算法设计与分析

Computer Algorithm Design & Analysis

2024.12

王多强

QQ：1097412466

Chapter 22

Elementary Graph Algorithms

基本的图算法

22.4 基本的分支 - 限界法

分支 - 限界法：是在生成**当前 *E*-结点全部儿子**之后再生成其它活结点的儿子（**宽度优先策略**）且用**限界函数**来避免生成不包含答案结点子树的状态空间检索方法。

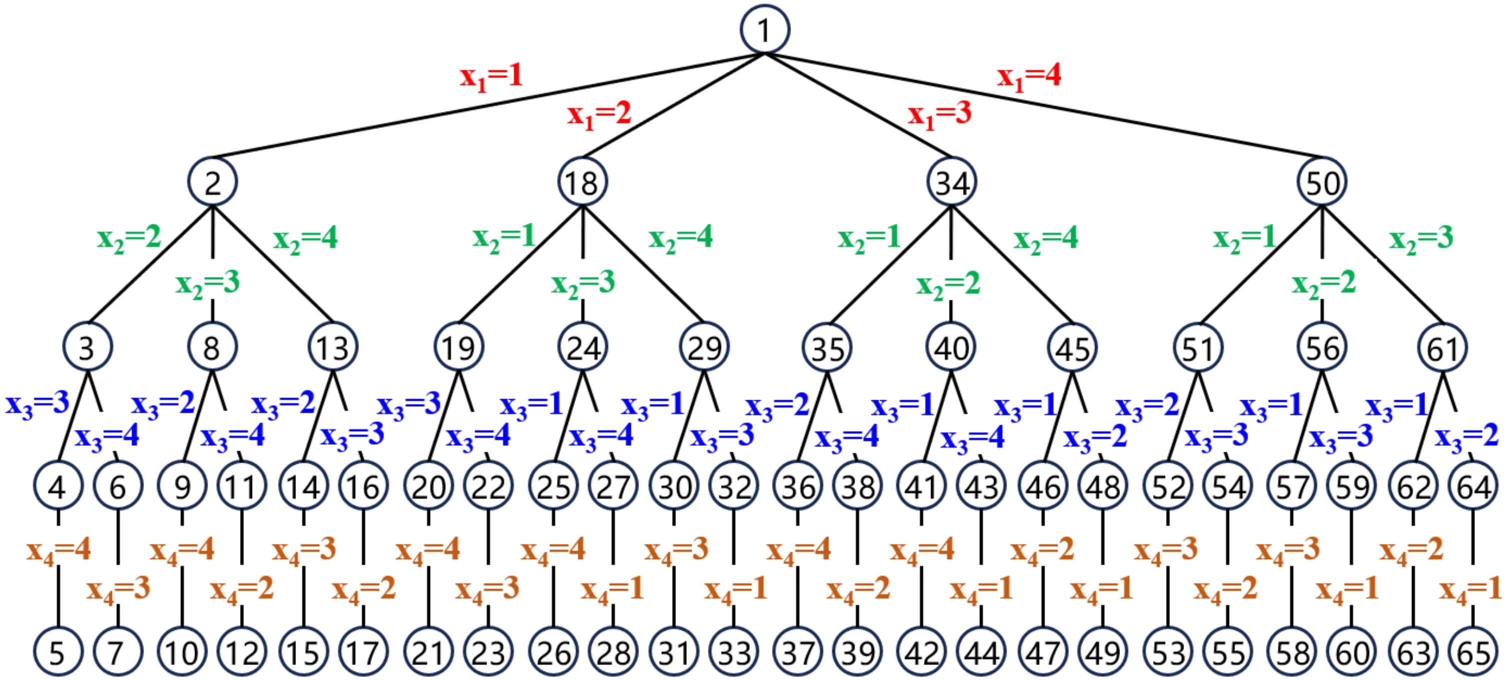
- **活结点表**

  活结点：自己已经被生成，但还没有**被检测**（**即还没有生成其全部儿子**）、有待进一步生成儿子结点的结点。

  活结点表的存储结构：**队列**（**BFS**）、**栈**（**D-Search**）

- **分支-限界法的两种基本设计策略：**
  - **FIFO 检索**：活结点表采用**队列**
  - **LIFO 检索**：活结点表采用**栈**

例 4-皇后问题的**分支-限界**搜索过程

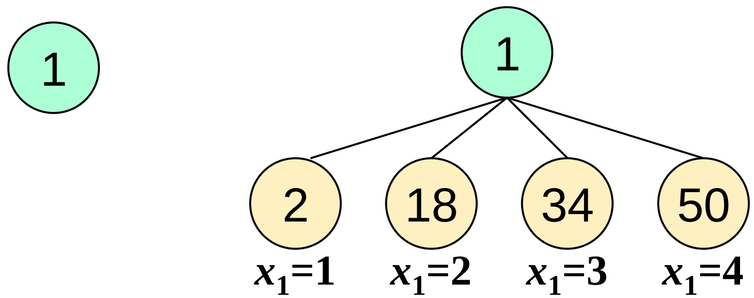


4-皇后问题完整的状态空间树

限界函数：如果 $(x_1, x_2, \ldots, x_{i-1})$ 是**根结点到当前** ***E*-结点** x_{i-1} **的路径**，那么具有**父-子标记**的所有儿子结点 x_i 是这样的一些结点，它们使得 $(x_1, x_2, \ldots, x_{i-1}, x_i)$ 表示没有两个皇后处于相互攻击状态的一种棋局。

