算法设计与分析

Computer Algorithm Design & Analysis

2020.10

群名称：算法设计与分析2020
群 号：271069522

吕志鹏
zhipeng.lv@hust.edu.cn

群名称：算法设计与分析
群 号： 271069522

Chapter 22
Elementary Graph Algorithms

基本的图算法

图算法

图论问题渗透整个计算机科学，图算法对于计算机学科至关重要。

- 很多计算问题可以归约为图论问题。
- 本部分对图论中比较重要的一些问题进行讨论。

基本约定和表述：

一个图G通常表示为G=(V,E)，其中

- V：G中结点的集合，|V|表示结点数。
- E：G中边的集合，|E|表示边数。
- 通常用|V|和|E|两个参数表示算法输入的规模。
  - 在渐近记号中，通常用V代表|V|、用E代表|E|，如O(VE)。
  - 在算法中，用G.V表示图G的结点集，G.E 表示图G的边集。

22.1 图的表示

对于图G=(V,E)，可以用两种方法表示：邻接表、邻接矩阵。

1. 邻接表

对图G=(V,E)而言，其邻接表是一个包含|V|条链表的数组Adj：

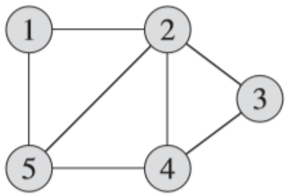
- 在Adj中，每个结点u∈V有一条链表Adj[u]，包含所有与结点u之间有边相连的结点v。
- 通常用G.Adj[u]表示结点u在邻接表Adj中的邻接链表。

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)。

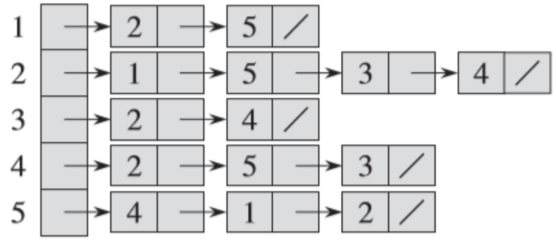
- 对于有向图，所有邻接链表的长度之和等于|E|；
- 对于无向图，所有邻接链表的长度之和等于2|E|。

例

一个无向图的邻接表表示如下：



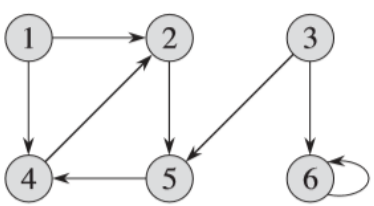
(a)



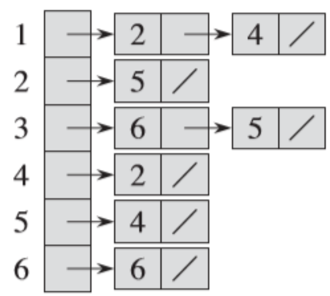
(b)

一个有5个结点和7条边的无向图G　　G的邻接表表示，每条边被保存了两次

一个有向图的邻接表表示如下：



(a)



(b)

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

通常将图G中的结点编号为1, 2, …, |V|。图G的邻接矩阵是一个|V|╳|V|的矩阵A=(a_{ij})，并定义为：

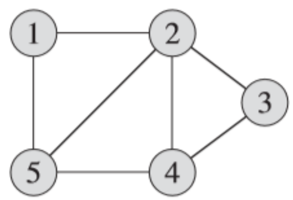
$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E , \\ 0 & \text{otherwise} . \end{cases}$$

邻接矩阵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^2)。

- 无向图的邻接矩阵A是一个对称矩阵，因此A也是自己的转置，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，可以省近乎一半的空间。
  - 但以上性质不适用于有向图。

例

一个无向图的邻接矩阵表示如下：



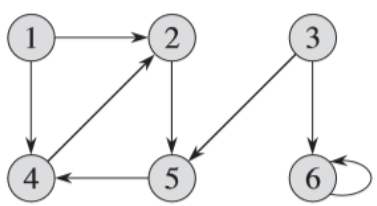
(a)

$$
\begin{array}{c|ccccc|}
 & 1 & 2 & 3 & 4 & 5 \\
\hline
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 1 & 1 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 1 & 0 & 1 \\
5 & 1 & 1 & 0 & 1 & 0 \\
\hline
\end{array}
$$

(c)

一个有5个结点和7条边的无向图G　　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)

$$
\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
\hline
1 & 0 & 1 & 0 & 1 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 0 \\
3 & 0 & 0 & 0 & 0 & 1 & 1 \\
4 & 0 & 1 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
\end{array}
$$

(c)

一个有6个结点和8条边的有向图G　　G的邻接矩阵表示，非对称结构

权重图

权重图：图中的每条边都带有一个权重的图。

- 权重值通常以**权重函数** $\omega:E\to R$ 给出。

- 用**邻接表**表示权重图
  - 将边$(u,v)\in E$的权重值$\omega(u,v)$存放在u的邻接链表结点中，作为其属性。

- 用**邻接矩阵**表示权重图
  - 对于边$(u,v)\in E$，令邻接矩阵$A[u][v] = \omega(u,v)$。
  - 若(u,v)不是E中的边，则令$A[u][v] = NIL$，或∞、0。

注:

- 稀疏图一般用邻接表表示
  - 稀疏图：边数|E|远小于$|V|^2$的图。
- 稠密图更倾向于用邻接矩阵表示
  - 稠密图：边数|E|接近$|V|^2$的图。
- 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。

22.2 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，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。

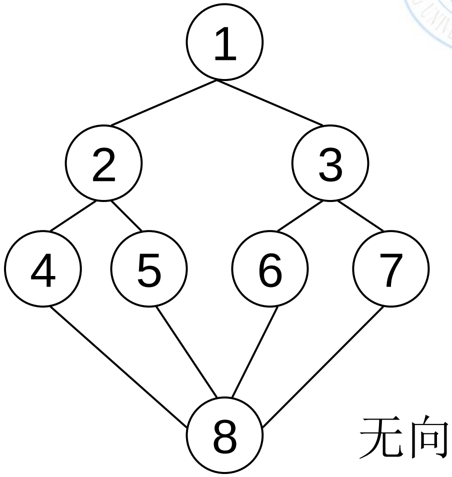
③ 若未检测结点表为空，则算法终止；否则

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，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	

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	

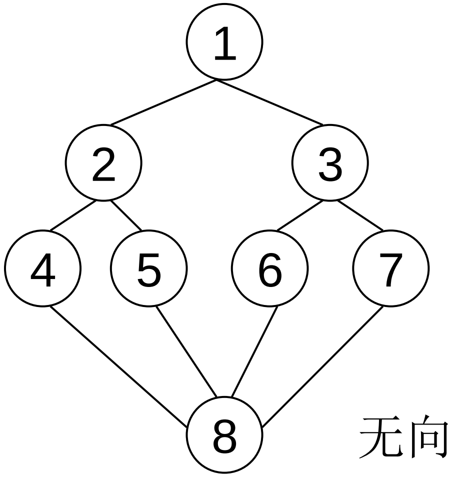
检测结点7（结点7出队列）：

队列状态：

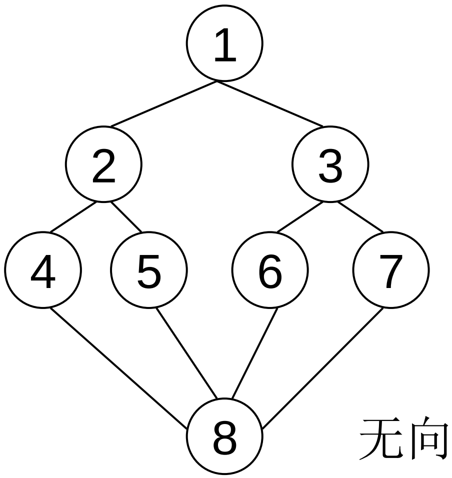
8	

检测结点8（结点8出队列）：

队列状态：



无向图G



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：**用数学归纳法**证明

设G=(V,E)是一个(有向或无向)图，v∈V。

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。

⑴ 若d(v,w)≤1，则显然所有这样的w都将被访问。

⑵ 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的流程，在u被访问时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∈V且w≠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：$\mathbf{O(n^2)}$

