第四章 数据结构

- 4.1 基本概念
- 4.2 线性表
- 4.3 栈和队列
- 4.4 树
- 4.5 二叉树
- 4.6 二叉树的线性表示及生成
- 4.7 任意次树与二叉树之间的转换
- 4.9 图

4.9 图

4.9.1 图的定义

※图

设B=(K,R)是数据结构，K={k_1, k_2, …, k_n}，R只有**一种关系** R={r}，r={(k_s, k_t), | k_s, k_t∈K}，则称B为图。

※顶点

在图的术语中，把结点node称为顶点vertex。

因此，在图中结点集合记为**V={v_1, v_2 , …, v_n}**。

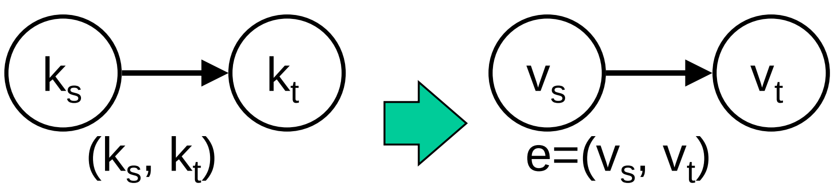
※边

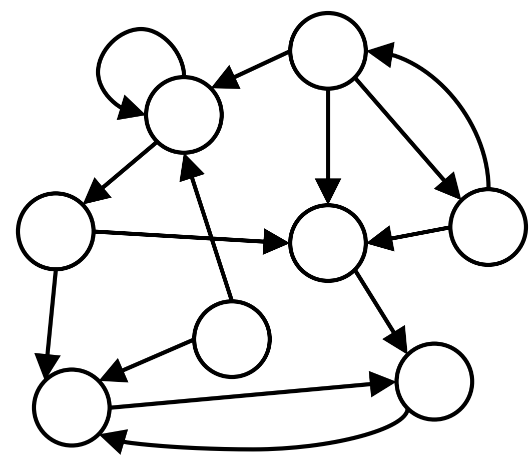
对于r中的(k_s, k_t)，k_s,k_t∈K，记为(v_s, v_t)，v_s,v_t∈V。称为从v_s到v_t存在一条边，记为**e_i=(v_s, v_t)**。

所有的边组成边的集合(边集)E，即

E={e_1,…,e_m, | e_i=(v_s,v_t), v_s,v_t ∈V}，

因而可将图记为**G=(V,E)**。





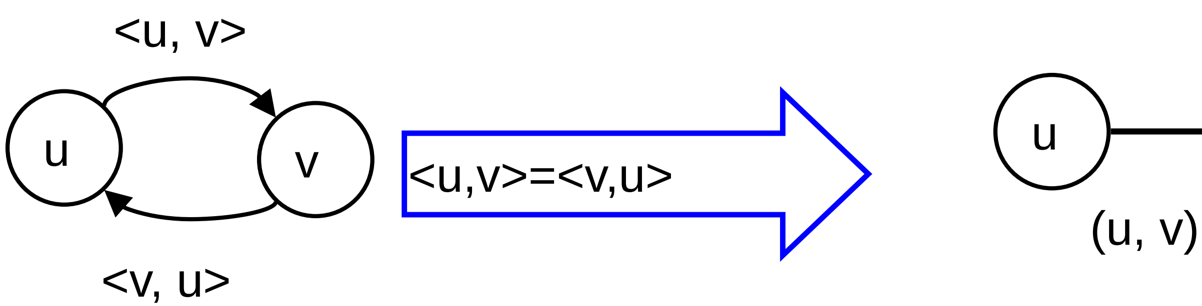
※有向边

在图G=(V,E)中，若u, v∈V，(u, v)∈E，且(u, v)≠(v, u)，则称(u, v)为有向边，记为 e=<u, v>，并称u为起点，v为终点。

※无向边

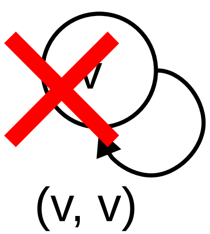
若u, v∈V，(u, v)和(v, u)∈E，且(u, v)=(v, u)，则称e=(u, v)为无向边。

同时在图形表示时，用**无箭头线表示e=(u, v)**。



※自向边

在图中，若v∈V，(v, v)∈E，则称(v, v)为自向边。
(在本课程中，不考虑自向边。)



※无向图

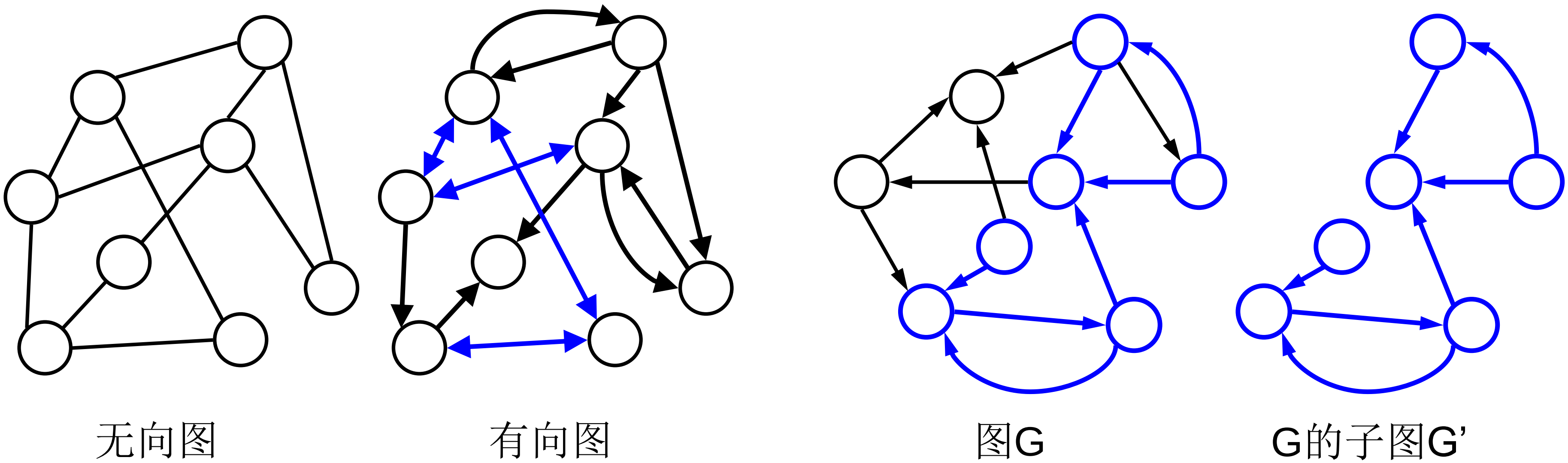
在G=(V,E)图中，若对所有的u,v∈V，e=(u, v)∈E，同时存在e'=(v, u)∈E，且e=e'，则称G为无向图。

