算法设计与分析

Computer Algorithm Design & Analysis

2024.3

王多强

QQ：1097412466

Chapter 22

Elementary Graph Search Algorithms

图的基本搜索算法

图论问题渗透到整个计算机科学，图算法对于计算机学科至关重要。很多计算问题可以归纳为图问题。本部分对图相关的一些比较重要的问题进行讨论。

22.1 图的表示

一个图 G 由结点和边组成，记为 $G = (V, E)$，其中

- V：G 的结点集合，$|V|$ 表示结点数。
- E：G 的边集合，$|E|$ 表示边数。E 中的每条边用一对结点的**序偶**表示，记为 (u, v)，$u, v \in V$。

算法中，用 $G.V$ 表示图 G 的结点集，$G.E$ 表示图 G 的边集。并通常用 $|V|$ 和 $|E|$ 两个参数表示算法的**输入规模**，而在渐近记号中，通常用 V 代表 $|V|$、用 E 代表 $|E|$，如 $O(VE)$。

1、图的表示：邻接表和邻接矩阵

图可以用**邻接表**和**邻接矩阵**两种方法表示。

(1) 邻接表

图 $G = (V, E)$ 的邻接表是一个包含 $|V|$ 条链表的数组，记为 $Adj[1..|V|]$：V 中的每个结点 u 在 Adj 中有一条单链表，其中包含所有与 u **有边相邻接**的结点，$Adj[u]$是该单链表的**表头结点**。所有结点的邻接单链表的表头结点组织在数组 Adj 中。

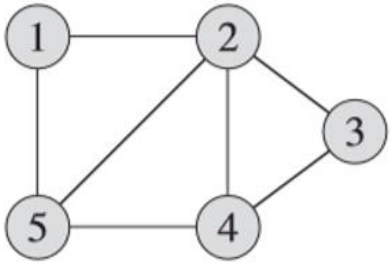
一般用 $G.Adj[u]$ 引用 G 的邻接表中结点 u 的邻接单链表。

邻接表可用于表示有向图也可用于表示无向图，存储空间需求均为 $O(V+E)$。

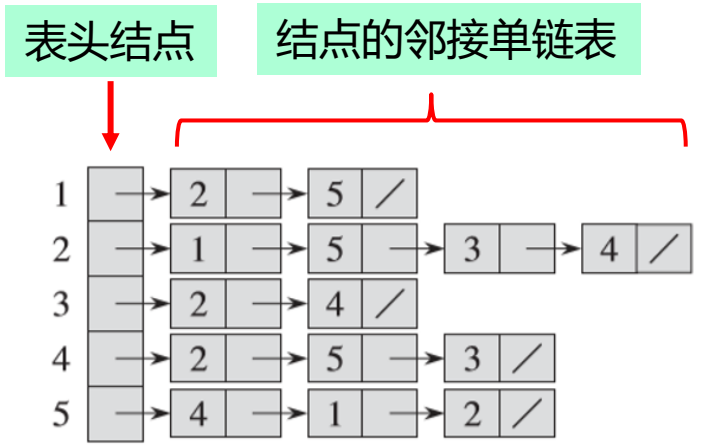
- 对于**有向图**，所有结点的邻接链表的长度之和等于 $|E|$；
- 对于**无向图**，所有结点的邻接链表的长度之和等于 $2|E|$。

例：

一个无向图的邻接表表示如下：

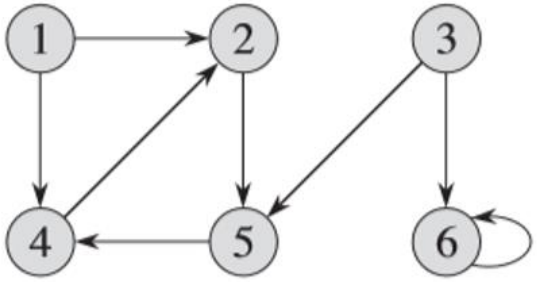


一个有**5**个结点和7条边的无向图 ***G***

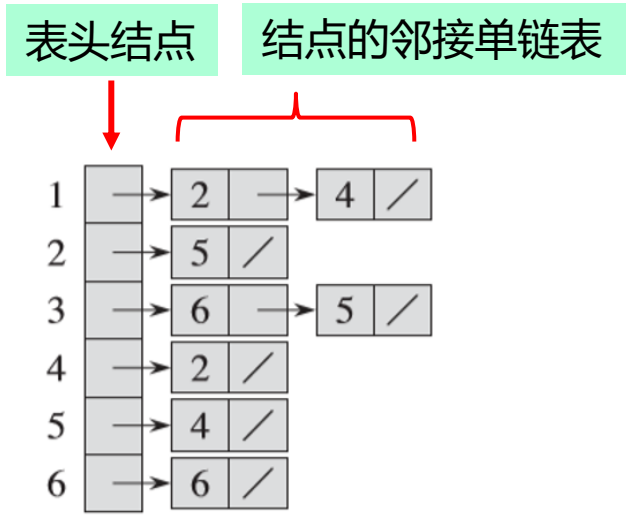


G 的邻接表表示，其中每条边被保存了两次

一个有向图的邻接表表示如下：



一个有**6**个结点和8条边的有向图 ***G***



G 的邻接表表示

（2）邻接矩阵

不失一般性，将图 G 中的结点编号为 $1, 2, \ldots, |V|$。图 G 的邻接矩阵是一个$|V|\times|V|$ 的方阵 $A = (a_{ij})_{1\le i,\, j\le |V|}$，并定义为：

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

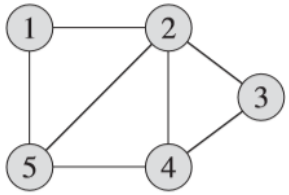
同样，邻接矩阵表可用于表示有向图也可用于表示无向图，其存储空间需求均为 $O(V^2)$。（方阵的大小）

无向图的邻接矩阵 A 是一个**对称矩阵**：A也是自己的**转置矩阵**，即 $A = A^T$。为了节省空间，无向图的邻接矩阵可以用上三角或下三角矩阵表示，可以省接近一半的空间。

有向图的邻接矩阵不是对称矩阵，以上性质不适用。

例

一个无向图的邻接矩阵表示如下：



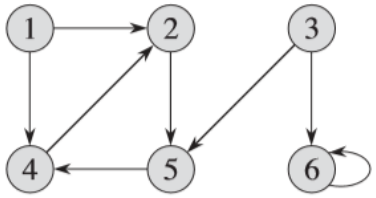
(a)

一个有5个结点和7条边的无向图 ***G***

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

G的邻接矩阵表示，对称矩阵

一个有向图的邻接矩阵表示如下：



(a)

一个有6个结点和8条边的有向图 ***G***

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

G 的邻接矩阵表示，非对称结构

邻接表和邻接矩阵的相关性质：

(1) **稀疏图**（边数 $|E|$ 远小于 $|V|^2$ 的图）一般用**邻接表**表示更省空间。

(2) **稠密图**（边数 $|E|$ 接近 $|V|^2$ 的图）一般用邻接矩阵表示更好。

(3) 邻接矩阵可用于需要**快速判断任意两个结点之间是否有边相连**的应用场景。如果用邻接表表示，为判断一条边 (u, v)是否是图中的边，需要**在邻接链表 $Adj[u]$ 中搜索**，效率较低。而用邻接矩阵表示时，直接判断元素 $A[u][v]$ 的取值即可。

(4) 使用邻接矩阵遍历图，一般会有不低于 $O(n^2)$ 的时间复杂度（每个元素都要访问到）；使用邻接表遍历图，一般会有不低于 $O(E)$ 的时间复杂度（每个结点的邻接链表都要访问到）。

2、权重图

图中的边带有**权重**的图称为**权重图**。

- 权重值通常以**权重函数** $W:E\rightarrow R$ 给出（一个边集到实数集的映射）。

(1) 用**邻接表**表示权重图

将边 $(u, v)\in E$ 的权重值 $W(u, v)$ 存放在 u 邻接链表的结点中。

(2) 用**邻接矩阵**表示权重图

- 若边 $(u, v)\in E$，则令邻接矩阵的元素 $A[u][v] = W(u, v)$。
- 若 (u, v) 不是 E 中的边，则令 $A[u][v] = \text{NIL}$，或 ∞、0。

22.2 图的基本检索和周游

- **基本概念：**

  被检测：在图中，如果一个结点的**所有邻接结点**都被访问了，则称该结点**被检测**了。

  检　索：访问图中的**某些结点**而不一定是全部结点的计算过程。

  周　游：访问图中**所有结点**的计算过程，也称为**遍历**。

- **经典的图检索和周游算法：**
  - **BFS**（宽度优先检索）、**BFT**（宽度优先周游）
  - **DFS**（深度优先检索）、**DFT**（深度优先周游）

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 然后访问邻接于 v 且目前尚未被访问的所有结点，这样结点 v 就**被检测**了，而 v 的这些邻接结点是**新的未被检测的结点**。将这些结点依次放到一个称为**未检测结点表**的**队列 Q** 中。

③ 若 Q **为空**，则算法终止；否则

④ 取 Q 的**表头结点**作为下一个**待检测**结点，重复 ② ~ ③。直到 Q **为空**，算法终止。

宽度优先检索算法

```
procedure BFS(v)
  //宽度优先检索图 G，v 是开始结点，VISITED(1:n)是一个标志数组，初始值为 VISITED(i) = 0, 1≤ i ≤n
  VISITED(v)←1   // 一个结点被访问了，就将该结点的visited值置1，表示被访问过了，
                 // v 是第一个被访问的结点，故首先置 VISITED(v)←1
  u←v   //u 指示当前正在被检测的结点
  将 Q 初始化为空   // Q 是保存未检测结点的队列//
  loop
      for 邻接于 u 的所有结点 w do
          if  VISITED(w) = 0 then     //只对未被访问过的结点处理//
              call ADDQ(w, Q)         //ADDQ 将 w 加入到队列 Q 的末端。队列：先进先出//
              VISITED(w)←1            //置 w 的访问标记为1，表示 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，然后开始对下一个
   repeat                                结点的检测//
end BFS
```

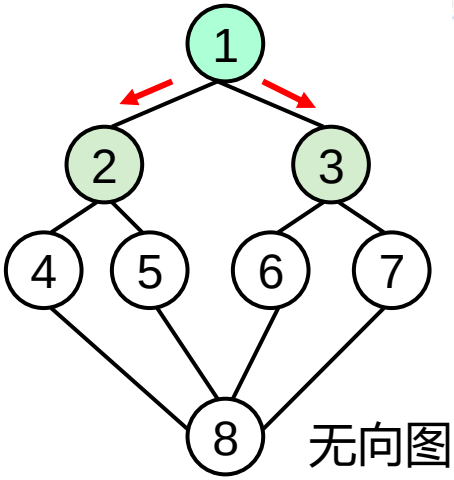
例:

检测结点1:

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

队列状态:

2	3	



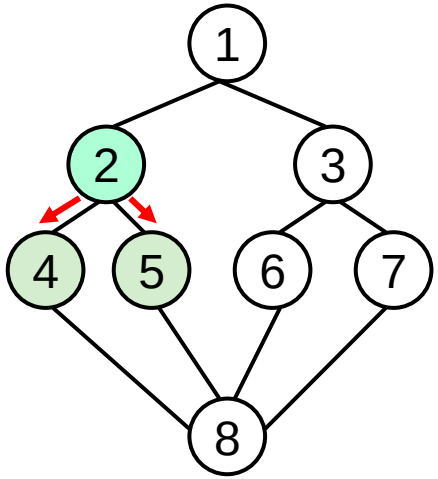
无向图 ***G***

检测结点2（结点2出队列）:

visited(4) =1、visited(5)=1

队列状态:

3	4	5	

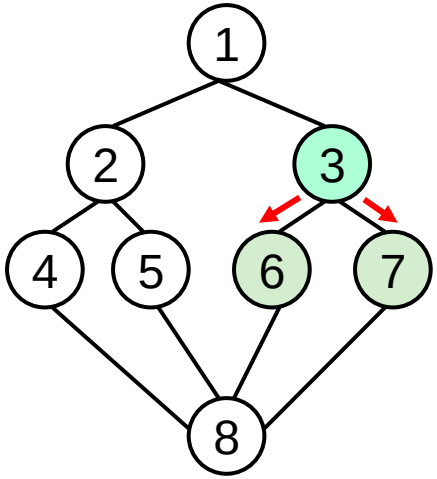


检测结点3（结点3出队列）:

visited(6) =1、visited(7)=1

队列状态:

4	5	6	7	

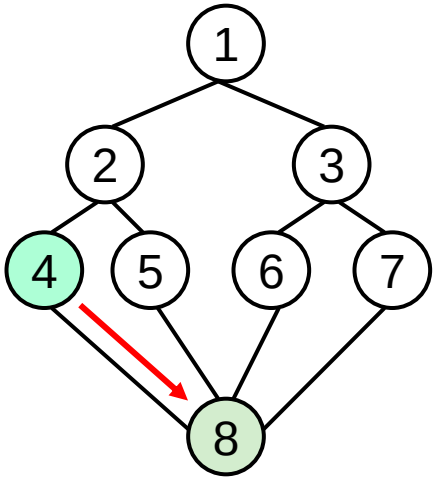


检测结点4（结点4出队列）：

visited(8) =1

队列状态：

5	6	7	8	



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	

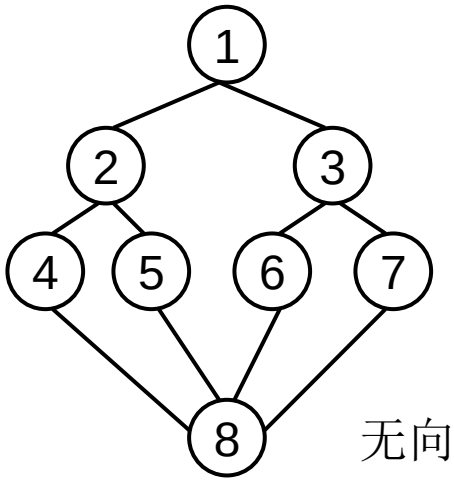
检测结点7（结点7出队列）：

队列状态：

8	

检测结点8（结点8出队列）：

队列状态：



无向图 ***G***

对于上图G，BFS 得到的结点访问序列：

1　2　3　4　5　6　7　8

定理1 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，$v \in V$。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 $(w \in V)$ 的**最短路径**的长度。

(1) 若 $d(v,w) \leq 1$，显然所有这样的 w 都将被访问（自己到自己或有一条边可以到达的结点）。

(2) 假设对所有 $d(v,w) \leq r$ 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（w 的前驱结点）。则有：$d(v,u)=r$。

根据归纳假设，u 可通过 BFS 访问到。

若 $u \neq v$ 且 $r \geq 1$，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 在这之前未被访问过，则此刻就将被访问。

证毕。

定理2 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，v 不会放到队列 Q 中，而其它结点 w 仅在 $\mathbf{VISITED}(w) = 0$ 时由 $\mathbf{ADDQ}(w, Q)$ 加入队列并置 $\mathbf{VISITED}(w) = 1$，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 是一个 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“检测 v 的那一刻”被放到队列中，故 Q 至少也应有 $\Omega(n)$ 的空间。

同时，VISITED(n) 本身需要 $\Theta(n)$ 的空间。

所以 $s(n, e) = \Theta(n)$.

