采用邻接矩阵表示法创建无向网

4 5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600

【算法思想】

(1) 输入总顶点数和总边数。
(2) 依次输入点的信息存入顶点表中。
(3) 初始化邻接矩阵，使每个权值初始化为极大值。
(4) 构造邻接矩阵。

【算法描述】

```
4  5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

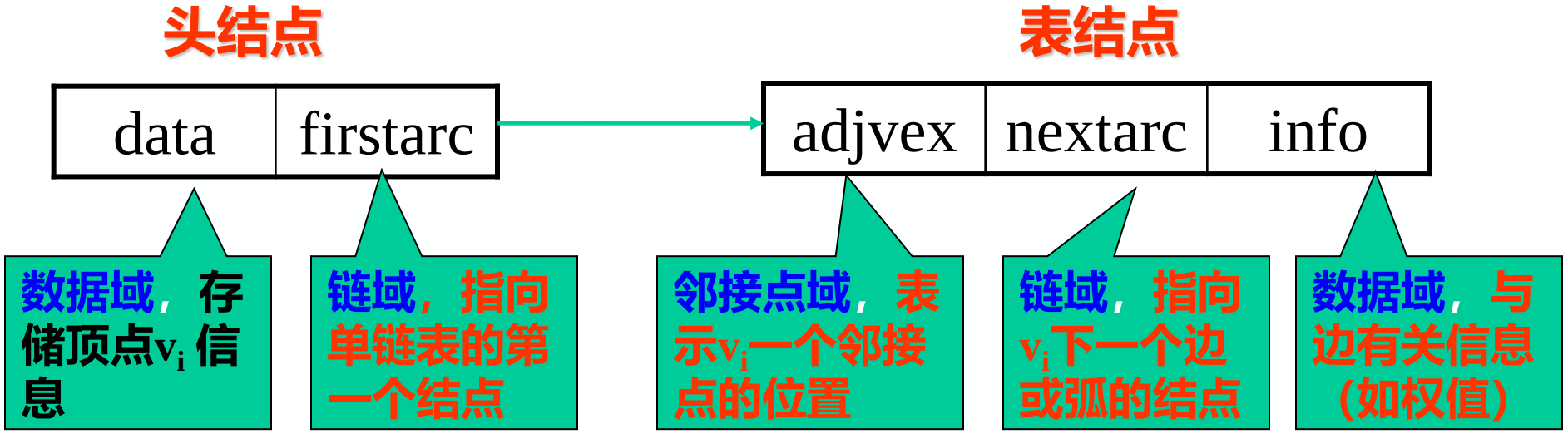
```
Status CreateUDN(AMGraph &G){
  //采用邻接矩阵表示法，创建无向网G
  cin>>G.vexnum>>G.arcnum;  //输入总顶点数，总边数
  for(i = 0; i<G.vexnum; ++i)
    cin>>G.vexs[i];                    //依次输入点的信息
  for(i = 0; i<G.vexnum;++i)           //初始化邻接矩阵，边的权值均置为极大值
    for(j = 0; j<G.vexnum;++j)
      G.arcs[i][j] = MaxInt;
  for(k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

```
4    5
A B C D
A B 500
A C 200
A D 150
B C 400
C D 600
```

```
int LocateVex(MGraph G,VertexType u)
{//存在则返回u在顶点表中的下标;否则返回-1
  int i;
  for(i=0;i<G.vexnum;++i)
    if(u==G.vexs[i])
      return i;
  return -1;
}
```

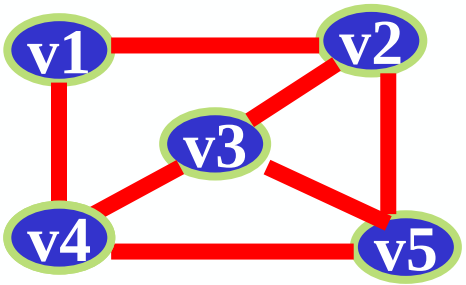
邻接表（链式）表示法

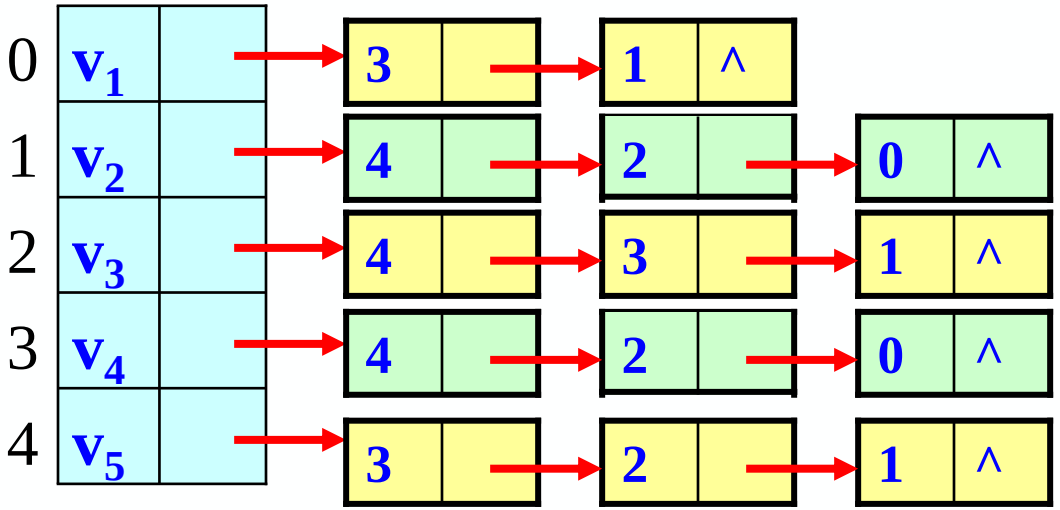
- 对每个顶点v_i建立一个**单链表**，把与v_i有关联的**边的信息链接**起来，每个结点设为3个域；



- 每个单链表有一个**头结点**（设为2个域），存v_i信息；
- 每个单链表的**头结点另外用顺序存储**结构存储。

无向图的邻接表表示



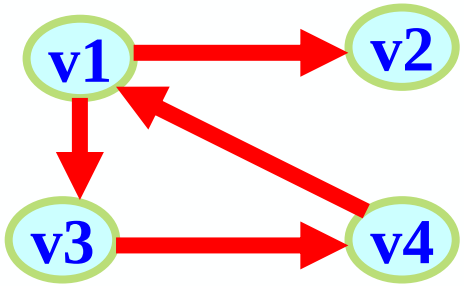


注：邻接表不唯一，因各个边结点的链入顺序是任意的

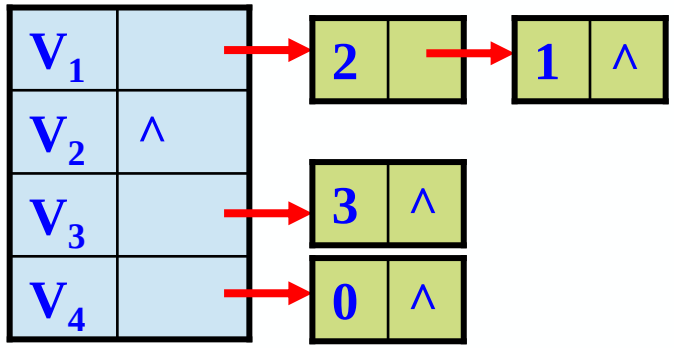
空间效率为O(n+2e)。
若是稀疏图(e<<n^2)，比邻接矩阵表示法O(n^2)省空间。

TD(Vi)=单链表中链接的结点个数

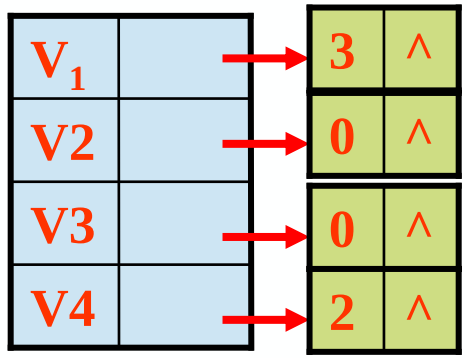
有向图的邻接表表示



邻接表(出边)



逆邻接表(入边)



空间效率为O(n+e)