采用**FIFO分支-限界法**检索4-皇后问题的状态空间树：

***E* 结点**　　**扩展 *E* 结点得到的状态空间树**　　**活结点表（队列）**

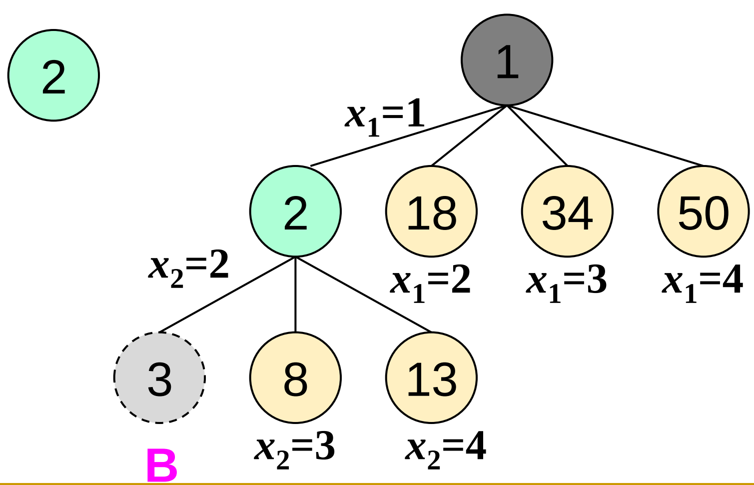


head ← **tail**

2	18	34	50				

扩展结点1，得新结点2, 18, 34, 50

活结点2、18、34、50 入队列



head ← **tail**

18	34	50	8	13			

扩展结点2，得新结点3, 8, 13

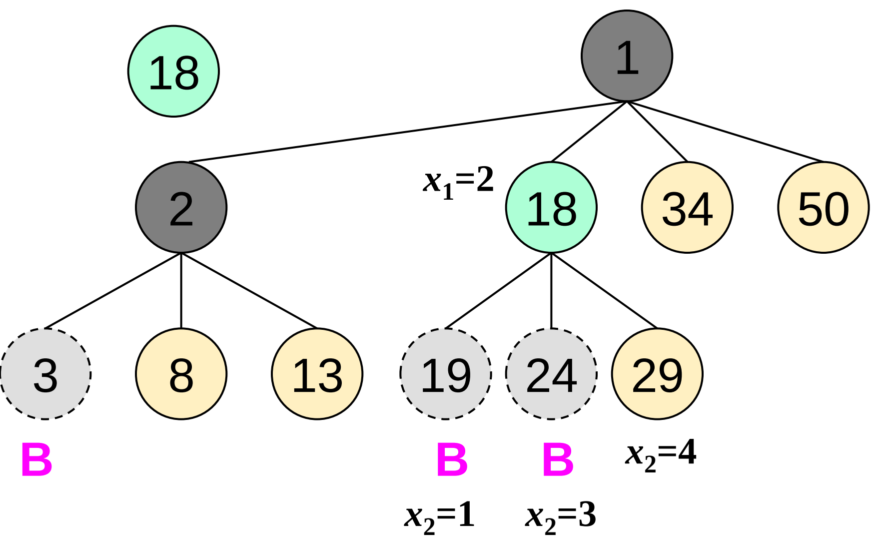
利用限界函数杀死结点3

活结点8、13 入队列

采用FIFO分支-限界法检索4-皇后问题的状态空间树（续）：

E 结点

扩展 *E* 结点得到的状态空间树



活结点表（队列）

head ← tail

34	50	8	13	29			

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

采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

***E* 结点**

34

扩展 *E* 结点得到的状态空间树

活结点表（队列）

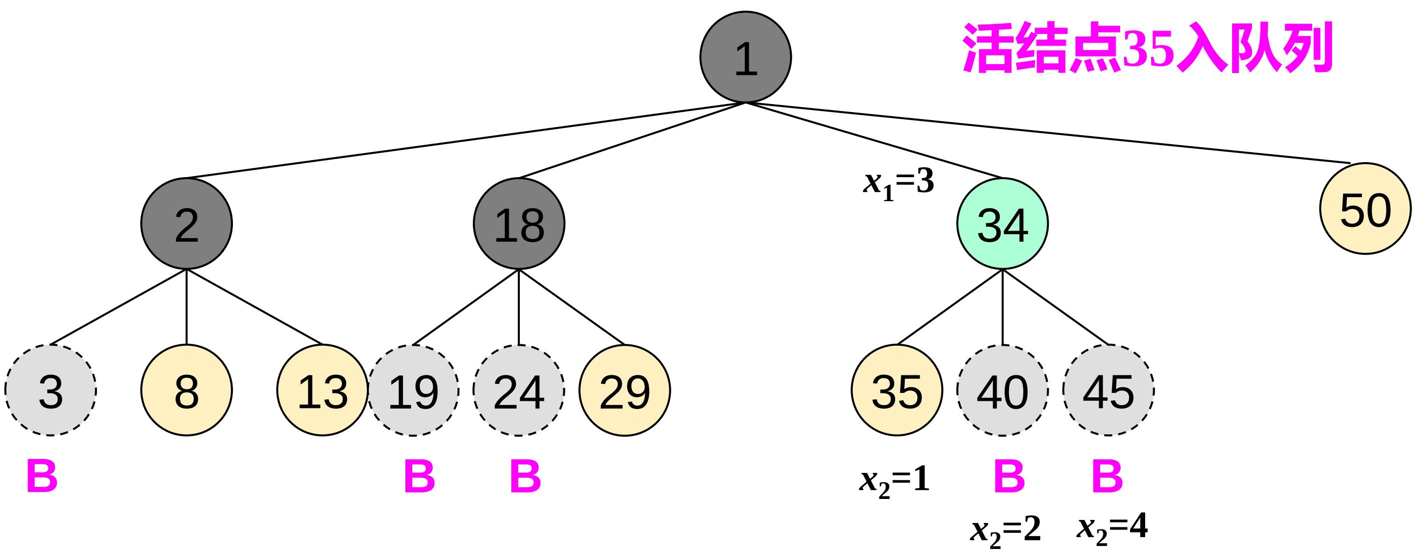
head ← tail

50	8	13	29	35			

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



采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

E **结点**　　**扩展 *E* 结点得到的状态空间树**　　**活结点表（队列）**

50

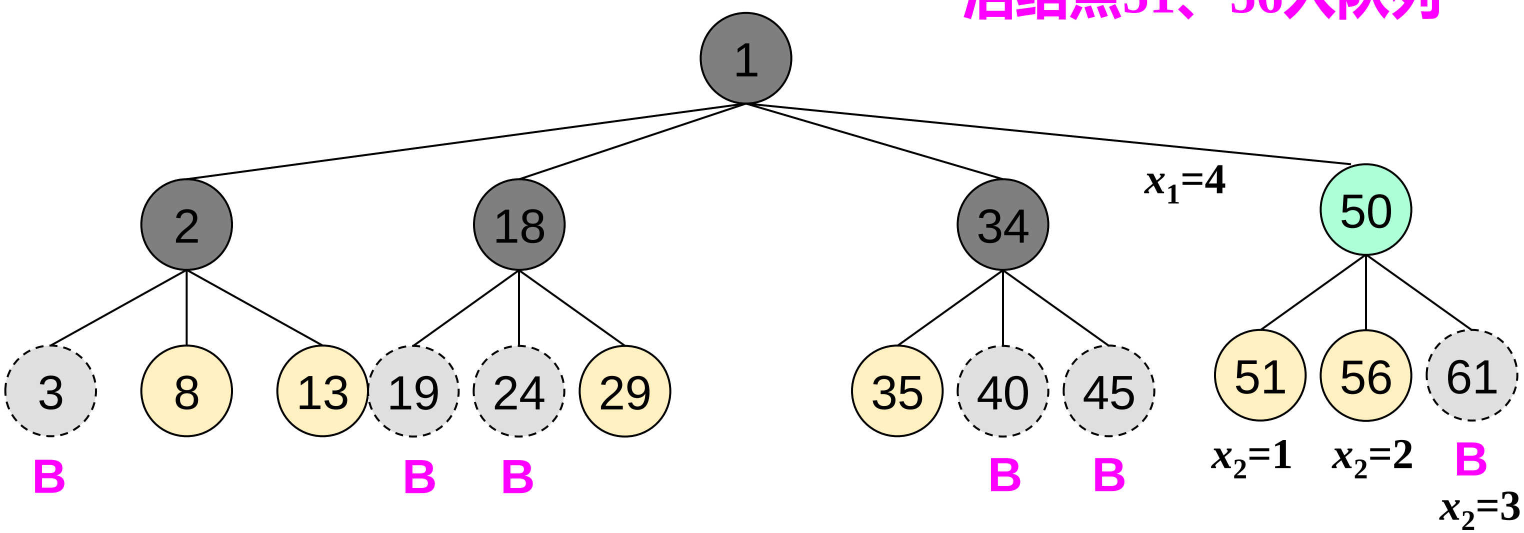