——这一结论对于使用邻接表或邻接矩阵都成立。

2）时间分析：分两种存储结构讨论。

（I）若 G 采用邻接表表示

- 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：$O(\Sigma d(u)) = O(e)$。

这是因为：循环将对 G 的每一个结点都至多会处理一次（当 $\text{visited}(u) = 1$ 时），此时对 u 的 $d(u)$ 个邻接结点都要进行判断（是否访问过）和处理（访问未访问过的节点）。

- **VISITED数组的初始化**时间：$O(n)$

所以，算法的总时间是：$O(n+e)$。

(II) 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，所以所有结点的处理时间是：$O(n^2)$。

所以算法的总时间是：$O(n^2)$

另外，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：

使用邻接表表示有：$t(n, e) = \Theta(n+e)$

或，使用邻接矩阵表示有：$t(n, e) = \Theta(n^2)$

证毕。

(2) 宽度优先周游

若 *G* 是**非连通无向图**或**非强连通有向图**，则**一次调用 BFS** 可能访问不到 *G* 中的所有结点。这个时候可从那些尚未被访问的结点开始重复多次调用 BFS，就可以**“周游”图 *G***。

图的宽度优先周游算法：

```
procedure BFT(G, n)   //宽度优先周游，n 是图 G 中的结点数//
    int VISITED(n)
    for i←1 to n do VISITED(i)←0 repeat   //将 VISITED 数组清0
    for i←1 to n do  // 对 VISITED(i) = 0 的结点调用BFS
        if VISITED(i) = 0 then call BFS(i)  endif
    repeat
end BFT
```

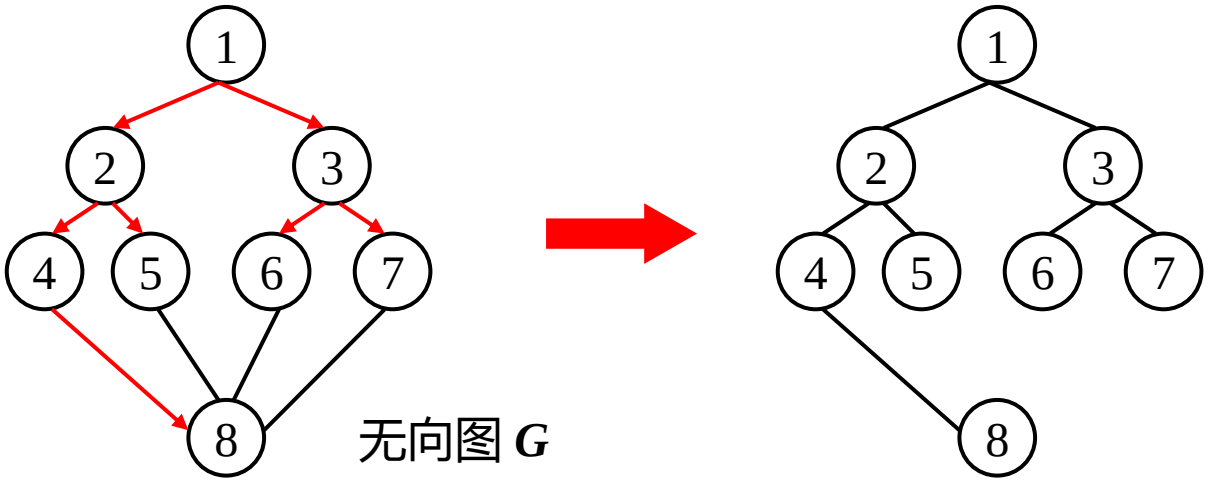
(3) 图周游算法的应用

- **判定图 G 的连通性**：若周游一个图调用 BFS 的**次数多于 1 次**，则说明图 G 是非连通的。
- **生成图 G 的连通分图**：对图 G，一次调用 BFS 中所访问到的所有结点和连接这些结点的边构成图 G 的一个连通分图。
- **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（从 i 到 j 有路径可达），则在**自反传递闭包矩阵 A^*** 中置 $A^*[i, j] = 1$。

(4) 宽度优先生成树

向前边：算法中由 u 达到一个未被访问过的结点 w 的边 (u, w) 称为一条**向前边**。

宽度优先生成树：记 T 是 BFS 处理过程中收集到的所有**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

BFS* 算法：收集向前边，终止时，T 构成 G 的一棵生成树

修改 BFS 算法，在第1行和第6行增加了语句$T\leftarrow\Phi$和$T\leftarrow T\cup\{(u, w)\}$。修改后的算法称为 BFS*。若 v 是连通无向图中任一结点，BFS* 算法终止时，T 中的边组成一棵生成树。

```
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    // 如果 w 之前未被检测//
            T←T∪{(u, w)}           // T 中加入一条向前边 //
            call ADDQ(w, Q)        //将 w 加入队列 Q//
            VISITED(w)←1           //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。**

故 T **是** G **的一棵生成树。**

注：**有 n 个结点和 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 且尚未被访问过的结点的中找出一个**新结点 w**，并从 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 // ***v*** 被访问

for 邻接于 *v* 的每个结点 *w* do

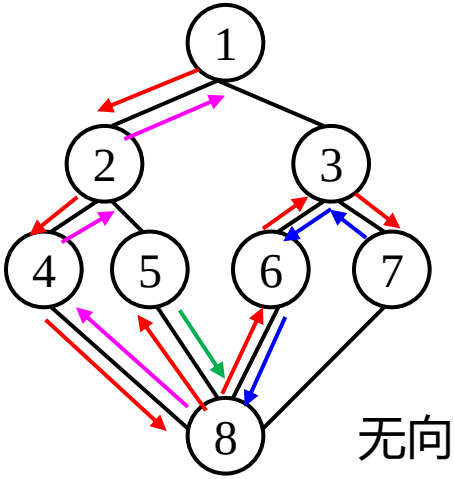
if VISITED(*w*) = 0 then call DFS(*w*) endif

repeat

递归，返回后再检测 *v* 的其它邻接点

END DFS

例:



无向图 G

对于上图 G，DFS 得到的结点访问序列:

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** 对一个有 n 结点 e 条边的图所花的时间和附加空间，则

- $s(n, e) = \Theta(n)$ （最大递归深度为 n）
- G 采用**邻接表**表示：$t(n, e) = \Theta(n + e)$，或
- G 采用**邻接矩阵**表示：$t(n, e) = \Theta(n^2)$

(2) 深度优先周游算法 DFT

与 BFS 算法同理，若 G 是**无向非连通图**或**非强连通有向图**，则**一次调用 DFS** 可能访问不到 G 中的所有结点。这个时候从那些尚未被访问过的结点开始重复多次调用 DFS 就可以**“周游”图 G**。

(3) 应用：

① **判定图 G 的连通性**

② **找 G 的连通分图**

③ **构造无向图的自反传递闭包矩阵**

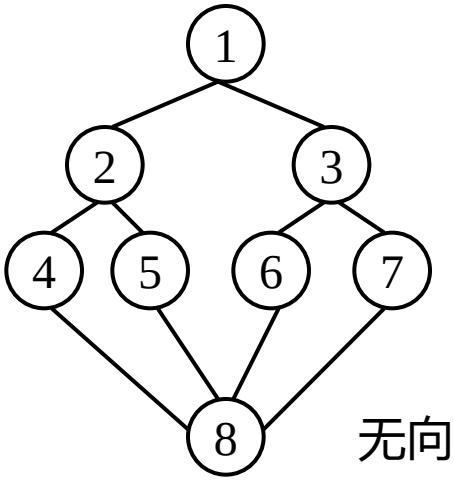
④ **深度优先生成树**

生成深度优先生成树的算法

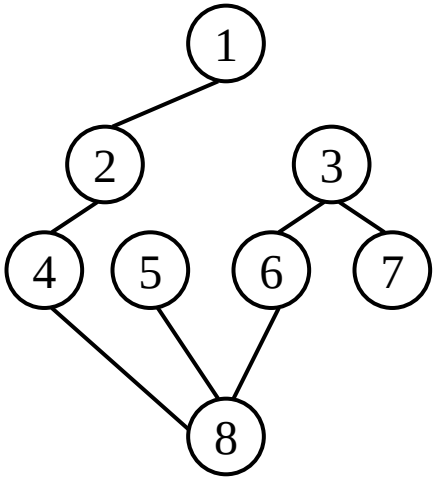
$T \leftarrow \Phi$

