Chapter 07 Graph

第七章 图

本章学习的线索

- 主要线索

图的定义、表示和存储 → 图的遍历及生成树 → 最小代价生成树和最短路径 → 拓扑排序关键路径

- 重点
  - 图的定义及表示
  - 图的遍历和生成树
  - 最小代价生成树和最短路径
  - 拓扑排序
- 难点
  - 图的遍历和最短路径

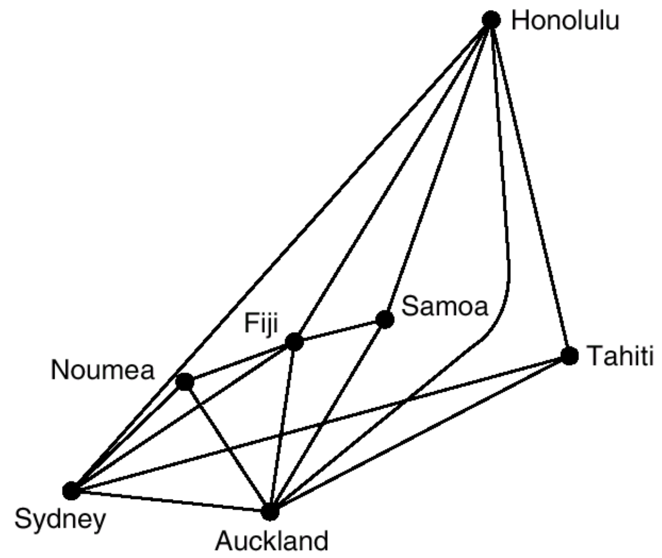
Contents

- Definition and notations of graph
- Storage structure of graph
- Graph traversal
- Connected component and spanning tree
- Mini spanning tree
- Shortest path
- Topological sorting & Critical path
- Conclusion

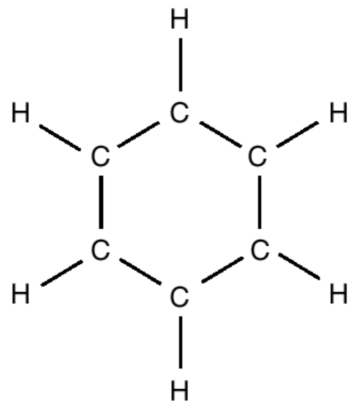
- 图 (Graph)是一种较线性结构和树更为复杂的数据结构。
- 在线性表中，一个元素只能和其直接前驱或直接后继相关；
- 在树中，一个结点可以和其下一层的所有孩子结点相关，以及上一层的双亲结点相关，但不能和其他的任何结点直接相关；
- 而对于图来说，图中任意两个结点之间都可以直接相关，因此图形结构非常复杂。
- 但是图的用途也极其广泛，已渗入到语言学、逻辑学、物理、化学、电讯工程、计算机科学以及数学等其他分支学科当中。

在本课程，我们主要讨论如何在计算机上实现图的操作，因此主要学习图的存储结构以及图的若干操作的实现。

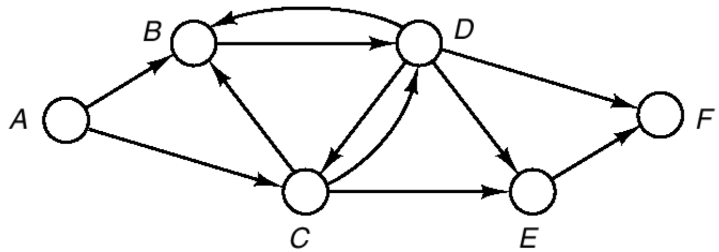
Example



Selected South Pacific air routes



Benzene molecule



Message transmission in a network

Contents

- Definition and notations of graph
- Storage structure of graph
- Graph traversal
- Connected component and spanning tree
- Mini spanning tree
- Shortest path
- Topological sorting & Critical path

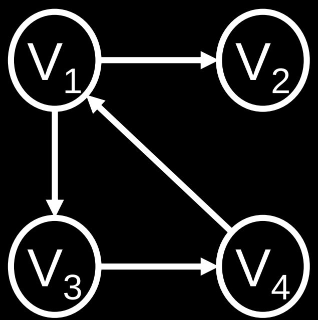
7.1 Definition and notations of graph

定义：图是一种网状的数据结构，结点之间的关系是任意的，即图中任何两个结点之间都可能直接相关。

Vertex (顶点)：图中的数据元素。设它的集合用V表示。
Arc (弧)：设两个顶点之间关系的集合用VR(Vertex Relationship)来表示，且v, w∊V，若<v, w>∊VR，则<v, w>表示从v到w的一条弧(Arc)。这里称v为弧尾(Tail)，w为弧头(Head)。

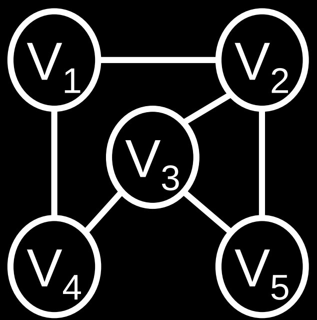
此时的图称为Directed graph (有向图)。
Undirected graph (无向图)：若<v, w> ∊VR必能推导出<w, v> ∊VR，即VR是对称的，则用无序对(v, w)代替有序对，表示v和w之间的一条边(Edge)。此时的图称为无向图。

二元组G=(V, A)表示有向图，其中
$V=\{v_1, v_2, \ldots, v_n\}$,
$A=\{<v_1,v_2>, <v_1,v_3>, \ldots <v_{n-1}v_n>\}$



有向图

二元组G=(V, E) 表示无向图，其中
$E=\{(v_1,v_2), (v_1,v_3), \ldots (v_{n-1}v_n)\}$



无向图

在以后的讨论中，我们不考虑顶点到其自身的弧或边，那么对于有n个顶点的无向图，边的最大数目为n(n-1)/2；对于n个顶点的有向图，弧的最大数目为n(n-1)。由此我们可以得到下面的定义：

Concepts of Graph

1. Categorization
2. Weight related & sub-graph
3. Adjacency related
4. Path related
5. Connectivity related
6. Spanning tree & forest

1. Categorization

Complete graph (完全图)：有n(n-1)/2条边的无向图。

Directed Complete graph (有向完全图)：有n(n-1)条边的有向图。

Sparse Graph **(稀疏图)**：有很少条边或弧的图(e < nlogn)

Dense Graph **(稠密图)**：(e>=nlogn)

2. Weight related & sub-graph

权 (Weight)：与图的边或弧相关的数值。

网 (Network)：带权的图。

Subgraph (子图)：设两个图G＝(V, {E})和G'=(V',{E'})，如果 V'⊆V且 & E' ⊆E，则称G'为G的子图。

3. Adjacency related

邻接点：对于无向图G＝(V,{E})，如果边(v, w)∈E，则称顶点v和w互为邻接点(Adjacent)；边(v, w)依附于(Incident)顶点v和w，或者说，v和w相关联。

Degree (顶点的度D)：和该顶点相关联的边的数目。

OutDegree (顶点的出度OD)：以该顶点为弧尾的弧的数目。

InDegree (顶点的入度ID)：以该顶点为弧头的弧的数目。

$$D(v_i) = OD(v_i) + ID(v_i)$$

$$e = \sum_{i=1}^{n} D(v_i)/2$$

$$e = \sum_{i=1}^{n} ID(v_i) = \sum_{i=1}^{n} OD(v_i)$$

4. Path related

Path (路径)：在图中从顶点v到顶点w所经过的所有顶点的序列。

Simple path (简单路径)：序列中顶点不重复出现的路径。

Loop (回路或环)：第一个顶点和最后一个顶点相同的路径。

Simple Loop (简单回路或环)：除第一个和最后一个顶点，其余顶点不重复出现的路径。

Rooted graph (有根图)：在有向图中，若存在一顶点v，从该顶点有路径可以到图中其它所有顶点，则称此有向图为有根图，v称为图的根。

5. Connectivity related

Connected (连通)：在无向图中，如果从v到w存在路径，则称v和w是连通的。

Connected graph (连通图)：无向图G中如果任意两个顶点v_i,v_j之间都是连通的，则称图G是连通图。

Connected component (连通分量)：无向图中的*极大连通子图*。

Strong connected graph (强连通图)：在有向图G中，如果对于每一对$v_i,v_j \in V$, $v_i \neq v_j$，从v_i到v_j和从v_j到v_i都存在路径，则称G是强连通图。