head ← tail

8	13	29	35	51	56		

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



采用FIFO分支-限界法检索4-皇后问题的状态空间树（续）：

E 结点

扩展 *E* 结点得到的状态空间树

活结点表（队列）

8

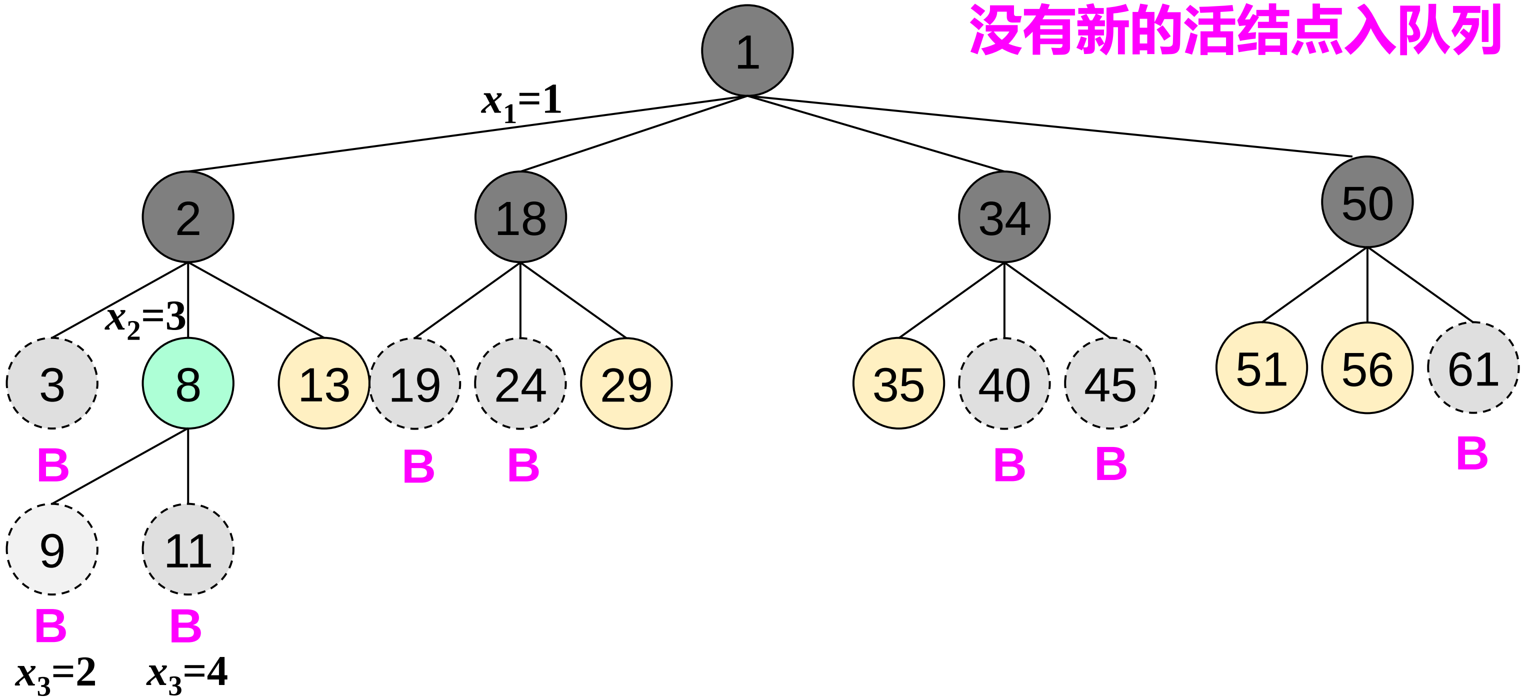
head ← tail

13	29	35	51	56			

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

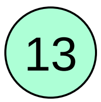


采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

E **结点**

扩展 *E* 结点得到的状态空间树

活结点表（队列）



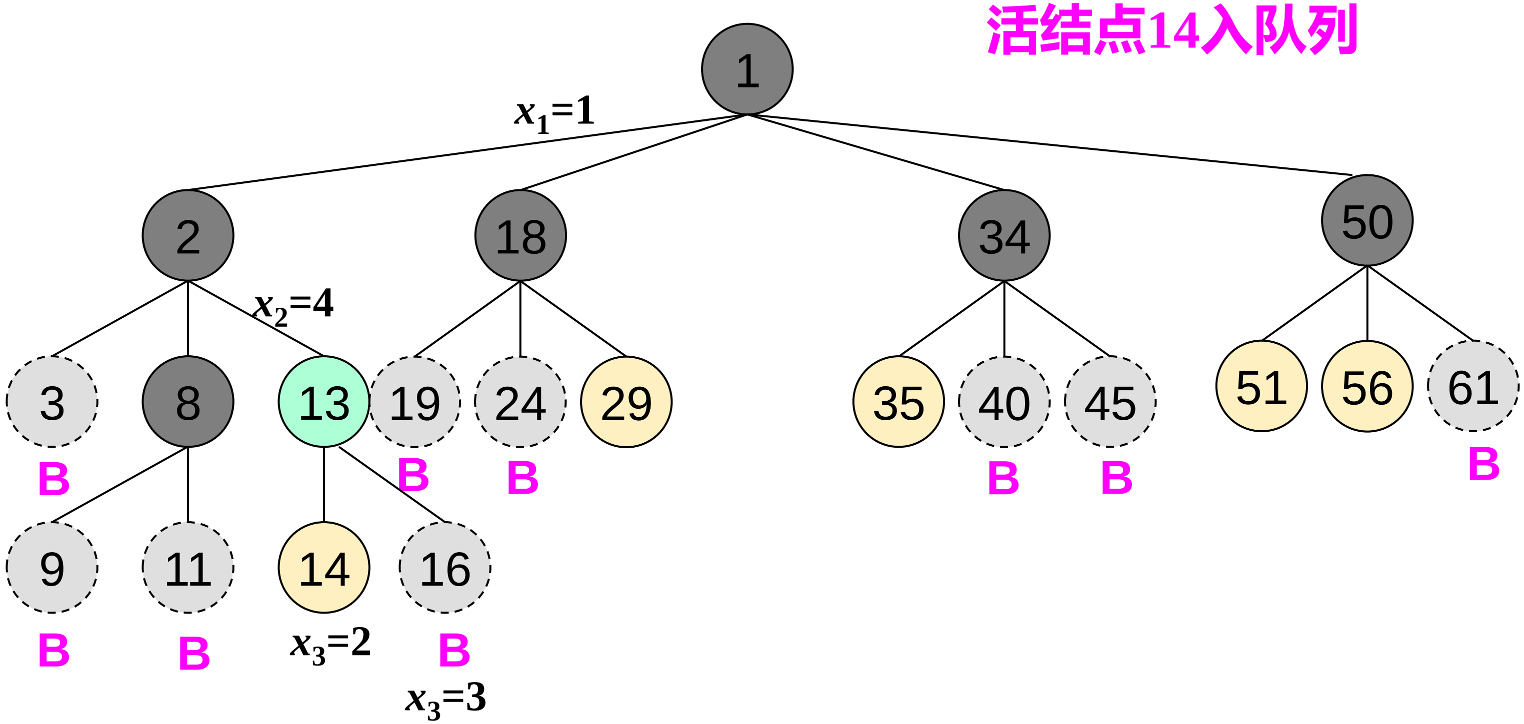
head ← tail

29	**35**	**51**	**56**	**14**			

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



采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

E **结点**

扩展 *E* 结点得到的状态空间树

活结点表（队列）

29

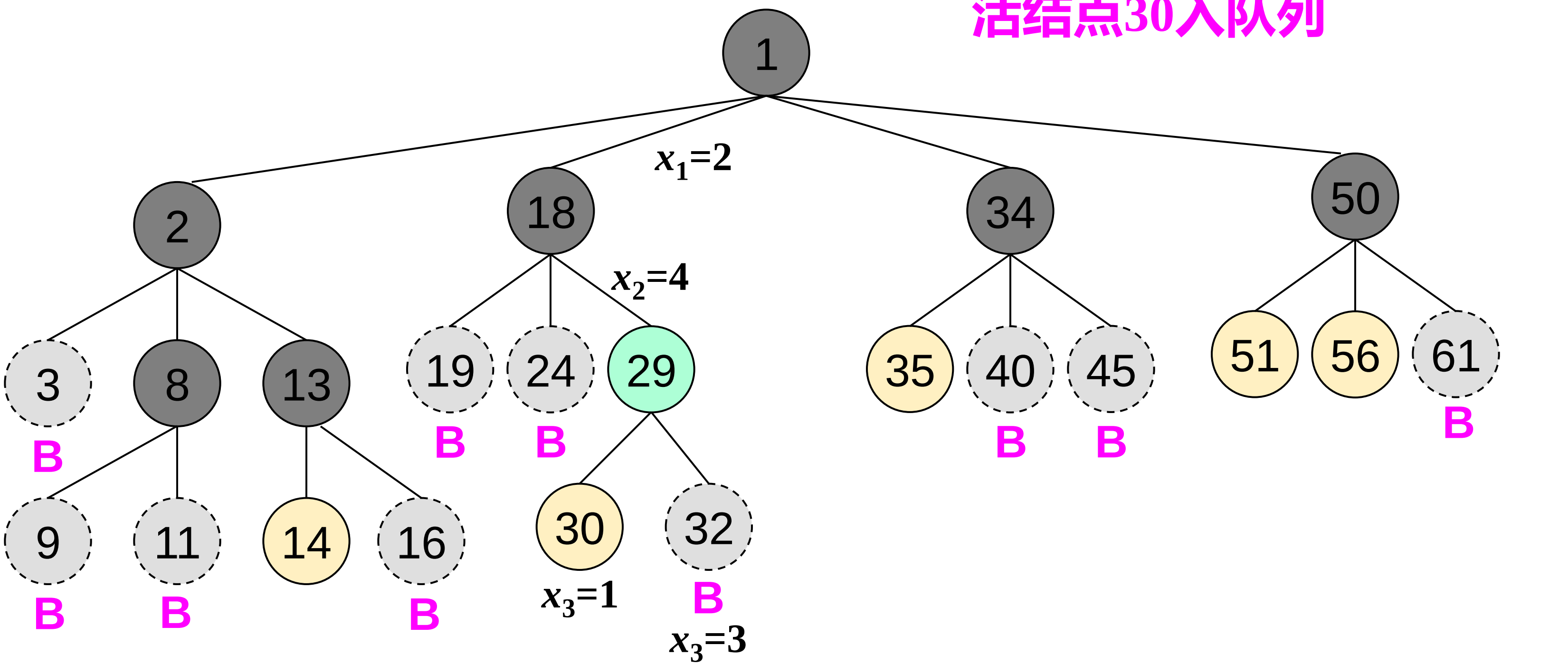
head ← tail

35	51	56	14	30			

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



采用FIFO分支-限界法检索4-皇后问题的状态空间树（续）：

E 结点　　扩展 *E* 结点得到的状态空间树　　活结点表（队列）