```
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∪{(v, w)}  //收集向前边
            call DFS(w, T)
        endif
    repeat
END DFS*
```

DFS 中由 u 到达一个未访问结点 w 的边 (u, w)也称为**向前边**。用**边集** T 收集**向前边**，修改后的算法称为**DFS***。 **DFS*** 算法终止时，T **中的边组成** G **的一棵生成树**。



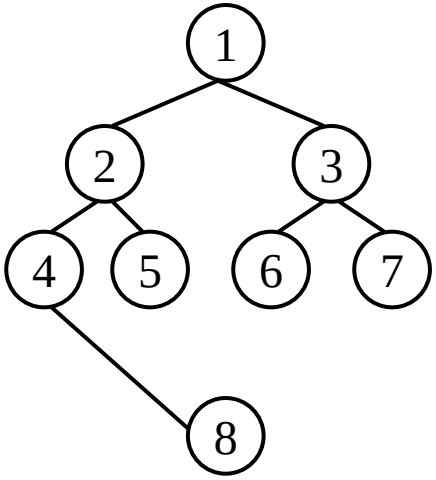
无向图 *G*



G 的深度优先生成树

结点的深度优先访问序列

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



G 的宽度优先生成树

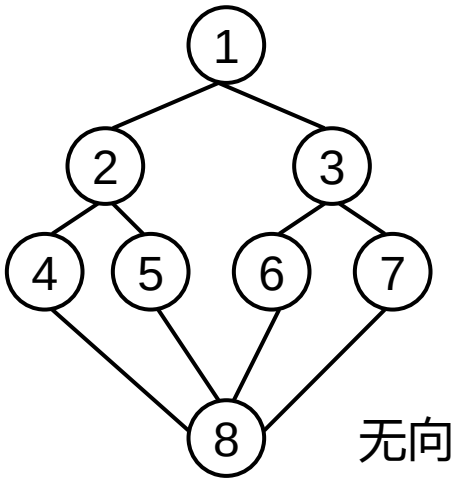
结点的宽度优先访问序列

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

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：栈是先进后出，结点被压入栈中后将以**相反的次序**出栈。



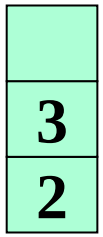
无向图 *G*

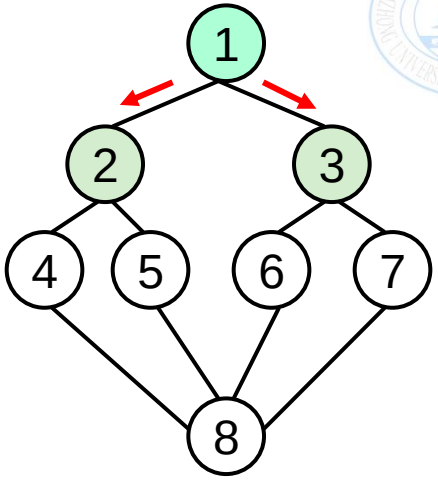
例:

检测结点1:

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

栈状态:



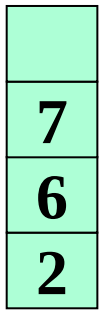


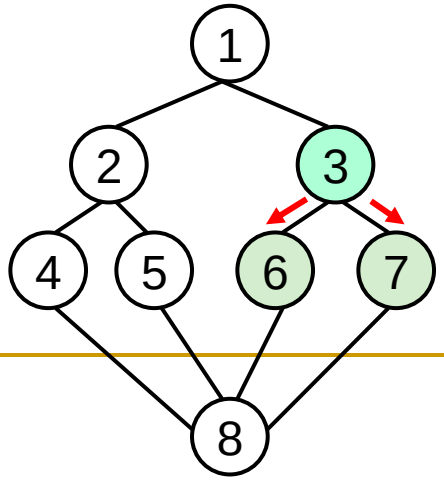
无向图 ***G***

检测结点3（结点3出栈）：

visited(6) =1、visited(7)=1

栈状态:

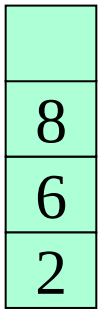


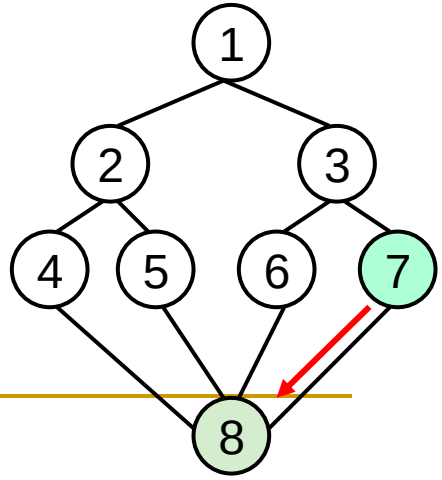


检测结点7（结点7出栈）：

visited(8) =1

栈状态:

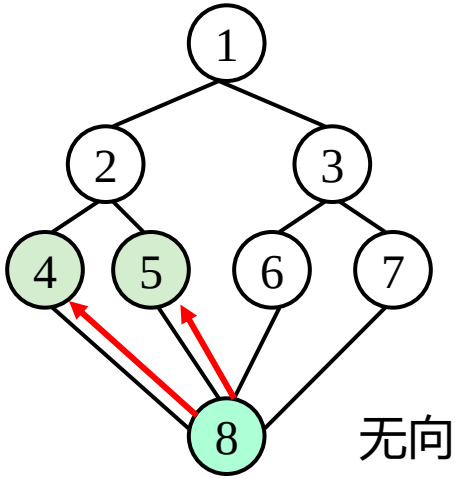




检测结点8（结点8出栈）：

visited(4) =1、 visited(5) =1

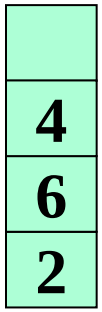
栈状态：



无向图 ***G***

检测结点5（结点5出栈）：

栈状态：

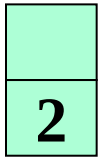


检测结点4（结点4出栈）：

栈状态：

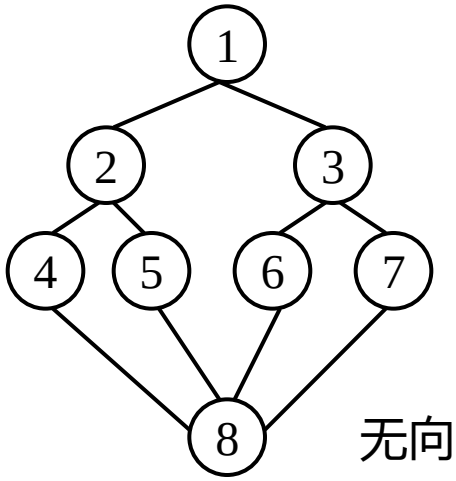
检测结点6（结点6出栈）：

栈状态：



检测结点2（结点2出栈）：

栈状态：



无向图 ***G***

对于上图G，**D_Search** 得到的结点访问序列:

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

对照：**BFS** 的结点访问序列：**1, 2, 3, 4, 5, 6, 7, 8**

五大常用的算法设计基本策略：

分治、动态规划、贪心、回溯和分支限界

22.3 回溯法

回溯法是算法设计的基本方法之一。用于求问题的一组**特定性质的解**。

- 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** $(x_1, \ldots, x_n)$ **解向量**来表示;

- 其中每个 x_i 取自于某个**有穷集** S_i。

(2) 求解问题的目标:

- 求取一个满足某**规范函数** $P(x_1, \ldots, x_n)$ 所规定的条件的**解向量，如极值或其它条件**。

例：**分类（排序）问题**

对 $A(1{:}n)$ 的元素分类问题

- 用 n **元组**表示解：$(x_1, x_2, \ldots, x_n)$
- x_i：表示第 i 小元素在**原始数组**里的**下标**，

  取自有穷集 $S_i = [1..n]$。
- **规范函数**P：$A(x_i) \le A(x_{i+1})$，$1 \le i < n$

1、如何求取满足规范函数的元组?

- **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**候选元组总数**为

    $$m = m_1 m_2 \ldots m_n$$

    存在的问题：盲目求解，**计算量大**，甚至不可行。
- 寻找其它**有效的策略**
  - **回溯** 或 **分支限界法**

回溯、分支限界法能带来什么样的改进?

- 进行**系统化搜索**，**避免盲目求解。**