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。

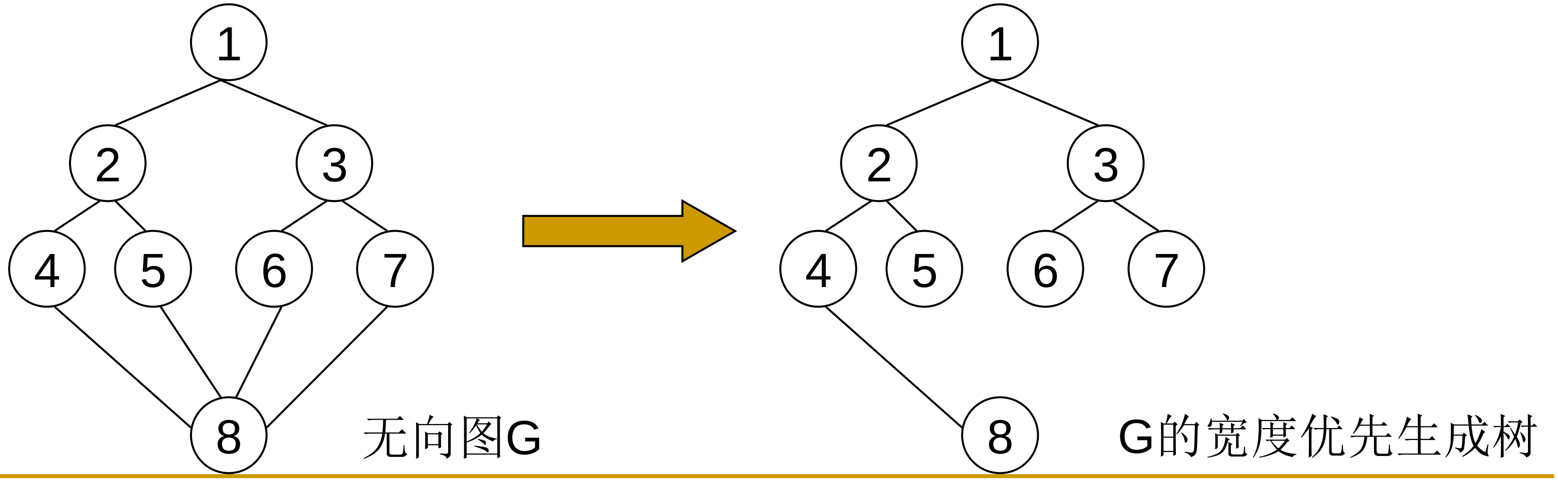
● 宽度优先生成树

向前边：BFS中由u到达未访问结点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**。



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*。

```
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*
```

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**。

证明：

若G是n个结点的连通图，则这n个结点都要被访问，所以除起始点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

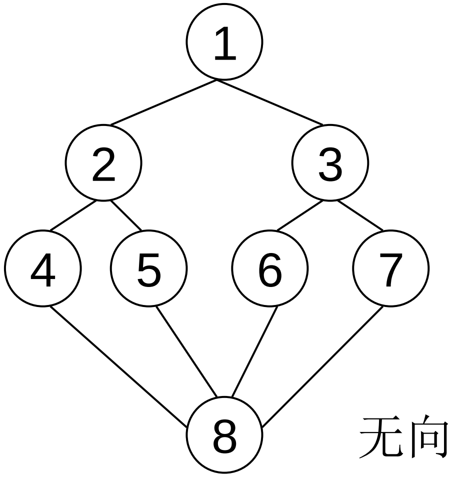
(1) 深度优先检索

从结点v开始，首先访问v，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n结点的图G＝(V,E)以及初值置零的数组VISITED(1:n)。//
    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