Strong connected component (强连通分量)：有向图中的极大强连通子图。

6. Spanning tree & forest

连通图的**生成树**：是**连通图**的一个*极小连通子图*，它含有图中的全部n个顶点，但只有足以构成一棵树的n-1条边。因此，对于生成树而言，只要再增加一条边，就会出现环。而如果图中有n个顶点，小于n-1条边，则该图是非连通图；但有n-1条边的图却不一定是生成树。

如果一个有向图恰有一个顶点入度为0，其余顶点入度均为1，则该图必定是一棵有向树。所有树可以看成是图的特例。

有向图的**生成森林**：由若干棵有向树组成，含有图中全部顶点，但只有构成若干棵不相交的有向树的弧。

7.2

图的存储结构

图的存储方式

一、邻接矩阵（重点）

二、邻接表 （重点）

三、十字链表

四、邻接多重表

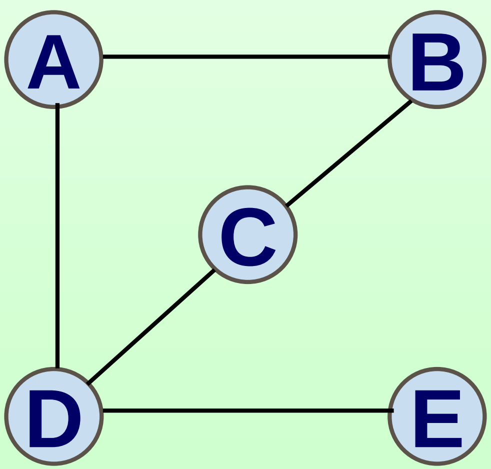
一、图的邻

TD(Vi)： **第i行**非零元素的个数，或
第i列非零元素的个数

- 一维数组：用于
- 二维数组：用于存储图中顶点之间关联关系–邻接矩阵

定义:矩阵的元素为

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



无向图

$$\begin{array}{c c} & \begin{array}{ccccc} A & B & C & D & E \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \\ E \end{array} & \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}$$

对称矩阵

OD(Vi)： 第i行非零元素的个数，

ID(Vi): 第i列非零元素的个数

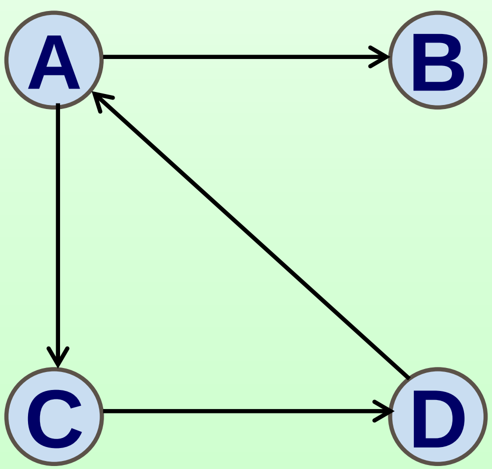
一、图的邻

一维数组：用于

二维数组：用于存储图中顶点之间关联关系—邻接矩阵

定义:矩阵的元素为

$$A[i,j]=\begin{cases}1 & 若 <v_i,v_j> 或 (v_i,v_j)\in VR \\ 0 & 反之\end{cases}$$



有向图

$$\begin{array}{c c} & \begin{array}{cccc} A & B & C & D \end{array} \\ \begin{array}{c} A \\ B \\ C \\ D \end{array} & \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

非对称矩阵

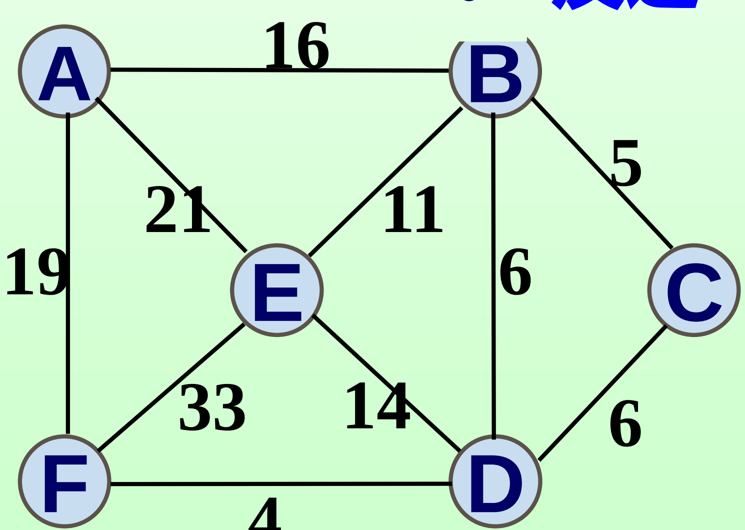
一、图的邻接矩阵存储表示

一维数组：用于存储顶点信息。

二维数组：用于存储图中顶点之间关联关系—邻接矩阵

定义:矩阵的元素为

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



无向网

$$\begin{array}{c|cccccc} & A & B & C & D & E & F \\ \hline A & \infty & 16 & \infty & \infty & 21 & 19 \\ B & 16 & \infty & 5 & 6 & 11 & \infty \\ C & \infty & 5 & \infty & 6 & \infty & \infty \\ D & \infty & 6 & 6 & \infty & 14 & 4 \\ E & 21 & 11 & \infty & 14 & \infty & 33 \\ F & 19 & \infty & \infty & 4 & 33 & \infty \end{array}$$

对称矩阵

一、图的邻接矩阵存储表示

特点：

Ⅰ.存储空间
- 无向图：n(n+1)/2
- 有向图（网）：n^2

Ⅱ.便于运算
- 无向图：$TD(v_i) = \sum_{j=1}^{n} A[i,j]$
- 有向图（网）：
  - $OD(v_i) = \sum_{j=1}^{n} A[i,j]$
  - $ID(v_i) = \sum_{j=1}^{n} A[j,i]$

一、图的数组(邻接矩阵)存储表示

```
# define  INFINITY   INT_MAX            //最大值  ∞
# define  MAX_VERTEX_NUM    20           //最大顶点个数
typedef  enum { DG,DN,UDG,UDN}  GraphKind;
typedef  struct  ArcCell{
        VRType      adj;                 //弧是否相通，或权值
        InfoType    *info;               //弧的相关信息
}ArcCell,
AdjMatrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];

typedef  struct {
        VertexType    vexs[MAX_VERTEX_NUM]; //顶点向量
        AdjMatrix     arcs;                 //邻接矩阵
        int     vexnum, arcnum;          //图的当前顶点和弧数
        GraphKind     kind;              //图的种类标志
}Mgraph;
```

算法7.1 在邻接矩阵的存储结构上创建图

```
Status CreateGraph( MGraph &G )
{   scanf(&G.kind); // 自定义输入函数，读入一个随机值
    switch (G.kind)
   {
    case  DG: return CreateDG(G);   // 构造有向图G
    case  DN: return CreateDN(G);   // 构造有向网G
    case UDG: return CreateUDG(G);  // 构造无向图G
    case UDN: return CreateUDN(G);  // 构造无向网G
    default : return ERROR;
    }
} // CreateGraph
```

