# 算法设计与分析

Computer Algorithm Design & Analysis

2019.11

王多强

dqwang@mail.hust.edu.cn

群名称：2019-算法

群　号：835135560

群名称：2019-算法

群　号：835135560

# 一、图的检索和周游

**被检测：** 在图中，当某结点的所有邻接结点都被访问了时，

称该结点**被检测**了。

经典的图检索算法:

- 宽度优先检索（BFS）
- 深度优先检索（DFS）

# 1. 图的宽度优先检索和周游

## （1）宽度优先检索

① 从结点v开始，首先访问结点v，并给v标上**已访问标记**。

② 访问邻接于v且目前尚未被访问的所有结点，此时结点**v被检测**，而v的这些邻接结点是**新的未被检测的结点**。将这些结点依次放置到一个称为**未检测结点表**的**队列**中。

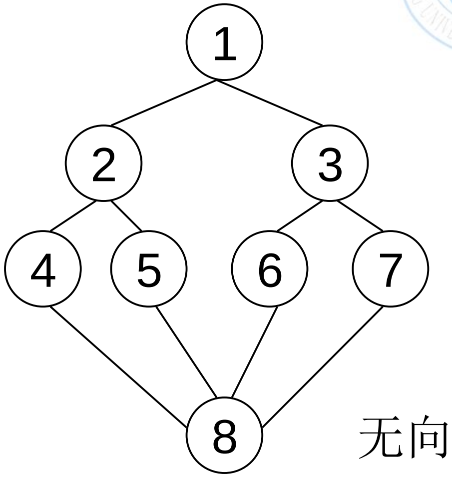
③ 若未检测结点表为空，则算法终止；否则

④ 取未检测结点表的表头结点作为下一个待检测结点，重复上述过程。直到Q为空，算法终止。

# 宽度优先检索算法

```
procedure BFS(v)
  //宽度优先检索G，它从结点v开始。所有已访问结点被标记为VISITED(i)=1。//
  VISITED(v)←1     //VISITED(1:n)是一个标志数组，初始值为VISITED(i)=0, 1≤i≤n //
  u←v
  将Q初始化为空          //Q是未检测结点的队列//
  loop
      for 邻接于u的所有结点w do
          if VISITED(w)=0 then     //w未被访问//
              call ADDQ(w,Q)       //ADDQ将w加入到队列Q的末端//
              VISITED(w)←1         //同时标示w已被访问//
          endif
      repeat
      if Q 为空 then return endif
      call DELETEQ(u,Q)      //DELETEQ取出队列Q的表头，并赋给变量u//
   repeat
end BFS
```

# 例：



无向图G

**检测结点1:**

visited(1) =1、visited(2)=1、 visited(3)=1

队列状态：

| | | |
|---|---|---|
| 2 | 3 | |

**检测结点2**（结点2出队列）：

visited(4) =1、Visited(5)=1

队列状态：

| | | | |
|---|---|---|---|
| 3 | 4 | 5 | |

**检测结点3**（结点3出队列）：

visited(6) =1、Visited(7)=1

队列状态：

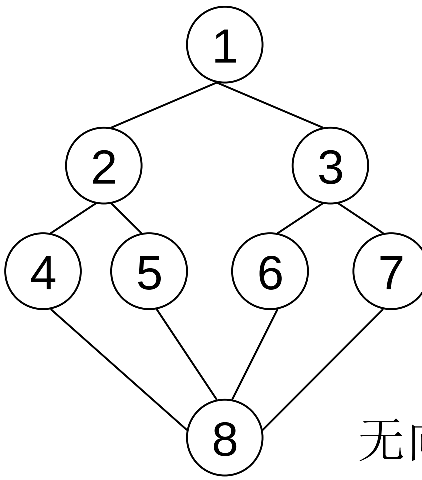
| | | | | |
|---|---|---|---|---|
| 4 | 5 | 6 | 7 | |

**检测结点4**（结点4出队列）：

visited(8) =1

队列状态：

| | | | | |
|---|---|---|---|---|
| 5 | 6 | 7 | 8 | |




无向图G

**检测结点5**（结点5出队列）：

队列状态：

| | | | |
|---|---|---|---|
| 6 | 7 | 8 | |

**检测结点6**（结点6出队列）：

队列状态：

| | | |
|---|---|---|
| 7 | 8 | |

**检测结点7**（结点7出队列）：

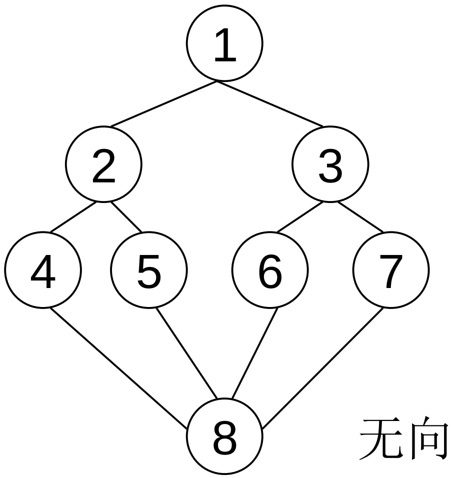
队列状态：

| | |
|---|---|
| 8 | |

**检测结点8**（结点8出队列）：

队列状态：

| |
|---|
| |




无向图G

BFS的结点访问序列：

1 2 3 4 5 6 7 8

# 定理7.2 算法BFS可以访问由v可到达的所有结点

证明：

设G=(V,E)是一个(有向或无向)图，v∈V。用**数学归纳法**证明如下：

记d(v,w)是由v到某一可到达结点w(w∈V)的最短路径的长度。

(1) 若d(v,w)≤1，则显然所有这样的w都将被访问。

(2) 假设对所有d(v,w)≤r的结点都可被访问。则当d(v,w)=r+1时有：

设w是V中具有d(v,w)=r+1的一个结点，u是从v到w的最短路径上紧挨着w的前一个结点。则有：d(v,u)=r。

根据归纳假设，u可通过BFS被访问到。

若u≠v,且r≥1。根据BFS的处理规则，u将在被访问时刻被放到未被检测结点队列Q上，而在另一时刻u将从队列Q中移出。此时，所有邻接于u且尚未被访问的结点将被访问。若结点w在这之前未被访问，则此刻将被访问到。

证毕。

**定理7.3** 设t(n,e)和s(n,e)是算法BFS在任一具有n个结点和e条边的图G上所花的时间和附加空间。

- 若G由邻接表表示，则$t(n,e)=\Theta(n+e)$和$s(n,e)=\Theta(n)$。
- 若G由邻接矩阵表示，则$t(n,e)=\Theta(n^2)$和$s(n,e)=\Theta(n)$

证明：

1) 空间分析

根据算法的处理规则，结点v不会放到队列Q中。结点w，$w\in V$且$w\neq v$，仅在VISITED(w)=0时由ADDQ(w,Q)加入队列，并置VISITED(w)=1，所以每个结点（除v)至多只有一次机会被放入队列Q中。

至多有n-1个这样的结点考虑，故总共至多做n-1次结点加入队列的操作。需要的队列空间至多是n-1。所以s(n,e)=O(n)(其余变量所需的空间为O(1))。

而当G是一v与其余的n-1个结点都有边相连的图时，邻接于v的全部n-1个结点都将在“同一时刻”被放在队列上，故Q至少也应有Ω(n)的空间。

同时，VISITED(n)本身需要Θ(n) 的空间。

所以s(n,e)=Θ(n)——这一结论与使用邻接表或邻接矩阵无关。

2） 时间分析

分两种存储结构讨论。

（1）G采用邻接表表示时，判断邻接于u的结点将在d(u)时间内完成，这里，若G是无向图，则d(u)是u的度；若G是有向图，则d(u)是u的出度。

- 所有结点的处理时间：O(Σd(u))=O(e)。

  注：嵌套循环中对G中的每一个结点至多考虑一次。

- VISITED数组的初始化时间：O(n)

所以，算法总时间：**O(n+e)**。

（2）若G采用邻接矩阵表示，判断邻接于u的所有结点需要$\Theta(n)$的时间，则所有结点的处理时间：$O(n^2)$

算法总时间：$O(n^2)$

如果G是一个由v可到达所有结点的图，则将检测到V中的所有结点，所以上两种情况所需的总时间至少应是$\Omega(n+e)$和$\Omega(n^2)$。

所以，$t(n,e)=\Theta(n+e)$ 使用邻接表表示

或， $t(n,e)=\Theta(n^2)$ 使用邻接矩阵表示

证毕。

# （2）宽度优先周游

图的宽度优先周游算法

```
procedure BFT(G,n)
    //G的宽度优先周游//
    int VISITED(n)
    for i←1 to n do VISITED(i)←0 repeat
    for i←1 to n do  //反复调用BFS//
        if VISITED(i)=0 then call BFS(i)  endif
    repeat
end BFT
```

**注**：若G是无向连通图或强连通有向图，则一次调用BFS即可完成对G的周游。否则，需要多次调用BFS。

# 图周游算法的应用

●判定图G的连通性：若调用BFS的次数多于1次，则G为非连通的。

●生成图G的连通分图：一次调用BFS中所访问到的所有结点及连接这些结点的边构成一个连通分图。

●构造无向图**自反传递闭包矩阵$A^*$**：若两个结点i，j在同一个连通分图中，则在自反传递闭包矩阵中$A^*[i,j]=1$。

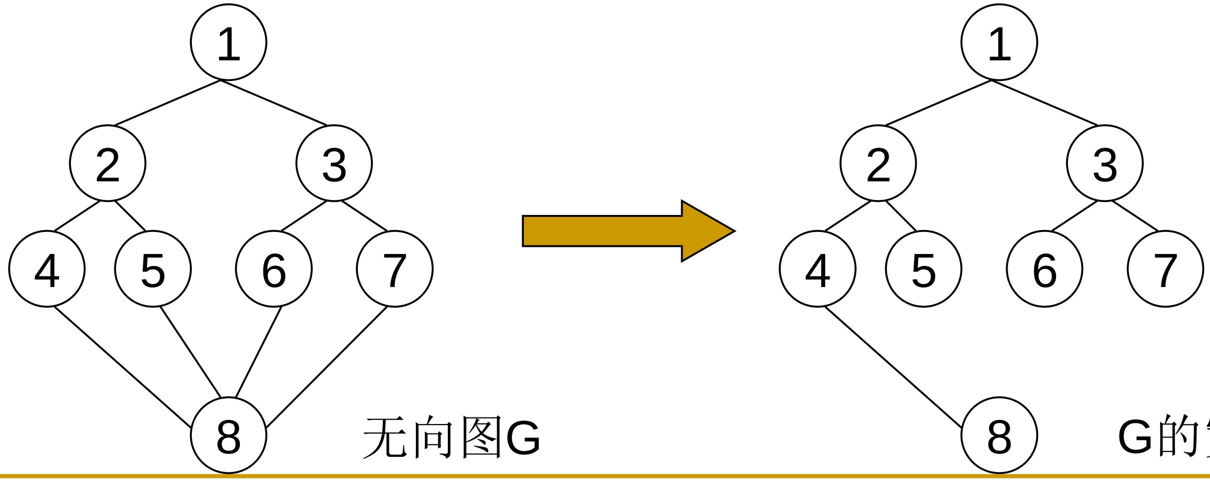
## ● 宽度优先生成树

**向前边**：BFS中由u达到未访问结点w的边(u,w)称为**向前边**。

**宽度优先生成树**：

记T是BFS中处理的所有向前边集合。

若G是连通图，则BFS终止时，T构成一棵生成树，称为图G的**宽度优先生成树**。




无向图G

G的宽度优先生成树

修改算法BFS，在第1行和第6行分别增加语句T←Φ和T←T∪{(u,w)}。修改后的算法称为BFS*。若v是连通无向图中任一结点，调用BFS*，算法终止时，**T中的边组成G的一棵生成树**。

```
procedure BFS*(v)
    VISITED(v)←1;u←v
    T←Φ
    将Q初始化为空
    loop
        for 邻接于u的所有结点w do
            if VISITED(w)=0 then      //w未被检测//
                T←T∪{(u,w)}
                call ADDQ(w,Q)        //ADDQ将w加入到队列Q的末端//
                VISITED(w)←1          //同时标示w已被访问//
            endif
        repeat
        if Q 为空 then return endif
        call DELETEQ(u,Q)             //DELETEQ取出队列Q的表头，并赋给变量u//
    repeat
end BFS*
```