- **逐步构造元组分量**，**不断修正规范函数**，并逐步测试正在构造中的 n 元组部分向量 $(x_1, \ldots, x_i)$，看其能否导致问题的解。
- **如果判定** $(x_1, \ldots, x_i)$ **不可能导致问题的解**，则将后面可能要测试的 $m_{i+1} \ldots m_n$ **个向量一概略去——剪枝**，这相对硬性处理**大大减少了计算量。**

一些相关概念：

- **约束条件**：问题的解需要满足的条件称为约束条件。

  又分为**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 $S_i = \{l_i \leq a \leq u_i\}$

- **可行解**：满足**显式约束条件**的元组称为问题的**可行解**。

- **解空间**：问题的**可行解集合**构成这个问题的**解空间**。

- **隐式约束条件**：用来规定一个问题的解空间中那些**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1 8-皇后问题

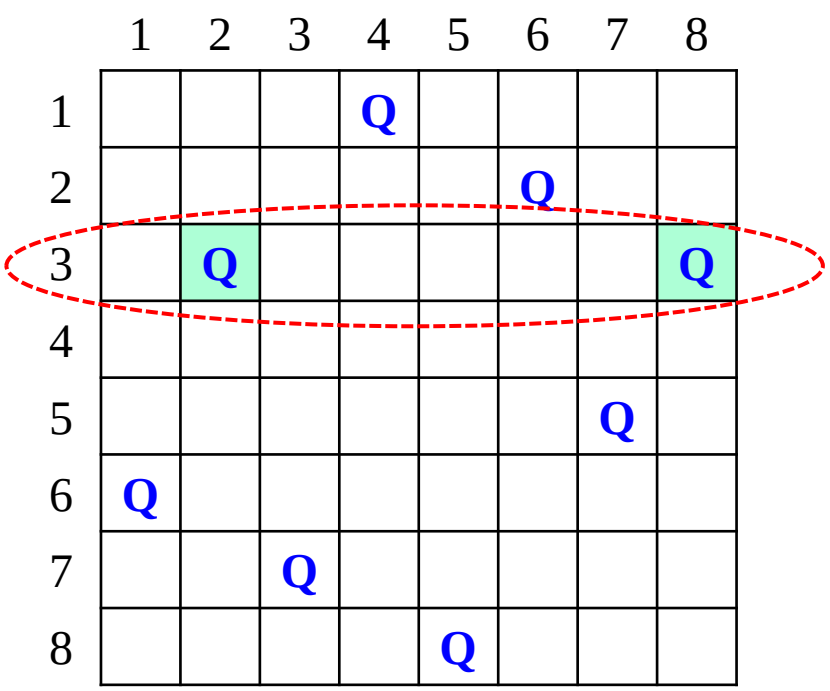
在一个 **8×8** 棋盘上放置 **8 个皇后**，且使得每两个皇后之间都不互相“攻击”，即：**任何两个皇后都不在同一行、同一列或同一条斜角线上。**

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

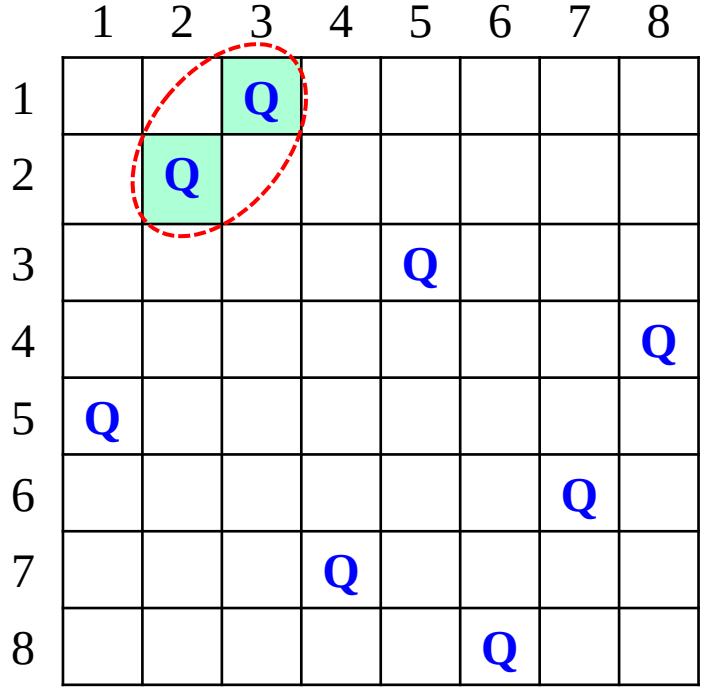
	1	2	3	4	5	6	7	8
1			Q					
2					Q			
3		Q						
4								Q
5	Q							
6							Q	
7				Q				
8						Q		

8-皇后问题的合法放置

8-皇后问题的非法放置



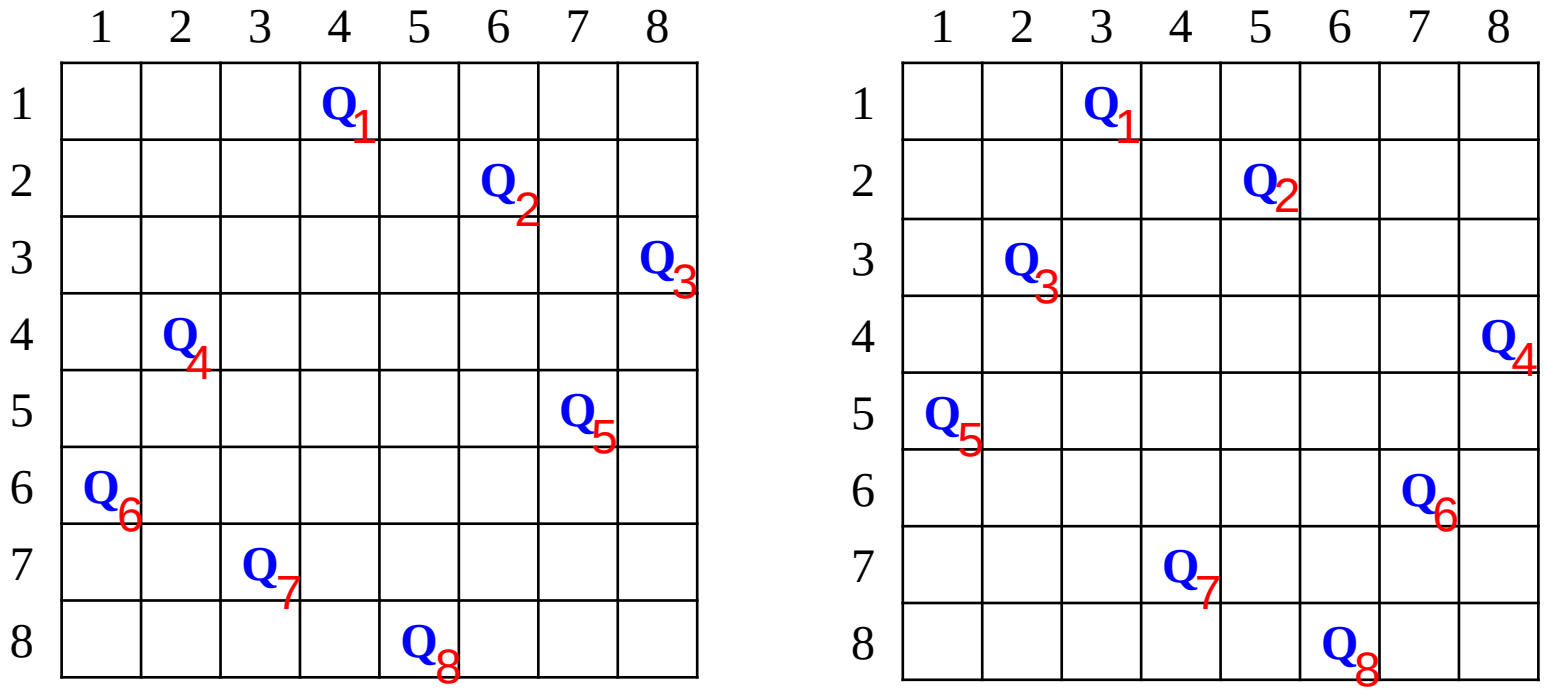
有同行的两个皇后



有同对角线的两个皇后

- **行、列号**：行号1~8，列号1~8；
- **皇后编号**：8个皇后也编号1~8。

不失一般性，**约定皇后 *i* 放在第 *i* 行的某一列上。**如：



数字标注的是皇后的编号

8-皇后问题的合法放置

解的表示：基于上述约定，**问题的解**就可以用 **8-元组** $(x_1, \ldots, x_8)$ 表示，其中 x_i 是**皇后 i 所在的列号**。

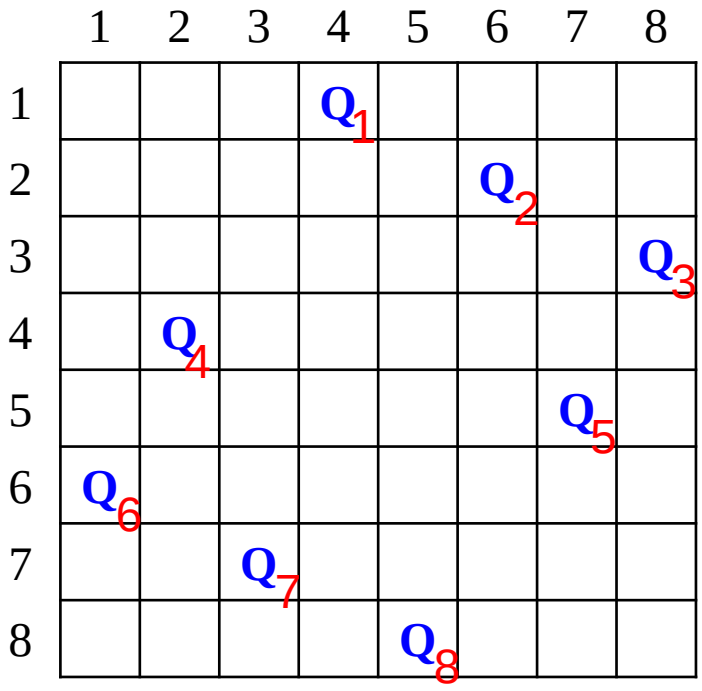
这里，

- **显式约束条件**：$S_i = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $1 \le i \le 8$
- **解空间**：所有可能的 8元组，由于每个 x_i 均可取值 1~8，所以总共有 8^8 个元组。
- **隐式约束条件**：用来描述 x_i 之间的约束关系，要求**任何两个 x_i 不可相同**（不在同一列）且**任何两个皇后不可以在同一条斜角线上**。

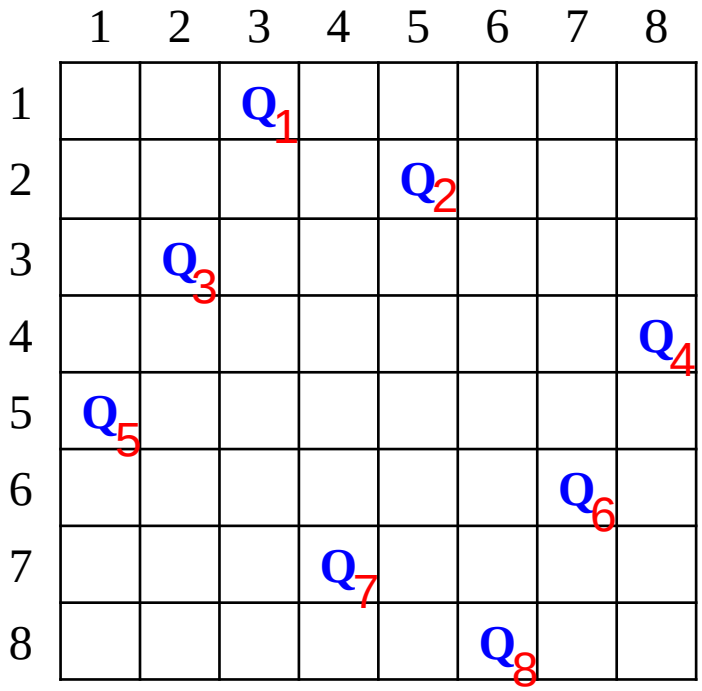
由隐式约束条件可知：**解元组中只能是 1~8 的排列**。

这样，不同的元组最多有 8! 个。

如前面给出的 **8**-皇后问题的两个解可表示为：



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



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

x_i 是皇后 i 所在的列号

例2 子集和数问题

已知 n 个正数的集合 $W=\{w_1, w_2, \ldots, w_n\}$ 和一个正数 M。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* 元单位向量**表示）：(1, 1, 0, 1) 和 (0, 0, 1, 1)

元组也有不同的表示形式

子集和数问题解的表示：

形式一：***k*-元组**

***k*-元组**：$(x_1, x_2, \ldots, x_k)$, $1 \le k \le n$，其中每个 x_j 为选中的元素在**原集合**中的**下标**。

特点：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** $1 \le j \le n \}$。

隐式约束条件：1）**没有两个** x_i **是相同的**；

2） $\Sigma w_{xi} = M$；

3） $x_i < x_{i+1}$，$1 \le i < n$

（按下标从小到大的顺序选元素，**避免重复元组**）

形式二： *n*-元组

n-元组：$(x_1, x_2, \ldots, x_n)$，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。