算法7.2 采用数组（邻接矩阵）表示法，构造无向网G

```
Status CreateUDN(MGraph &G)
{ scanf("%d,%d,%d",&G.vexnum, &G.arcnum, &IncInfo);
for (i=0; i<G.vexnum; i++ )
   scanf("%c",&G.vexs[i]); // 构造顶点向量
for (i=0; i<G.vexnum; ++i )  // 初始化邻接矩阵
for (j=0; j<G.vexnum; ++j )
        G.arcs[i][j].adj ={ INFINITY,NULL};

for (k=0; k<G.arcnum; ++k )  {  // 构造邻接矩阵
scanf("%c%c%d", &v1 ,&v2, &w);
i = LocateVex(G, v1);  j = LocateVex(G, v2); // v1和v2在G中位置
G.arcs[i][j].adj = w;
if (IncInfo) scanf(G.arcs[i][j].info); // 输入弧的相关信息
G.arcs[j][i].adj= G.arcs[i][j].adj; // 置<v1,v2>的对称弧<v2,v1>
}
return OK;
} // CreateUDN
```

二、图的邻接表表示法（链式存储法）

表头结点： 所有表头结点以顺序结构存储。

边表： 对图中每个顶点建立一个单链表，第i个单链表中的结点表示依附于顶点v_i 的边。

Ⅰ.表头结点

vexdata	firstarc

Ⅱ.表结点

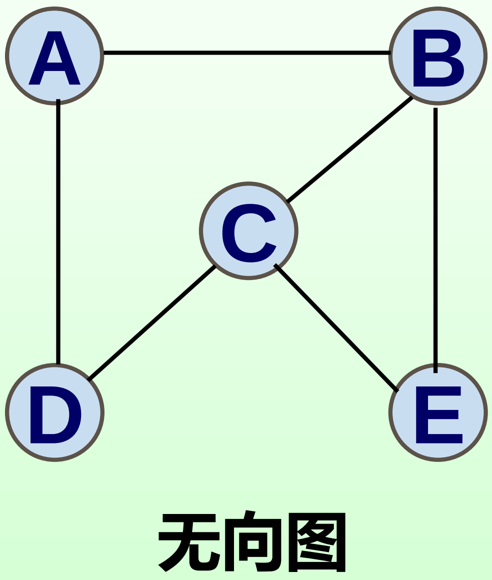
图

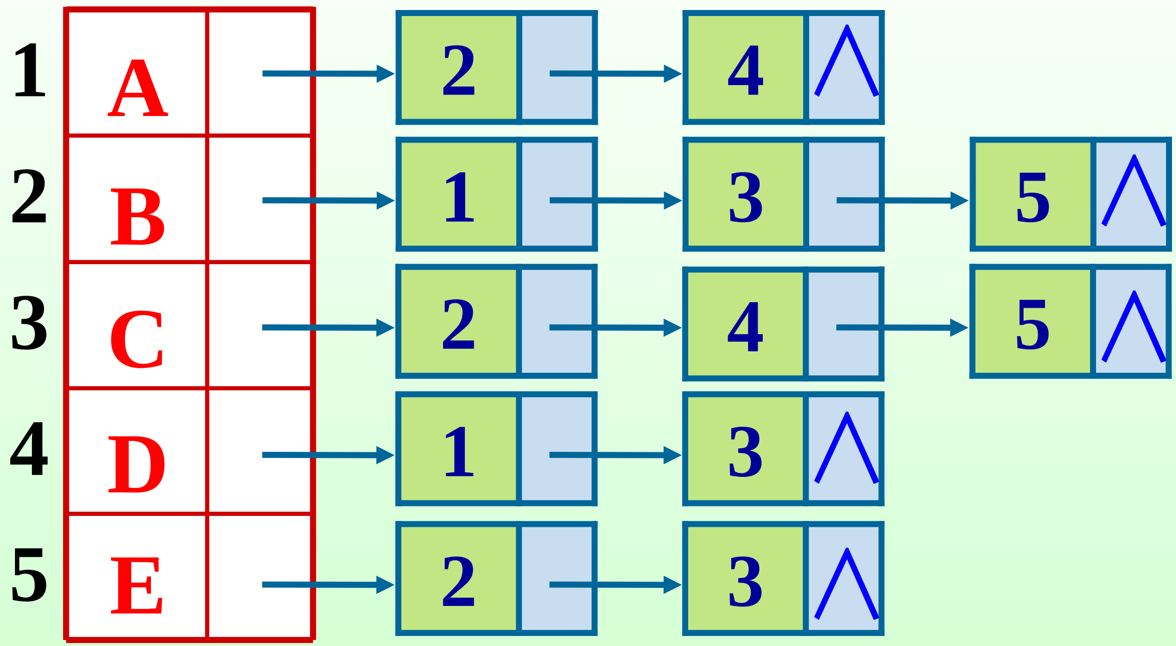
adjvex	nextarc

网

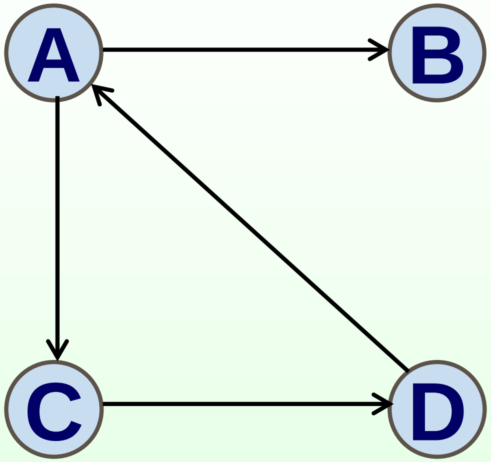
adjvex	info	nextarc

二、图的邻接表表示法（链式存储法）

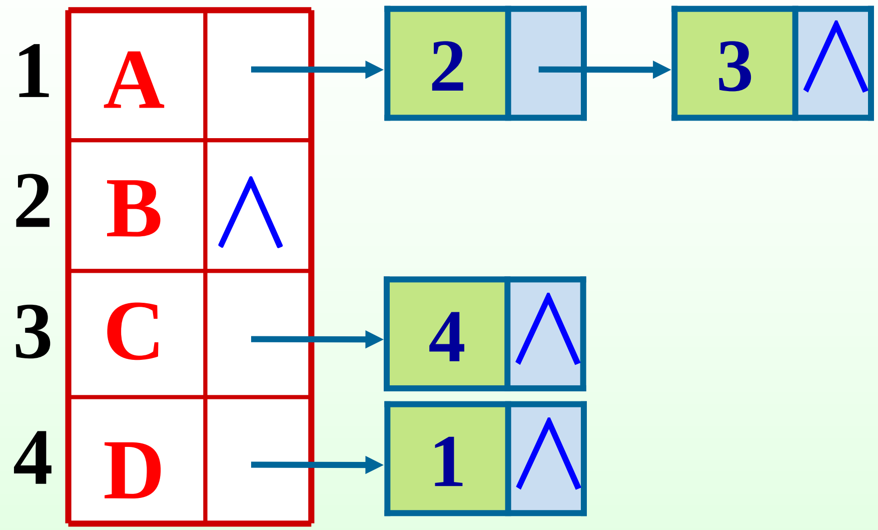




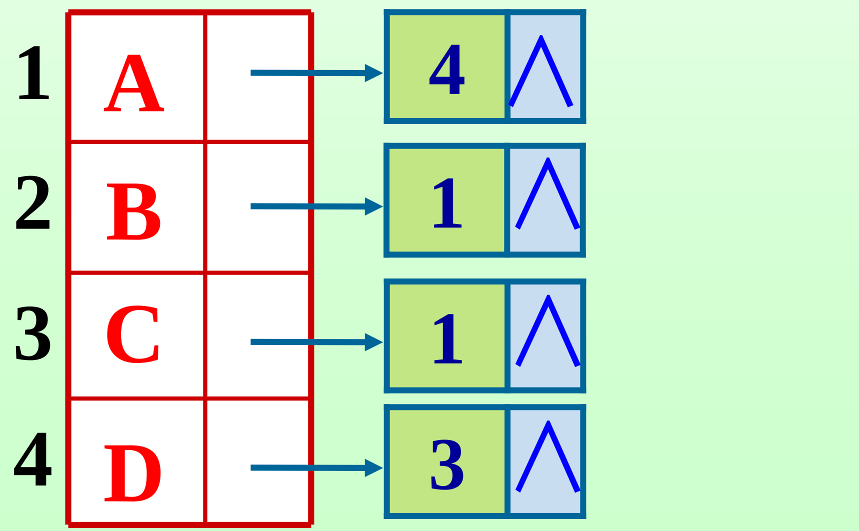
二、图的邻接表表示法（链式存储法）



有向图



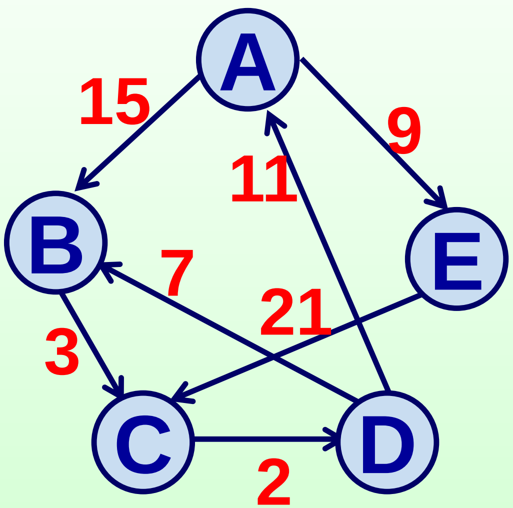
邻接表



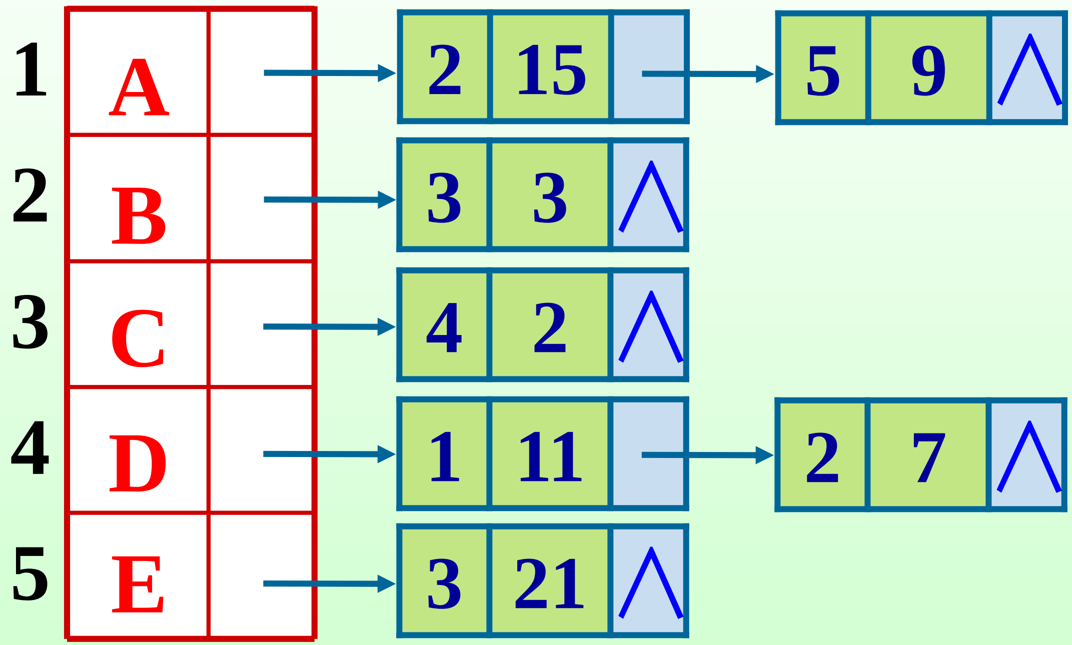
逆邻接表

可见，在有向图的邻接表中不易找到以该顶点为弧头的弧。

二、图的邻接表表示法（链式存储法）



有向网



二、图的邻接表表示法（链式存储法）

特点：

Ⅰ.存储空间
- 无向图：n+2e
- 有向图（网）：n+e

Ⅱ.便于运算

- 无向图：TD（v_i）=第i个单链表上结点的个数
- 有向图（网）：
  - OD（v_i）=第i个单链表上结点的个数
