華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

算法设计与分析

刘渝
Liu_yu@hust.edu.cn

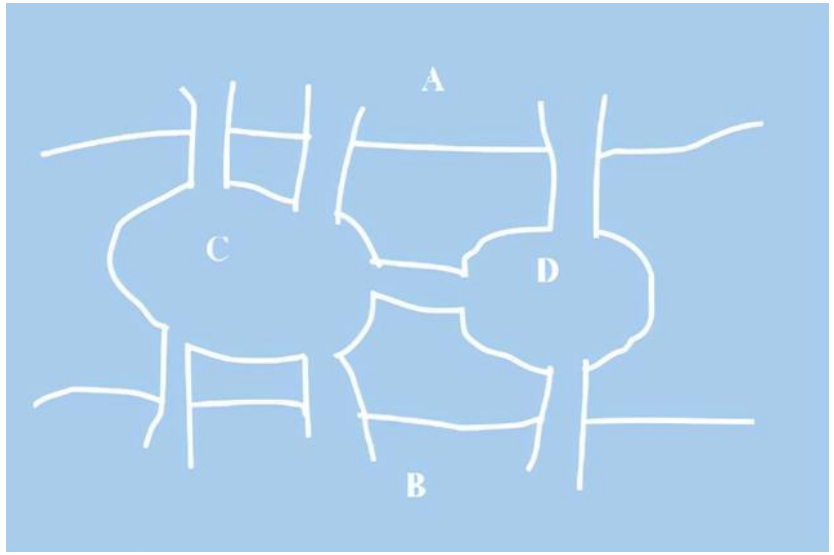
2025秋季-华科-计算机
24级CS

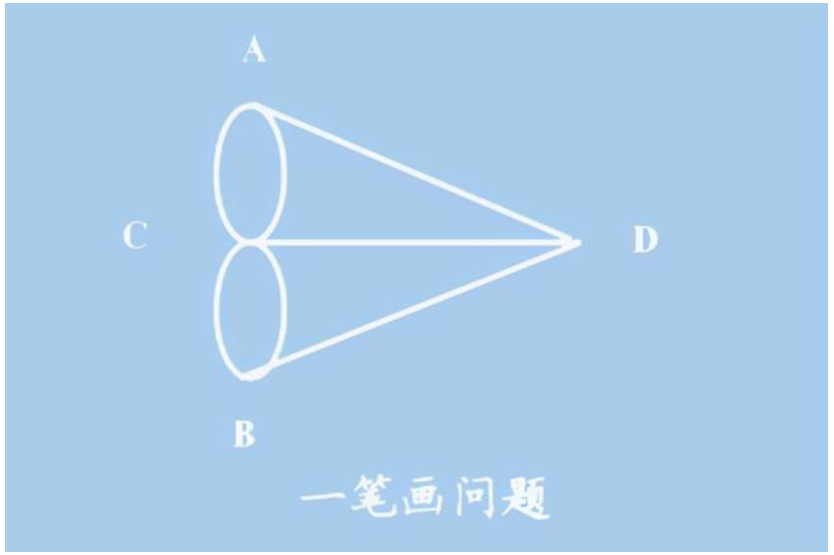
算法分析与设计
第二十二章
基本的图算法

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。





图算法

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。

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

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)。

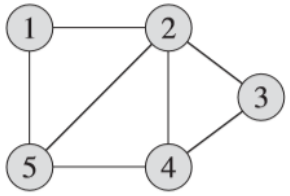
- 对于有向图，所有邻接链表的长度之和等于|E|；
- 对于无向图，所有邻接链表的长度之和等于2|E|。

图的表示

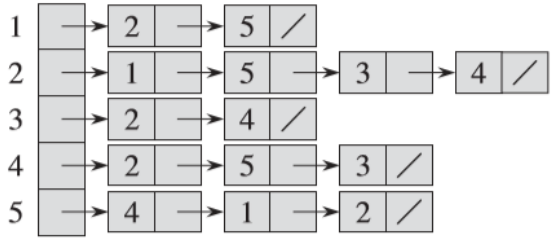
对于图G=(V,E)，可以用两种方法表示：邻接表、邻接矩阵

邻接表

无向图的邻接表：

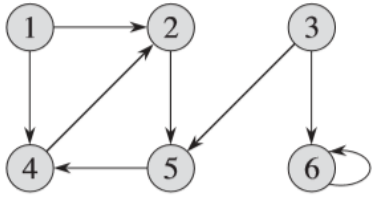


(a)

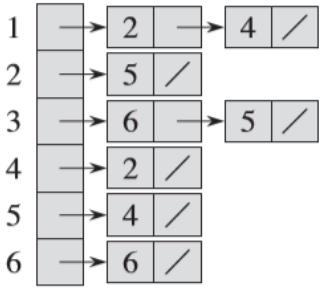


(b)

有向图的邻接表：



(a)



(b)

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

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$O(V^2)$。

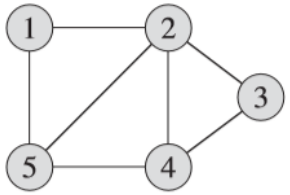
- 无向图的邻接矩阵A是一个对称矩阵，因此A也是自己的转置，即$A=A^T$。
- 为了节省空间，无向图的邻接矩阵可以用上三角或下三角矩阵表示，可以省近乎一半的空间。
- 但以上性质不适用于有向图。

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

邻接矩阵

无向图的邻接矩阵：

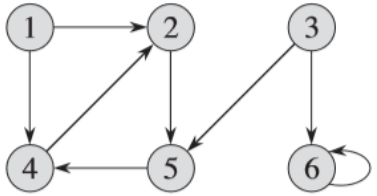


(a)

	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

(c)

有向图的邻接矩阵：



(a)

	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

(c)

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 ω:E→R 给出。

邻接表

- 将边(u,v)∈E的权重值ω(u,v)存放在u的邻接链表结点中,作为其属性

邻接矩阵

- 对于边(u,v)∈E，令邻接矩阵A[u][v] = ω(u,v)。
- 若(u,v)不是E中的边，则令A[u][v] =NIL，或∞、0。

稀疏图

边数|E|远小于$|V|^2$的图　一般用**邻接表**表示

稠密图

边数|E|远接近$|V|^2$的图　一般用**邻接矩阵**表示

- **邻接矩阵**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。

目　录

搜索和遍历

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

经典的图搜索算法：

- 广度优先搜索（BFS）
- 深度优先搜索（DFS）

广度优先搜索

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。

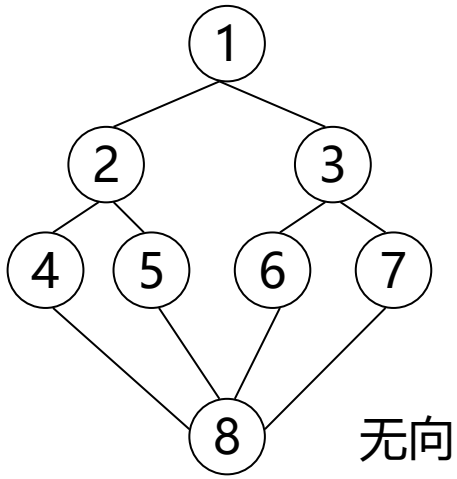
③ 若未检测结点表为空，则算法终止；否则

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。

广度 伪代码

```
procedure BFS(v)
  //广度优先搜索，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
```

Example

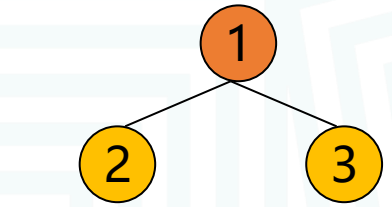


无向图G

检测结点1:

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

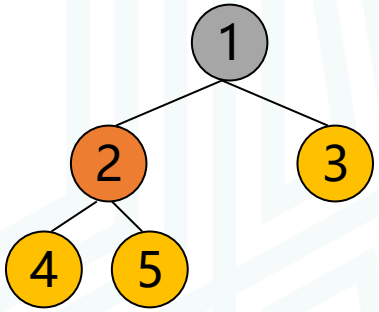
队列状态：



检测结点2（结点2出队列）：

visited(4)=1、visited(5)=1

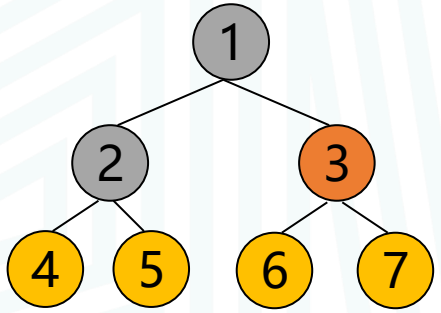
队列状态：



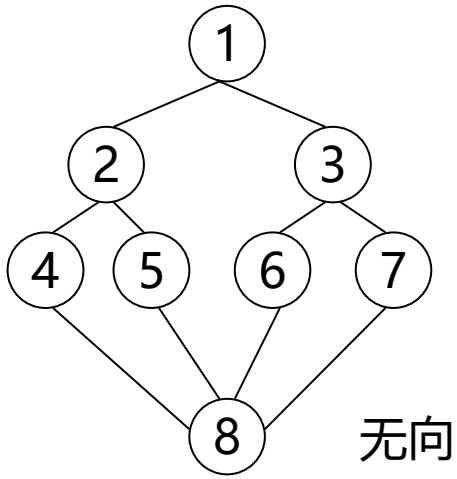
检测结点3（结点3出队列）：

visited(6)=1、visited(7)=1

队列状态：



Example



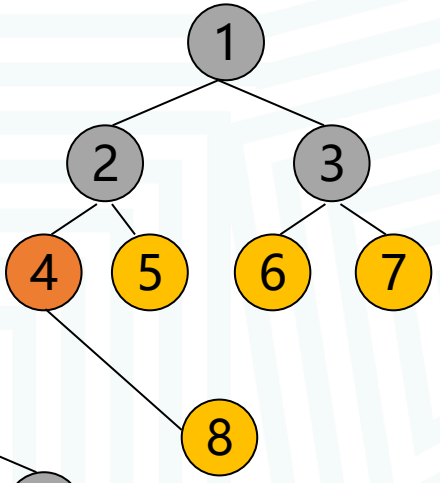
无向图G

检测结点4 （结点4出队列）：

visited(8)=1

队列状态：

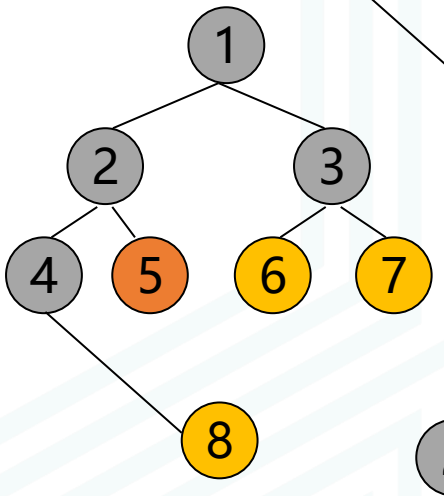
5	6	7	8	



检测结点5（结点5出队列）：

队列状态：

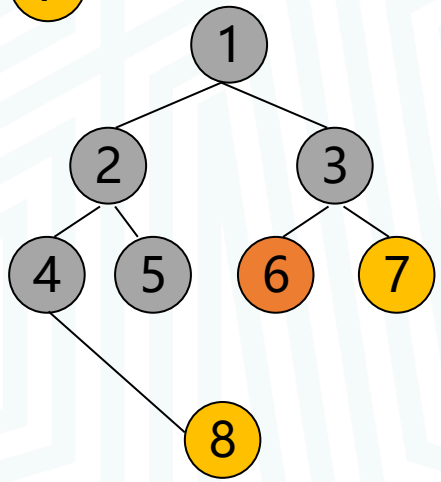
6	7	8	



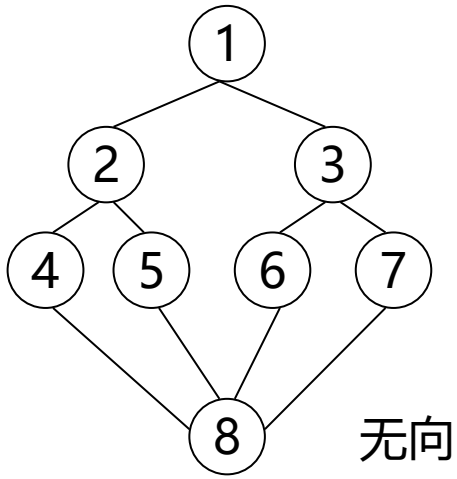
检测结点6（结点6出队列）：

队列状态：

7	8	



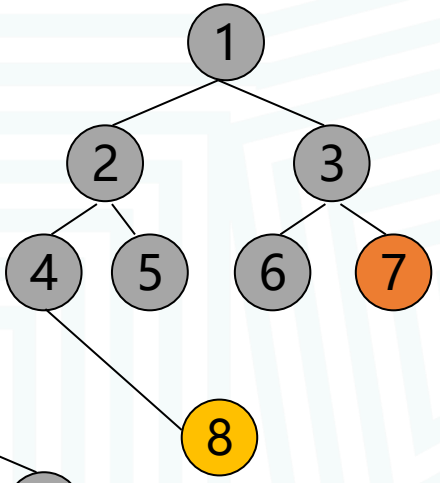
Example



无向图G

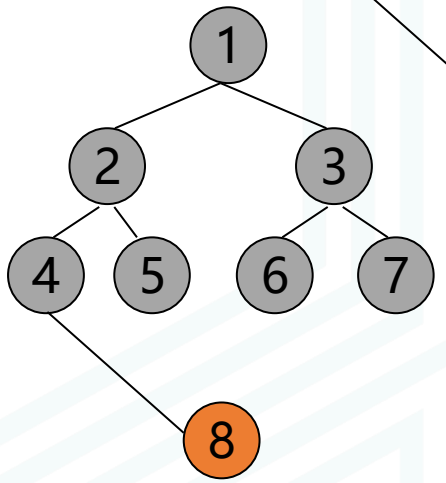
检测结点7（结点7出队列）：

队列状态：



检测结点8（结点8出队列）：

队列状态：



BFS的结点访问序列：1 2 3 4 5 6 7 8

广度 定理7.2

算法BFS可以访问由v可到达的所有结点

证明 数学归纳法

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。

(1) 若d(v,w)≤1，则显然所有这样的w都将被访问

广度 定理7.2

算法BFS可以访问由v可到达的所有结点

证明 数学归纳法

(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。

广度 定理7.2

算法BFS可以访问由v可到达的所有结点

证明 数学归纳法

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。

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 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。

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 空间分析

至多有n-1个这样的结点，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))。

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 空间分析

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“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。

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 时间分析 邻接表

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。

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 时间分析 邻接表

- 所有结点的处理时间：O(Σd(u))=O(e)。
- VISITED数组的初始化时间：O(n)

所以，算法总时间：**O(n+e)**

广度 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)=Θ(n+e)和s(n,e)=Θ(n)。
- 若G由邻接矩阵表示，则t(n,e)=Θ(n^2)和s(n,e)=Θ(n)

证明 时间分析 邻接矩阵

G采用邻接矩阵表示时，判断邻接于u的所有结点需要Θ(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)**

广度 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 时间分析 邻接矩阵

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。**

广度优先遍历

```
procedure BFT(G,n)
    //G的广度优先遍历//
    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
```

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的遍历。否则，需要多次调用BFS。

广度 图遍历算法的应用

- 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$A^*[i,j]=1$。

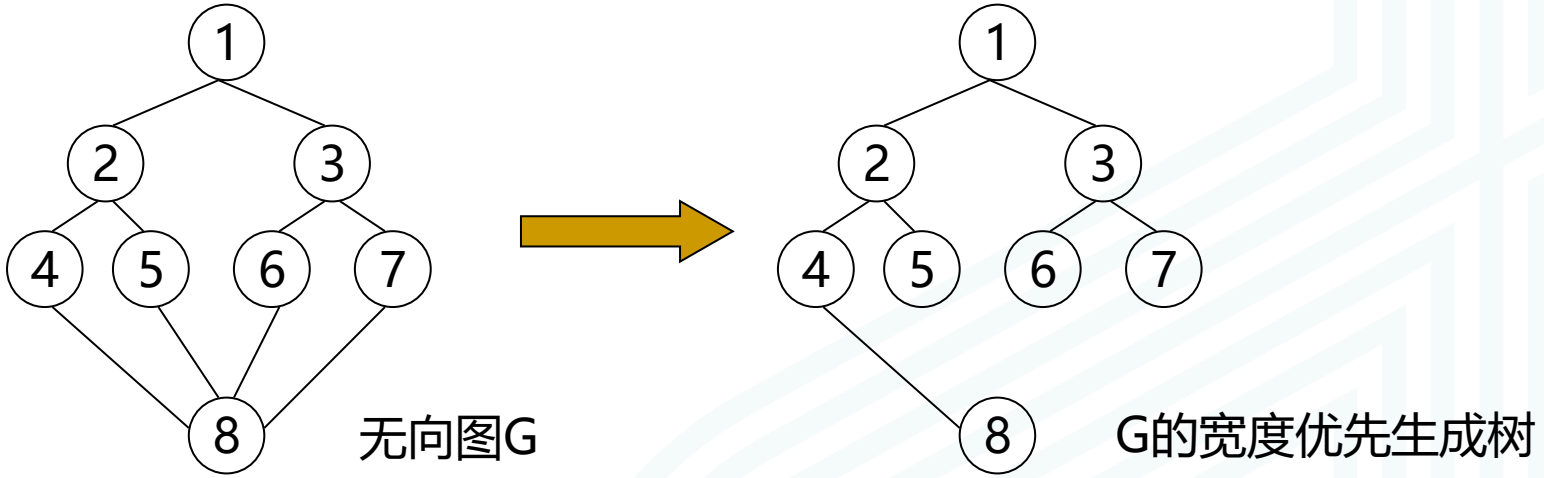

华中科技大学计算机科学与技术学院
School of Computer Science &Technology, HUST

广度优先生成树

定义

BFS中由u到达未访问结点w的边(u,w)称为**向前边**

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**。



伪代码

```
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*
```

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**。

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。**

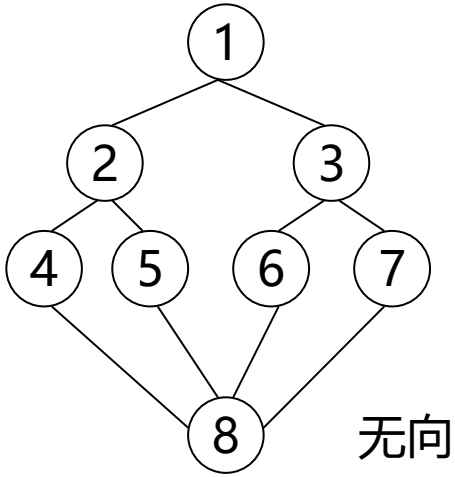
同时，对于连通图G，T将包含由起始结点v到其它结点的路径，所以**T是连通的。**

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**

深度优先搜索

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



无向图G

DFS结点访问序列：

1→2 →4 →8 →5 →6 →3 →7

深度 伪代码

```
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
```

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

深度优先遍历

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

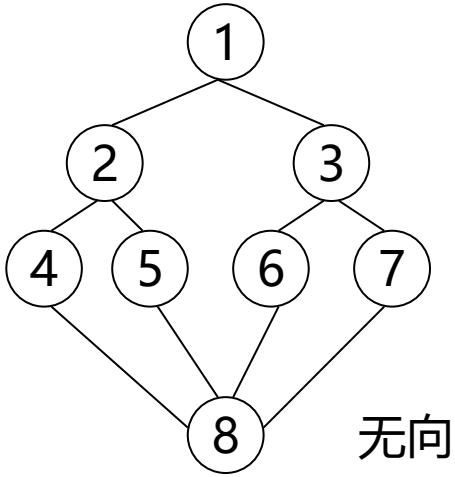

华中科技大学计算机科学与技术学院
School of Computer Science &Technology, HUST

深度优先生成树

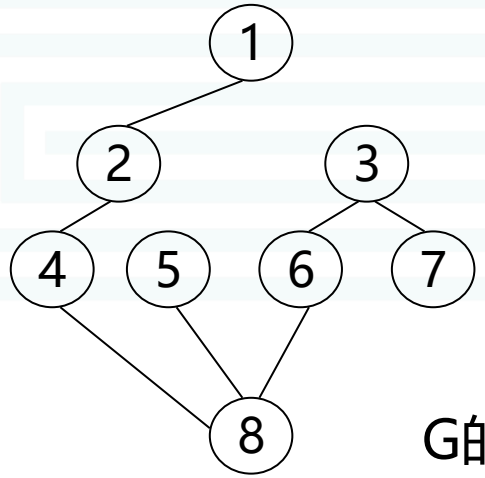
伪代码

```
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*
```

对比

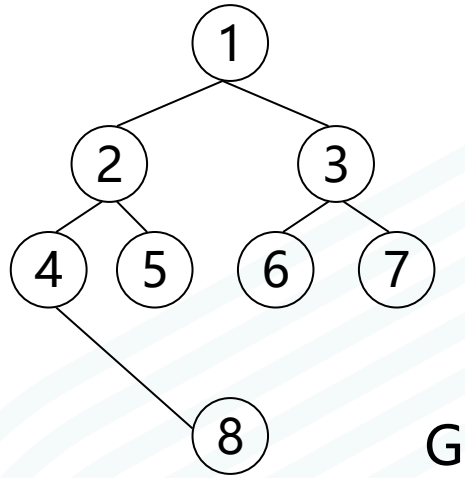


无向图G



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

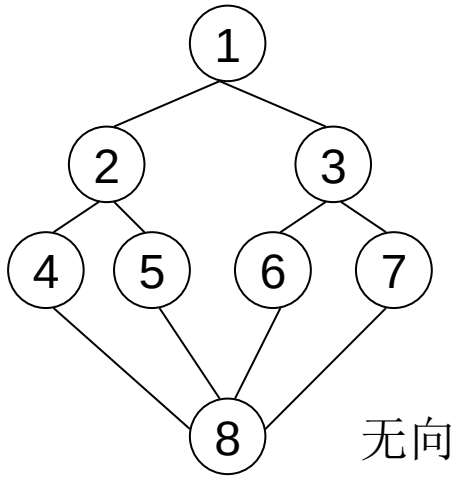
G的宽度优先生成树


华中科技大学计算机科学与技术学院
School of Computer Science &Technology, HUST

深度检索

定义

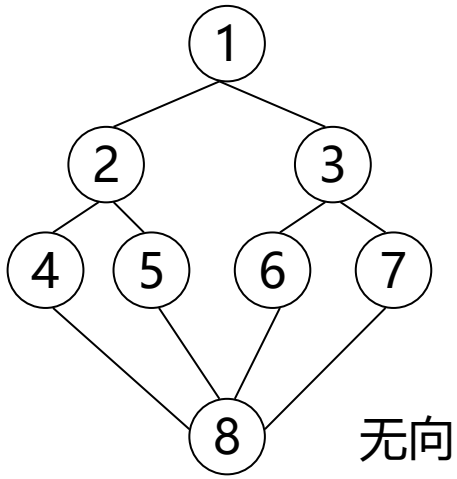
改造BFS算法，**用栈来保存未被检测的结点**，则得到的新的检索算法称为深度检索（D_Search）算法