※有向图

若有任一边e=(u, v)∈E，而(v, u)∉E，**或者**存在e=(v, u)∈E, e'=(u, v)∈E，但e≠e'，则称G为有向图。

※子图

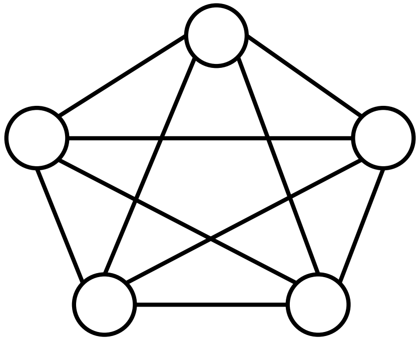
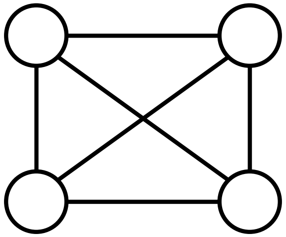
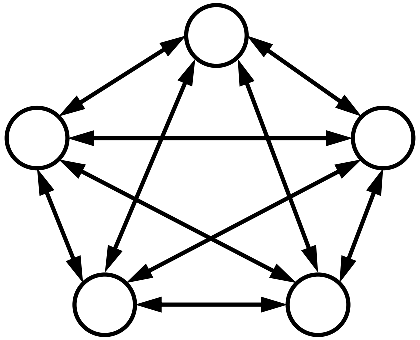
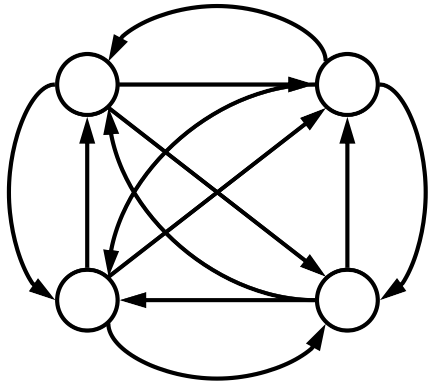
设G'=(V', E')，G=(V,E)，若成立V'⊆V，E'⊆E(称V'包含于V，E'包含于E)，即对所有的u,v∈V'，都存在u,v∈V，同时对所有的e=(u, v)∈E'，都存在e=(u, v)∈E，则称G'为G的子图。



※完全图

在无向图G中，若对任何两个顶点u和v，都存在无向边，称G为无向完全图。

在有向图G中，若对任何两顶点u和v，都存在边e=<u,v>和e’=<v,u>，称G为有向完全图。

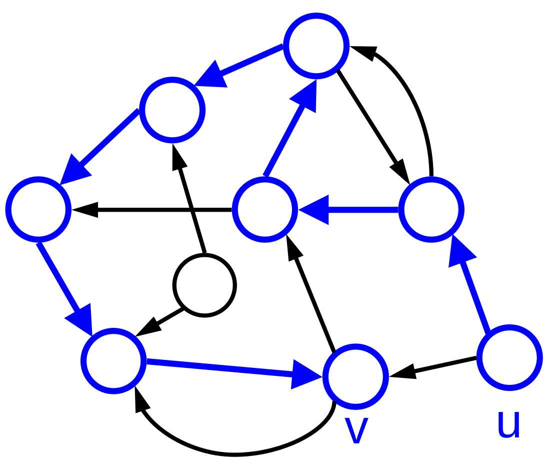





※通路

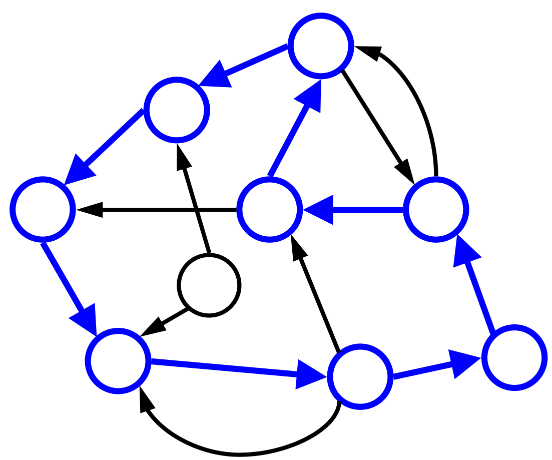
在图G=(V, E)中，如果存在一个顶点序列(v_0, v_1, v_2, …, v_k)，所有的(v_i, v_{i+1})$\in$E | i=0, …, k-1，而且v_0=u，v_k=v，则称在u和v之间存在通路。

※回路

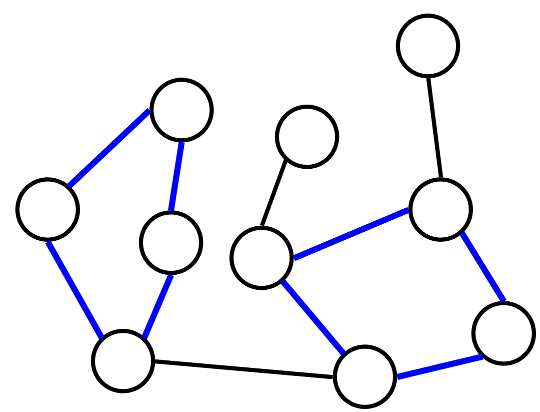
在图G =(V,E)中，若存在通路(v_0, v_1, v_2, …, v_k)，而v_0=v_k，则这个顶点序列称为回路。



