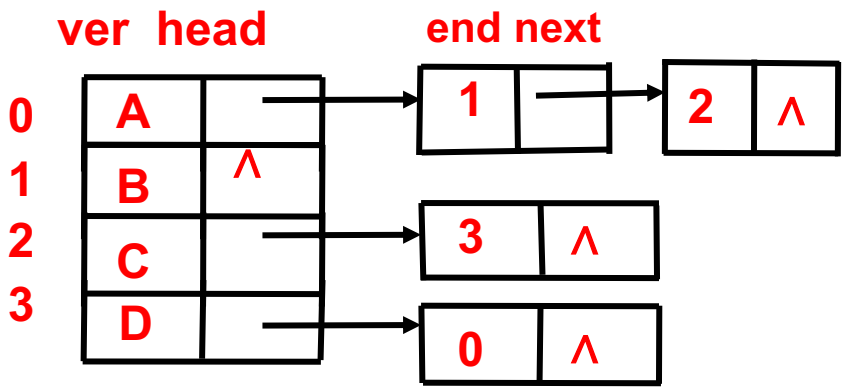
```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>
    ::adjListGraph(int vSize, const TypeOfVer d[])
{
   Vers = vSize; Edges = 0;

   verList = new verNode[vSize];
   for (int i = 0; i < Vers; ++i) verList[i].ver = d[i];
}
```



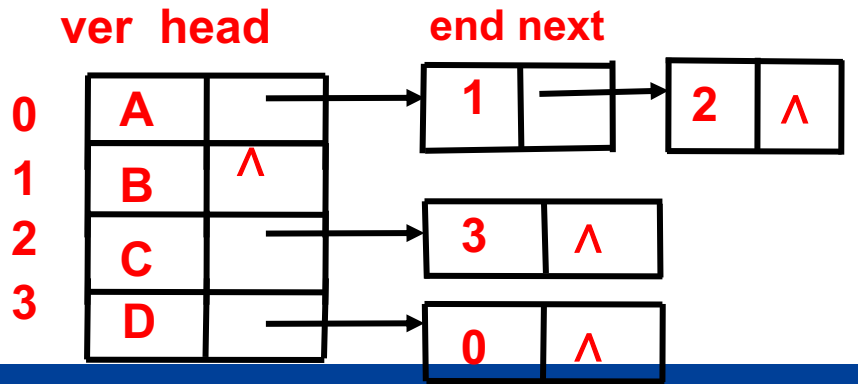
析构函数

```
template <class TypeOfVer, class TypeOfEdge>
adjListGraph<TypeOfVer, TypeOfEdge>::~adjListGraph()
{    int i;
     edgeNode *p;
    for (i = 0; i < Vers; ++i)
        while ((p = verList[i].head) != NULL)  {
            verList[i].head = p->next;
            delete p;
        }
    delete [] verList;
}
```



Insert函数（插入边）

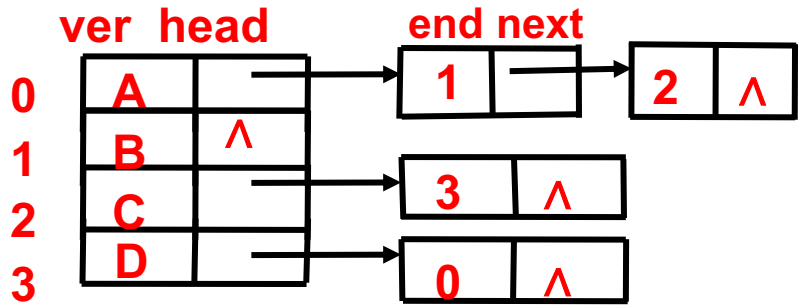
```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::
   insert(TypeOfVer x, TypeOfVer y, TypeOfEdge w)
{
   int u = find(x), v = find(y);
   verList[u].head = new edgeNode(v, w, verList[u].head );//插表头
   ++Edges;
}
```



Remove函数（删除边）类同不带表头结点的单链表

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer,TypeOfEdge>::remove(TypeOfVer x,TypeOfVer y)
{    int u = find(x), v = find(y);
   edgeNode *p = verList[u].head, *q;

    if (p == NULL) return;  //结点u没有相连的边
    if (p->end == v) {      //单链表中的第一个结点就是被删除的边
      verList[u].head = p->next;
       delete p; --Edges;
       return;
    }
```




```
    while (p->next !=NULL && p->next->end != v) p = p->next;//查找被删除的边
    if (p->next != NULL) {  //删除
       q = p->next;        p->next = q->next;        delete q;       --Edges;    }//同链表删除
}
```

Exist函数

```
template <class TypeOfVer, class TypeOfEdge>
bool adjListGraph<TypeOfVer, TypeOfEdge>
            ::exist(TypeOfVer x, TypeOfVer y) const
{
    int u = find(x), v = find(y);
   edgeNode *p = verList[u].head;