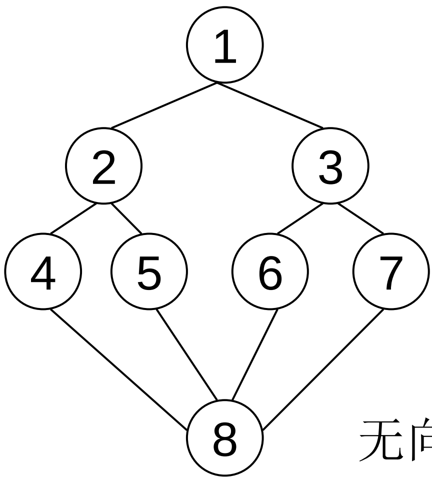
② 连通分图

③ 无向图的自反传递闭包矩阵

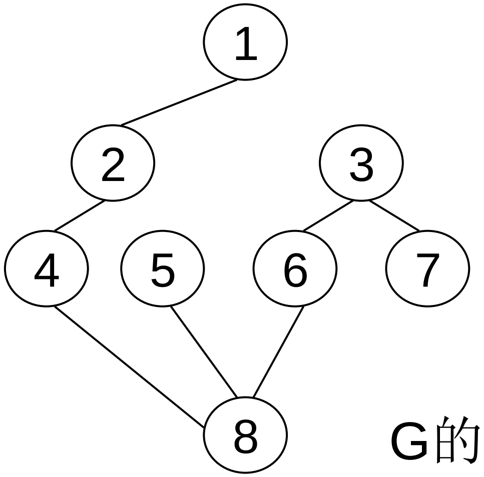
④ 深度优先生成树

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*
```

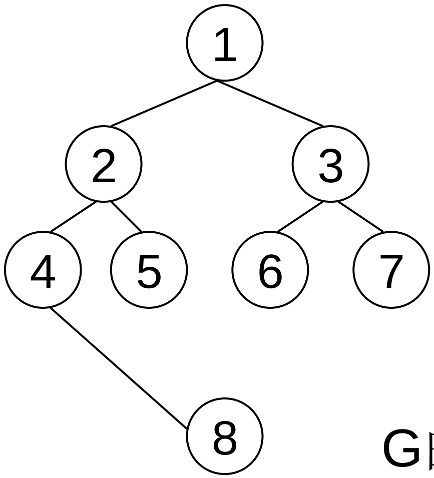


无向图G



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

G的深度优先生成树



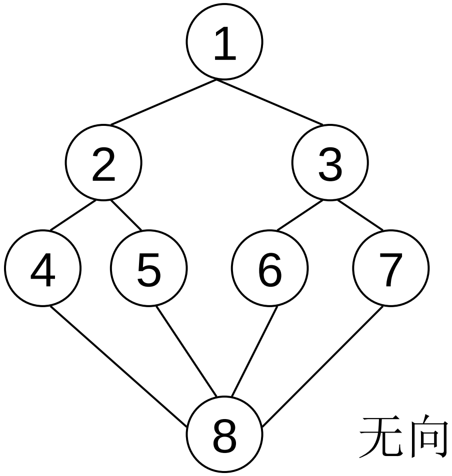
1,2,3,4,5,6,7,8

G的宽度优先生成树

3. D_Search：深度检索

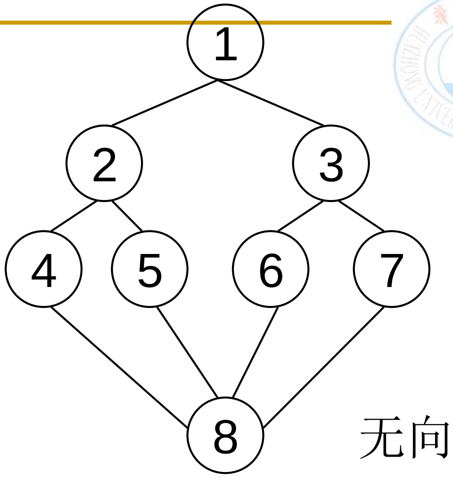
改造BFS算法，**用栈来保存未被检测的结点**，则得到的新的检索算法称为深度检索（D_Search）算法。

注：结点被压入栈中后将以相反的次序出栈。



无向图G

例：

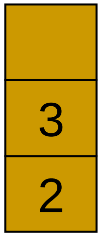


无向图G

检测结点1:

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

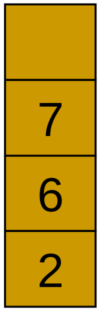
栈状态：



检测结点3（结点3出栈）：

visited(6) =1、Visited(7)=1

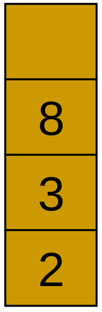
栈状态：



检测结点7（结点7出栈）：

visited(8) =1

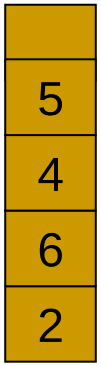
栈状态：

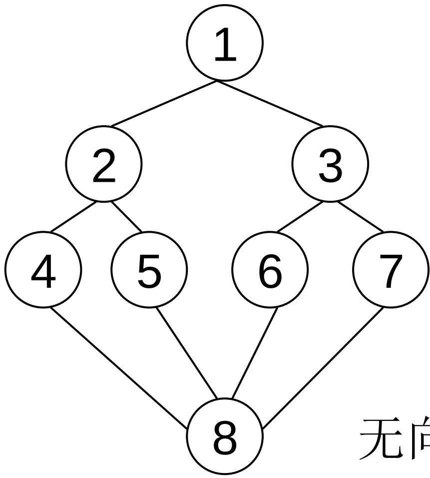


检测结点8（结点8出栈）：

visited(4) =1、 visited(5) =1

栈状态：

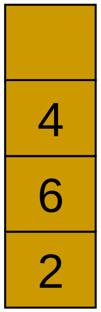




无向图G

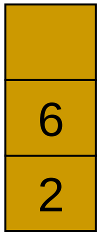
检测结点5（结点5出栈）：

栈状态：



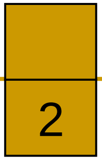
检测结点4（结点4出栈）：

栈状态：



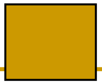
检测结点6（结点6出栈）：

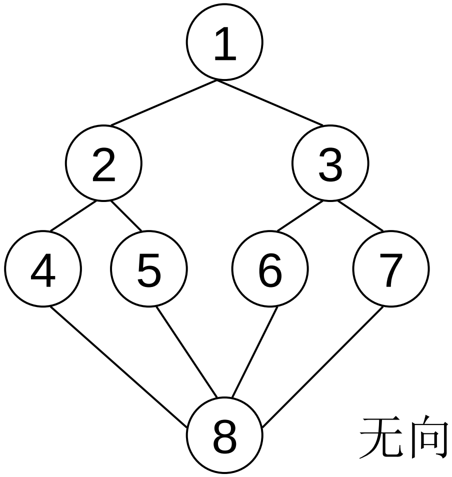
栈状态：



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