35

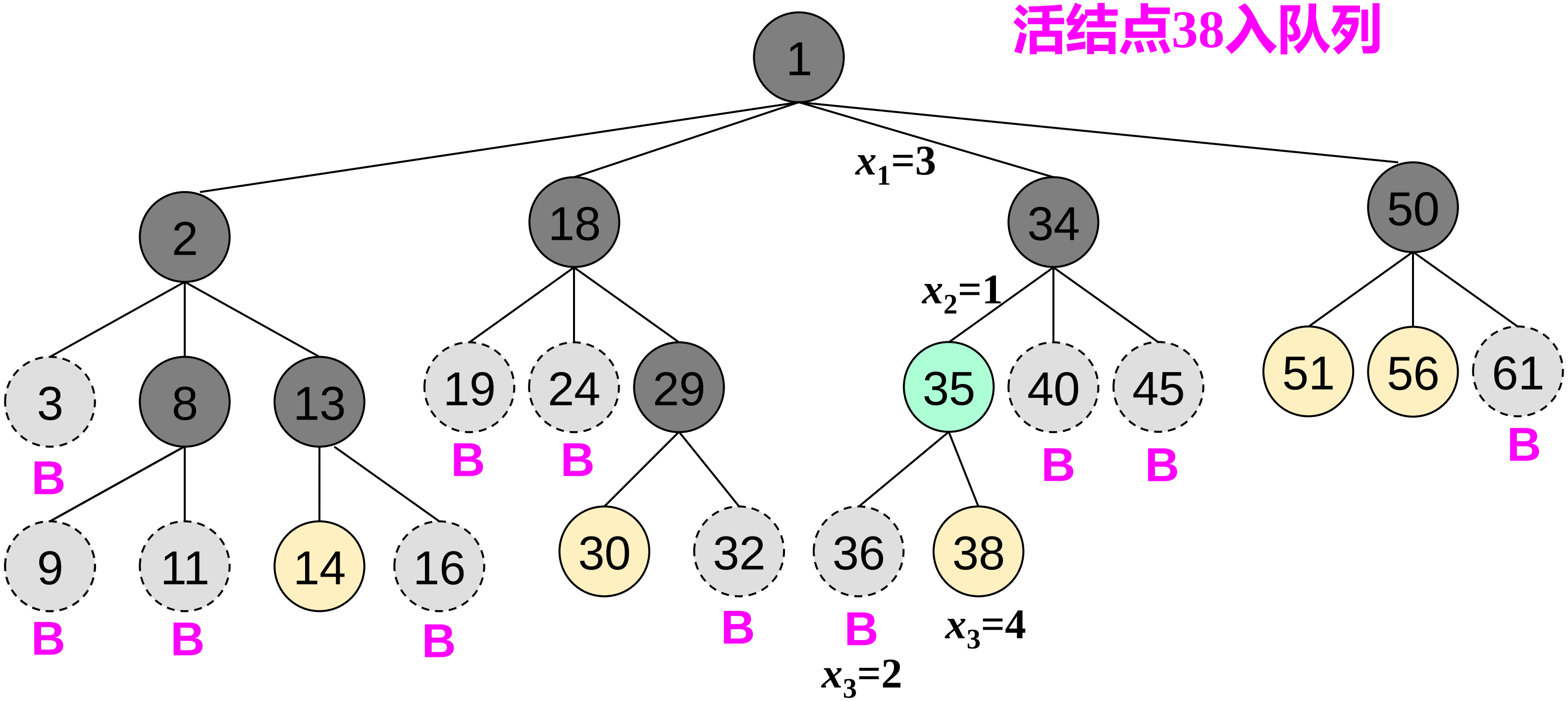
head ← tail

51	56	14	30	38			

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



采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

***E* 结点**　　　　**扩展 *E* 结点得到的状态空间树**　　　　**活结点表（队列）**

51

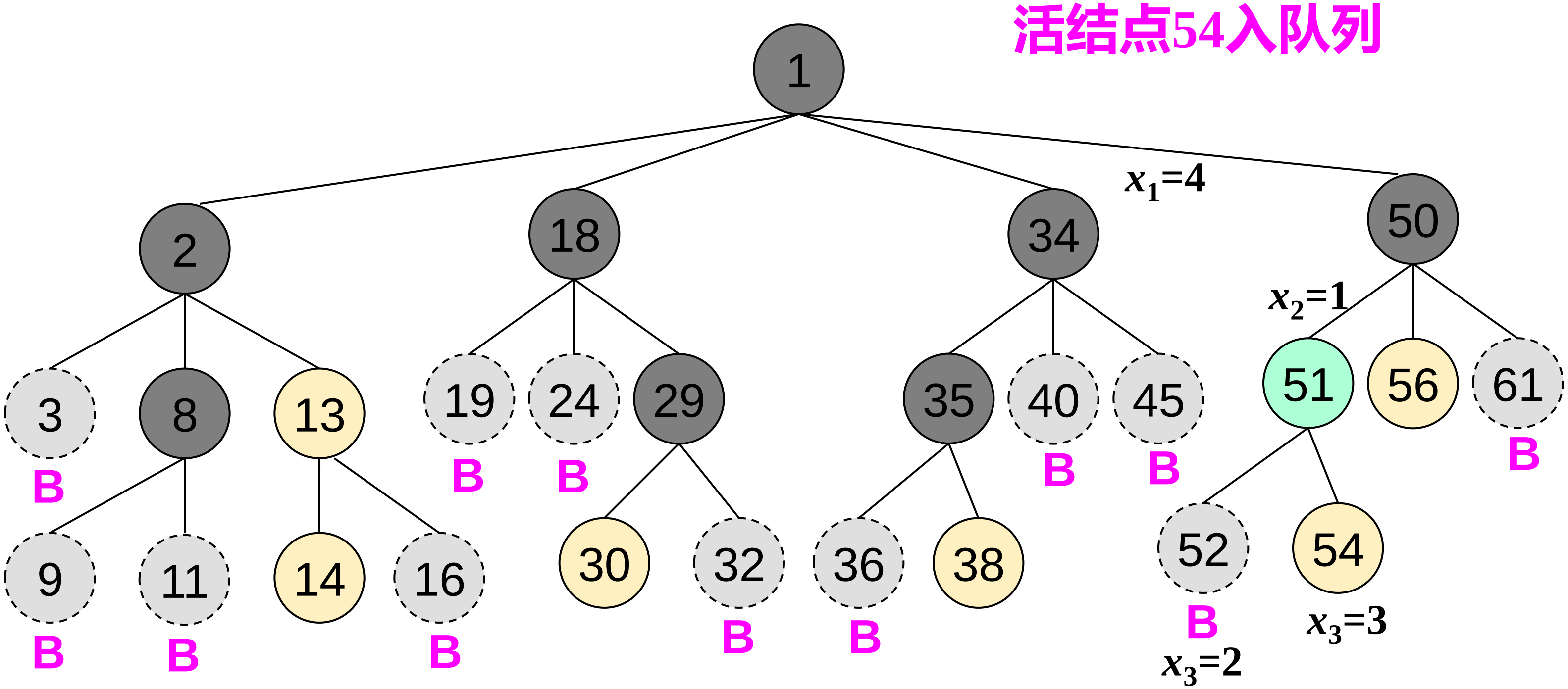
head ← tail

56	14	30	38	54			

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



采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

***E* 结点**　　　　**扩展 *E* 结点得到的状态空间树**　　　　**活结点表（队列）**

56

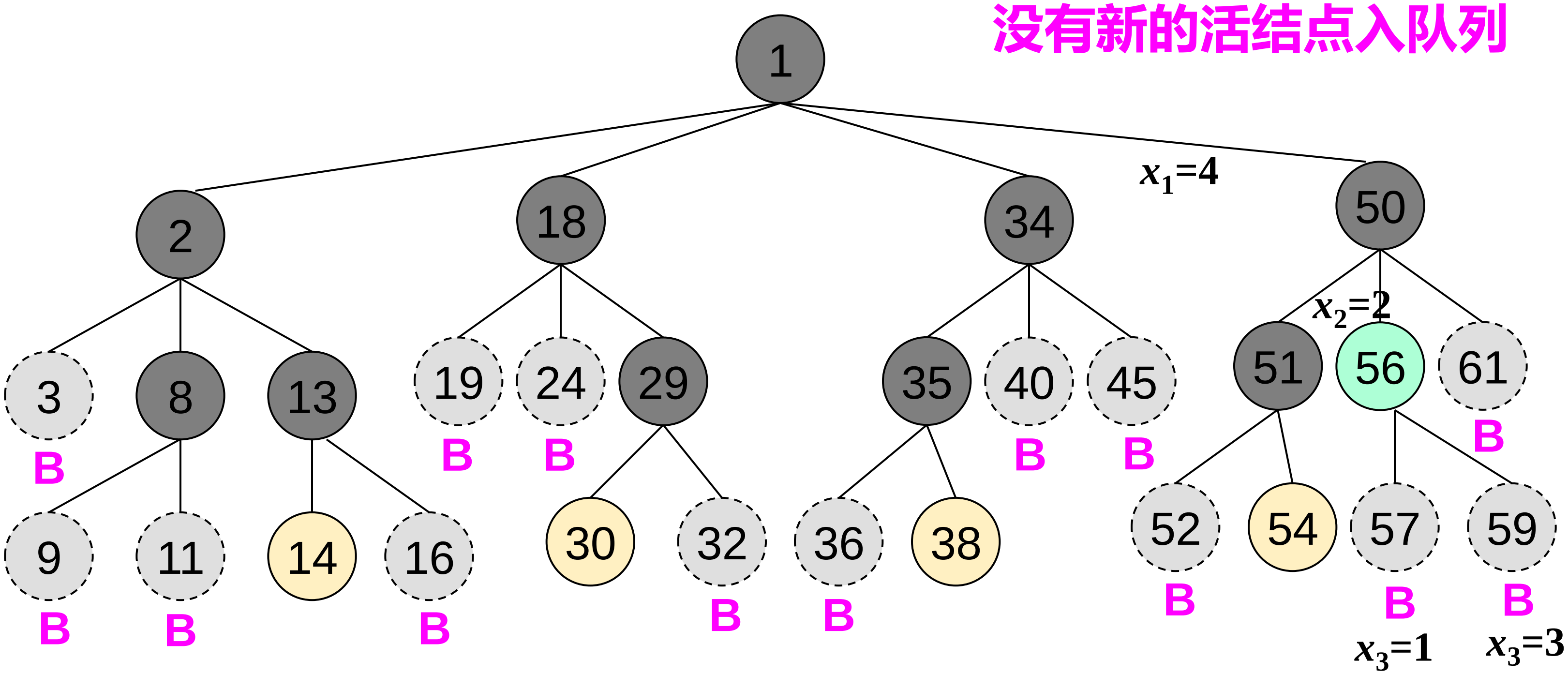
head ← **tail**

14	30	38	54				

扩展结点56，得新结点**57, 59**

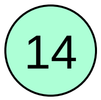
利用限界函数杀死结点57、59

没有新的活结点入队列



采用**FIFO分支-限界法**检索4-皇后问题的状态空间树（续）：

E 结点



扩展 E 结点得到的状态空间树

活结点表（队列）

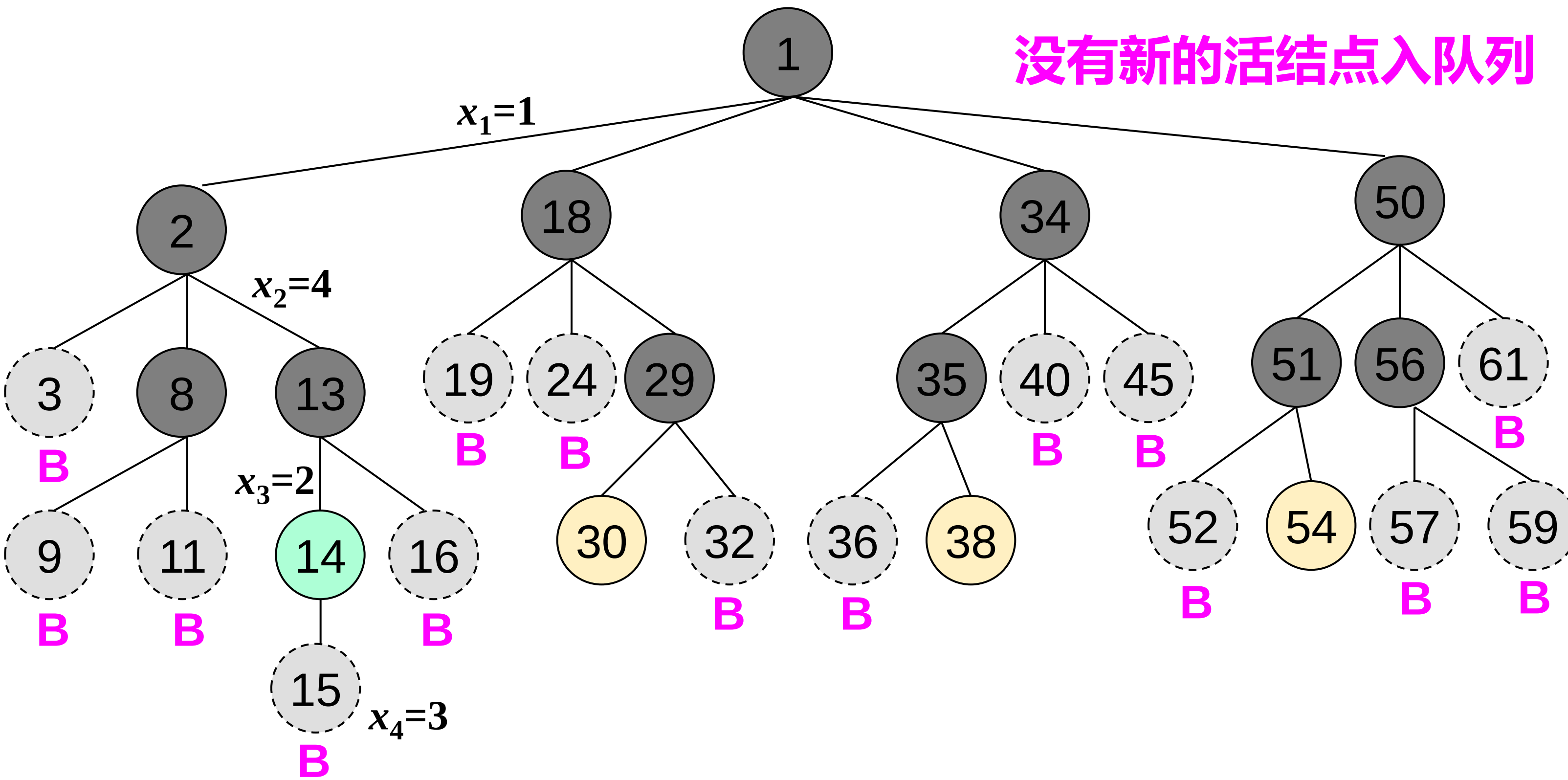
head ← tail

30	38	54					

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