    while (p !=NULL && p->end != v) p = p->next;
    if (p == NULL) return false; else  return true;
}
```

深度优先搜索(遍历)

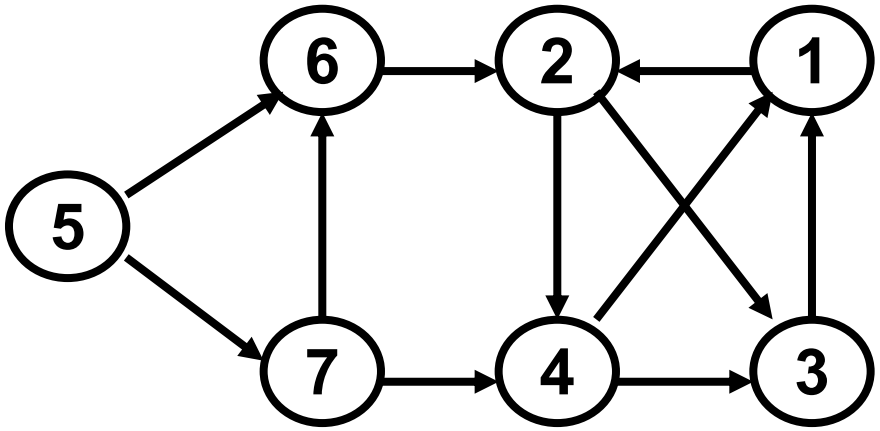
1、选中第一个被访问的顶点；

2、对顶点作已访问过的标志；

3、依次从顶点的未被访问过的第一个、第二个、第三个…… 邻接顶点出发，进行深度优先搜索；

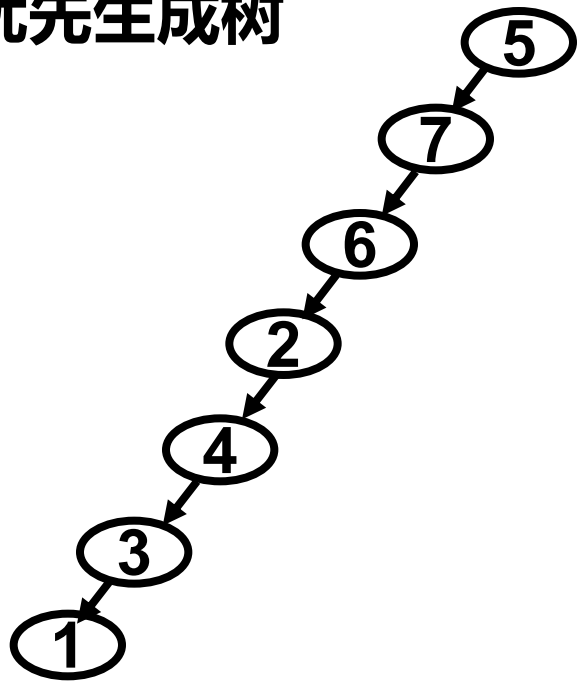
4、如果还有顶点未被访问，则选中一个起始顶点，转向2；

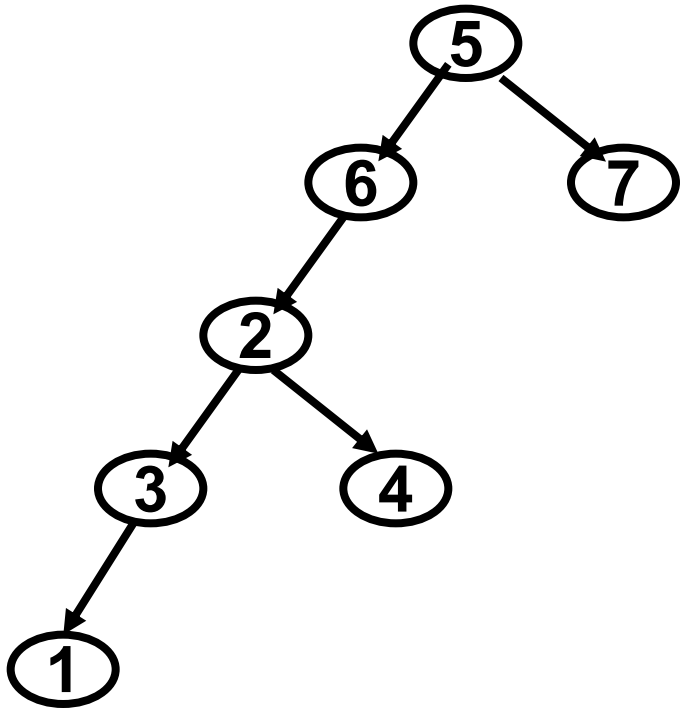
5、所有的顶点都被访问到，则结束。

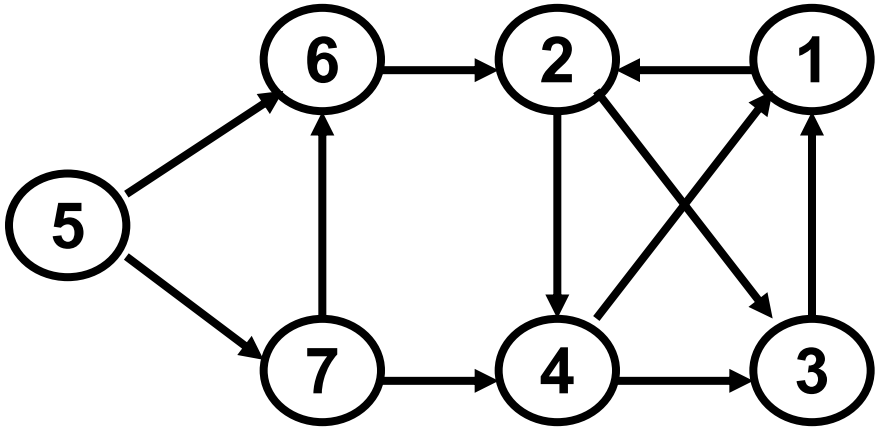


从结点5开始进行深度优先的搜索，则遍历序列可以为：5，7，6，2，4，3，1，
也可以为：
5，6，2，3，1，4，7。

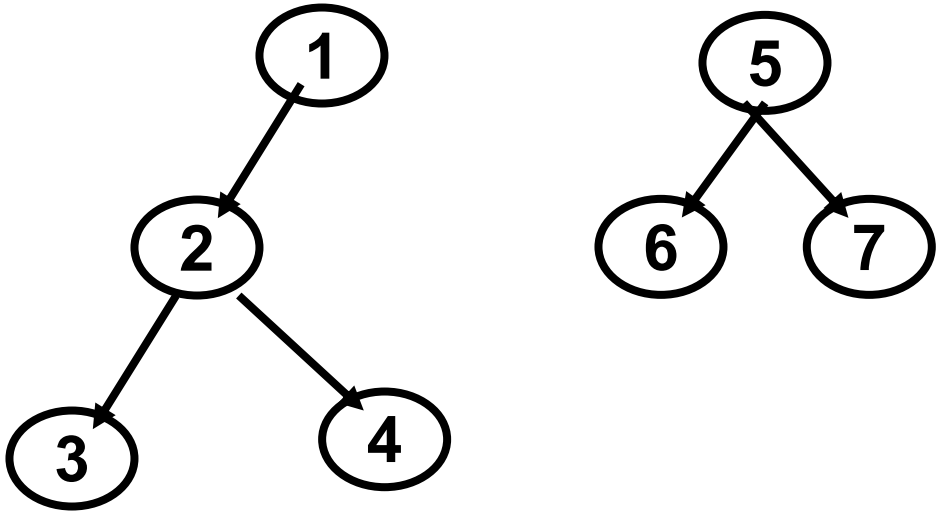
深度优先生成树







从结点1开始深度优先搜索生成森林

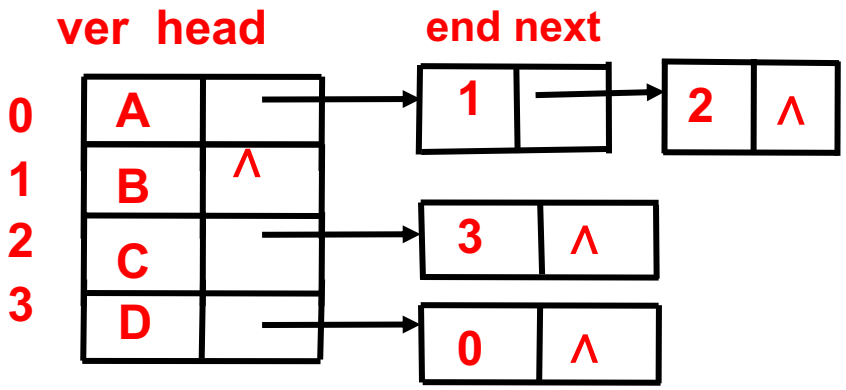


深度优先搜索算法实现

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::dfs() const
{bool *visited = new bool[Vers];

 for (int i=0; i < Vers; ++i) visited[i] = false;

 cout << "当前图的深度优先遍历序列为： " << endl;
 for (i = 0; i < Vers; ++i) {
        if (visited[i] == true) continue;
        dfs(i, visited);//调用私有DFS函数
        cout << endl;
    }
}
```



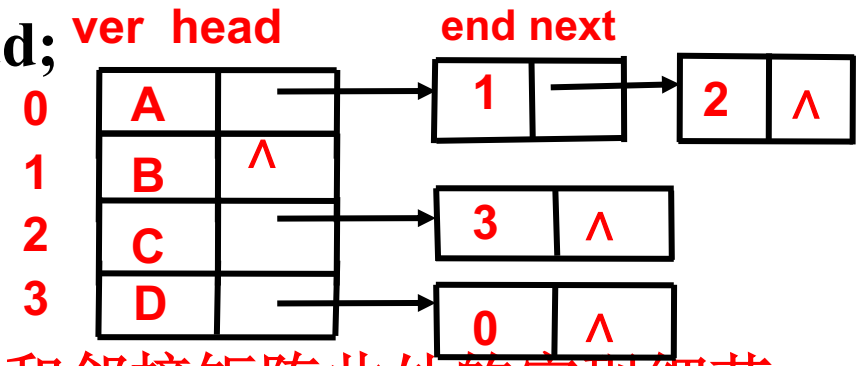
设置一个数组visited，记录节点是否被访问过

对图中尚未访问的节点反复调用深度优先搜索，形成深度优先搜索树或森林

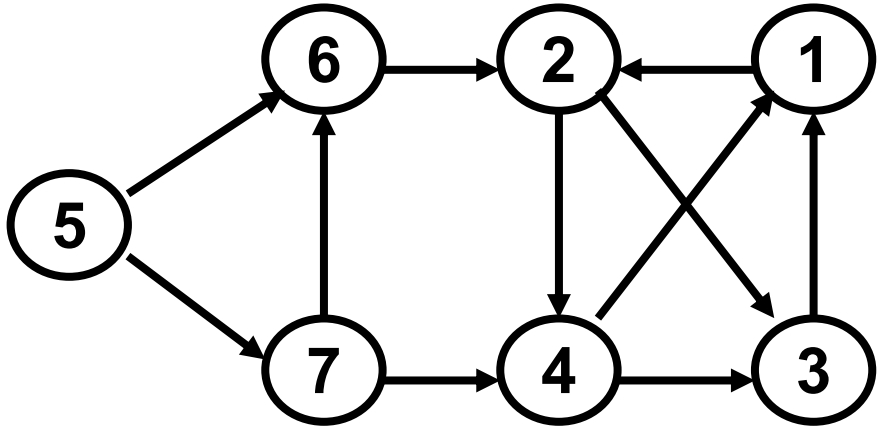
深度优先搜索算法实现（续）

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::dfs
                        (int start, bool visited[]) const
{ edgeNode *p = verList[start].head;