2、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**。那么采用何种形式组织问题的解空间？

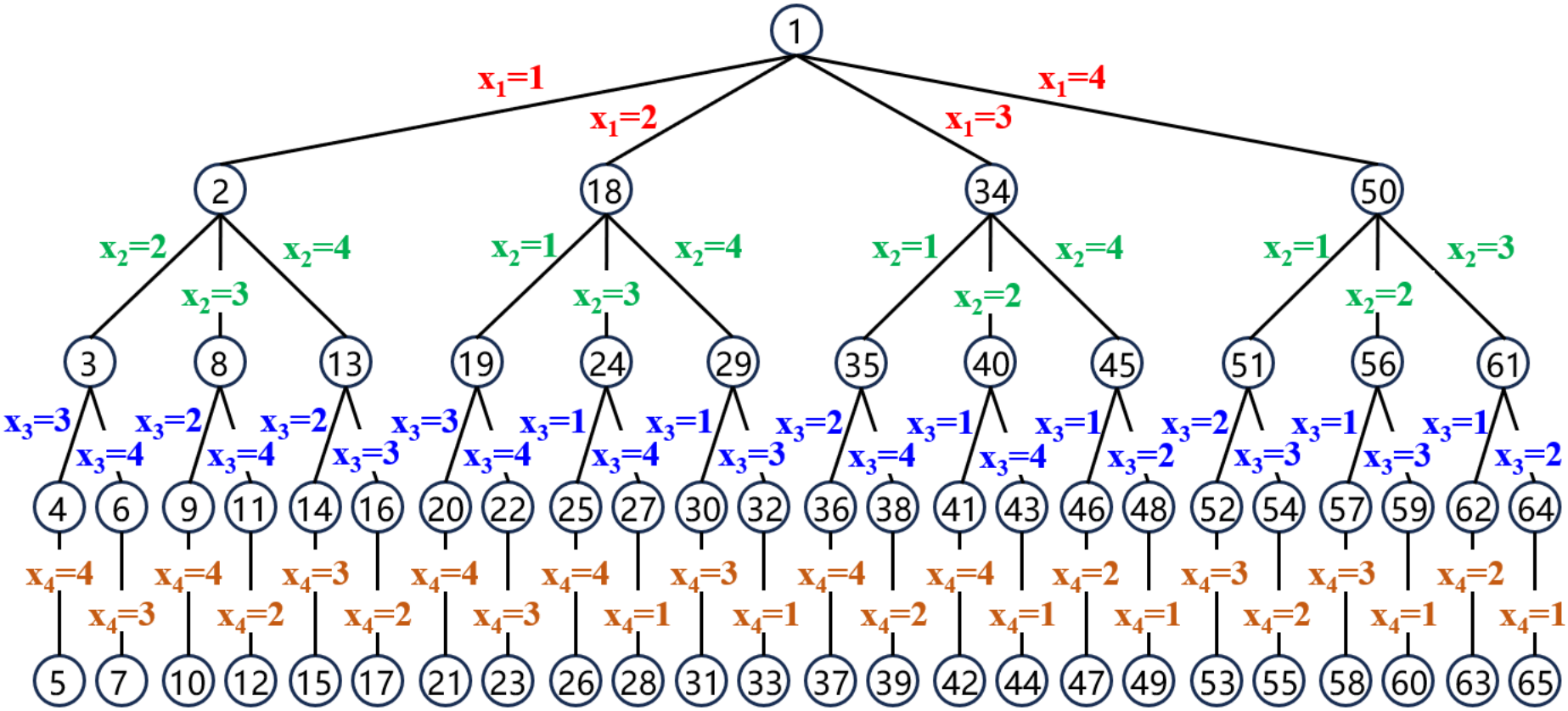
可以用**树结构**组织解空间——**状态空间树**。

例 ***n*-皇后问题**。

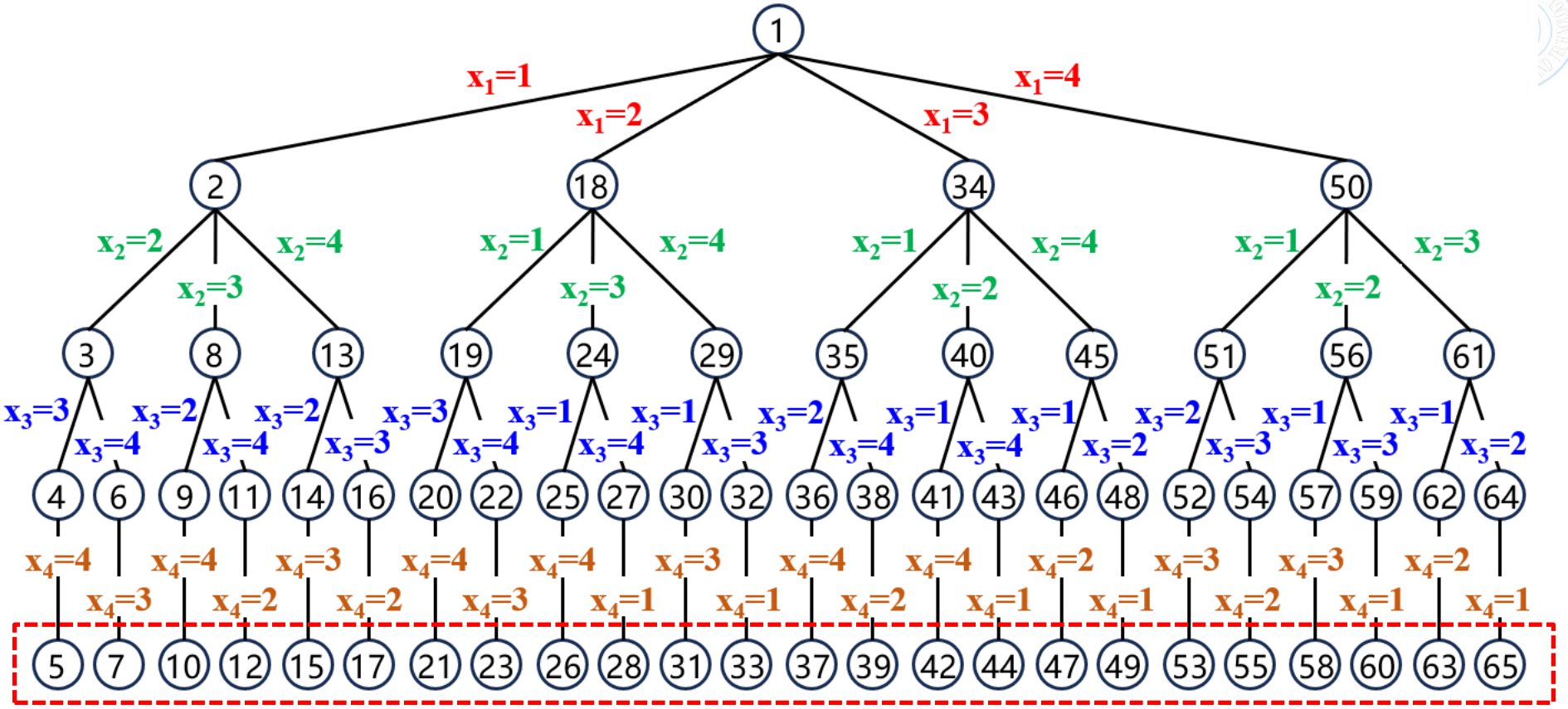
8-皇后问题的推广：在 $n \times n$ 的棋盘上放置 n **个皇后**，使它们不会相互攻击。

解空间：由 $n!$ 个 ***n*-元组**组成；每个元组是 **1~*n*** 的**排列**。

实例：**4-皇后问题**的**解空间树结构**如下所示：



边：从上往下，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 级结点到 2 级结点的边给出了 x_1 的各种取值：1、2、3、4。



解空间：**从根结点到叶结点的路径**给出了**解空间**的**可行解**，由从根结点到叶结点路径上的边标记 (x_1, x_2, x_3, x_4) 表示。

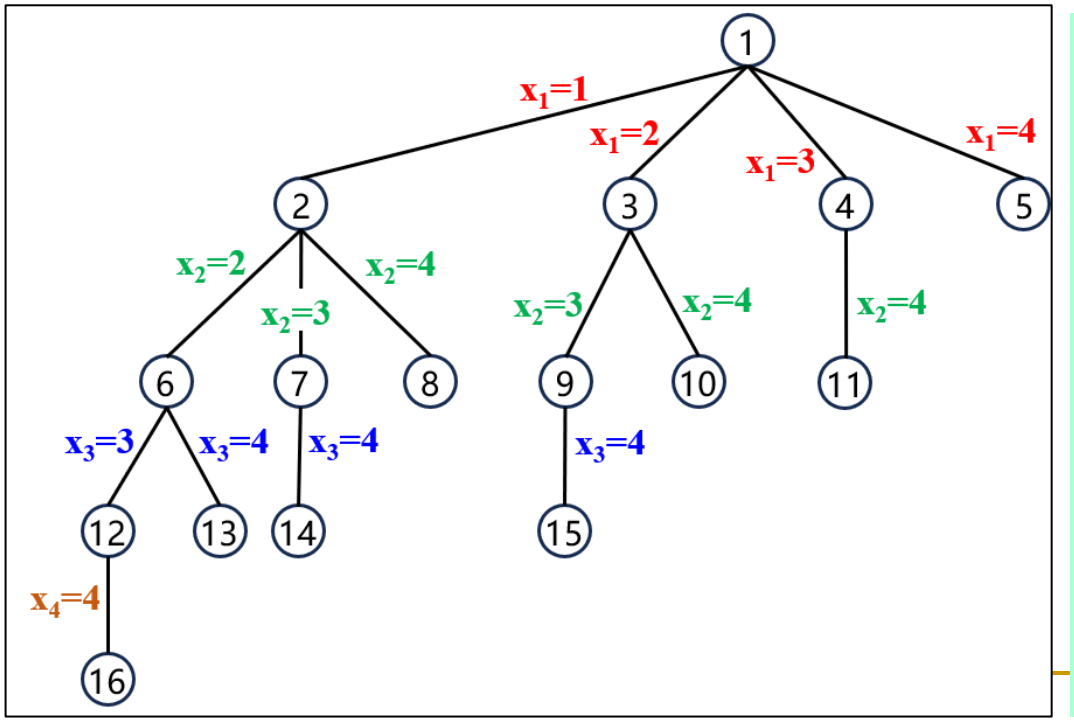
注：总共有24个叶结点，对应24个可行解，恰好是 1~4 **的排列**，(4!=24) —— 故也称**排列树**。

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 $(x_i < x_{i+1})$

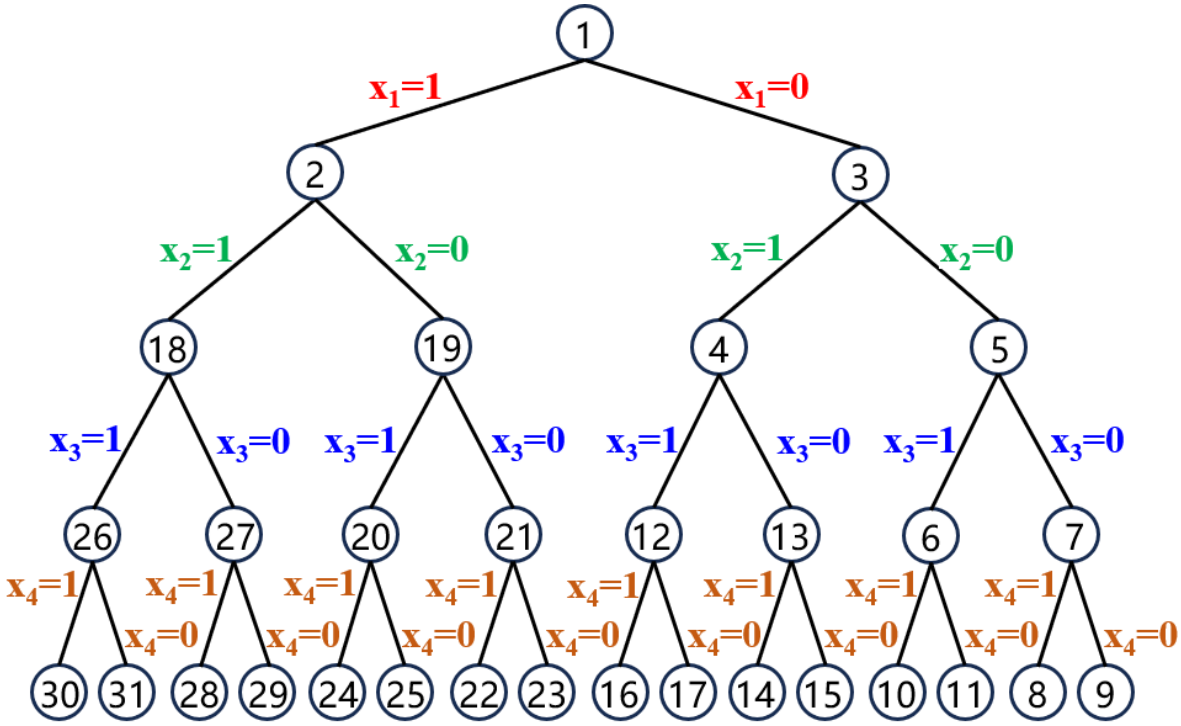
树边标记：由 i 级结点到 $i+1$ 级结点的一条边用 x_i 来表示，表示 k-**元组**里的第 i 个元素是原集合中下标为 x_i 的元素。



解空间由根结点到任何结点的所有路径给出：( ), (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。

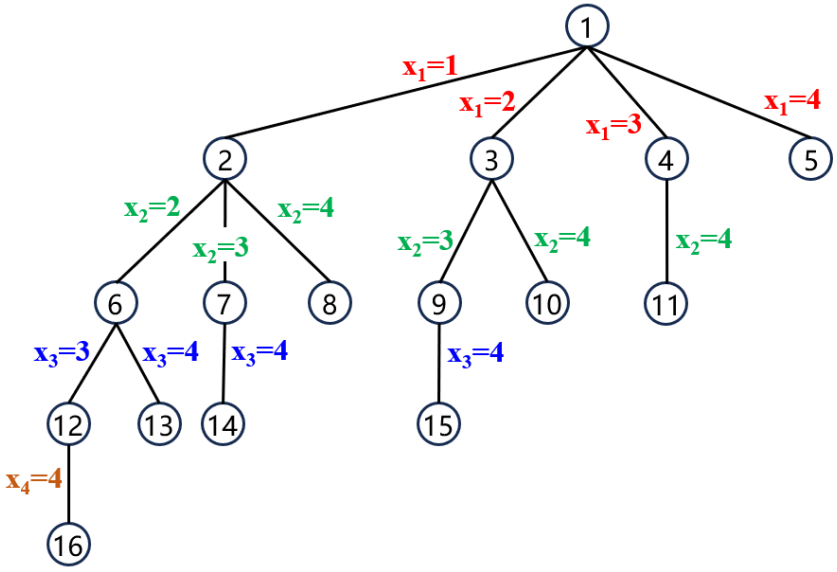
树边标记：由 i 级结点到 $i+1$ 级结点的那些边用 x_i 的值来标记，$x_i = 1$ 表示选择了 i 元素，0 表示没有选择。

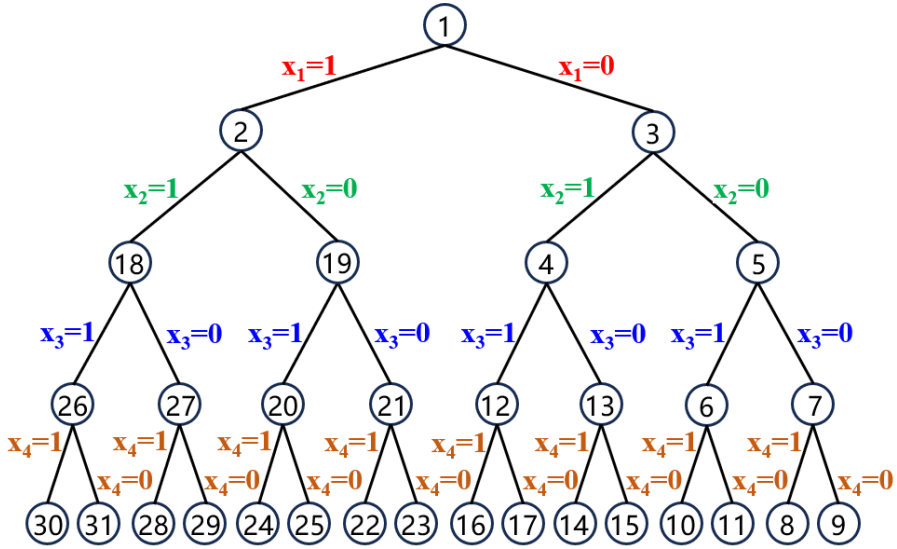


解空间由根到叶结点的所有路径确定。共有 16 个元组。

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的解的表示形式和相应的不同形态的状态空间树。





关于状态空间树的一些基本概念

- **状态空间树**：解空间的树结构。
- **问题状态**：状态空间树中的每个结点代表问题的一个**状态**，称为**问题状态**。
- **状态空间**：**问题状态的集合**，由根结点到各个结点的路径确定。
- **解 状 态**：是这样的一些**问题状态** S，对于这些问题状态，由根到 S 的路径可以确定这个问题**解空间中的一个元组**。
- **答案状态**：是这样的一些**解状态** S，对于这些解状态，由根到 S 的路径确定了**问题的一个解**（解是满足隐式约束条件的元组）。

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**：当前正在生成儿子结点的活结点。（正在扩展的结点）
  - **死结点**：不需要再进一步生成儿子结点或者儿子结点已全部生成的结点。

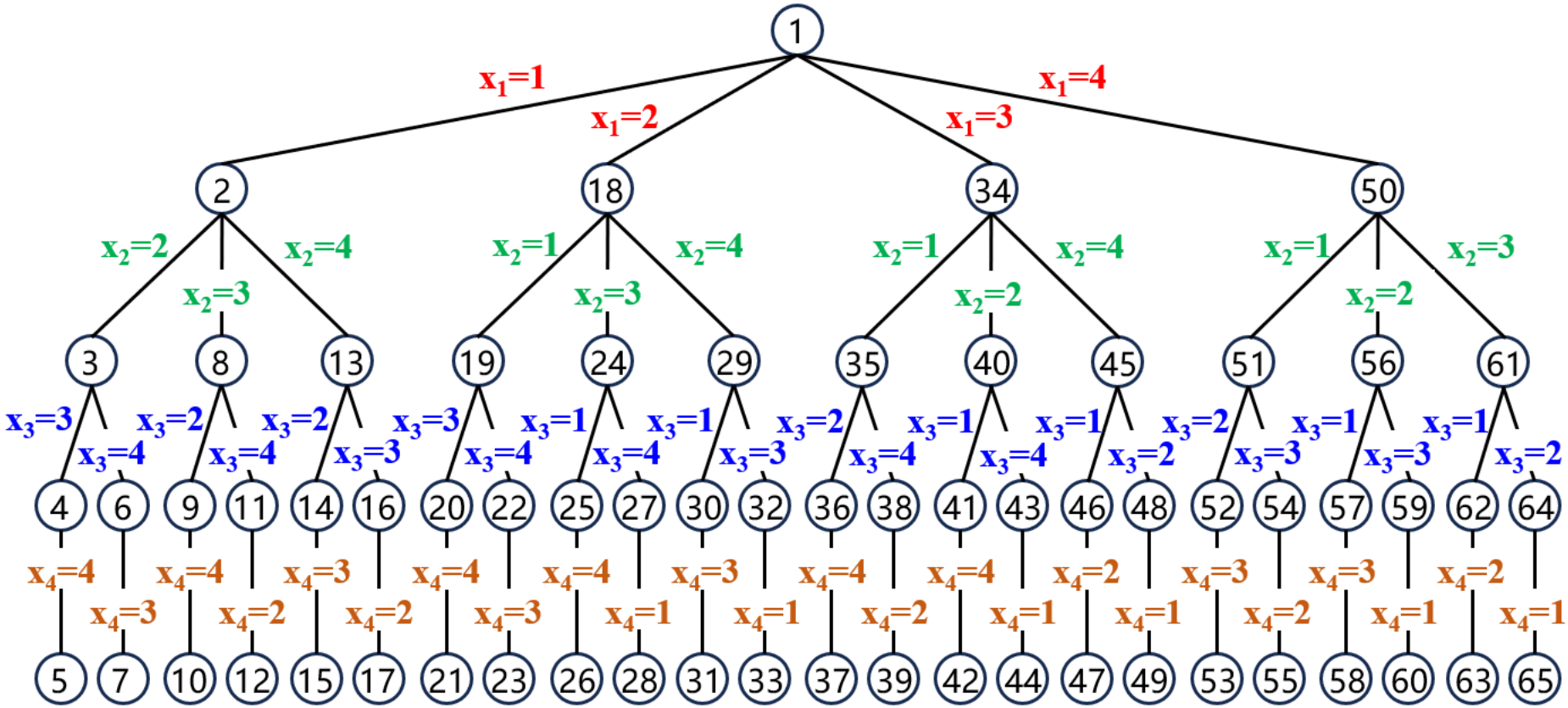
构造状态空间树的两种策略

（1）**深度优先策略**：当当前 *E*-结点 *R* 一旦生成一个新儿子结点 *C* 时，*C* 马上变成一个新的 *E*-结点，之后开始对 *C* 的检测；当完全检测了结点 *C* 之后（*C* 的儿子结点扩展完毕返回后），*R* 结点再次成为 *E*-结点并继续扩展它的其它儿子结点。

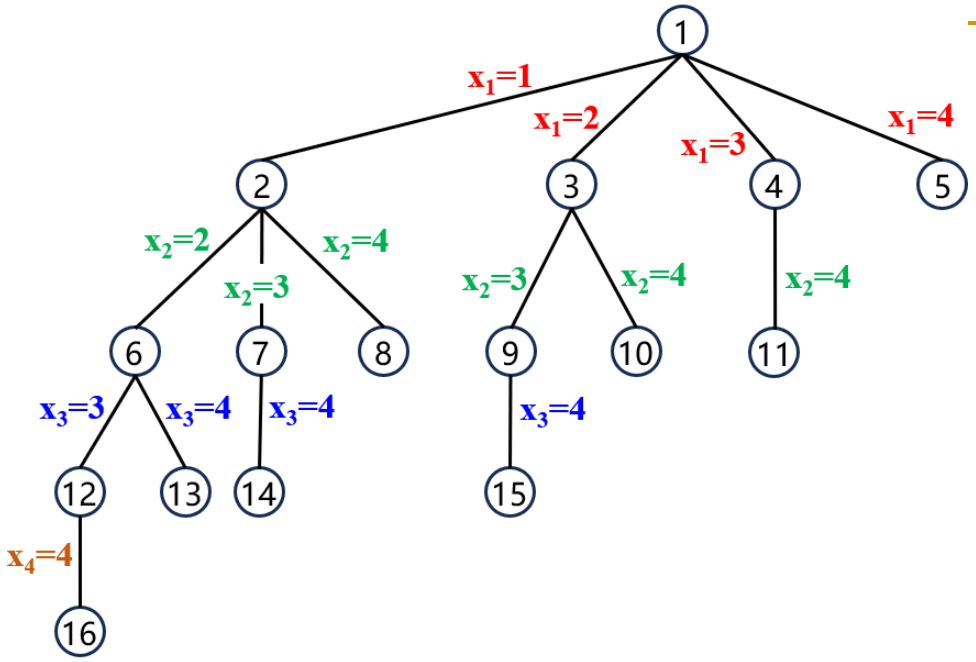
（2）**宽度优先策略**：对当前 *E*-结点，一次性扩展出它的所有儿子结点而自己变成死结点，然后再扩展其它待检测的结点。

限界函数：在结点生成的过程中，定义一个**限界函数**，用来**杀死一些还没有生成全部儿子结点的活结点**（这些活结点是那些已无法满足限界函数的条件、不可能导致问题答案的结点，再检测已无意义），从而加快整个算法的检测速度。

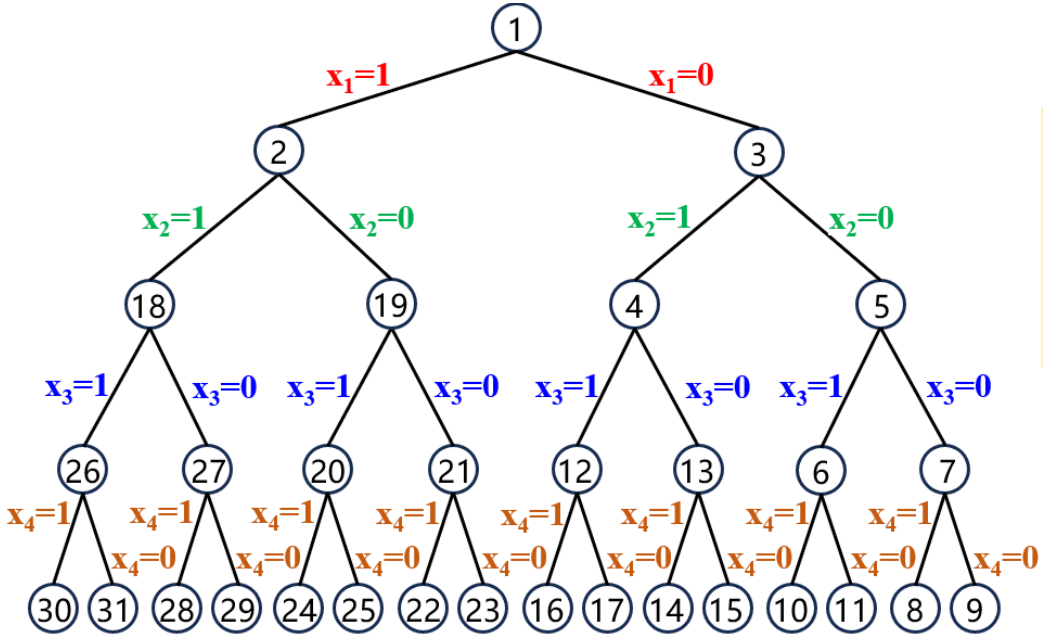
- **回溯法**：使用限界函数的**深度优先**结点生成方法称为**回溯法**（backtracking）。
- **分支-限界方法**：使用限界函数的**宽度优先**结点生成方法称为**分支-限界方法**（branch-and-bound)。



深度优先策略下的结点生成次序（看**结点编号**）



使用**队列**的**宽度优先**策略下的结点生成次序(**BFS**)