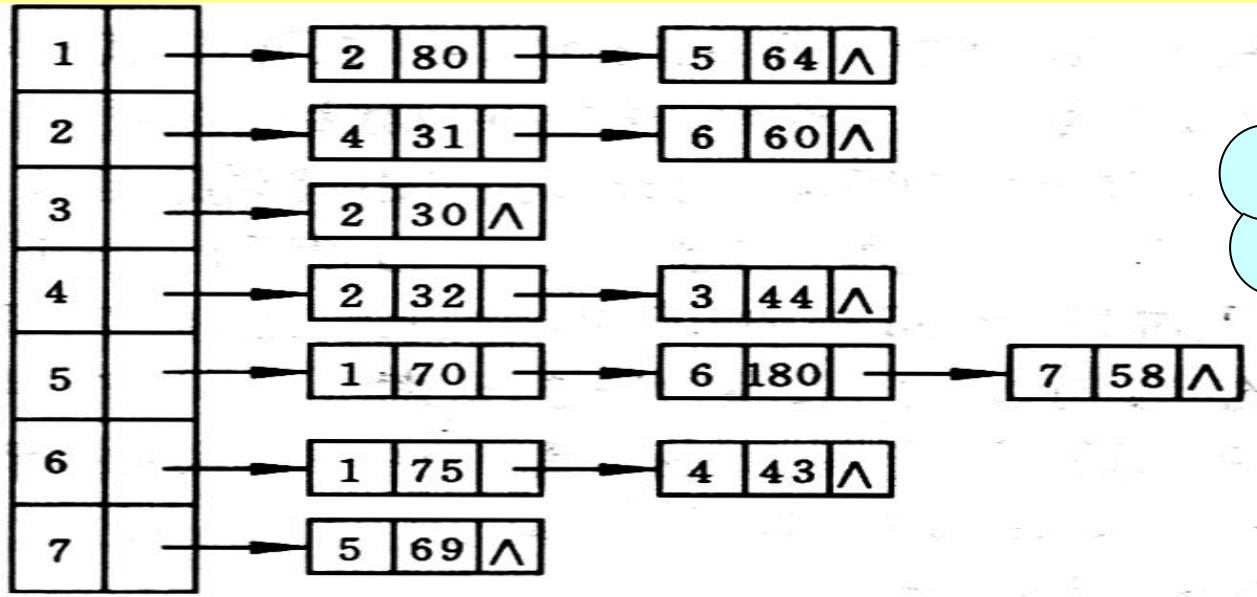
出度
入度
度:

OD(Vi) = 单链出边表中链接的结点数

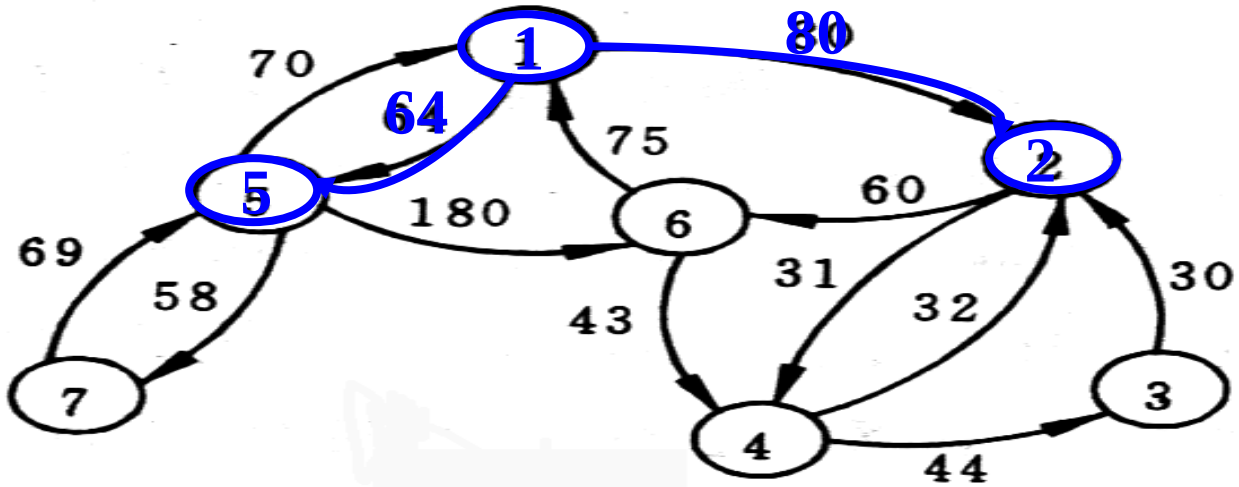
ID(Vi) = 邻接点域为Vi的弧个数

$$TD(Vi) = OD(Vi) + ID(Vi)$$

已知某网的邻接（出边）表，请画出该网络。



当邻接表的存储结构形成后，图便唯一确定！



邻接表的存储表示

```
#define MVNum 100                          //最大顶点数
typedef struct ArcNode{                    //边结点
    int adjvex;                            //该边所指向的顶点的位置
    struct ArcNode * nextarc;              //指向下一条边的指针
    OtherInfo info;                        //和边相关的信息
}ArcNode;
typedef struct VNode{
    VerTexType data;                //顶点信息
    ArcNode * firstarc;             //指向第一条依附该顶点的边的指针
}VNode, AdjList[MVNum];                    //AdjList表示邻接表类型
typedef struct{
    AdjList vertices;                      //邻接表
    int vexnum, arcnum;                    //图的当前顶点数和边数
}ALGraph;
```

采用邻接表表示法创建无向网

【算法思想】

(1) 输入总顶点数和总边数。

(2) 依次输入点的信息存入顶点表中，使每个表头结点的指针域初始化为NULL。

(3) 创建邻接表。

【算法描述】

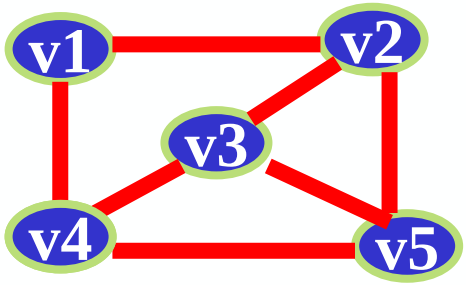
```
Status CreateUDG(ALGraph &G){
    //采用邻接表表示法，创建无向图G
   cin>>G.vexnum>>G.arcnum;            //输入总顶点数，总边数
  for(i = 0; i<G.vexnum; ++i){         //输入各点，构造表头结点表
    cin>> G.vertices[i].data;          //输入顶点值
    G.vertices[i].firstarc=NULL;       //初始化表头结点的指针域为NULL
  }//for
  for(k = 0; k<G.arcnum;++k){                    //输入各边，构造邻接表
    cin>>v1>>v2;                                 //输入一条边依附的两个顶点
    i = LocateVex(G, v1);  j = LocateVex(G, v2);
    p1=new ArcNode;                              //生成一个新的边结点*p1
      p1->adjvex=j;                              //邻接点序号为j
      p1->nextarc= G.vertices[i].firstarc;  G.vertices[i].firstarc=p1;
   //将新结点*p1插入顶点vi的边表头部
   p2=new ArcNode; //生成另一个对称的新的边结点*p2
      p2->adjvex=i;                              //邻接点序号为i
      p2->nextarc= G.vertices[j].firstarc;  G.vertices[j].firstarc=p2;
   //将新结点*p2插入顶点vj的边表头部
  }//for
  return OK;
}//CreateUDG
```