通路u到v



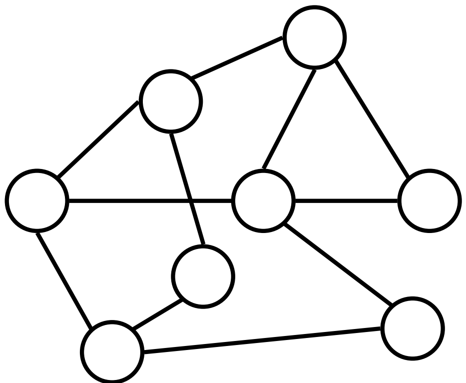
存在回路



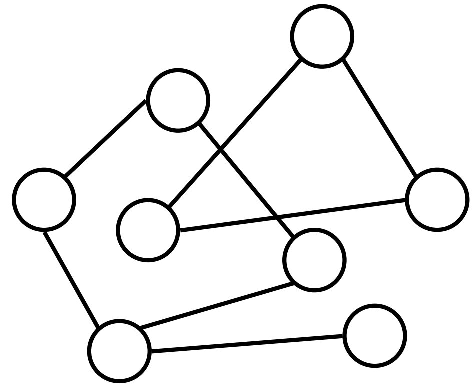
存在回路

⊙连通图

在无向图G=(V,E)中，如果任何两个顶点之间都存在通路，则称G为连通图。



连通图



非连通图

4.9.2 图的存储

图的存储有多种形式，这里只讨论两种常用的形式，即邻接矩阵和邻接表。

※邻接矩阵

设G=(V, E)是图，V={v_1, v_2 , …, v_n}，则可定义一个n×n的邻接矩阵A。矩阵元素A[i,j](i, j=1,…,n)被定义为

$$A[i, j] = \begin{cases} 1 & 若(v_i, v_j)\in E \\ 0 & 若(v_i, v_j)\notin E \end{cases}$$

例如，无向图G_1和有向图G_2可以分别用邻接矩阵A_1和邻接矩阵A_2表示，A_2中的粗体数字表示非对称的有向边，称为非对称元素。

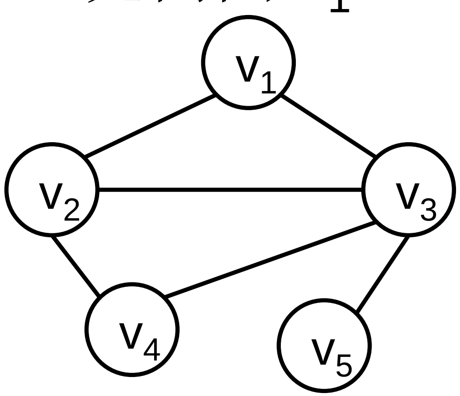
邻接矩阵A_1

$$A_1=\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

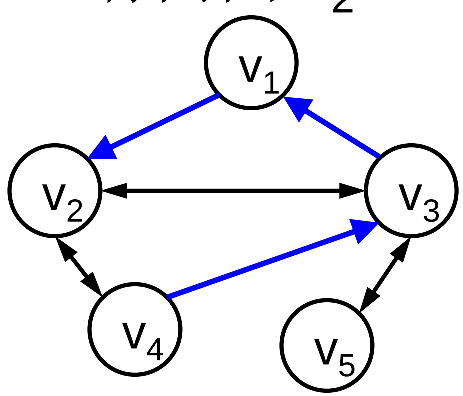
邻接矩阵A_2

$$A_2=\begin{pmatrix} 0 & \mathbf{1} & \mathbf{0} & 0 & 0 \\ \mathbf{0} & 0 & 1 & 1 & 0 \\ \mathbf{1} & 1 & 0 & \mathbf{0} & 1 \\ 0 & 1 & \mathbf{1} & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

无向图G_1



有向图G_2



※邻接表

在图G=(V,E)中，若$(v_i,v_j)\in E$, $v_i,v_j\in V$，则称v_j是v_i的邻接顶点。邻接表是指仅仅存储各顶点的邻接顶点的信息。

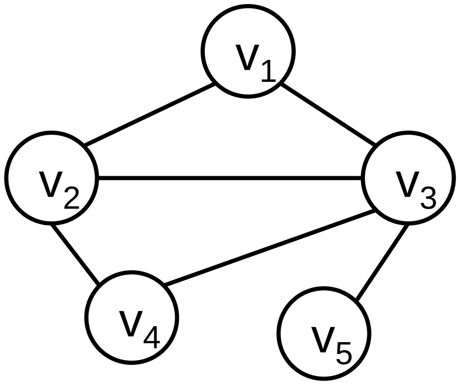
邻接表由表头和表链组成，表头用于存放所有顶点的数据场，表链用于存放每个v_i的邻接顶点的地址。

※邻接表存储单元示例

表头的存储单元：用next存放表头中下一单元的地址，用link存放表链的首地址。

表链的存储单元：用vertex存放顶点在表头中的地址，用next存放表链中下一单元的地址。

图G的邻接表及存储单元如下图所示。



图G=(V, E)

表头的存储单元

αv	δv	next	link
20	v_1	60	110
60	v_2	40	210
40	v_3	180	310
180	v_4	160	410
160	v_5	φ	510

210	20	220
220	180	230
230	40	φ
410	160	420
420	40	φ
520	40	φ

表链的存储单元

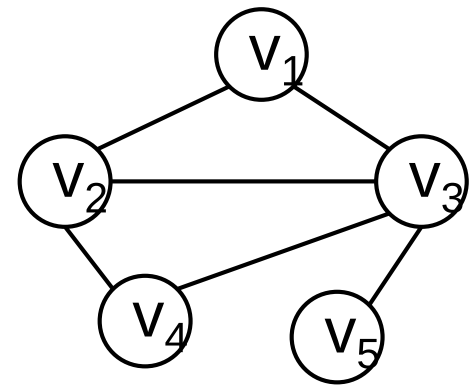
αv	vertex	next
110	60	120
120	40	φ
310	60	320
320	20	330
330	180	340
340	160	φ

指向表头中的顶点

※用数组存储的邻接表

存储邻接表的数据定义

```
#define    M     1000
#define VERTEX    struct vertex
VERTEX            /* 表头结构       */
{
   long   key;
   short  *link;     /* 表链首地址    */
   short num_link; /* 表链长度      */
};
VERTEX  Vertex[M];
```