  - ID（v_i）扫描整个邻接表 → 逆邻接表

邻接表存储表示

```
#define  MAX_VERTEX_NUM     20

typedef  struct  ArcNode
{       int  adjvex;
        struct ArcNode    *nextarc;
        InfoType   *info;
}ArcNode;             //边表结点
```

邻接表存储表示

```
typedef  struct   VNode
 {  VertexType  data;
    ArcNode *firstarc;
  }Vnode,  AdjList[MAX_VERTEX_NUM];  //表头结点

typedef   struct
{     AdjList  vertices;
      int   vexnum , arcnum;     //顶点数和弧数
      int   kind;                //图的种类标志
}ALGraph; //基于邻接表的图
```

建立无向图的邻接表

```
CreatAdjList(ALGraph ga ) //顶点数目n，边的数目e
 { int i,j,k; ArcNode *s;
    for (i=0;  i<n; i++)
     {  ga. vertices [i]. data =getchar( );
        ga. vertices [i]. firstarc =NULL; }
    for (k=0; k<e; k++)
     {  scanf(“%d%d”,&i,&j);
         s=(ArcNode *)malloc(sizeof(ArcNode)); //采用头插入法
         s->adjvex=j;   s-> nextarc = ga. vertices [i]. firstarc;
         ga. vertices [i]. firstarc =s;

         s=(ArcNode *)malloc(sizeof(ArcNode));
         s->adjvex=i;   s-> nextarc = ga. vertices [j]. firstarc;
         ga. vertices [j]. firstarc =s;
      }
   }
```

图的两种存储结构比较：邻接矩阵与邻接表

	邻接矩阵	邻接表
存储表示	唯一	不唯一
适宜存储	稠密图	稀疏图
无向图顶点的度	第i行：$TD(V_i)$ 或 第i列：$TD(V_i)$	第i个单链表上结点的个数
有向图顶点的度	第i行：$OD(V_i)$， 第i列：$ID(V_i)$， $TD(V_i)= ID(V_i)+ OD(V_i)$	第i个单链表上结点的个数： $OD(V_i)$ 求ID(Vi)则需遍历整个邻接表 逆邻接表