  cout << verList[start].ver << '\t';
  visited[start] = true;
  while (p != NULL){ //注意邻接表和邻接矩阵此处的实现细节
        if (visited[p->end] == false) dfs(p->end, visited);
        p = p->next;
  }
}
```



调用dfs



结果为：

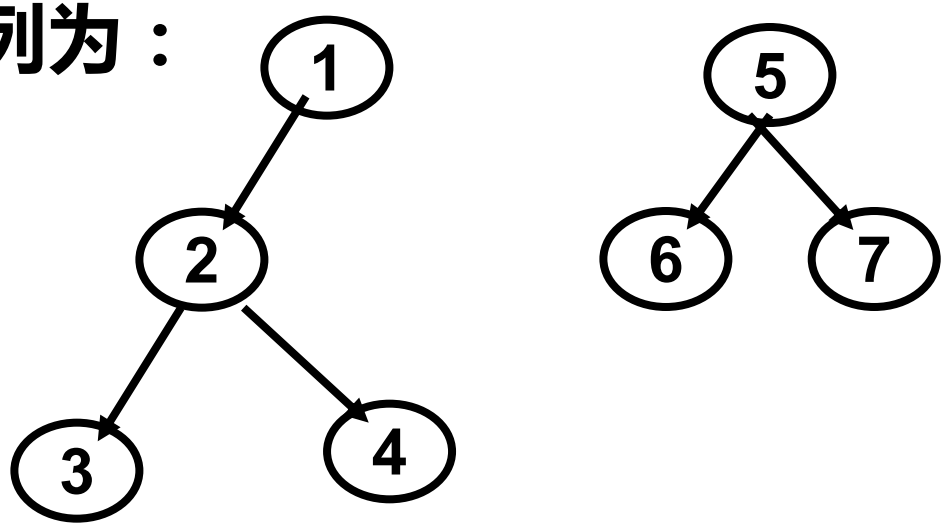
当前图的深度优先搜索序列为：

1 2 3 4

5 6 7

即对应于左图的

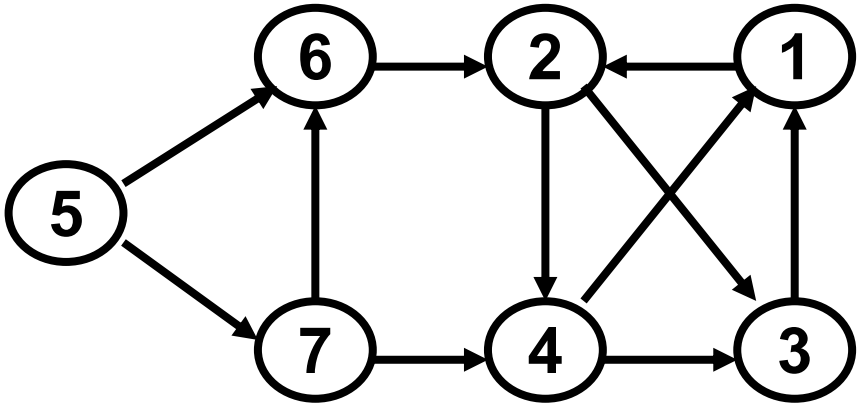
深度优先生成森林



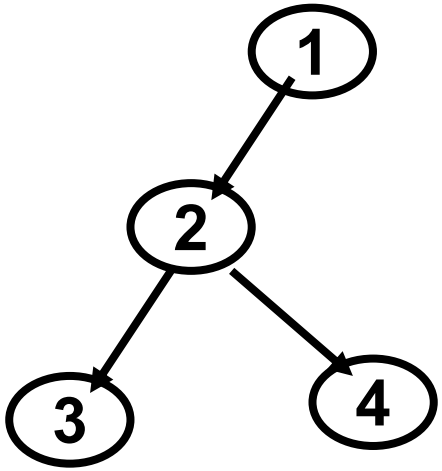
- 如果图是用邻接表来表示，则时间代价和顶点数 |V| 及边数 |E| 相关，即是O(|V|+|E|)。
- 如果图是用邻接矩阵来表示，则所需要的时间是$O(|V|^2)$。

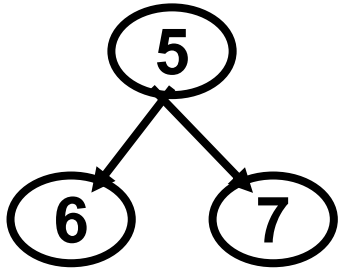
广度优先搜索

类似于树的层次遍历

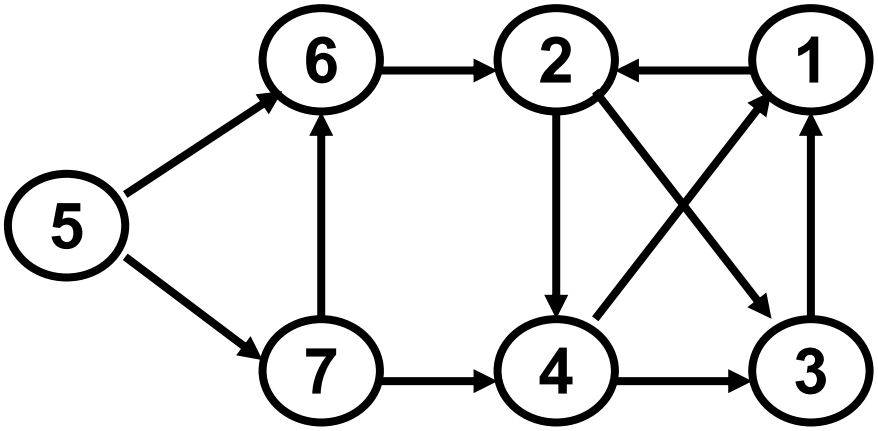


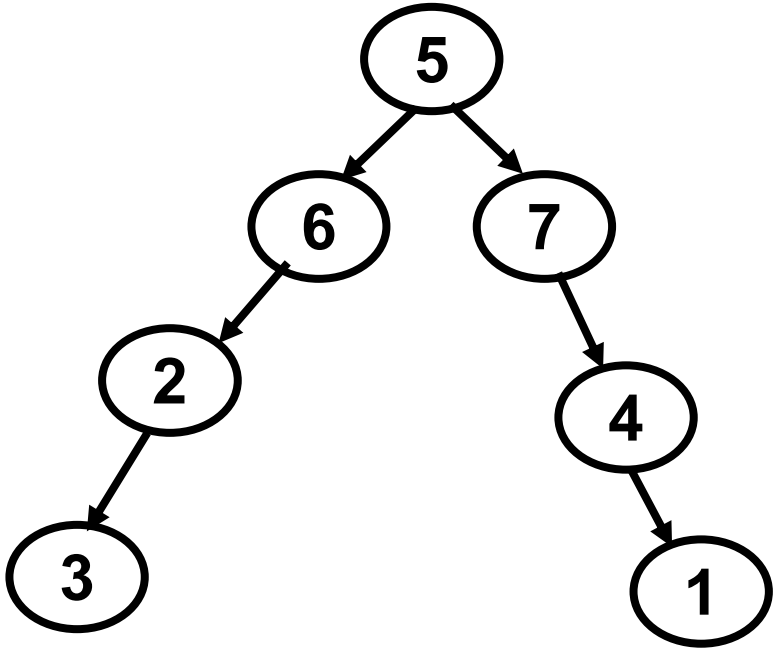
假设按照顶点序号小的先访问，序号大的后访问，则上图广度优先访问序列为：1，2，3，4，5，6，7。





从不同的结点开始可以得到不同的搜索序列。例如，从5开始得到的广度优先遍历序列为：5，6，7，2，3，4，1。



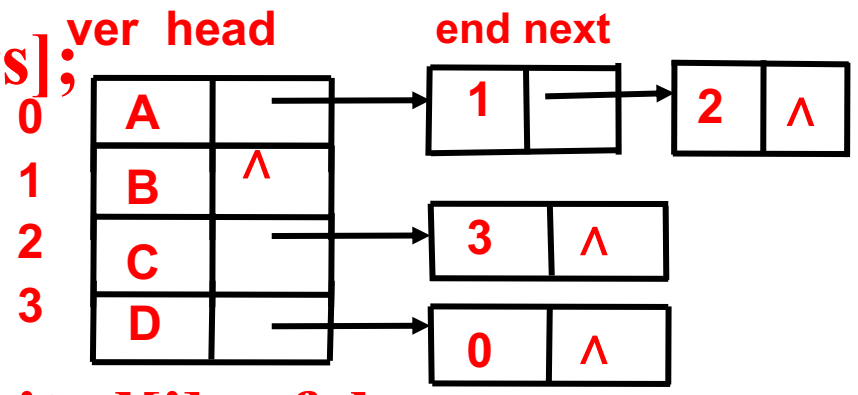


广度优先搜索算法思路

- 记录每个结点是否已被访问。
- 将当前被访问结点的后继结点，依次放入一个队列
- 重复取队列的队头元素进行处理，直到队列为空。对出队的每个元素，首先检查该元素是否已被访问。如果没有被访问过，则访问该元素，并将它的所有的没有被访问过的后继入队。
- 检查是否还有结点未被访问。如果有，重复上述两个步骤

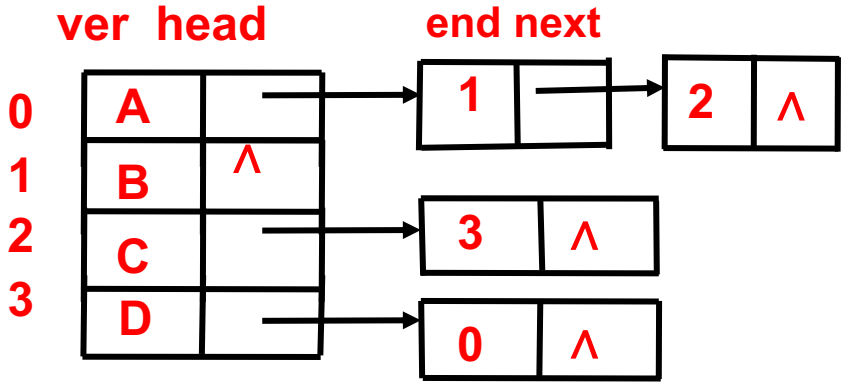
广度优先搜索（遍历）算法

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::bfs()
      const
{ bool *visited = new bool[Vers];
  int currentNode;
  linkQueue<int> q;
  edgeNode *p;
  for (int i=0; i < Vers; ++i) visited[i] = false;
  cout << "当前图的广度优先遍历序列为： "
       << endl;
//记录每个结点是否已被访问
```