使用**栈**的**宽度优先**策略下的结点生成次序（**D-Search**）

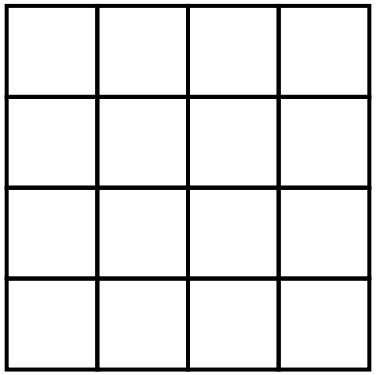
例：4-皇后问题的回溯法求解

- **限界函数**：如果 $(x_1, x_2, \ldots, x_{i-1})$ 是根结点到当前 E 结点 x_{i-1} 的路径，那么 x_{i-1} 的**儿子结点** x_i 是这样的一些结点：它们使得 $(x_1, x_2, \ldots, x_{i-1}, x_i)$ 表示没有两个皇后处在相互攻击状态的一种棋盘格局（否则就没必要生成）。

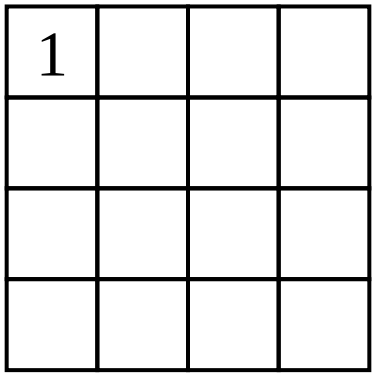
- **开始状态**：根结点1，此时棋盘为空，还没有放置任何皇后。

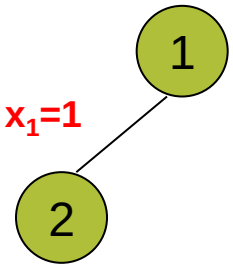
- **生成结点的过程就是依次考察皇后 1~ n 合法放置位置的过程**。

- 按照编号“**自然数递增**”的次序考虑结点并生成 4-皇后问题状态空间树中一个结点的儿子结点的过程：

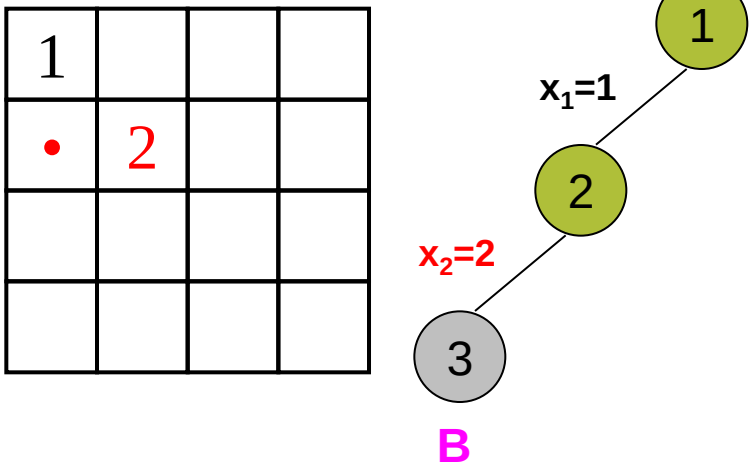


根结点1，开始状态，**唯一的活结点**
解向量：（）





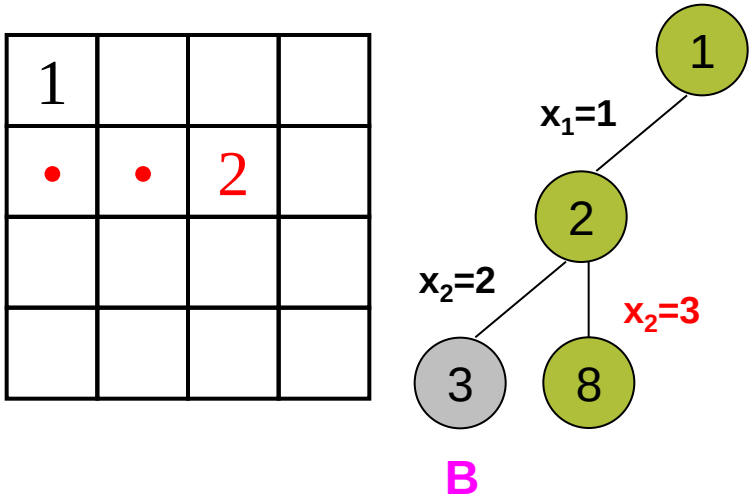
生成结点2，表示皇后1被放到第 1 行的第 1 列上，该结点是从根结点开始生成的第一个结点。
解向量：（1）
之后，**结点2变成新的 E 结点，下一步扩展结点2**



由结点2生成结点3，即皇后2被放到第 2 行第 2 列。

利用限界函数杀死结点3（**1、2结点处在对角线上了。**杀手结点3后，其后的结点 4，5，6，7 就不会生成——**剪枝**）。

回到结点2 继续扩展。

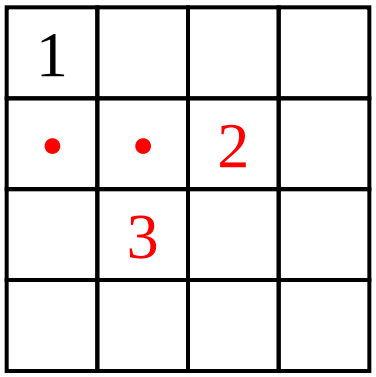


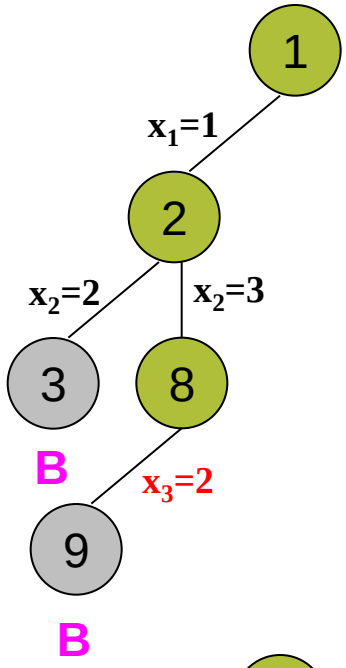
再由结点2生成结点8，即皇后2放到第 2 行第 3 列。

结点8变成新的 *E* 结点。

解向量：（1，3）

从结点8继续扩展。

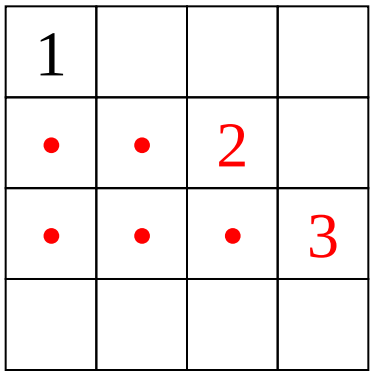


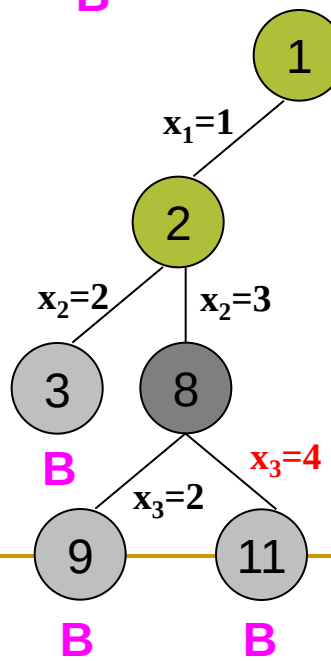


由结点8生成结点9，即皇后3放到第3行第2列。

利用限界函数杀死结点9（结点10不会生成）。

返回结点8继续扩展。





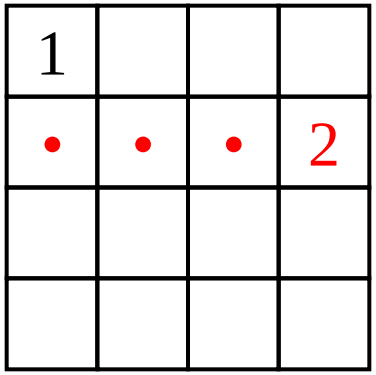
由结点8生成结点11，即皇后3放到第3行第4列。

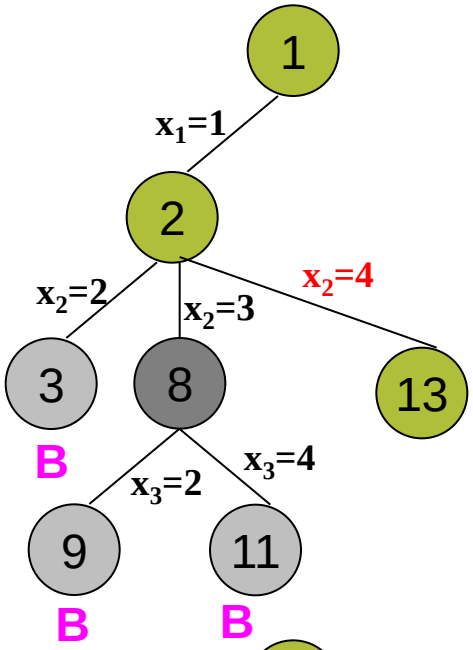
利用限界函数杀死结点11（结点12不会生成）。

返回结点8再继续。

但结点8的所有儿子已经生成，且没有导出答案结点，所以结点8变成死结点。

返回结点2继续扩展。



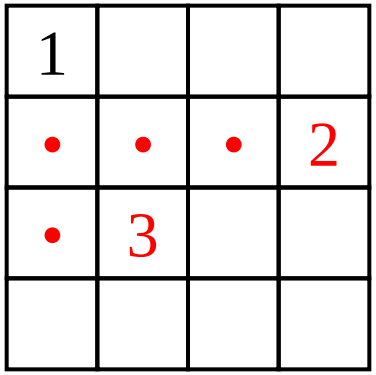


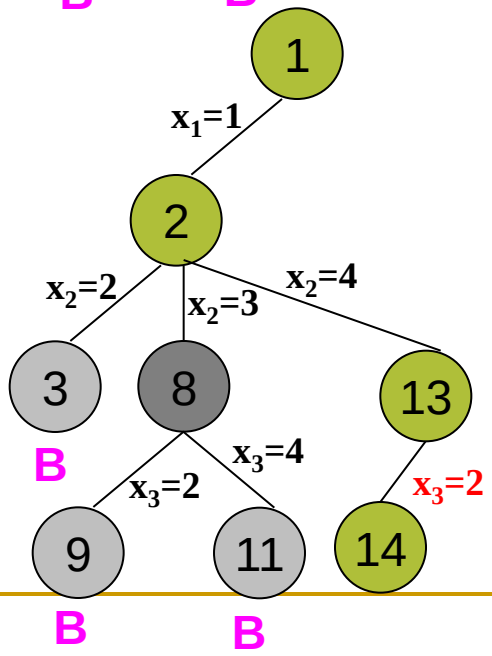
由结点2生成结点13，即皇后2放到第2行第4列。

结点13变成新的 *E* 结点。

解向量：（1，4）

从结点13继续扩展。





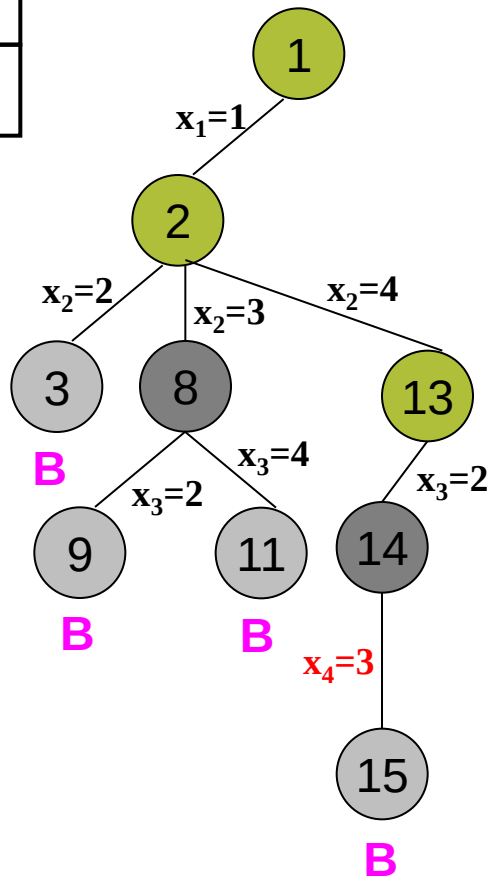
由结点13生成结点14，即皇后3放到第3行第2列。

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	



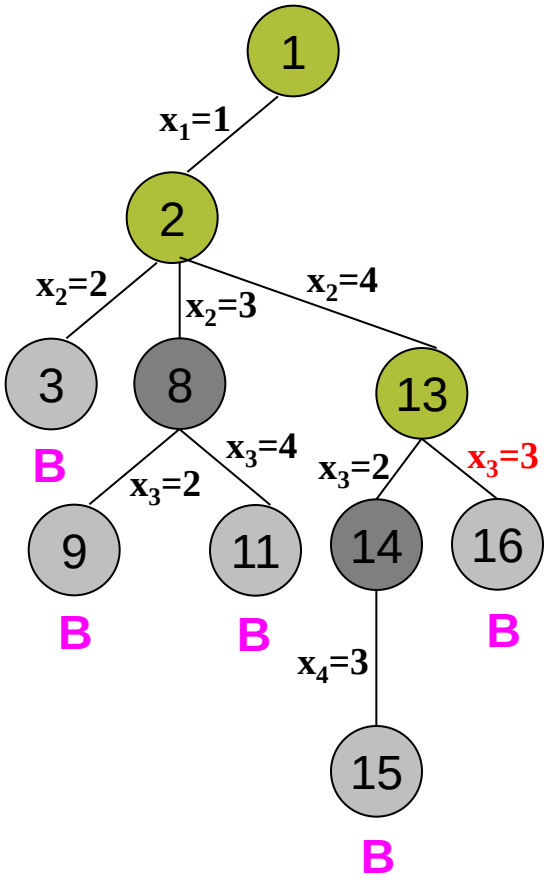
由结点14生成结点15，即皇后4放到第 4 行第 3 列。

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。

返回结点13继续扩展。

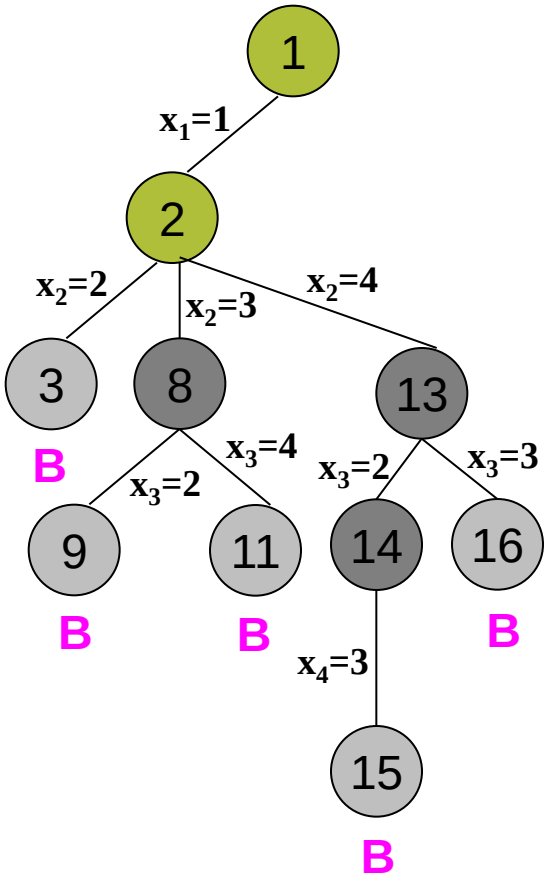
1			
•	•	•	2
•	•	3	



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。

1			
•	•	•	2
•	•	3	

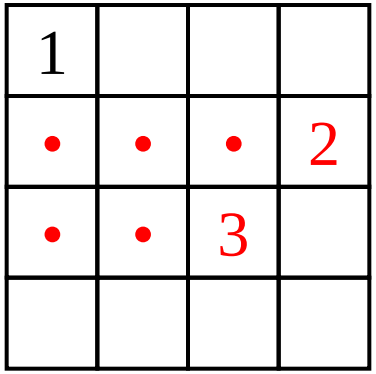


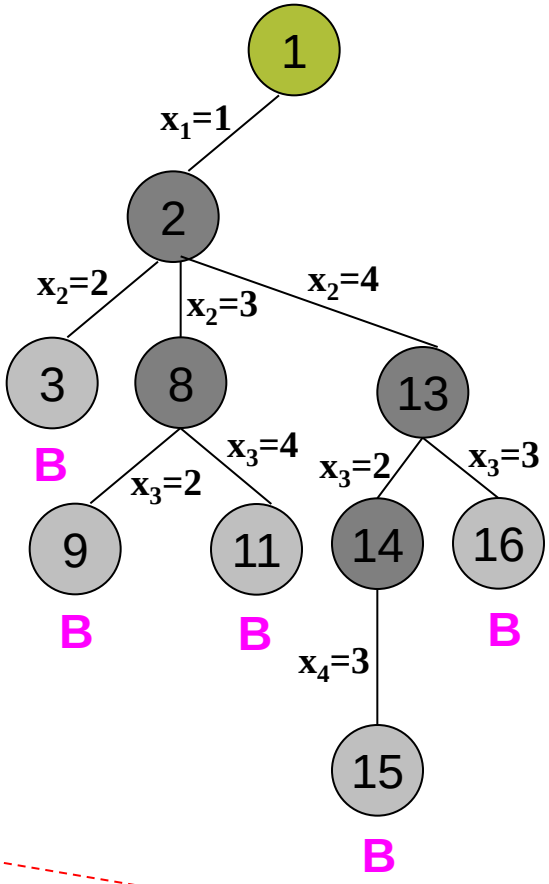
由结点13生成结点16，即皇后3放到第 3 行第 3 列。

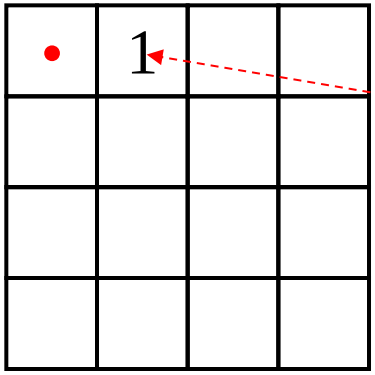
利用限界函数杀死结点16。

结点13不能导致答案结点，变成死结点。

返回结点2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。

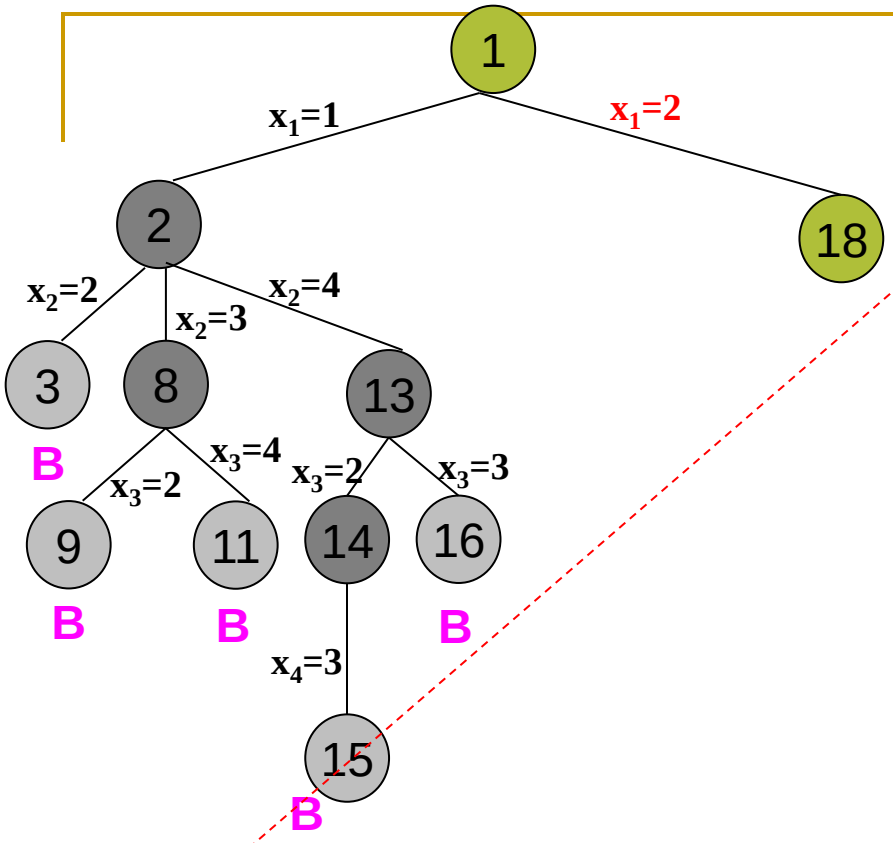
结点13不能导致答案结点，变成死结点。

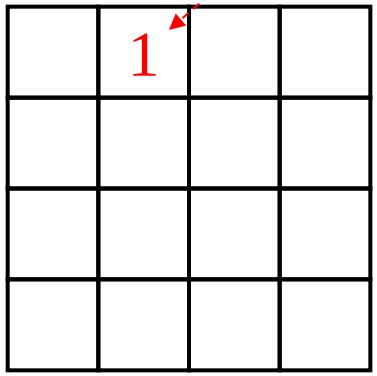
返回结点2继续扩展。