## 证明：

若G是n个结点的连通图，则这n个结点都要被访问。除起始点v以外，其它n-1个结点都将被放且仅将被放到队列Q上一次，从而T将正好包含n-1条边，且这些边是各不相同的。即T是关于n个结点n-1边的无向图。

同时，对于连通图G，T将包含由起始结点v到其它结点的路径，所以T是连通的。

则T是G的一棵生成树。

注：**有n个结点且正好有n-1条边的连通图恰好是一棵树。**

# 2. 深度优先检索和周游

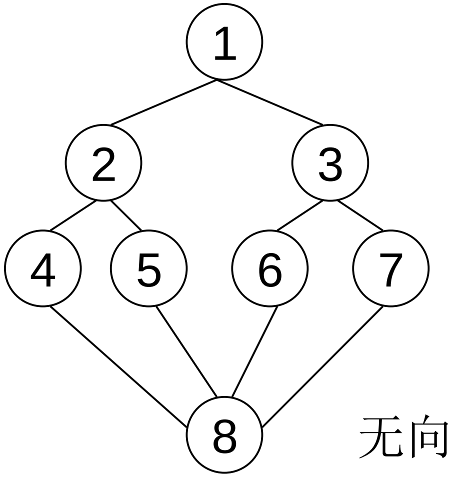
## (1) 深度优先检索

从结点v开始，首先访问v，并给v标上已访问标记；然后中止对v的检测，并从邻接于v且尚未被访问的结点的中找出一个结点w开始新的检测。在w被检测后，再恢复对v的检测。当所有可到达的结点全部被检测完毕后，算法终止。

### 图的深度优先检索算法

```
procedure DFS(v)
//已知一个n结点图G＝(V,E)以及初值已置为零的数组VISITED(1:n)，访问由v可以到达的所有结点。//
    VISITED(v)←1
    for 邻接于v的每个结点w do
        if VISITED(w)＝0 then call DFS(w) endif
    repeat
END DFS
```

例：



无向图G

DFS结点访问序列：

1→2 →4 →8 →5 →6 →3 →7

## 性质：

① DFS可以访问由v可到达的所有结点

② 如果t(n,e)和s(n,e)表示DFS对一n结点e条边的图所花的时间和附加空间，则

- $s(n,e)=\Theta(n)$
- $t(n,e)= \Theta(n+e)$　G采用邻接表表示，或
- $t(n,e)= \Theta(n^2)$　G采用邻接矩阵表示

## (2) 深度优先周游算法DFT

反复调用DFS，直到所有结点均被检测到。

应用：

① 判定图G的连通性

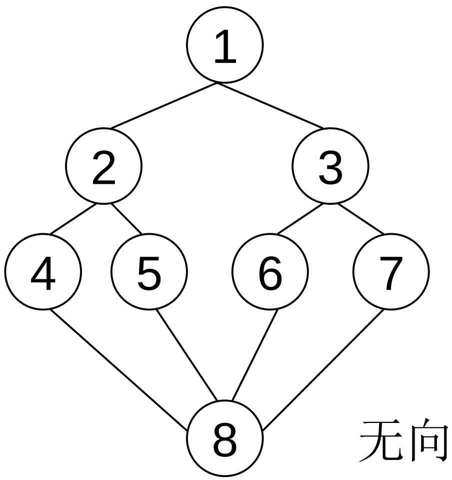
② 连通分图

③ 无向图的自反传递闭包矩阵

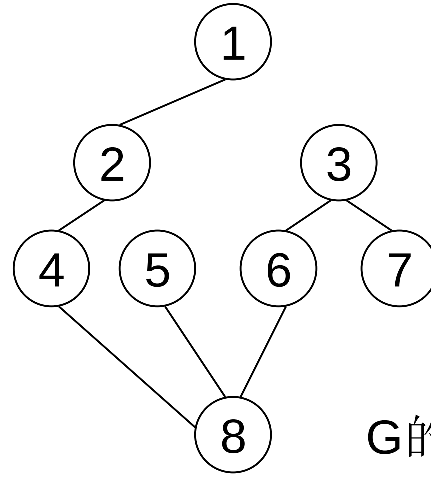
④ 深度优先生成树

# 生成深度优先生成树的算法

```
T←Φ
procedure DFS*(v, T)
  VISITED(v)←1
   for 邻接于v的每个结点w do
       if VISITED(w)＝0 then
            T←T∪{(u,w)}
            call DFS(w,T)
       endif
   repeat
END DFS*
```

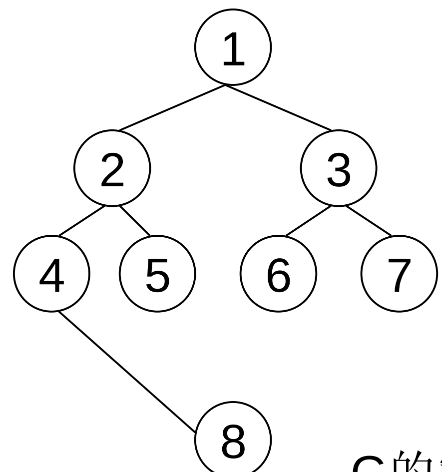



无向图G




1,2,4,8,5,6,3,7

G的深度优先生成树



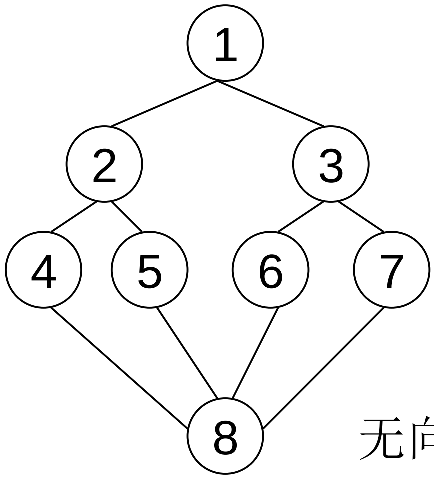

1,2,3,4,5,6,7,8

G的宽度优先生成树

# 3. D_Search：深度检索

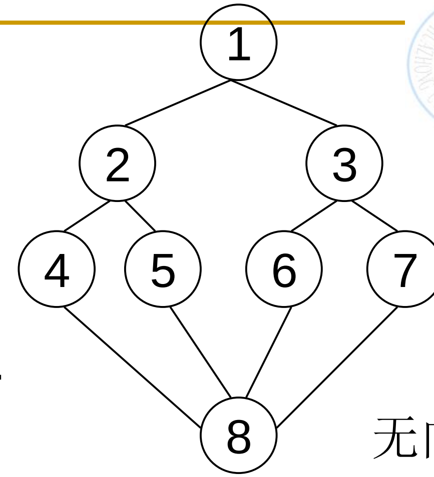
改造BFS算法，用栈来保存未被检测的结点，则得到的新的检索算法称为深度检索（D_Search）算法。

注：结点被压入栈中后将以相反的次序出栈。




无向图G

例：

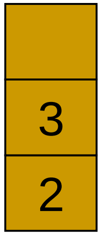



无向图G

**检测结点1**:

visited(1) =1、Visited(2)=1、 Visited(3)=1

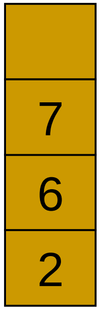
栈状态：




**检测结点3**（结点3出栈）：

visited(6) =1、Visited(7)=1

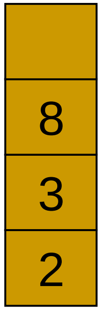
栈状态：




**检测结点7**（结点7出栈）：

visited(8) =1

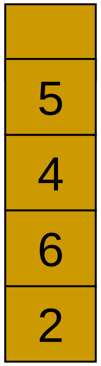
栈状态：

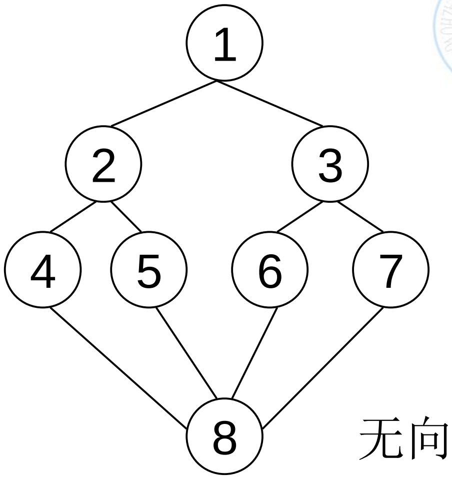


**检测结点8**（结点8出栈）：

visited(4) =1、 visited(5) =1

栈状态：

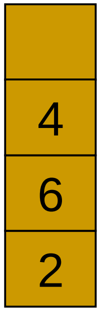



无向图G

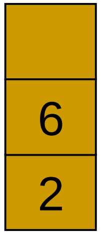
**检测结点5**（结点5出栈）：

栈状态：



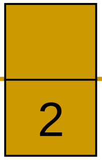

**检测结点4**（结点4出栈）：

栈状态：



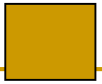

**检测结点6**（结点6出栈）：

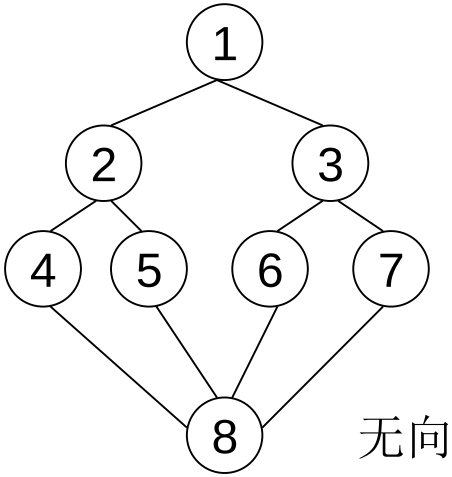
栈状态：




**检测结点2**（结点2出栈）：

栈状态：






无向图G

D_Search的结点访问序列:

1, 2, 3, 6, 7, 8, 4, 5

五大常用算法：

分治、动态规划、贪心、回溯和分支限界

参考： http://blog.csdn.net/yapian8/article/details/28240973

带有剪枝的搜索：回溯和分支限界

# 二、回溯法

回溯法是算法设计的基本方法之一。用于求解问题的一组特定性质的解或满足某些约束条件的最优解。

**什么样的问题适合用回溯法求解呢?**

1）问题的解可用一个n元组$(x_1,...,x_n)$的向量来表示；

- 其中的$x_i$取自于某个**有穷集$S_i$**。

2）问题的求解目标是求取一个使某一规范函数$P(x_1,...,x_n)$取极值或满足该规范函数条件的向量（也可能是满足P的所有向量）。

# 如何求取满足规范函数的元组?

1） 硬性处理法(brute force)