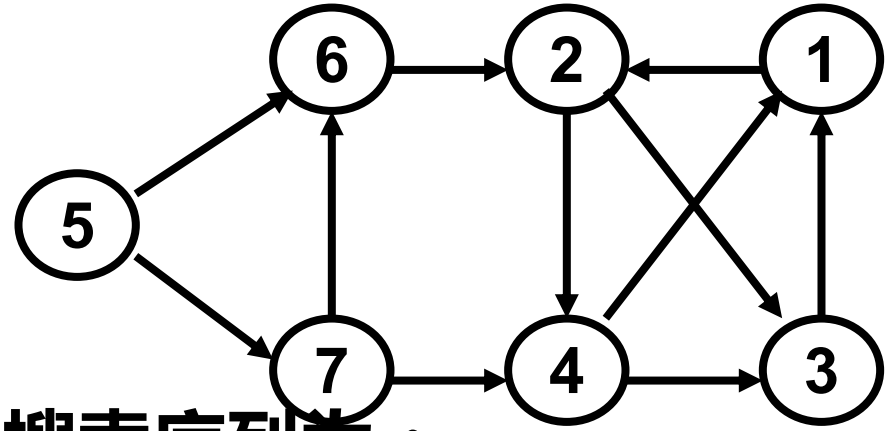


广度优先搜索算法（续）

```
for (i = 0; i < Vers; ++i)
    {
    if (visited[i] == true) continue;
     //此算法缺省从0下标结点开始遍历
    q.enQueue(i);
     while (!q.isEmpty())
            {
              currentNode = q.deQueue();
              if (visited[currentNode] == true) continue;
              cout << verList[currentNode].ver << '\t';
              visited[currentNode] = true;
              p = verList[currentNode].head;
              while (p != NULL)
                  {
                   if (visited[p->end] == false) q.enQueue(p->end);
                   p = p->next;
                   }
             }
    cout << endl;
    }
}
```



对图调用bfs



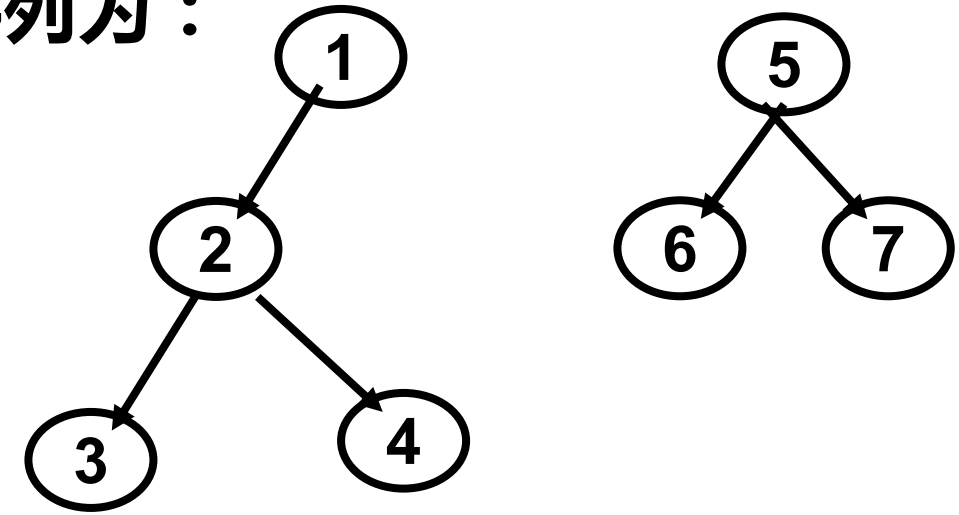
结果为：

当前图的广度优先搜索序列为：

1 2 3 4

5 6 7

广度优先生成森林



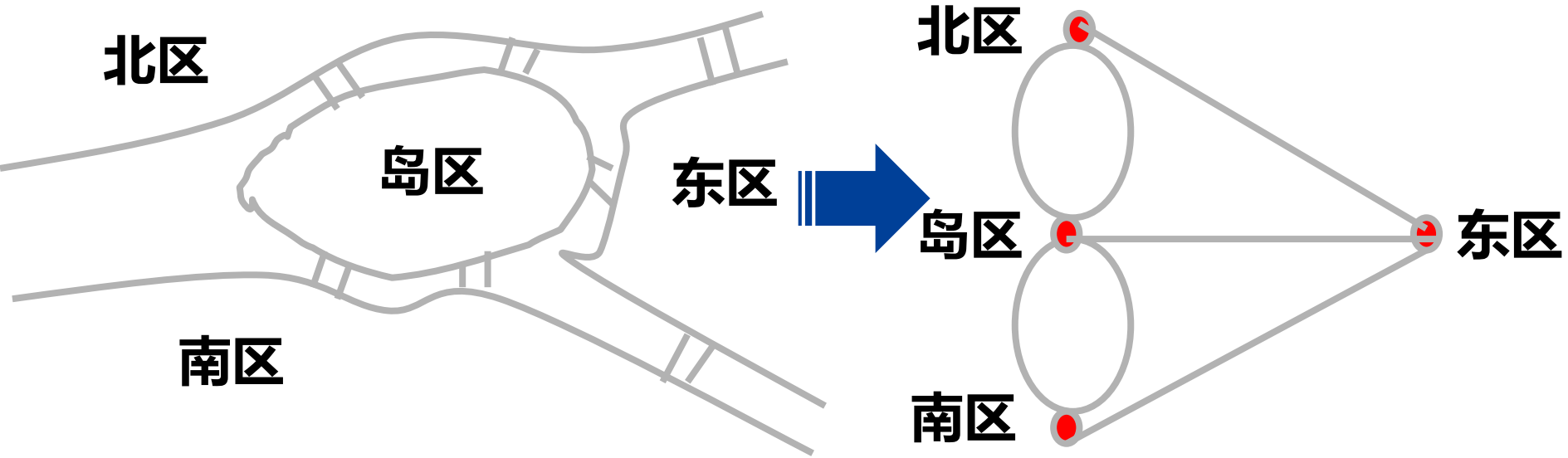
- 假如图是用邻接表存储，时间复杂度为O(|E|+|V|)
- 如果图是用邻接矩阵存储，时间复杂度为$O(|V|^2)$