五大常用算法设计策略：

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

22.3 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($x_1,\ldots,x_n$)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($x_1,\ldots,x_n$)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）暴力搜索法(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

回溯或分支限界法

回溯（分支限界）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**，这使得相对于暴力搜索大大减少了计算量。

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：用来规定解空间中那些满足规范函数的元组，

- 隐式约束条件描述了x_i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，使得任意两个皇后之间都不互相“攻击”，即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：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$W=\{w_1, w_2, \ldots, w_n\}$和正数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**$(x_1, x_2, \ldots, x_k)$, $1 \le k \le n$。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$1 \le j \le n \}$。

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）$x_i < x_{i+1}, 1 \le i < n$（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，从而需要有效地组织问题的解空间。

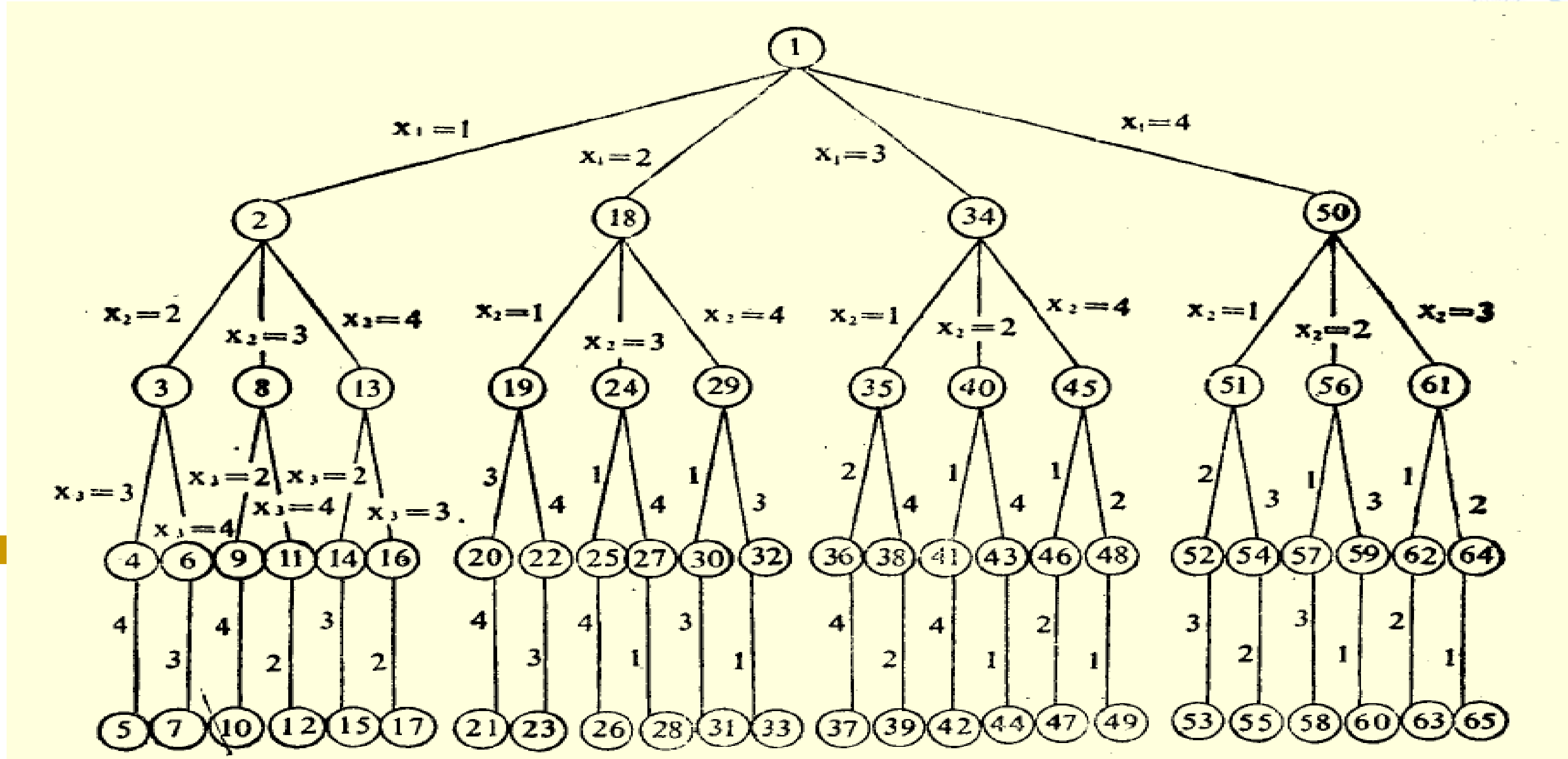
采用何种形式组织问题的解空间？

可以用**树结构**组织解空间，形成状态空间树。

例 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。

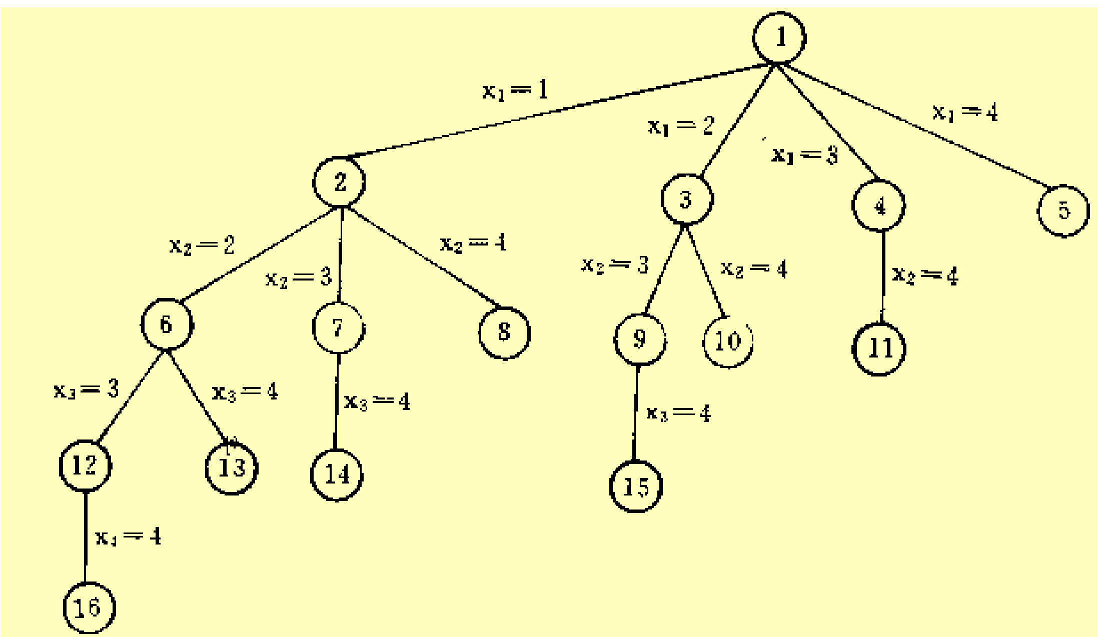