无向图G

注：结点被压入栈中后将以相反的次序出栈

过程



无向图G

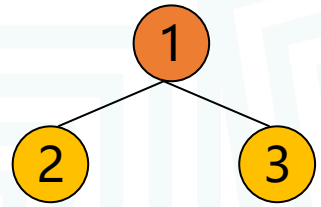
检测结点1 （结点1出栈）：

Visited(1)=1

Visited(2)=1

Visited(3)=1

栈状态：

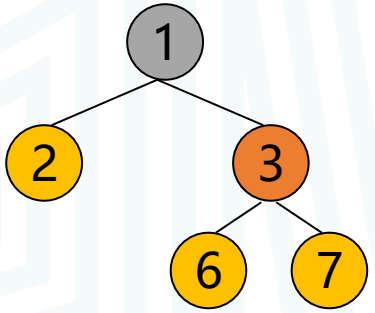


检测结点3（结点3出栈）：

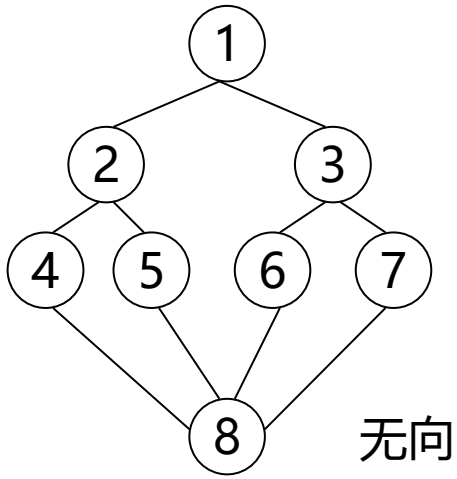
Visited(6)=1

Visited(7)=1

队列状态：



过程

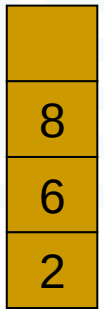


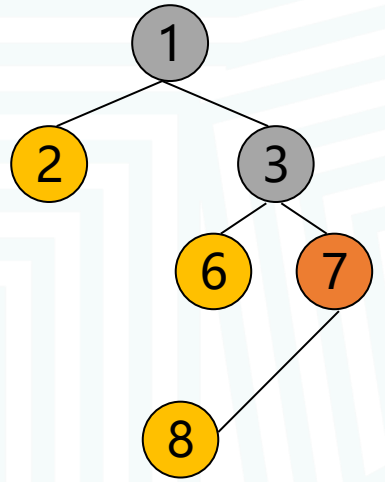
无向图G

检测结点7 （结点7出栈）：

Visited(8)=1

栈状态：



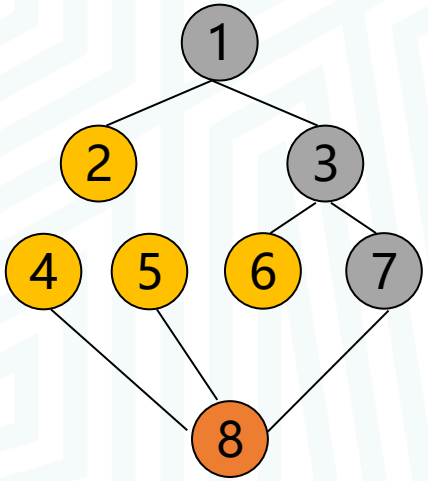


检测结点8（结点8出栈）：

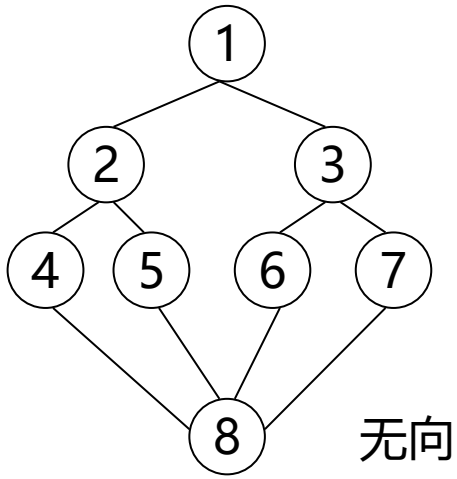
Visited(4)=1

Visited(5)=1

队列状态：



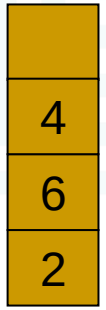
过程

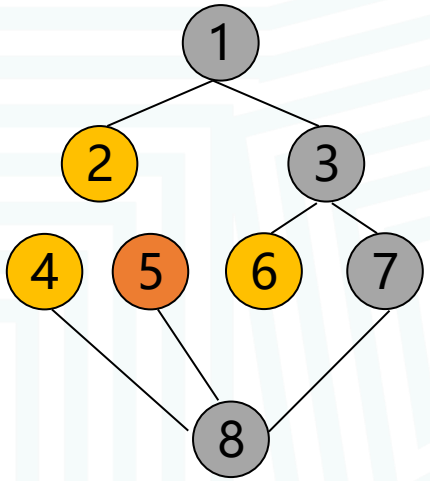


无向图G

检测结点5（结点5出栈）：

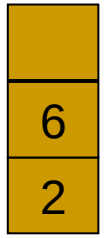
栈状态：

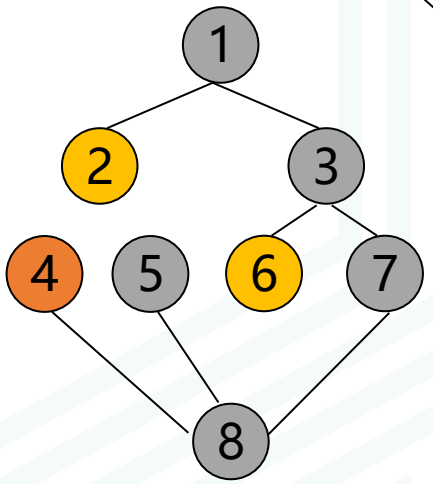




检测结点4（结点4出栈）：

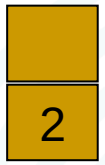
队列状态：

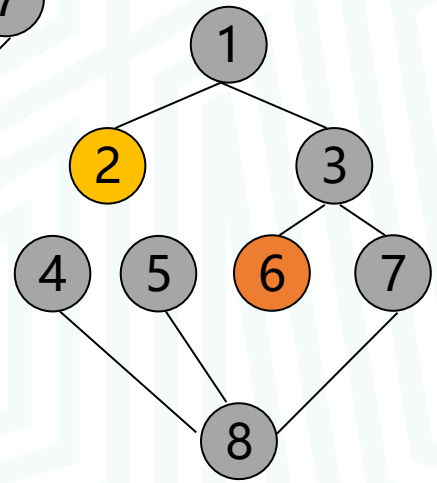




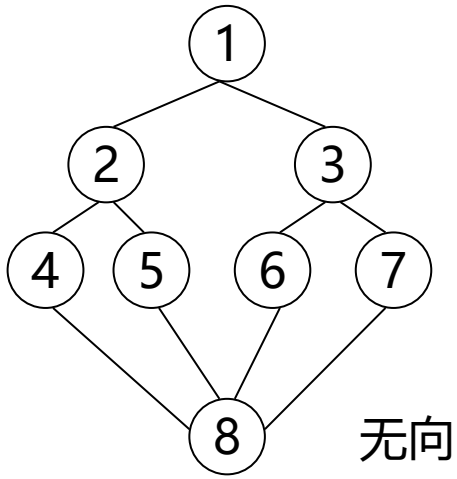
检测结点6（结点6出栈）：

队列状态：





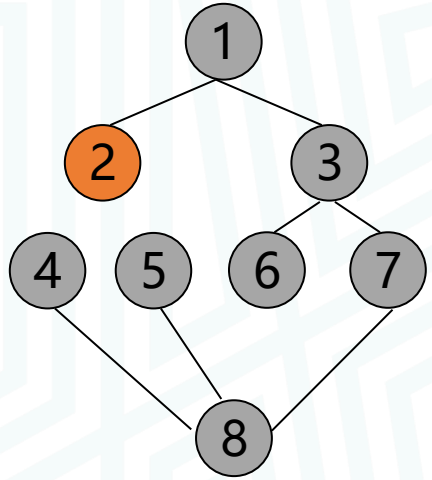
过程



无向图G

检测结点2 （结点2出栈）：

栈状态：



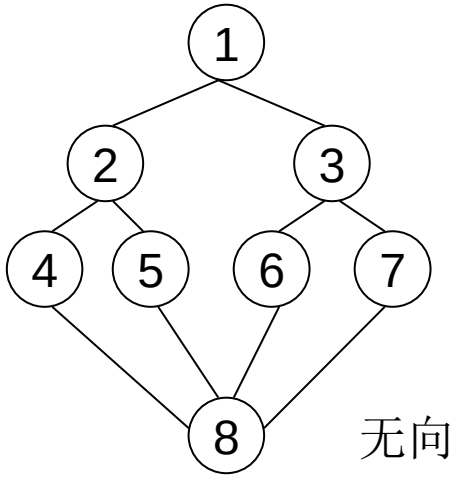
D_Search的结点访问序列: 1, 2, 3, 6, 7, 8, 4, 5

比较

BFS：使用队列保存未被检测的结点。结点按照广度优先的次序被访问和进、出队列。

DFS：使用栈保存未被检测的结点，结点按照深度优先的次序被访问并依次被压入栈中，并以相反的次序出栈进行新的检测。

D_Search：使用栈保存未被检测的结点，结点按照广度优先的次序被访问并被依次压入栈中，然后以相反的次序出栈进行新的检测。



无向图G

BFS访问序列：　1 2 3 4 5 6 7 8

DFS访问序列：　1 2 4 8 5 6 3 7

D_Search访问序列：　1 2 3 6 7 8 4 5

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

适用场景

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）

求解

目标：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

  $$m = m_1 m_2 ... m_n$$

- 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

改进

回溯（分支限界）能带来什么改进

- 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（$x_1,\ldots,x_i$），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 $S_i=\{0,1\}$

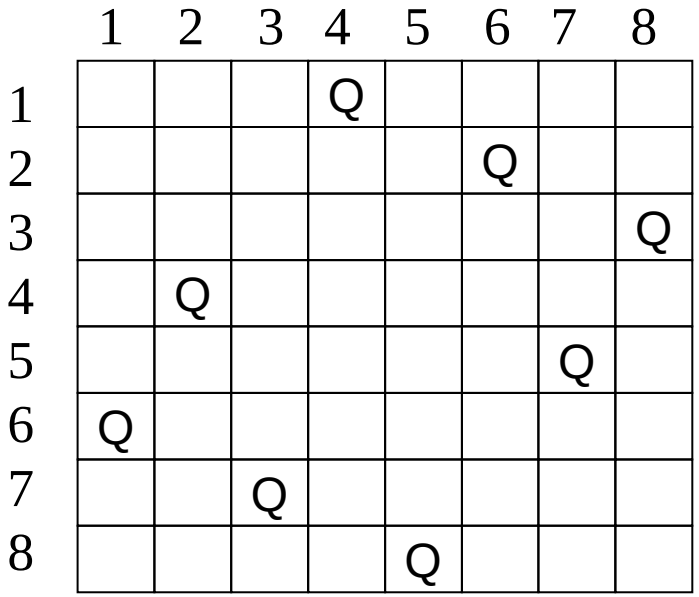
$l_i \leq x_i \leq u_i$　　即 $S_i=\{l_i \leq a \leq u_i\}$

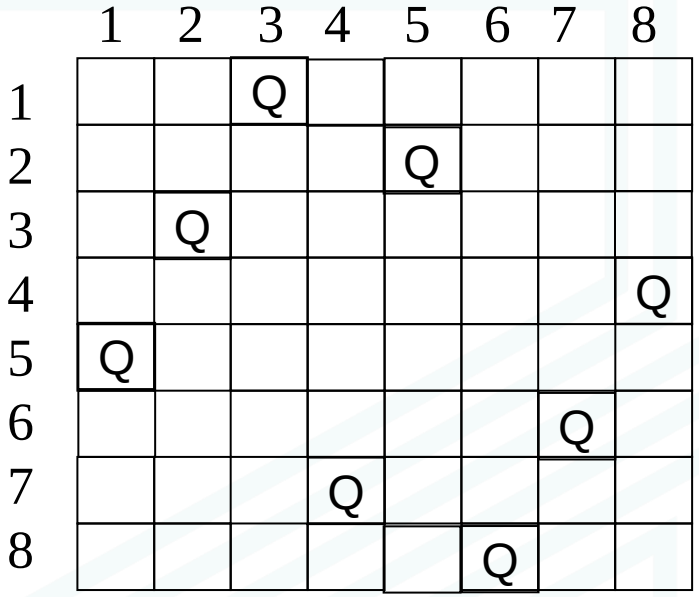
解空间：实例I的满足显式约束条件的所有元组，构成I的解空间，即所有x_i合法取值的元组的集合——可行解。

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。

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





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）

8皇后问题

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,...,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≤i≤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），有8！个

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。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 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）

解空间：所有可能的不同元组，总共有2^k个元组

子集和数问题 表示

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）

解空间：所有可能的不同元组，总共有2^n个元组

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

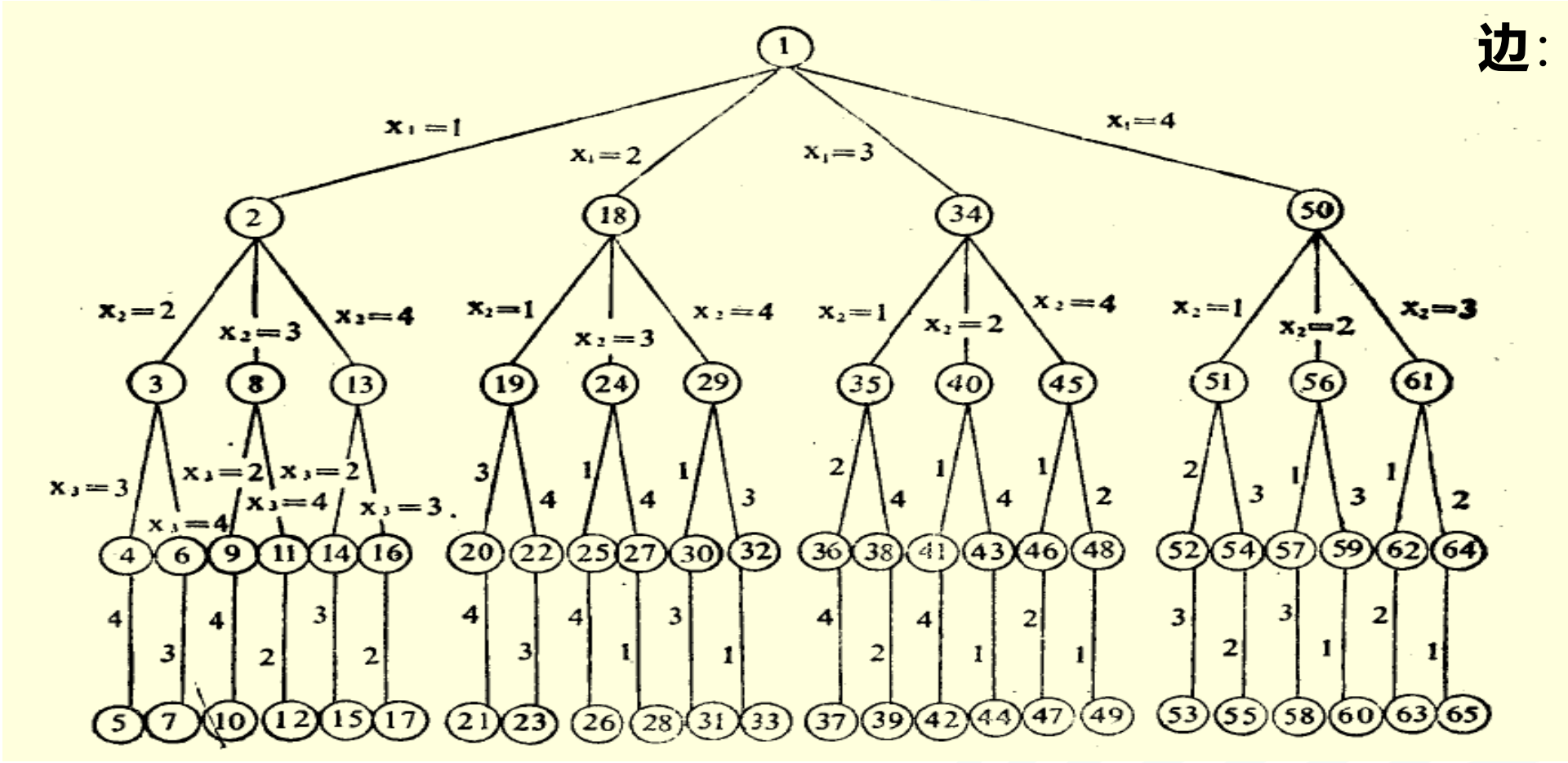
回溯法将通过系统地检索给定问题的解空间来求解，从而需要有效地组织问题的解空间

用**树结构**组织解空间，形成状态空间树

例 n-皇后问题，即在n×n的棋盘上放置n个皇后，使得它们不会相互攻击

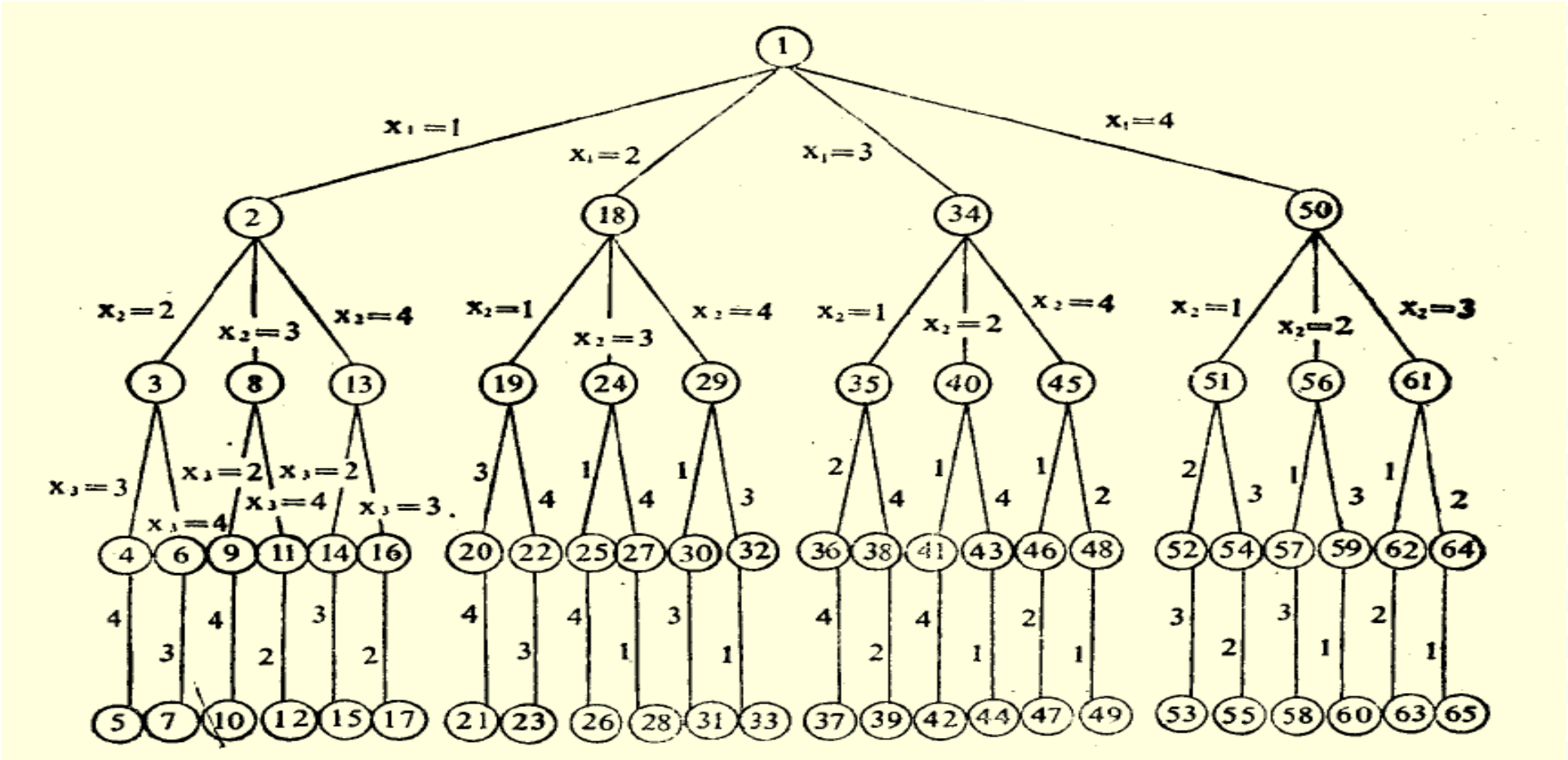
解空间：排列问题，解空间由n!个n-元组组成

4皇后问题 解空间树结构



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。

4皇后问题 解空间树结构



解空间：由从根结点到叶结点的所有路径所定义。

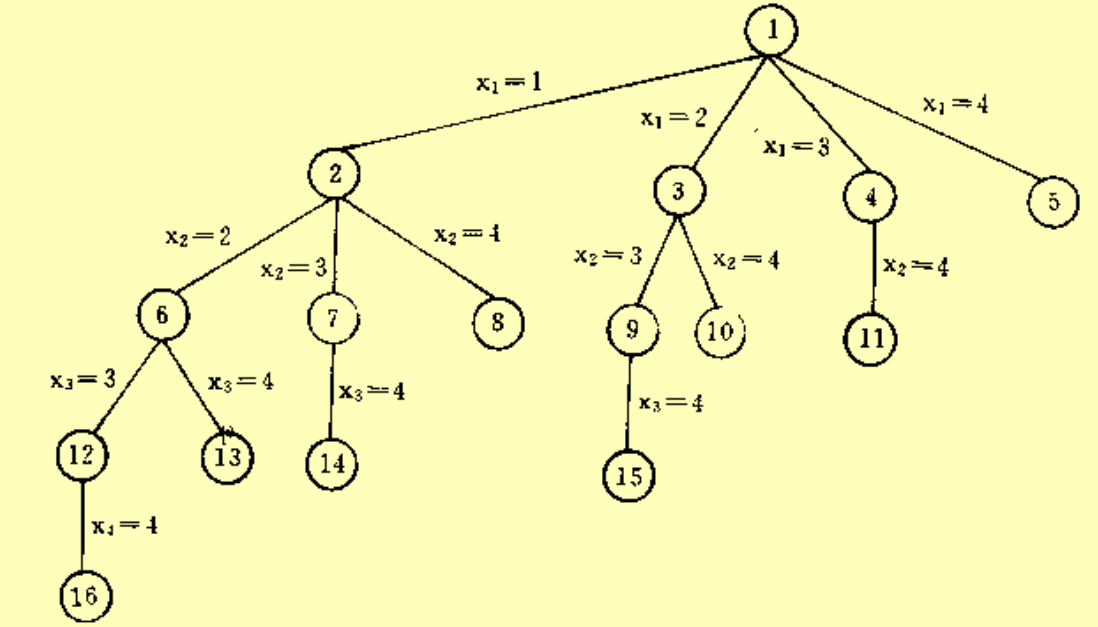
共有4！＝24个叶结点，反映了4元组的所有可能排列

——称为**排列树**

子集和数问题 解空间树结构

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

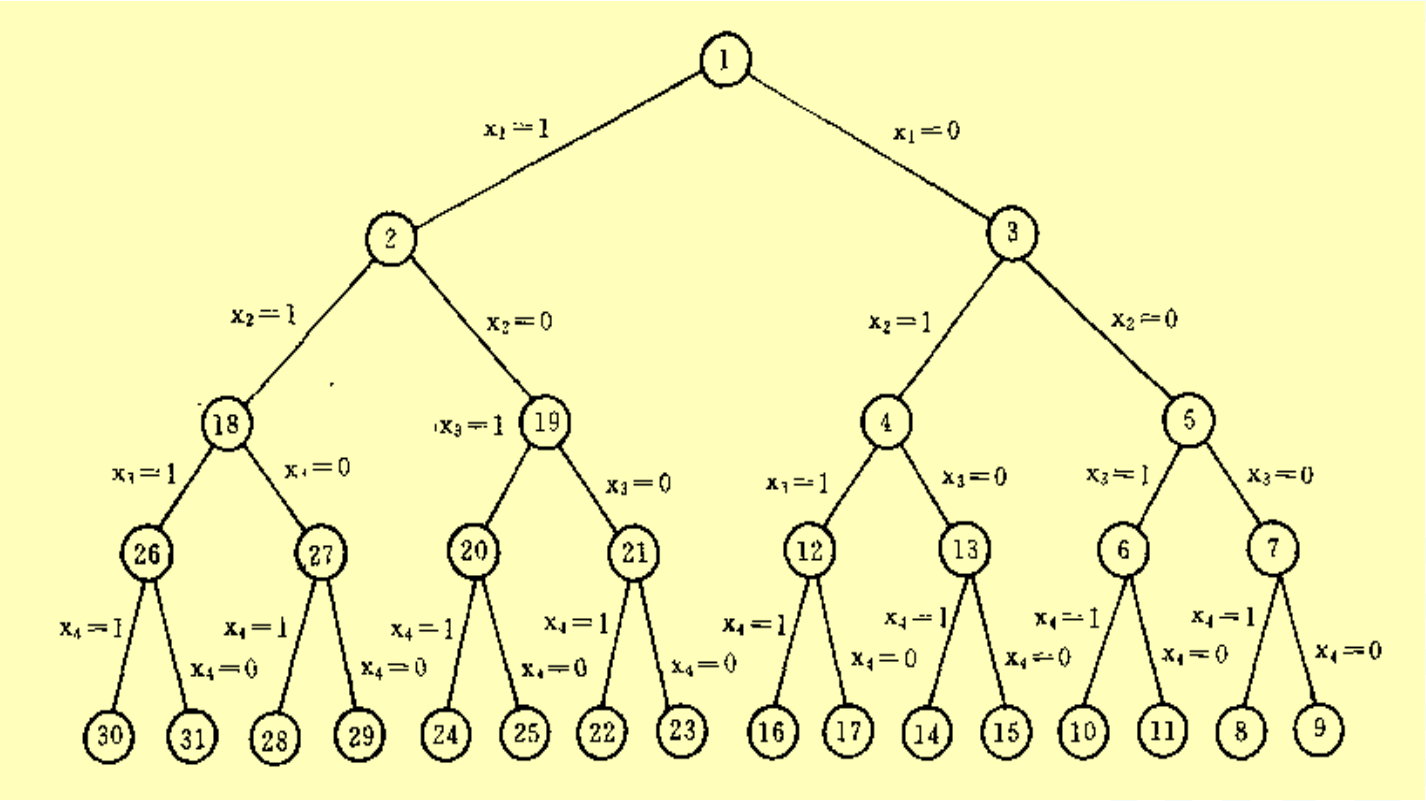


解空间由树中的根结点到任何结点的所有路径所确定，包括：(1), (1,2)，(1,2,3),(1,2,3,4), (1,2,4), (1,3,4), (1,4),(2), (2,3)等。共有16个可能的元组（结点1代表空集）

子集和数问题 解空间树结构

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。

状态空间树 概念

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 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

当前正在生成其儿子结点的活结点

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

状态空间树 构造策略

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

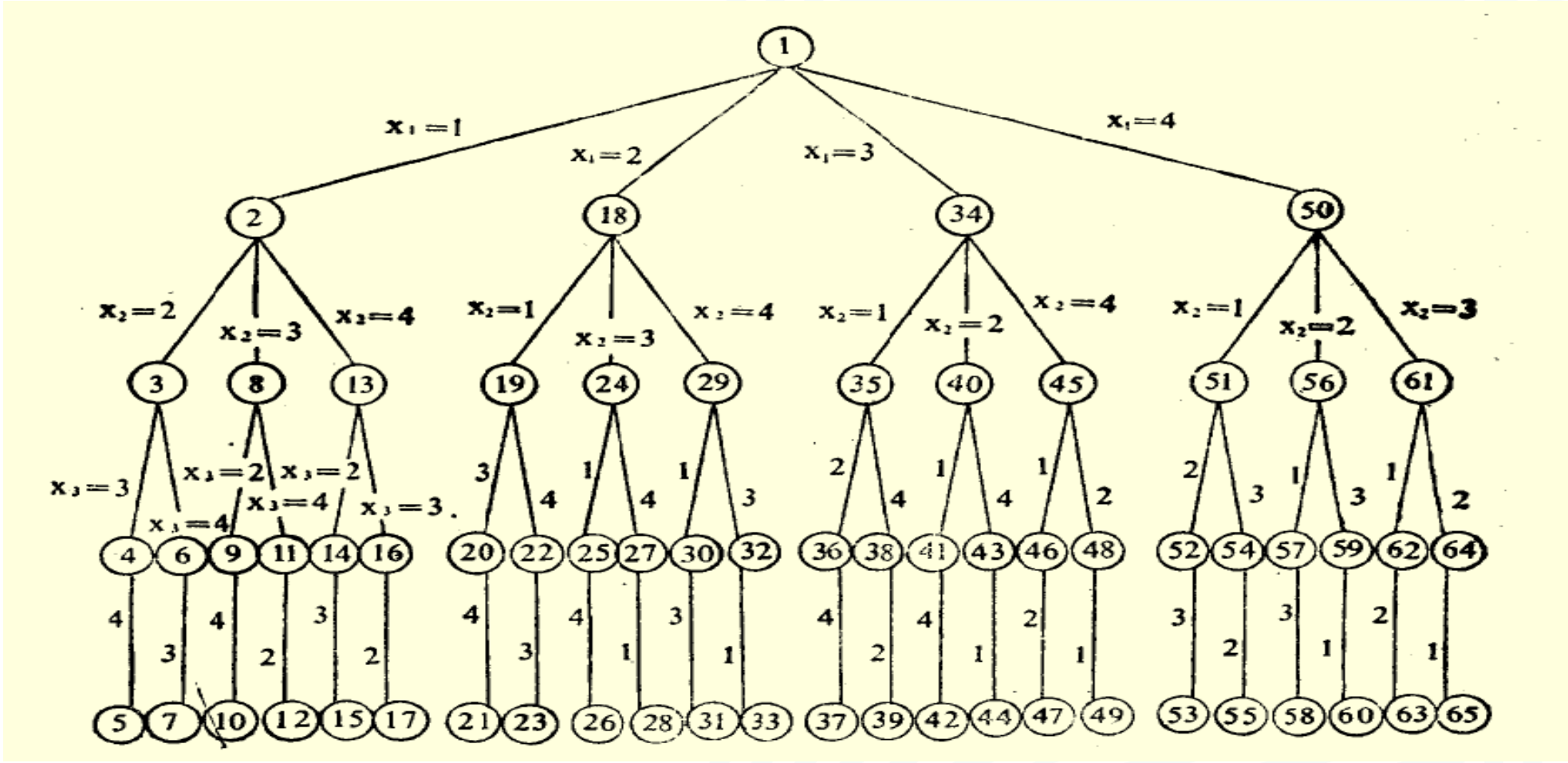
状态空间树 构造策略

- **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）

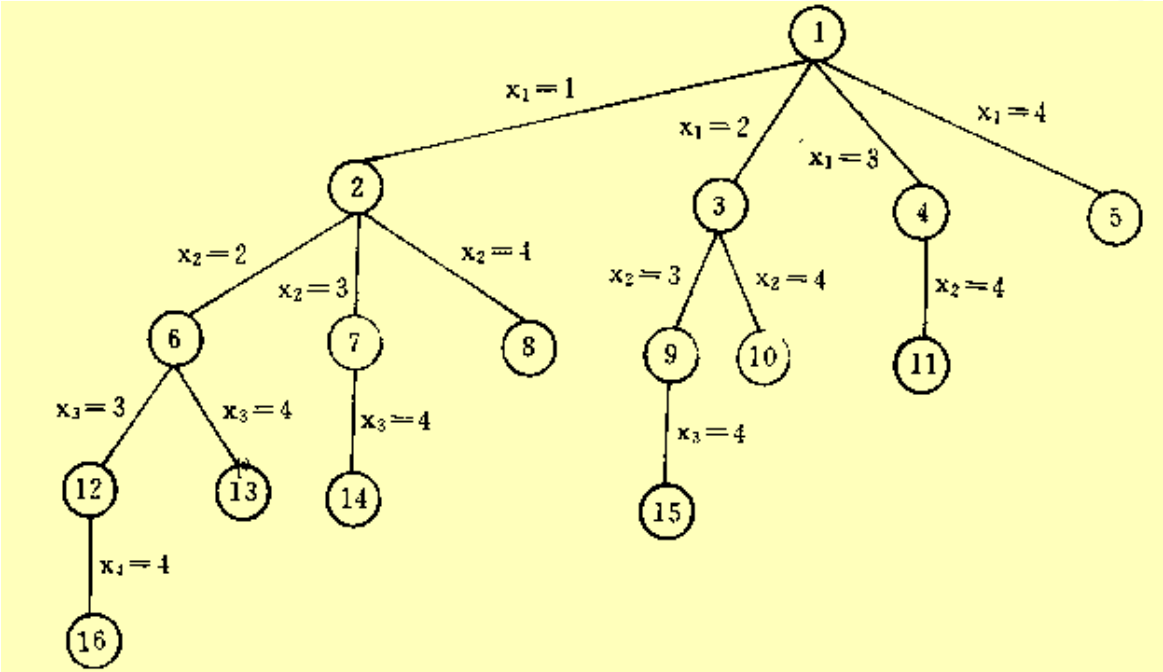
- **分支-限界方法**：使用限界函数的**E结点一直保持到死为止**的状态结点生成方法称为分支-限界方法（branch-and-bound）