无向图的连通性

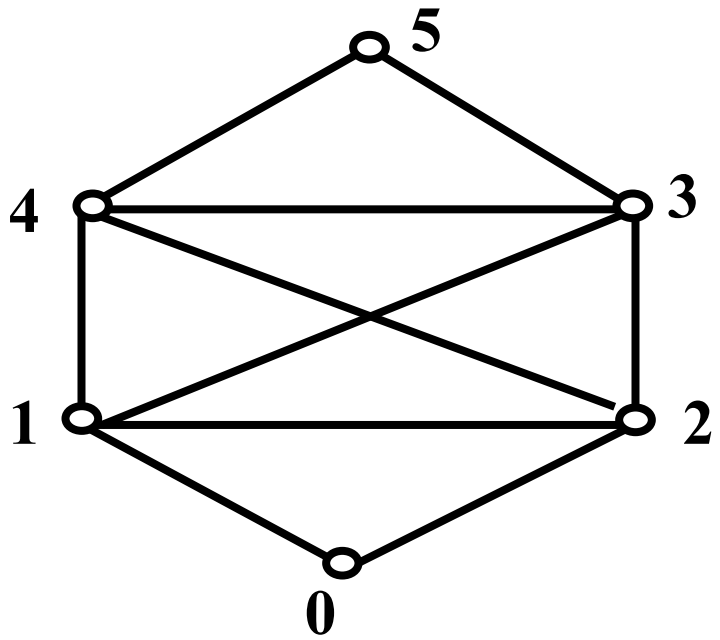
- 深度优先搜索和广度优先搜索都可以用来测试无向图的连通性。
- 如果无向图是连通的，则从无向图中的任意结点出发进行深度优先搜索或广度优先搜索都可以访问到每一个结点。访问的次序是一棵深度/广度优先生成树。
- 如果图是非连通的，深度/广度优先搜索可以找到一片深度/广度优先生成森林。每棵树就是一个连通分量。对无向图来说，深度/广度优先搜索可以找到了它的所有连通分量。

欧拉回路

- 哥尼斯堡七桥问题：能否找到一条走遍这七座桥，而且每座桥只经过一次，最后又回到原出发点的路径。



如果有奇数桥的地方不止两个，满足要求的路径是找不到的。
如果只有两个地方有奇数桥，可以从这两个地方之一出发，经过所有的桥一次，再回到另一个地方。
如果都是偶数桥，从任意地方出发都能回到原点。



从5出发，深度优先遍历，
先找到 5->4->3->5

在路径上找一个尚有边未被访问的结点，如：4，开始另一次深度优先遍历。得到路径4->2->1->4

将第二条路径拼接到第一条路径上，得到：
5->4->2->1->4->3->5

3号结点还有未访问的边，从3号结点再开始一次深度优先遍历，得到路径3->1->0->2->3

将第三条路径拼接到第一条路径上，得到：
5->4->2->1->4->3->1->0->2->3->5

寻找欧拉回路算法思路

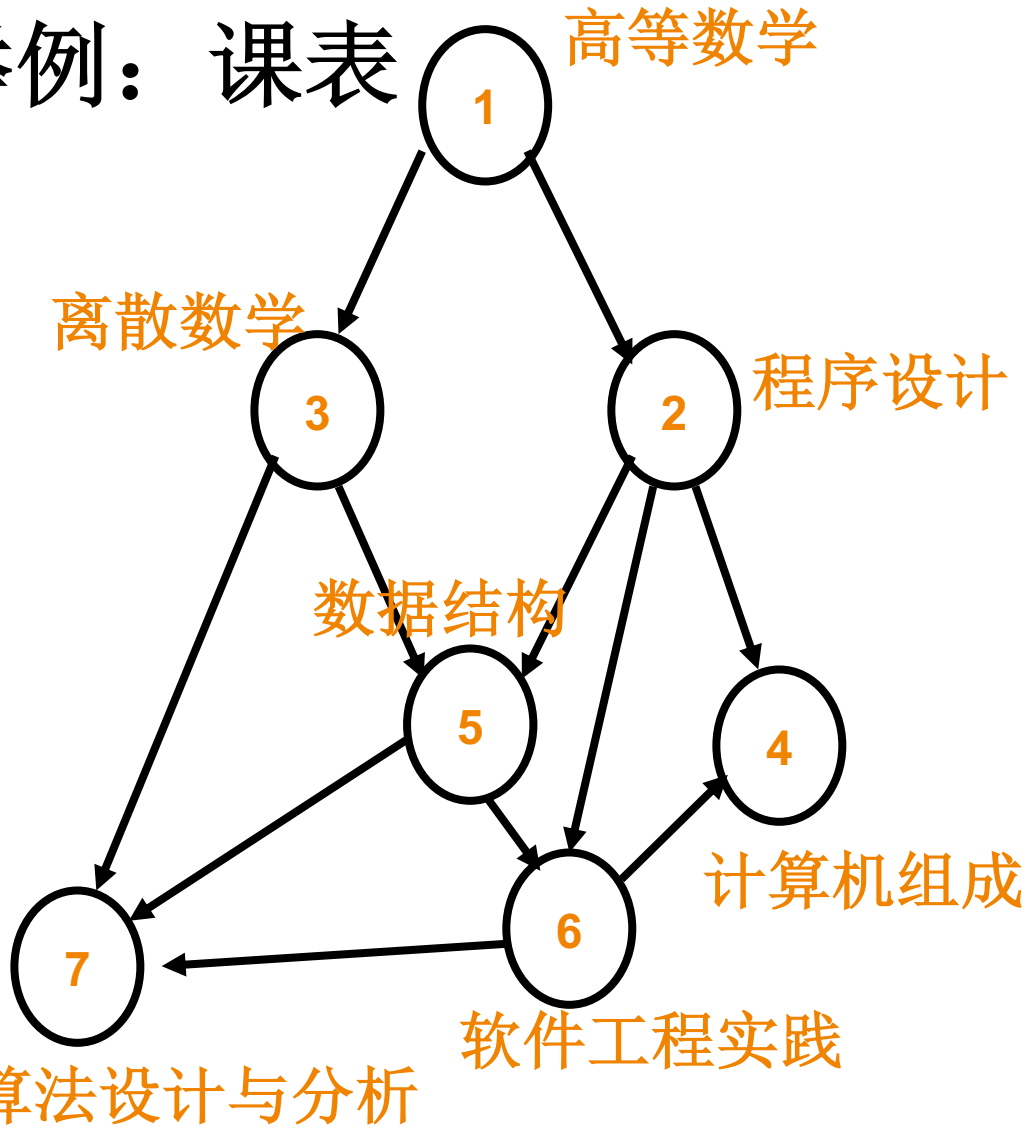
- **检查存在性**
- **找出回路：**
  - **执行一次深度优先的搜索。从起始结点开始，沿着这条路一直往下走，直到无路可走。**
  - **路径上是否有一个尚有未访问的边的顶点。如果有，开始另一次深度优先的搜索，将得到的遍历序列拼接到原来的序列中，直到所有的边都已被访问。**