其中，表头用结构数组Vertex实现存储，

表链用动态数组Vertex[i].link实现存储。

表头

αv	key	link
0	v_1	F160
1	v_2	F240
2	v_3	F310
3	v_4	F430
4	v_5	F560

表链

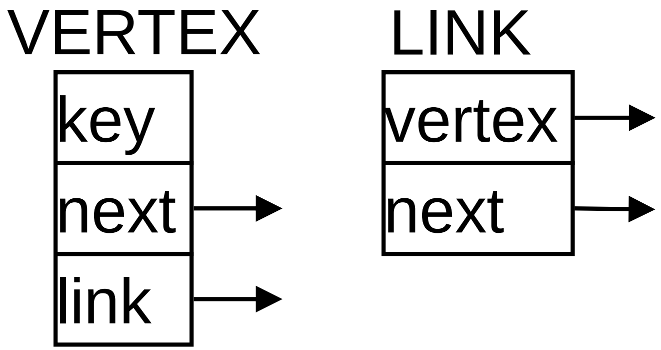
link[0]	link[1]	link[2]	link[3]
1	2		
0	2	3	
0	1	3	4
1	2		
3			

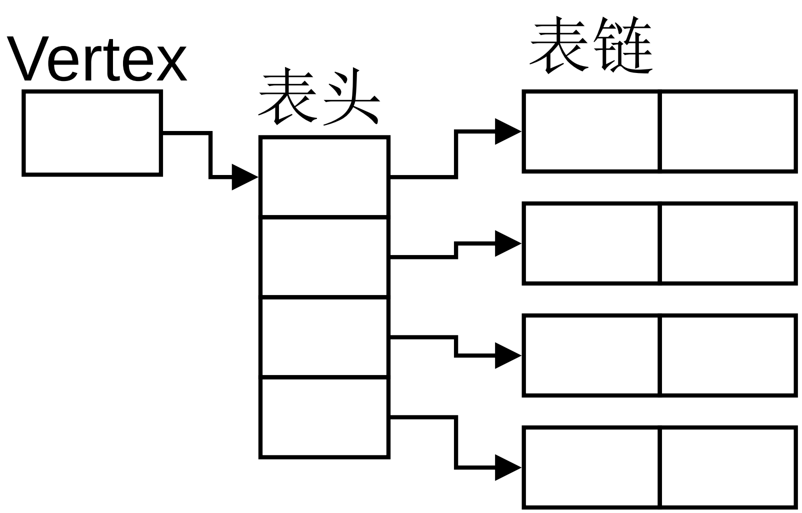
※用链表存储的邻接表

存储邻接表的数据定义

```
#define VERTEX  struct vertex
#define LINK       struct link
VERTEX       /* 表头结构   */
{
    long  key;  /* 数据场        */
    /* 表头中下一顶点指针  */
    VERTEX   *next;
    /* 顶点的表链指针          */
    LINK        *link;
};
LINK              /* 表链结构   */
{
    /* 表头中的顶点指针        */
    VERTEX   *vertex;
    /* 表链中下一结点指针  */
    LINK        *next;
};
VERTEX   *Vertex;
```

表头和表链分别是VERTEX和LINK类型的链表，因而称为用链表存储的邻接表。Vertex指向表头，表头的成员指针link指向各个表链。





※图的基本操作

⊙查找顶点或查找子图

⊙添加顶点或合并两个图

⊙删除顶点或子图

⊙遍历全部顶点

⊙寻找通路或回路

⊙…

4.9.3 图的遍历

※遍历的定义

从某一顶点出发开始访问，沿着边逐个访问顶点，直至遍及全图。当图为连通时，遍历是可能的。

※常用的遍历方法

DFS：深度优先搜索法(Depth First Search)

BFS：广度优先搜索法(Breadth First Search)

对树来说，

树的DFS就是树的前序遍历

树的BFS就是树的层次遍历

※DFS(深度优先搜索法)

☉算法描述

0) 初始化：所有顶点标记为未访问。

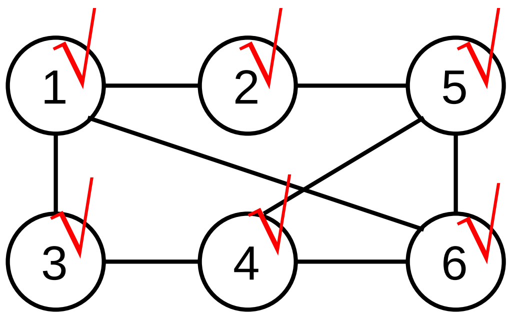
1) 指定一个顶点V，开始DFS。

2) 对给定的一个顶点V：

如果该点已访问过，则跳过。

如果该点未访问过，则访问之，并加注已访问的标记；

然后对于V的所有邻接顶点逐个执行DFS。

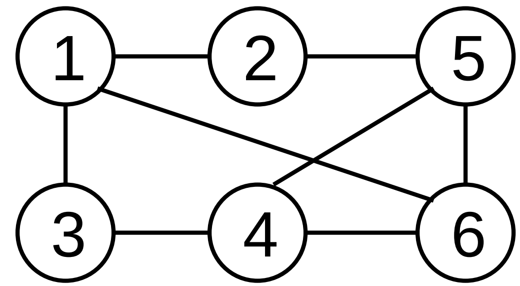


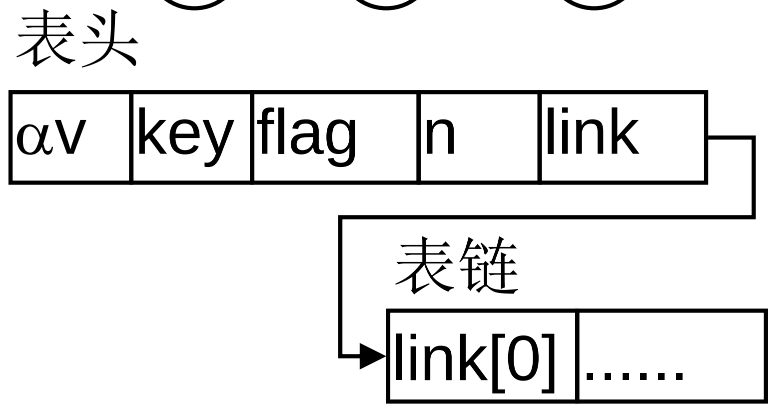
※用数组存储的邻接表实现DFS算法

⊙数据定义

```
#define    M    1000
#define    UNVISITED   'U'
#define    VISITED       'V'
#define    VERTEX  struct vertex
VERTEX
{
   long   key;
   char  flag;  /* 访问标记           */
   short  n;       /* 表链数组长度     */
   short  *link;  /* 表链动态数组     */
};
VERTEX  Vertex[M];
```