邻接表表示法的特点

优点：空间效率高，容易寻找顶点的邻接点；

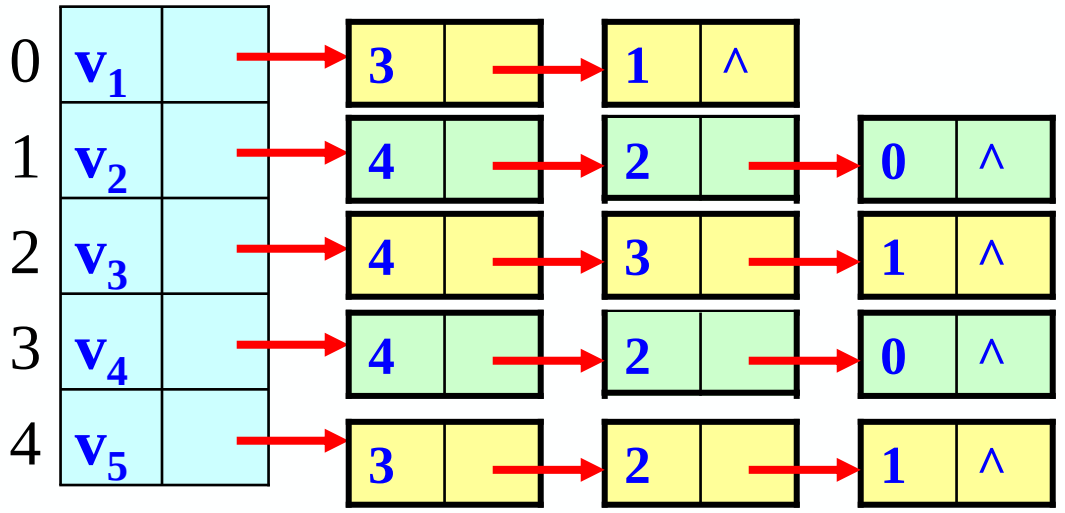
缺点：判断两顶点间是否有边或弧，需搜索两结点对应的单链表，没有邻接矩阵方便。

邻接矩阵与邻接表表示法的关系



（ v1 v2 v3 v4 v5 ）

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



1. 联系：邻接表中每个链表对应于邻接矩阵中的一行，链表中结点个数等于一行中非零元素的个数。

邻接矩阵与邻接表表示法的关系

2. 区别：

① 对于任一确定的无向图，邻接矩阵是**唯一**的（行列号与顶点编号一致），但邻接表**不唯一**（链接次序与顶点编号无关）。

② 邻接矩阵的空间复杂度为$O(n^2)$,而邻接表的空间复杂度为$O(n+e)$。

3. 用途： 邻接矩阵多用于**稠密图**；而邻接表多用于**稀疏图**

十字链表---用于有向图

•结点表中的结点的表示：

data	firstin	firstout

data：结点的数据域，保存结点的数据值。

firstin: 结点的指针域，给出自该结点出发的的第一条边的边结点的地址。

firstout：结点的指针场，给出进入该结点的第一条边的 边结点的地址。

•边结点表中的结点的表示：

info	tailvex	headvex	hlink	tlink

info:边结点的数据域，保存边的权值等。

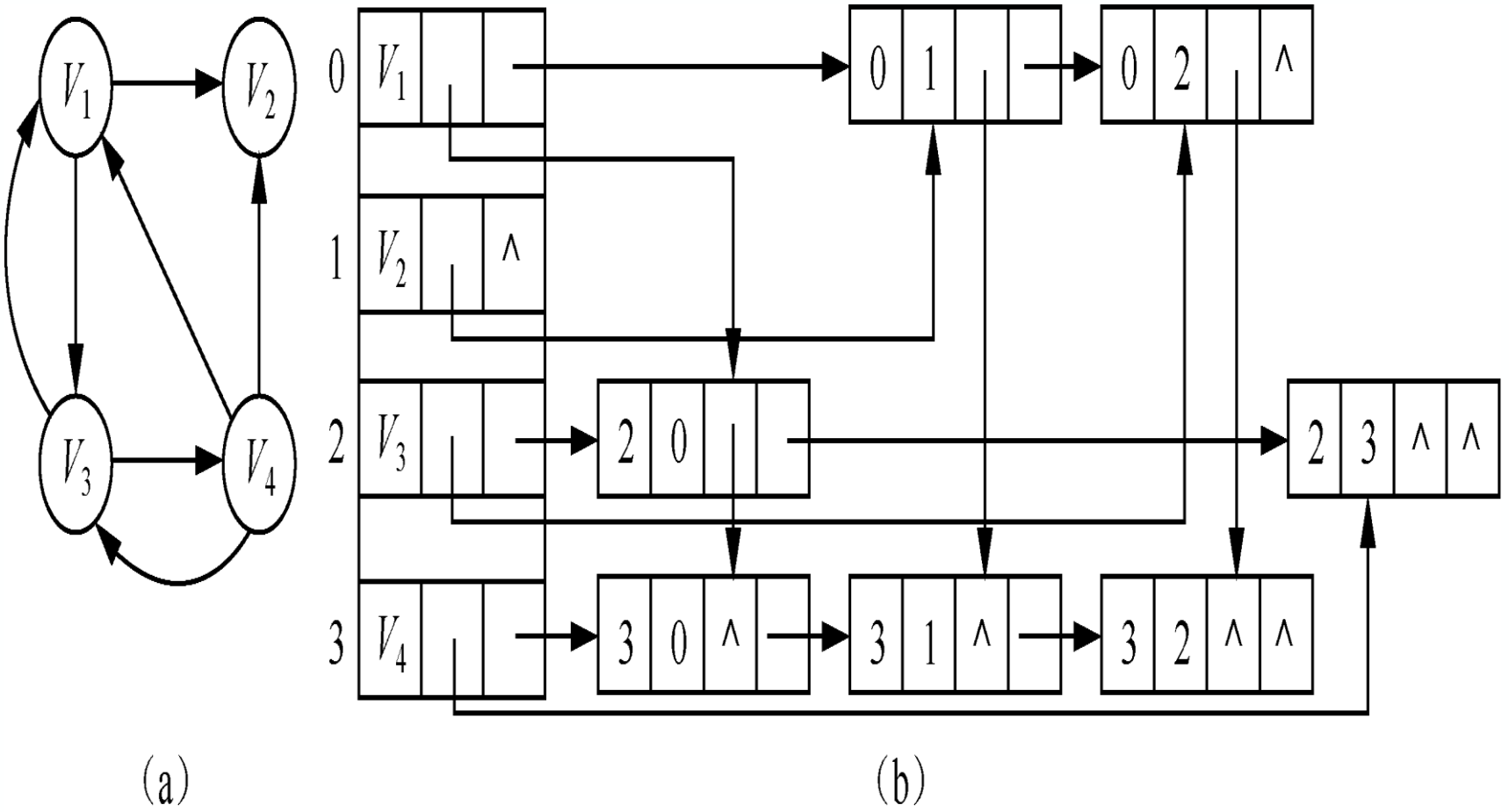
tailvex: 本条边的出发结点的地址。

headvex:本条边的终止结点的地址。

hlink:终止结点相同的边中的下一条边的地址。

tlink:出发结点相同的边中的下一条边的地址。

十字链表



十字链表---用于无向图

•结点表中的结点的表示：

data	firstedge

data:结点的数据域，保存结点的数据值。

firstedge: 结点的指针域，给出自该结点出发的的第一条边的边结点的地址。

•边结点表中的结点的表示：

mark	ivex	ilink	jvex	jlink	info

ivex: 本条边依附的一个结点的地址。

ilink: 依附于该结点（地址由ivex给出）的边中的下一条边的的地址。

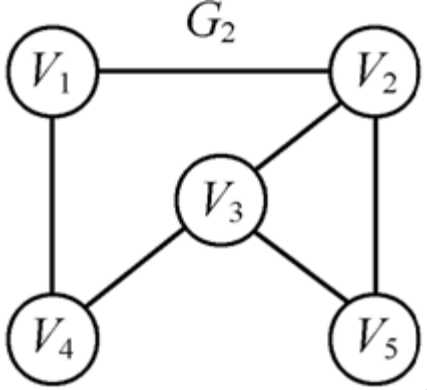
jvex: 本条边依附的另一个结点的地址。

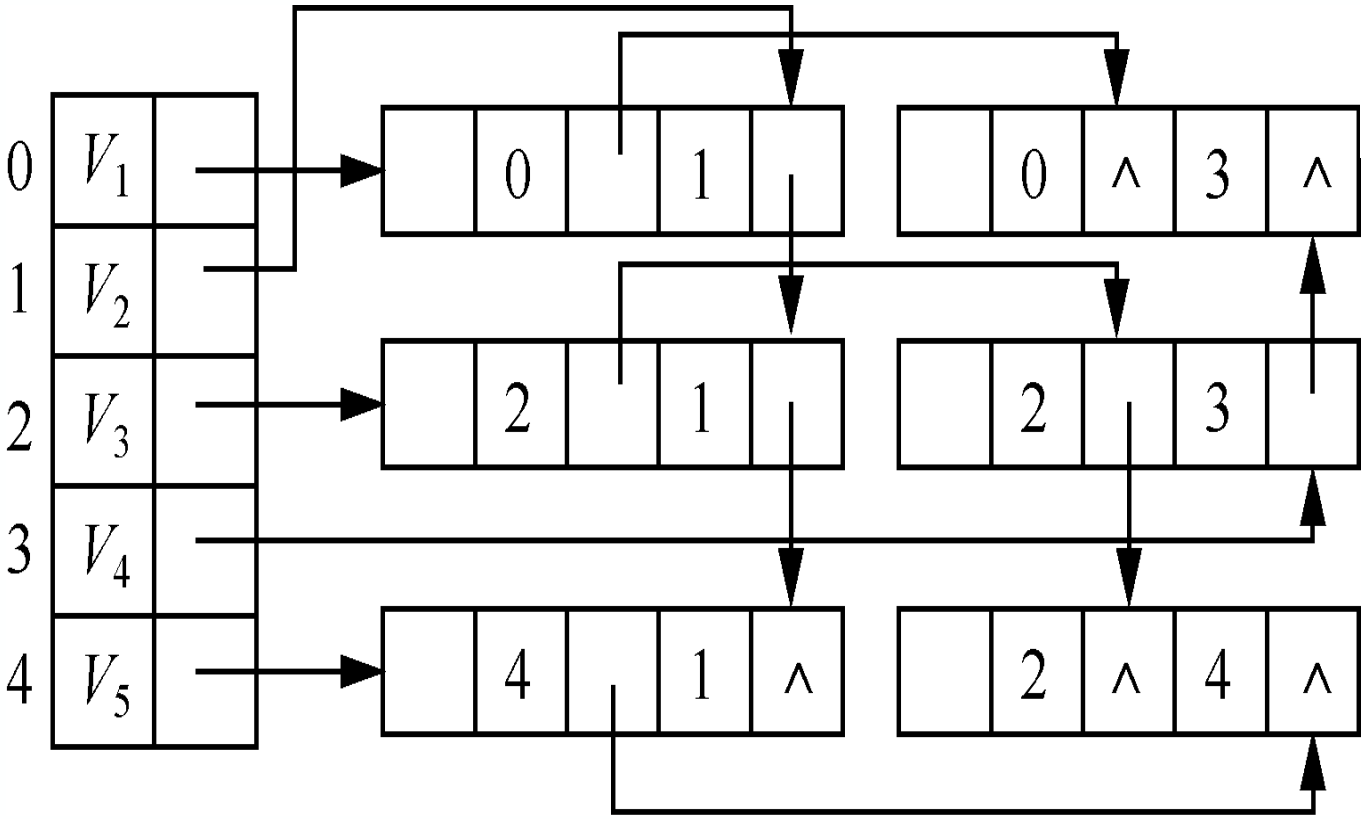
jlink: 依附于该结点（地址由jvex给出）的边中的下一条边的的地址。

info: 边结点的数据域，保存边的权值等。

mark:边结点的标志域，用于标识该条边是否被访问过。

十字链表





6.5 图的遍历

遍历定义：从已给的连通图中某一顶点出发，沿着一些边访遍图中所有的顶点，且使每个顶点仅被访问一次，就叫做图的遍历，它是图的基本运算。

遍历实质：找每个顶点的邻接点的过程。

图的特点：图中可能存在回路，且图的任一顶点都可能与其它顶点相通，在访问完某个顶点之后可能会沿着某些边又回到了曾经访问过的顶点。

人工智能--搜索问题

有3个传教士和3个野人来到河边准备渡河，河岸有一条船，每次最多可坐2个人。问传教士为安全起见，应如何规划摆渡方案，使得在任何时刻，在河两岸以及船上传教士人数不能少于野人人数?