- 枚举，列出所有候选解，逐个检查是否为所需要的解

  假定集合$S_i$的大小是$m_i$，则候选元组个数为

  $$m = m_1 m_2 \ldots m_n$$

- 缺点：盲目求解，计算量大，甚至不可行

2）寻找其它有效的策略

回溯或分枝限界法

# 回溯（分枝限界）法带来什么样的改进?

- 对可能的元组进行**系统化搜索**，避免盲目求解。
- 在求解的过程中，**逐步构造元组分量**，并在此过程中，通过不断修正的规范函数（限界函数）去测试正在构造中的n元组的部分向量（$x_1, \cdots, x_i$），看其能否导致问题的解。
- 如果判定（$x_1, \cdots, x_i$）不可能导致问题的解，则将后面可能要测试的$m_{i+1}\cdots m_n$个向量一概略去——**剪枝**，这使得相对于硬性处理大大减少了计算量。

# 概念

- **约束条件**：问题的解需要满足的条件。

  可以分为**显式约束条件**和**隐式约束条件**。

  **显式约束条件**：一般用来规定每个$x_i$的取值范围。

  如： $x_i \geq 0$ 即$S_i$={所有非负实数}

  $x_i=0$或$x_i=1$ 即 $S_i=\{0,1\}$

  $l_i \leq x_i \leq u_i$ 即$S_i=\{l_i \leq a \leq u_i\}$

  **解空间**：实例I的**满足显式约束条件**的所有元组，构成I的解空间，即所有$x_i$合法取值的元组的集合——可行解。

  **隐式约束条件**：用来规定I的解空间中那些满足规范函数的元组，隐式约束条件描述$x_i$彼此之间的关系和应满足的条件。

# 例：8-皇后问题

在一个8×8棋盘上放置8个皇后，且使得每两个皇后之间都不互相“攻击”：每两个皇后都不在同一行、同一列或同一条斜角线上。

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | Q | | | | |
| 2 | | | | | | Q | | |
| 3 | | | | | | | | Q |
| 4 | | Q | | | | | | |
| 5 | | | | | | | Q | |
| 6 | Q | | | | | | | |
| 7 | | | Q | | | | | |
| 8 | | | | | Q | | | |

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | | | Q | | | | | |
| 2 | | | | | Q | | | |
| 3 | | Q | | | | | | |
| 4 | | | | | | | | Q |
| 5 | Q | | | | | | | |
| 6 | | | | | | | Q | |
| 7 | | | | Q | | | | |
| 8 | | | | | | Q | | |

**行、列号**：1…8

**皇后编号**：1…8, 不失一般性，约定皇后i放到第i行的某一列上。

**解的表示**：可以用8-元组$(x_1,\ldots,x_8)$表示，其中$x_i$是皇后i所在的列号。

**显式约束条件**：$S_i=\{1,2,3,4,5,6,7,8\}$, $1\leqslant i\leqslant 8$

**解空间**：所有可能的8元组，共有$8^8$个。

**隐式约束条件**：用来描述$x_i$之间的关系，即没有两个$x_i$可以相同且没有两个皇后可以在同一条斜角线上。

由隐式约束条件可知：可能的解只能是（1,2,3,4,5,6,7,8）的置换（排列），最多有8！个。

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | | | | Q | | | | |
| 2 | | | | | | Q | | |
| 3 | | | | | | | | Q |
| 4 | | Q | | | | | | |
| 5 | | | | | | | Q | |
| 6 | Q | | | | | | | |
| 7 | | | Q | | | | | |
| 8 | | | | | Q | | | |

图中的解表示为一个8-元组为（4，6，8，2，7，1，3，5）

另一个解是：（3，5，2，8，1，4，6，7）

# 例 子集和数问题

已知n个正数的集合W={$w_1$, $w_2$, …, $w_n$}和正数M。要求找出W中的和数等于M的所有子集。

例： n＝4，（$w_1$, $w_2$, $w_3$, $w_4$）＝（11，13，24，7），M=31。则满足要求的子集有：

- **直接用元素表示**：（11，13，7）和（24，7）
- k-**元组（用元素下标表示）**：（1，2，4）和（3，4）
- n-**元组（用n元向量表示）**：（1，1，0，1）和（0，0，1，1）

# 子集和数问题解的表示：

**形式一：**

问题的解为**k-元组**$(x_1, x_2, \ldots, x_k)$, $1≤k≤n$。不同的解可以是大小不同的元组，如(1,2,4)和(3,4)。

显式约束条件：$x_i \in \{ j \mid j$为整数且$1≤j≤n \}$。

隐式约束条件：1）没有两个$x_i$是相同的；

2）$w_{xi}$的和为M；

3）$x_i < x_{i+1}, 1≤i<n$（避免重复元组）

**形式二：**

解由**n-元组**$(x_1, x_2, \ldots, x_n)$表示，其中$x_i \in \{0,1\}$。如果选择了$w_i$，则$x_i=1$，否则$x_i=0$。

**例：**（1，1，0，1）和（0，0，1，1）

特点：所有元组具有统一固定的大小。

显式约束条件：$x_i \in \{0,1\}$，$1 \le i \le n$;