假定图G已经存储，即用数组存储的邻接表如图所示。

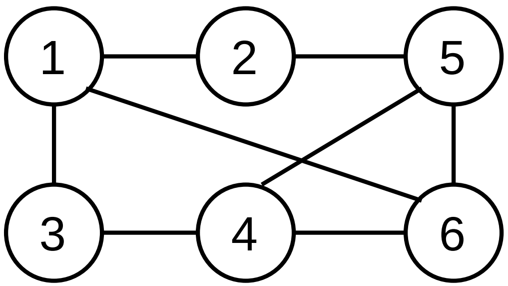




	表头				表链
0	v_1	flag	3	link	1 2 5
1	v_2	flag	2	link	0 4
2	v_3	flag	2	link	0 3
3	v_4	flag	3	link	2 4 5
4	v_5	flag	3	link	1 3 5
5	v_6	flag	3	link	0 3 4

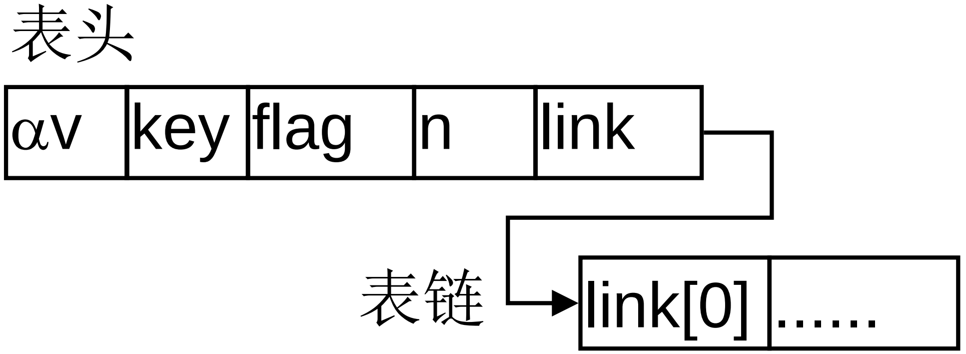
⊙DFS程序（**递归实现**）

```
void  dfs(short v)
{
    short  *link, i;
    printf("vertex=%ld\n", Vertex[v].key);
    Vertex[v].flag = VISITED;
    for(i=0, link=Vertex[v].link; i<Vertex[v].n; i++)
        if(Vertex[ link[i] ].flag == UNVISITED)
            dfs( link[i] );
}
```



如果选Vertex[0]为首个遍历的顶点，则dfs函数的调用形式为：

dfs(0);



※BFS(广度优先搜索法)