共有4！＝24个叶结点，反映了4元组的所有可能排列——称为**排列树**。

例 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**（$x_i < x_{i+1}$）

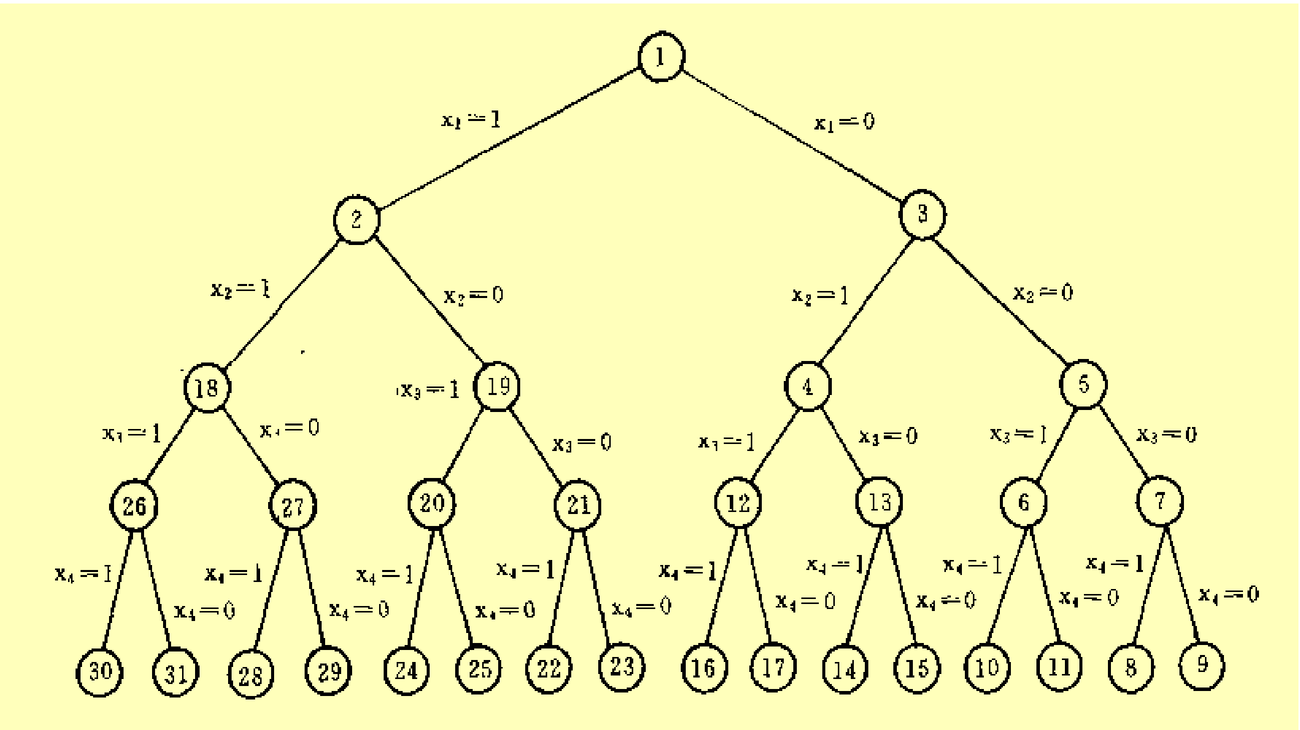
树边标记：由i级结点到i+1级结点的一条边用x_i来表示，表示k-元组里的第i个元素是已知集合中下标为x_i的元素。



解空间由树中的根结点到任何结点的所有路径所确定，包括：(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。

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**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。（静态）
- **E-结点**（expansion node）：当前正在生成其儿子结点的活结点。（动态）
- **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

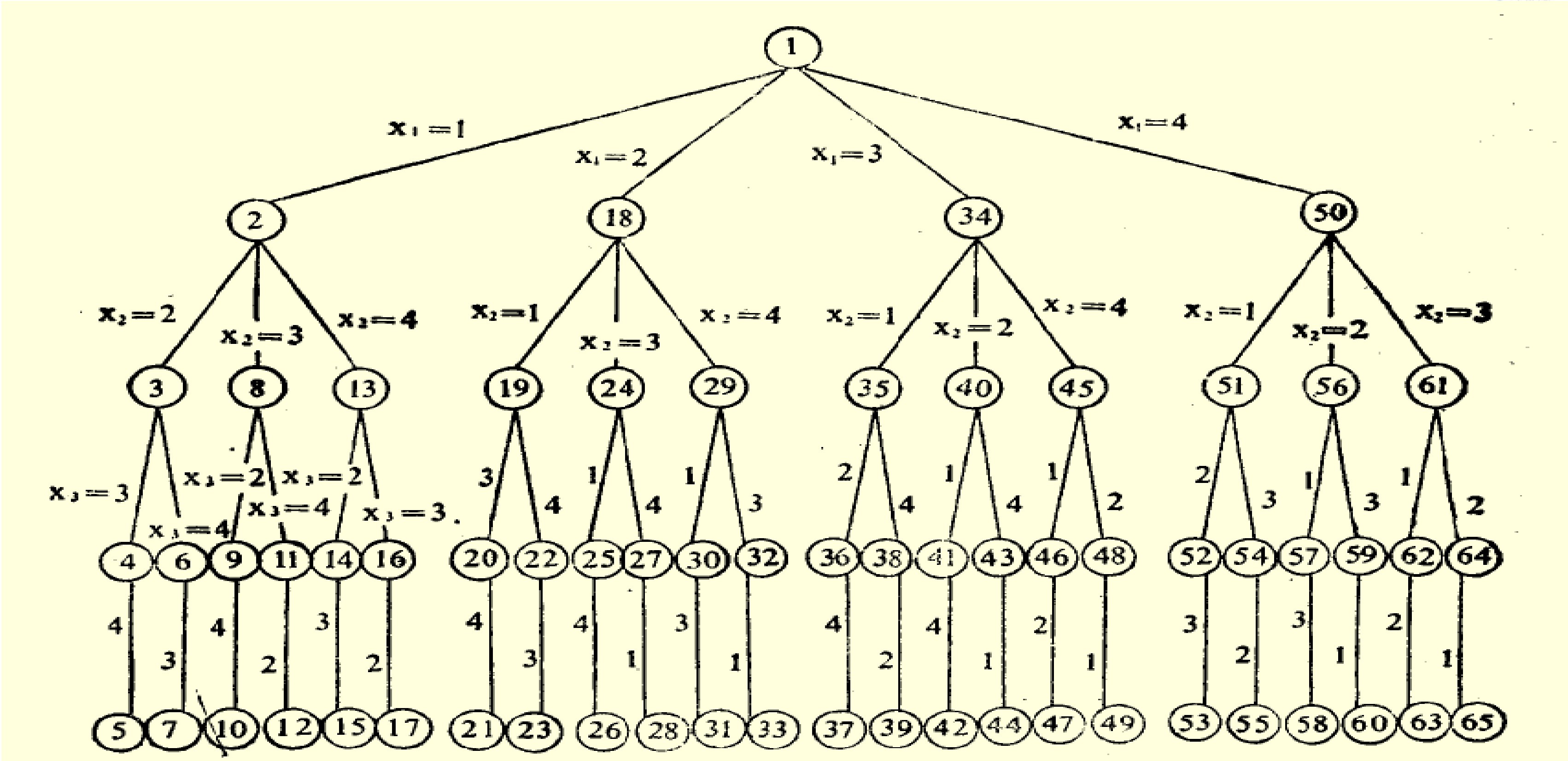
1. 深度优先策略：当E-结点R一旦生成一个新的儿子C时，C就变成一个新的E-结点，当完全检测了子树C之后，R结点再次成为E-结点。

2. 宽度优先策略：一个E-结点一直保持到变成死结点为止。

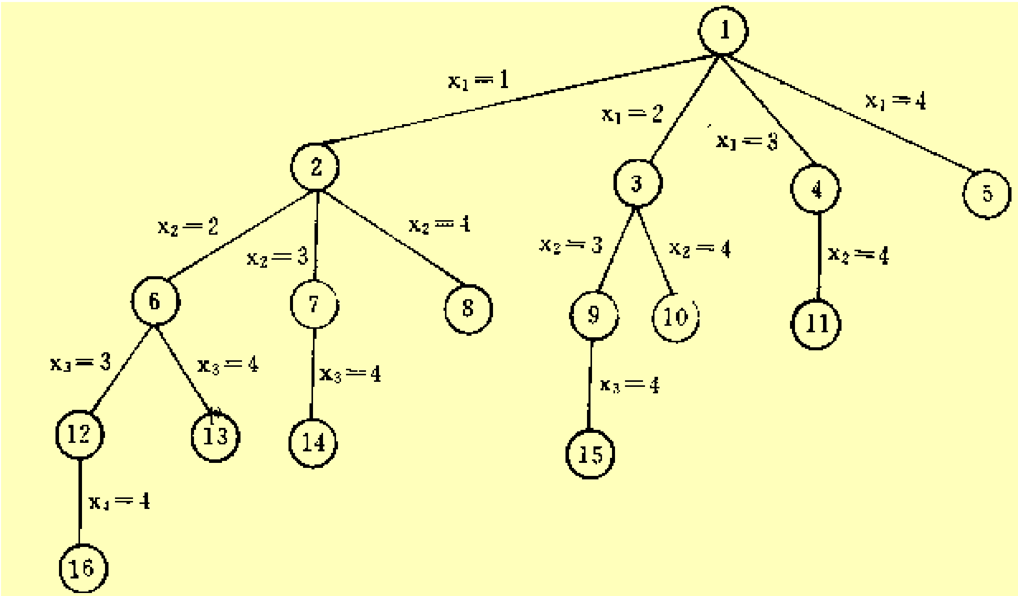
限界函数：在结点生成的过程中，定义一个限界函数，用来杀死还没有生成全部儿子结点的一些活结点

——这些活结点已无法满足限界函数的条件，因此不可能导致问题的答案。

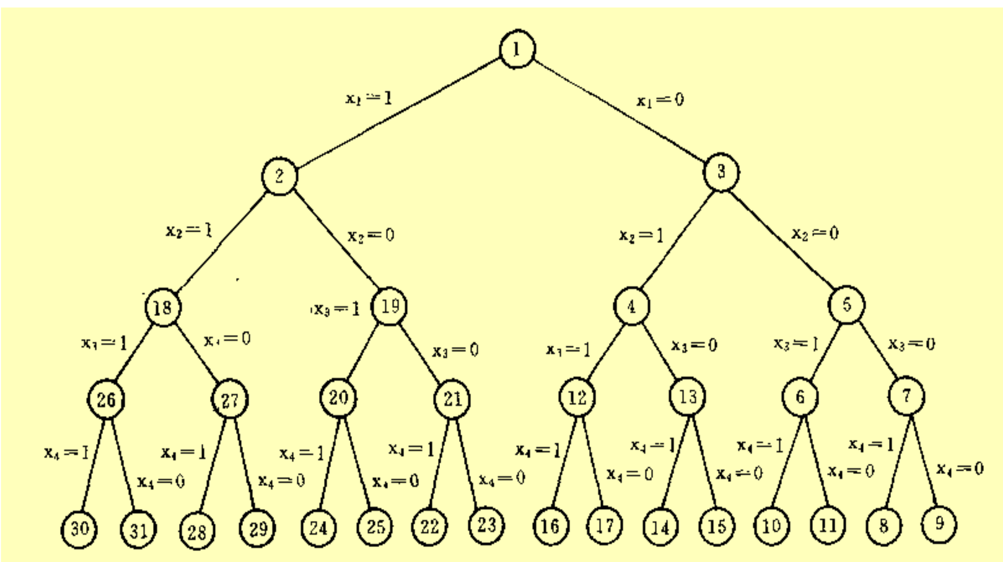
- 回溯法：使用限界函数的**深度优先**状态结点生成方法称为回溯法（backtracking）
- 分支-限界方法：使用限界函数的**E结点一直保持到死为止**的状态结点生成方法称为分支-限界方法（branch-and-bound）



- 深度优先策略下的结点生成次序（结点编号）（没有剪枝的情形）



利用**队列**的宽度优先策略下的结点生成次序(BFS)



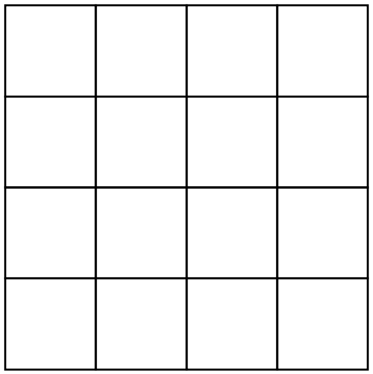
利用**栈**的宽度优先策略下的结点生成次序 (D-Search)

（没有剪枝的情形）

例：4-皇后问题的回溯法求解

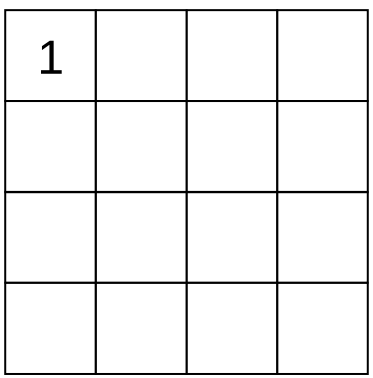
- **限界函数**：如果（$x_1,x_2,...,x_{i-1}$）是到当前E结点的路径，那么x_{i-1}的儿子结点x_i是一些这样的结点，它们使得（$x_1,x_2,...,x_{i-1},x_i$）表示没有两个皇后处在相互攻击状态的一种棋盘格局。