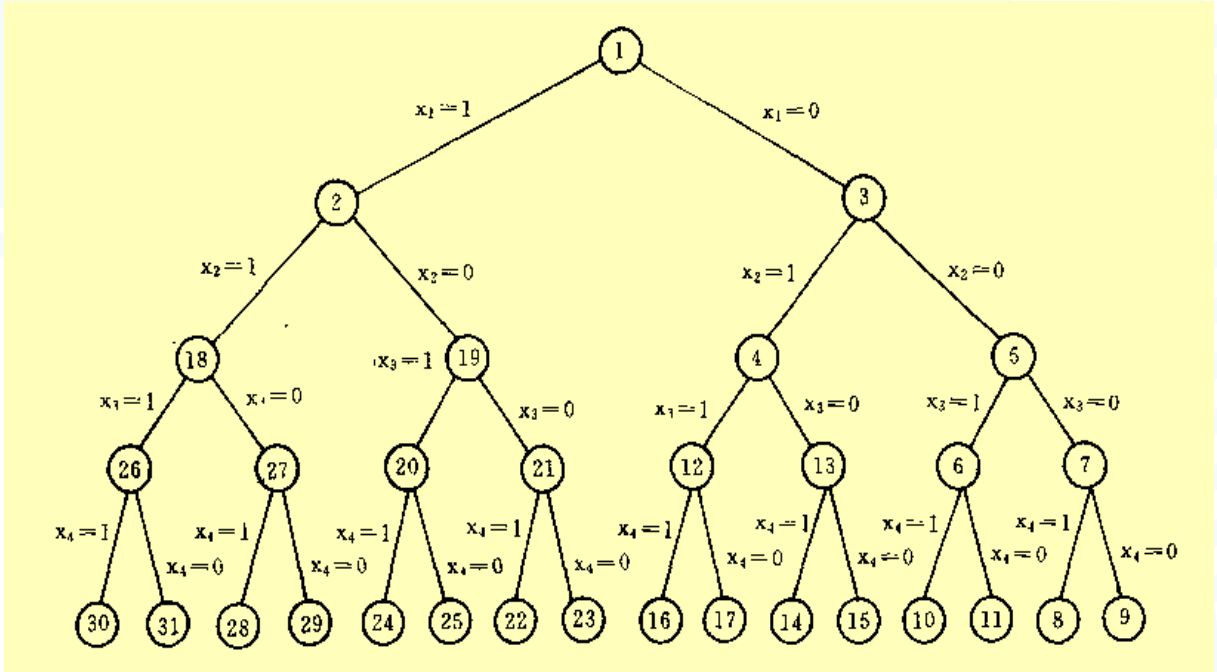
状态空间树 深度优先

- 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）



状态空间树 深度优先





利用**队列**的宽度优先策略下的结点生成次序(BFS)

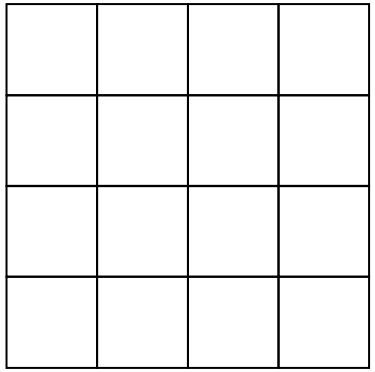
利用**栈**的宽度优先策略下的结点生成次序(D-Search)

4皇后问题 回溯法求解

- 限界函数：如果（$x_1,x_2,\ldots,x_{i-1}$）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（$x_1,x_2,\ldots,x_{i-1},x_i$）表示没有两个皇后处在相互攻击状态的一种棋盘格局。
- 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。

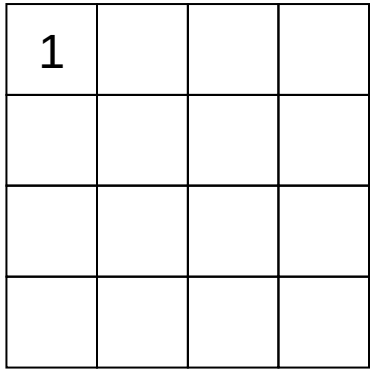
4皇后问题 回溯法求解

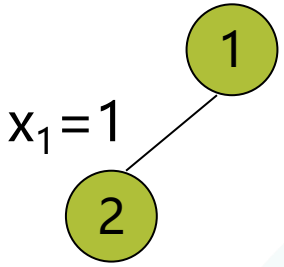
按照自然数递增的次序生成4皇后问题状态空间树中结点的儿子结点。



根结点1，开始状态，唯一的活结点

解向量： ()





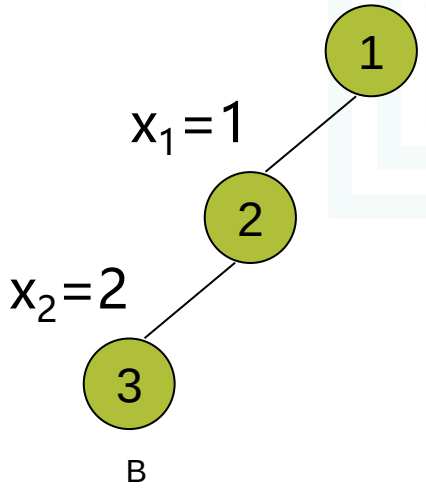
生成结点2，表示皇后1被放到第1行的第1列上，该结点是从根结点开始第一个被生成结点。

解向量： (1)

结点2变成新的E结点，下一步扩展结点2

4皇后问题 回溯法求解

1			
•	2		



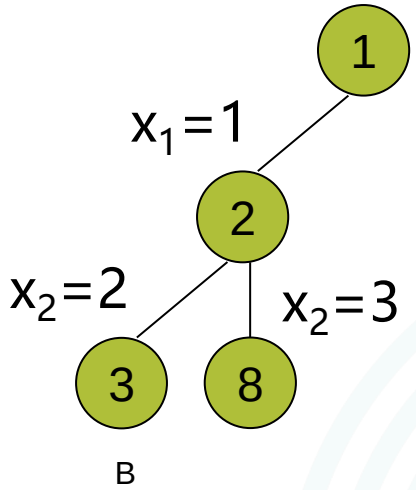
由结点2生成结点3，表示皇后2放到第2行第2列。

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）

1			
•	•	2	



由结点2生成结点8，表示皇后2放到第2行第3列。

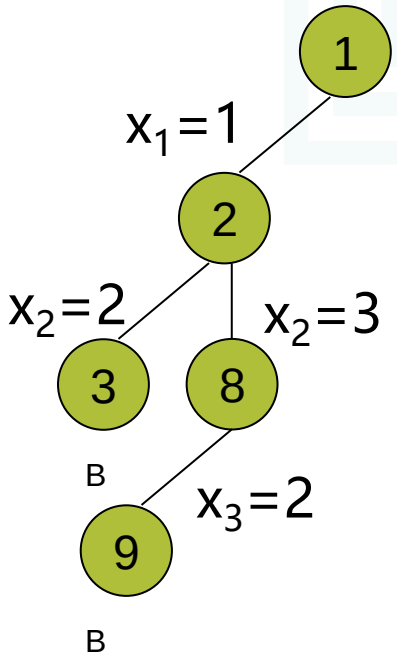
结点8变成新的E结点。

解向量：（1，3）

从结点8继续扩展。

4皇后问题 回溯法求解

1			
•	•	2	
	3		



由结点8生成结点9，表示皇后3放到第3行第2列。

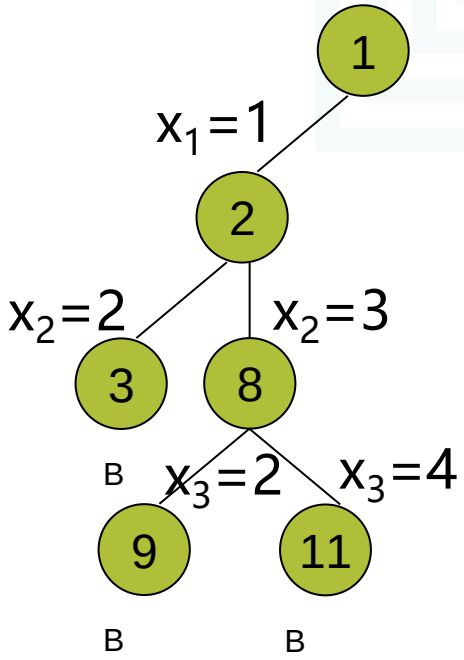
利用限界函数杀死结点9。

返回结点8继续扩展。

（9后面的结点10不会生成）

4皇后问题 回溯法求解

1			
•	•	2	
•	•	•	3



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）

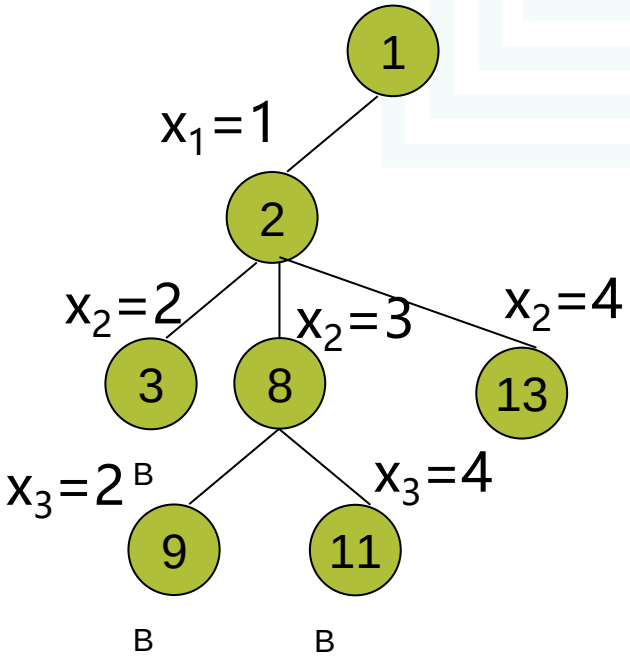
结点8的所有儿子已经生成，变成死结点，且没有找到答案结点。

结点8被杀死。

返回结点2继续扩展。

4皇后问题 回溯法求解

1			
•	•	•	2



由结点2生成结点13，表示皇后2放到第2行第4列。

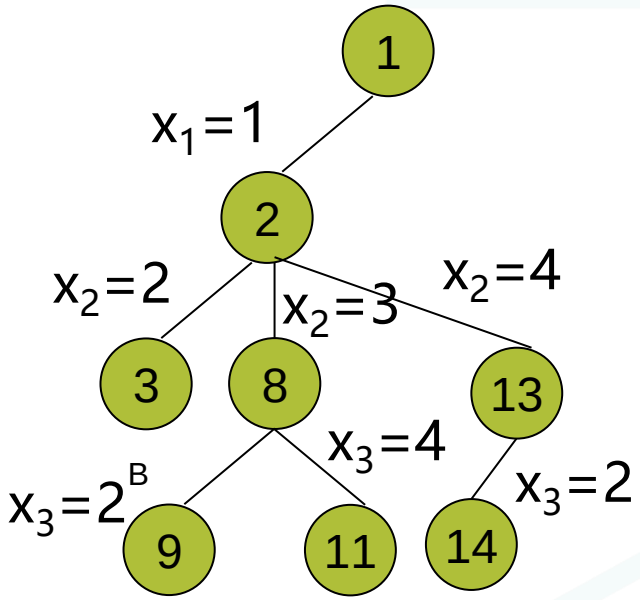
结点13变成新的E结点。

解向量：（1，4）

从结点13继续扩展。

4皇后问题 回溯法求解

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。

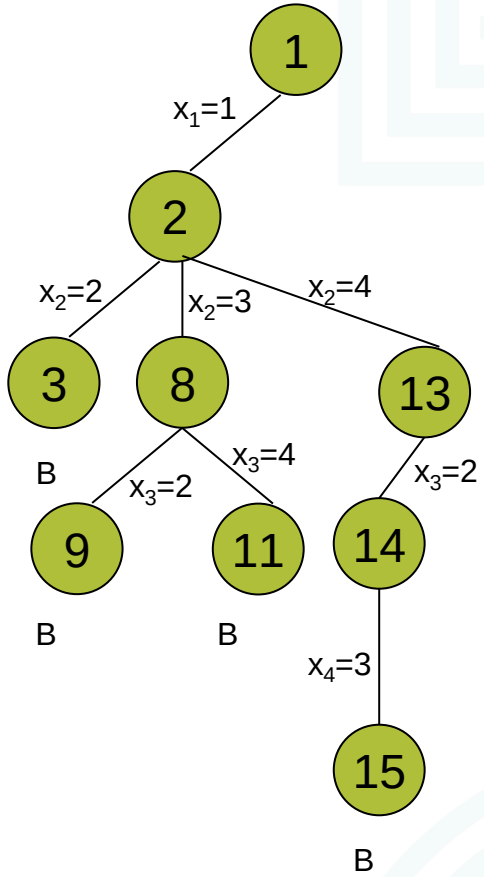
结点14变成新的E结点。

解向量：（1，4，2）

从结点14继续扩展。

4皇后问题 回溯法求解

1			
•	•	•	2
•	3		
•	•	4	



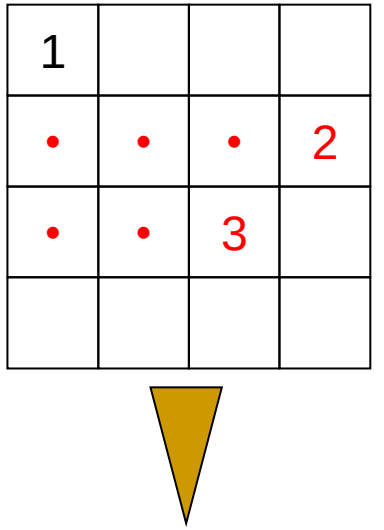
由结点14生成结点15，表示皇后4放到第4行第3列。

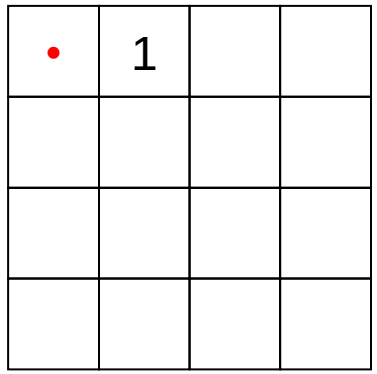
利用限界函数杀死结点15。

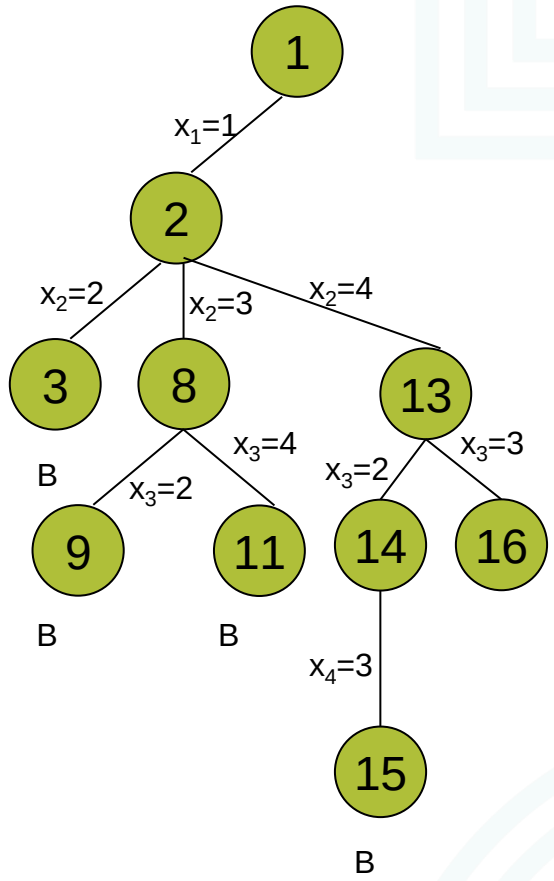
返回结点14，结点14不能导致答案结点，变成死结点，被杀死。

返回结点13继续扩展。

4皇后问题 回溯法求解







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。

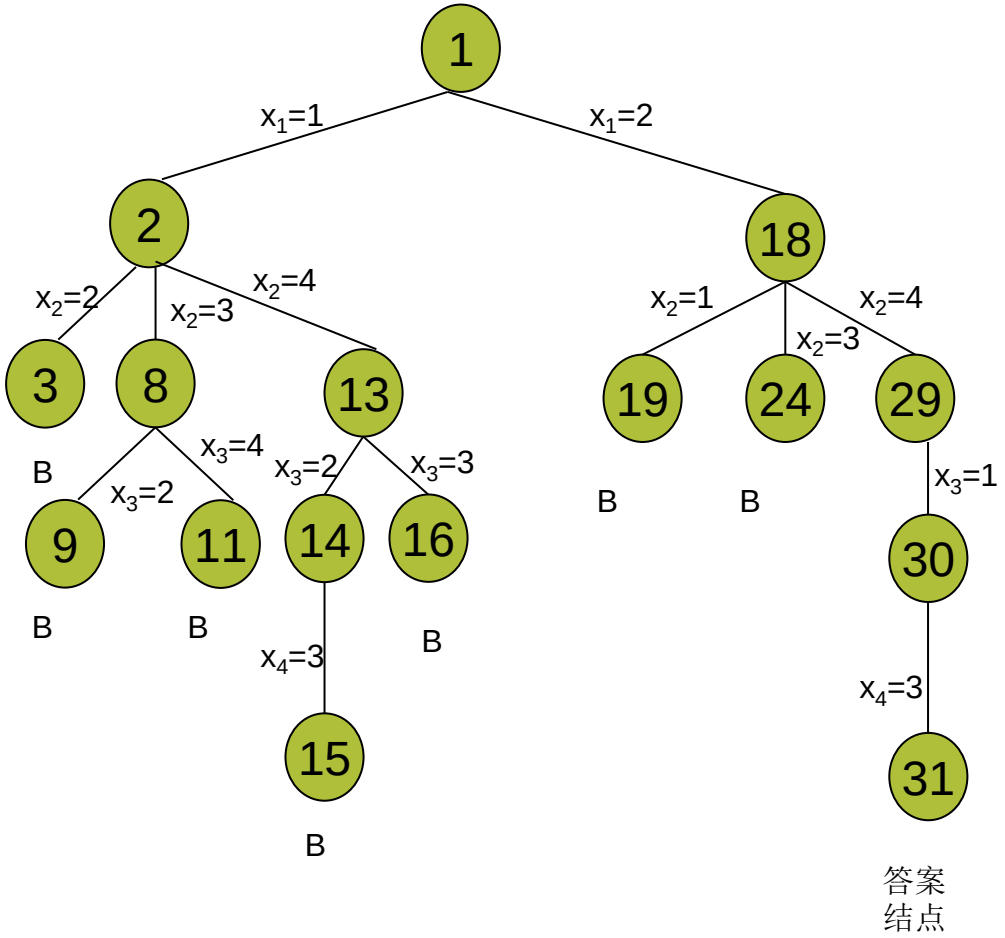
返回结点2继续扩展。

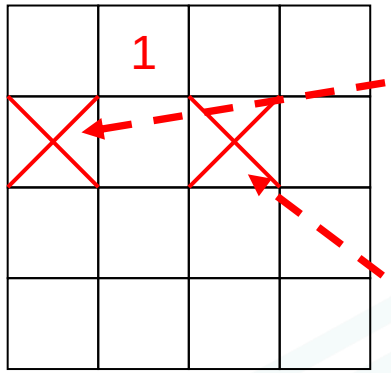
结点2不能导致答案结点，变成死结点，被杀死。

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。

4皇后问题 回溯法求解





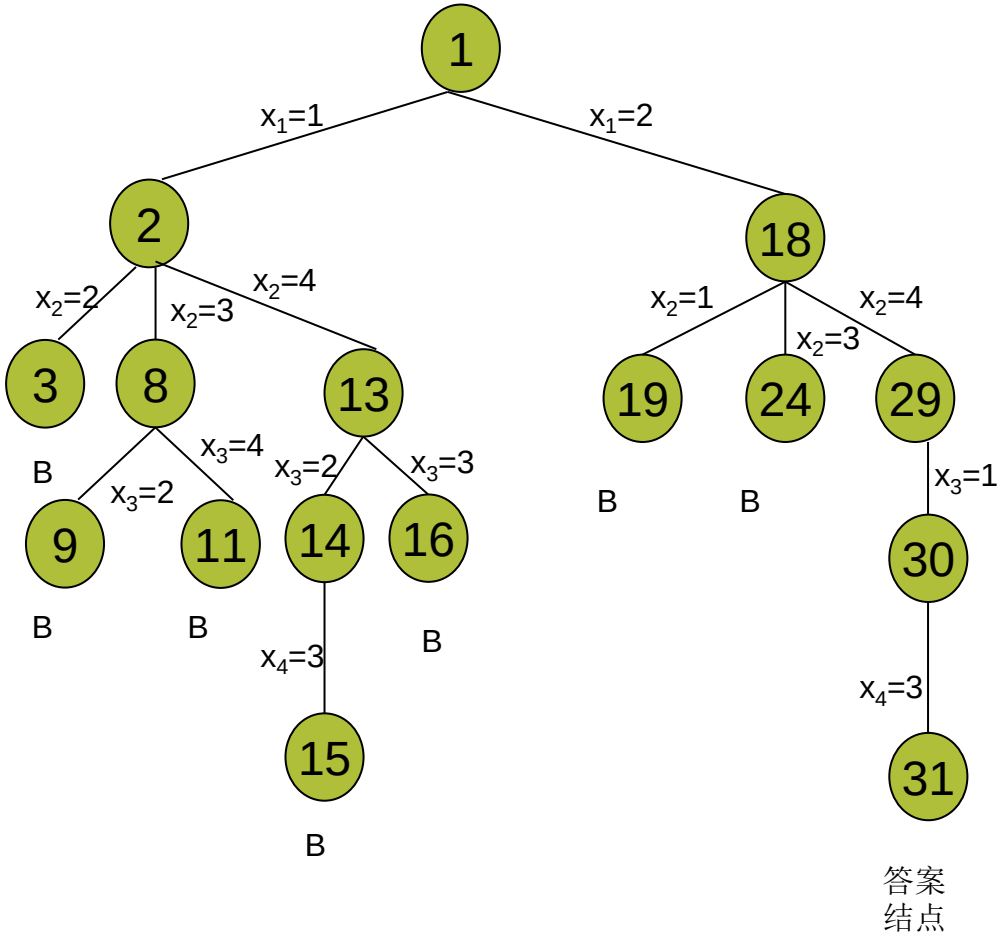
由结点1生成结点18，即皇后1放到第1行第2列。结点18变成E结点。

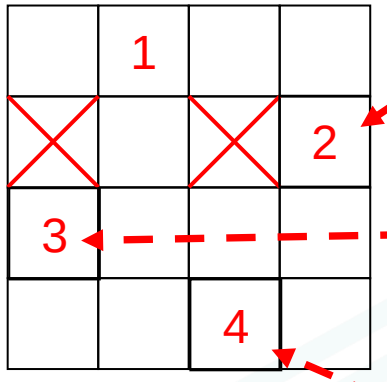
扩展结点18生成结点19，即皇后2放到第2行第1列。

利用限界函数杀死结点19。

返回结点18，生成结点24，即皇后2放到第2行第3列。

4皇后问题 回溯法求解





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。

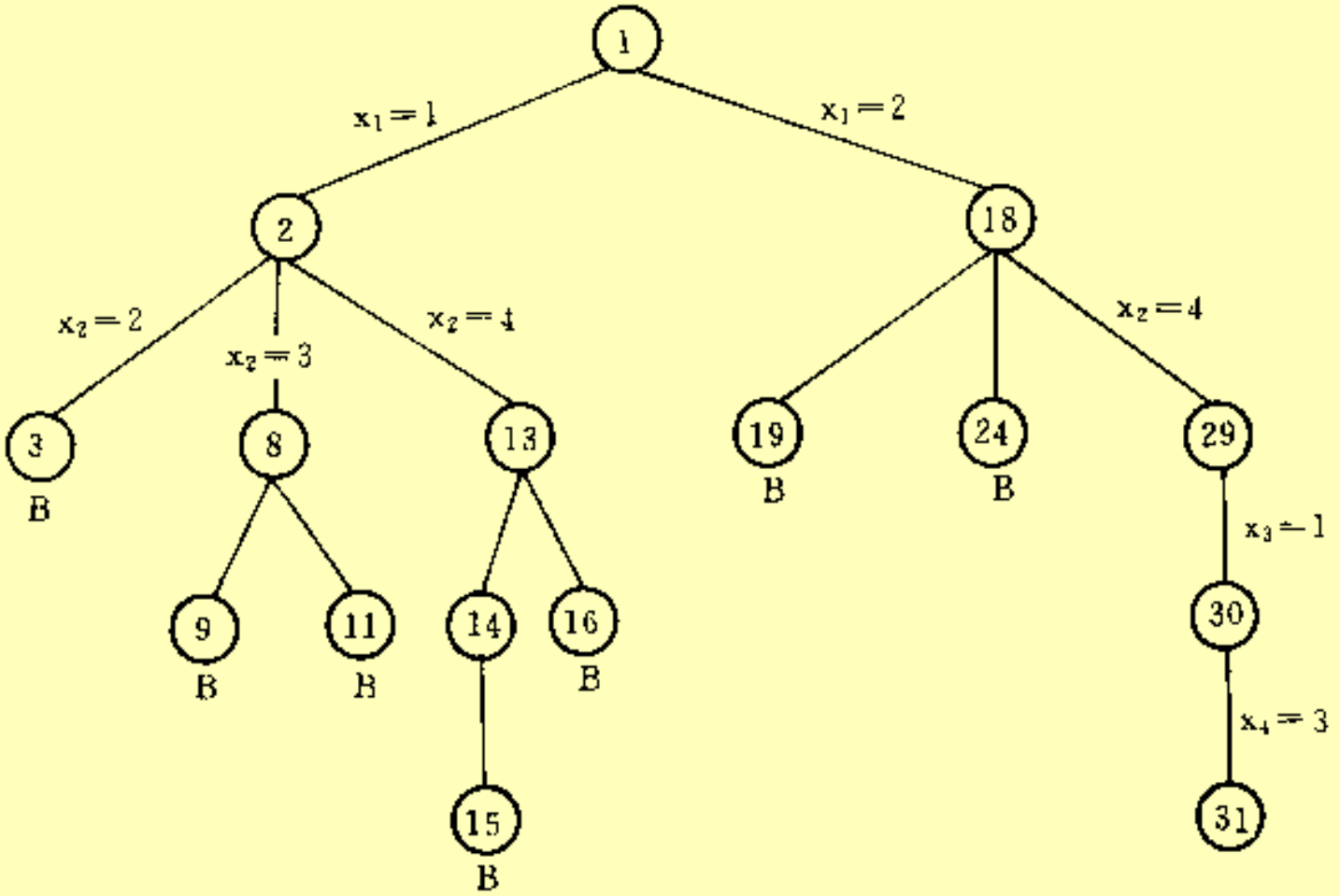
扩展结点30生成结点31，即皇后4放到第4行第3列。

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)。

4皇后问题 回溯法求解

	1	2	3	4
1				
2				
3				
4				

4皇后问题 回溯法求解



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)。

算法描述

- 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$x_i$。

设计思想

- 首先，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；并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。