⊙算法描述

0) 初始化：所有顶点标记为未访问。

1) 指定一个顶点，送入**队列**。

2) 只要**队列**非空，从队列中取出一个顶点V，

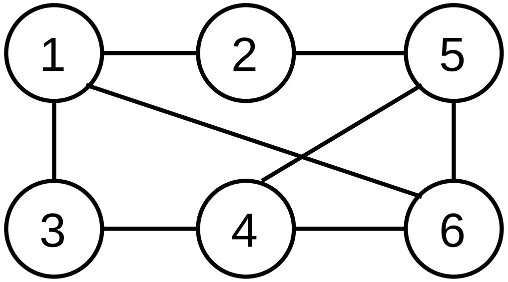
重复执行：

如果V已访问，跳过；

如果V未访问，加访问标记，打印；

将V的所有未访问过的邻接顶点送入队列。

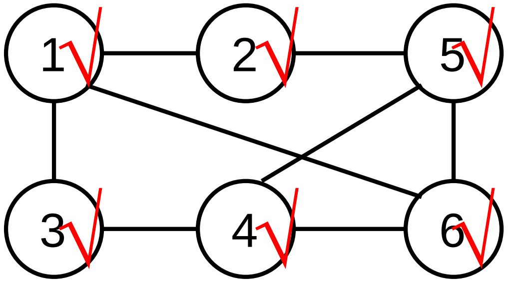
当队列空时结束遍历。

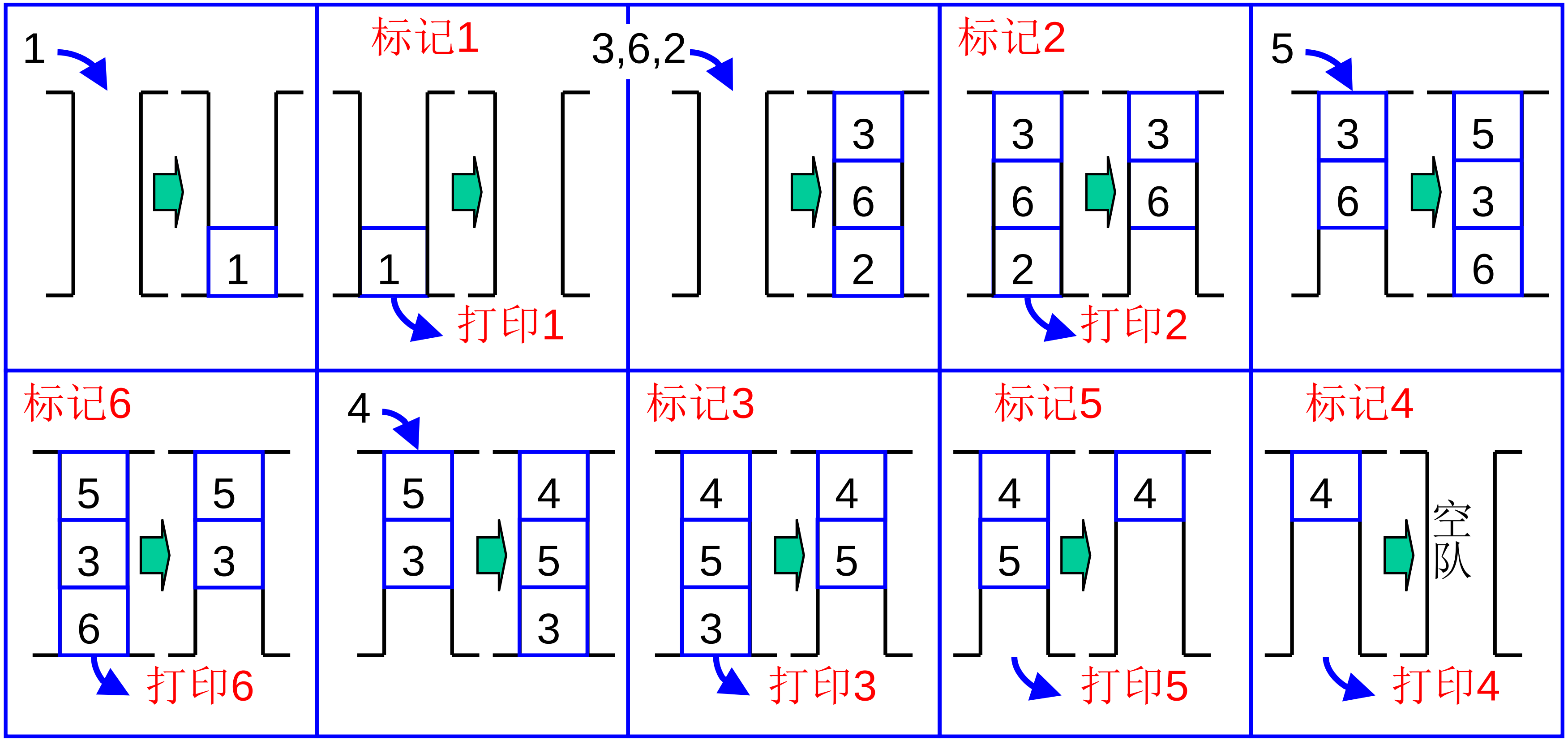


⊙BFS算法演示

1) 指定一个顶点，送入队列。
2) 只要队列非空，从队列中取出一个顶点V，重复执行：
   如果V已访问过，跳过；
   如果V未访问过，加访问标记，打印；
   将V的所有未访问过的邻接顶点送入队列。

当队列空时结束遍历。

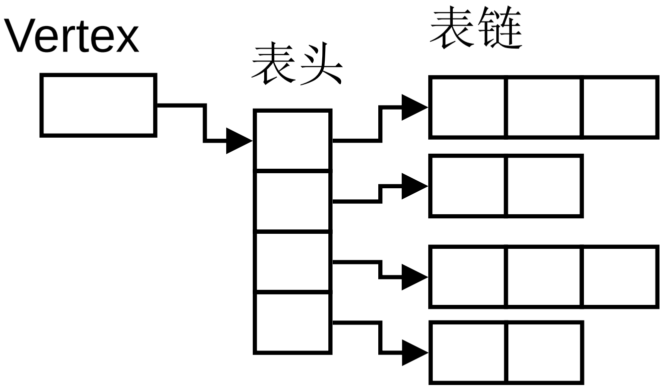


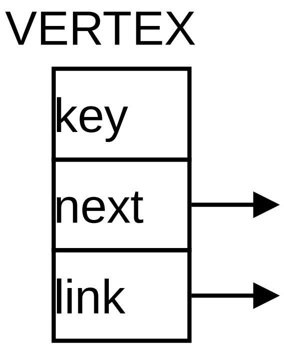


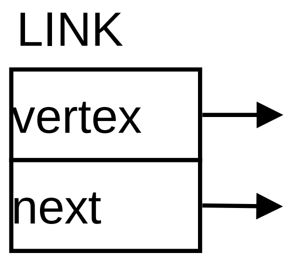
※用链表存储的邻接表实现BFS算法

⊙数据定义

```
#define   UNVISITED    0  /* 未访问   */
#define   VISITED      1  /* 已访问   */
#define   VERTEX     struct vertex
#define   LINK       struct link
VERTEX   /* 表头结构 */
{
    long    key;            /* 顶点数据值            */
    short     flag;   /* 访问标记，初始为未访问   */
    VERTEX  *next;          /* 表头链接指针          */
    LINK       *link;       /* 表链首指针            */
};
LINK        /* 表链结构 */
{
    VERTEX  *vertex;        /* 顶点指针              */
    LINK       *next;       /* 表链链接指针          */
};
VERTEX  *Vertex;          /* 邻接表首指针          */
void enQueue(VERTEX *v);  /* 进队函数              */
VERTEX  *deQueue(void);   /* 出队函数              */
```







※用链表存储的邻接表实现BFS算法

⊙BFS程序

```
void  bfs(VERTEX *first)    /* first为首个遍历的顶点*/
{
    VERTEX *vertex;
    LINK  *link;
    enQueue(first);                            /* first进队          */
    while( (vertex = deQueue()) )              /* 只要队非空        */
    {
        if(vertex->flag == UNVISITED)   /* 如果vertex未访问  */
        {
            printf("vertex=%ld\n", vertex->key); /* 打印vertex   */
            vertex->flag = VISITED;     /* vertex加标记      */
            for(link=vertex->link; link; link=link->next)
                if(link->vertex->flag == UNVISITED)
                    enQueue(link->vertex);/* 未访问邻接顶点进队*/
        }
    }
}
```

假设已编制队列操作函数enQueue和deQueue，图已经存储，即Vertex非空。如果选定首先遍历的顶点为v，则bfs函数的调用形式为：

bfs(v);

4.9.4 图的应用举例

- 最短路径(1959年)
- 欧拉回路(1736年)
- 哈密顿回路(1859年)
- 巡回售货员(货郎担)
- 地图的四色问题(1852年)
- 迷宫问题
- 皇后问题(1850年)
- ……