- **开始状态**：根结点1，此时表示棋盘为空，还没有放置任何皇后。
- **结点的生成**：依次考察皇后1——皇后n的位置。

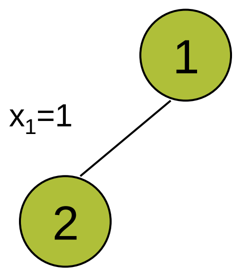
按照自然数递增的次序生成4皇后问题状态空间树中结点的儿子结点。



根结点1，开始状态，唯一的活结点

解向量：()

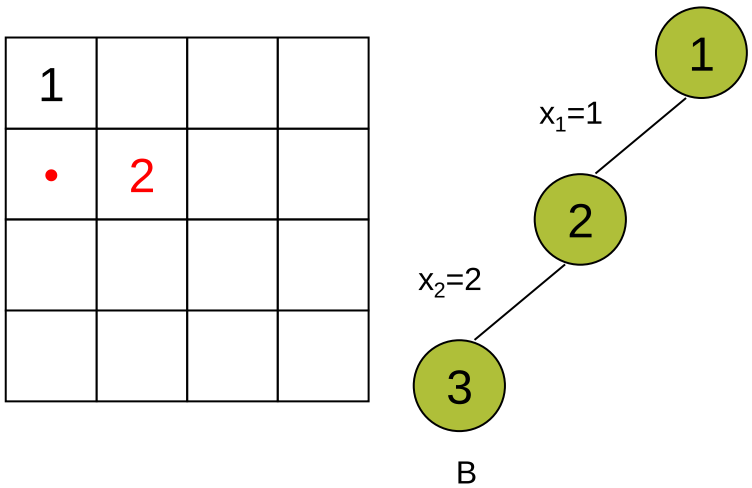




生成结点2，表示皇后1被放到第1行的第1列上，该结点是从根结点开始第一个被生成结点。

解向量：(1)

结点2变成新的E结点，下一步扩展结点2

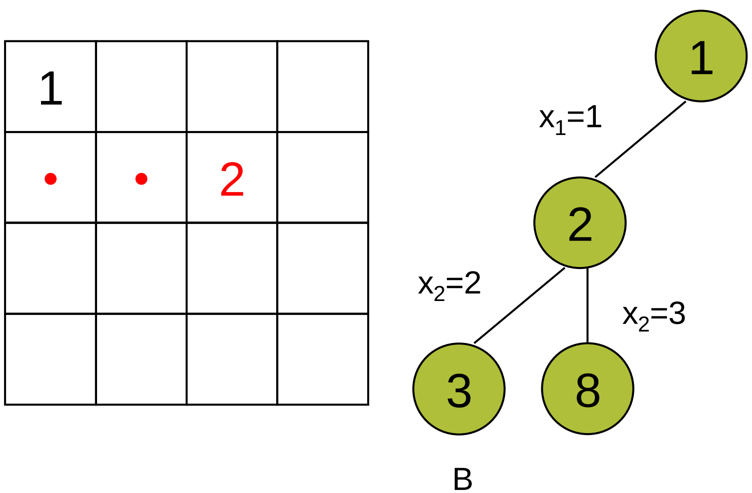


由结点2生成结点3，表示皇后2放到第2行第2列。

利用限界函数杀死结点3。

返回结点2继续扩展。

（3后面的结点4，5，6，7不会生成）

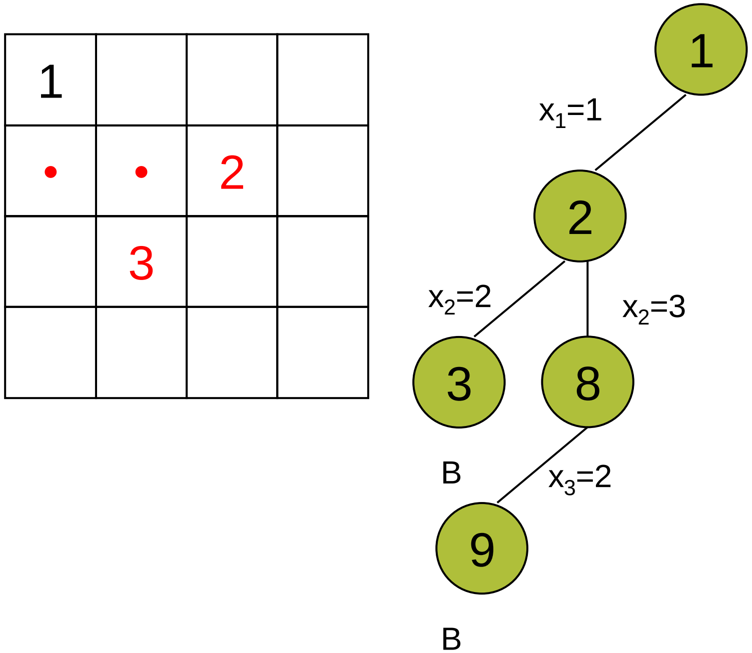


由结点2生成结点8，表示皇后2放到第2行第3列。

结点8变成新的E结点。

解向量：（1，3）

从结点8继续扩展。

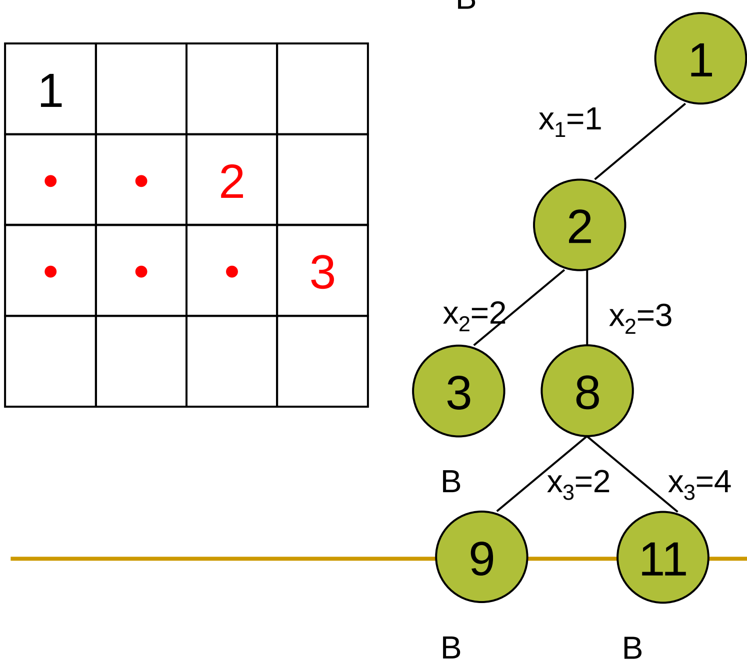


由结点8生成结点9，表示皇后3放到第3行第2列。

利用限界函数杀死结点9。

返回结点8继续扩展。

（9后面的结点10不会生成）



由结点8生成结点11，表示皇后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。

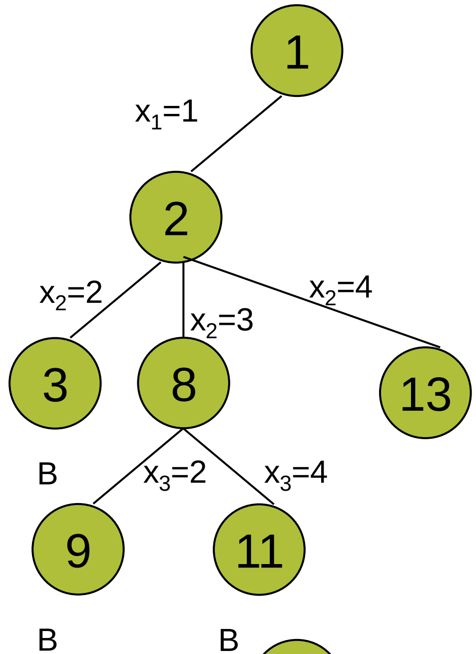
（11后面的结点12不会生成）

结点8的所有儿子已经生成，变成死结点，且没有找到答案结点。

结点8被杀死。

返回结点2继续扩展。

1			
•	•	•	2



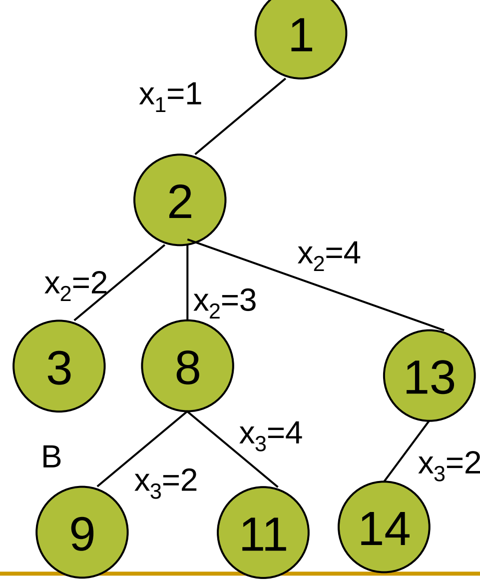
由结点2生成结点13，表示皇后2放到第2行第4列。

结点13变成新的E结点。

解向量：（1，4）

从结点13继续扩展。

1			
•	•	•	2
•	3		



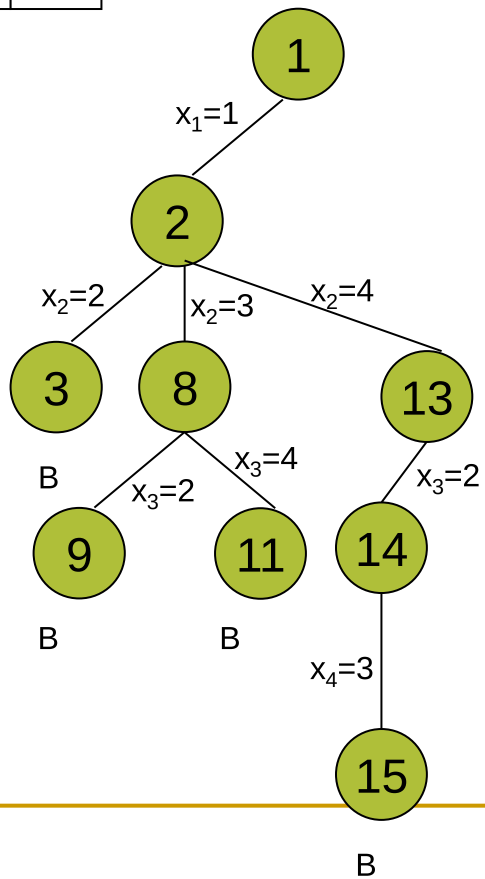
由结点13生成结点14，表示皇后3放到第3行第2列。

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	

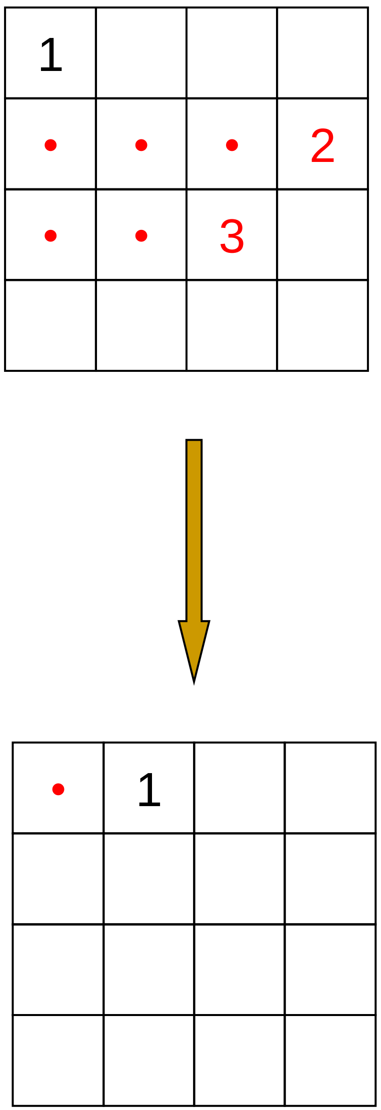


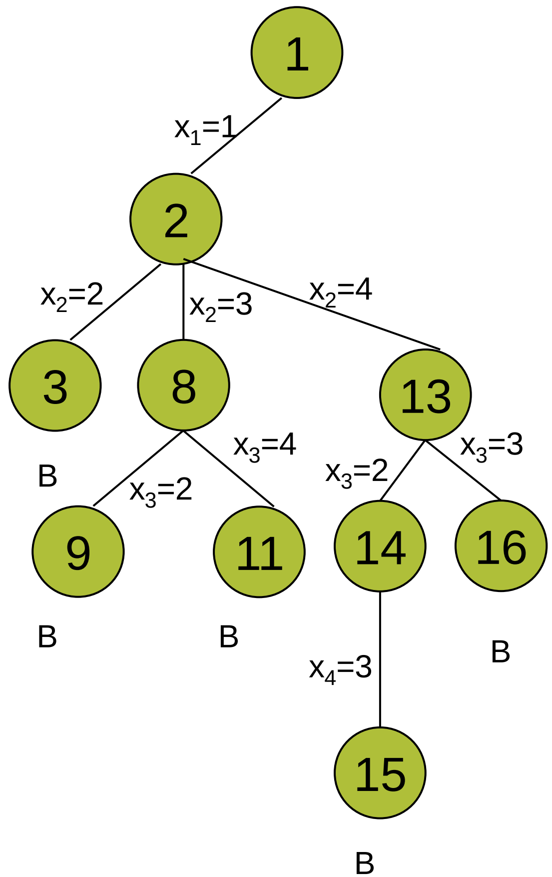
由结点14生成结点15，表示皇后4放到第4行第3列。