设计思想

- 然后，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。

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。

递归表示

```
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
```

- 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。

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。


华中科技大学计算机科学与技术学院
School of Computer Science &Technology,HUST

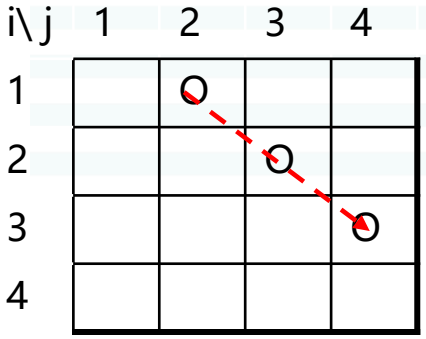
N-皇后问题

求解

- n元组： $(x_1,x_2,\ldots,x_n)$
- 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？

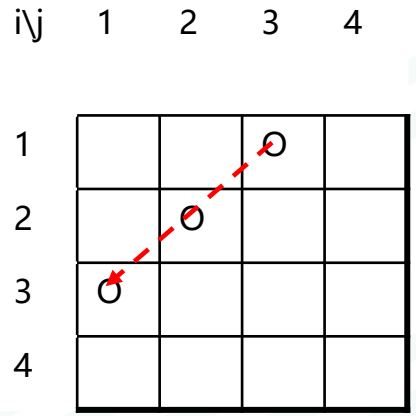
求解

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 - 列**”值。



左上方——右下方
相同的“行 - 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

求解

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伪代码

```
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
```

N-Queens伪代码

```
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                                //找到一个位置//
             then   if  k=n                       //是一个完整的解吗？//
                       then print(X)              //是，输出解向量//
                       else k←k+1; X(k)←0         //否，转下一皇后//
                    endif
                                                       else
                                                            k←k-1
                                                       endif
                                                  repeat
                                             end NQUEENS
```


华中科技大学计算机科学与技术学院
School of Computer Science &Technology, HUST

子集和数问题

求解

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

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。

仅当满足上述两个条件时，限界函数B(X(1),...X(k))=true

伪代码

```
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
```

//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，　$\sum_{i=1}^{n}W(i)\geq M$　　//

向前看两步,可以的话才进行下一步处理

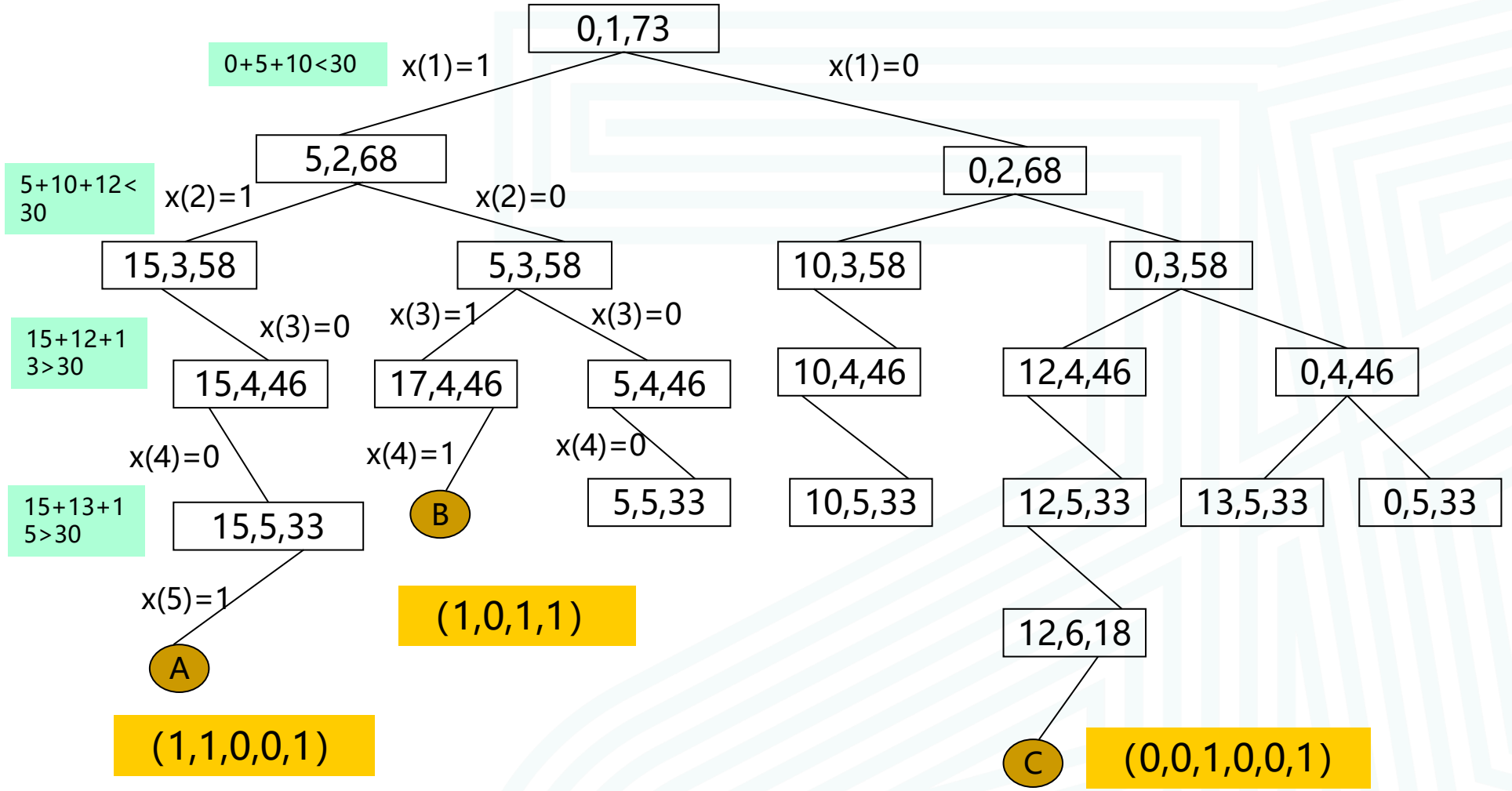
伪代码

```
//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
```

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

树型实例

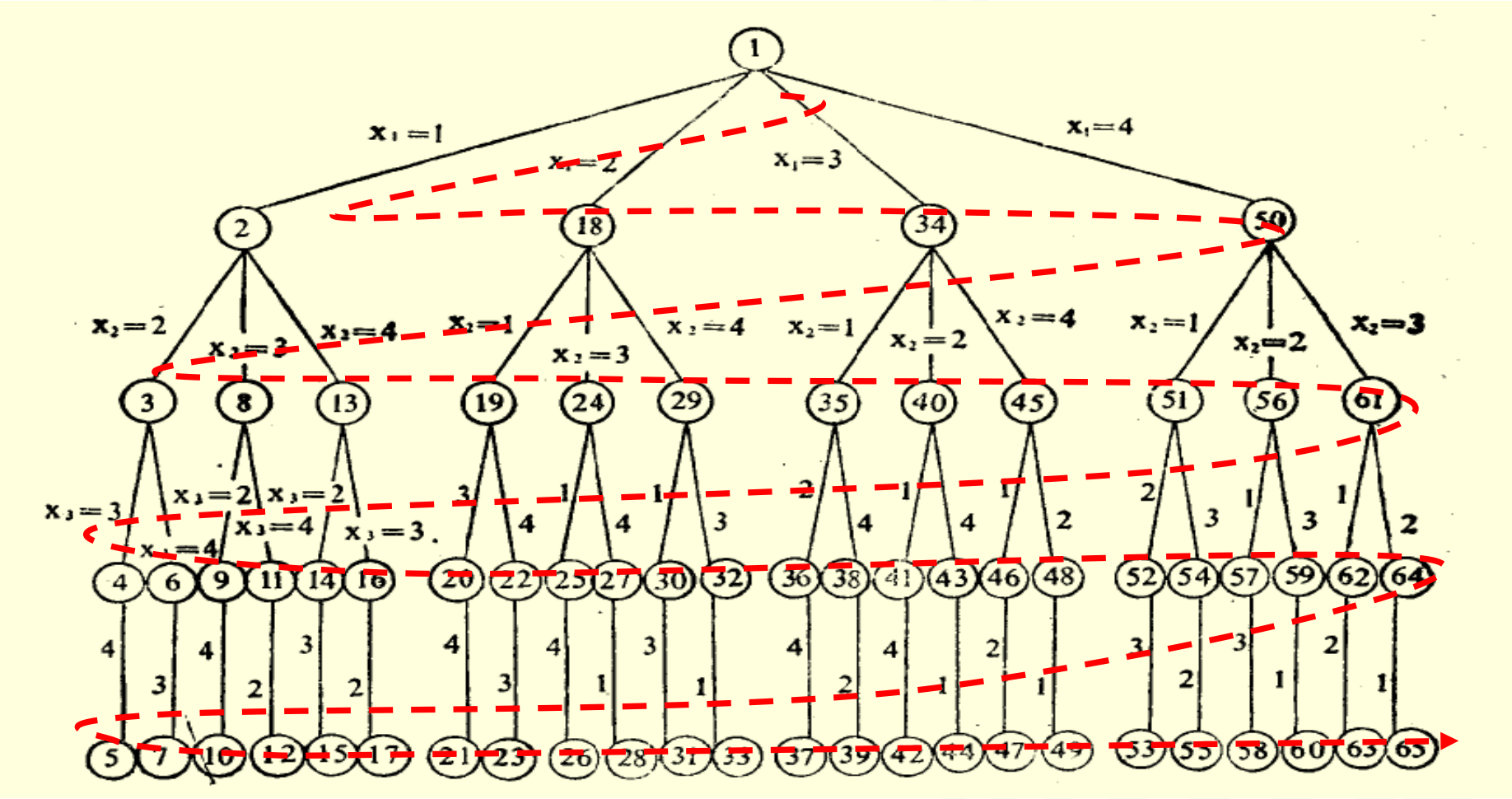
- n=6, M=30, W(1:6)=(5,10,12,13,15,18)
- 方形结点：s，k，r，圆形结点：输出答案的结点，共生成20个结点



分支限界法

- **分支 - 限界法**：采用广度优先策略，在生成当前E-结点全部儿子之后再生成其它活结点的儿子，且用限界函数帮助避免生成不包含答案结点子树的状态空间的检索方法。
- **活结点表**：
  - 活结点：已经被生成，但还没有被检测的点
  - 存储结构：队列（First In First Out, BFS）、栈（Last In First Out, D-Search）
- **两种基本设计策略**：
  - FIFO检索：活结点表采用队列
  - LIFO检索：活结点表采用栈

4皇后状态空间树



- 限界函数：如果$(x_1,x_2,\ldots,x_{i-1})$是到当前E结点的路径，那么具有父－子标记的所有儿子结点x_i是一些这样的结点，它们使得$(x_1,x_2,\ldots,x_{i-1},x_i)$表示没有两个皇后正在相互攻击的一种棋盘格局。

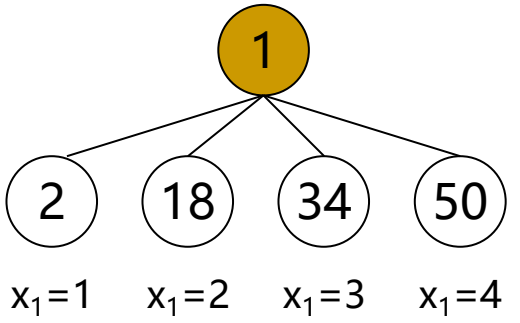
4皇后问题 FIFO分支限界 状态空间树

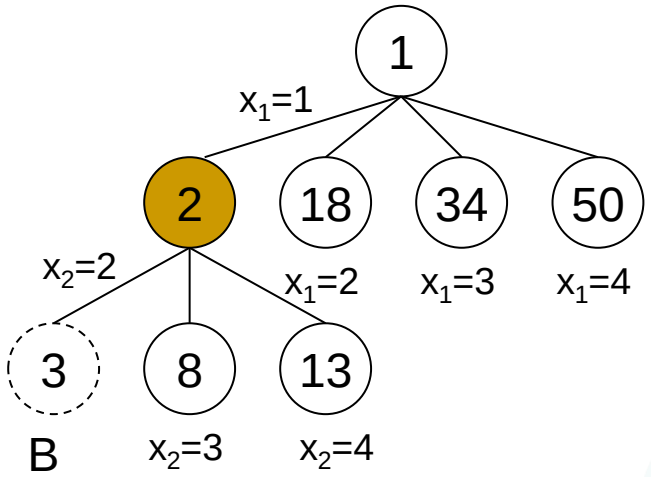
E结点

1

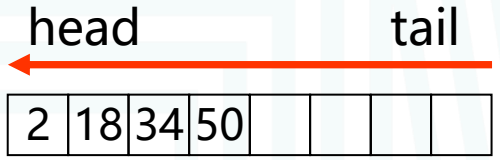
2

扩展E结点得到的状态空间树



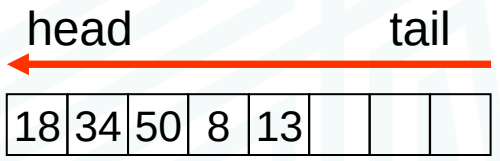


活结点表（队列）



扩展结点1，得新结点2，18，34，50

活结点2、18、34、50入队列



扩展结点2，得新结点3，8，13

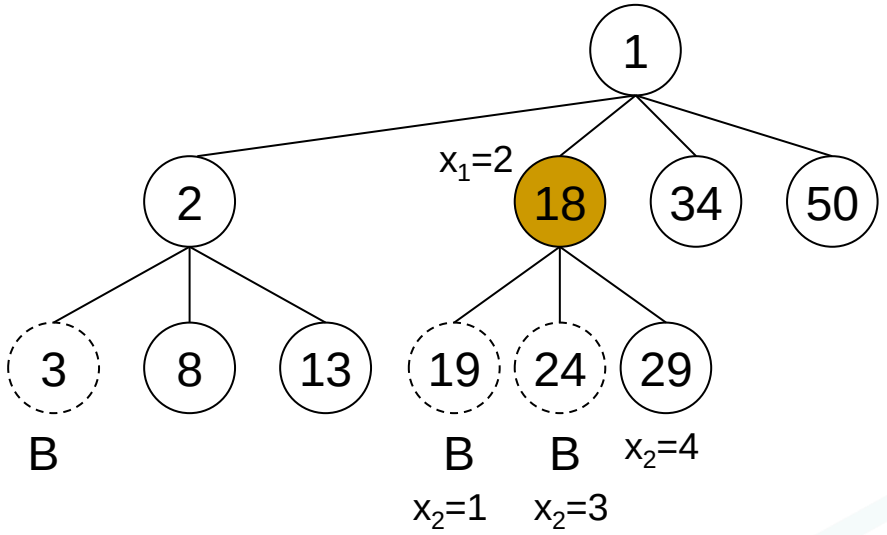
利用限界函数杀死结点3

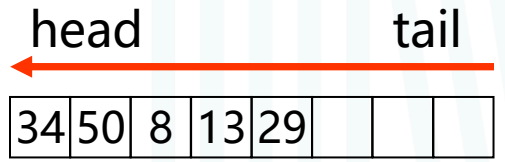
活结点8、13入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）

18





扩展结点18，得新结点19，24，29

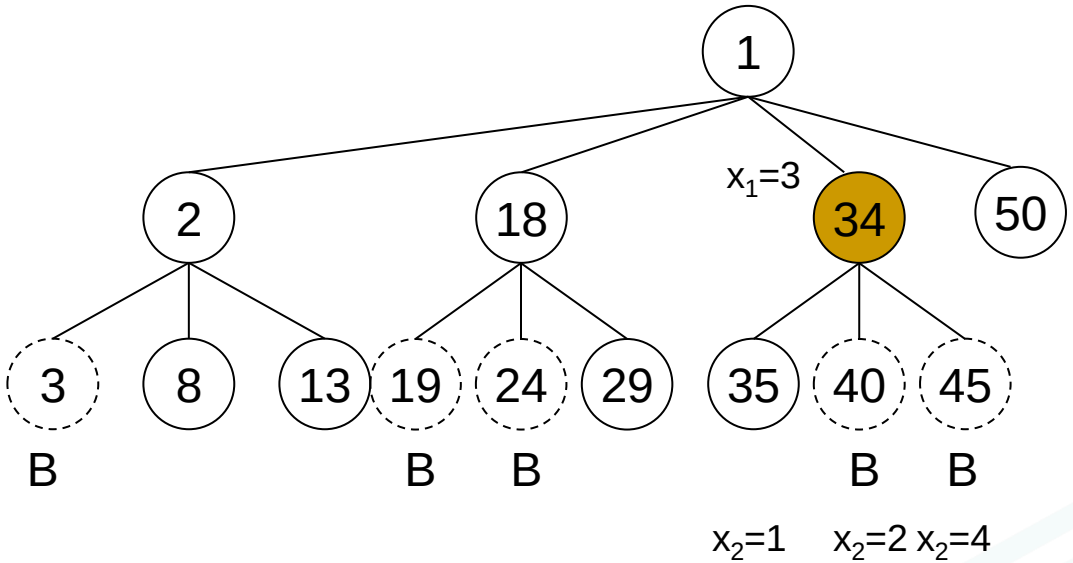
利用限界函数杀死结点19、24

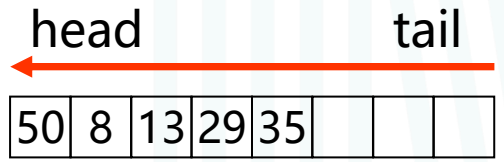
活结点29入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）