隐式约束条件：$\Sigma(x_i \times w_i) = M$

**解空间：**所有可能的不同元组，总共有$2^n$个元组

# 解空间的组织

回溯法将通过系统地检索给定问题的解空间来求解，这需要有效地组织问题的解空间——把元组表示成为结构化的组织形式。

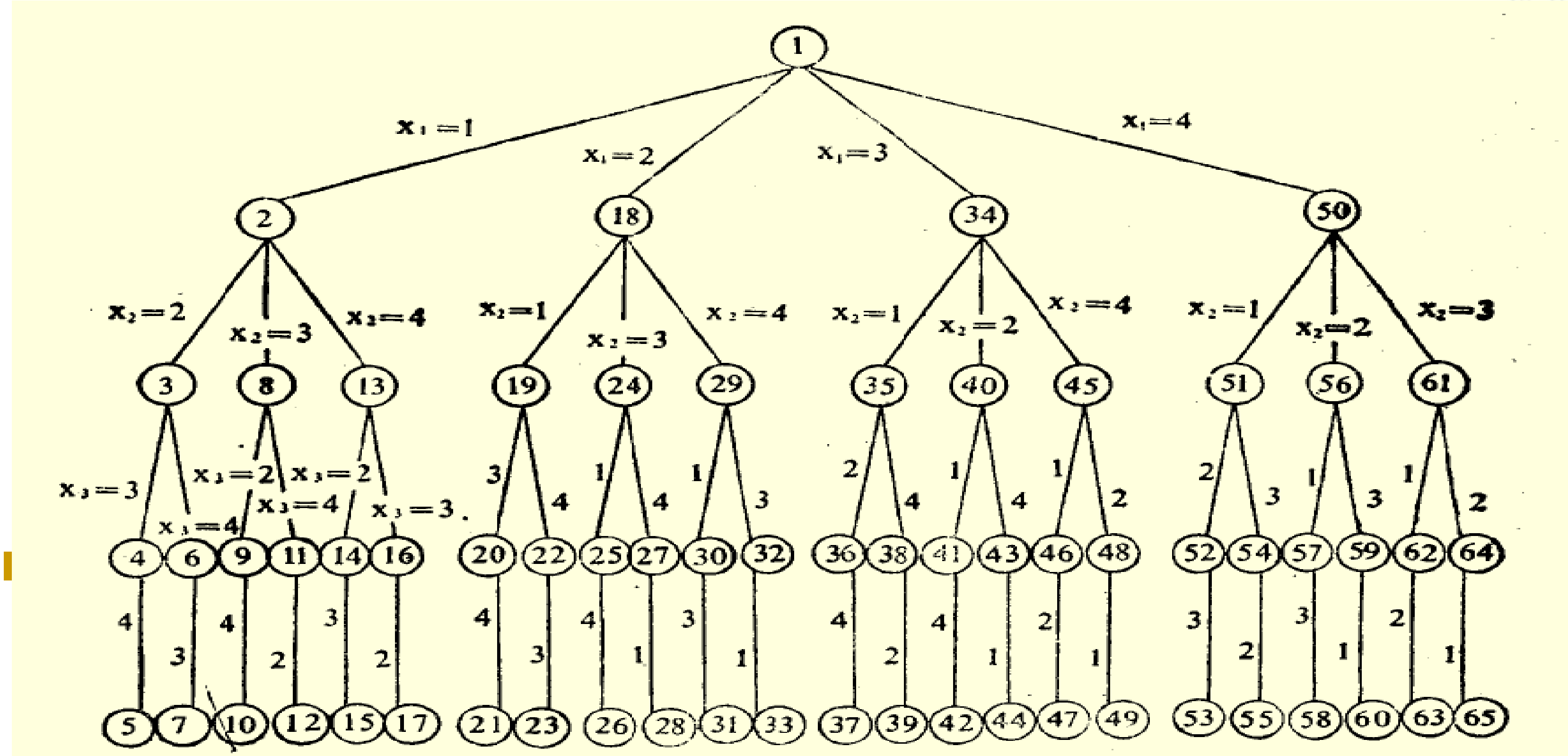
**采用何种形式组织问题的解空间？**

可以用树结构组织解空间——状态空间树。

例8.3 n-皇后问题。8皇后问题的推广，即在n×n的棋盘上放置n个皇后，使得它们不会相互攻击。

**解空间**：排列问题，解空间由n!个n-元组组成.

**实例**：4皇后问题的解空间树结构如下所示：




**边**：从i级到i+1级的边用$x_i$的值标记，表示将皇后i放到第i行的第$x_i$列。

如由1级到2级结点的边给出$x_1$的各种取值：1、2、3、4。

**解空间**：由从根结点到叶结点的所有路径所定义。

共有4！＝24个叶结点，反映了4元组的所有可能排列

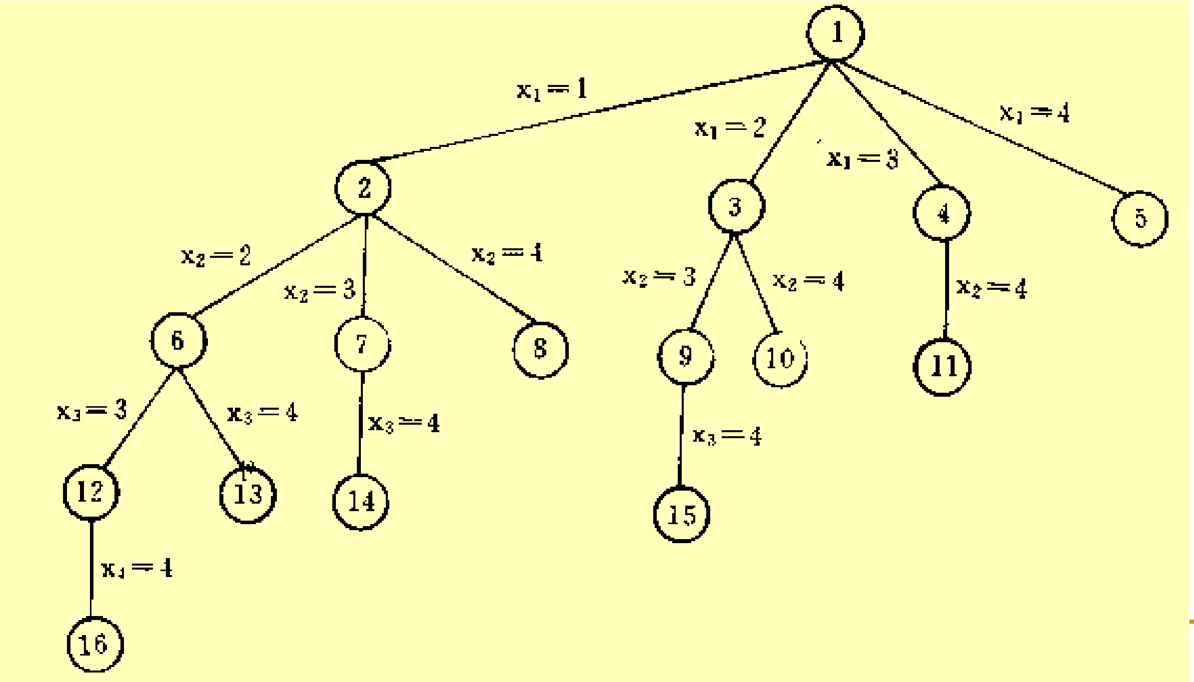
——称为排列树。

## 例8.4 子集和数问题的解空间的树结构

两种元组表示形式：

1）**元组大小可变**（$x_i < x_{i+1}$）

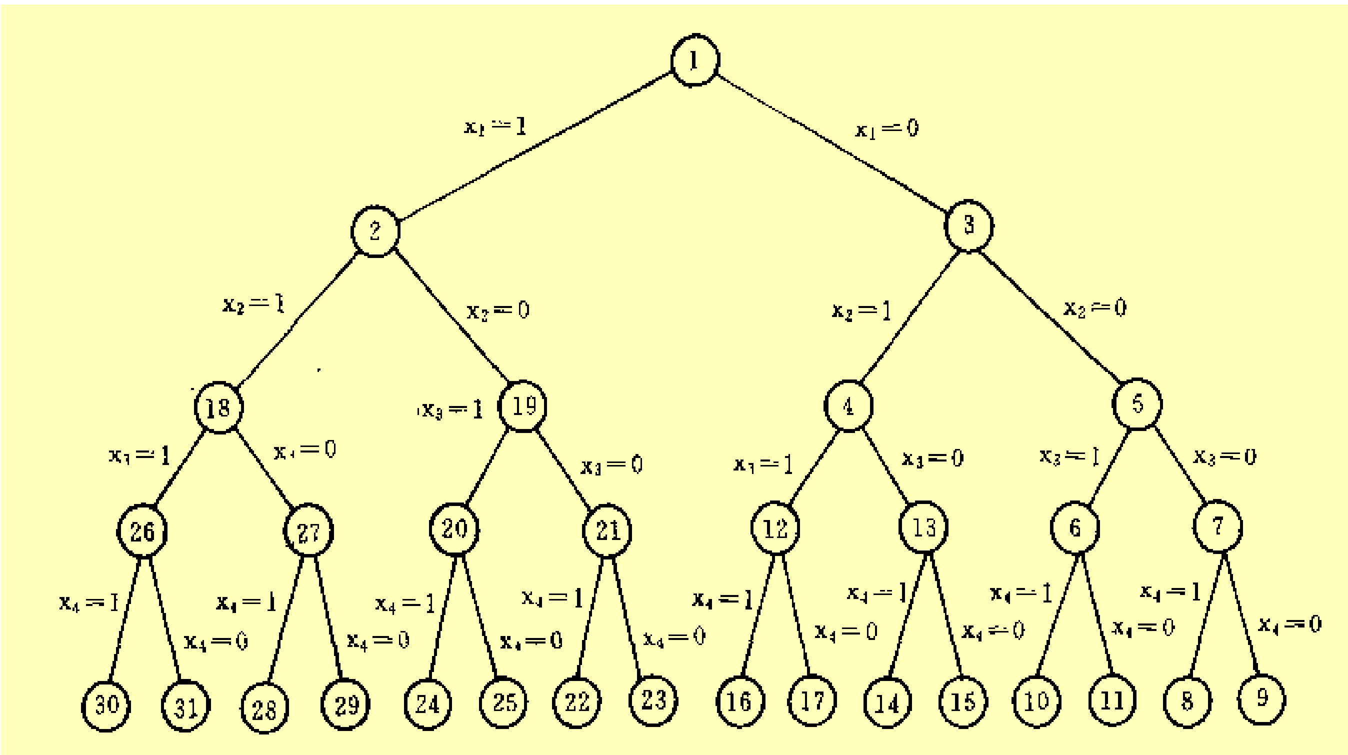
**树边标记**：由i级结点到i+1级结点的一条边用$x_i$来表示，表示k-元组里的第i个元素是已知集合中下标为$x_i$的元素。




解空间由树中的根结点到任何结点的所有路径所确定：(1), (1,2), (1,2,3),(1,2,3,4),(1,2,4), (1,3,4), (1,4),(2),(2,3)等。共有16个可能的元组（结点1代表空集）。

## 2）元组大小固定：每个都是n-元组

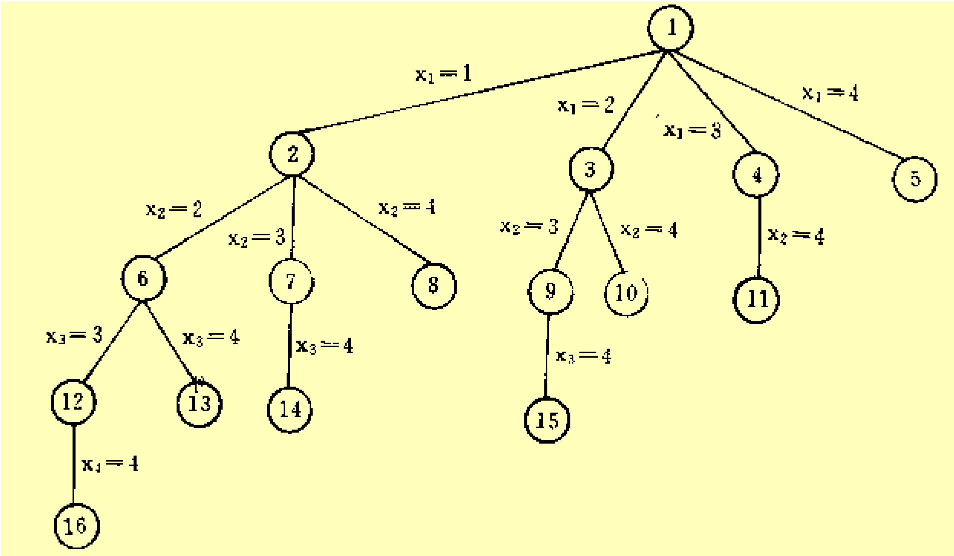
**树边标记**：由i级结点到i＋1级结点的那些边用$x_i$的值来标记，$x_i$＝1或0。

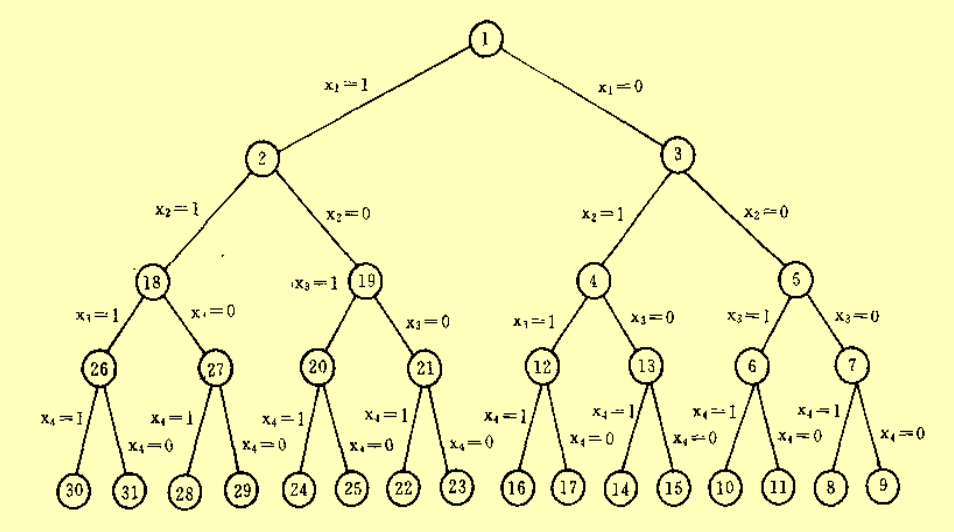



解空间由根到叶结点的所有路径确定。共有16个可能的元组。

共有$2^4$＝16个叶子结点，代表所有可能的4元组。

# 同一个问题可以有不同形式的状态空间树。





# 关于状态空间树的概念

- 状态空间树：解空间的树结构称为状态空间树(state space tree)
- 问题状态：树中的每一个结点代表问题的一个状态，称为问题状态 (problem state) 。
- 状态空间：由根结点到其他结点的所有路径确定了这个问题的状态空间(state space) 。
- 解状态：是这样一些问题状态S，对于这些问题状态，由根到S的那条路径确定了这个问题**解空间中的一个元组**(solution states) 。
- 答案状态：是这样的一些解状态S，对于这些解状态而言，由根到S的这条路径确定了这**问题的一个解**（满足隐式约束条件的解）(answer states) 。

# 状态空间树的构造：

以问题的初始状态作为**根结点**，然后系统地生成其它问题状态的结点。

在状态空间树生成的过程中，结点根据被检测情况分为三类：

- **活结点**：自己已经生成,但其儿子结点还没有全部生成并且有待生成的结点。
- **E-结点**（正在扩展的结点）：当前正在生成其儿子结点的活结点。（ expansion node ）
- **死结点**：不需要再进一步扩展或者其儿子结点已全部生成的结点。