三、十字链表

将有向图的邻接表和逆邻接表结合在一起，就得到了有向图的另一种链式存储结构——十字链表。

头结点

vexinfo	firstin	firstout

表结点

tailvex	headvex	arcinfo	hnext	tnext

vexinfo ： 顶点的信息

firstin ： 第一条关联入弧结点

firstout ： 第一条关联出弧结点

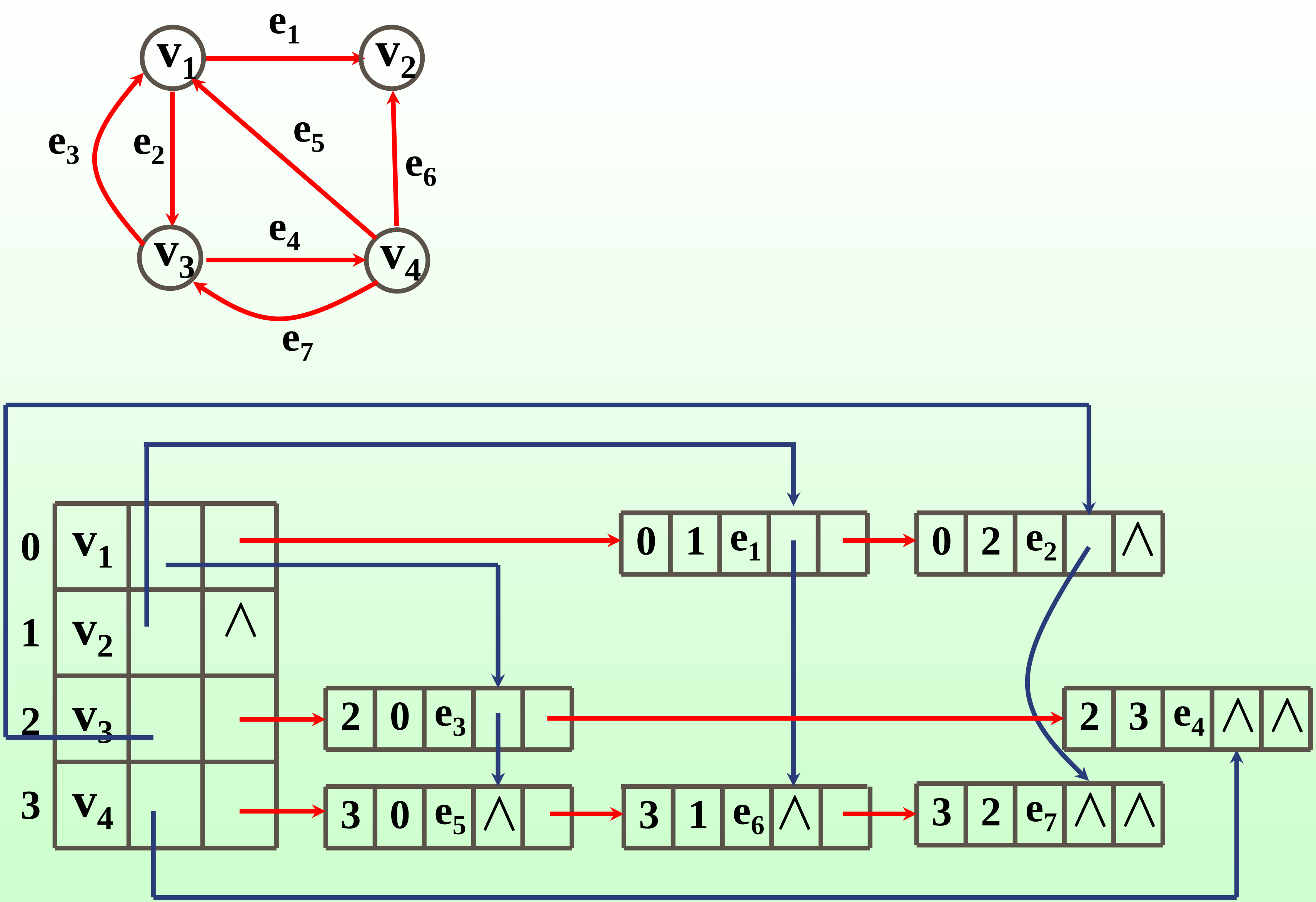
tailvex ： 弧尾顶点位置

headvex ： 弧头顶点位置

arcinfo ： 弧的信息

tnext ： 弧尾相同的下一条弧

hnext ： 弧头相同的下一条弧


e1
v1
v2
e3
e2
e5
e6
e4
v3
v4
e7
0
v1
0
1
e1
0
2
e2
∧
1
v2
∧
2
v3
2
0
e3
2
3
e4
∧
∧
3
v4
3
0
e5
∧
3
1
e6
∧
3
2
e7
∧
∧