在每一次渡河后，都会有几种渡河方案供选择，究竟哪种方案最有利？ 这就是搜索问题。

人工智能--搜索问题

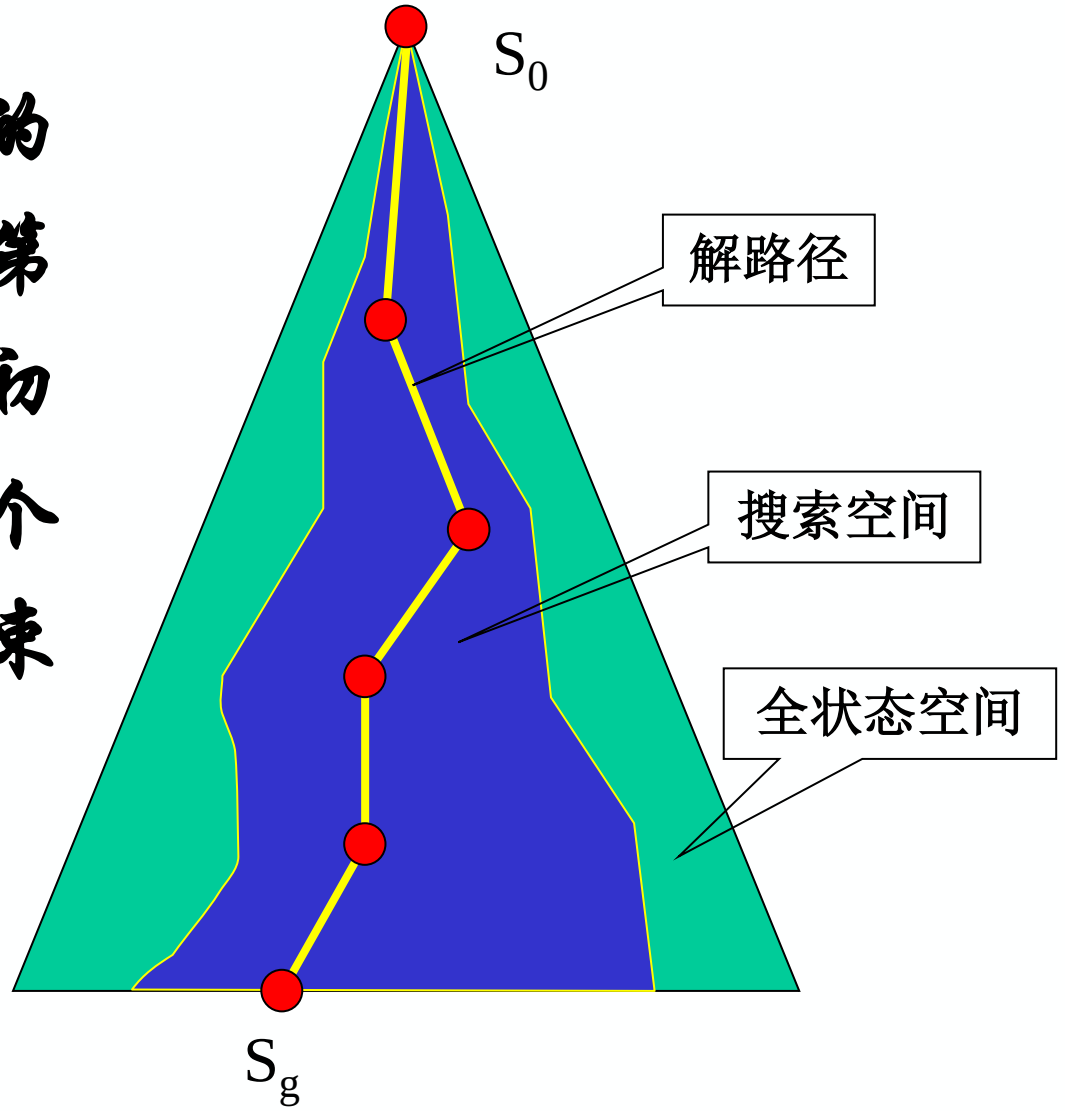
适用情况：难以获得求解所需的全部信息；更没有现成的算法可供求解使用。

概念：依靠经验，利用已有知识，根据问题的实际情况，不断寻找可利用知识，从而构造一条代价最小的推理路线，使问题得以解决的过程称为搜索

对这类问题，一般我们都转换为状态空间的搜索问题。

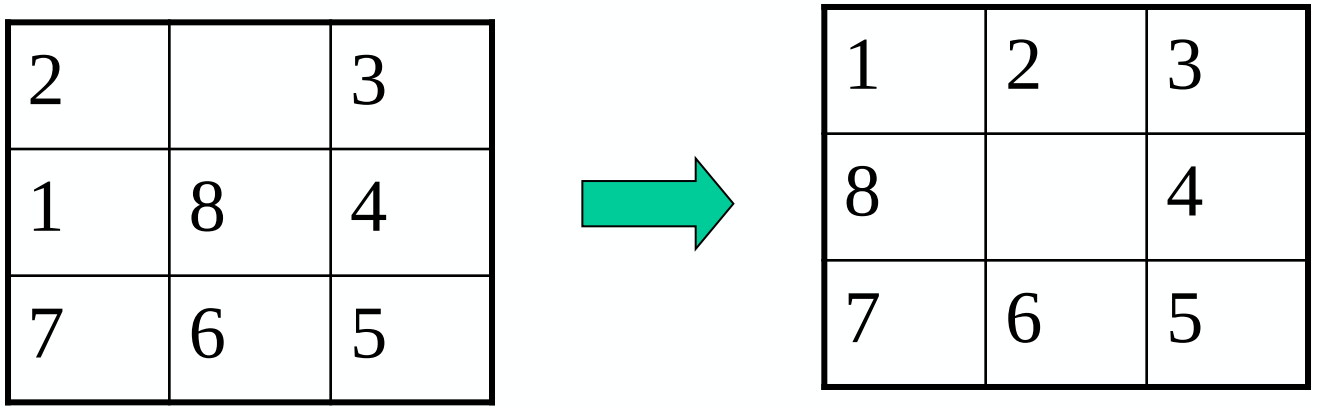
如传教士和野人问题，可用在河左岸的传教士人数、野人人数和船的情况来表示。即，初始时状态为（3，3，1），结束状态为（0，0，0），而中间状态可表示为（2，2，0）、（3，2，1）等等。

这类问题的解，就是一个合法状态的序列，其中序列中第一个状态是问题的初始状态，而最后一个状态则是问题的结束状态。



例: 8数码难题

- 在一个3×3的方框内放有8个编号的小方块，紧邻空位的小方块可以移入到空位上，通过平移小方块可将某一布局变换为另一布局。请给出从初始状态到目标状态移动小方块的操作序列。