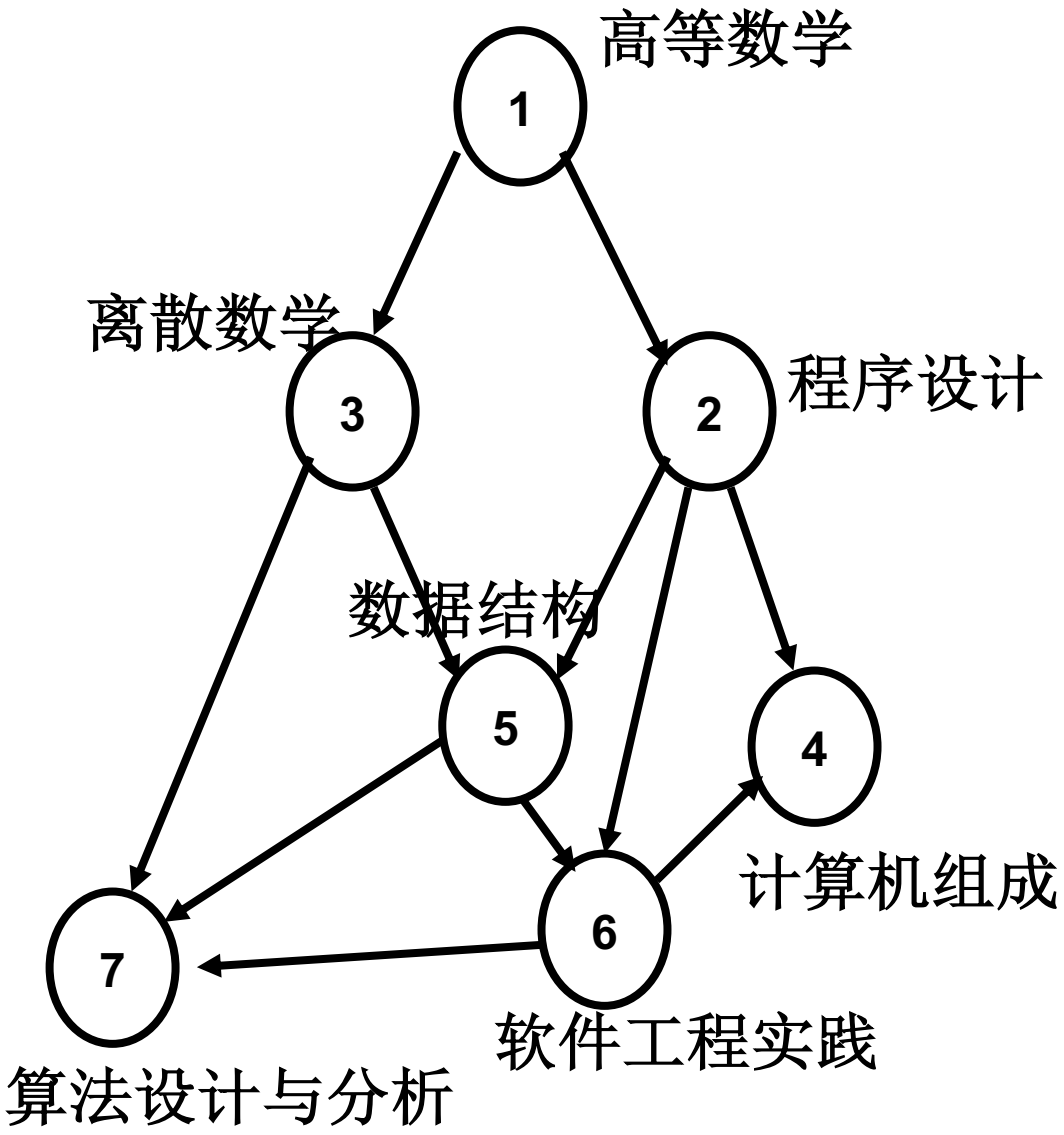
拓扑排序（AOV）

设G=（V，E）是一个具有n个顶点的有向无环图。V中的顶点序列V_1，V_2，…，V_n称为一个拓扑序列，当且仅当该序列满足下列条件：若在G中，从Vi到Vj有一条路径，则序列中Vi必须排在Vj的前面。

拓扑排序的应用举例：课表

用有向图表示课程前导关系。节点集为课程集合。如果课程i是j的前导课程，则有一条边。





可行的排课：

方案1：

1，3，2，5，6，7，4

方案2：

1，2，3，5，6，4，7

方案3：

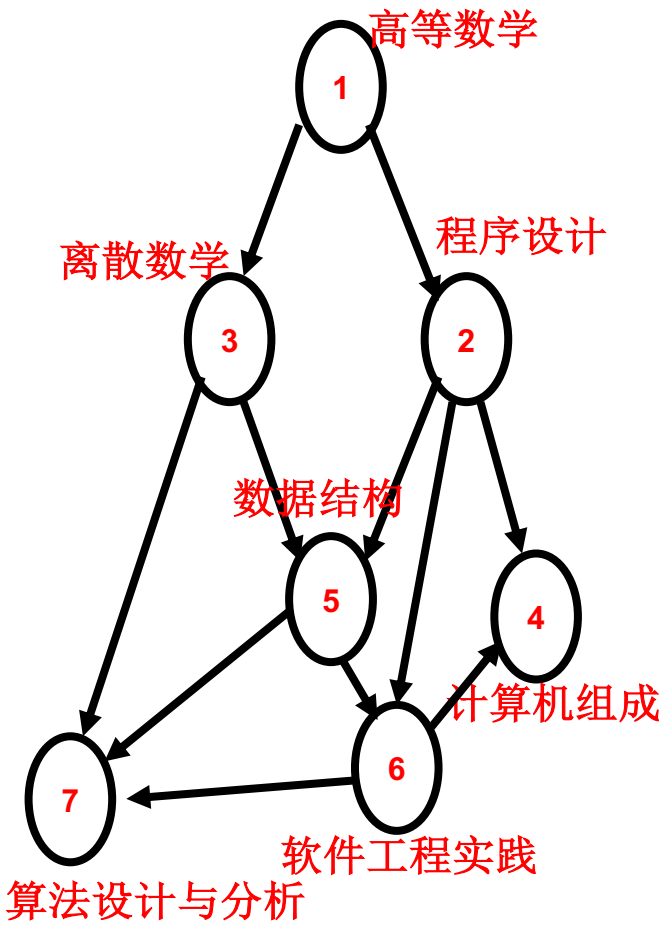
1，2，3，5，6，7，4

……

找出拓扑排序的过程

- 第一个输出的结点（序列中的第一个元素）：必须无前驱，即入度为0
- 后驱：必须等到它的前驱输出之后才输出。
- 无前驱及后件的结点：任何时候都可输出。
- 逻辑删除法：当某个节点被输出后，就作为该节点被删除。所有以该节点作为前驱的所有节点的入度减1。

高等数学	0						
程序设计	1	0					
离散数学	1	0	0				
计算机组成	2	2	1	1	1	0	
数据结构	2	2	1	0			
软件工程实践	2	2	1	1	0		
算法设计与分析	3	3	3	2	1	0	0



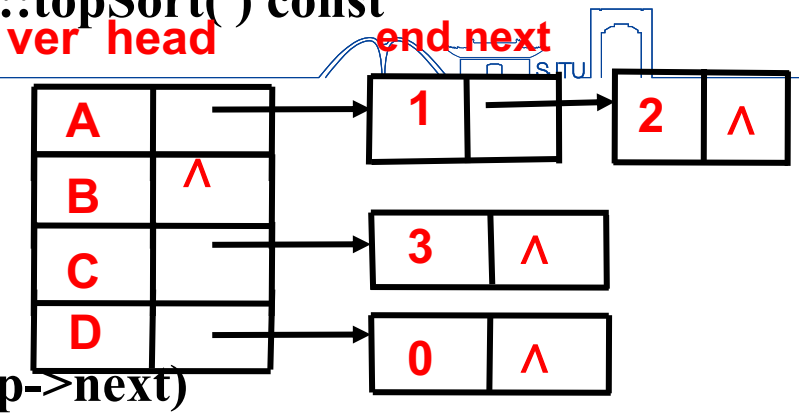
输出：

高等数学，程序设计，离散数学，数据结构， 软件工程实践，计算机组成，算法设计与分析

拓扑排序的实现

- 计算每个结点的入度，保存在数组inDegree中；
- 检查inDegree中的每个元素，将入度为0的结点入队（考虑一下，入栈行不行？）；
- 不断从队列中将入度为0的结点出队，输出此结点，并将该结点的后继结点的入度减1；如果某个邻接点的入度为0，则将其入队。