四、邻接多重表

邻接表是无向图的一种很有效的存储结构，在邻接表中容易求得顶点和边的各种信息；

但在邻接表中，每一条边都有两个结点表示，因此在某些对边进行的操作(例如对搜索过的边做标记)中就需要对每一条边处理两遍；

故引入邻接多重表实现无向图的存储结构。

邻接多重表的结构与十字链表相似

头结点

vexinfo	firstedge

表结点

mark	ivex	inext	jvex	jnext	info

vexinfo ： 顶点的信息

firstedge ： 第一条关联边结点

mark ： 标志域，是否遍历过

ivex ： 边的第一个顶点位置

inext ： 顶点 i 的下一条关联边

jvex ： 边的另一个顶点位置

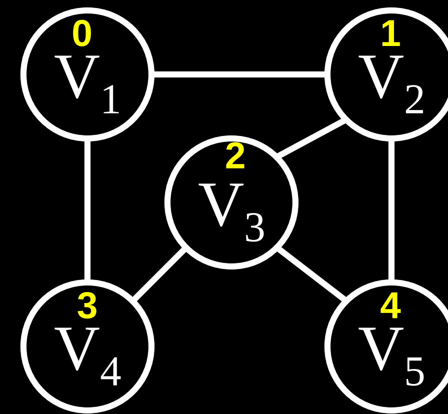
jnext ： 顶点 j 的下一条关联边

info ： 边的信息

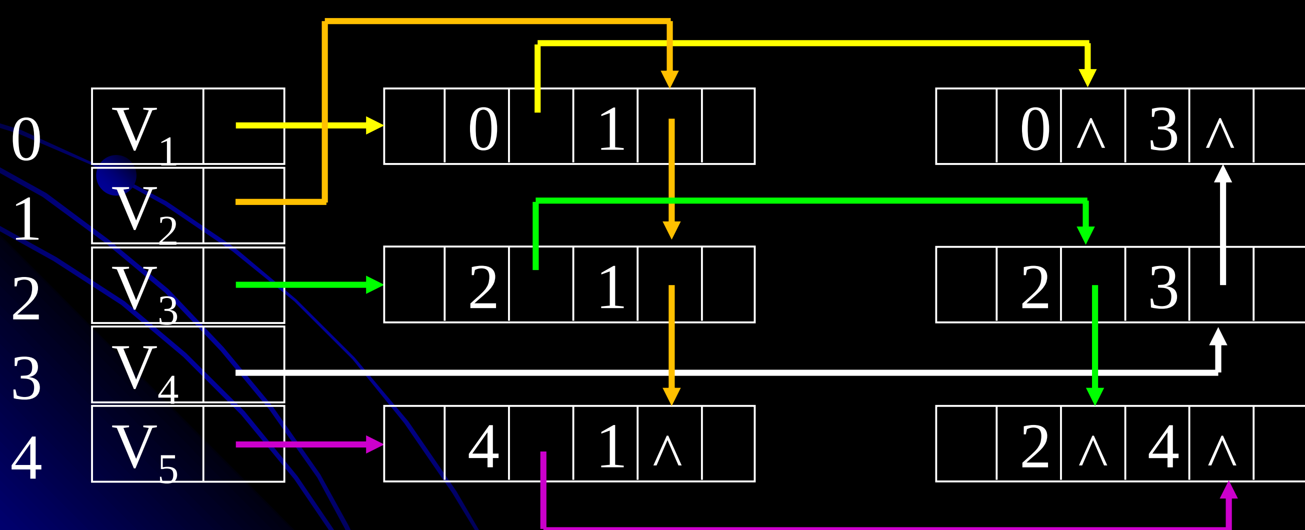
End of 7.2

头结点	vexinfo	firstedge

表结点	mark	ivex	inext	jvex	jnext	info



Undirected graph G_2



7.3
图的遍历

基本概念

遍历： 从图中某个顶点出发遍历图，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。

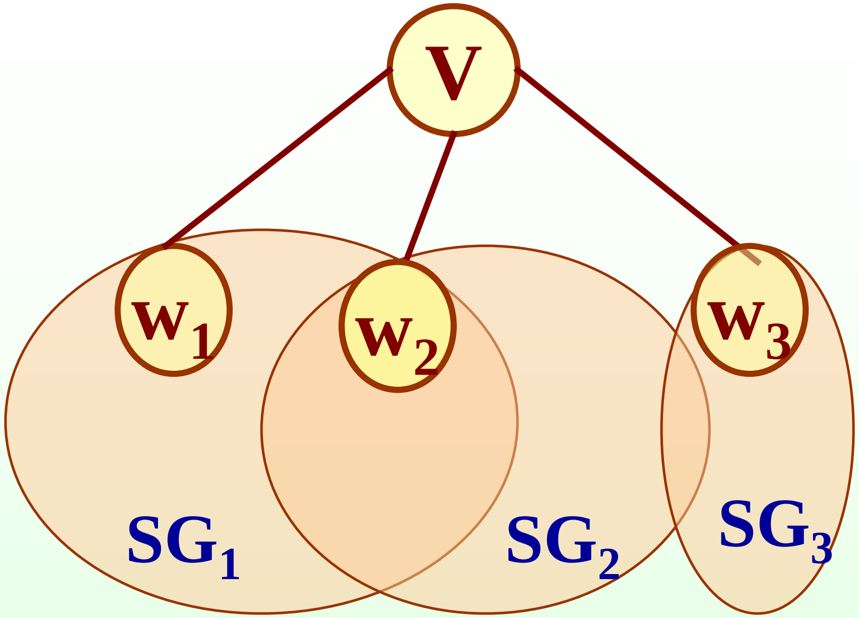
深度优先搜索 Depth First Search

广度优先搜索 Breadth First Search

一.深度优先搜索DFS—基本思想

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。

类似于树的先根次序遍历

访问顶点 V ：

for (W_1、W_2、W_3)

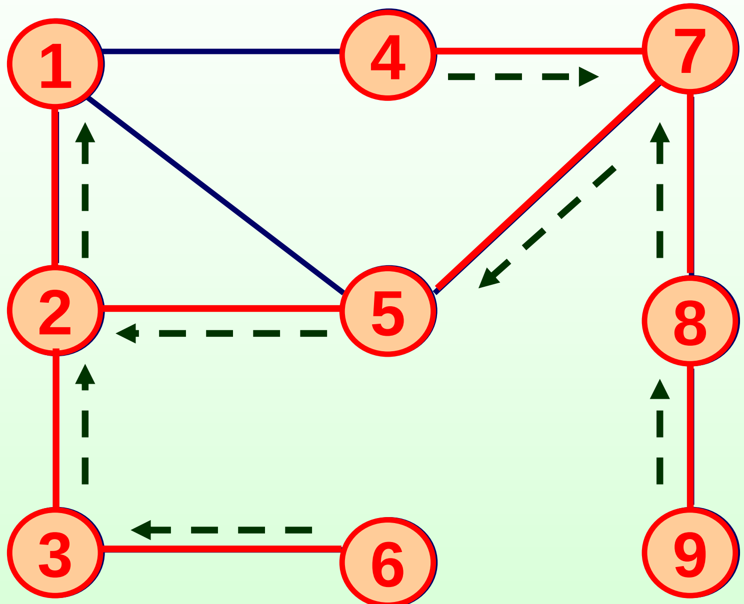
若该邻接点W未被访问,

则从它出发进行深度优先搜索遍历。

一.深度优先搜索DFS—基本思想

深度优先搜索 1 2 3 6 5 7 4 8 9

例： 回到1结束！



深度优先搜索树

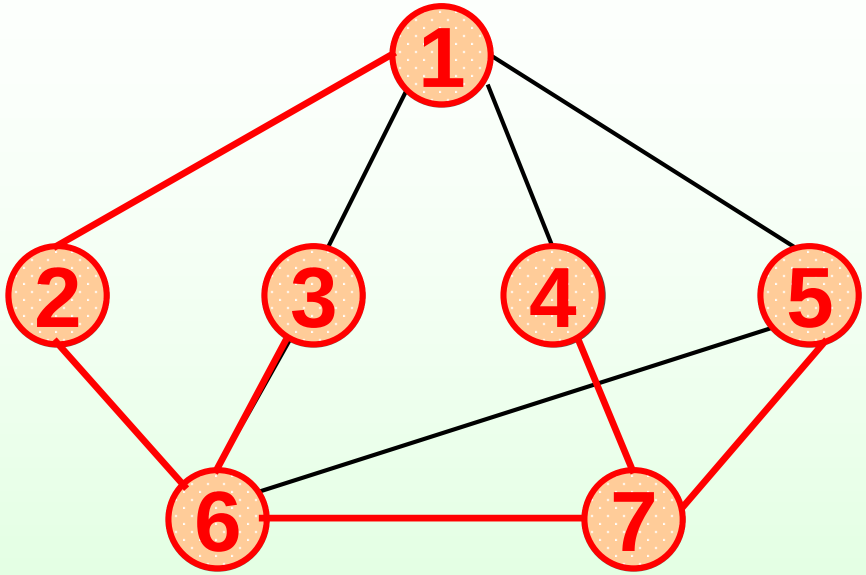
如何判别V的邻接点是否被访问?

解决的办法是：

每个顶点设置一个标志用于检查是否被访问过，

即设置数组visited[n]，visited[i]= $\begin{cases} 0 & \text{未访问} \\ 1 & \text{已访问} \end{cases}$