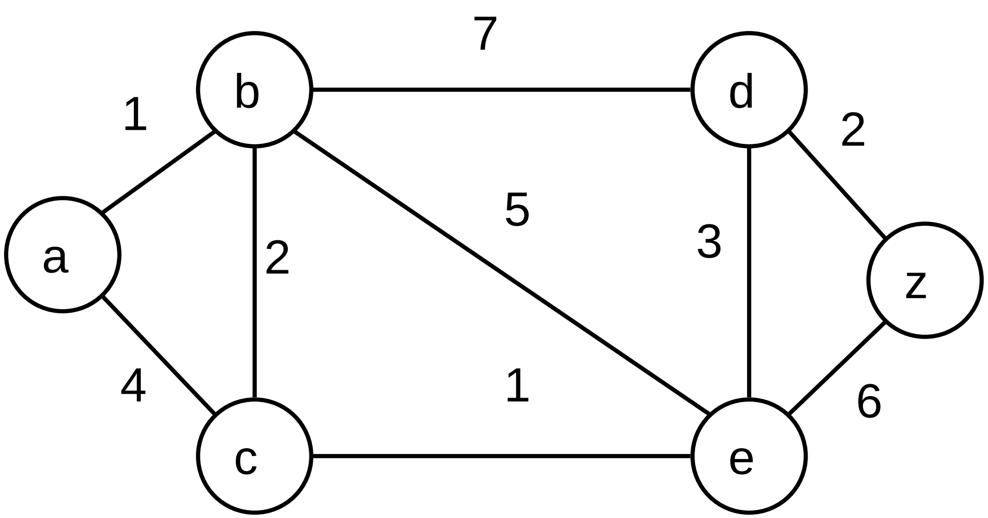
※图的应用举例

⊙最短路径

E. W. Dijkstra, 1959年提出，称为Dijkstra算法。

定义一个带权图G=(V,E,w)，对所有的边e=(u,v)∈E|u,v∈V, w(u,v)是边e的权(w可以是边e的路程，费用等)。

给定一个起点u和一个终点v，求从u出发到v的一条最短路径。

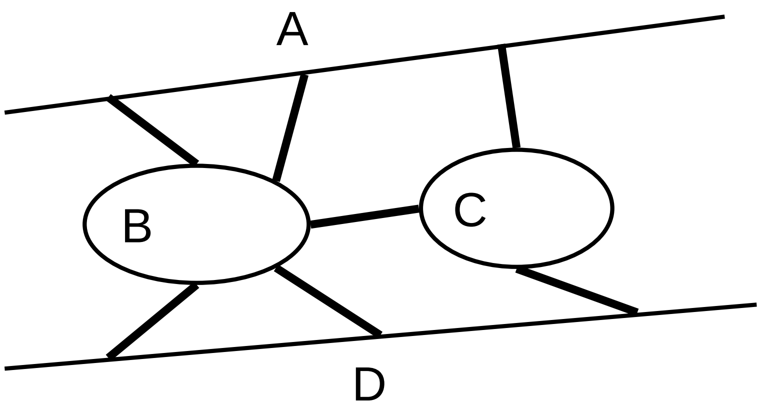


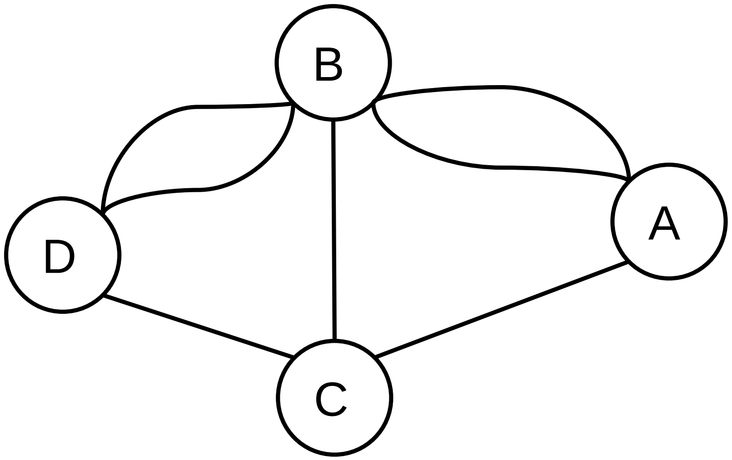
※图的应用举例

⊙欧拉回路(哥尼斯堡七桥问题)

1727年，欧拉的朋友向欧拉提出一个问题：在哥尼斯堡的匹格河上有七座桥，能否从任何一个地方出发通过每座桥一次且仅一次后回到原地? 1736年欧拉写出了第一篇图论的论文。

哥尼斯堡七桥问题等价于在图中寻找一条回路，使得它的**每一条边出现一次且仅一次**，也就是如何一笔画的问题。



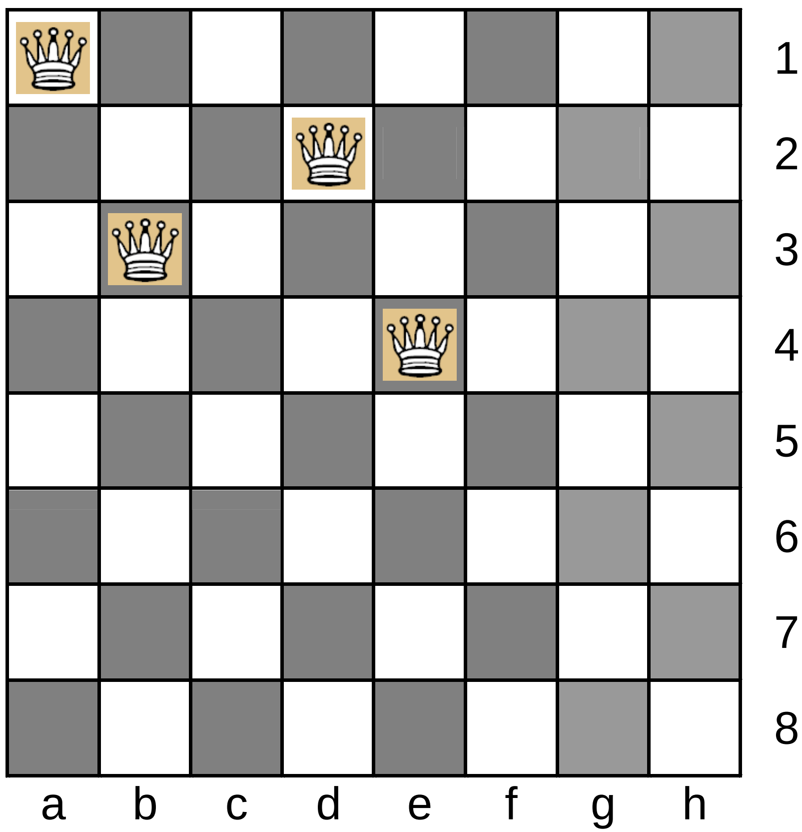


※图的应用举例

⊙皇后问题

皇后问题是一个古老而著名的问题，是回溯算法的典型例题。该问题是十九世纪著名的数学家高斯1850年提出：在8X8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。