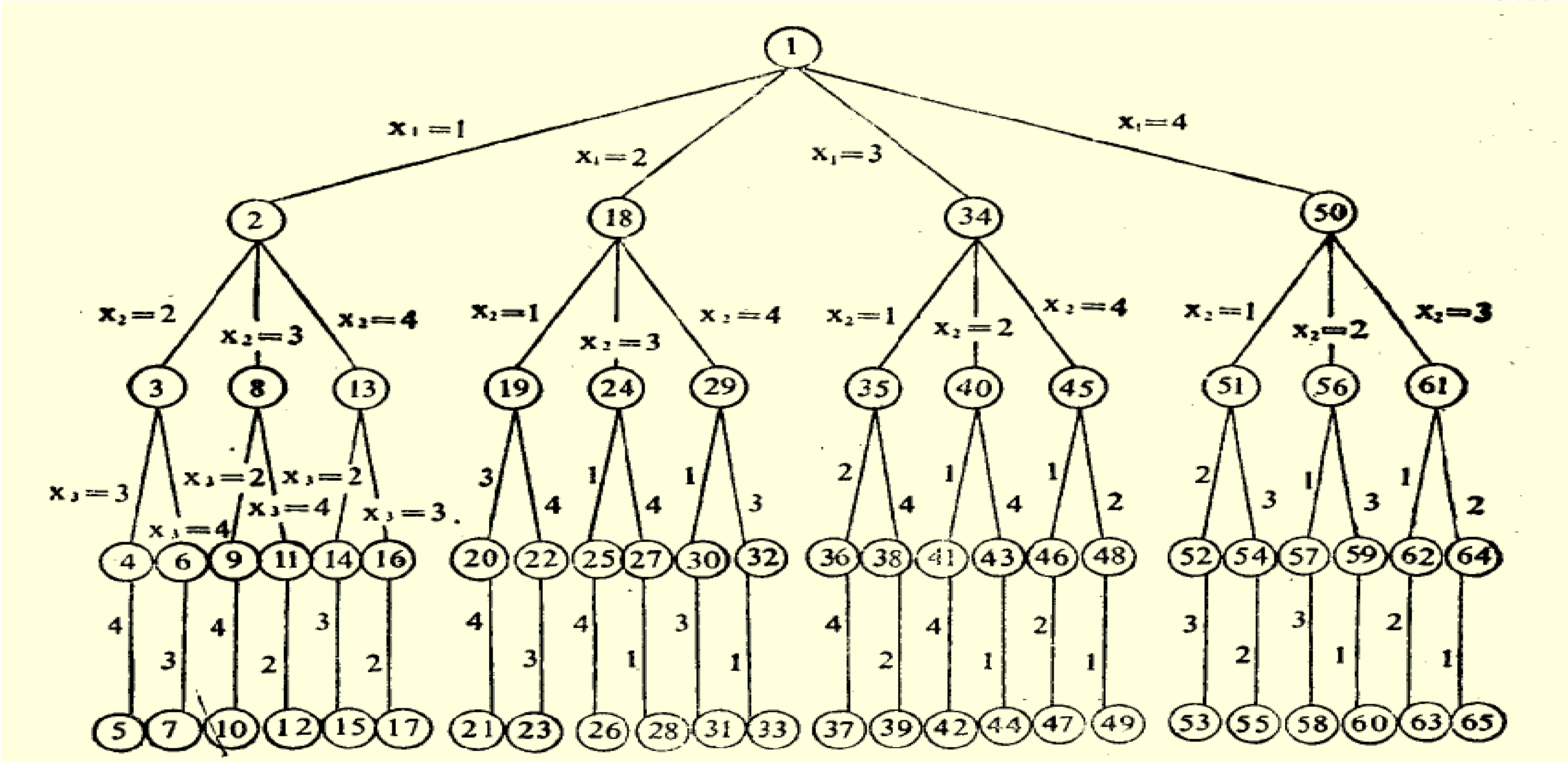
# 构造状态空间树的两种策略

1. 深度优先策略：当E-结点R一旦生成一个新的儿子C时，C就变成一个新的E-结点，当完全检测了子树C之后，R结点再次成为E-结点。

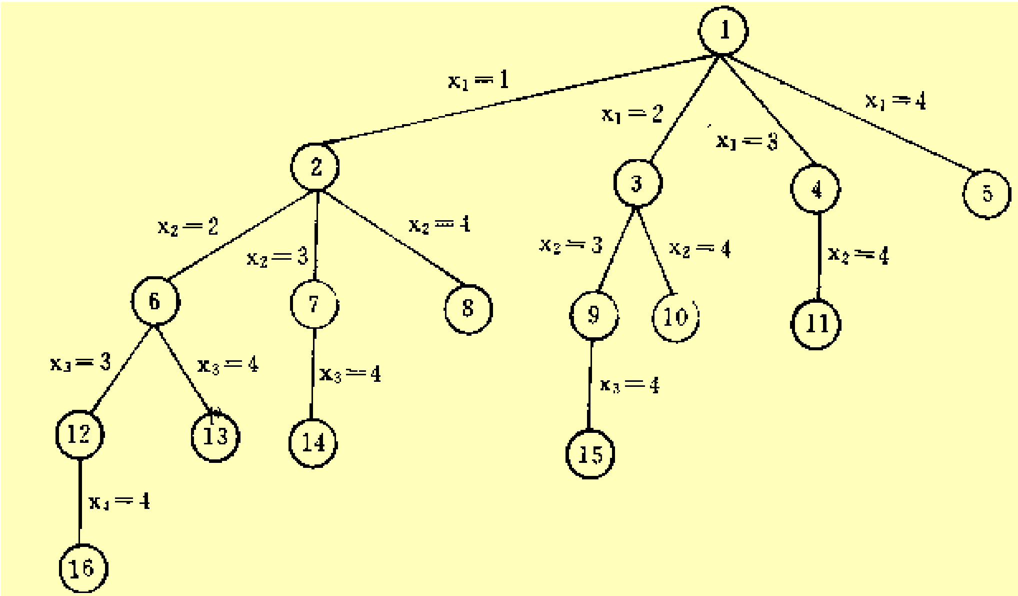
2. 宽度优先策略：一个E-结点一直保持到变成死结点为止。

限界函数：在结点生成的过程中，定义一个限界函数，用来杀死还没有全部生成儿子结点的一些活结点——这些活结点已无法满足限界函数的条件，不可能导致问题的答案。

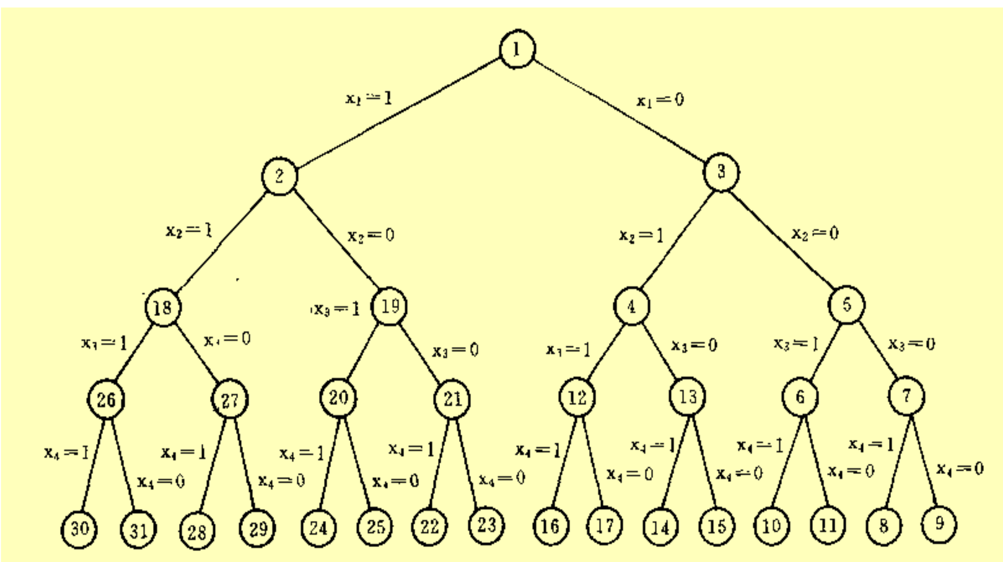
- 回溯法：使用限界函数的**深度优先**结点生成方法称为回溯法（backtracking）
- 分支-限界方法：E结点一直保持到死为止的状态生成方法称为分支-限界方法（branch-and-bound）




- 深度优先策略下的结点生成次序（结点编号）




- 利用**队列**的宽度优先策略下的结点生成次序(BFS)

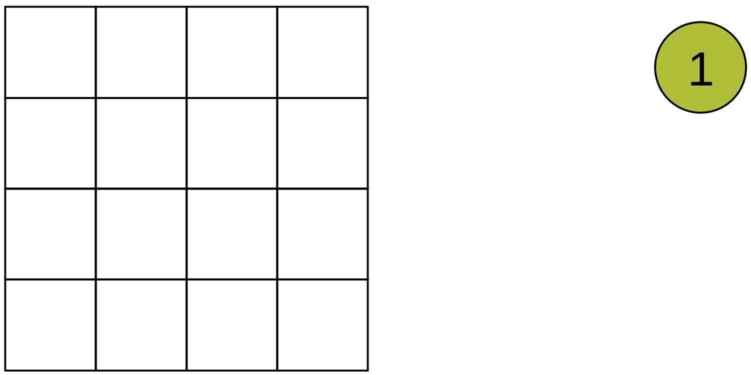



- 利用**栈**的宽度优先策略下的结点生成次序（D-Search）

# 例：4-皇后问题的回溯法求解

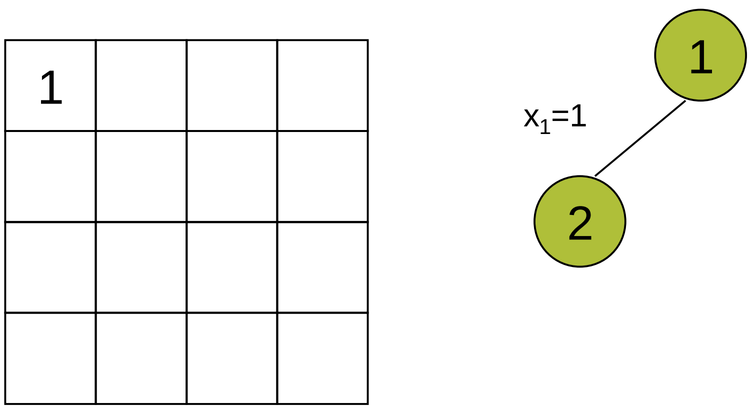
- **限界函数**：如果（$x_1,x_2,...,x_{i-1}$）是到当前E结点的路径，那么$x_{i-1}$的儿子结点$x_i$是一些这样的结点，它们使得（$x_1,x_2,...,x_{i-1},x_i$）表示没有两个皇后处在相互攻击状态的一种棋盘格局。
- **开始状态**：根结点1，此时棋盘为空，还没有放置任何皇后。
- **结点的生成**：依次考察皇后1——皇后n的位置。