采用FIFO分支-限界法检索4-皇后问题的状态空间树（续）：

E 结点　　扩展 *E* 结点得到的状态空间树　　活结点表（队列）

30

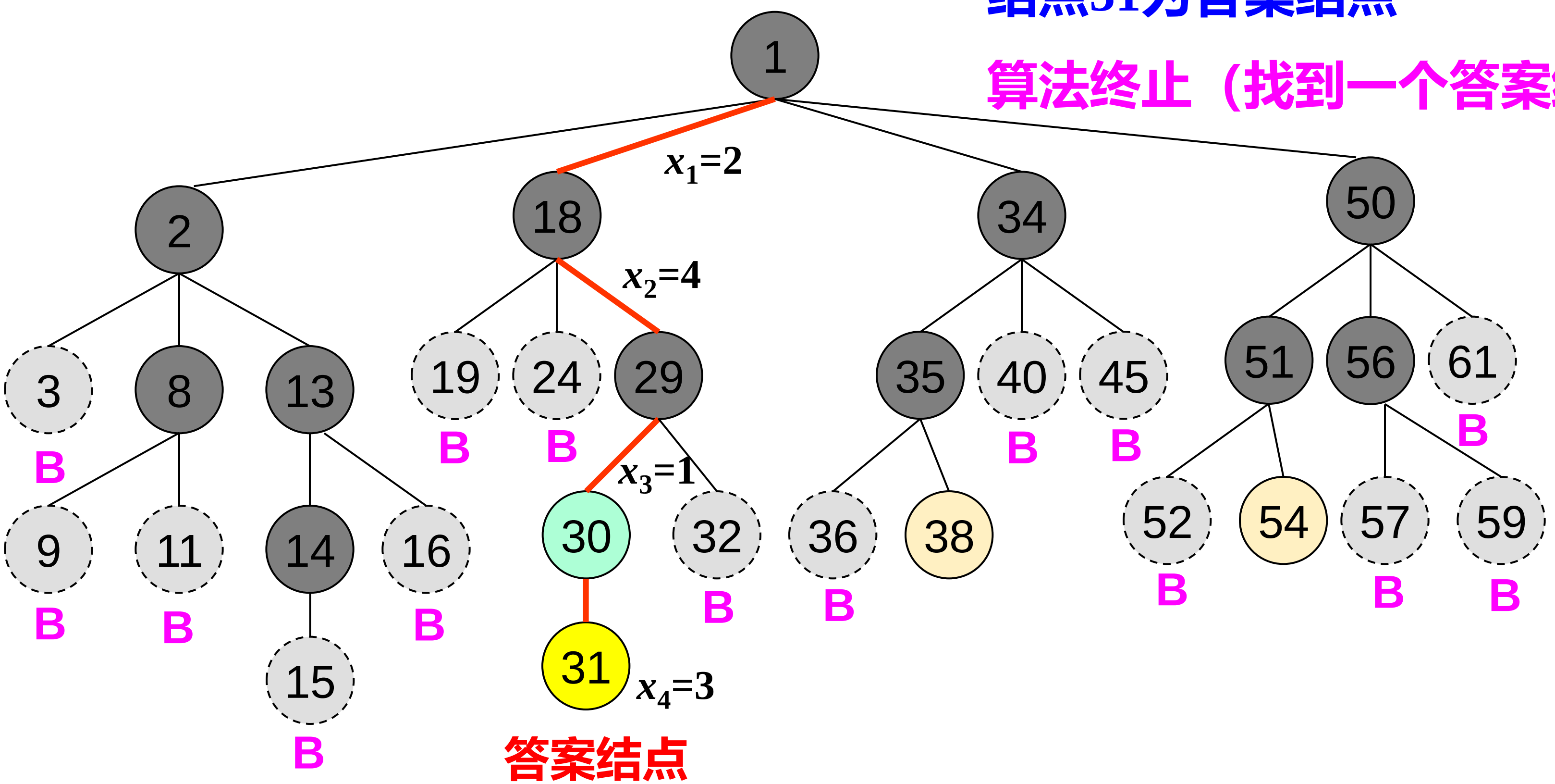
head ← tail

38	54						

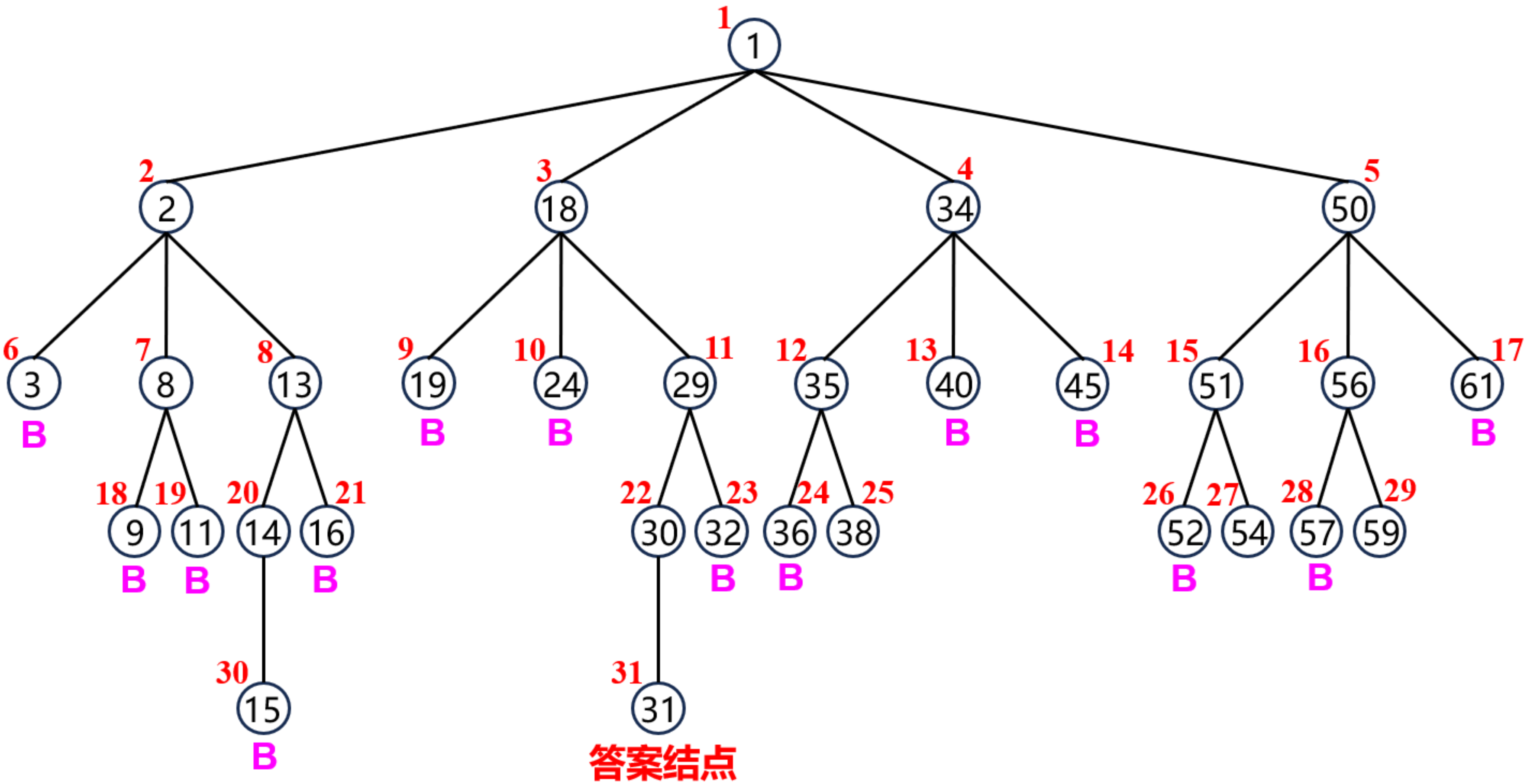
扩展结点30，得新结点31

结点31为答案结点

算法终止（找到一个答案结点）



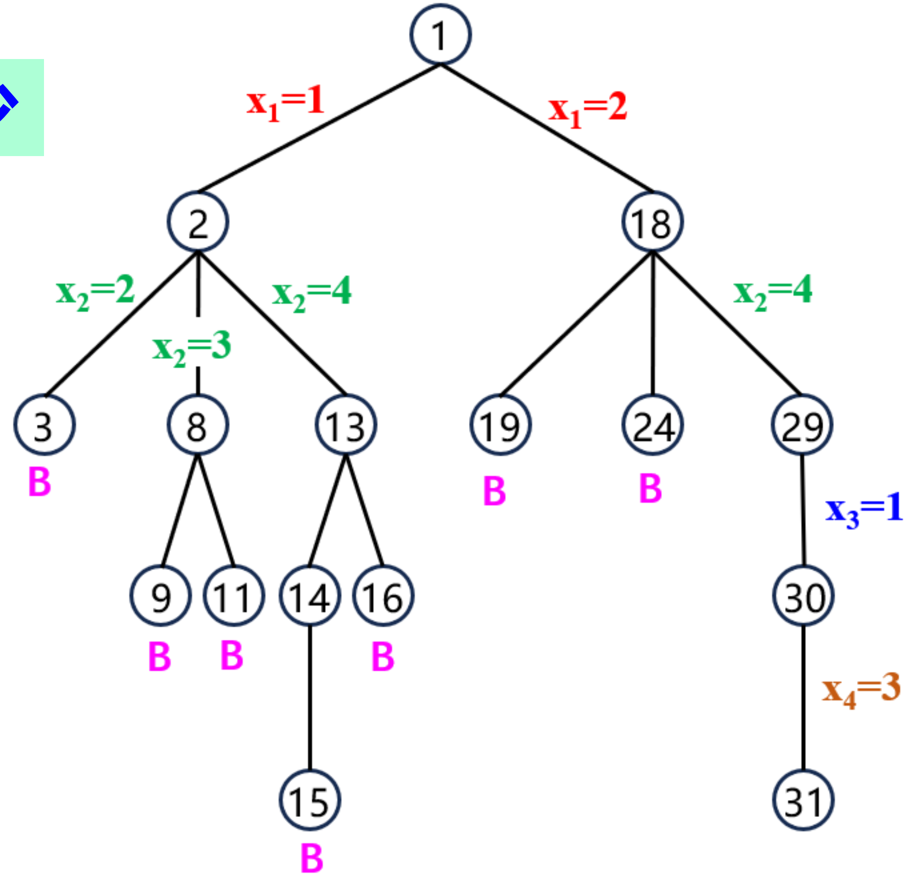
采用FIFO分支-限界法检索4-皇后问题得到的状态空间树



4-皇后问题： 回溯 Vs. FIFO分支-限界

扩展出来的结点较少

回溯
Win !



回溯法求解4-皇后问题生成的状态空间树

22.5 LC-检索（Least Cost，A^*算法，启发式搜索算法）

- 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**。

- **如何解决?**
  - **做某种排序，让可能导致答案结点的活结点排在前面！**
  - **怎么排序?**
    - 寻找一种"**有智力**"的**排序函数 $C(\cdot)$**，**用 $C(\cdot)$ 来选取下一个 *E* 结点**，加快到达一答案结点的检索速度。

      如尽快找到结点：**29→30→31**

能否赋予一个比其它活结点高的优先级而使得30结点尽快成为 *E*-结点?

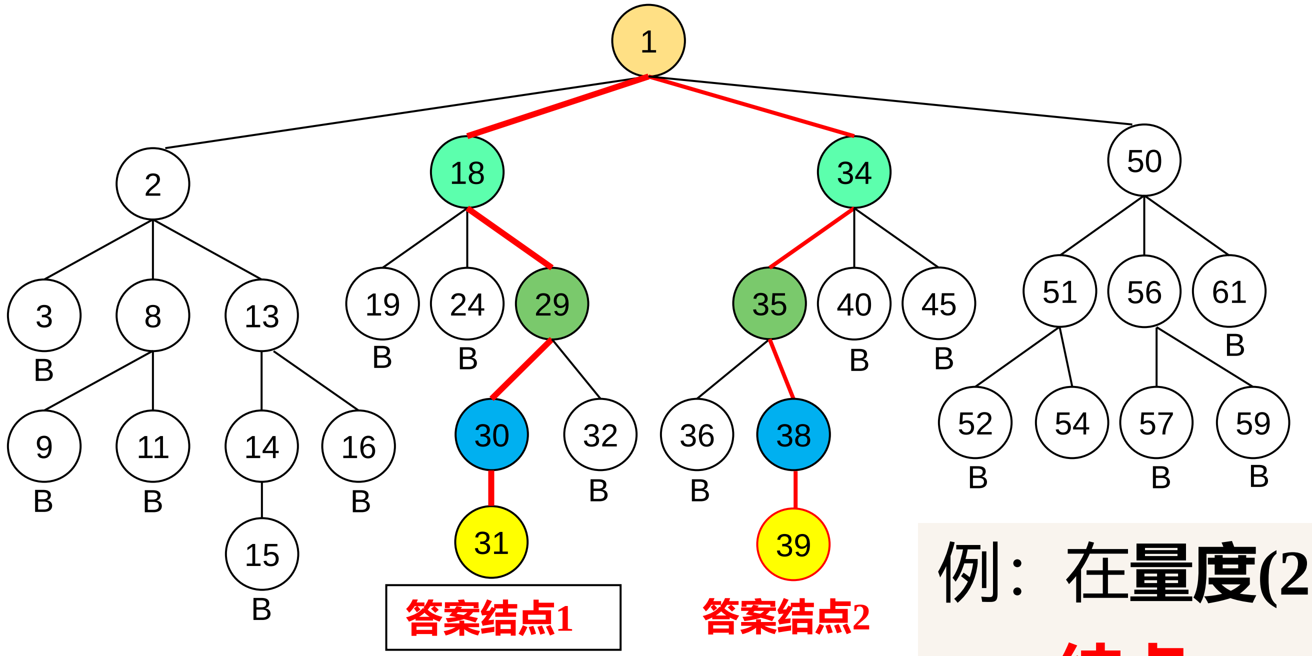
- **如何衡量结点的优先等级?**
  - 对于任一结点，用该结点**导致答案结点的成本**（**代价**）来衡量该结点的优先级：**成本越小越优先。**
- **如何衡量代价?**
  - 对任一结点 X，可以用两种标准来衡量**结点的代价**：