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，表示皇后3放到第3行第3列。

利用限界函数杀死结点16。

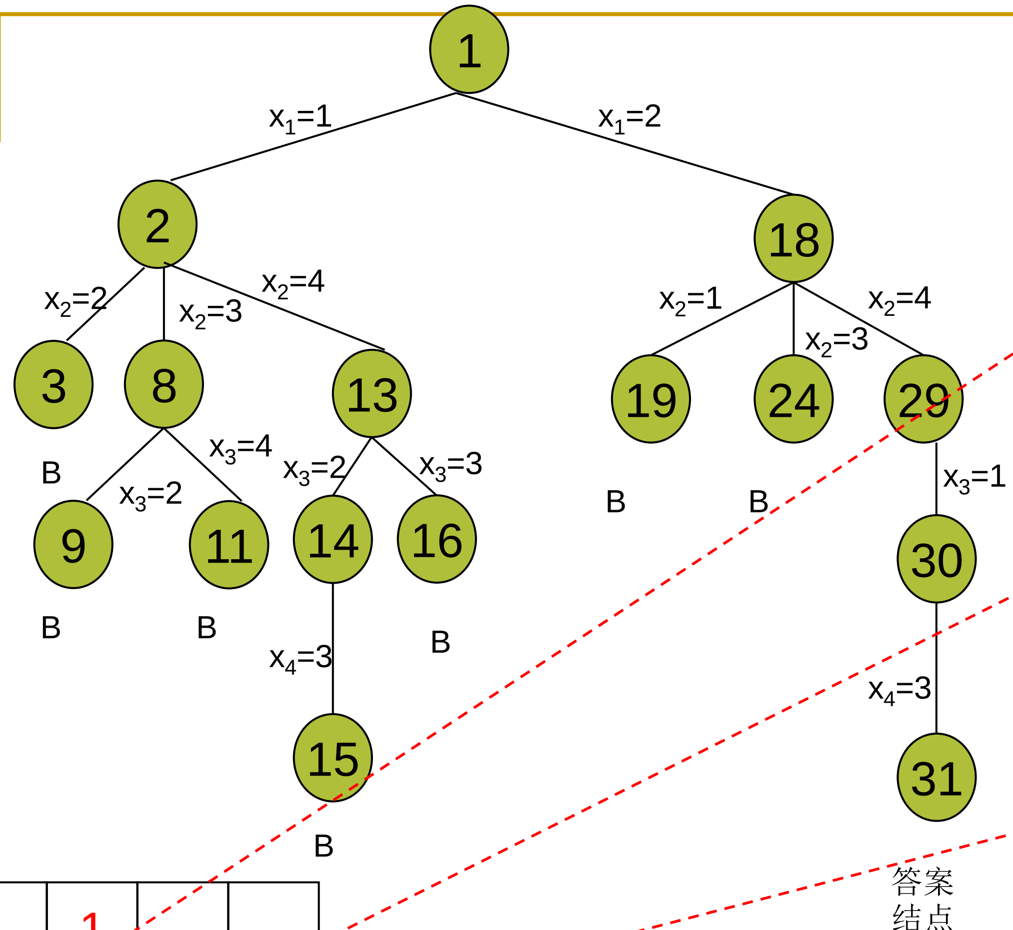
返回结点13，结点13不能导致答案结点，变成死结点，被杀死。

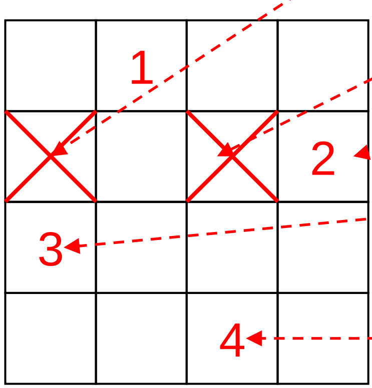
返回结点2继续扩展。

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。

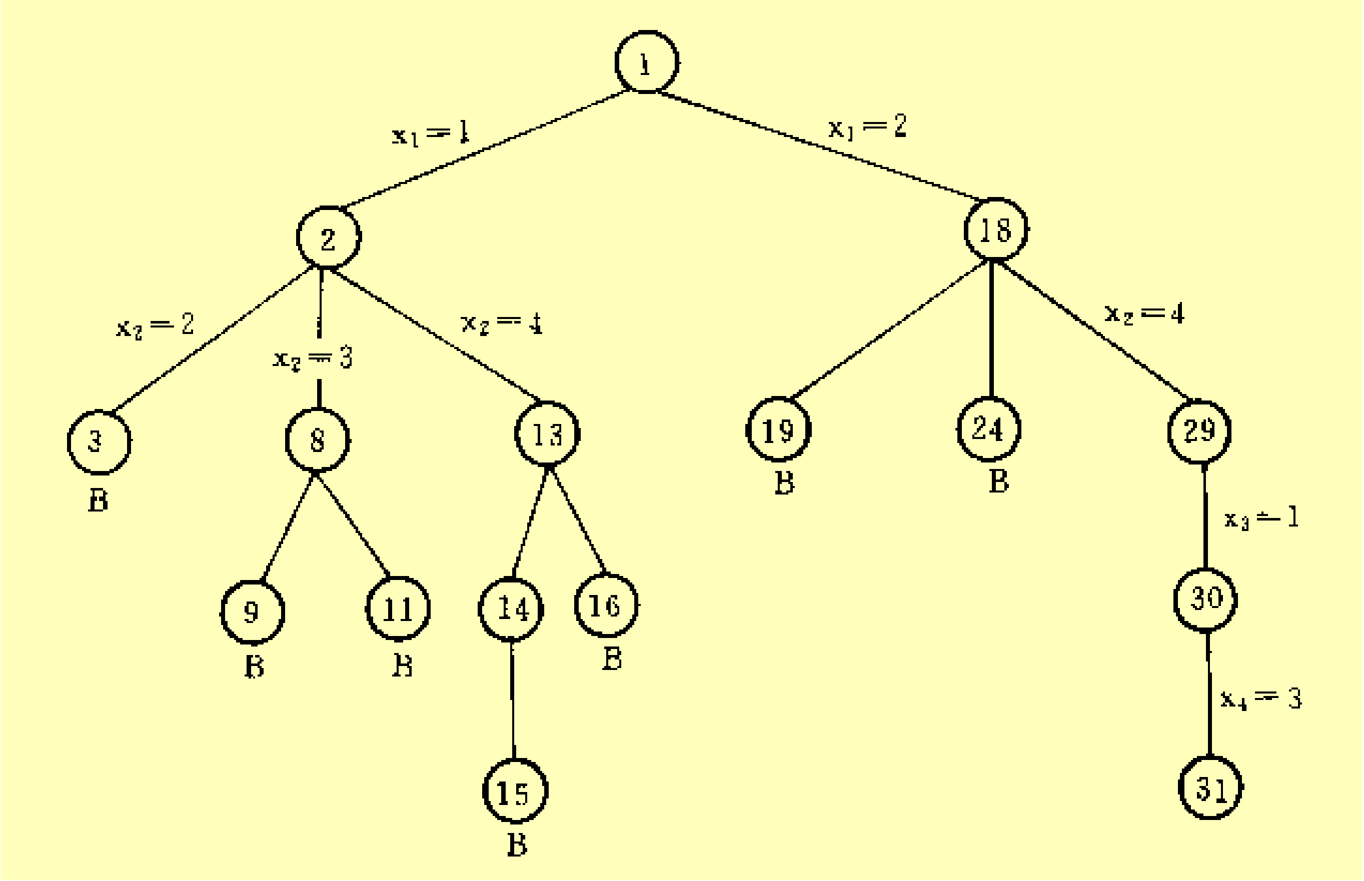
扩展结点30生成结点31，即皇后4放到第4行第3列。

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)。

用回溯法求解4-皇后问题所生成的树



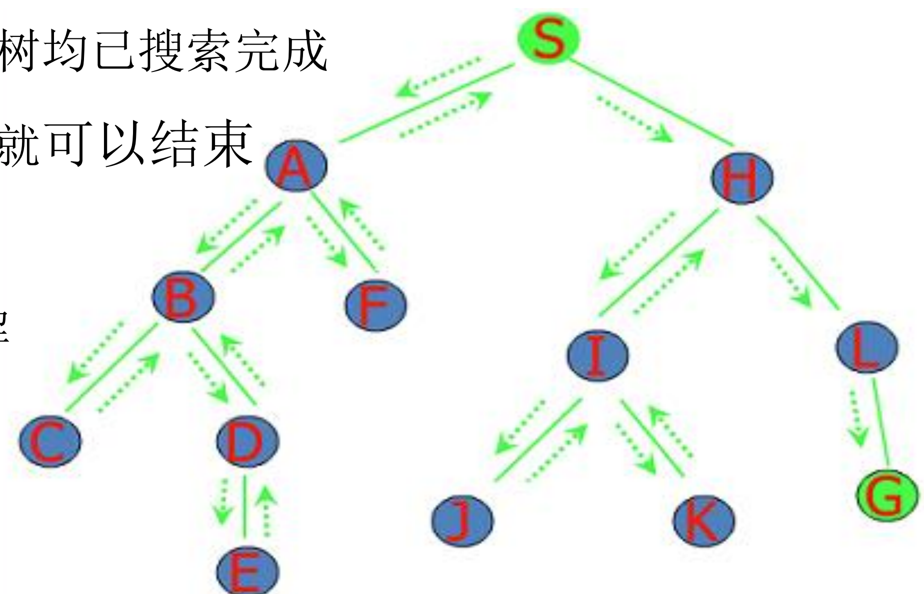
回溯算法的描述

- 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$x_i$。

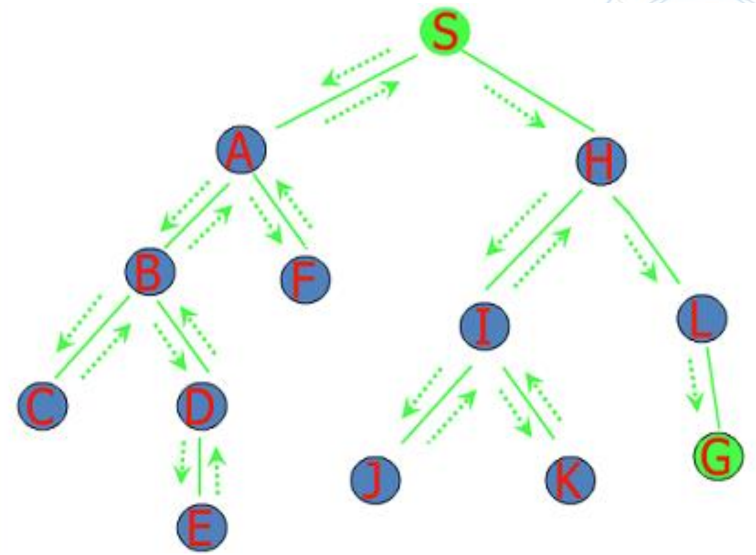
- 按深度优先的方法从开始结点进行搜索
  - 开始结点是第一个活结点，也是 E-结点。
  - 如果能从这个E-结点移动到一个新结点，那么这个新结点将变成活结点和新的E-结点 (旧的E-结点仍是一个活结点)。
  - 如果不能移到一个新结点，当前的E-结点就“死”了，然后**返回**到最近被考察的活结点（**回溯**），这个活结点重新变成E-结点。
  - 当找到了答案或者穷尽了所有的活结点时，搜索过程结束。

算法实现

- 1、按选优条件对T进行深度优先搜索，以达到目标。
- 2、从根结点出发深度优先搜索解空间树
- 3、当探索到某一结点时，要先判断该结点是否包含问题的解。
  - 如果包含，就从该结点出发继续按深度优先策略搜索；
  - 否则逐层向其祖先结点回溯（退回一步重新选择）
  - 满足回溯条件的某个状态的点称为“回溯点”
- 4、算法结束条件
  - 求所有解：回溯到根，且根的所有子树均已搜索完成
  - 求任一解：只要搜索到问题的一个解就可以结束
- 5、DFS搜索两种方式来实现:
  - **非递归方式:**思路更加清晰，便于理解
  - **递归方式:**代码更加简洁高效



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
       else  k ←k+1  //没到答案结点，考虑下一层结点，继续搜索//
    else
       k ←k-1   //所有的X(k)都不可行，回溯到上一层结点//
    endif
  repeat
end BACKTRACK
```