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::topSort( ) const
{  linkQueue<int> q;
   edgeNode *p;
   int current, *inDegree = new int[Vers];
   for (int i = 0; i < Vers; ++i)   inDegree[i] = 0;
   for ( i = 0; i < Vers; ++i)
        for (p = verList[i].head; p != NULL; p = p->next)
            ++inDegree[p->end];
      }
   for (i = 0; i < Vers; ++i)   if (inDegree[i] == 0) q.enQueue(i);
   cout << "拓扑排序为：  " << endl;
   while( !q.isEmpty( ) ){
       current = q.deQueue( );
       cout << verList[current].ver << '\t';
       for (p = verList[current].head; p != NULL; p = p->next)
           if( --inDegree[p->end] == 0 )    q.enQueue( p->end );
    }
   cout << endl;
}
```

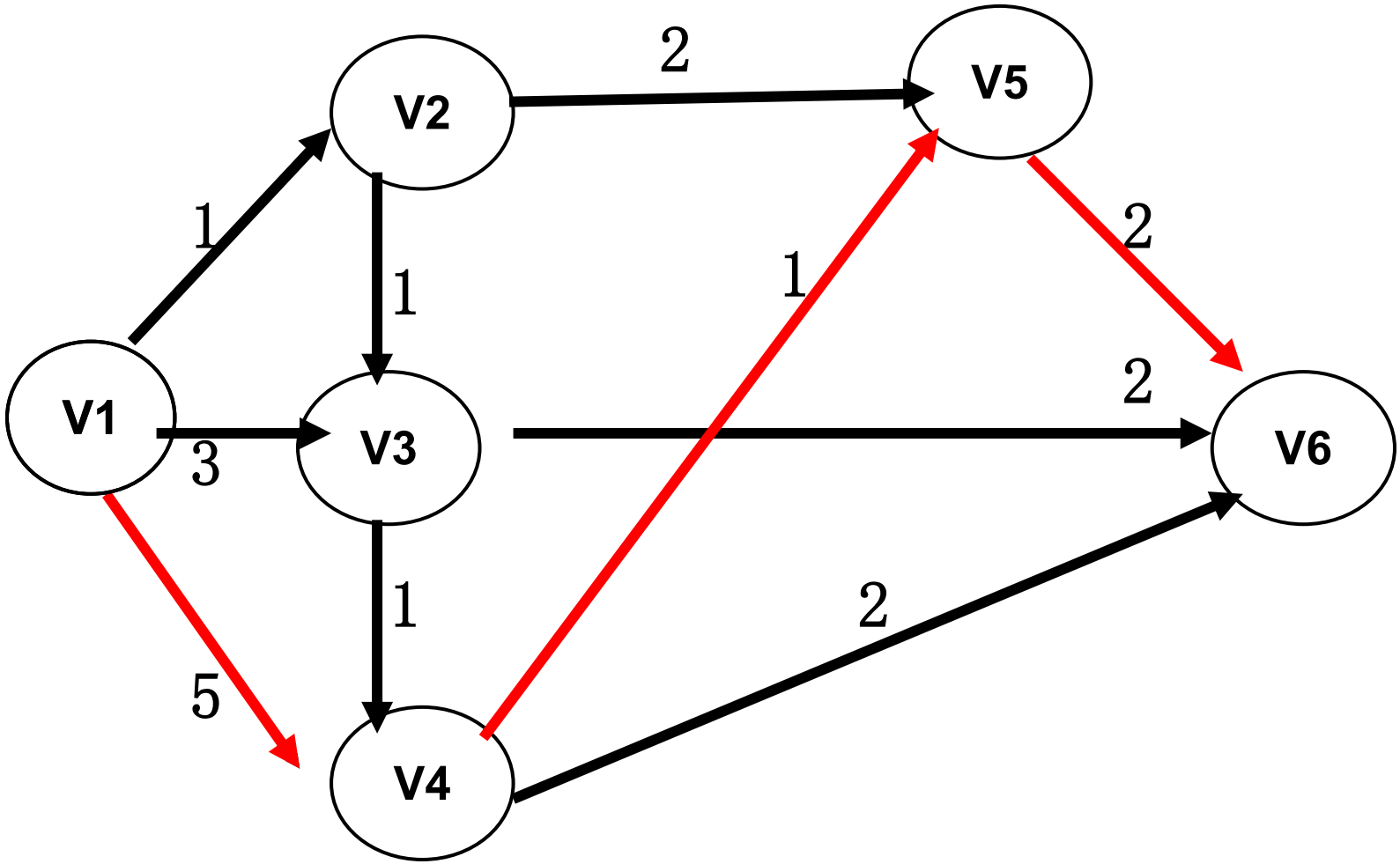


时间复杂度

- 如果图以邻接表表示
- 计算入度需要O（|V|+|E|）的时间，搜索入度为0的结点需要O（|V|）的时间。每个结点入一次队、出一次队。每出一次队，需要检查它的所有后继结点，因此也需要O（|V|+|E|）的时间。所以总的执行时间也是O（|V|+|E|）

关键路径：计算工期和关键活动

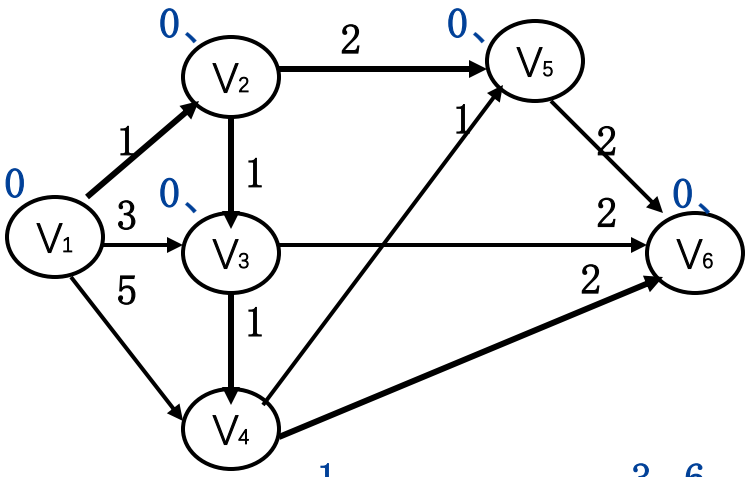
- **AOE网络**：顶点表示事件，有向边的权值表示某个活动的持续时间，有向边的方向表示事件发生的先后次序
- **AOE网络可用于描述整个工程的各个活动之间的关系，活动安排的先后次序。在此基础上，可以用来估算工程的完成时间以及那些活动是关键的活动。**

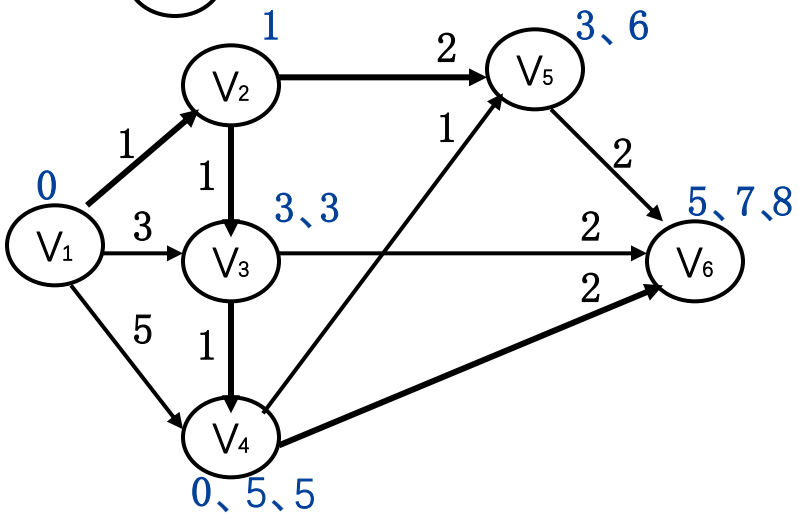


关键路径：（v1,v4,v5,v6），路径长度是8

- **完成整项工程至少的时间：起点**到终点的最长路径，即**关键路径**。
- **影响工程进度的活动：**活动时间余量为0的活动，称为**关键活动**。

关键路径





正向拓扑排序：

利用正向拓扑排序求最早发生时间

利用拓扑排序算法求事件结点的最早发生时间的执行步骤：

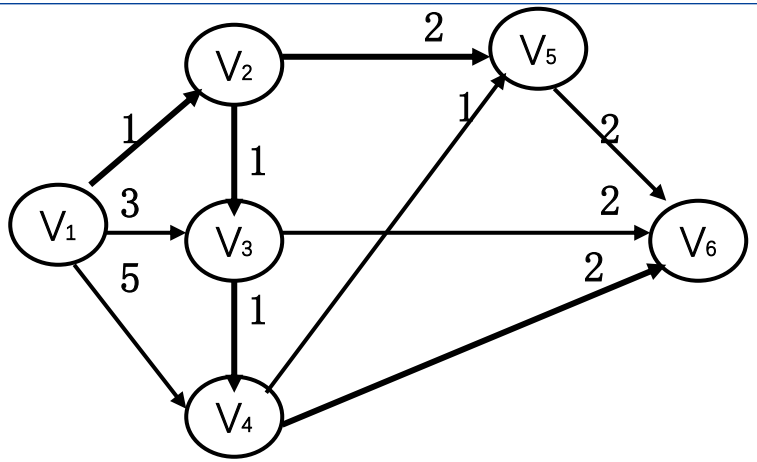
1、设每个结点（始发边）的最早发生时间为0，将入度为零的结点进栈。

2、将栈中入度为零的结点V取出，并压入另一栈，用于形成逆向拓扑排序的序列。。

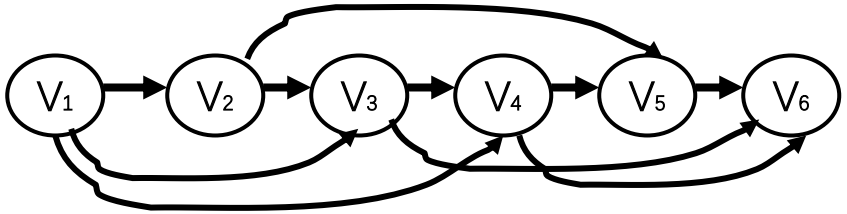
3、根据邻接表找到结点V的所有的邻接结点，将结点V的最早发生时间 + 活动的权值得到的和同邻接结点的原最早发生时间进行比较；如果该值大，则用该值取代原最早发生时间。另外，将这些邻接结点的入度减一。如果某一结点的入度变为零，则进栈。

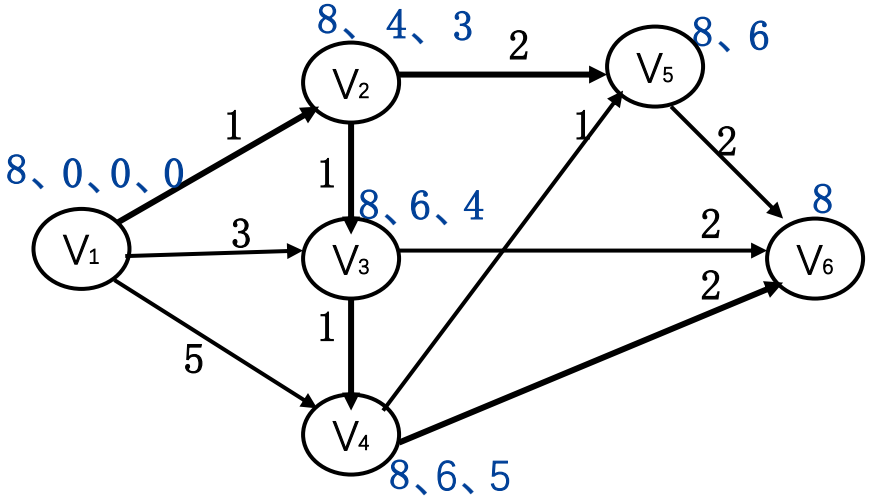
4、反复执行 2、3；直至栈空为止。

关键路径

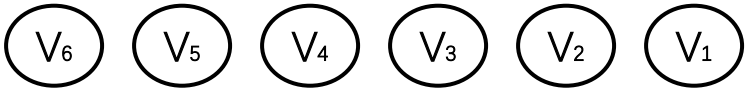


利用逆向拓扑排序求事件结点的最迟发生时间





逆向拓扑排序：



找关键路径过程

- 找出每个顶点的最早发生时间和最迟发生时间
  - 最早发生时间：每个直接前驱的最早发生时间加上从该前驱到该顶点的活动时间的最大者
  - 最迟发生时间：每个直接后继的最迟发生时间减去顶点到该直接后继的活动时间的最小者就是该顶点的最迟发生时间。
- 找出两个时间相等的顶点就是关键路径上的顶点

关键路径算法