    (1) 在生成一个答案结点之前，**子树 X 需要生成的结点数**：

    需要生成的结点数越多代价越高

    (2) 在子树 X中离 X 最近的那个**答案结点到 X 的路径长度**：

    路径越短代价越小



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。

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
31、39	0
其余结点	$\geq 3, 2, 1$

- 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。

- **两种标准的特点：**

度量(1)：偏向于选择**生成儿子结点数目少**的结点作为 E 结点。

度量(2)：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。

1、结点成本函数 C

$C(\cdot)$：“有智力”的排序函数，能够度量每个结点的成本，并依据成本排序，优先选择**成本最小的活结点**作为下一个待扩展的 E 结点。$C(\cdot)$ 又称为**结点成本函数**。

- **结点成本函数 $C(X)$ 的取值:**

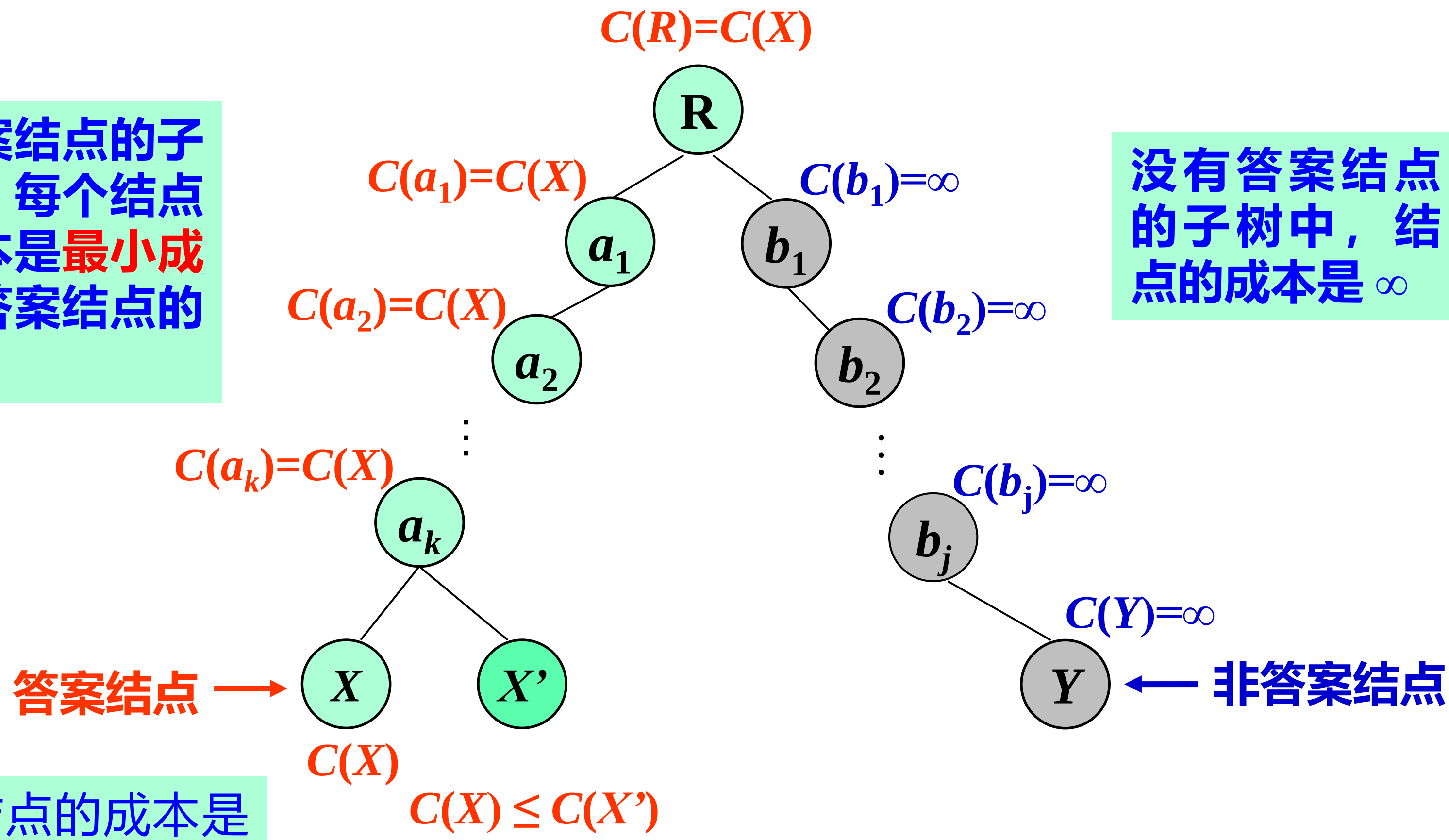
(1) 如果 X **是答案结点**，则 $C(X)$ 等于由状态空间树的**根结点到 X 的成本** (即所用的代价，可以是层级数、计算的时间函数等);

(2) 如果 X **不是答案结点**且**子树 X 中不包含任何答案结点**，则 $C(X) = \infty$。

(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**。

$C(R)$ 等于子树中最小成本的答案结点的成本

有答案结点的子树中，每个结点的成本是最小成本的答案结点的成本。



没有答案结点的子树中，结点的成本是 ∞

答案结点的成本是根结点到 X 的成本

结点成本函数的计算

计算结点成本函数的困难

想一下怎么计算 $C(\cdot)$?

事实上，**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**。

因此，计算结点成本的**精确值是不现实的** —— 相当于求解原始问题。

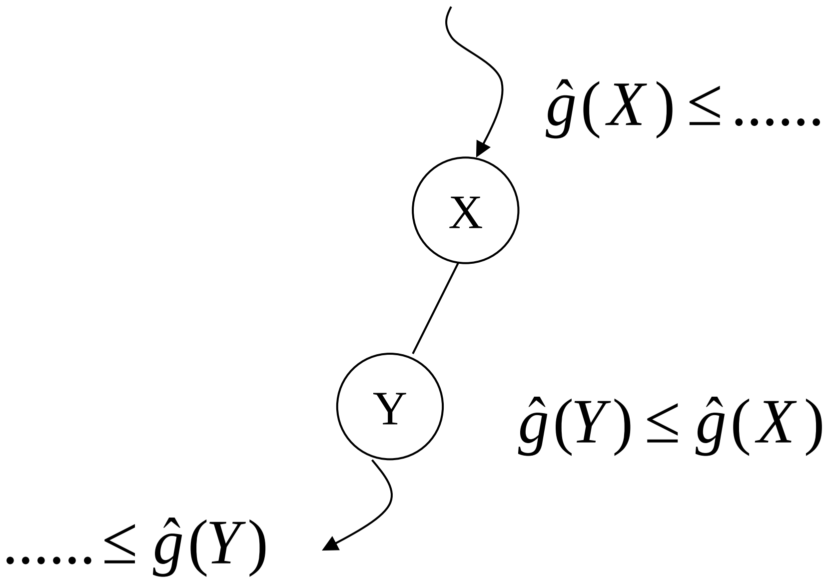
- 引入**结点成本的估计函数：** $\hat{c}(X)$

  $\hat{c}(X)$ 由两部分组成：$h(X)$ 和 $\hat{g}(X)$

其中，

$\hat{g}(X)$：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 • 到答案结点的估计成本

纵深检索：对任何结点，都是先检索完其儿子结点才能生成其它结点（**深度优先**）。

1）如果 $\hat{g}(X) = c(X)$，最理想！

2）否则，如果**估计有误**，则可能导致方向偏离而不能很快地找到更靠近根的答案结点。如：$\hat{g}(W) < \hat{g}(Z)$，但可能 Z 比 W 更接近答案结点。

如何避免单纯考虑 $\hat{g}(X)$ 造成的纵深检查?

（1）引进 $h(X)$ 改进成本估计函数。

$h(X)$：根结点到结点 X 的已发生的成本。

（2）改进的**结点成本估计函数** $\hat{c}(X)$

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

其中，$f(\bullet)$ 是一个关于成本的非降函数。

- 非零的 $f(\bullet)$ 可以减少算法作偏向于纵深检查的可能性，它**强使**算法优先检索**更靠近答案结点**但**又离根较近**的结点。

2、LC-检索的过程

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**：依据层级数来生成结点，令

  $\hat{g}(X)=0$；$f(h(X))=X$ 的级数（已发生成本）.