## 按照自然数递增的次序生成4皇后问题状态空间树中结点的儿子结点。




根结点1，开始状态，唯一的活结点

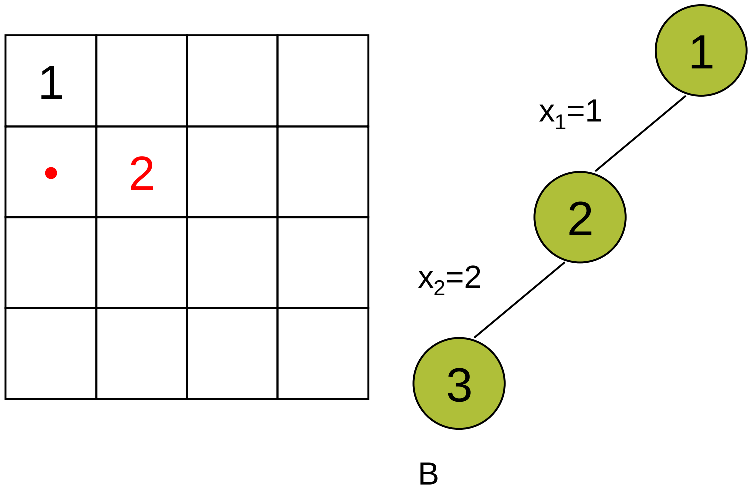
解向量：（ ）




生成结点2，表示皇后1被放到第1行的第1列上，该结点是从根结点开始第一个被生成结点。

解向量：（1）

结点2变成新的E结点，下一步扩展结点2

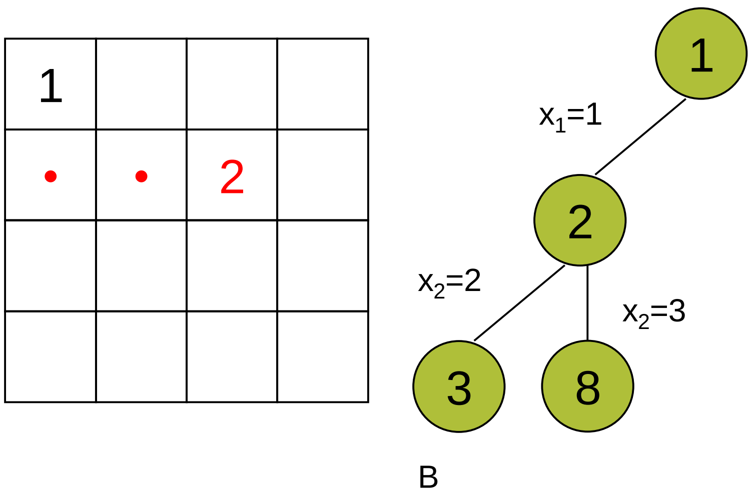



由结点2生成结点3，即皇后2放到第2行第2列。

利用限界函数杀死结点3。

返回结点2继续扩展。

（结点4，5，6，7不会生成）

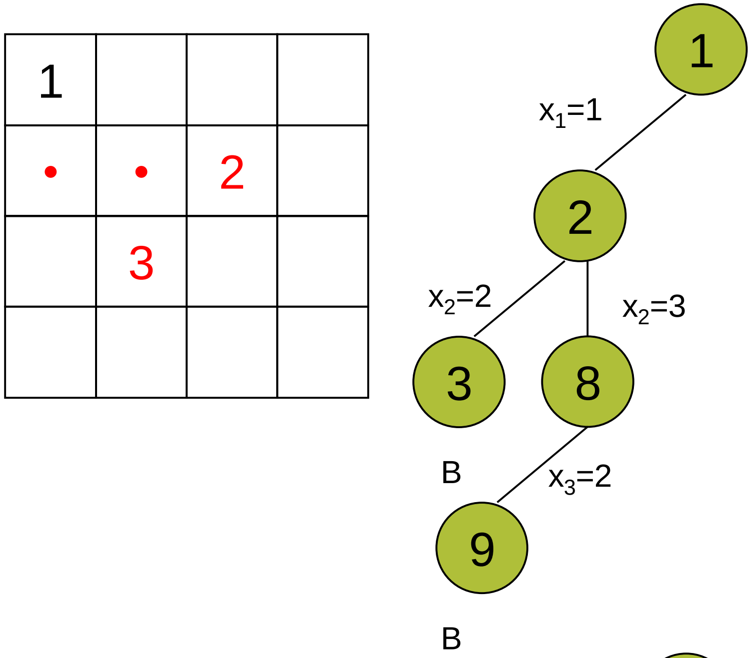



由结点2生成结点8，即皇后2放到第2行第3列。

结点8变成新的E结点。

解向量：（1，3）

从结点8继续扩展。

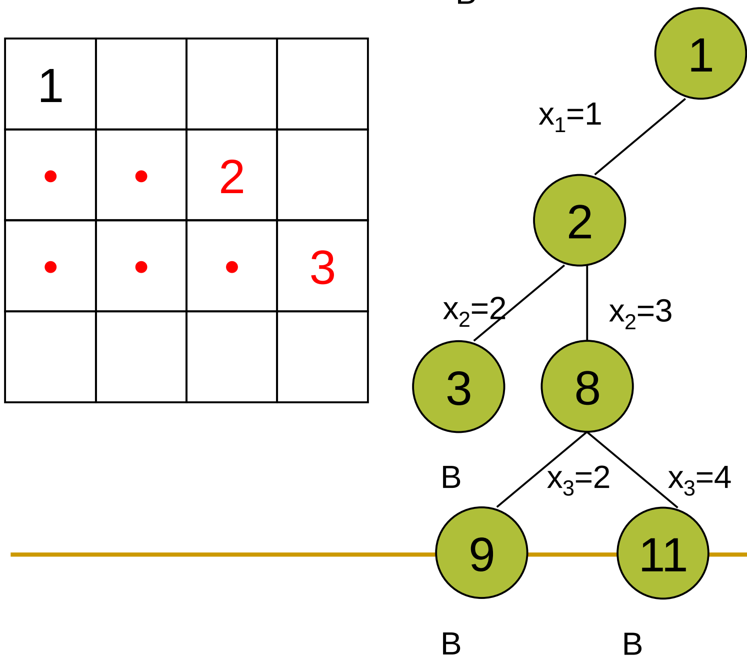



由结点8生成结点9，即皇后3放到第3行第2列。

利用限界函数杀死结点9。

返回结点8继续扩展。

（结点10不会生成）




由结点8生成结点11，即皇后3放到第3行第4列。

利用限界函数杀死结点11。

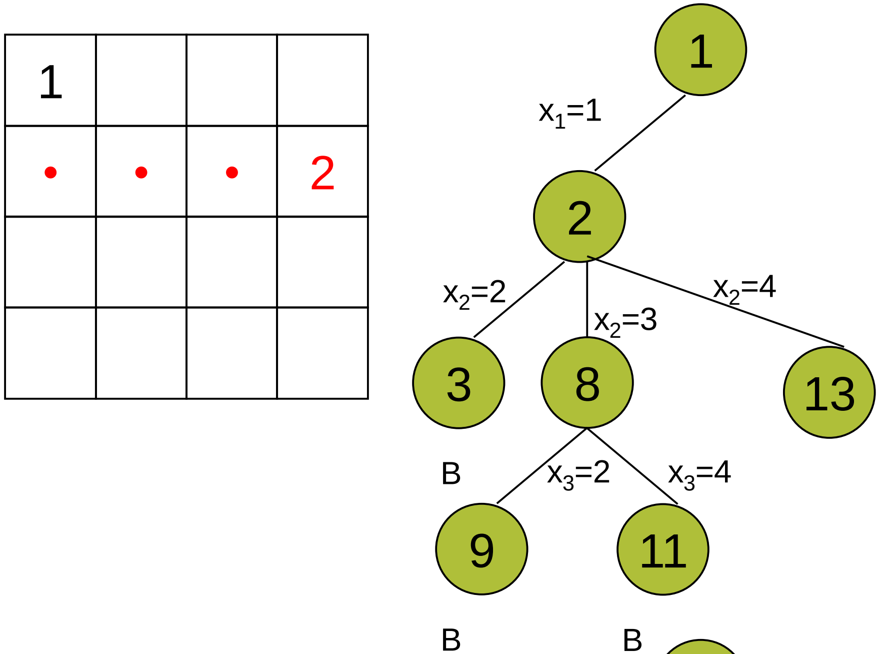
返回结点8继续。

（结点12不会生成）

结点8的所有儿子已经生成，但没有导出答案结点，变成死结点。

结点8被杀死。

返回结点2继续扩展。

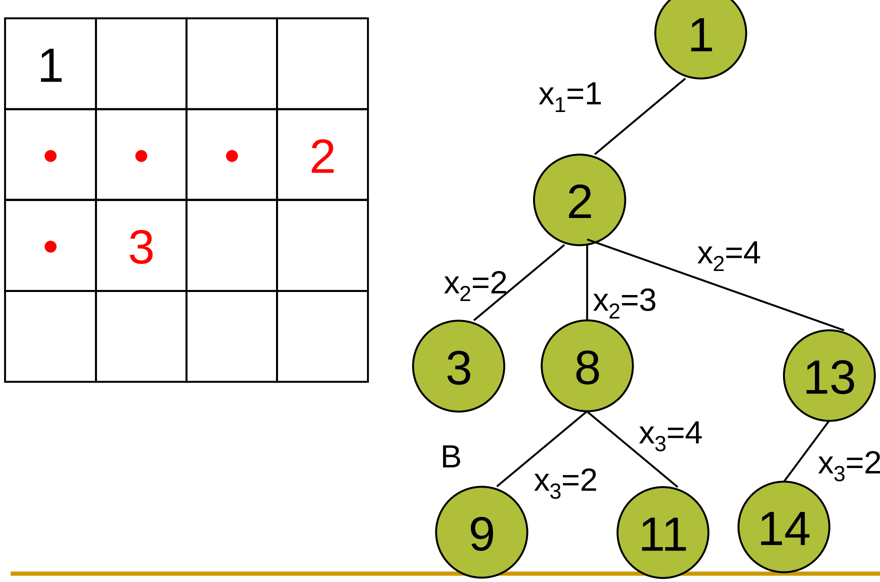



由结点2生成结点13，即皇后2放到第2行第4列。

结点13变成新的E结点。

解向量：（1，4）

从结点13继续扩展。



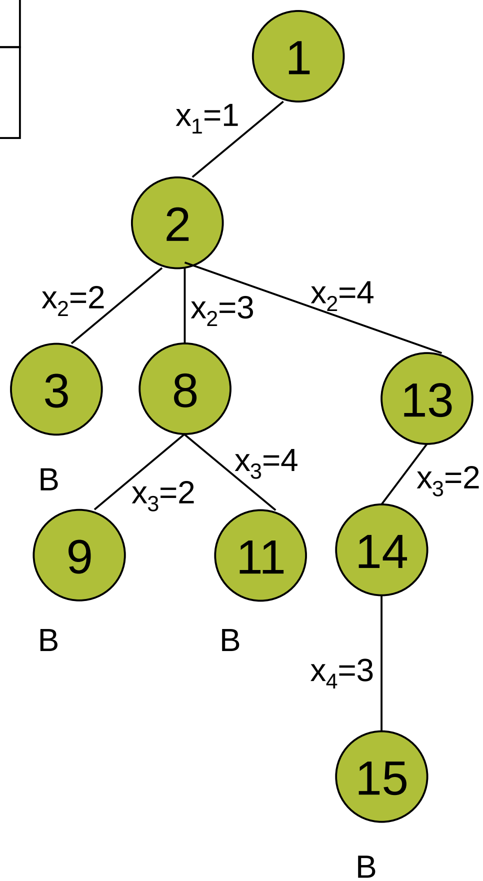

由结点13生成结点14，即皇后3放到第3行第2列。

结点14变成新的E结点。

解向量：（1，4，2）

从结点14继续扩展。

| | | | |
|---|---|---|---|
| 1 | | | |
| • | • | • | 2 |
| • | 3 | | |
| • | • | 4 | |