34





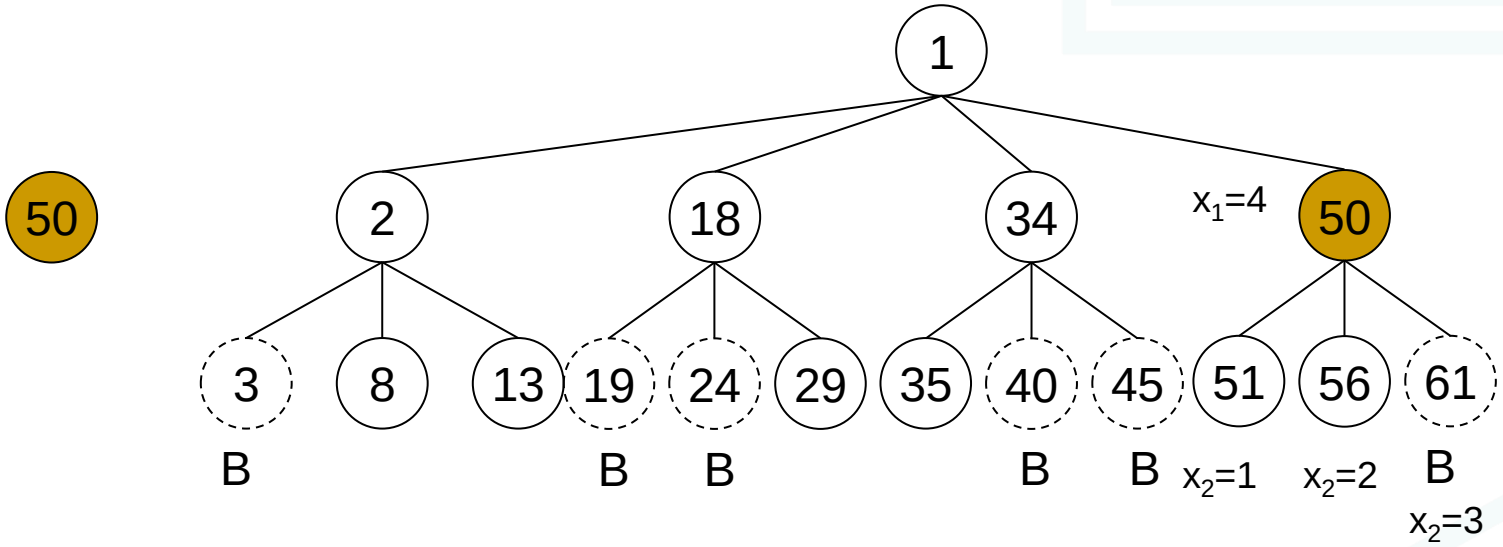
扩展结点34，得新结点35，40，45

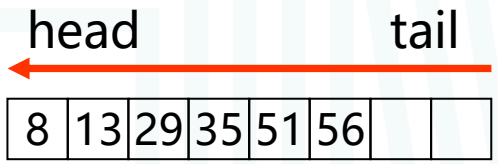
利用限界函数杀死结点40、45

活结点35入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





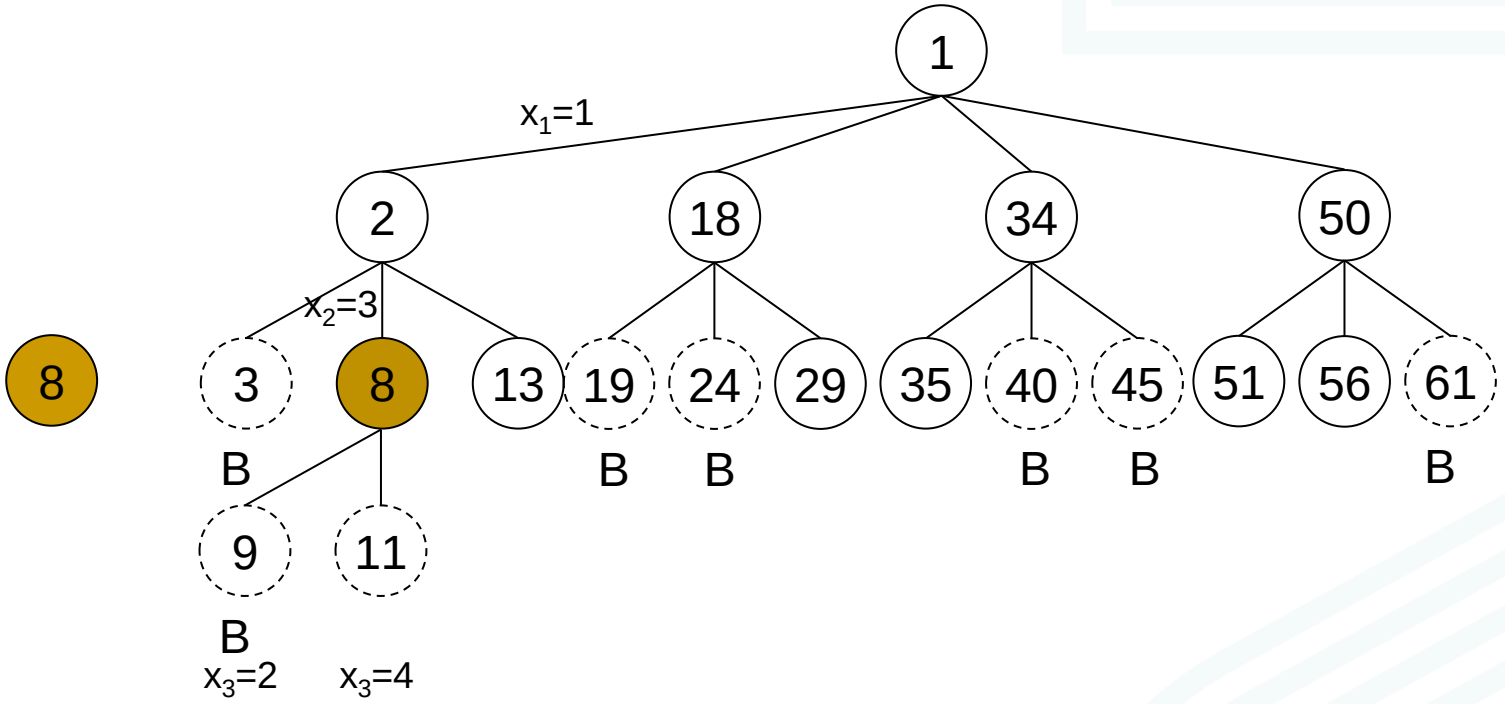
扩展结点50，得新结点51，56，61

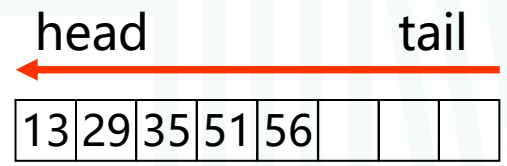
利用限界函数杀死结点61

活结点51、56入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





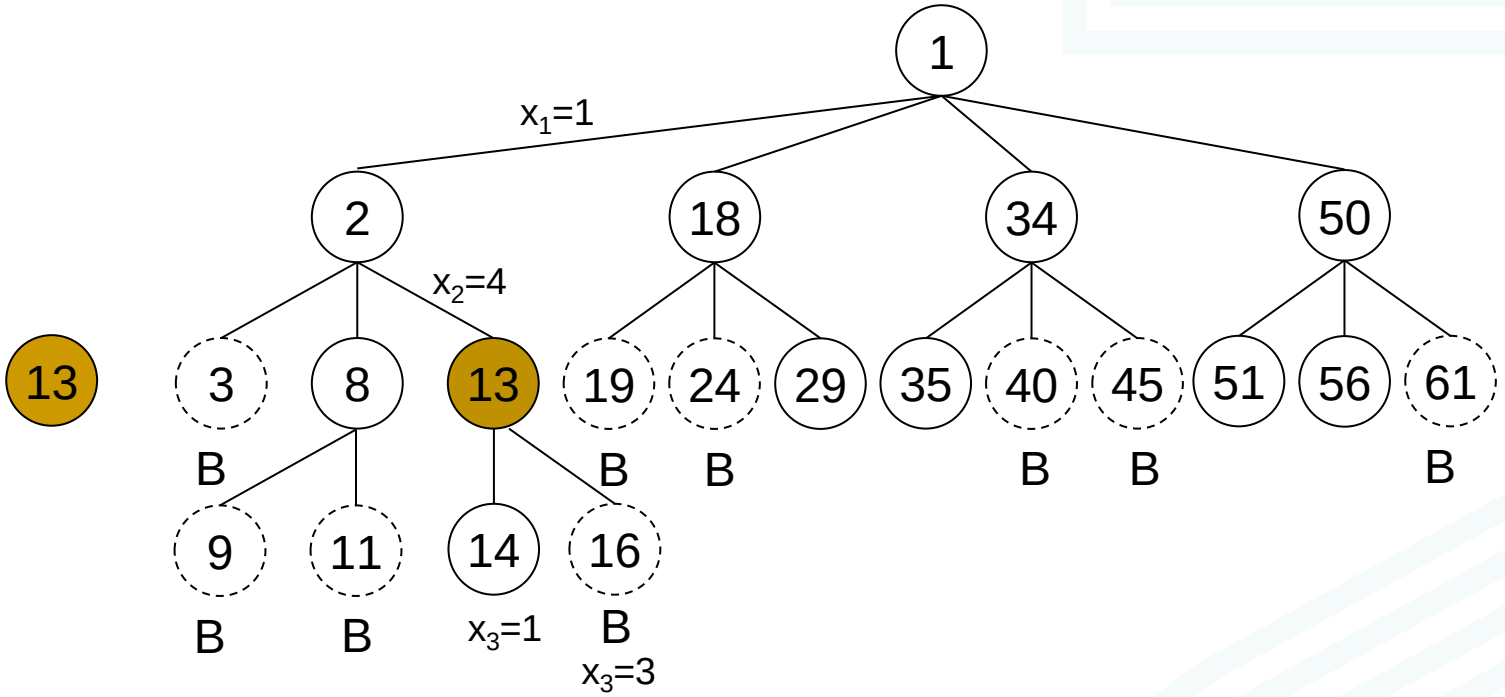
扩展结点8，得新结点9，11

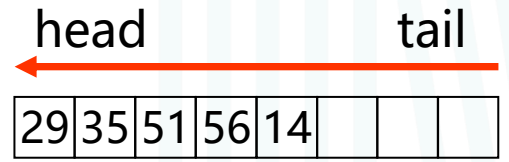
利用限界函数杀死结点9、11

没有新的活结点入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





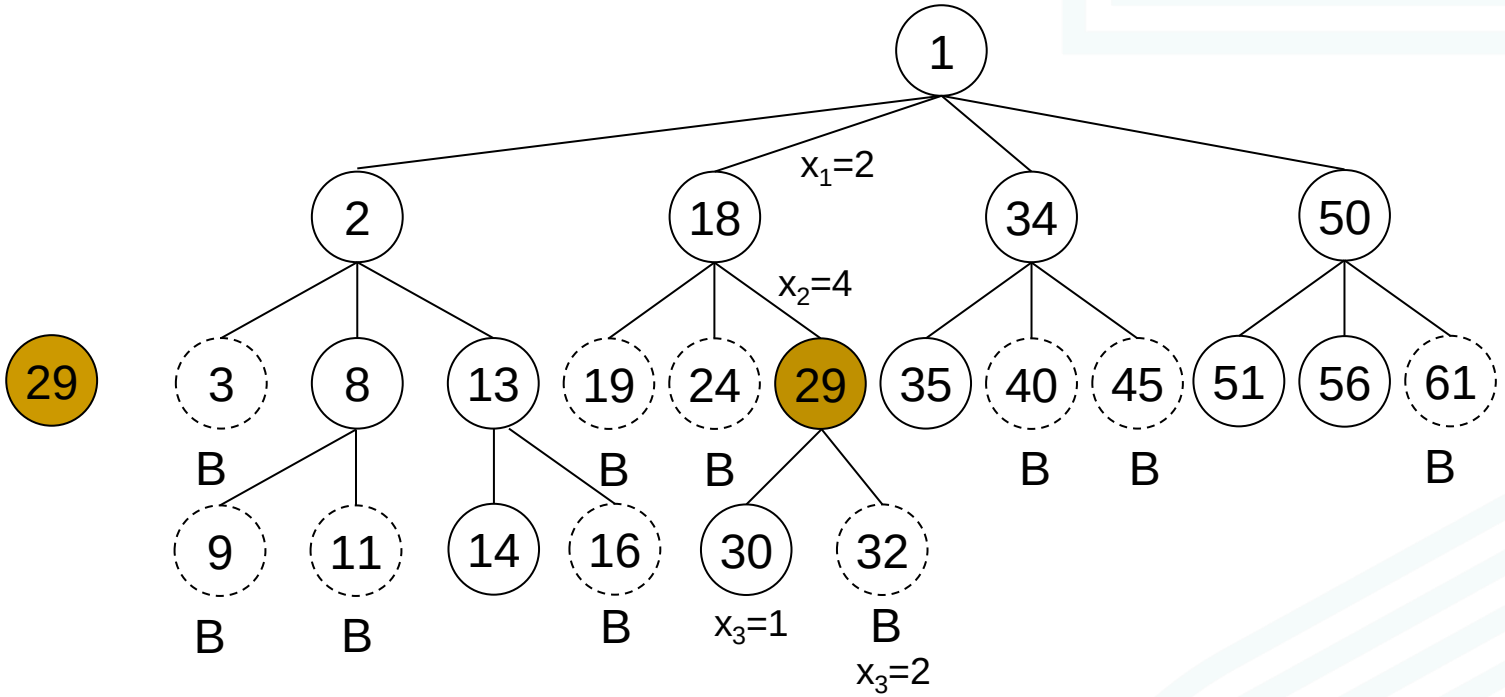
扩展结点13，得新结点14，16

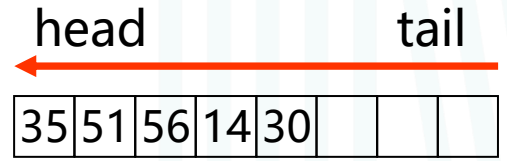
利用限界函数杀死结点16

活结点14入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





扩展结点29，得新结点30，32

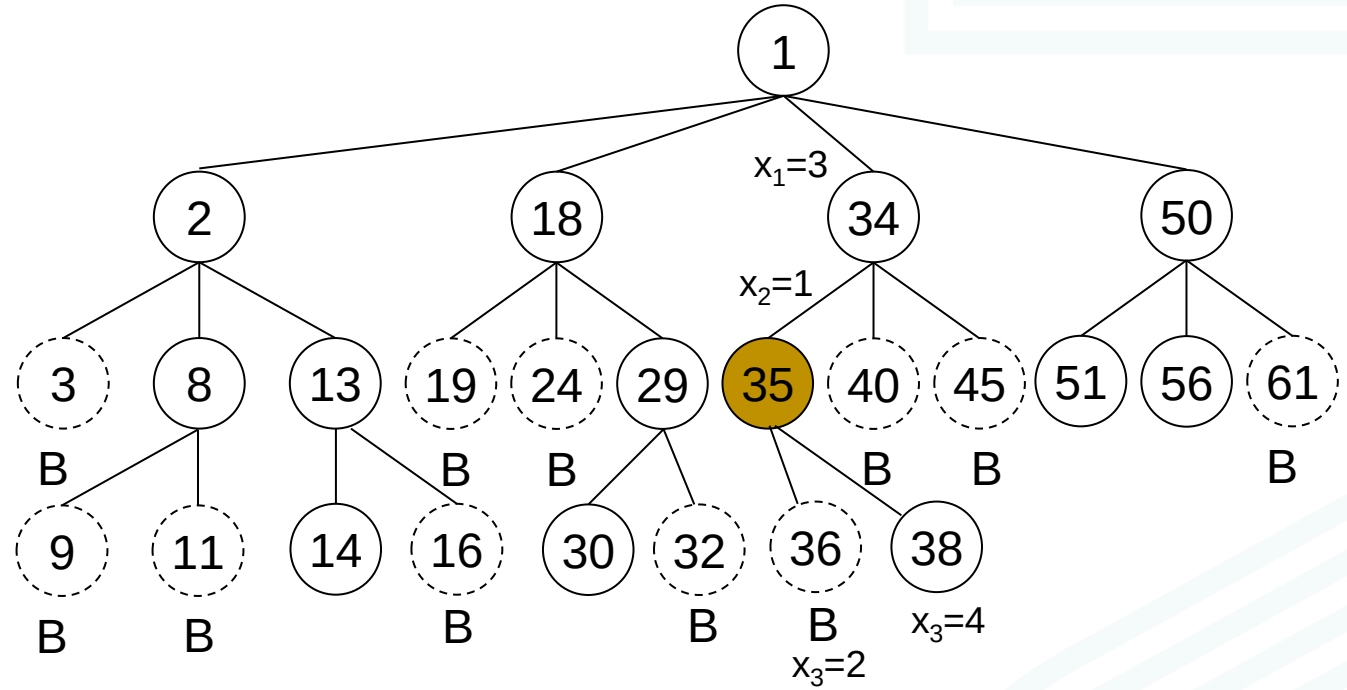
利用限界函数杀死结点32

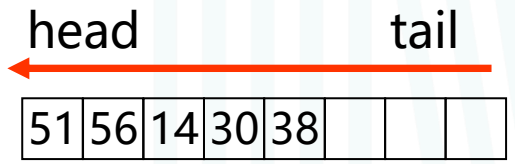
活结点30入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）

35





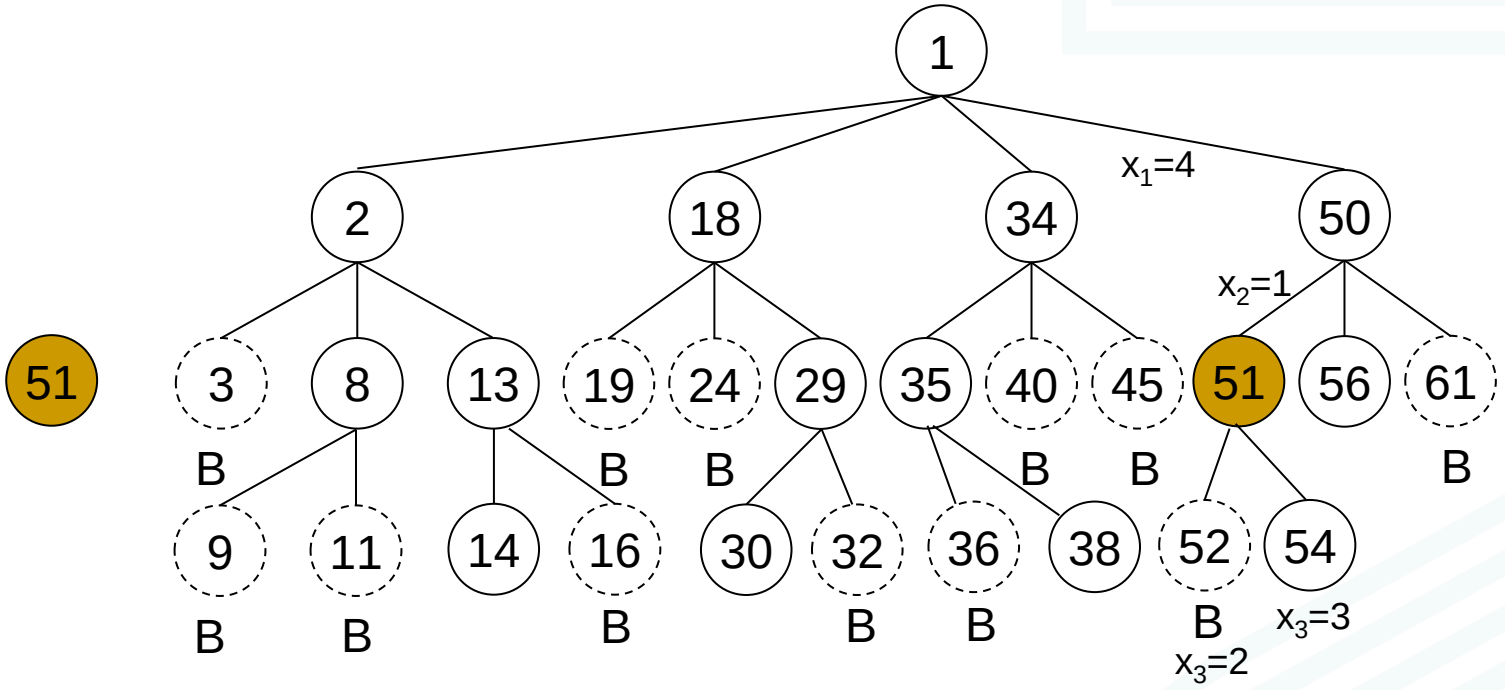
扩展结点35，得新结点36，38

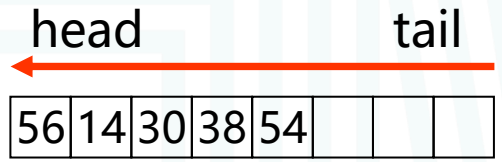
利用限界函数杀死结点36

活结点38入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





扩展结点51，得新结点52，54

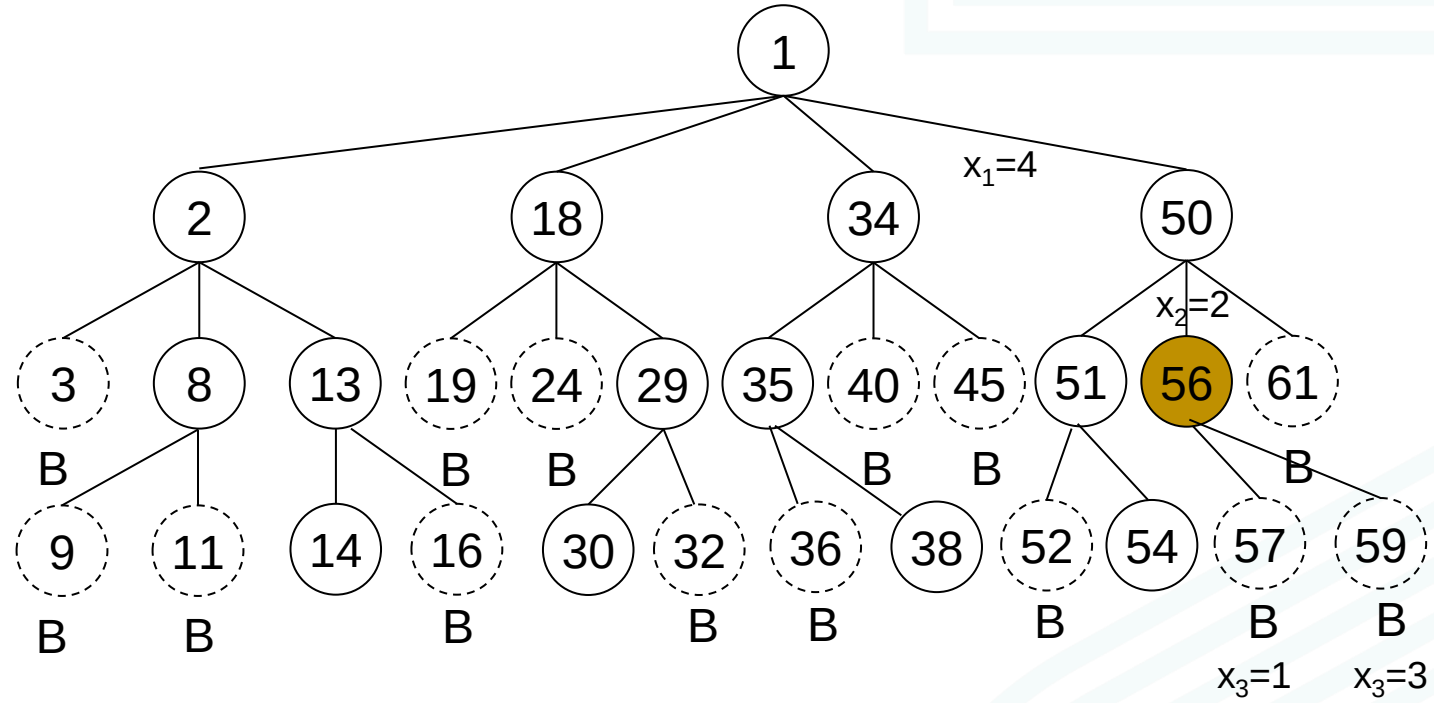
利用限界函数杀死结点52

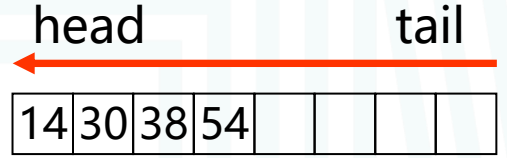
活结点54入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）