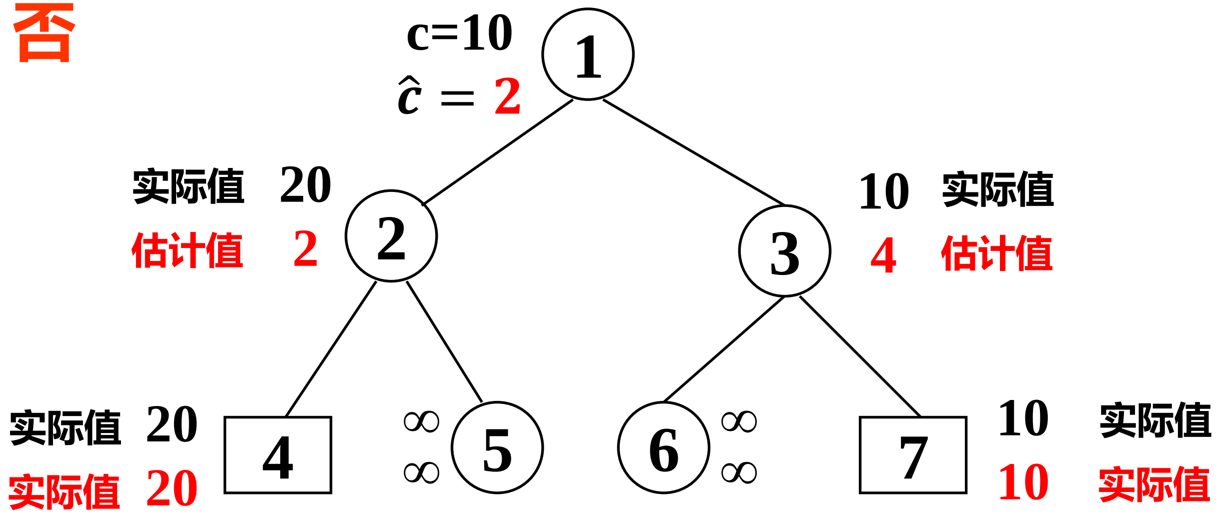
- **D-Search**：令 $f(h(X))=0$；而当 Y 是 X 的一个儿子时，总有 $\hat{g}(X)\geq\hat{g}(Y)$。

LC分支-限界检索：使用**限界函数**的LC-**检索。**

```
procedure LC(T, ĉ)      // ĉ 是结点成本估计函数//
  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    // 父指针，用于构建结点到根的路径//
     repeat
     if 活结点表为空 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 E，并从活结点表中删除//
  repeat
end LC
```

4、最优化问题：找最小成本答案结点

- 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**

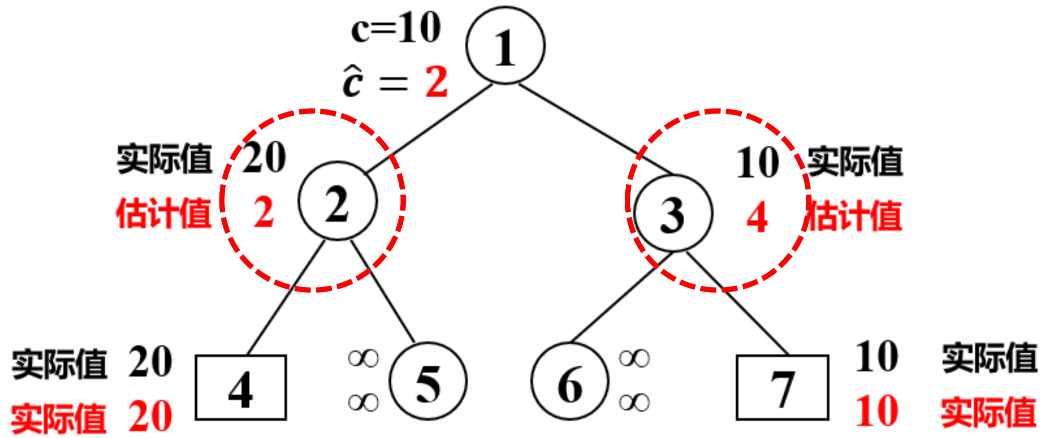


- 首先扩展**结点**1，得**结点**2, 3；注 $\hat{c}(2)=2<\hat{c}(3)=4$；
- 然后扩展结点2，得答案结点4，$c(4) = 20$；
- **实际最小成本的答案结点是**7，$c(7) = 10$。

依赖"**估计**"的**准确性**

◆ 造成上述问题的根本原因在于估计不准确：

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



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 总会找到一个最小成本的答案结点（如果状态空间树中有答案结点的话）。

定理2 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 $\hat{c}(Y) < \hat{c}(Z)$。

那么在使用 $\hat{c}(X)$ 作为 $c(X)$ 的估计值时，算法LC到达一个最小成本的答案结点终止。

证明：（略）

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** $\hat{c}(X) = c(X)$ 。

以上改进的目的：保证算法能找到**最小成本的答案结点**。

以下**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。

◆ 找最小成本答案结点的算法——LC1

```
procedure LC1(T, ĉ)      //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    // 父指针，用于构建结点到根的路径//
      repeat
      if 活结点表为空 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 LC1
```

LC1 能在加快搜索速度的基础上**保证找到成本最小的答案结点。**

定理3：令 $\hat{c}(\bullet)$ 是满足如下条件的函数，

在状态空间树 T 中，对于**每一个结点** X，有 $\hat{c}(X) \le c(X)$，而对于 T 中的**每一个答案结点** X，有 $\hat{c}(X) = c(X)$。

如果算法在第一个 ***return*** 处终止，则**所找到的答案结点是具有最小成本的答案结点。**

证明：（略）

22.6 15-谜问题

问题描述：在一个分成 **4×4** 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**，如图 (b) 。

合法移动：每次将一个**邻接于空格**的牌**移到空格位置**。

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

- **问题状态**：15 块牌在棋盘上的**任意一种排列**规定了当前棋盘的一个状态。
- **初始状态**：由**初始排列**给出，可为任意给定的一种排列，如图a。
- **目标状态**：由**目标排列**给出。目标状态是确定的。

不失一般性，设目标状态是 15 块牌+空格的顺序排列（如图b），其中**空格可视为编号为 16 的牌。**

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

(a) 一种初始排列

问题状态

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

(b) 目标排列

目标状态

1、目标状态是否可由初始状态到达?

若状态 *t* 可由一个起始状态 *s* 经过一系列合法移动而变换得到，则称**状态 *t* 可由状态 *s* 到达**。

一个**状态** *s*，经过任意合法移动而能够到达的所有状态的集合称为**状态 *s* 的可达状态空间**。显然**可达状态空间**中的所有状态都可由 *s* 到达。

可证明： 对15-迷问题，并不是所有的初始排列都能变换成目标排列。

因此在求解15-迷问题前，首先要**判定目标状态是否在初始状态的可达状态空间中**。如果不在，则该初始状态下的15-迷问题无解（即无法通过合法移动将 15 块牌从该初始状态下的初始排列变换成目标排列）。

◆ 如何判定目标状态在初始状态的可达状态空间中?

(1) **给棋盘的方格编号**：按**目标状态**各块牌在棋盘上的位置给所在的方格进行编号。

即：方格编号=目标状态下各块牌的编号，空格编号为 16

如：

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

对棋盘上的方格编号

(2) 记 **POSITION(*i*)** 为编号为 *i* 的牌的**初始位置**，即**初始状态时牌 *i* 所在的方格的编号**。**POSITION(16)** 表示**空格**的初始位置。

如图(a)，有：

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

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

(a) 一种初始排列

(3) 记 **LESS(*i*)** 是这样的牌 *j* 的数目（包括**空格**）：

$j < i$，但 **POSITION(*j*) > POSITION(*i*)**，

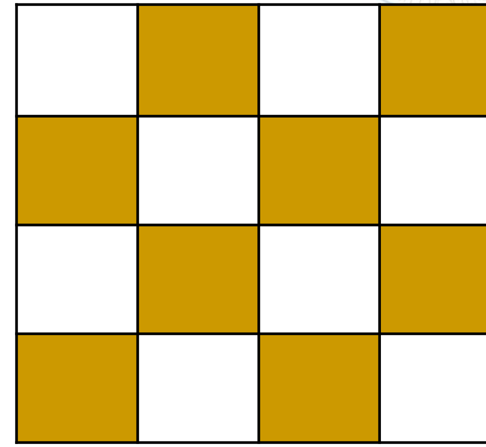
即：编号小于 *i* 但初始位置在 *i* 之后的牌的数目。

如上图，**LESS(1) = 0**；**LESS(4) = 1**；**LESS(12) = 6**

LESS(16) = 10（**空格**，“牌号”16）

（4）引入一个量 X

如图所示，初始状态时，



- 若空格落在**橙色**方格上，则 $X = 1$；
- 若空格落在**白色**方格上，则 $X = 0$。

则：**目标状态是否在初始状态的可达状态空间中的判别条件**由下面的定理给出：

定理：对初始状态，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。

包括空格

证明：（略）

如初始状态图(a):