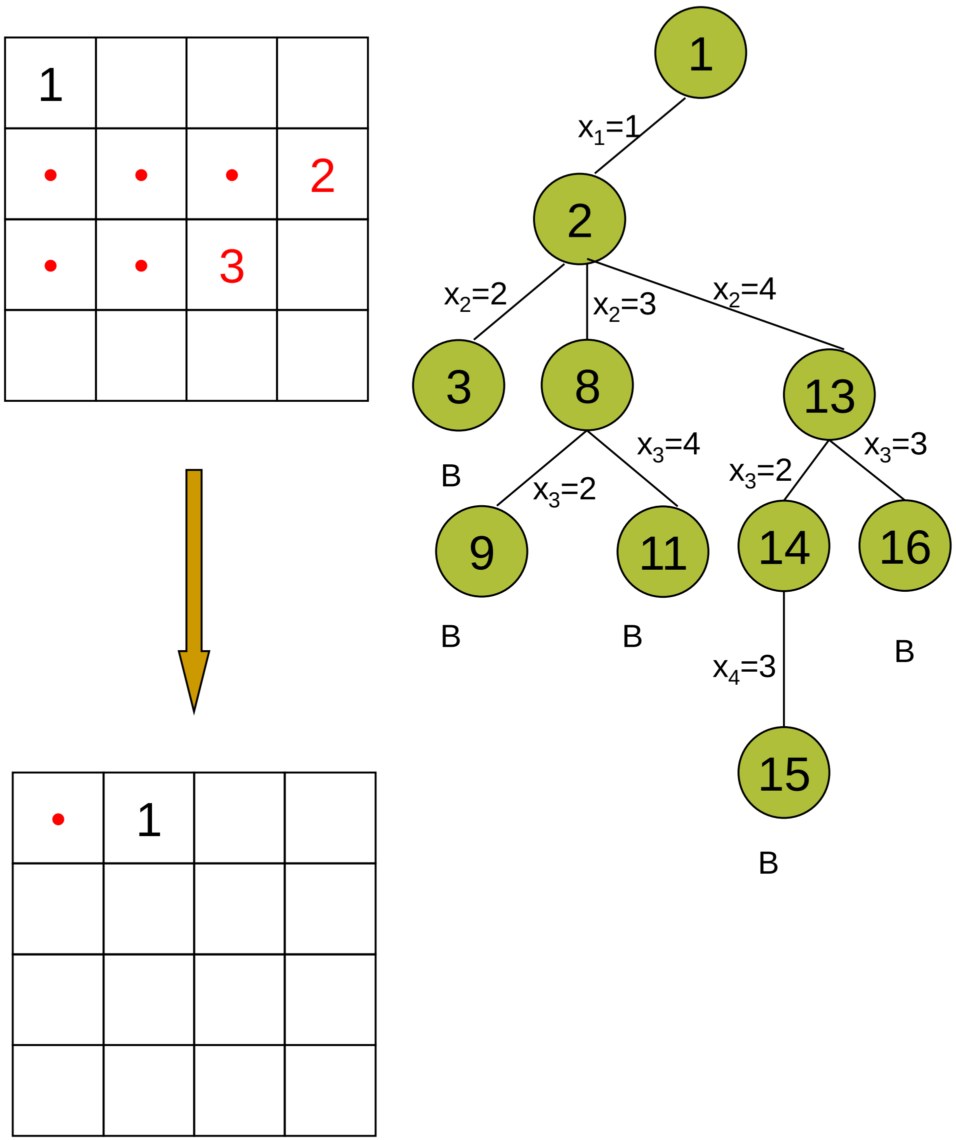



由结点14生成结点15，即皇后4放到第4行第3列。

利用限界函数杀死结点15。

返回结点14，结点14不能导致答案结点，变成死结点，被杀死。

返回结点13继续扩展。




由结点13生成结点16，即皇后3放到第3行第3列。

利用限界函数杀死结点16。

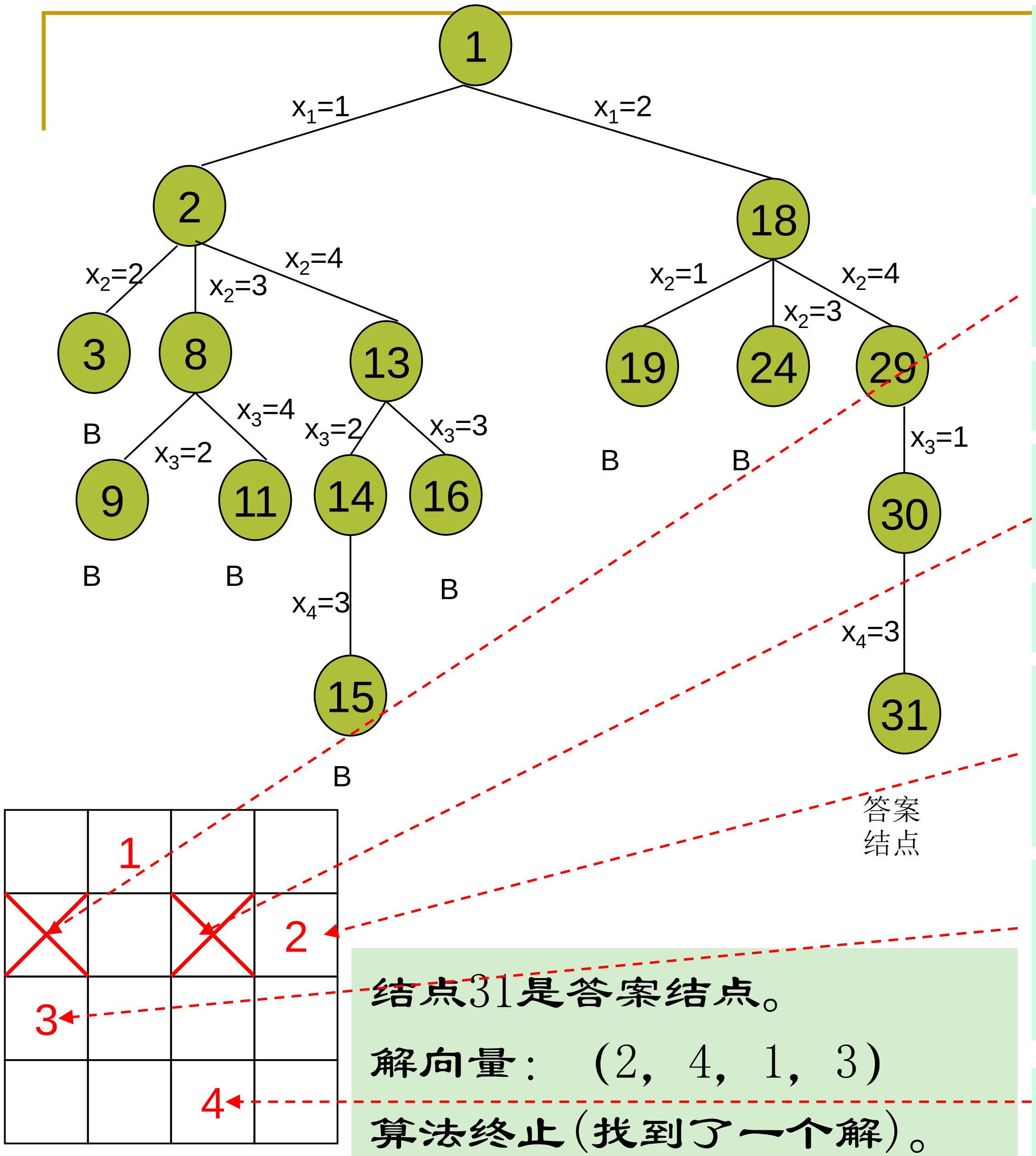
返回结点13，结点13不能导致答案结点，变成死结点，被杀死。

返回结点2继续扩展。

结点2不能导致答案结点，变成死结点，被杀死。

返回结点1继续扩展。

由结点1生成结点18，即皇后1放到第1行第2列。




由结点1生成结点18，即皇后1放到第1行第2列。结点18变成E结点。

扩展结点18生成结点19，即皇后2放到第2行第1列。

利用限界函数杀死结点19。

返回结点18，生成结点24，即皇后2放到第2行第3列。

利用限界函数杀死结点24。

返回结点18，生成结点29，即皇后2放到第2行第4列。结点29变成E结点。

扩展结点29生成结点30，即皇后3放到第3行第1列。结点30变成E结点。

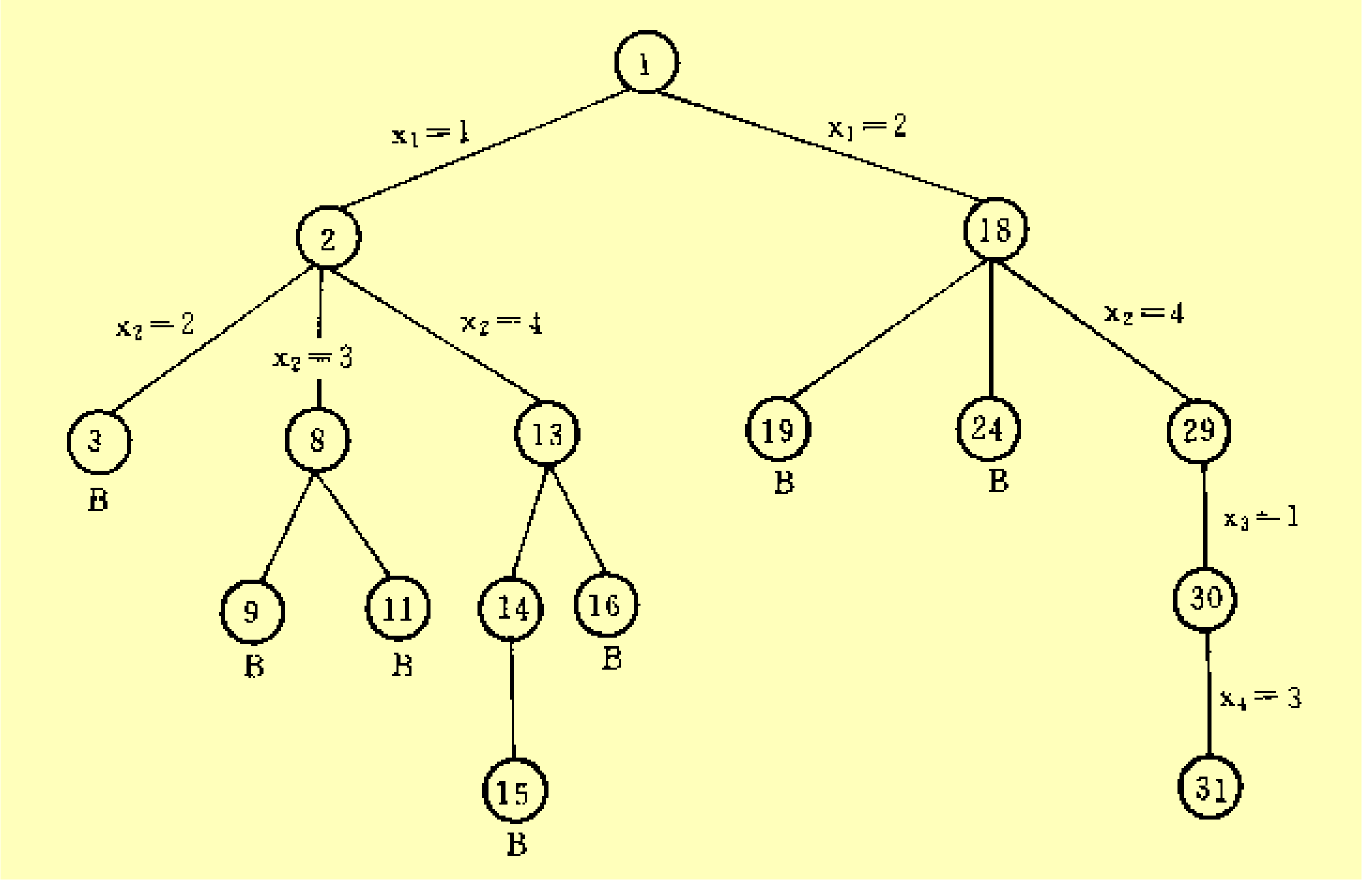
扩展结点30生成结点31，即皇后4放到第4行第3列。

结点31是答案结点。

解向量：（2，4，1，3）

算法终止（找到了一个解）。

# 回溯法求解4-皇后问题所生成的树



# 回溯算法的描述

- 设$(x_1, x_2, \ldots x_{i-1})$是由根到结点$x_{i-1}$的路径。
- $T(x_1, x_2, \ldots x_{i-1})$是下述所有结点$x_i$的集合，它使得对于每一个$x_i$，$(x_1, x_2, \ldots x_{i-1}, x_i)$是由根到结点$x_i$的路径。
- **限界函数$B_i$**：如果路径$(x_1, x_2, \ldots x_i)$不可能延伸到一个答案结点，则$B_i(x_1, x_2, \ldots x_i)$取假值，否则取真值。
- 解向量$X(1:n)$中的每个$x_i$即是选自集合 $T(x_1, x_2, \ldots x_{i-1})$且使$B_i$为真的$x_i$。