56





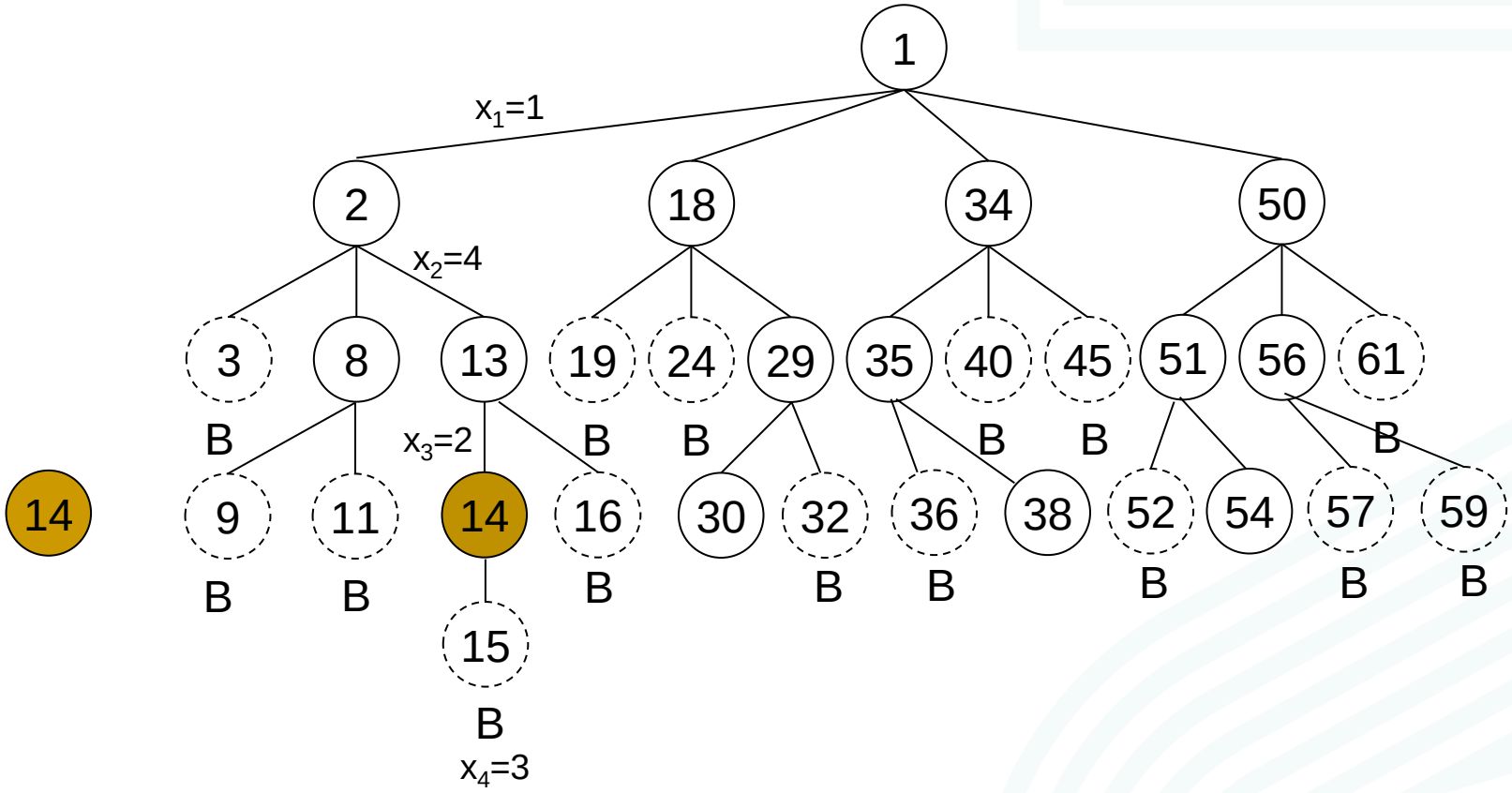
扩展结点56，得新结点57，59

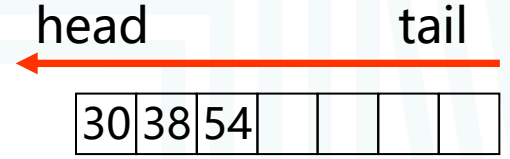
利用限界函数杀死结点57、59

没有新的活结点入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）





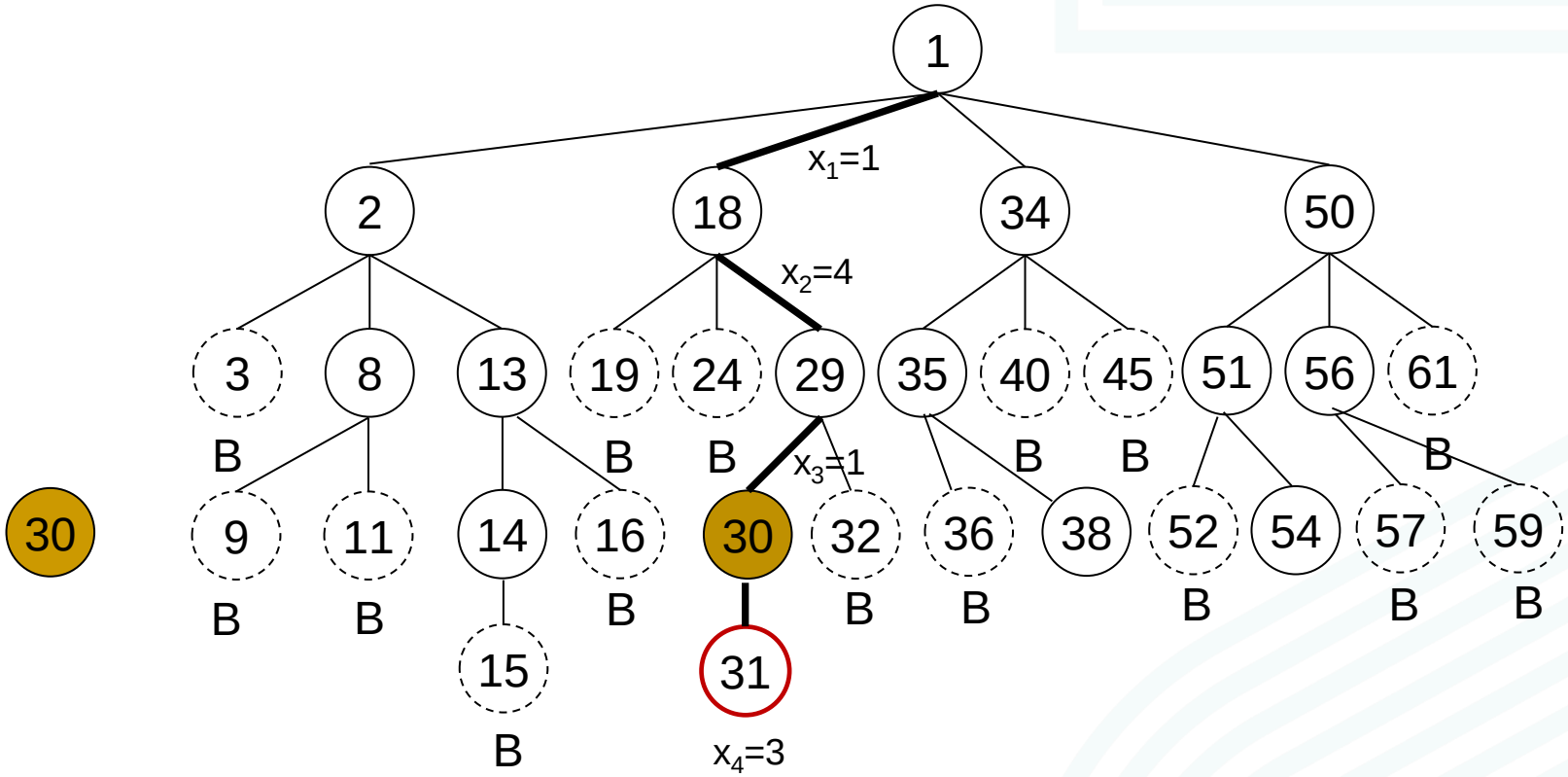
扩展结点14，得新结点15

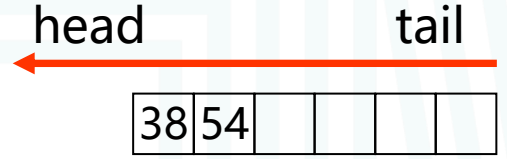
利用限界函数杀死结点15

没有新的活结点入队列

4皇后问题 FIFO分支限界 状态空间树

E结点　　扩展E结点得到的状态空间树　　活结点表（队列）



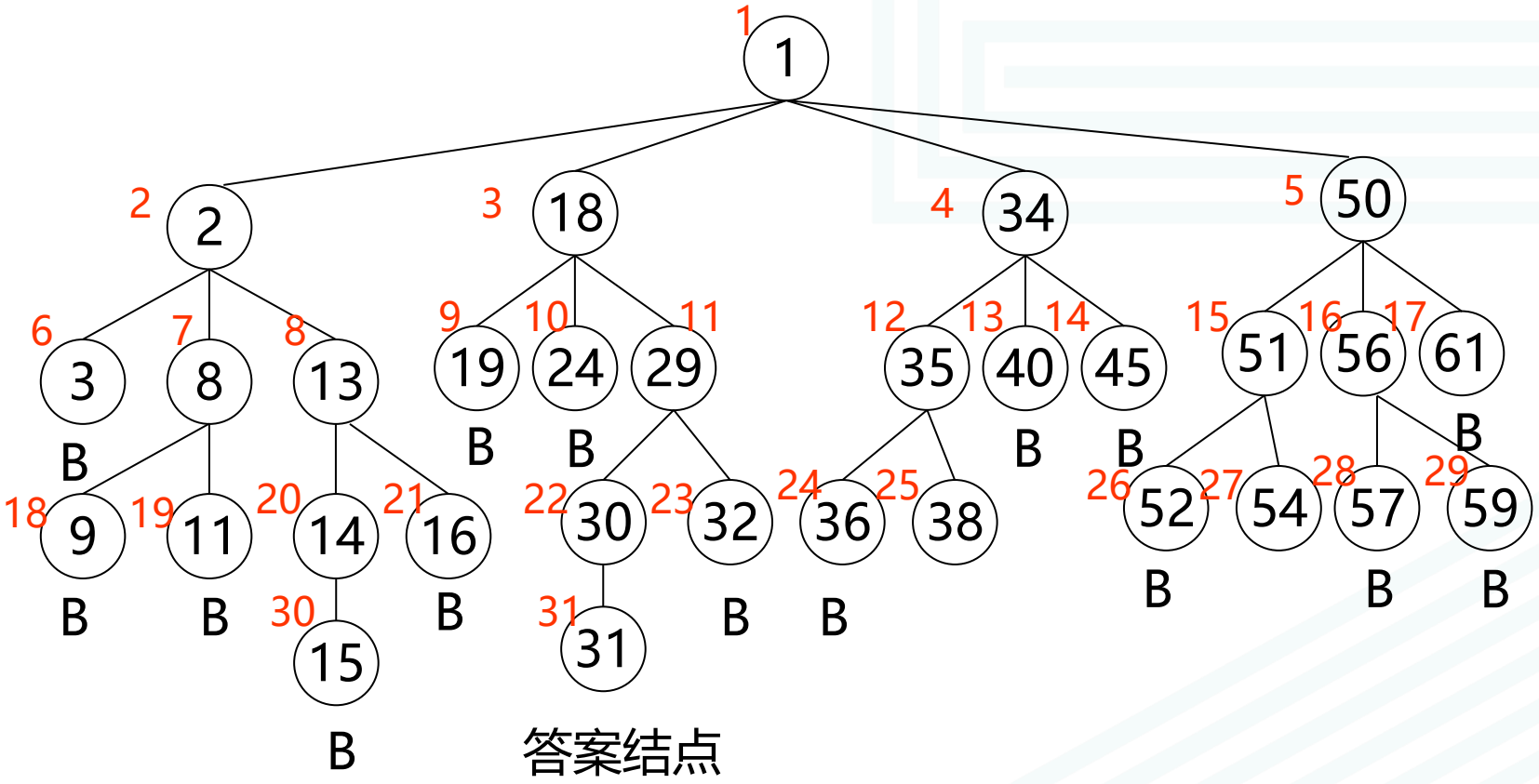


扩展结点30，得新结点31

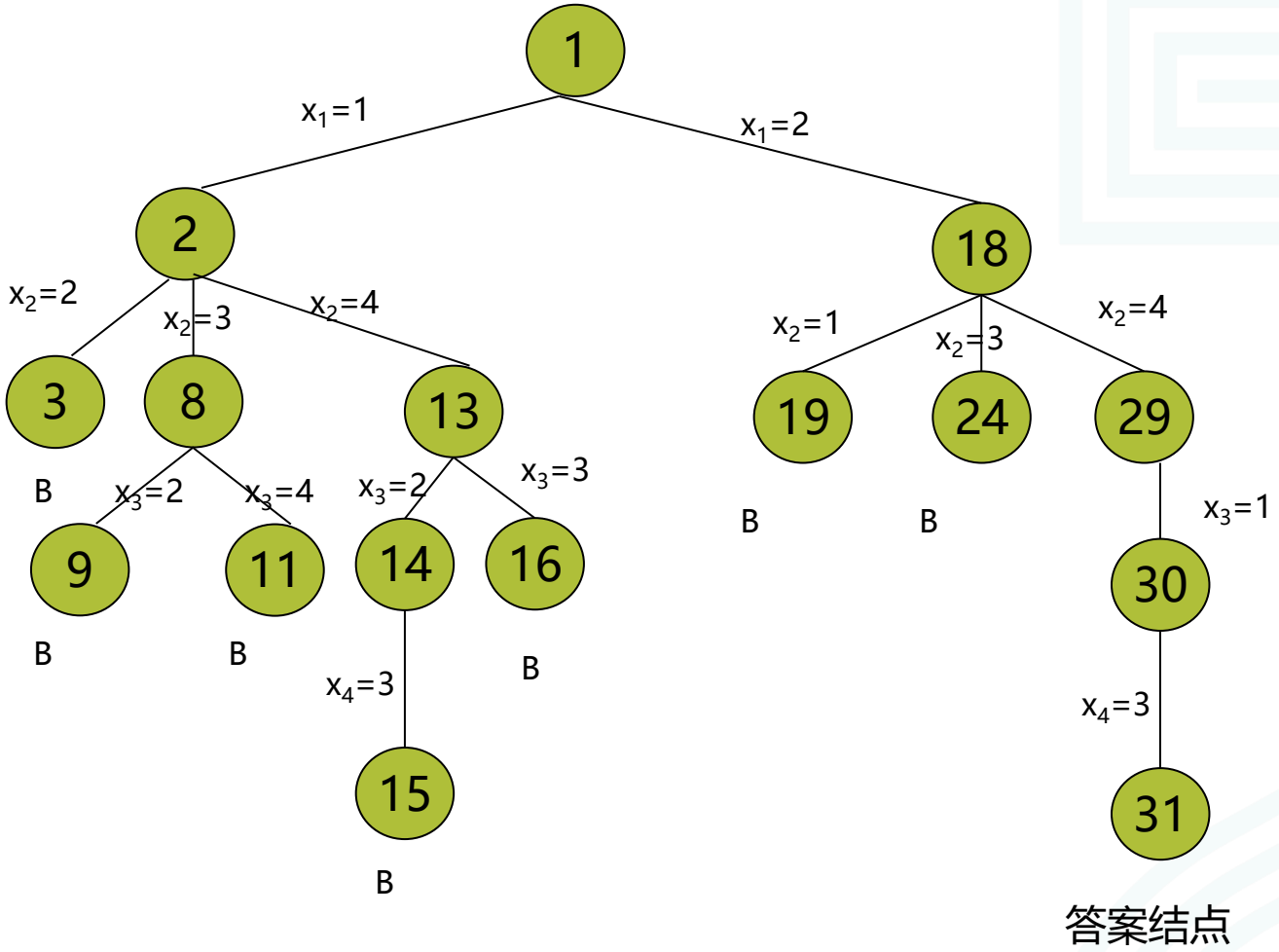
结点31为答案结点

算法终止（找到一个答案）

4皇后问题 FIFO分支限界 状态空间树



4皇后问题 FIFO分支限界 状态空间树 对比



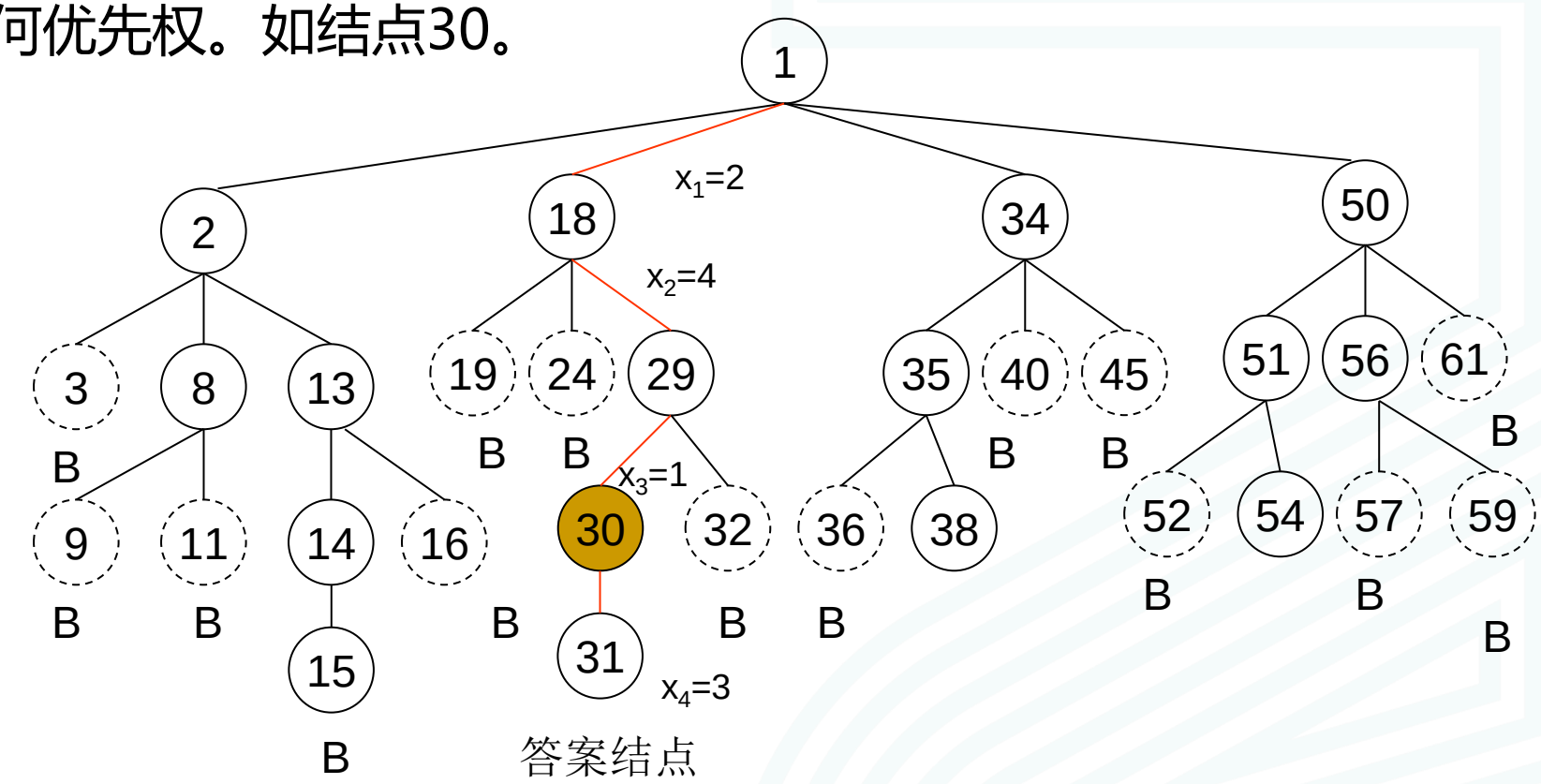
回溯

Win！

LC-检索 （Least Cost， A*算法）

- LIFO和FIFO分支-限界法存在的问题

对下一个E-结点的选择规则过于死板。对于有可能快速检索到一个答案结点的结点没有给出任何优先权。如结点30。



LC-检索 （Least Cost， A*算法）

- 如何解决?
  - 做某种排序，让可以导致答案结点的活结点排在前面!
  - 新问题：**怎么排序?**
  - **寻找一种“有智力”的排序函数C(•)**， 用C(·)来选取下一个E结点，加快到达一答案结点的检索速度。

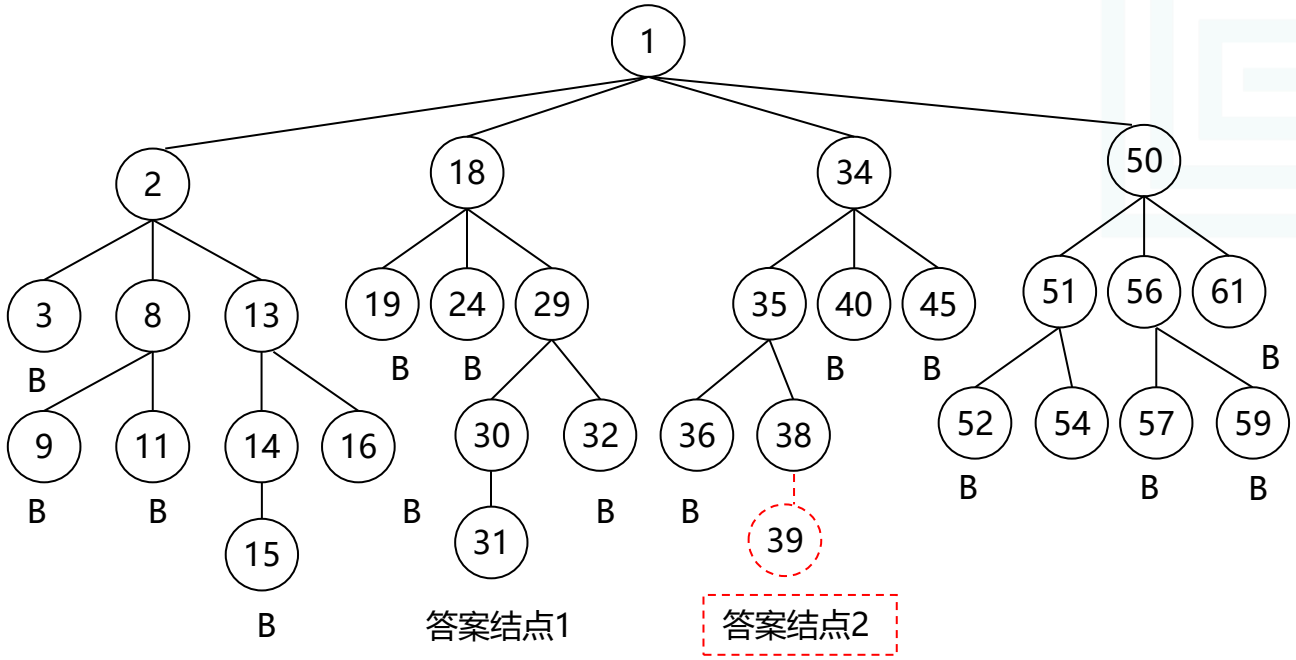
如结点30，29→30→31

能否赋予一个比其它活结点高的优先级使其尽快成为E结点?

LC-检索 （Least Cost， A*算法）

- 如何衡量结点的优先级？
  - 对于任一结点，用该结点导致答案结点的成本（代价）来衡量该结点的优先级——**成本越小越优先**
  - 对任一结点X，可以用两种标准来衡量结点的代价：
    1）在生成一个答案结点之前，子树X需要生成的结点数。
    2）在子树X中离X最近的那个答案结点到X的路径长度。

4皇后问题 代价



例：在量度2)下各结点的代价:

结点	代价
1	4
18，34	3
29，35	2
30，38	1
其余结点(除31、39)	≥ 3，2，1

量度2)：在子树X中离X最近的那个答案结点到X的路径长度。

偏向于选择由根到最近的那个答案结点路径上的结点作为E结点进行扩展

结点成本函数

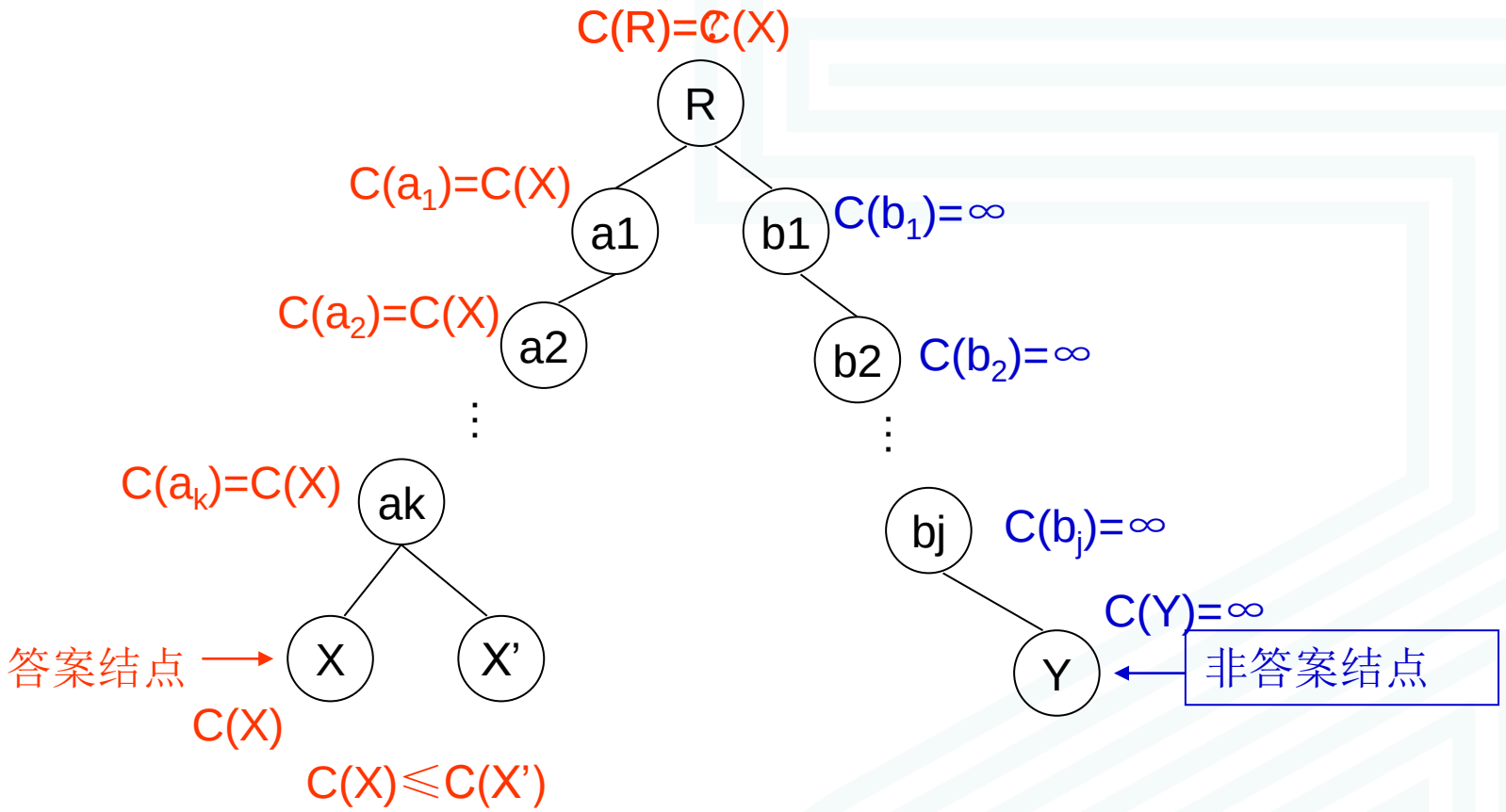
- C(·)：“有智力”的排序函数，依据成本排序，优先选择成本最小的活结点作为下一个E结点进行扩展。C(·)又称为“结点成本函数”
- 结点成本函数C(X)的取值：

1）如果X是答案结点，则C(X)是由状态空间树的根结点到X的成本(即所用的代价，可以是级数、计算复杂度等)。

2) 如果X不是答案结点且子树X不包含任何答案结点，则C(X) = ∞

3) 如果X不是答案结点但子树X包含答案结点，则C(X)应等 于子树X中具有最小成本的答案结点的成本

结点成本函数



结点成本函数的计算

结点成本函数 问题

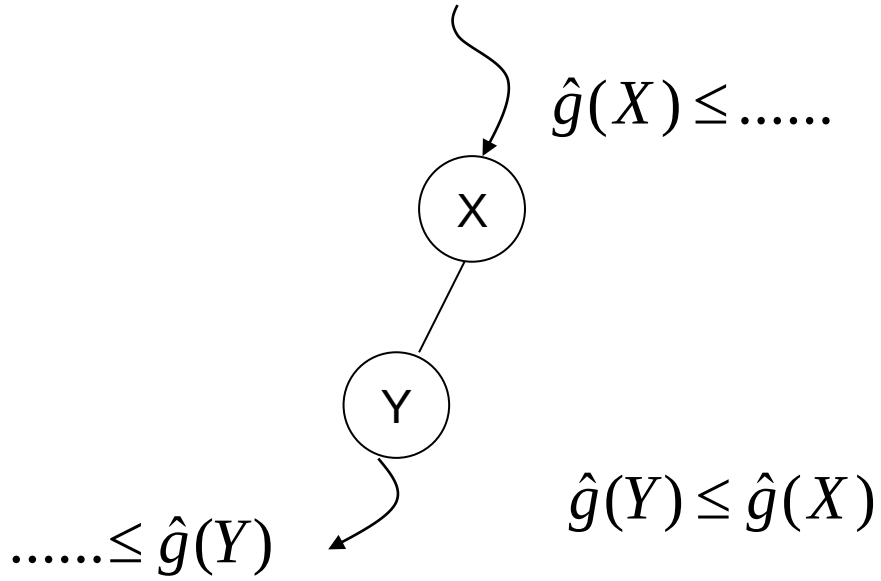
- 计算结点X的代价通常要检索子树X才能确定，因此计算C(X)的工作量和复杂度与解原始问题是相同的。
- 计算结点成本的精确值是不现实的——相当于求解原始问题。怎么办？
- 结点成本的估计函数 $\hat{c}(X)$

  包括两部分：$\hat{g}(X)$ 和 *h(X)*

结点成本函数

$\hat{g}(X)$ 是由X到达一个答案结点所需成本的估计函数。

性质：单纯使用 $\hat{g}(X)$ 选择E结点会导致算法偏向纵深检查。



$\hat{g}(\bullet)$ 是X到答案结点的最小成本

纵深检查:

1）如果 $\hat{g}(X)=C(X)$，最理想！

2）否则，可能导致不能很快地找到更靠近根的答案结点。

特例： Z比W更接近答案结点，

但 $\hat{g}(W)<\hat{g}(Z)$ 。

结点成本函数 纵深检查

如何避免单纯考虑$\hat{g}(X)$造成的纵深检查?

- 引进*h(X)* 改进成本估计函数。
- *h(X)*：根结点到结点X的成本——已发生成本。

改进的**结点成本估计函数** $\hat{c}(X)$

$$\hat{c}(X) = f(h(X)) + \hat{g}(X)$$

- f(·)是一个非降函数。
- 非零的f(·)可以减少算法作偏向于纵深检查的可能性，它迫使算法优先检索**更靠近答案结点**但又**离根较近**的结点。

LC-检索

选择 $\hat{c}(\bullet)$ 值最小的活结点作为下一个E-结点的状态空间树检索方法。

(Least Cost Search)

特例：

- BFS： 依据级数来生成结点，令 $\hat{g}(X)=0$； f (h(X)) = X 的级数
- D-Search：令 f (h(X)) = 0；而当Y是X的一个儿子时，
  总有 $\hat{g}(X) \geq \hat{g}(Y)$。

LC分支-限界检索：带有限界函数的LC-检索

LC-检索

设： T是一棵状态空间树

- c(X)是T的结点成本函数
- $\hat{c}(X)$ 是成本估计函数
- 如果X是一个答案结点或者是一个叶结点，则c(X)= $\hat{c}(X)$ 。

LC过程用 $\hat{c}(\bullet)$ 去寻找一个答案结点

LC-检索

```
procedure LC(T, ĉ )      //为找答案结点检索T，ĉ 为结点成本估计函数//
  if T是答案结点 then 输出T; return endif   //T为答案结点，输出T//
  E ← T    //E - 结点//
  将活结点表初始化为空
  loop
      for E的每个儿子X do
          if X是答案结点 then 输出从X到T的路径；return endif
          call ADD(X)  //X是新的活结点，ADD将X加入活结点表中//
          PARENT(X) ← E //指示到根的路径//
      repeat
      if 不再有活结点 then   print(“no answer code”); stop endif
      call LEAST(E)  //从活结点表中找 ĉ 最小的活结点，赋给X，并从活结点表中删除//
  repeat
end LC
```

找到答案结点，输出到根的路径

LEAST(X)：在活结点表中找一个具有最小成本估计值的活结点，从活结点表中删除这个结点，并将此结点放在变量X中返回。

ADD(X)：将新的活结点X加到活结点表中。

活结点表：以**min-堆**结构（优先队列）存放。

LC-检索 对比

1）FIFO-检索及D-检索是LC算法的特例：

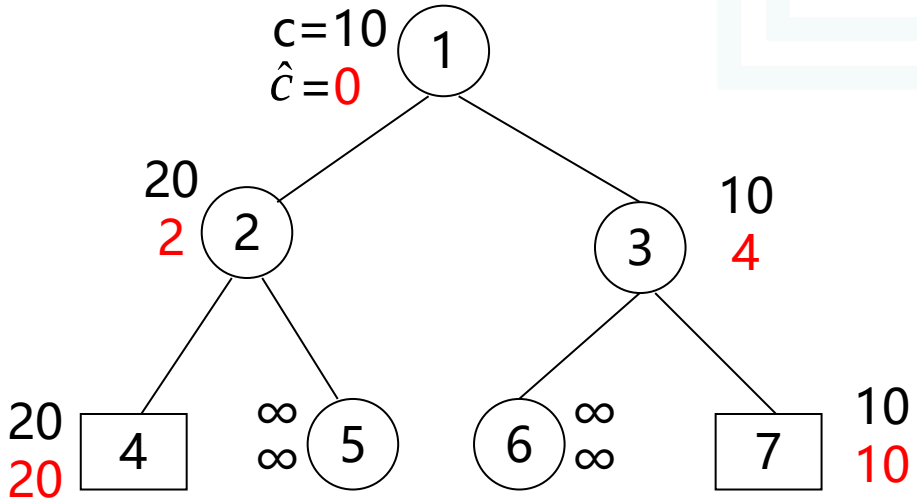
- ⊙ 若活结点表采用队列，用LEAST(X)和ADD(X)从队列中 删除或加入元素，并**依据级数**来生成结点，即令 $\hat{g}(X) = 0$；f (h(X)) = X的级数，则LC-检索就变成了 FIFO-检索。
- ⊙ 若活节点表采用栈，用LEAST(X)和ADD(X)从栈中删除或加入元素，并**令f (h(X)) = 0**；而当Y是X的一个儿子时，总有 $\hat{g}(X) \geq \hat{g}(Y)$，则LC就变成了 D-检索。

2）算法的不同之处在于：对下一个E-结点的选择规则不同。

LC-检索 特性

- 当有多个答案结点时，LC是否一定找得到具有最小成本的答案结点呢?

否



原因：可能存在这样的结点X和Y：c(X) > c(Y)，但 $\hat{c}(X) < \hat{c}(Y)$

- 首先扩展结点1，得结点2，3；$\hat{c}(2) = 2 < \hat{c}(3) = 4$；
- 然后扩展结点2，得答案结点4，c(4)=20;
- 实际最小成本的答案结点是7，c(7) = 10

LC-检索 改进

改进策略1：

- 约定：对每一对c(X) < c(Y)的结点X和Y，有 $\hat{c}(X) < \hat{c}(Y)$
- 目标：使得LC能够找到一个最小成本的答案结点。

定理： 在有限状态空间树T中，对于每一个结点X，令$\hat{c}(X)$是c(X)的估计值且具有以下性质：

对于每一对结点Y、Z，当且仅当 c(Y)<c(Z) 时，有 $\hat{c}(Y) < \hat{c}(Z)$ 。在使用 $\hat{c}(X)$ 作为c(X)的估计值时，算法LC到达一个最小的成本答案结点终止。

证明： （略）

LC-检索 改进

改进策略2：

对结点成本函数做如下限定：对于每一个结点X有 $\hat{c}(X) \le c(X)$

且对于答案结点X有 $\hat{c}(X) = c(X)$

算法LC1：找最小成本答案结点的改进算法，该算法可以找到成本最小的答案结点。

LC-检索 最小成本答案结点算法

```
procedure LC1(T, ĉ )
 //为找出最小成本答案结点检索T, ĉ 为具有上述性质的结点成本估计函数//
  E ← T    //第一个E - 结点//
  置活结点表为空
  loop
      if E是答案结点 then 输出从E到T的路径；return endif
      for E的每个儿子X do
          call ADD(X)  //X是新的活结点，ADD将X加入活结点表中//
          PARENT(X) ← E //指示到根的路径//
      repeat
      if 不再有活结点 then    print(“no answer code”); stop endif
      call LEAST(E)  //从活结点表中找    最小的活结点，赋给X，并从活结点表中删除//
  repeat
end LC1
```

LC-检索 最小成本答案结点算法

估计函数选择不同，对寻路结果有哪些影响呢?