搜索引擎的两种基本抓取策略

爬行和抓取

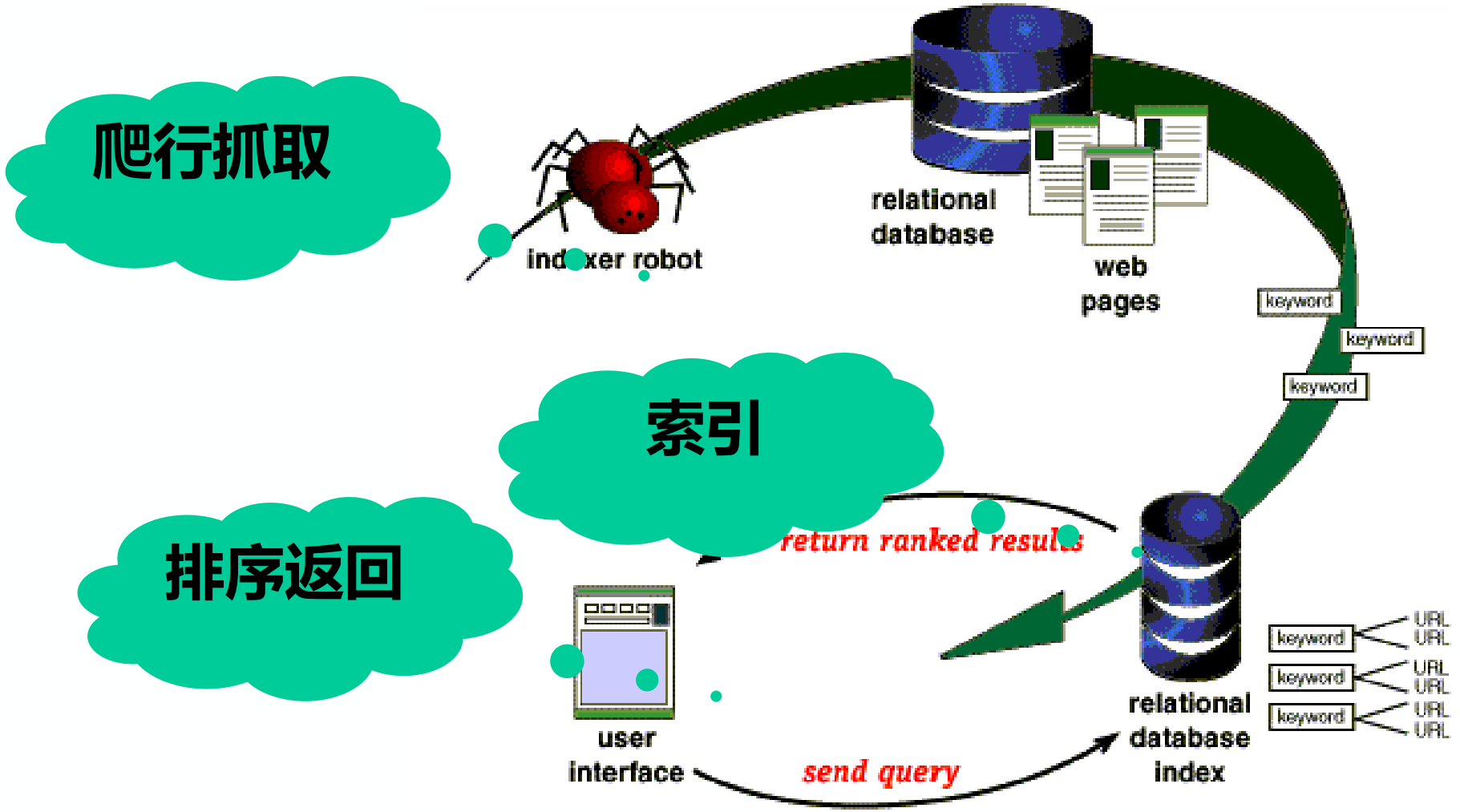
- 搜索引擎蜘蛛通过跟踪链接访问网页，获得页面HTML代码存入数据库

预处理

- 索引程序对抓取来的页面数据进行文字提取、中文分词、索引等处理，以备排名程序调用

排名

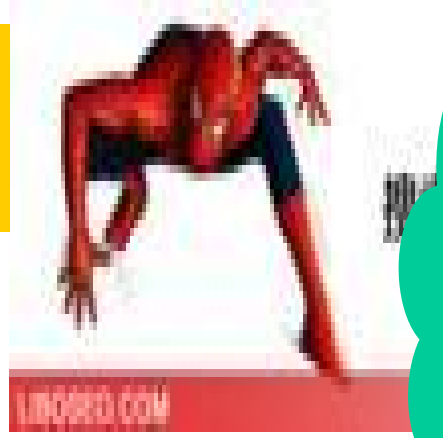
- 用户输入关键词后，排名程序调用索引库数据，计算相关性，然后按一定格式生成搜索结果页面


爬行抓取
索引
排序返回
indexer robot
relational
database
web
pages
keyword
keyword
keyword
return ranked results
user
interface
send query
relational
database
index
keyword
keyword
keyword
URL
URL
URL
URL
URL
URL