```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>
                                ::criticalPath( ) const
{
   TypeOfEdge *ee = new TypeOfEdge[Vers],
                       *le = new TypeOfEdge[Vers];
   int *top = new int[Vers], *inDegree = new int[Vers];
   linkQueue<int> q;
   int i;
   edgeNode *p;
```

关键路径算法（续）

```
// 找出拓扑序列，放入数组top
for (i = 0; i < Vers; ++i) {      //计算每个结点的入度
        inDegree[i] = 0;
        for (p = verList[i].head; p != NULL; p = p->next)
                ++inDegree[p->end];
}
for (i = 0; i < Vers; ++i)          //将入度为0的结点入队
       if (inDegree[i] == 0) q.enQueue(i);
i = 0;
     while( !q.isEmpty( ) ) {
        top[i] = q.deQueue( );
        for (p = verList[top[i]].head; p != NULL; p = p->next)
       if( --inDegree[p->end] == 0 )    q.enQueue( p->end );
         ++i;
 }
```

关键路径算法（续）

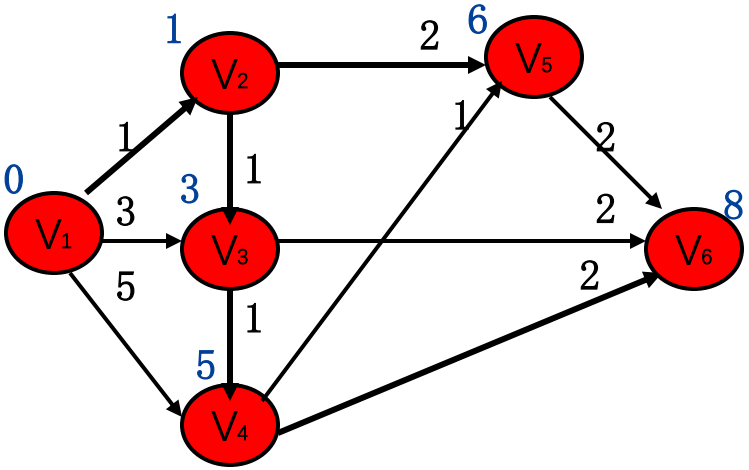
```
// 找最早发生时间
for (i = 0; i < Vers; ++i) ee[i] = 0;
for (i = 0; i < Vers; ++i) {          // 找出最早发生时间存于数组ee
  for (p = verList[top[i]].head; p != NULL; p = p->next)
        if (ee[p->end] < ee[top[i]] + p->weight )
           ee[p->end] = ee[top[i]] + p->weight;
}
 // 找最晚发生时间
for (i = 0; i < Vers; ++i) le[i] = ee[Vers -1];
for (i = Vers - 1; i >= 0 ; --i)        // 找出最晚发生时间存于数组le
   for (p = verList[top[i]].head; p != NULL; p = p->next)
        if(le[p->end] - p->weight < le[top[i]]  )
             le[top[i]] = le[p->end] - p->weight;
```

关键路径算法（续）

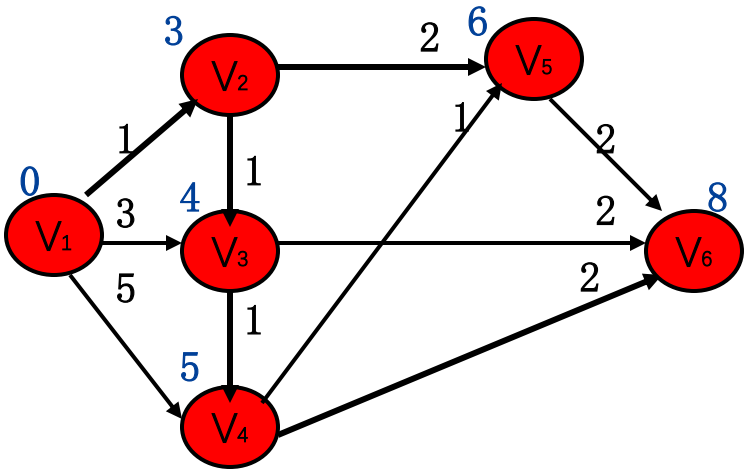
```
// 找出关键路径
   for (i = 0; i < Vers; ++i)
      if (le[top[i]] == ee[top[i]])
         cout << "(" << verList[top[i]].ver
              << ",  " << ee[top[i]] << ")  ";
}
```

关键路径

- 实例的事件结点的最早发生时间



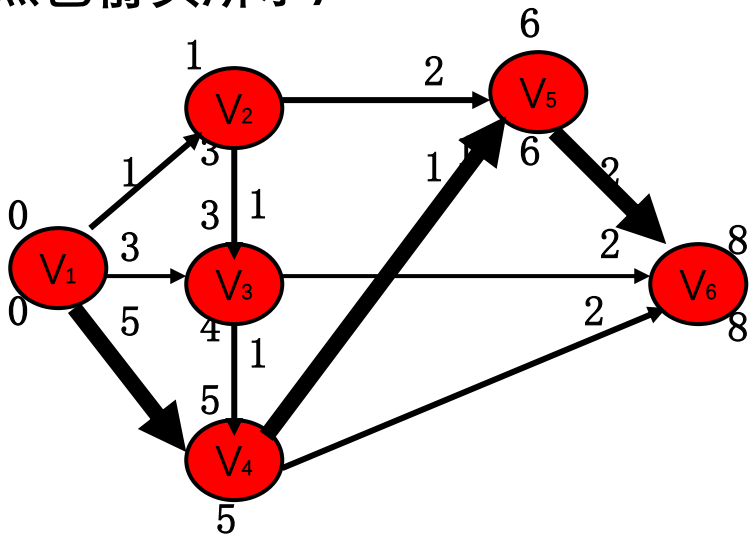
- 实例的事件结点的最迟发生时间



边	最早发生时间	最迟发生时间	
$V_1 \to V_2$	0	2	
$V_1 \to V_3$	0	1	
$V_1 \to V_4$	0	0	关键活动
$V_2 \to V_3$	1	3	
$V_2 \to V_5$	1	4	
$V_3 \to V_4$	3	4	
$V_3 \to V_6$	3	6	
$V_4 \to V_5$	5	5	关键活动
$V_4 \to V_6$	5	6	
$V_5 \to V_6$	6	6	关键活动

关键路径

- 实例的关键路径（粗黑色箭头所示）



关键路径可能有多条

•缩短工期必须缩短关键活动所需的时间

- 时间代价：O(e+n)

总结

- 图是一种有着广泛用途的数据结构。
- 图主要用邻接矩阵、邻接表进行存储。
- 图的遍历：深度优先搜索和广度优先搜索。
- 图的应用：
- 检测无向图的连通性
- 寻找无向图的欧拉回路
- 拓扑排序与关键路径

Thanks! & QA