1、**当估算的距离** $\hat{g}(X)$ **完全等于实际距离时**，也就是每次扩展的那个点都准确的知道选它以后，路径距离是多少，这样就不用乱选了，每次都选最小的那个，一路下去，肯定就是最优的解，而且基本不用扩展其它的点。

2、**如果估算距离** $\hat{g}(X)$ **小于实际距离时**，则到最后一定能找到一条最短路径，但是有可能会经过很多无效的点。

3、**如果估算距离** $\hat{g}(X)$ **大于实际距离时**，有可能就很快找到一条通往目的地的路径，但是却不一定是最优的解。

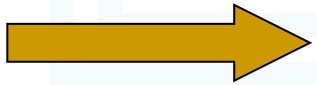

华中科技大学计算机科学与技术学院
School of Computer Science &Technology,HUST

15-迷问题

描述

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

（a）一种初始排列



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

（b）目标排列

问题描述：在一个分成16格的方形棋盘上放有15块编了号的牌。对于这些牌给定的一种初始排列（如图(a)），要求通过一系列的**合法移动**将初始排列转换成目标排列（图(b)）。

合法移动：每次将一个邻接于空格的牌移动到空格位置。

描述

- 问题状态：15块牌在棋盘上的任一种排列。

  初始状态：初始排列（任意给定的）

  目标状态：目标排列（确定的）

- 棋盘存在**16!**种（约20万亿种）不同排列。
  - 对于一给定的初始状态，可达状态约为这些排列的一半。

描述

- **目标状态是否可由初始状态到达?**
  - 若由初始状态到某状态存在一系列合法移动，则称该状态可由初始状态到达。
  - 对于15-谜问题，并不是所有的初始状态都能变换成目标状态的。

判定 目标状态在初始状态的状态空间中

1）给棋盘的方格位置编号：按目标状态各块牌在棋盘上的排列给对应方格编号，空格为16。

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

目标排列

判定 目标状态在初始状态的状态空间中

2）记POSITION(i)为编号为i的牌在初始状态中的位置

POSITION(16)表示空格的位置。

例图(a)

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

POSITION(1:16)= (1,5,2,3,7,10,9,13,14,15,11,8,16,12,4,6)

1出现在1的位置，2出现在5的位置 ....

判定 目标状态在初始状态的状态空间中

3）记LESS(i)是这样牌j的数目

j < i，但POSITION(j) > POSITION(i)，

即：编号小于i但初始位置在i之后的牌的数目。

例：LESS(1)=0; LESS(4)=1; LESS(12)=6

1 2 3 < 4 且 P(1)=1 P(2)=5 P(3)=2 P(4)=3

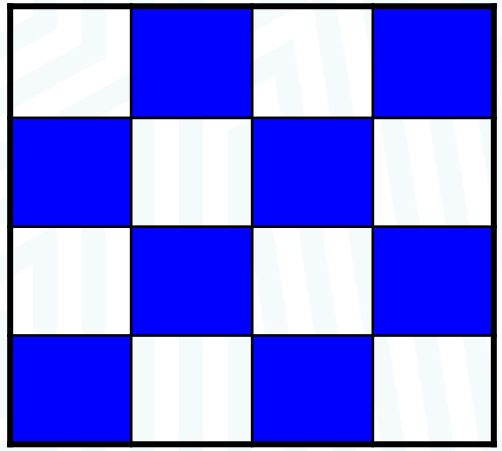
1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

4）引入一个量 X

如图所示，初始状态时，

若空格落在**蓝色**方格上，则X = 1；

若空格落在**白色**方格上，则X = 0。



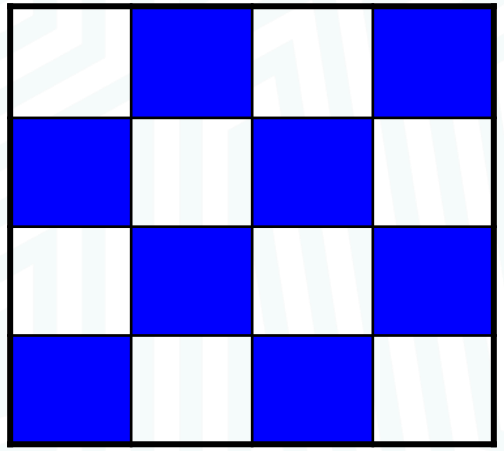
判定 目标状态在初始状态的状态空间中

- 目标状态是否在初始状态的状态空间中的判别条件由以下定理给出：

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

定理：当且仅当 $\sum_{i=1}^{16} LESS(i) + X$ 是偶数时，目标状态可由此初始状态到达。

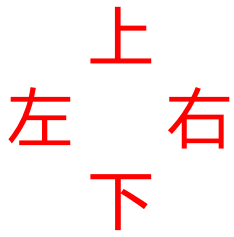
证明 （略）



构造 状态空间树

从初始状态出发，每个结点的儿子结点表示由该状态通过一次合法的移动而到达的状态。

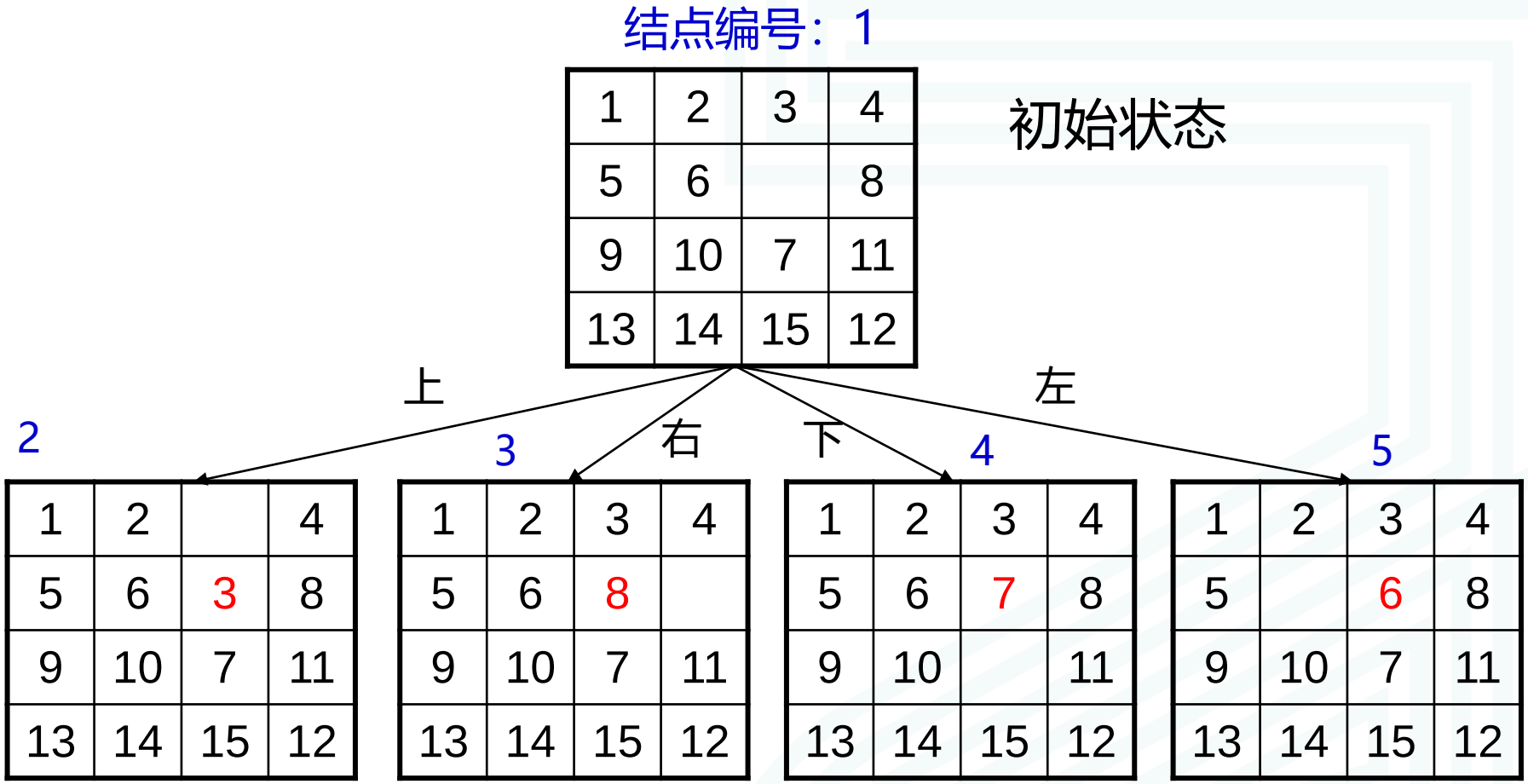
注：移动牌与移动空格是等效的，以空格的移动表示合法移动。空格的一次合法移动有四个可能的方向：



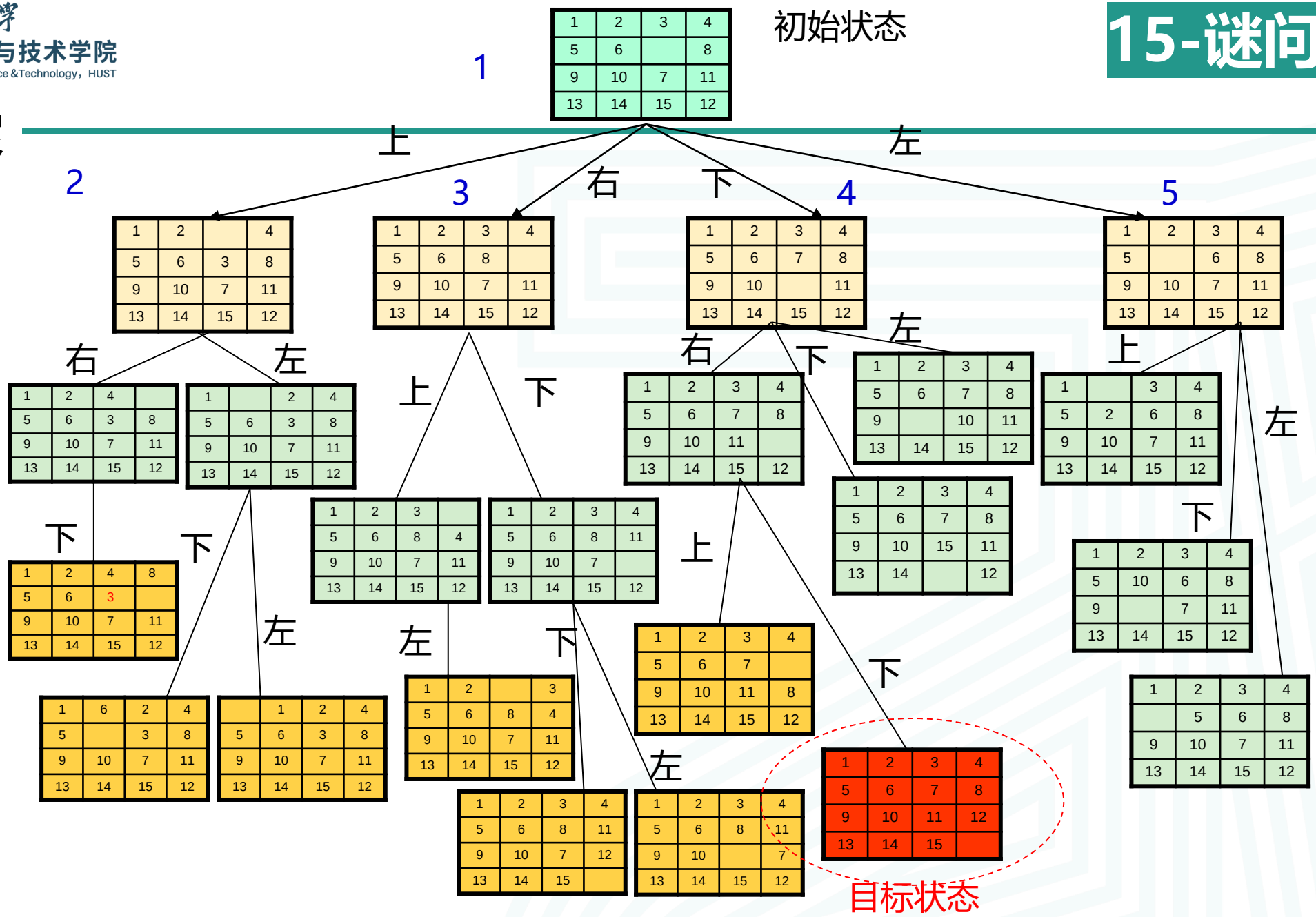
1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

剪枝策略：若P的儿子中有与P的父结点重复的，则剪去该分枝。

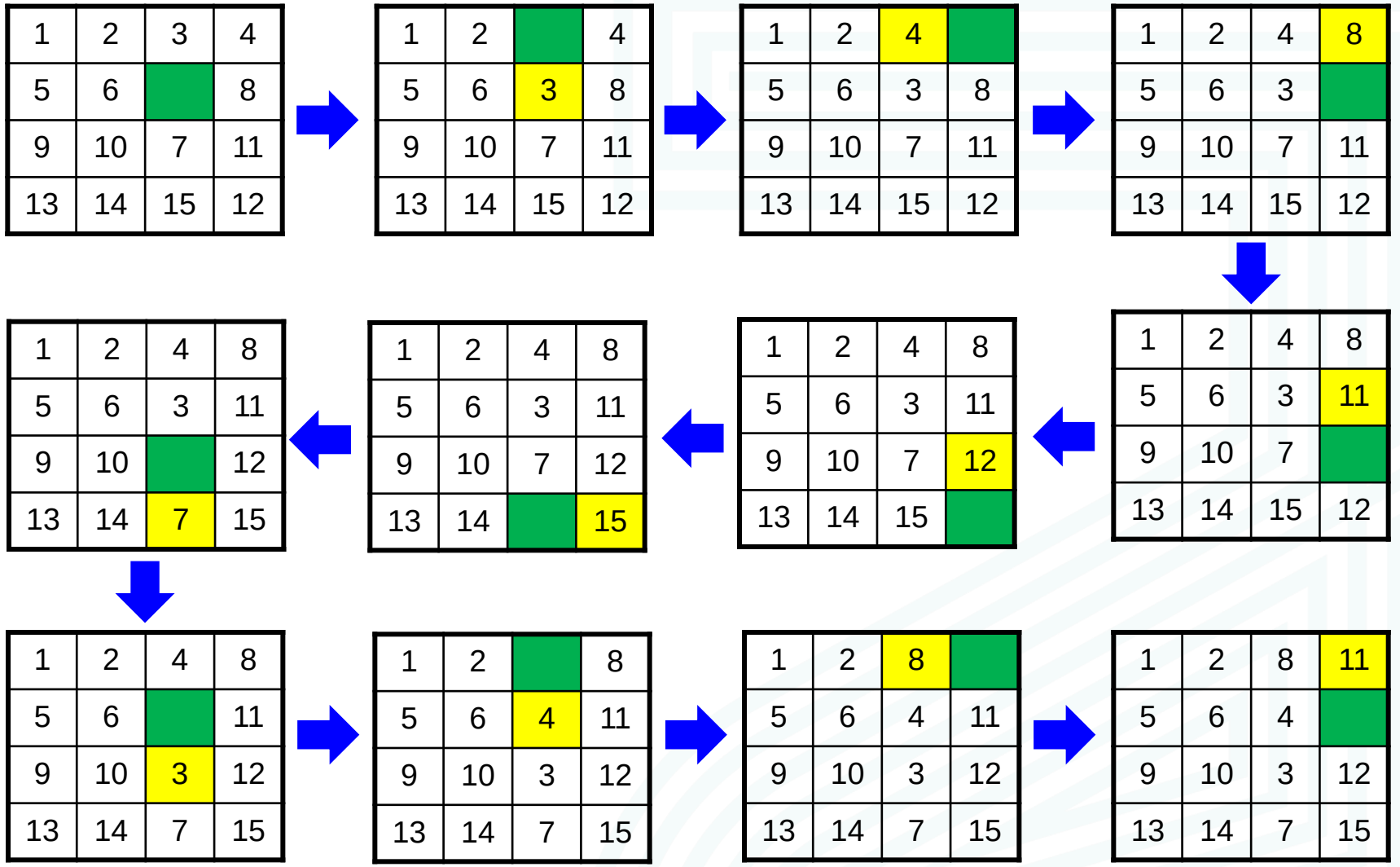
FIFO检索



FIFO检索



深度优先检索（部分结点）


1 2 3 4
5 6 8
9 10 7 11
13 14 15 12
1 2 4
5 6 3 8
9 10 7 11
13 14 15 12
1 2 4 8
5 6 3
9 10 7 11
13 14 15 12
1 2 4 8
5 6 3 11
9 10 7
13 14 15 12
1 2 4 8
5 6 3 11
9 10 7 12
13 14 15
1 2 4 8
5 6 3 11
9 10 7 12
13 14 15
1 2 4 8
5 6 3 11
9 10 12
13 14 7 15
1 2 4 8
5 6 11
9 10 3 12
13 14 7 15
1 2 8
5 6 4 11
9 10 3 12
13 14 7 15
1 2 8 11
5 6 4
9 10 3 12
13 14 7 15

问题

呆板和盲目

回顾：

是否能够找到一种“智能”方法，对不同的实例做不同的处理？

策略：给状态空间树中的每个结点赋予成本值c(X)

如何定义c(X)？

如果实例有解，则将**由根出发到最近目标结点的路径长度**作为成本赋给路径上的每个结点。

问题

该方法实际上是不可操作的——c(X)不可能通过简单的方法求出。精确计算c(X)的工作量与求解原始问题相同

估算法：定义成本估计函数

$$\hat{c}(X) = f(X) + \hat{g}(X)$$

其中，

1）f(X)是由根到结点X的路径长度

2) $\hat{g}(X)$是以X为根的子树中由X到目标状态的一条最短路径长度的估计值——这里 $\hat{g}(X)$至少应是能把状态X转换成目标状态所需的最小移动数。故令，

$\hat{g}(X)$=不在其目标位置的非空白牌数目

Example

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

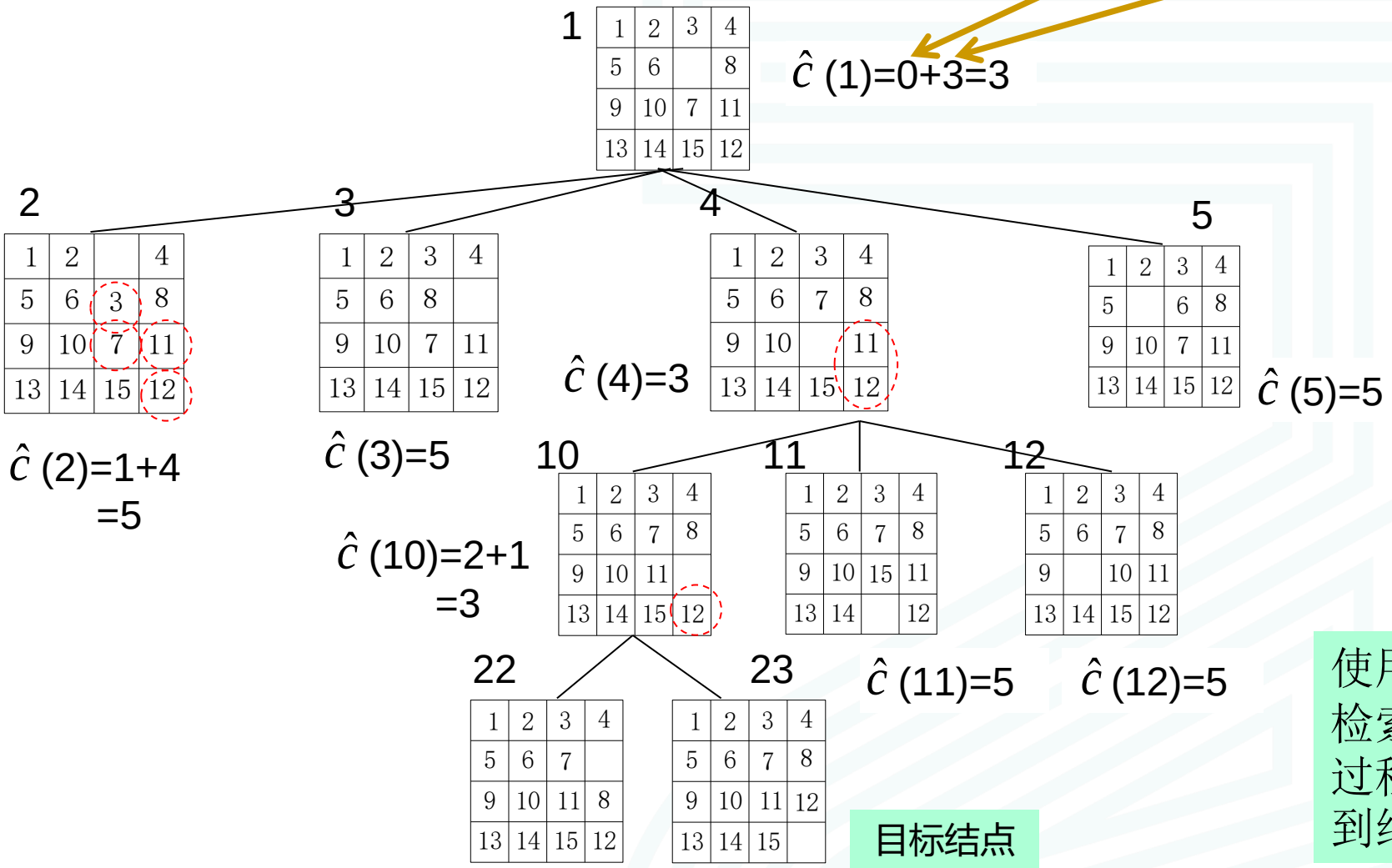
例：7、11、12号牌不在其位，故 $\hat{g}(X) = 3$

注：为达到目标状态所需的实际移动数>> $\hat{g}(X)$

$\hat{c}(X)$是c(X)的下界

Example 代价下LC-检索

$$\hat{c}(X) = f(X) + \hat{g}(X)$$



使用 $\hat{c}(X)$ 的LC-检索可以使检索过程很快地定位到结点23。

成本函数在分支-限界中的应用

假定每个答案结点X有一个与其相联系的c(X)，且找成本最小的答案结点

1）最小成本的下界

$\hat{c}(X)$ 为X的成本估计函数。当 $\hat{c}(X) \le c(X)$ 时，$\hat{c}(X)$ 给出了由结点X求解的最小成本的下界，作为启发性函数，减少选取**E结点**的盲目性。

2）最小成本的上界

能否定义最小成本的上界?

成本函数在分支-限界中的应用

最小成本的上界

定义U为最小成本解的成本上界，则：

作用：对具有 $\hat{c}(X) > U$ 的所有活结点可以被杀死，从而可以进一步使算法加速，减少求解的盲目性。

注：

根据c(X)的定义，由那些 $\hat{c}(X) > U$ 的结点X可到达的所有答案结点必有 $c(X) \geq \hat{c}(X) \geq U$ ，不可能具有更小的成本。故，当已经求得一个具有成本U的答案结点，那些有 $\hat{c}(X) > U$ 的所有活结点都可以被杀死。

成本函数在分支-限界中的应用

最小成本上界U的求取：

1）初始值：利用启发性方法赋初值，或置为∞

2）每找到一个新的答案结点后修正U，U取当前最小成本值。

注：只要U的初始值**不小于**最小成本答案结点的成本，利用U就不会杀死可以到达最小成本答案结点的活结点。

分支-限界求解最优化问题

- 最优化问题求能够使目标函数取极值的最优解。**如何把求最优解的过程表示成与分支-限界相关联的检索过程?**

- **可行解**：类似于n-元组的构造，把可行解可能的构造过程用“状态空间树”表示出来。
- **最优解**：把对最优解的检索表示成对状态空间树答案结点的检索。
- **成本函数**：每个结点赋予一个成本函数c(X)，并使得代表最优解的答案结点的c(X)是所有结点成本的最小值。

分支-限界求解最优化问题

直接用目标函数作为成本函数c(·)

1）代表可行解结点的c(X)就是该可行解的目标函数值。

2）代表不可行解结点的c(X)=∞；

3）代表部分解的结点的c(X)是根为X的子树中最小成本结点的成本。

- 答案结点 ⟷ 可行解
- 成本最小的答案结点 ⟷ 最优解
- 成本估计函数 $\hat{c}(X)$ 且要求有 $\hat{c}(X) \le c(X)$

  注： $\hat{c}(X)$ 根据目标函数进行估计。

分支-限界算法求带限期的作业排序问题

描述

假定有n个作业和一台处理机，作业i对应一个三元组（p_i,d_i,t_i）

其中，t_i：表示作业i需要t_i个单位处理时间；

d_i：表示完成期限；

p_i：表示若i在期限内未完成招致的罚款。

求解目标：从n个作业的集合中选取子集J，要求J中所有作业都能在各自期限内完成并且使得不在J中的作业招致的罚款总额最小——最优解。

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;　　$(p_2,d_2,t_2) = (10,3,2)$;

$(p_3,d_3,t_3) = (6,2,1)$;　　$(p_4,d_4,t_4) = (\ 3,1,1)$;