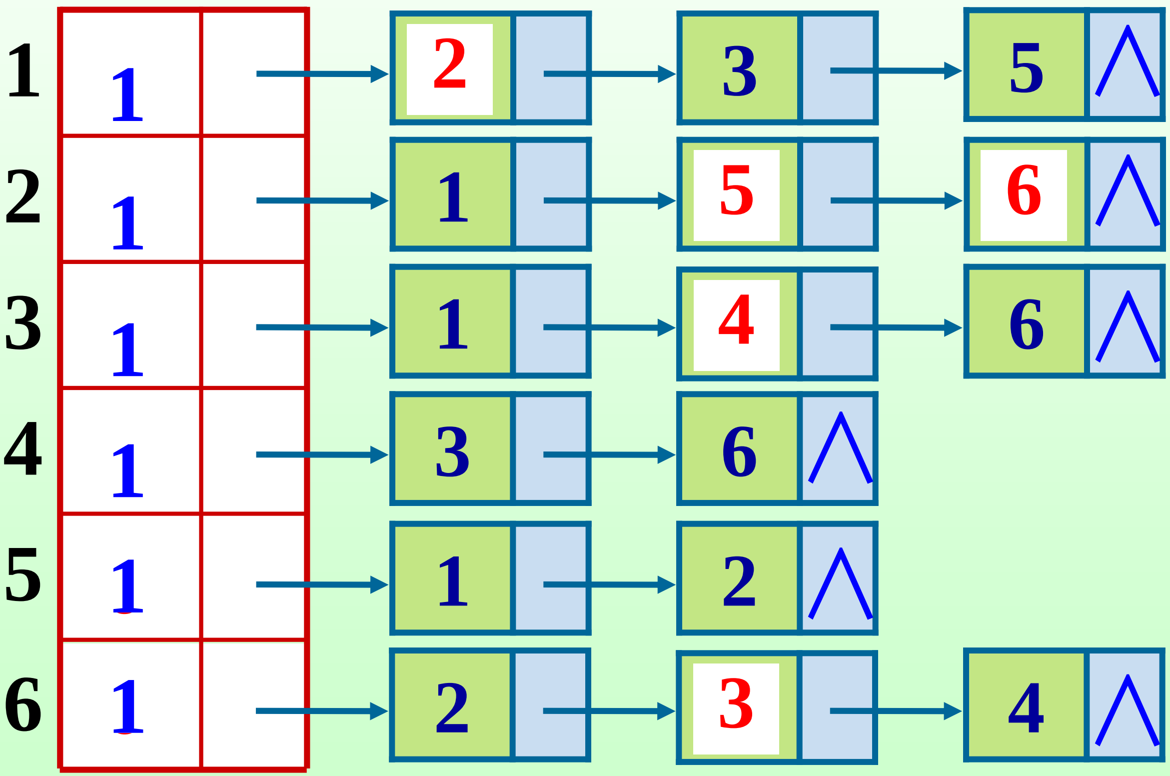
一.深度优先搜索DFS—基本思想

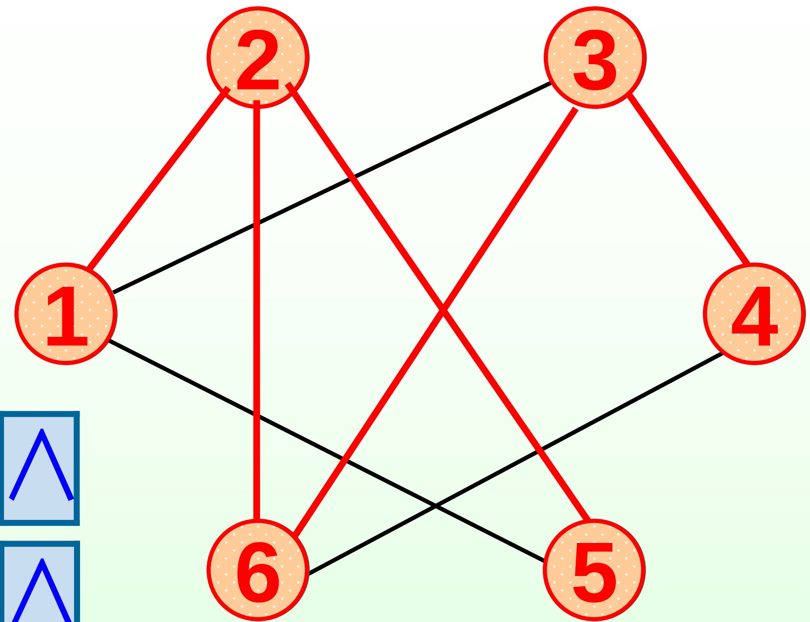


	1	2	3	4	5	6	7
访问标志:	T	T	T	T	T	T	T
访问次序:	1	2	6	3	7	4	5

DFS—邻接表存储

从1出发,深度优先遍历顺序为: 1 2 5 6 3 4

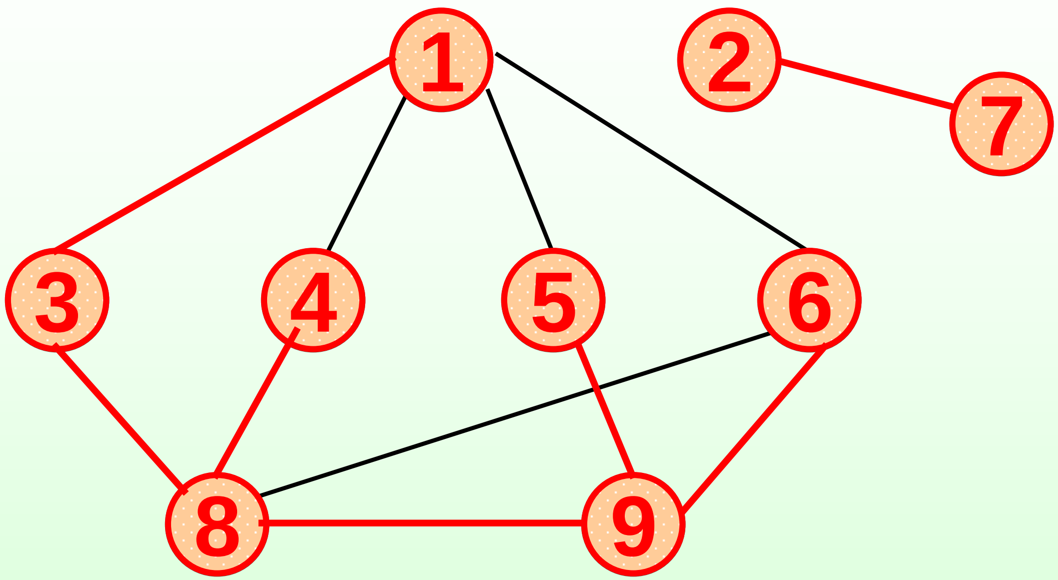




非连通图的深度优先搜索遍历

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。

非连通图的深度优先搜索遍历



	1	2	3	4	5	6	7	8	9
访问标志:	T	T	T	T	T	T	T	T	T
访问次序:	1	3	8	4	9	5	6	2	7

深度优先搜索遍历算法实现

由算法思想知，这是一个递归过程。因此，先设计一个从某个顶点(编号)为v0开始深度优先搜索的函数，便于调用。

在遍历整个图时，可以对图中的每一个未访问的顶点执行所定义的函数。

```
typedef enum {FALSE , TRUE} BOOLEAN ;
BOOLEAN Visited[MAX_VEX] ;
```

深度优先搜索遍历算法实现

```
void DFS(ALGraph *G , int v)

{   LinkNode *p ;
    Visited[v]=TRUE ;
    Visit[v] ; /* 置访问标志，访问顶点v */
    p=G->AdjList[v].firstarc; /* 链表的第一个结点 */
    while (p!=NULL)
      { if (!Visited[p->adjvex]) DFS(G, p->adjvex) ;
      /* 从v的未访问过的邻接顶点出发深度优先搜索 */
      p=p->nextarc ;
      }
}
```

深度优先搜索遍历算法实现

```
void DFS_traverse (ALGraph *G)

{   int v ;
    for (v=0 ; v<G->vexnum ; v++)
    Visited[v]=FALSE ; /* 访问标志初始化 */



    for (v=0 ; v<G->vexnum ; v++)
    if (!Visited[v]) DFS(G , v);
}
```

深度优先搜索遍历算法 分析

遍历时，对图的每个顶点至多调用一次DFS函数。

其实质就是对每个顶点查找邻接顶点的过程，取决于存储结构。

当图有e条边，其时间复杂度为O(e)，总时间复杂度为O(n+e) 。

二.广度优先搜索BFS

连通图的广度优先搜索遍历

Ⅰ.从图中某个顶点v_0出发，首先访问v_0；

Ⅱ.依次访问v_0各个未被访问的邻接点；

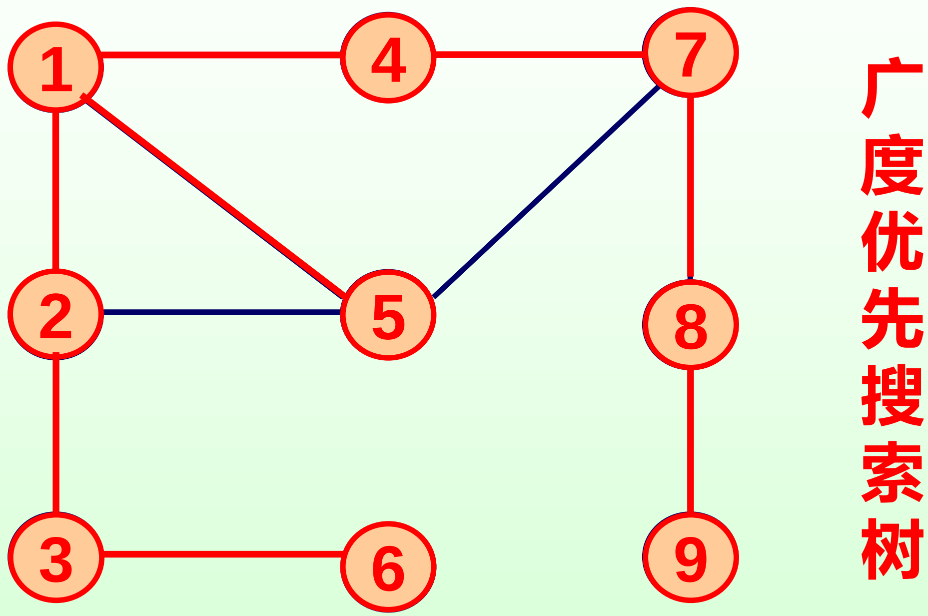
Ⅲ.分别从这些邻接点出发，依次访问它们的各个未被访问的邻接点。访问时应保证：如果v_i在v_k之前被访问，则v_i的所有未被访问的邻接点应在v_k所有未被访问的邻接点之前访问。重复Ⅲ，直到所有端结点均没有未被访问的邻接点为止。