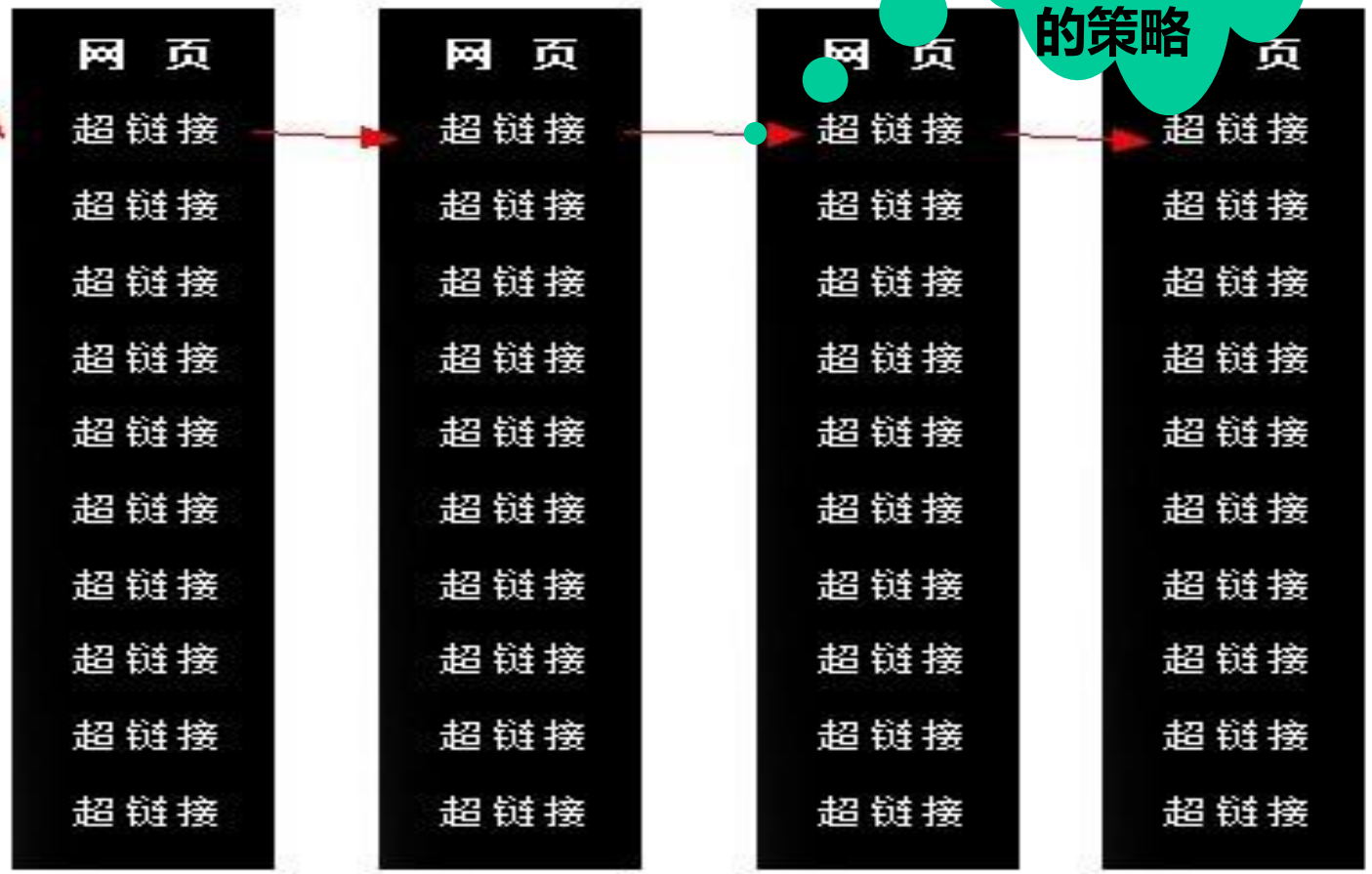
2020年6月1日

搜索引擎的两种基本抓取策略
---深度优先

蜘 蛛



如封建帝位的继承。不能深入的情况下才考虑其他分支的策略



搜索引擎的两种基本抓取策略
---广度优先

类似长幼有序的规则

蜘蛛

网 页 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接

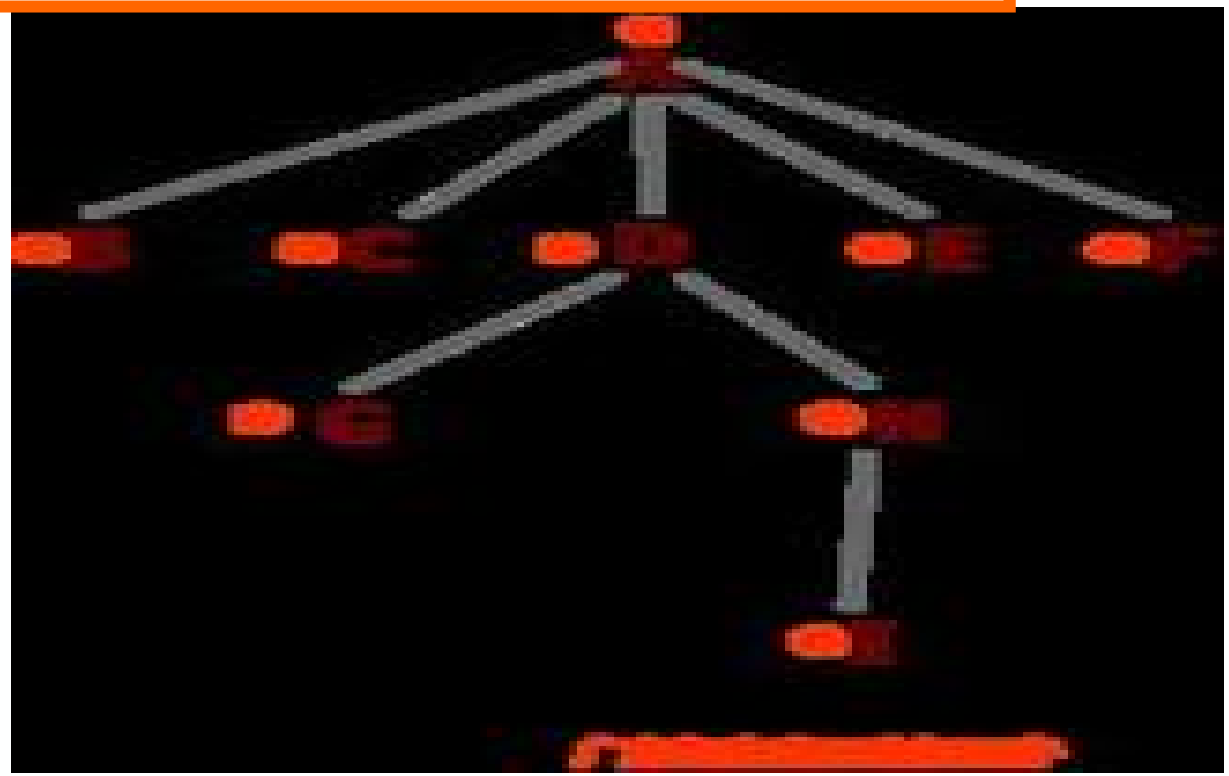
网 页 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接

网 页 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接

网 页 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接 超链接

两种策略结合=先广后深 +权重优先

先把这个页面所有的链接都抓取一次
再根据这些URL的权重来判定
URL的权重高，就采用深度优先，
URL权重低，就采用宽度优先或者不抓取



怎样避免重复访问?

解决思路： 设置辅助数组 *visited* [n]，用来标记每个被访问过的顶点。

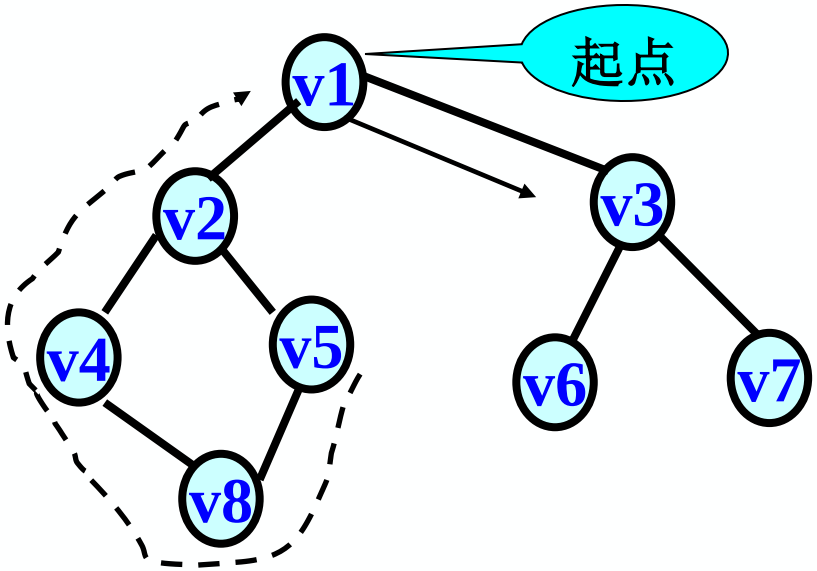
- ✓ 初始状态为0
- ✓ *i*被访问，改 *visited* [*i*]为1，防止被多次访问

图常用的遍历：

- ✓深度优先搜索
- ✓广度优先搜索

深度优先搜索(DFS - Depth_First Search)