# 回溯法的基本设计思想

- 第一步：为问题定义一个状态空间，这个空间必须至少包含问题的一个解
- 第二步：组织状态空间以便它能被容易地搜索。
  **典型的组织方法是图或树。**
- 第三步：按深度优先的方法从开始结点进行搜索
  - 开始结点是第一个活结点（也是 E-结点：expansion node）
  - 如果能从当前的E-结点移动到一个新结点，那么这个新结点将变成一个活结点和新的E-结点，旧的E-结点仍是一个活结点。
  - 如果不能移到一个新结点，当前的E-结点就“死”了（不再是一个活结点），那么便只能**返回**到最近被考察的活结点（回溯），这个活结点变成了当前的E-结点。
  - 当我们已经找到了答案或者回溯尽了所有的活结点时，搜索过程结束。

# 回溯法的一般框架

```
procedure BACKTRACK(n)
    integer k, n; local X(1:n)
    k←1
    while k>0 do
        if 还剩有没检验过的X(k)使得
            X(k) ∈T(X(1),…X(k-1)) and  B(X(1),…X(k))=true
      then
          if(X(1),…,X(k)) 是一条已抵达一答案结点的路径
          then print(X(1),…,X(k)) endif
          k ←k+1  //考虑下一个集合//
      else
          k ←k-1   //回溯到先前的集合//
      endif
    repeat
end BACKTRACK
```

- 回溯方法的抽象描述。该算法求出所有答案结点。
- 在X(1),…,X(k-1)已经被选定的情况下，T(X(1),…,X(k-1))给出X(k)的所有可能的取值。限界函数B(X(1),…,X(k))判断哪些元素满足隐式约束条件。

# 回溯算法的递归表示

- 回溯方法的递归程序描述。
- 调用：RBACKTRACK(1) 。
- 进入算法时，解向量的前k-1个分量X(1),…,X(k-1)已赋值。

```
procedure RBACKTRACK(k)
    global n, X(1:n)

    for 满足下式的每个X(k)
        X(k) ∈T(X(1),…X(k-1)) and  B(X(1),…X(k))=true do

        if(X(1),…,X(k)) 是一条已抵达一答案结点的路径
        then   print(X(1),…,X(k))
        endif

        call RBACKTRACK(k+1)

    repeat
end RBACKTRACK
```

说明：当k>n时， T(X(1),…X(k-1))返回一个空集，算法不再进入for循环。算法印出所有的解，元组大小可变。

# 8.2 n-皇后问题

- n元组：（$x_1,x_2,...,x_n$）
- 怎么判断是否形成了互相攻击的格局?
  - 不在同一行上：约定不同的皇后在不同的行
  - 不在同一列上：$x_i \neq x_j$，（$i, j \in [1:n]$）
  - 不在同一条斜角线上：如何判定？

1）在同一斜角线上的由左上方到右下方的每一个元素有相同的“**行－列**”值。

| i\ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | O | | |
| 2 | | | O | |
| 3 | | | | O |
| 4 | | | | |

左上方——右下方相同的“行－列”值

1－2＝2－3＝3－4

2）在同一斜角线上的由右上方到左下方的每一个元素有相同的“**行＋列**”值。

| i\ j | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 1 | | | O | |
| 2 | | O | | |
| 3 | O | | | |
| 4 | | | | |

右上方——左下方相同的“行＋列”值

1＋3＝2＋2＝3＋1

**判别条件**：假设两个皇后被放置在(i, j)和(k, l)位置上，

则仅当：　　$i-j=k-l$ 或 $i+j=k+l$

时，它们在同一条斜角线上。

**即**：$j-l = i-k$ 或 $j-l = k-i$

**亦即**：当且仅当 **$|j-l| = |i-k|$** 时，两个皇后在同一斜角线上。

过程**PLACE(k)**根据以上判别条件，判定**皇后k**是否可以放置在当前位置X(k)处——满足下述条件即可：

- 不等于前面的X(1)，…， X(k-1)的值，且
- 不能与前面的k-1个皇后在同一斜角线上。

# Place算法

```
procedure PLACE(k)
  //如果皇后k可以放在第k行第X(k)列，则返回true，否则返回false//
    global X(1:k); integer i,k
    i← 1
    while i < k do
        if   X(i)=X(k)  //在同一列上//
            or ABS(X(i)-X(k))=ABS(i-k)   //在同一斜角线上//
            then return(false)
        endif
        i← i+1
    repeat
    return(true)
end PLACE
```

# NQUEENS算法

```
procedure NQUEENS(n)
//在n×n棋盘上放置n个皇后，使其不能相互攻击。算法求出所有可能的位置//
 integer k,n, X(1:n);
   X(1)← 0; k←1                                  //k是当前行，X(k)是当前列//
   while k>0 do                                   //对所有的行执行以下语句//
       X(k)←X(k)+1                                //移到下一列//
       while X(k) ≤ n and not PLACE(k) do         //检查是否能放置皇后//
            X(k)←X(k)+1                           //当前X(k)列不能放置，后推一列//
       repeat
       if X(k)≤n                                  //找到一个位置//
          then   if  k=n                          //是一个完整的解吗？//
                   then print(X)                  //是，打印解向量//
                   else k←k+1; X(k)←0            //否，转下一皇后//
                 endif
       else
          k←k-1
       endif
   repeat
end NQUEENS
```

# 8.3 子集和数问题

- 元组大小固定：n元组（$x_1,x_2,\ldots,x_n$），$x_i$＝1或0
- 结点：对于i级上的一个结点，其左儿子对应于$x_i$=1，右儿子对应于$x_i$＝0。
- 限界函数的选择

约　定：**W(i)按非降次序排列**

条件一：$$\sum_{i=1}^{k} W(i)X(i)+\sum_{i=k+1}^{n} W(i)\geq M$$

条件二：$$\sum_{i=1}^{k} W(i)X(i)+W(k+1)\leq M$$

仅当满足上述两个条件时，限界函数B(X(1),…X(k))=true

注：如果不满足上述条件，则X(1),…X(k)根本不可能导致一个答案结点。

# 子集和数的递归回溯算法

//W(i)按非降次序排列，

$$s=\sum_{i=1}^{k-1}W(i)X(i), r=\sum_{i=k}^{n}W(i)$$

W(1)≤M, $\sum_{i=1}^{n}W(i)\geq M$ //

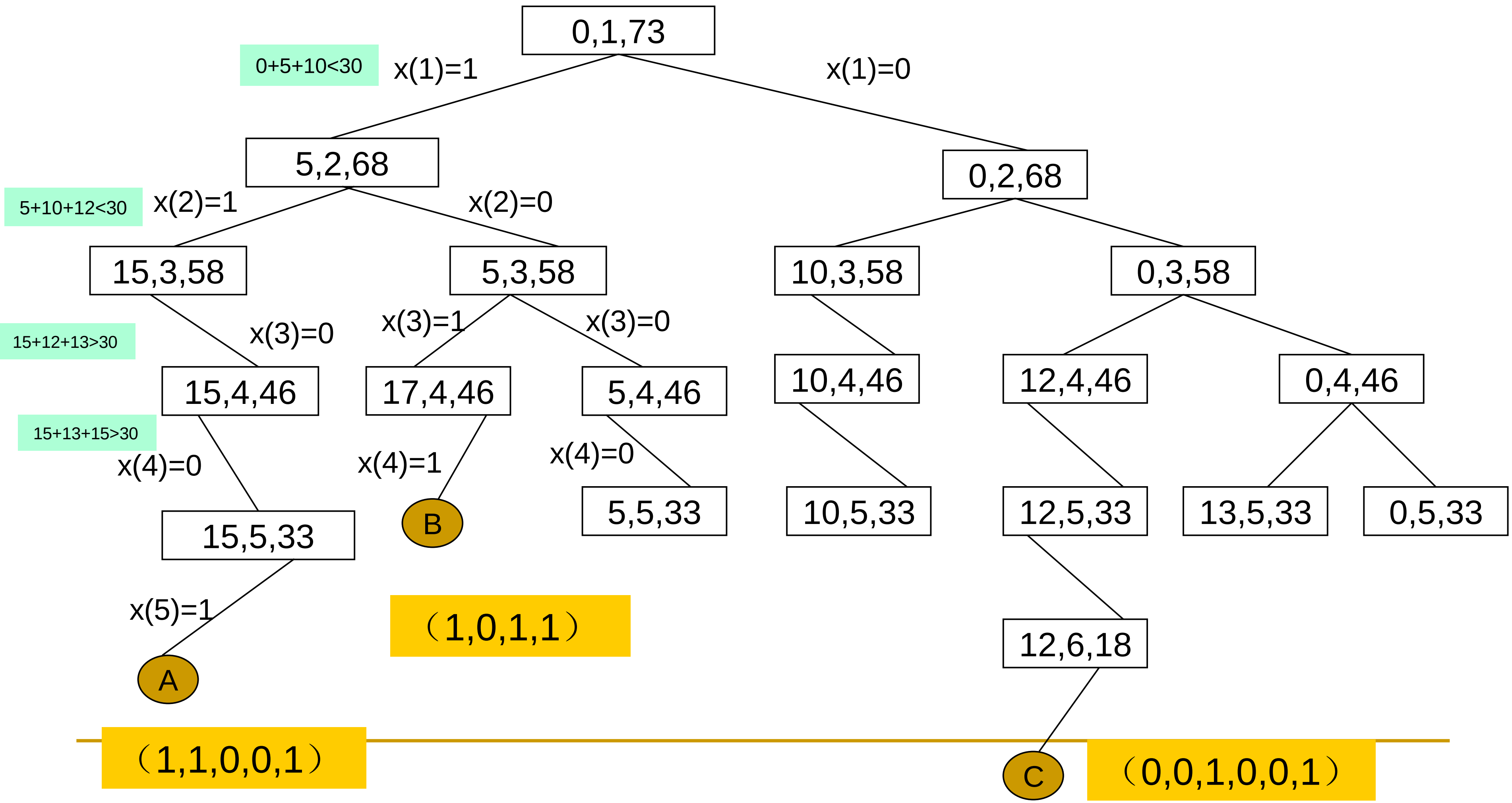
```
procedure SUMOFSUB(s,k,r)
    global integer M,n; global real W(1:n);
    global boolean X(1:n) , real r,s; integer k,j
    X(k)←1                          //生成左儿子，B_{k-1}=true,s+W(k)≤M//
    if s+W(k)=M then                //找到答案//
            print(X(j),j←1 to k)    //输出答案//
    else   if s+W(k)+W(k+1)<=M then   //确保Bk＝true//
              call SUMOFSUB(s+W(k),k+1,r-W(k))
          endif
    endif
    //生成右儿子，计算B_k的值//
    if  s+r-W(k)>=M and s+W(k+1)<=M    //确保B_k＝true//
         then   X(k)←0
                call SUMOFSUB(s,k+1,r-W(k))
    endif
end SUMOFSUB
```

向前看两步,可以的话才进行下一步处理

首次调用SUMOFSUB(0,1, $\sum_{i=1}^{n}W(i)$)

# SUMOFSUB的一个实例

- n=6, M=30, W(1:6)=(5,10,12,13,15,18)
- 方形结点：s，k，r，圆形结点：输出答案的结点，共生成20个结点



- 作业：

（1）分派问题一般陈述如下：给n个人分派n件工作，把工作j分配给第i个人的成本为COST(i,j)。设计一个回溯算法，在给每个人分派一件不同工作的情况下使得总成本最小。

（2）设W=(5,7,10,12,15,18,20)和M=35，使用过程SUMOFSUB找出W中使得和数等于M的全部子集并画出所生成的部分状态空间树。