高斯认为有76种方案。1854年在柏林的象棋杂志上不同的作者发表了40种不同的解，后来有人用图论的方法解出92种结果。

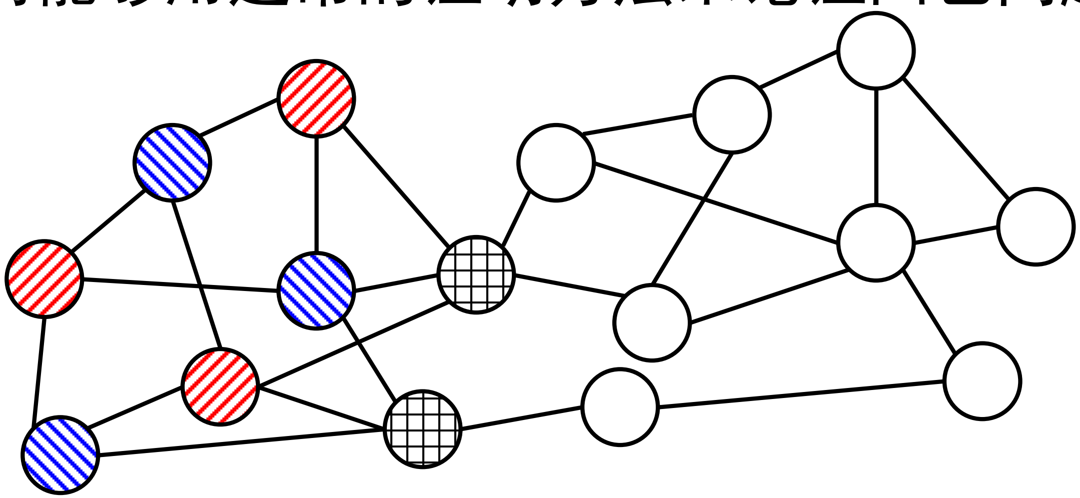


※图的应用举例

⊙地图的四色问题

1852年英国青年格思里(Guthrie)在画地图时发现：如果相邻的两国着上不同的颜色，那么画任何一张地图只要四种颜色就够了。这就是地图的四色问题。

100多年来，许多科学家的证明都失败了。直到1976年美国的阿尔培和哈根两位教授用计算机化了1200多个小时才得以证明。但是至今仍没有能够用通常的证明方法来论证四色问题。

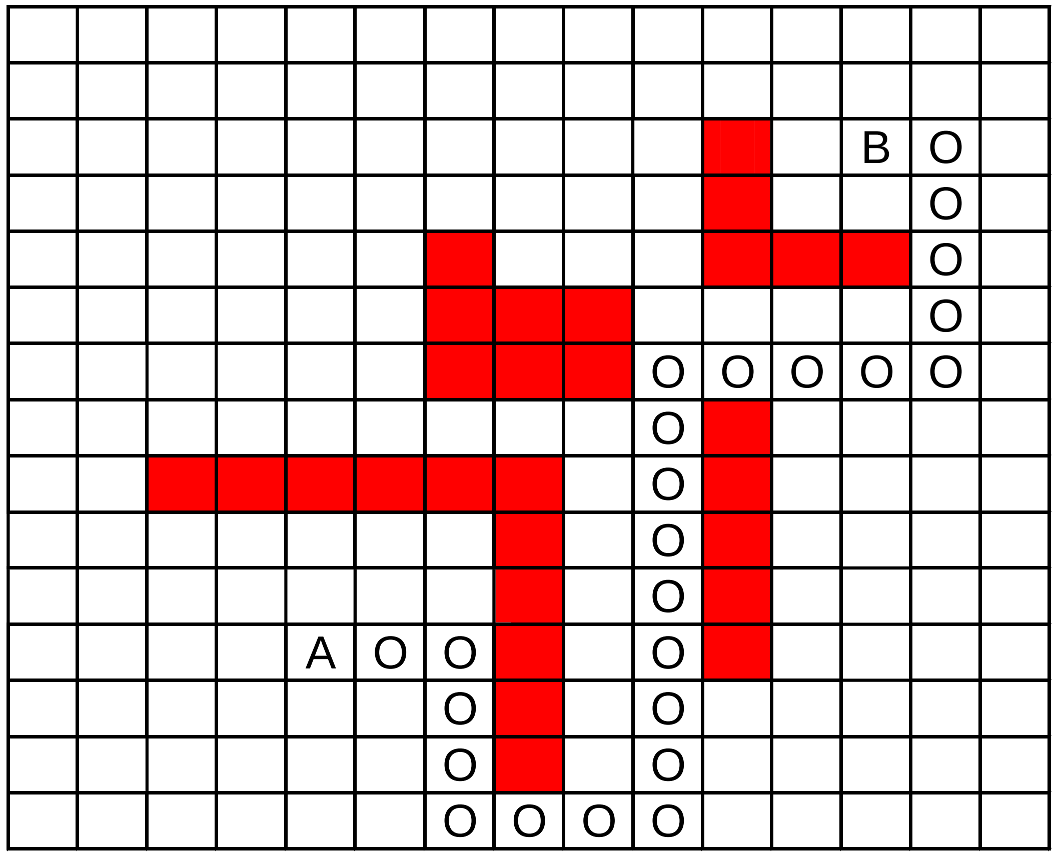


印刷电路板的分层问题。将图中的边对应于导线，顶点对应于接点，没有导线连接的两个接点间可能要装配元件。由于同一层上导线不允许交叉，问如何设计具有最少层数的印刷电路板？

※图的应用举例

⊙迷宫问题

从A到B间存在障碍，如何找到最佳的路径？



在集成电路和印刷电路板的布线时，迷宫算法(Lee氏算法)是一个最基本也是最有效的算法。