回溯算法的递归表示

- ◆ 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。

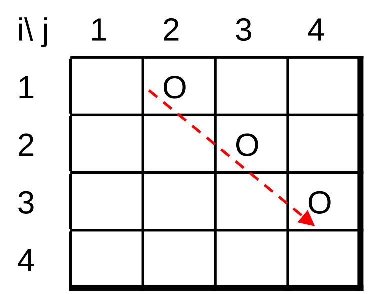
```
procedure RBACKTRACK(k)
    global n, X(1:n)
    if(k>n)  RETURN;
    for 满足下式的每个X(k)
        X(k) ∈T(X(1),…X(k-1)) and  B(X(1),…X(k))=true do
        if(X(1),…,X(k)) 是一条已抵达一答案结点的路径
            then   print(X(1),…,X(k))
        endif
        call RBACKTRACK(k+1)
    repeat
end RBACKTRACK
```

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。算法输出所有的解，元组大小可变。

n-皇后问题的求解

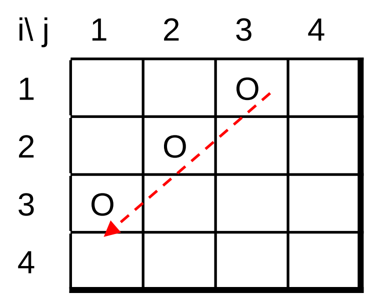
- n元组：（$x_1,x_2,...,x_n$)
- 怎么判断是否形成了互相攻击的格局?
  - 不在同一行上：约定不同的皇后在不同的行
  - 不在同一列上：$x_i \neq x_j$，（$i, j \in [1:n]$）
  - 不在同一条斜角线上：如何判定？

1）在由左上方到右下方的同一斜角线上的每一个元素有相同的“**行一列**”值。



左上方——右下方
相同的“行一列”值
1－2＝2－3＝3－4

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



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（第k行第X(k)列）——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
       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                                 //找到一个可以放置的位置//
          then  if  k=n                          //是一个完整的解吗？已到最后一行//
                   then print(X)                 //是，输出解向量//
                else                             //可放置，但还没到最后一个皇后//
                   k←k+1; X(k)←0                 //则转下一皇后，初始化该皇后位置//
                endif
       else
          k←k-1                                  //所有位置均无法放置，则退回到上一个皇后//
       endif
   repeat
end NQUEENS
```

子集和数问题的求解

- 元组大小固定：n元组（$x_1,x_2,...,x_n$），$x_i=1$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：$$\sum_{i=1}^{k} W(i)X(i)+\sum_{i=k+1}^{n} W(i)\ge M$$

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

仅当满足上述两个条件时，限界函数B(X(1),…X(k))=true

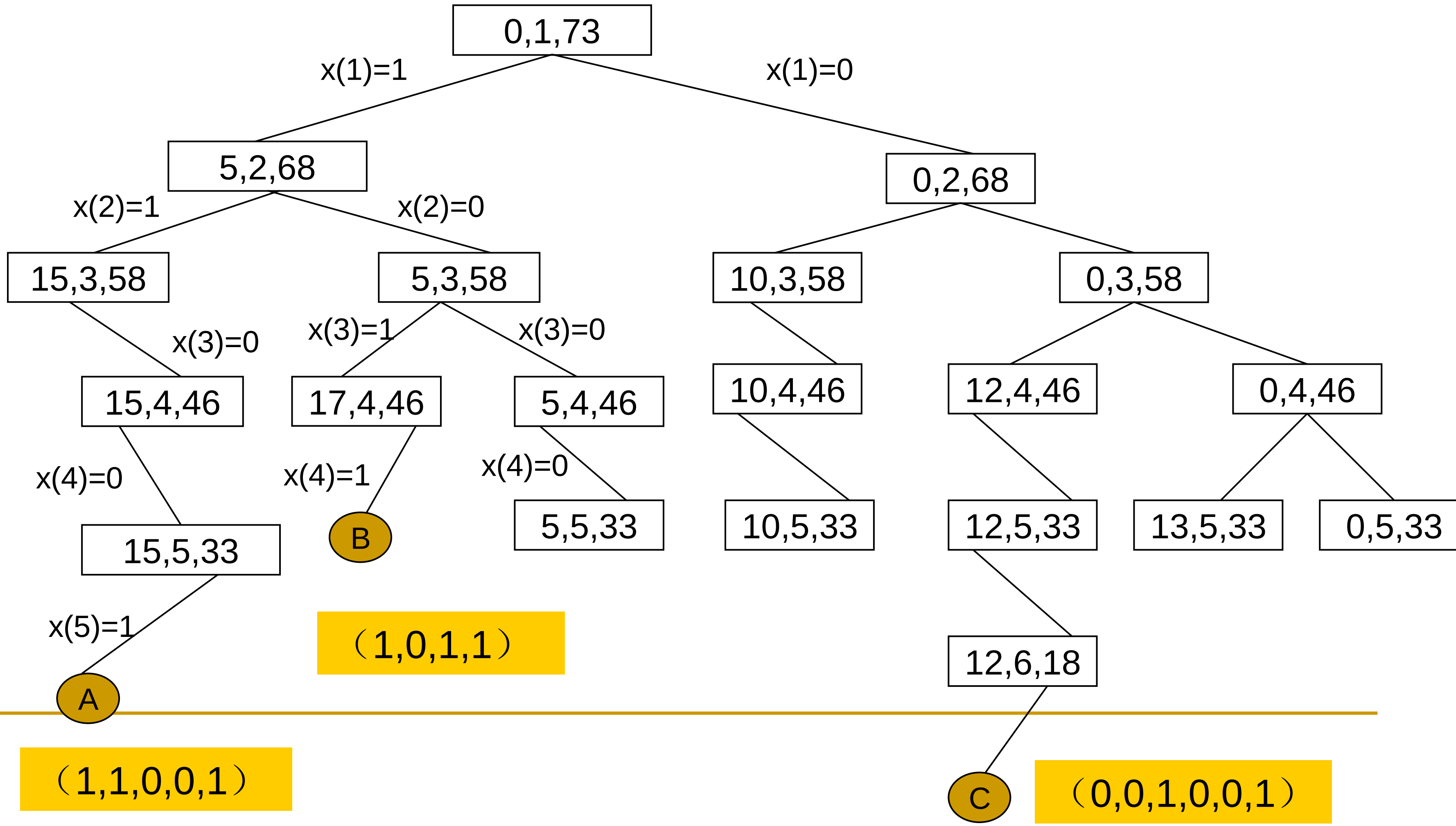
注：如果不满足上述条件，则X(1),…X(k)根本不可能导致一个答案结点。

子集和数的递归回溯算法

```
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 （s为已选元素之和，r为剩余所有元素之和，初始时s=0, r=所有元素之和）
    X(k)←1                                    //生成左儿子，Bk-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，算上W(k)，再加下一个数不会超过M//
                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，两个条件同时满足//
          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)
- 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
- **（每一层的k和r都是一样的，r表示剩余元素总和）**



- 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。