结点2不能导致答案结点，变成死结点。

返回结点1继续扩展。

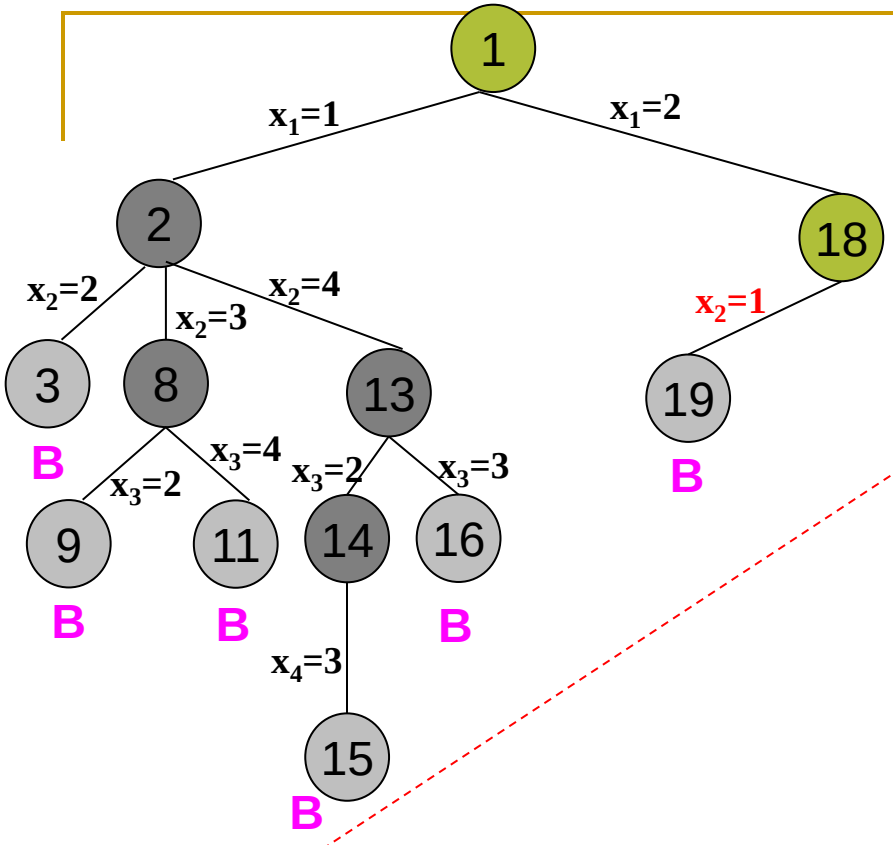
由结点1生成结点18，即皇后 1 放到第 1 行第 2 列。

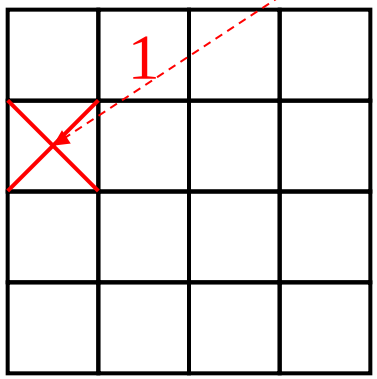




由结点1生成结点18，即皇后1放到第 1 行第 2 列。

结点18变成 *E* 结点。



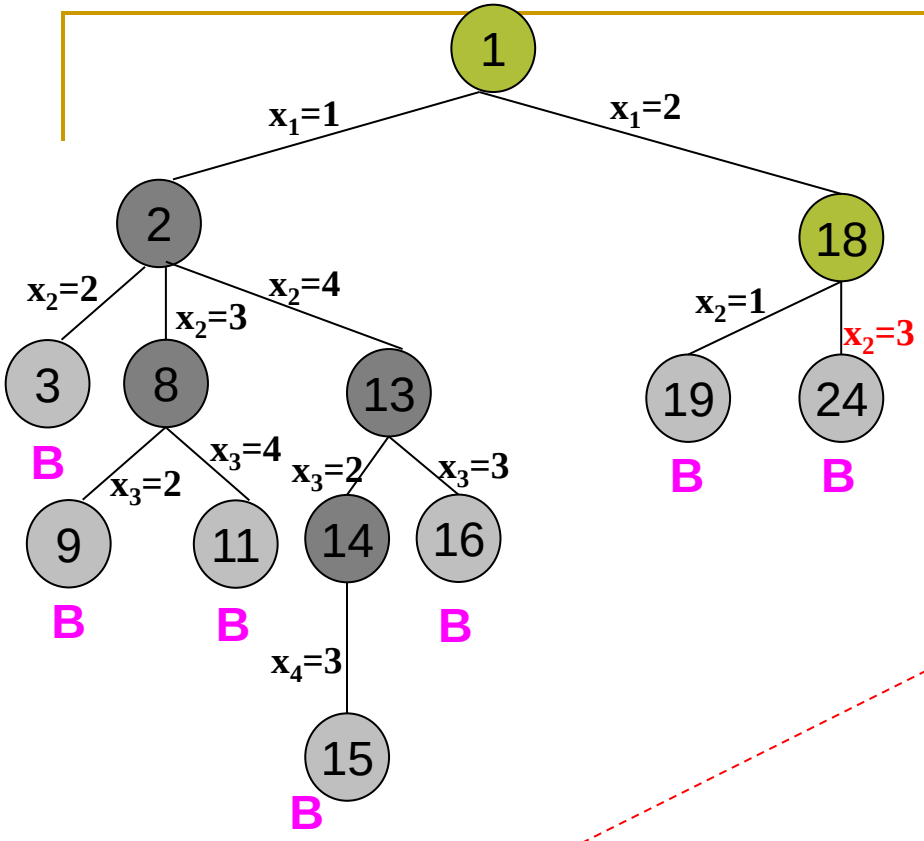


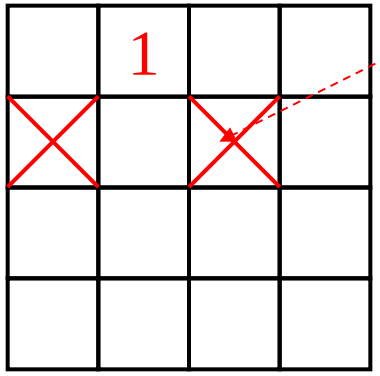
由结点1生成结点18，即皇后1放到第 1 行第 2 列。

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。





由结点1生成结点18，即皇后1放到第 1 行第 2 列。

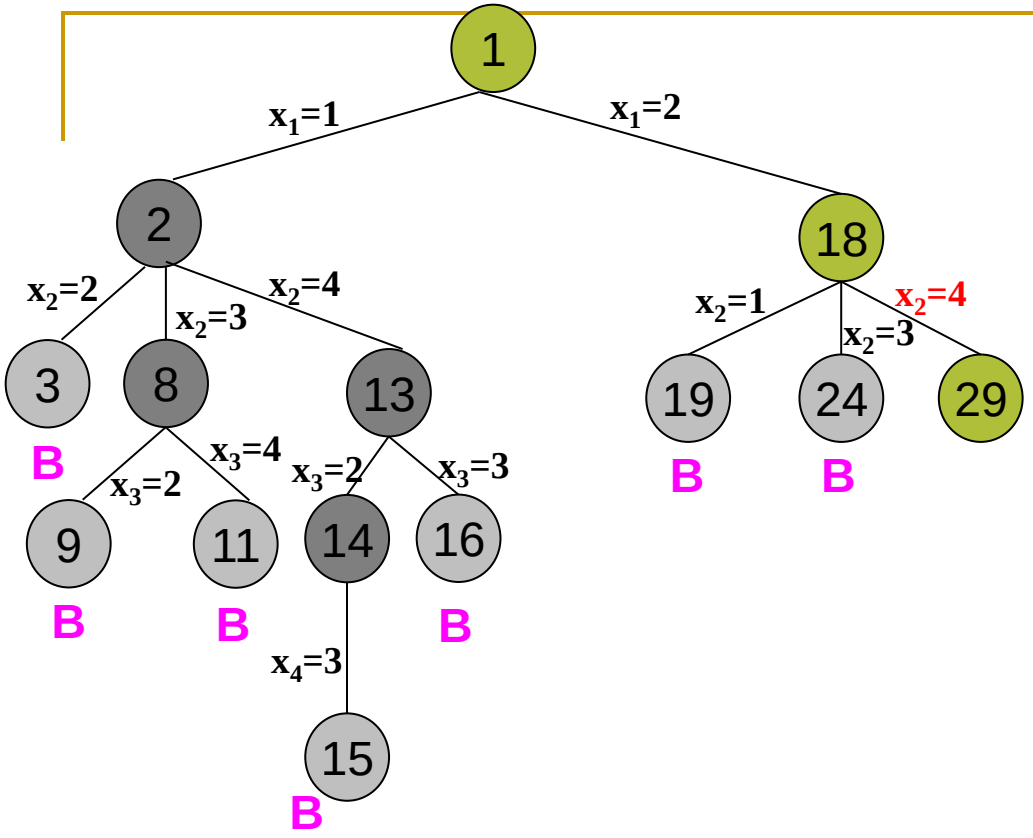
结点18变成 *E* 结点。

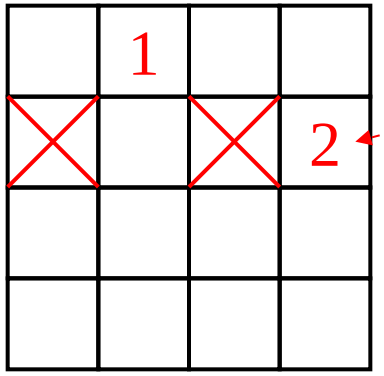
扩展结点18生成结点19，即皇后2放到第 2 行第 1 列。

利用限界函数杀死结点19。

扩张结点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。





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。

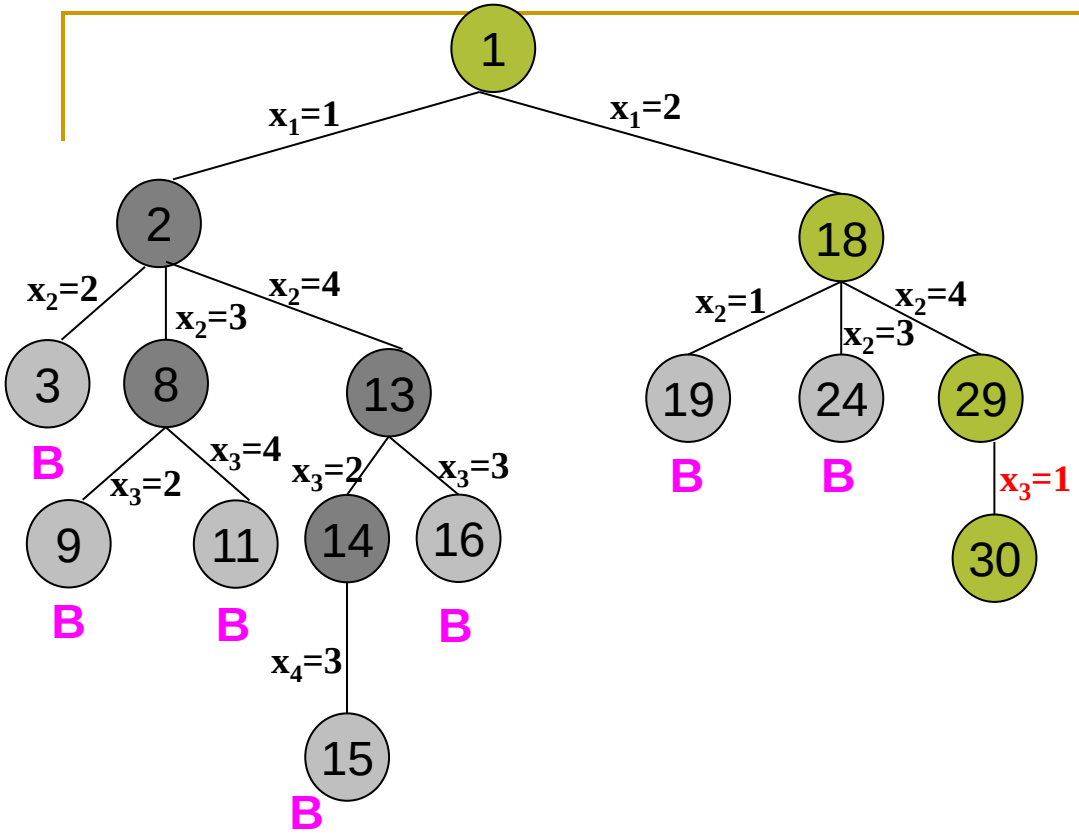
利用限界函数杀死结点19。

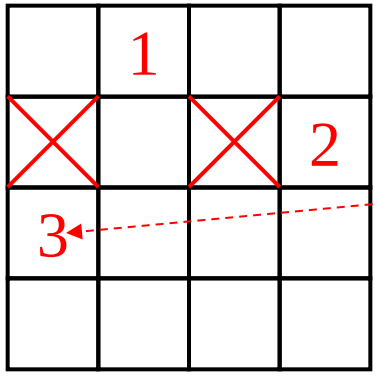
扩张结点18生成结点24，即皇后2放到第 2 行第 3 列。

利用限界函数杀死结点24。

再扩展结点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。





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。

扩张结点18生成结点24，即皇后2放到第 2 行第 3 列。

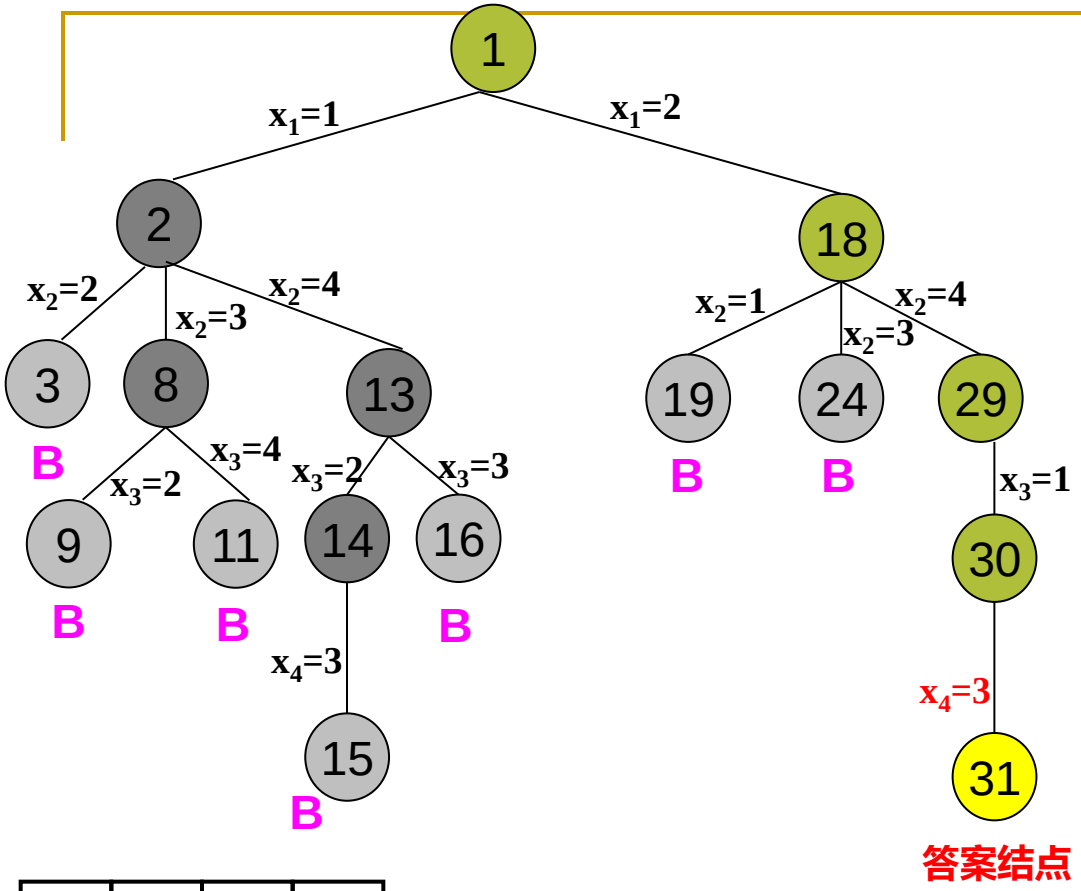
利用限界函数杀死结点24。

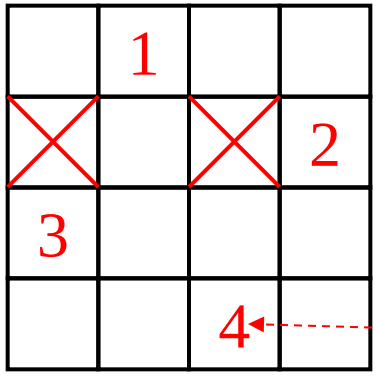
再扩展结点18，生成结点29，即皇后2放到第 2 行第 4 列。

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。





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)。

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。

扩张结点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。

再扩展结点18，生成结点29，即皇后2放到第 2 行第 4 列。

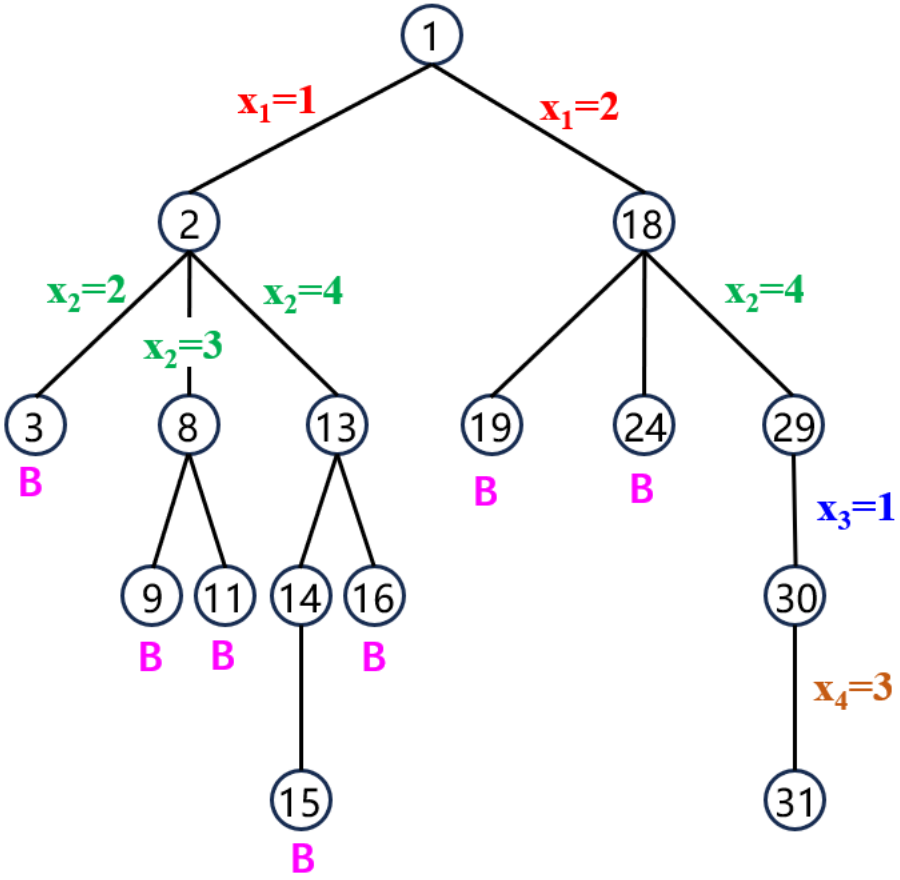
结点29变成 *E* 结点。

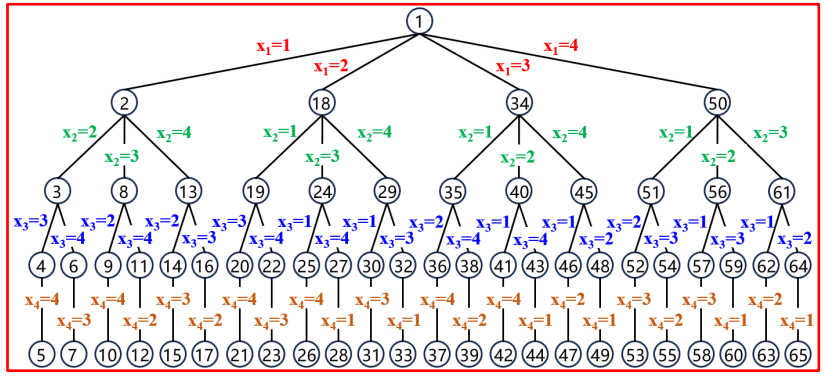
扩展结点29生成结点30，即皇后3放到第 3 行第 1 列。

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。

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。

$T(x_1, x_2, \ldots, x_{i-1})$ 是这样的所有结点 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（有相互攻击的皇后），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 都选自集合 $T(x_1, x_2, \ldots, x_{i-1})$ 且使 B_i 为真，即**合法取值且可行的 n 元组**。

回溯法求解的基本步骤

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（组织成**状态空间树**）。

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**。
- 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**（注：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，那么便**返回**到**最近被考察的活结点**（**回溯**），这个活结点再次成为 ***E*-结点**。
- 当找到答案结点或者检测完所有活结点后，搜索过程结束。

回溯法的一般框架

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