类似于树的层次遍历！

二.广度优先搜索BFS

广度优先搜索　1 2 4 5 3 7 6 8　9

例如：



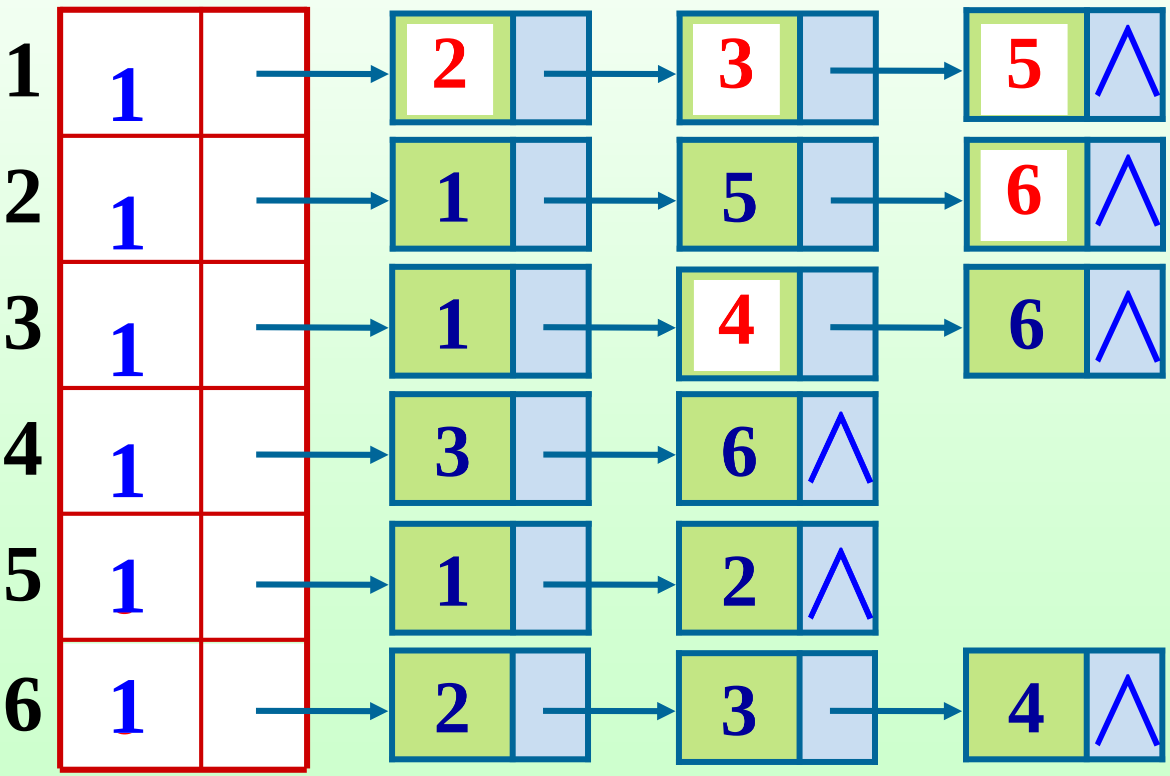
需要辅助队列Q，以便实现访问时应保证的一点，即如果v_i在v_k之前被访问，则v_i的所有未被访问的邻接点应在v_k所有未被访问的邻接点之前访问。

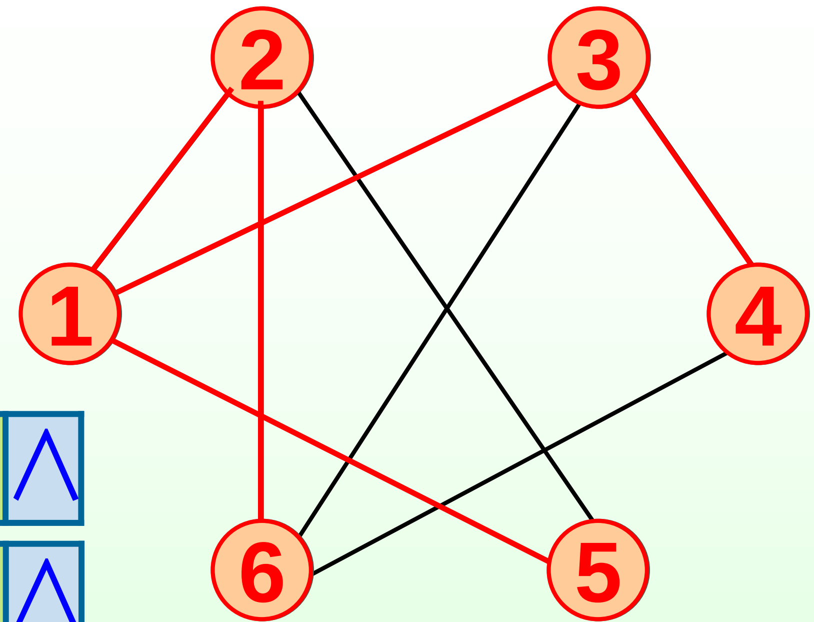
每个顶点设置一个标志用于检查是否被访问过，

即设置数组visited[n]，visited[i]= $\begin{cases} 0 & 未访问 \\ 1 & 已访问 \end{cases}$

BFS—邻接表存储

从1出发,广度优先遍历
顺序为: 1 2 3 5 6 4





BFS结束

广度优先搜索遍历算法实现

- 为了标记图中顶点是否被访问过，同样需要一个访问标记数组；
- 为了依此访问与vi相邻接的各个顶点，需要附加一个队列来保存访问vi的相邻接的顶点。

```
typedef enum {FALSE , TRUE} BOOLEAN ;
BOOLEAN Visited[MAX_VEX] ;
typedef struct Queue
{ int elem[MAX_VEX] ;
int front , rear ;
}Queue ; /* 定义一个队列保存将要访问顶点 */
```

```
void BFS_traverse (ALGraph *G)
{ int k ,v , w ;  LinkNode *p ; Queue *Q ; Q=(Queue *)malloc(sizeof(Queue)) ;
Q->front=Q->rear=0 ;  /* 建立空队列并初始化 */
for (k=0 ; k<G->vexnum ; k++) Visited[k]=FALSE ;  /* 初始化 */
for (k=0 ; k<G->vexnum ; k++)
    { v=G->AdjList[k].data ; /* 单链表的头顶点 */
    if (!Visited[v]) /* v尚未访问 */
        { Q->elem[++Q->rear]=v ; /* v入队 */
            while (Q->front!=Q->rear)
                { w=Q->elem[++Q->front] ;
                    Visited[w]=TRUE ; /* 访问标志 */
                    Visit(w) ; /* 访问队首元素 */
                    p=G->AdjList[w].firstarc ;
                        while (p!=NULL)
                            { if (!Visited[p->adjvex])
                                Q->elem[++Q->rear]=p->adjvex ;
                                p=p->nextarc ;  }  }  }  }    }
```

广度优先搜索遍历算法 分析

广度优先搜索算法遍历图与深度优先搜索算法遍历图的唯一区别是**邻接点搜索次序不同**。因此，广度优先搜索算法遍历图的总时间复杂度仍然为O(n+e) 。

图的遍历可以系统地访问图中的每个顶点。因此，图的遍历算法是图的最基本、最重要的算法，许多有关图的操作都是在遍历基础之上实现的。

End of 7.3