状态空间：问题的解空间由作业集(1,2,3,4)的所有可能的**子集**组成，共有2^4个元组。

状态空间树：两种表示形式，

1）元组大小可变的表示

表示选了哪些作业，用作业编号表示。

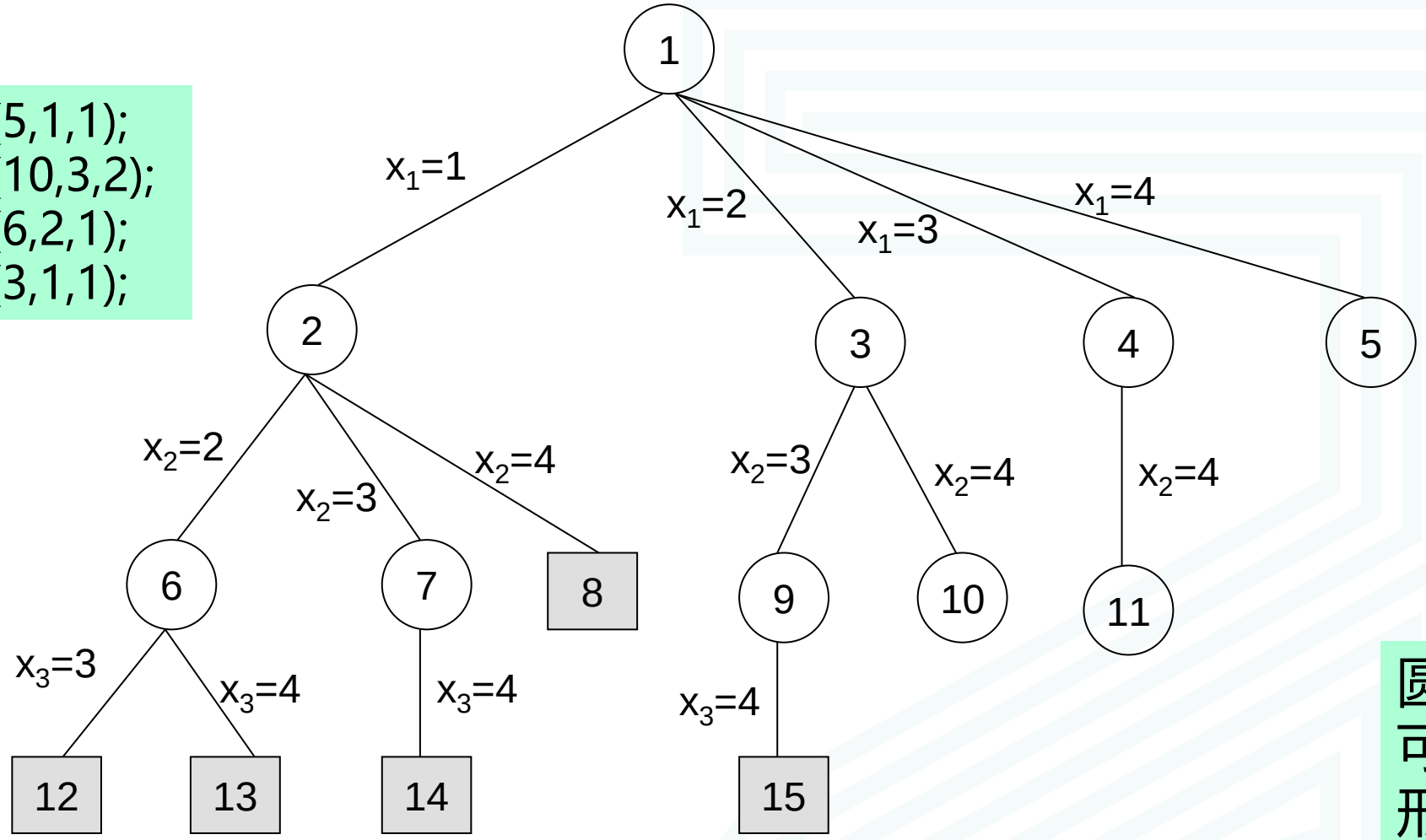
2）元组大小固定的表示

表示是否选中作业，每个作业对应一个0/1值

作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;



图(a) 采用大小可变的元组表示的状态空间树

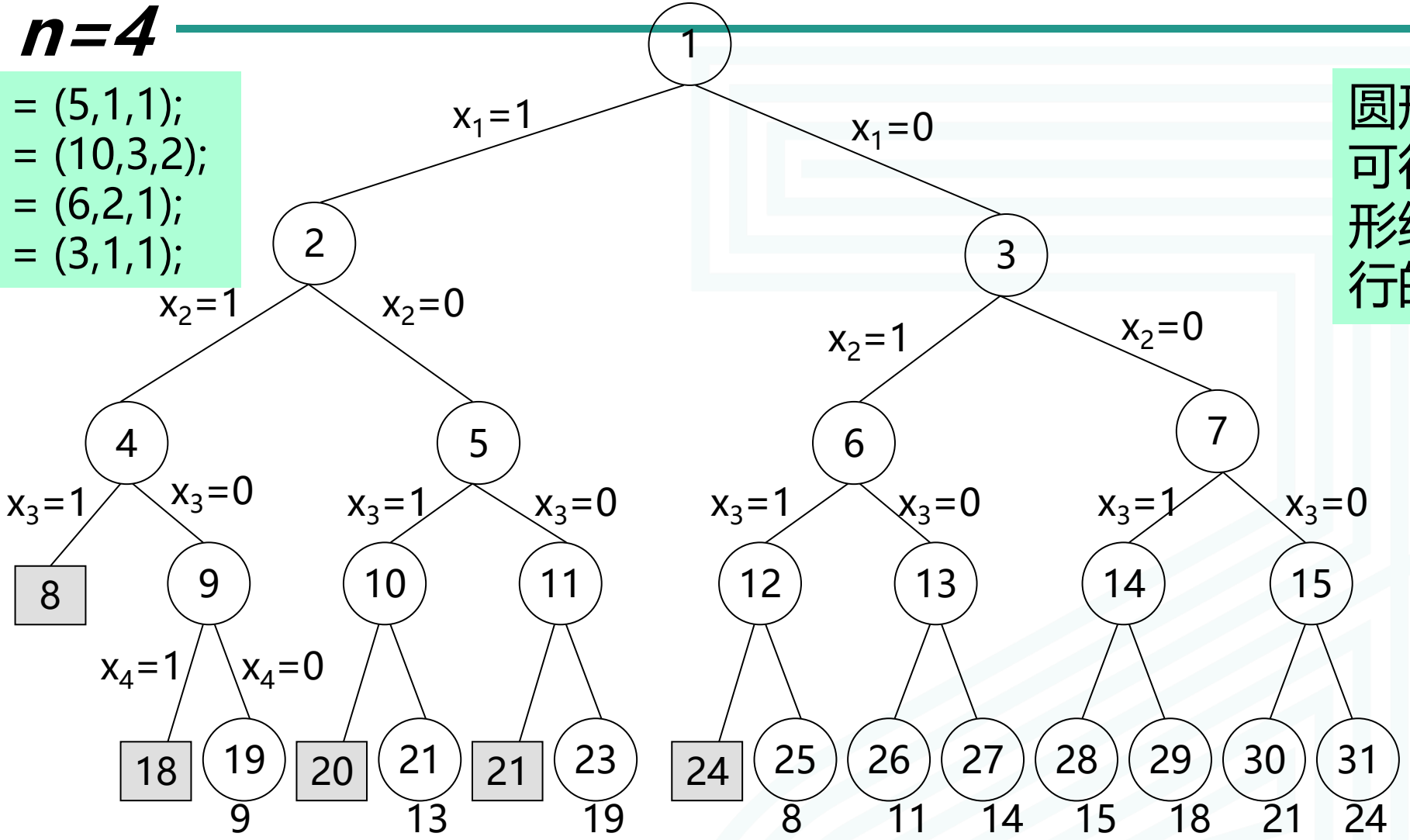
圆形结点都是可行结点，方形结点是不可行的子集合

作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;

圆形结点都是可行结点，方形结点是不可行的子集合



图(b) 采用大小固定的元组表示的状态空间树

实例：n=4

成本函数c(•)定义为：

- 对于圆形结点X，c(X)是根为X的子树中结点的最小罚款；
- 对于方形结点， c(X)=∞。

例：下图(c)， c(3)=8，c(2)=9， c(1)=8

下图(d)，c(2)=9，c(5)=13，c(6)=8， c(1)=8

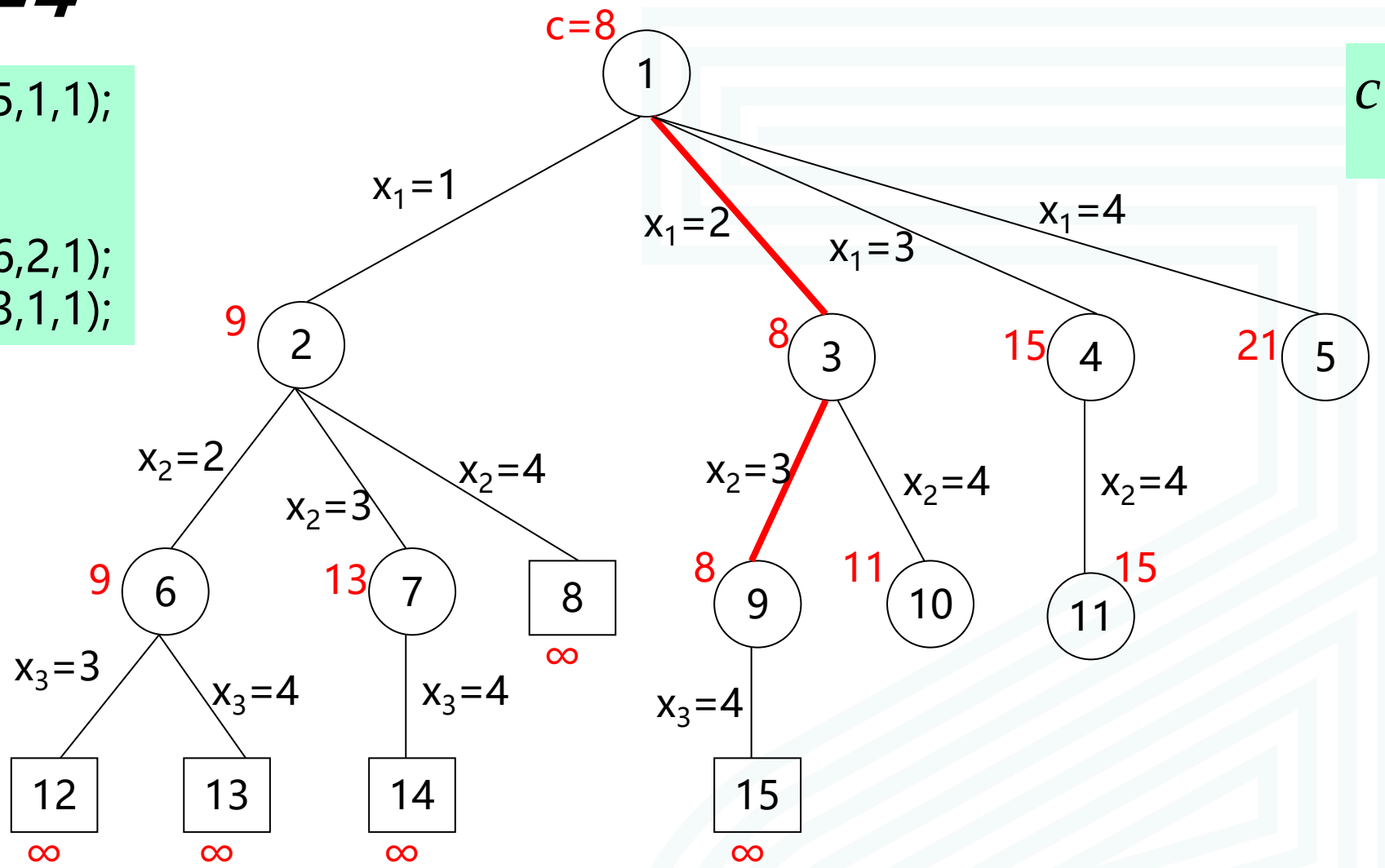
c(1)=最优解J对应的罚款

作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;

$$c(X) = \sum_{i \notin J} p_i$$

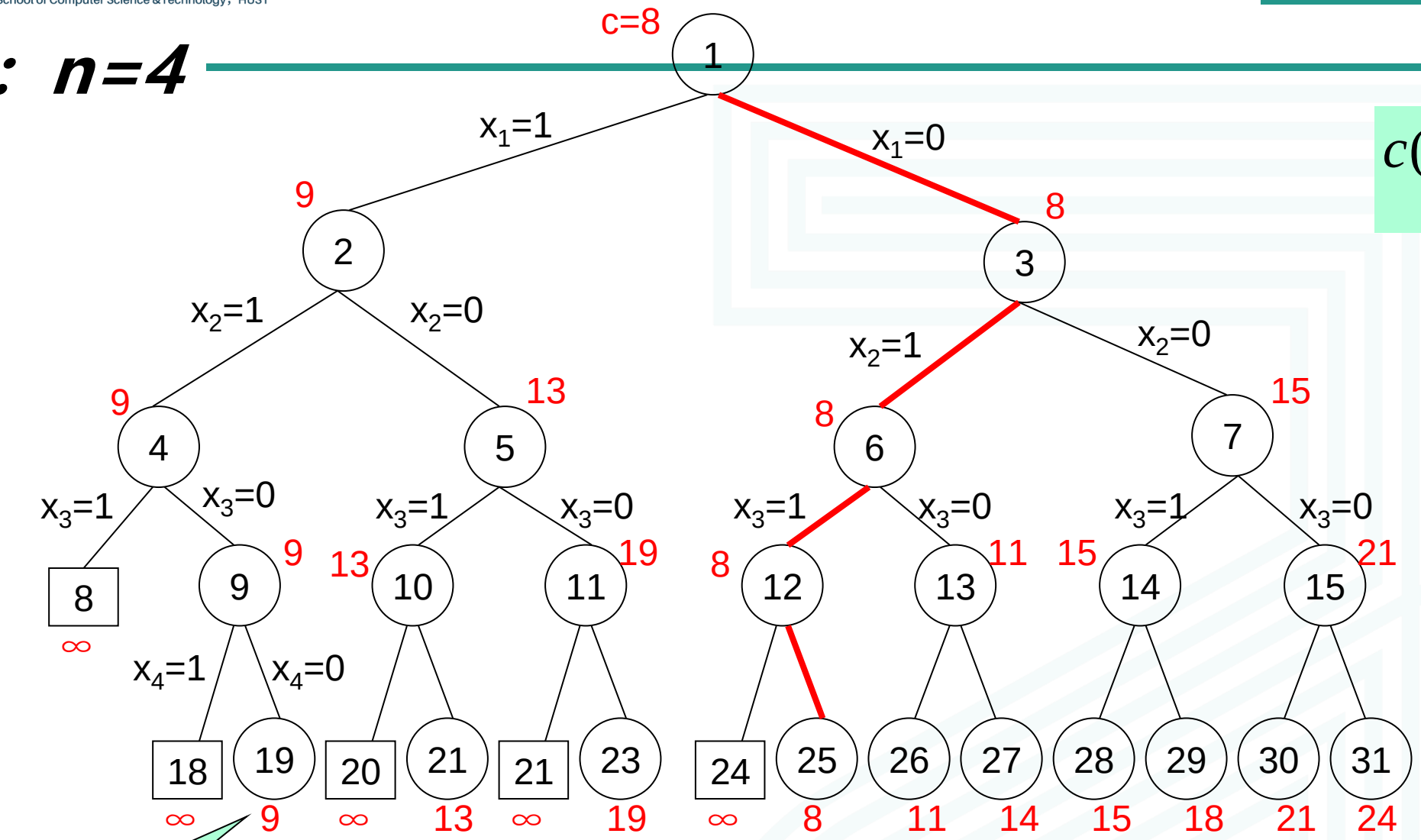


图(c)　采用大小可变的元组表示的状态空间树各结点的c值

作业排序问题

实例：n=4

$$c(X)=\sum_{i\notin J} p_i$$



图(d) 采用大小固定的元组表示的状态空间树各结点的c值

实例：n=4

成本估计函数 $\hat{c}(\bullet)$ 的定义:

设S_x是考察结点X时，已计入J中的作业的集合。

令 m = max{i|i∈S_x}，$\hat{c}(X)=\sum_{\substack{i<m\\ i\notin S_X}} p_i$

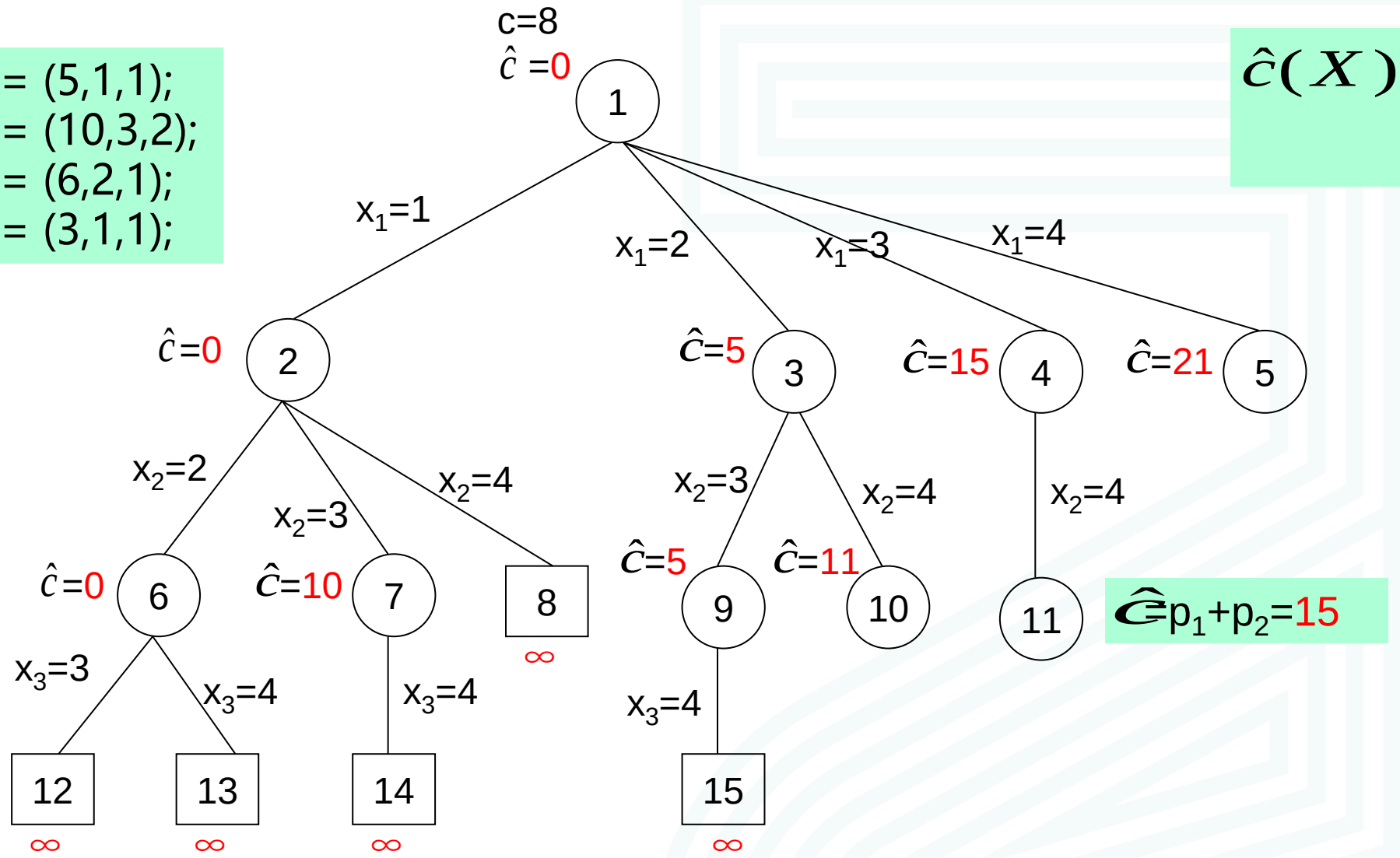
即$\hat{c}(X)$代表已经被考虑过但没有被计入J中的作业的罚款合计。这是已确定的罚款数,因此$\hat{c}(X)\le c(X)$，是$\hat{c}(X)$可以作为c(X)的估计值(下界)。

作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;

$$\hat{c}(X) = \sum_{\substack{i<m \\ i \notin S_X}} p_i$$

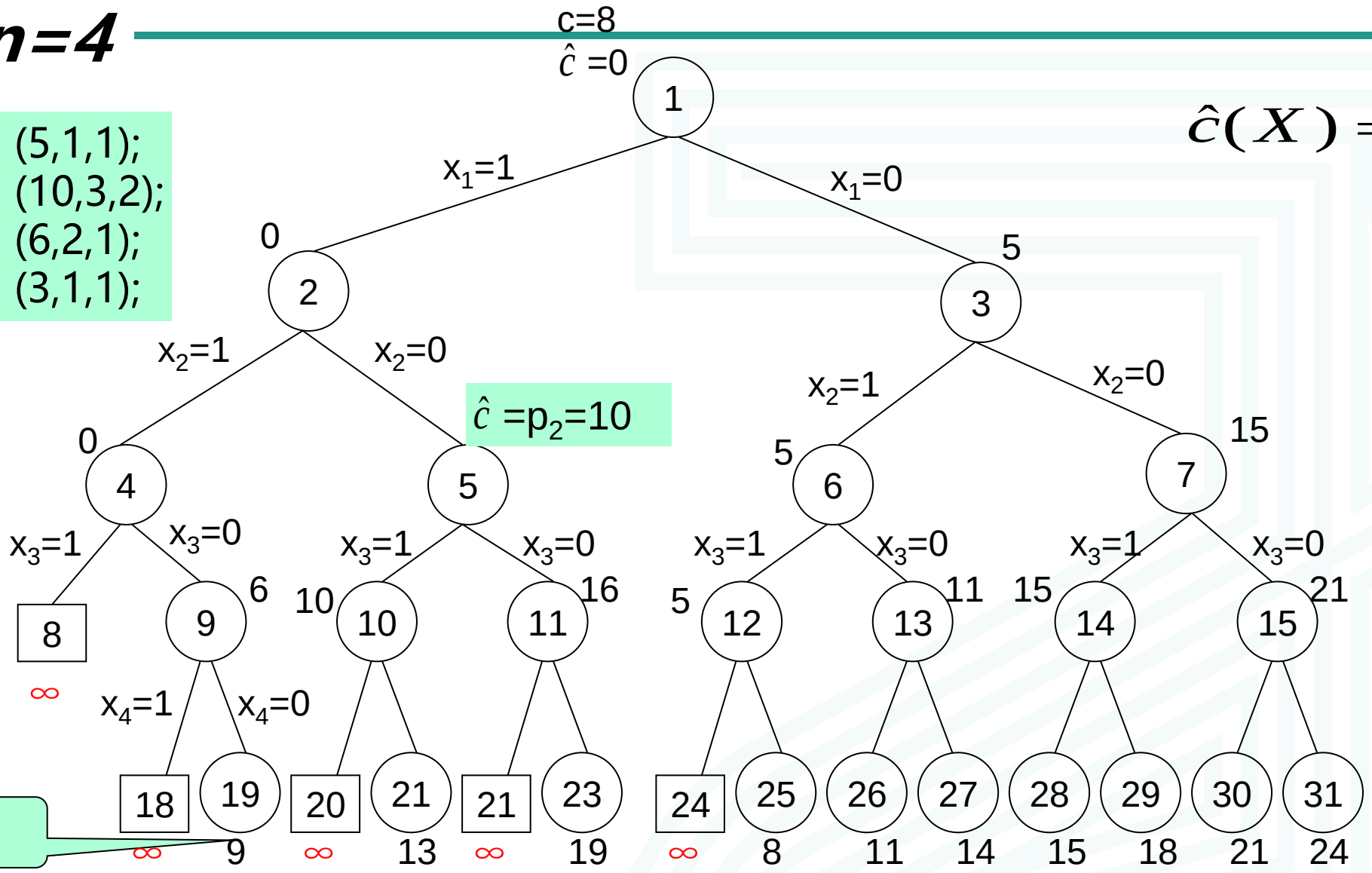


作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;

$$\hat{c}(X) = \sum_{\substack{i<m \\ i \notin S_X}} p_i$$



实例：n=4

成本估计函数上界的定义

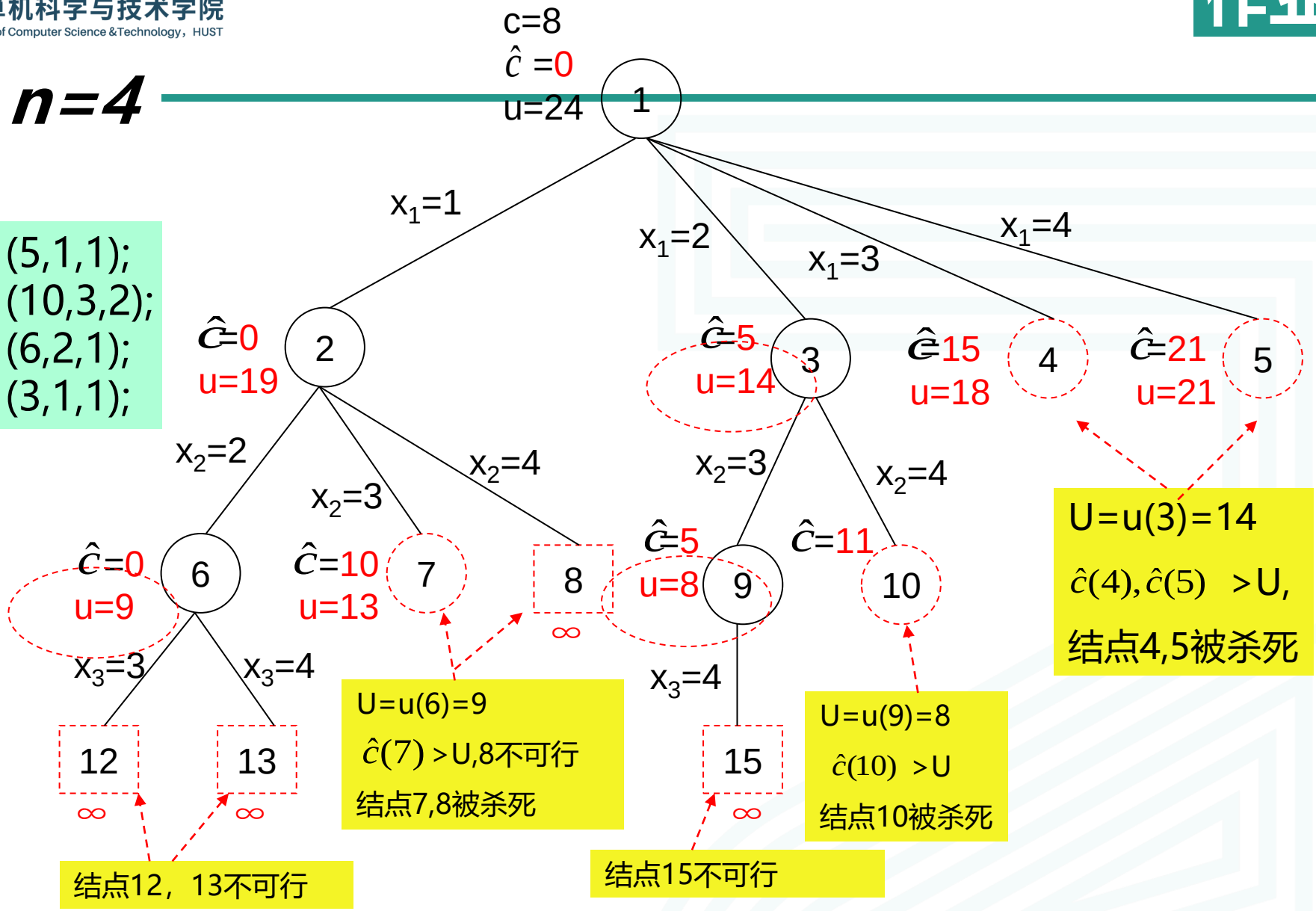
$$U(X)=\sum_{i\notin S_X} p_i$$

U(X)是c(X)的一个“简单”上界，代表目前没有被计入到J中的作业的罚款合计——可能的最多罚款，因此是目前罚款的上线。

作业排序问题

实例：n=4

$(p_1,d_1,t_1) = (5,1,1)$;
$(p_2,d_2,t_2) = (10,3,2)$;
$(p_3,d_3,t_3) = (6,2,1)$;
$(p_4,d_4,t_4) = (3,1,1)$;



$$\hat{c}(X) = \sum_{\substack{i<m \\ i \notin S_X}} p_i$$

$$u(1) = \sum_{1 \le i \le n} p_i$$

$$u(X) = \sum_{i \notin S_X} p_i$$

$$U = \min(u(X))$$

图9.7　采用大小可变的元组表示的状态空间树的成本估计值

作业：

華中科技大學
计算机科学与技术学院
School of Computer Science &Technology，HUST

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