基本思想：——仿树的先序遍历过程。



DFS 结果

v1→v2→v4→v8→
v5→v3→v6→v7

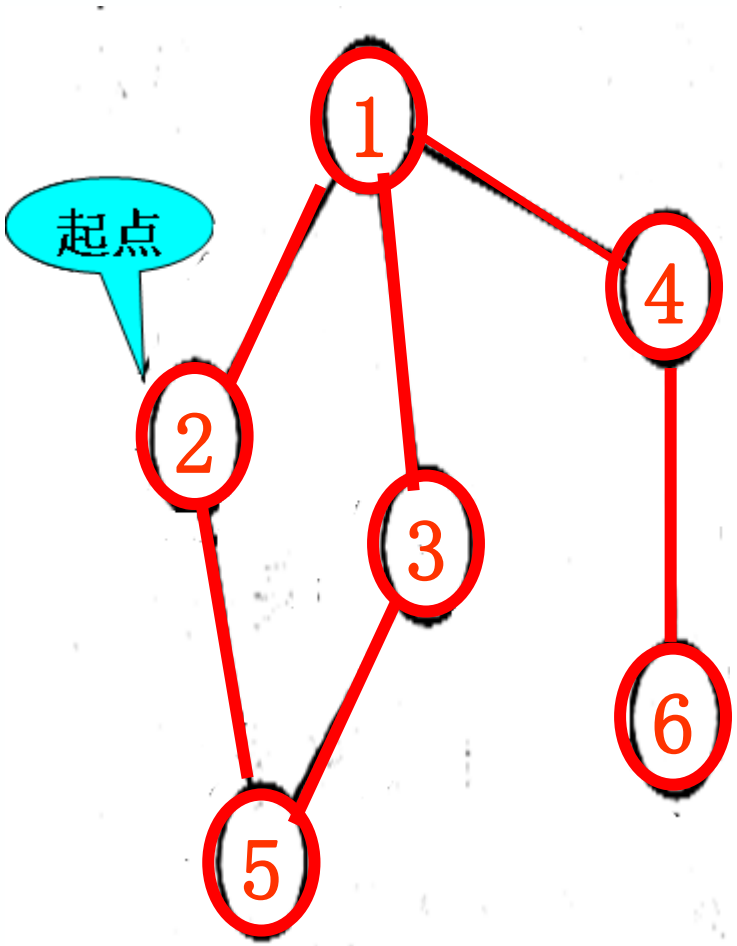
2020年6月1日

深度优先搜索的步骤

简单归纳：

- 访问起始点v；
- 若v的第1个邻接点没访问过，深度遍历此邻接点；
- 若当前邻接点已访问过，再找v的第2个邻接点重新遍历。

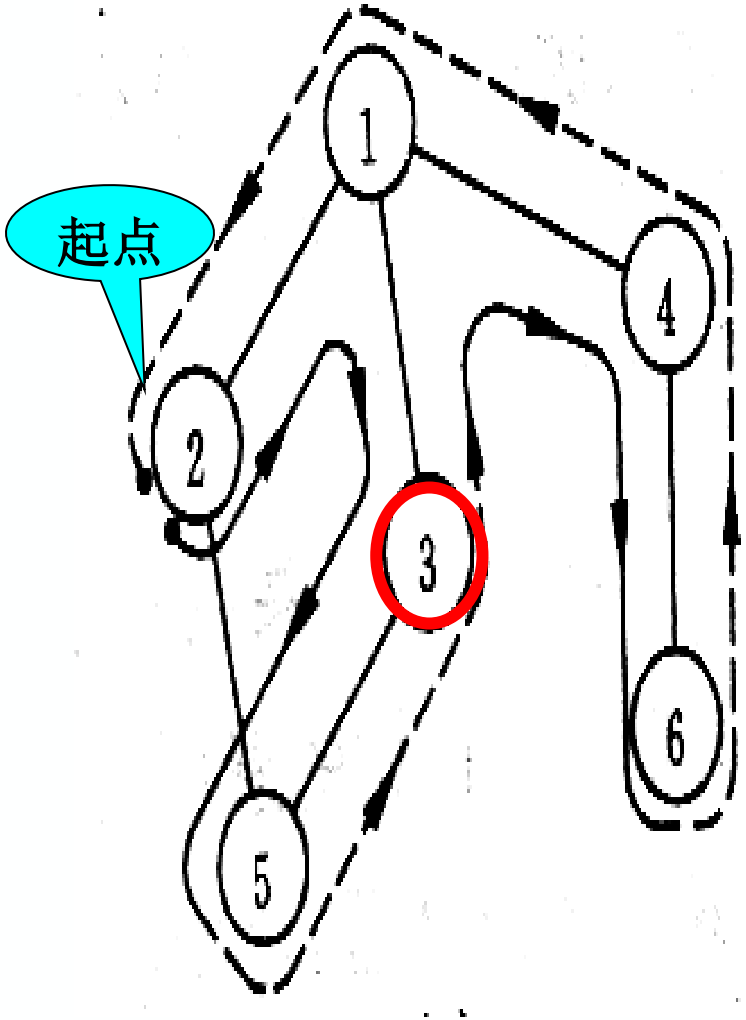
深度优先搜索练习



2→1→3→5→4→6

2020年6月1日

深度优先搜索的步骤



详细归纳：

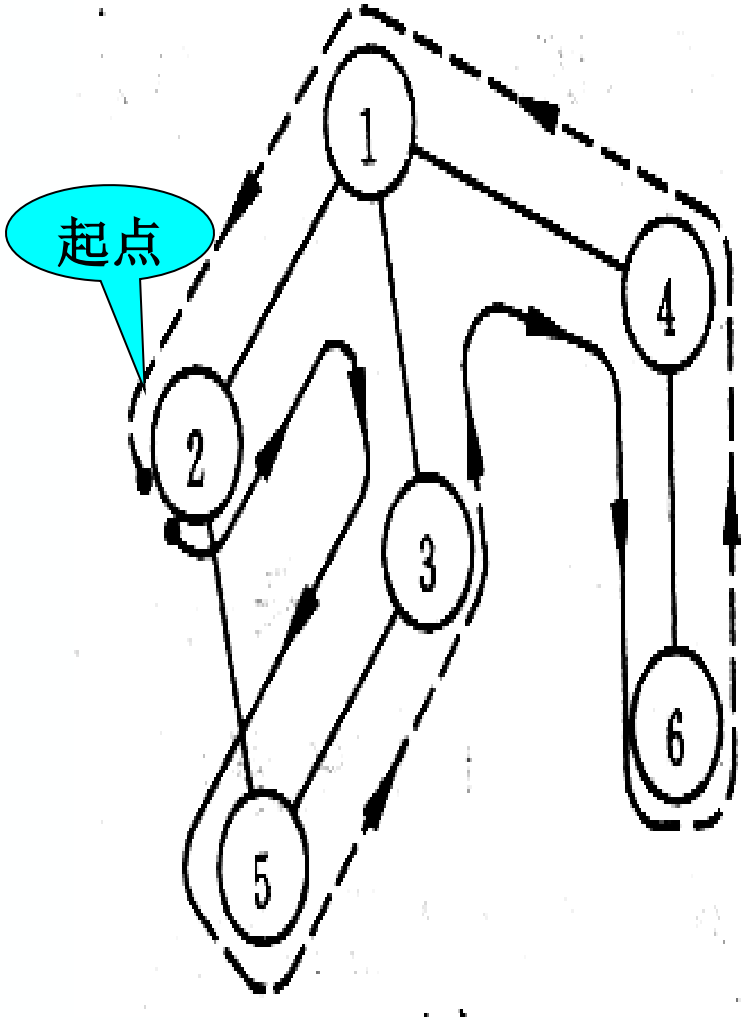
☛在访问图中某一起始顶点 v 后，由 v 出发，访问**它的任一邻接顶点** w_1；

☛再从 w_1 出发，访问**与 w_1 邻接但还未被访问过的顶点** w_2；

☛然后再从 w_2 出发，进行类似的访问，…

☛如此进行下去，直至到达所有的邻接顶点都被访问过的顶点 u 为止。

深度优先搜索的步骤



详细归纳：

☛接着，退回一步，**退到前一次刚访问过的顶点**，看是否还有其它没有被访问的邻接顶点。

如果有，则访问此顶点，之后再从此顶点出发，进行与前述类似的访问；

如果没有，就**再退回一步**进行搜索。重复上述过程，直到连通图中所有顶点都被访问过为止。

2→1→3→5→4→6

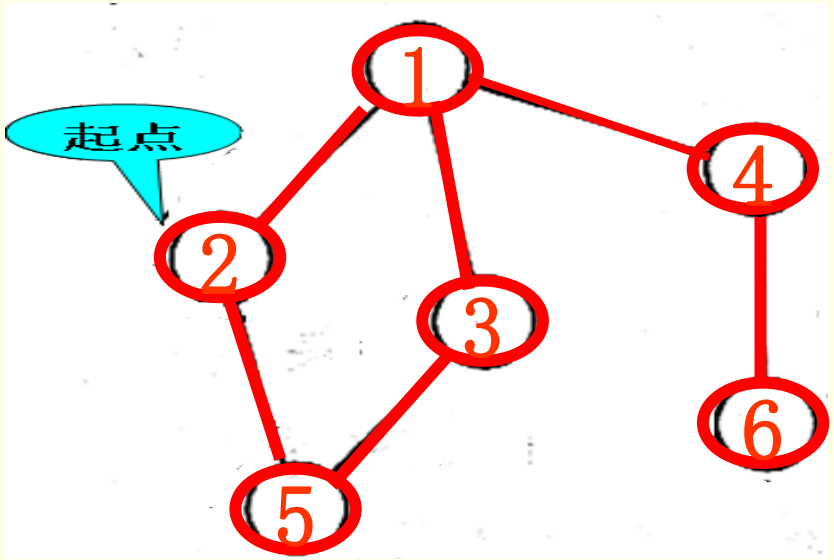
讨论1：计算机如何实现DFS？ ——开辅助数组 *visited* [n]!

邻接矩阵 A

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

辅助数组 *visited* [n]

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



DFS 结果

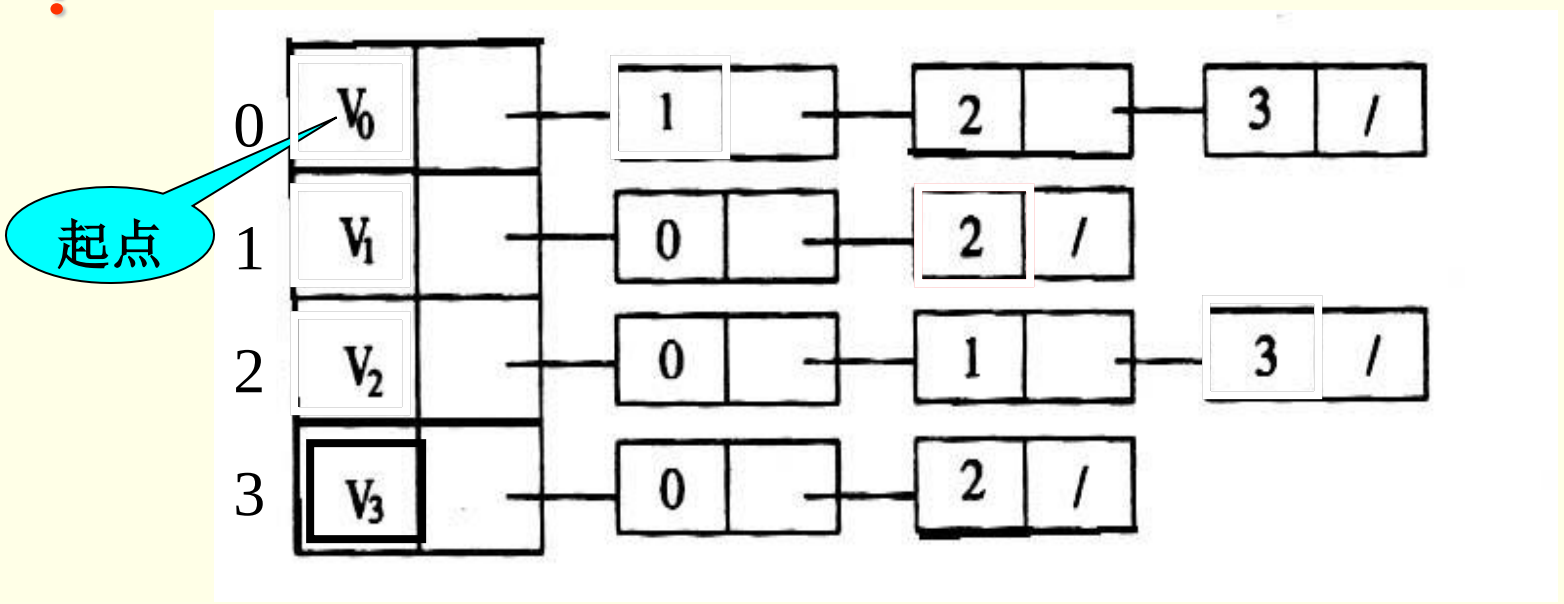
2→1→3→5→4→6

讨论2：DFS算法如何编程？——可以用递归算法！

```
void DFS(AMGraph G, int v){                    //图G为邻接矩阵类型
  cout<<v;  visited[v] = true;                 //访问第v个顶点
  for(w = 0; w< G.vexnum; w++)     //依次检查邻接矩阵v所在的行
      if((G.arcs[v][w]!=0)&& (!visited[w]))
          DFS(G, w);
    //w是v的邻接点，如果w未访问，则递归调用DFS
}
```

讨论3：在图的邻接表中如何进行DFS？

—照样借用*visited* [n]!