```
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  //处理“下一层次” （深度发展）//
        else
              k← k-1   //回到“上一层次”//
        endif
    repeat
end BACKTRACK
```

合法取值

可行

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
                  合法取值                    可行
        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
```

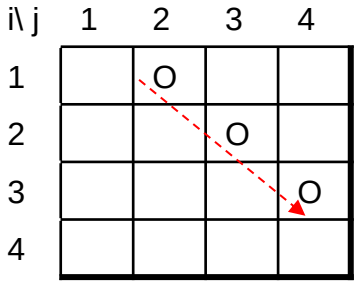
- 说明：当 ***k* > *n*** 时，***T*(*X*(1),…*X*(*k*-1))** 返回一个空集，算法不再进入for循环。
- 算法输出所有的解，元组大小可变。

22.2 n-皇后问题　　N 元组表示：$(x_1, x_2, \ldots, x_n)$

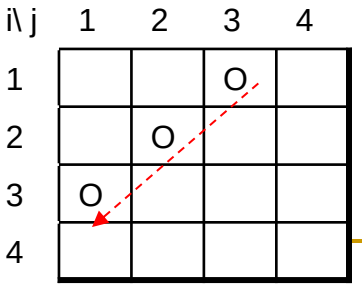
- 怎么判断是否形成了互相攻击的棋局？
  - **不在同一行上**：**约定不同的皇后在不同的行，自然解决**
  - **不在同一列上**：$x_i \neq x_j$，$(i, j \in [1:n])$
  - **不在同一条斜角线上**：**如何判定?**

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

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



左上方→右下方
相同的"**行 - 列**"值
1-2 = 2-3 = 3-4



右上方→左下方
相同的"**行 + 列**"值
1 + 3 = 2 + 2 = 3 + 1

处在对角线上的判别条件：

假设两个皇后被放置在 (i, j) 和 (k, l) 位置上，则仅当：

$$i - j = k - l \text{ 或 } i + j = k + l$$

时，它们在同一条斜角线上。

即：$i - k = j - l$ 或 $i - k = l - j$

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

下面的过程 **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——**即满足下述条件**：

① 不等于前面 $X(1)$，…，$X(k\text{-}1)$ 中的任何一个（不能在同一行），且

② 不能与前面的 $k\text{-}1$ 个皇后中的任一个在同一斜角线上。

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
```

$|i-k| = |j-l|$

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);
    k←1; X(1) ← 0;     // k是当前行，X(k) 是当前列，X(k)=0 表示先将 k 皇后放在棋盘左外侧//
    while k > 0 do          //对所有的行执行以下语句//
        X(k) ←X(k)+1       //尝试移到下一列//
        while X(k) ≤ n and not PLACE(k) do      //检查是否能放置皇后//
            X(k) ←X(k)+1                        //若当前的 X(k) 列不能放置，看下一列//
        repeat
        if X(k) ≤ n          //在 1~n 范围内找到一个可以放置的位置//
            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，测试下一皇后， 置新的 X(k)=0 ， 将新
                   endif                        皇后先放在棋盘左外侧//
        else
            k ← k - 1   //对当前 k，所有的列都测试完了，还未找到解，就回到上一级
        endif
    repeat
end NQUEENS
```

合法取值

22.3 子集和数问题

- **元组表示**：采用大小固定的 n 元组形式 $(x_1, x_2, \ldots, x_n)$，$x_i = 1$ 或 0。
- **结　　点**：对于 $i+1$ 级上的一个结点，如果为其父结点（i 级结点）的**左儿子**，则 $x_i = 1$；否则为**右儿子**时 $x_i = 0$。
- **限界函数的定义**：

约　定：$W(i)$ **按非降次序排列**：$W(1) \leq \ldots \leq W(i) \leq \ldots W(n)$

条件一：$\sum_{i=1}^{k} W(i)X(i) + \sum_{i=k+1}^{n} W(i) \geq M$ ← 总的来看还会存在解

条件二：$\sum_{i=1}^{k} W(i)X(i) + W(k+1) \leq M$ ← 下一个元素还可以考虑

仅当满足上述两个条件时，限界函数 $B(X(1), \ldots, X(k+1)) = true$.

注：如果不满足上述条件，则 $X(1),\ldots, X(k+1)$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，

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

$$W(1)\le M,\ \sum_{i=1}^{n}W(i)\ge M\ //$$

```
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     //从 k-1 级的一个结点进来时， 已确保B_k= true, s+W(k) ≤ M
    if  s+W(k) = M then                          //找到了答案（该位置是一个答案结点）//
          print(X(j),  j←1 to k)                 //输出答案//
    else   if s + W(k) + W(k+1) <= M then        //否则，尝试生成 k 级的左儿子，但要确保 B_{k+1} = true//
              call SUMOFSUB ( s + W(k), k + 1, r - W(k) )
          endif
    endif
    //然后尝试生成右儿子，同样要确保 B_{k+1} 的值为真//
    if  s + r - W(k) >= M and s + W(k+1) <= M     //此处不选 k，继续测试 k+1，要确保 B_{k+1} = true//
          then   X(k) ← 0
                 call SUMOFSUB (s, k + 1, r - W(k) )
    endif
end SUMOFSUB
```

k+1：下一步要测试的层级

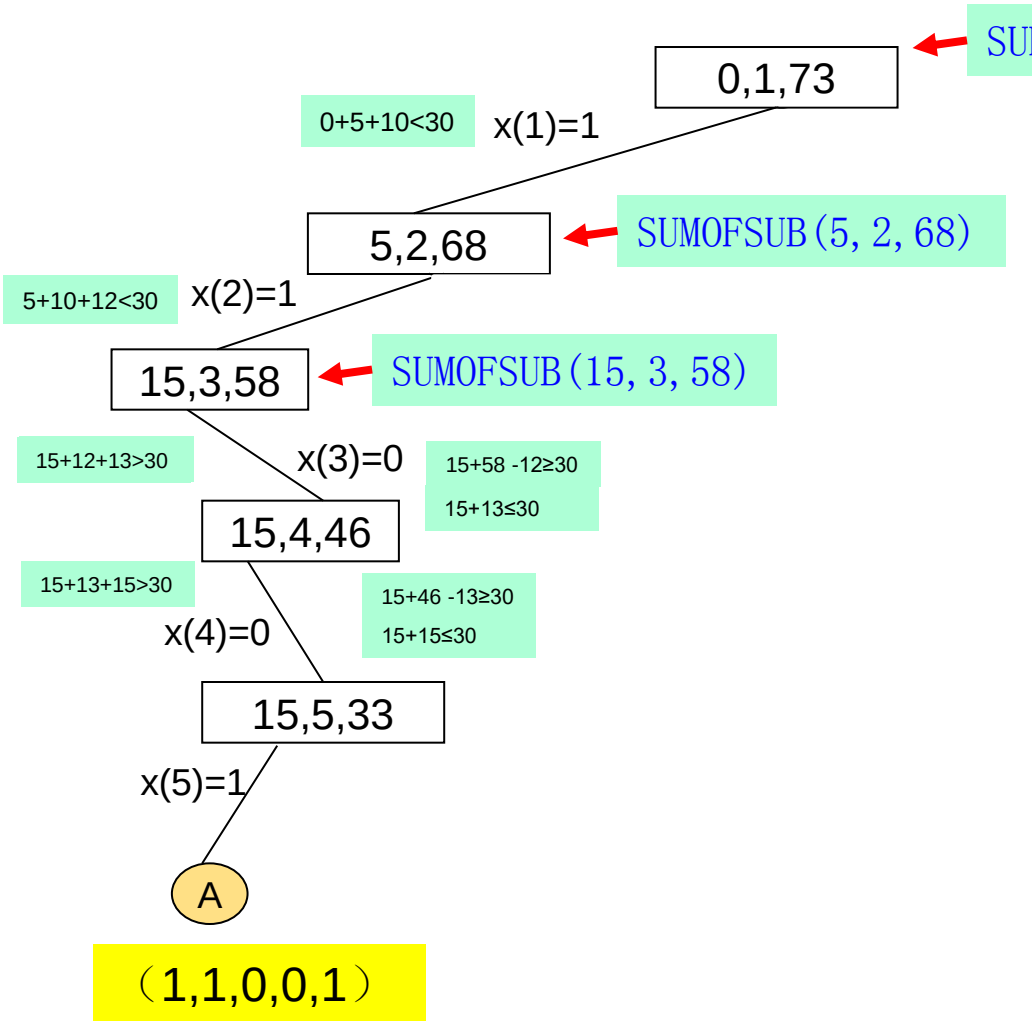
$$\sum_{i=1}^{k}W(i)X(i)+\sum_{i=k+1}^{n}W(i)\ge M$$

$$\sum_{i=1}^{k}W(i)X(i)+W(k+1)\le M$$

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

SUMOFSUB的一个实例

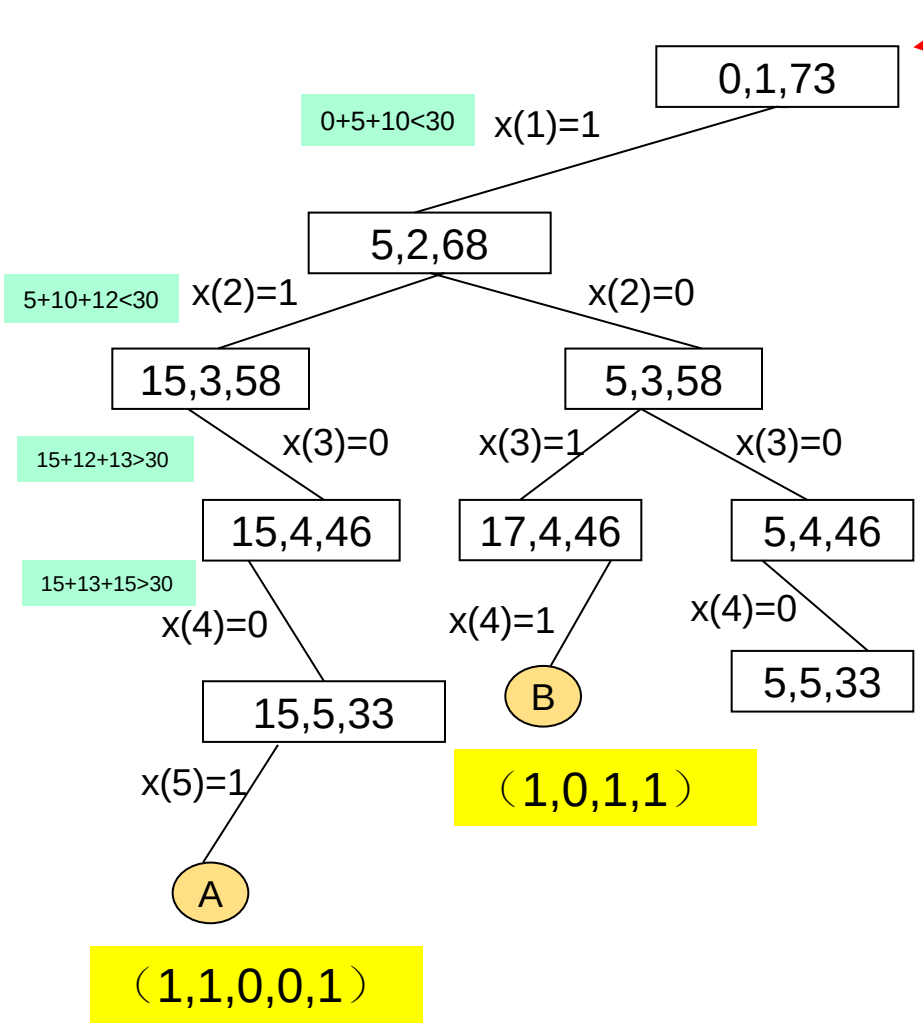
- $n = 6, M = 30, W(1:6) = (5, 10, 12, 13, 15, 18)$
- 方形结点：s，k，r
- 圆形结点：输出答案，共生成20个结点




```
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，Bk-1//
    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Bk的值//
    if  s+r-W(k)>=M and s+W(k+1)<=M     //确保Bk=true//
          then   X(k)←0
                 call SUMOFSUB(s,k+1,r-W(k))
    endif
end SUMOFSUB
```

SUMOFSUB的一个实例

- $n = 6, M = 30, W(1:6) = (5, 10, 12, 13, 15, 18)$




```
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，Bk-1//
    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Bk的值//
    if  s+r-W(k)>=M and s+W(k+1)<=M         //确保Bk=true//
          then   X(k)←0
                 call SUMOFSUB(s,k+1,r-W(k))
    endif
end SUMOFSUB
```

SUMOFSUB的一个实例

- $n = 6, M = 30, W(1:6) = (5, 10, 12, 13, 15, 18)$