辅助数组 *visited* [n]

0	0	1	1	1	1
1	0	0	1	1	1
2	0	0	0	1	1
3	0	0	0	0	1

DFS 结果

v0 → v1 → v2 → v3

讨论4：邻接表的DFS算法如何编程？ ——仍可用递归算法

```
void DFS(ALGraph G, int v){                    //图G为邻接表类型
  cout<<v;  visited[v] = true;                 //访问第v个顶点
  p= G.vertices[v].firstarc;     //p指向v的边链表的第一个边结点
while(p!=NULL){                  //边结点非空
  w=p->adjvex;                   //表示w是v的邻接点
  if(!visited[w])  DFS(G, w); //如果w未访问，则递归调用DFS
  p=p->nextarc;                  //p指向下一个边结点
 }
}
```

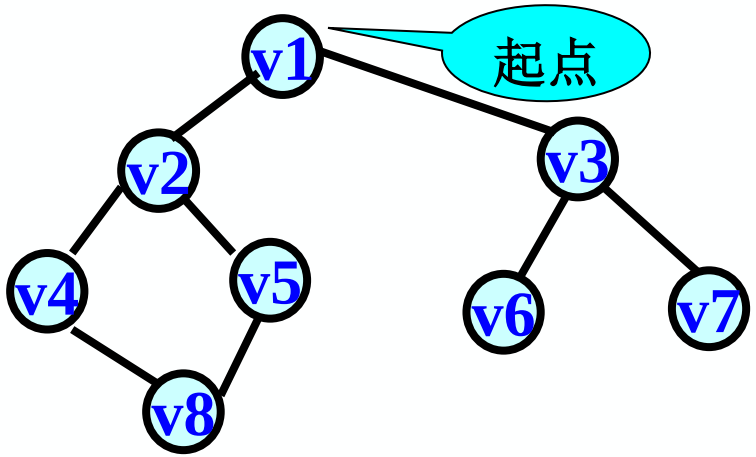
DFS算法效率分析

- 用邻接矩阵来表示图，遍历图中每一个顶点都要从头扫描该顶点所在行，时间复杂度为$O(n^2)$。
- 用邻接表来表示图，虽然有 $2e$ 个表结点，但只需扫描 e 个结点即可完成遍历，加上访问 n 个头结点的时间，时间复杂度为$O(n+e)$。

结论：
稠密图适于在邻接矩阵上进行深度遍历；
稀疏图适于在邻接表上进行深度遍历。

广度优先搜索(BFS - Breadth_First Search)

基本思想：——仿树的层次遍历过程



BFS 结果

v1→v2→v3→

v4→v5→v6→v7→v8

2020年6月1日

广度优先搜索的步骤

简单归纳：

- 在访问了起始点v之后，依次访问 v的邻接点；
- 然后再依次访问这些顶点中未被访问过的邻接点；
- 直到所有顶点都被访问过为止。

✓广度优先搜索是一种分层的搜索过程，每向前走一步可能访问一批顶点，不像深度优先搜索那样有回退的情况。

✓因此，广度优先搜索不是一个递归的过程，其算法也不是递归的。

【算法思想】

（1）从图中某个顶点*v*出发，访问*v*，并置visited[*v*]的值为true，然后将*v*进队。

（2）只要队列不空，则重复下述处理。

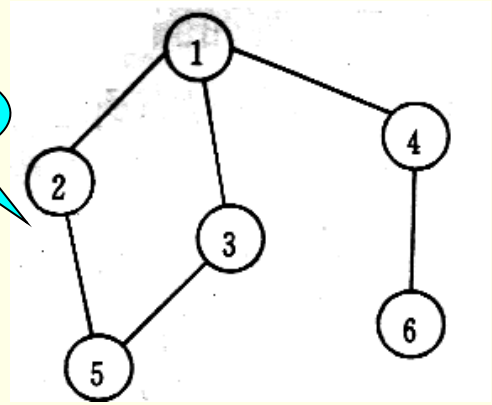
① 队头顶点*u*出队。

② 依次检查*u*的所有邻接点*w*，如果visited[*w*]的值为false，则访问*w*，并置visited[*w*]的值为true，然后将*w*进队。

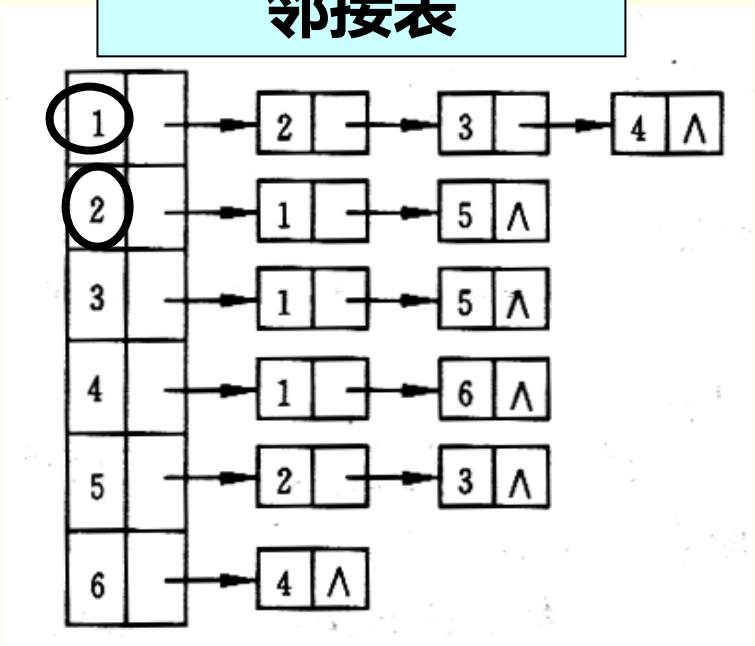
讨论1：计算机如何实现BFS？

—除辅助数组*visited* [n]外，还需再开一辅助队列

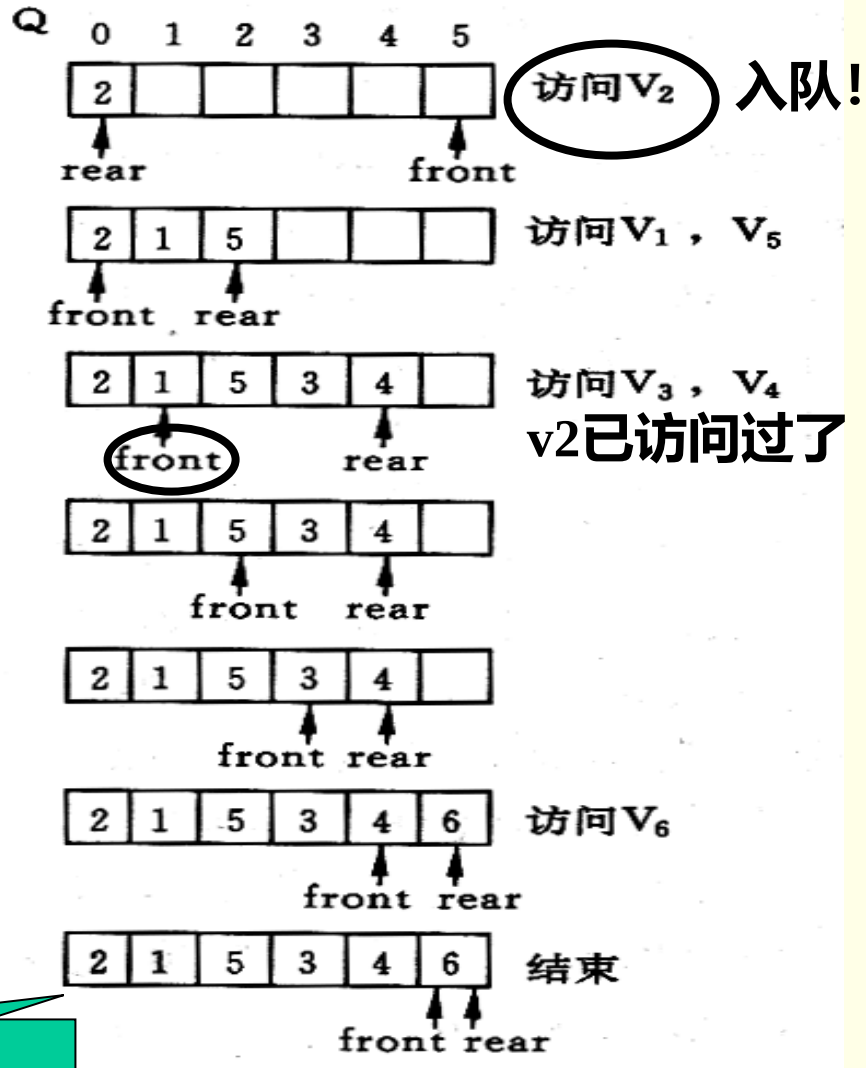
起点



邻接表



辅助队列



BFS 遍历结果

【算法描述】

```
void BFS (Graph G, int v){
   //按广度优先非递归遍历连通图G
   cout<<v; visited[v] = true;                    //访问第v个顶点
   InitQueue(Q);                                  //辅助队列Q初始化，置空
   EnQueue(Q, v);                                 //v进队
   while(!QueueEmpty(Q)){                         //队列非空
     DeQueue(Q, u);                               //队头元素出队并置为u
     for(w = FirstAdjVex(G, u); w>=0; w = NextAdjVex(G, u, w))
     if(!visited[w]){                        //w为u的尚未访问的邻接顶点
          cout<<w; visited[w] = true;          EnQueue(Q, w); //w进队
       }//if
   }//while
}//BFS
```

BFS算法效率分析

- 如果使用邻接矩阵，则BFS对于每一个被访问到的顶点，都要循环检测矩阵中的整整一行（n 个元素），总的时间代价为$O(n^2)$。
- 用邻接表来表示图，虽然有 2e 个表结点，但只需扫描 e 个结点即可完成遍历，加上访问 n个头结点的时间，时间复杂度为O(n+e)。