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

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

(a) 一种初始排列

$X = 0$

$$\sum_{i=1}^{16} LESS(i) + X$$

包括空格

$= LESS(1) + LESS(2) + \ldots + LESS(16) + X$

$= 0+0+1+1+0+0+1+0+0+0+3+6+0+4+11+10+0$

$= 37$

不可达

2、15-迷问题状态空间树的构造

从**初始状态**出发，**通过合法移动来生成后续可能的状态结点。**每个结点的儿子结点是通过一次合法移动而到达的状态，直到能导出目标状态（有解）。

另：**移动非空格牌**与**反向移动空格**是**等效**的，所以为方便描述，状态空间树中标注空格的移动（而不是其它牌的移动）。

空格的一次合法移动有四个可能的方向：**上、下、左、右**。

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

15-谜问题实例展示

设棋盘有如下初始棋局：

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

一棋局的初始状态

0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

$$\sum_{i=1}^{16} LESS(i) + X$$

$$= 0+0+0+0+0+0+0+1+1$$

$$+1+0+0+1+1+1+\mathbf{9}+\mathbf{1}$$

$= \mathbf{16}$ **可达**

剪枝策略：若结点 ***P*** **的某儿子结点**与 ***P*** **的父结点重复**的，则剪去该儿子分枝。

15-谜问题的FIFO检索

结点编号：1

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

初始状态

上

2

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

右

3

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

下

4

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

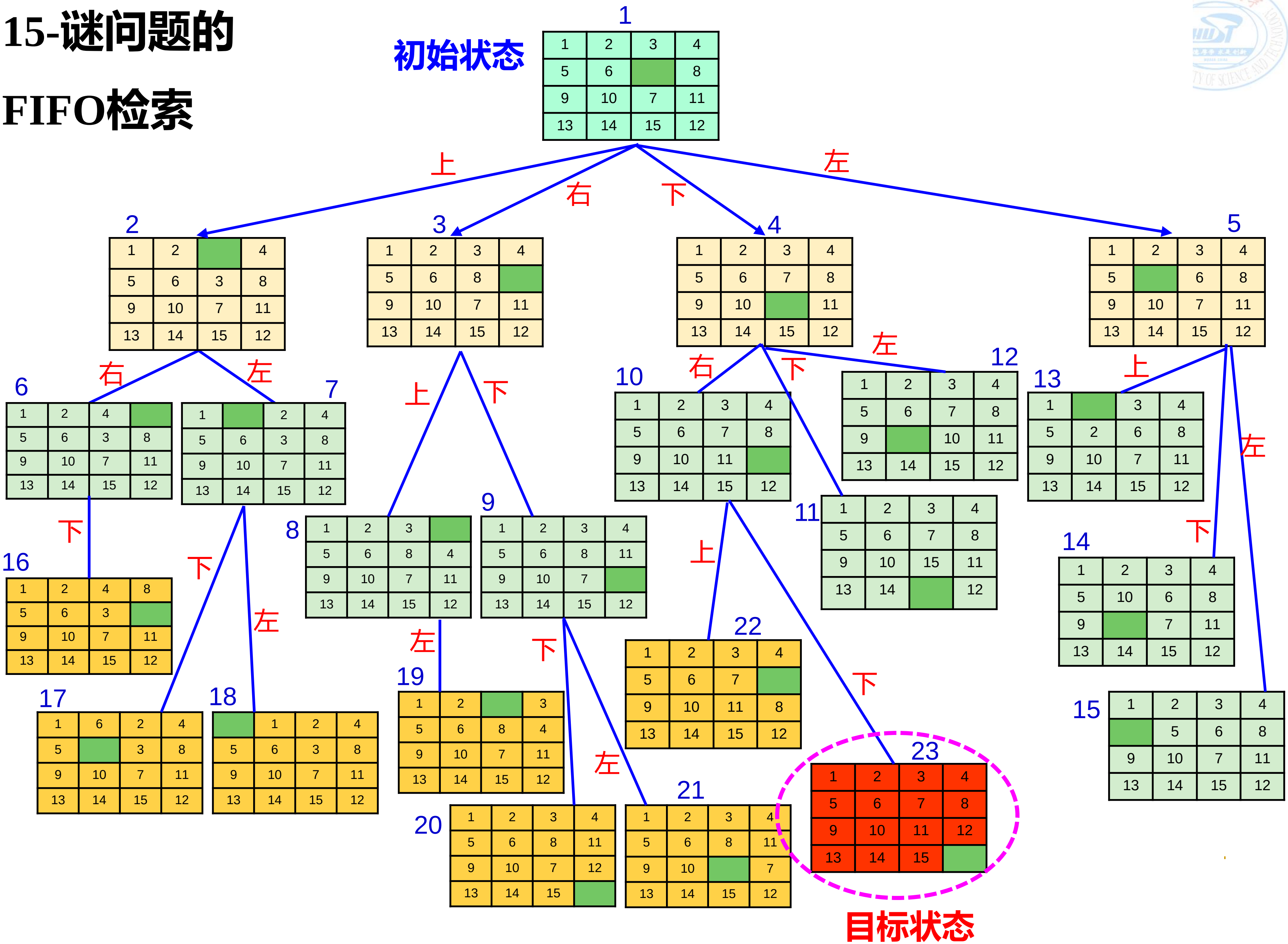
左

5

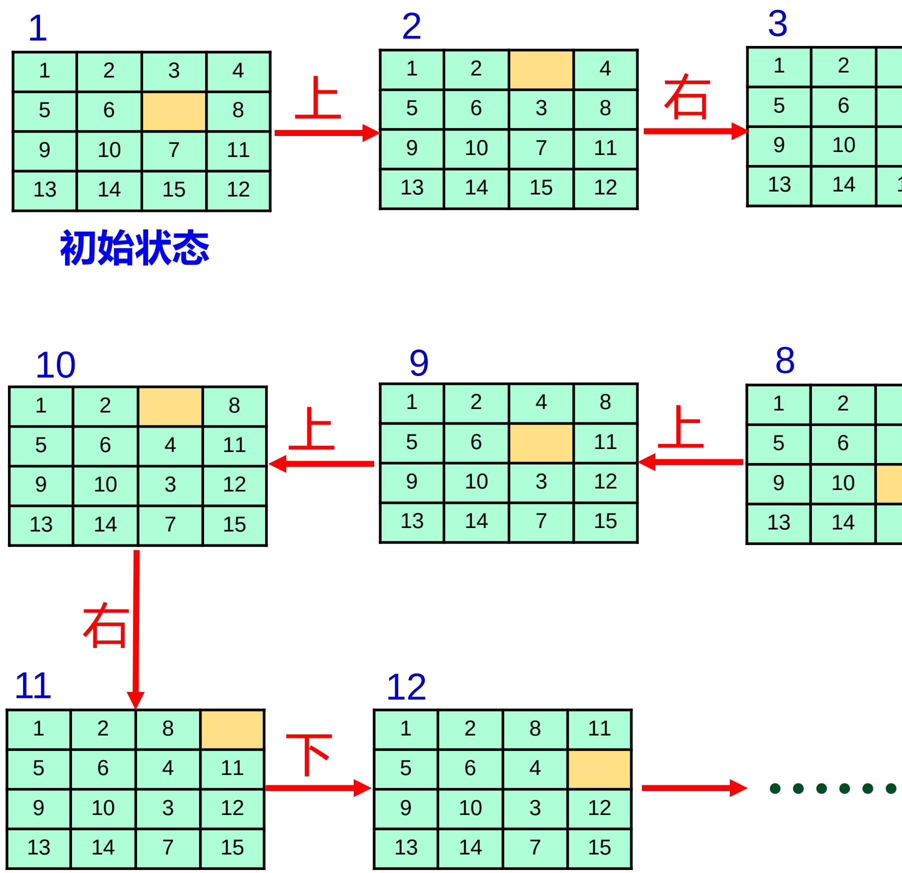
1	2	3	4
5		6	8
9	10	7	11
13	14	15	12

15-谜问题的

FIFO检索



15-谜问题的LIFO检索



3、15-谜问题的 LC-检索

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。等于什么呢?** ——至少应等于**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**

如下图，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 $>> \hat{g}(X)$，

$\hat{c}(X)$ **仅是** $c(X)$ **的下界**。

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

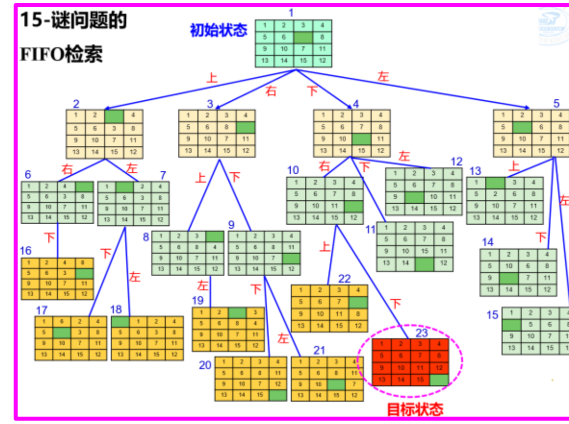
1	2	3	4
5	6		8
9	10	**7**	**11**
13	14	15	**12**

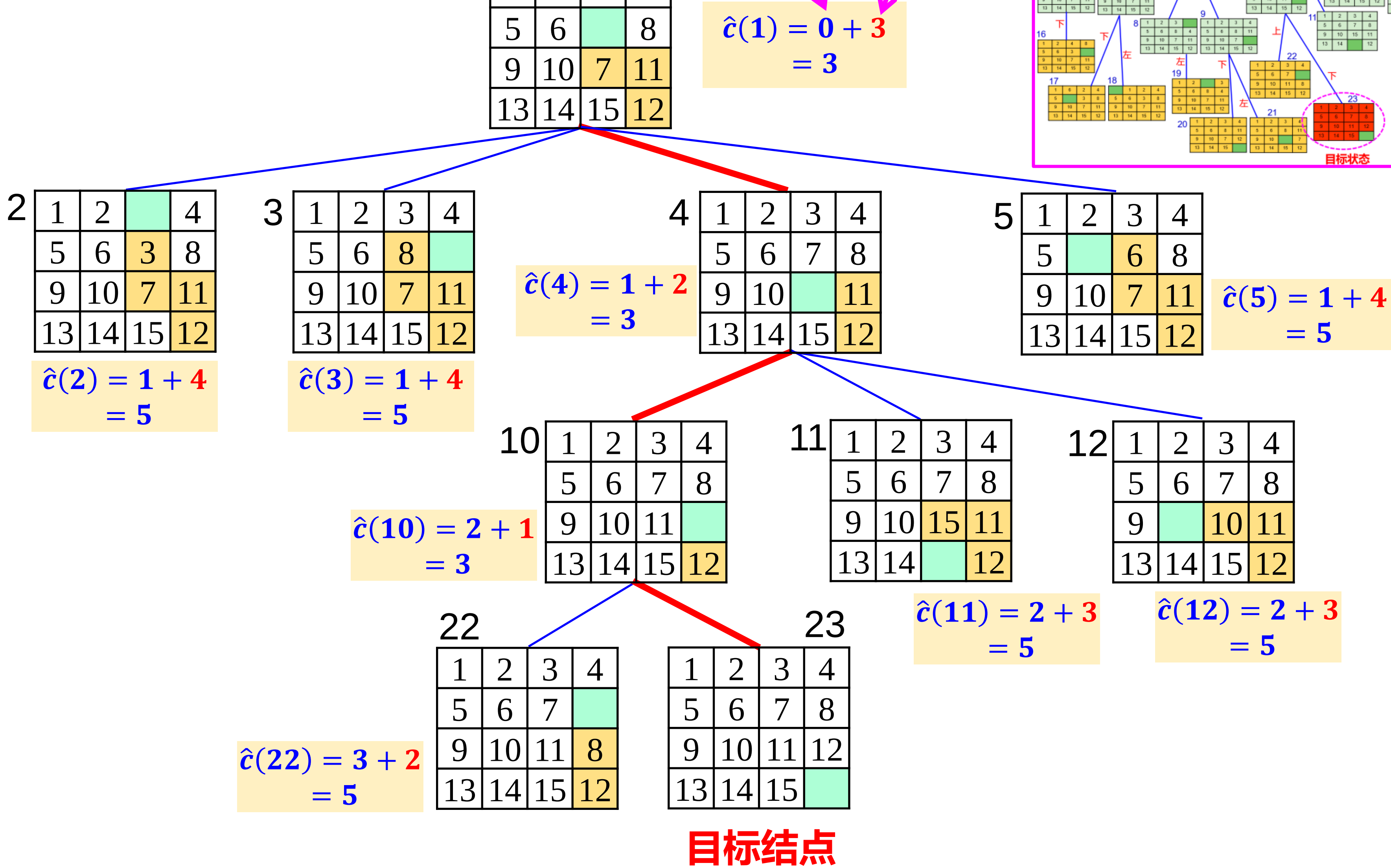
至少要把 7、11、12 移到它们的目标位置

注： $\hat{g}(X)$ 只是估计值，如何估计需要根据问题来进行“设计”。

例：使用 $\hat{c}(X)$ 的LC-检索

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





可见，使用 $\hat{c}(X)$ “**很快**”地找到了目标结点

22.7 利用分支-限界算法求解最优化问题

1、结点的成本函数：下界与上界

假定每个答案结点 X 有一个与其相联系的成本函数 $c(X)$，且**找成本最小的答案结点**。

(1) 最小成本的下界： $\hat{c}(\bullet)$

$\hat{c}(X)$ 为 X 的**成本估计函数**。当 $\hat{c}(X) \le c(X)$时，$\hat{c}(X)$ 给出由结点 X 求解的**最小成本的下界** —— **启发性函数**，**减少选取 E-结点的盲目性**。

(2) 最小成本的上界： U

能否定义**最小成本的上界**?

◆ 最小成本的上界

定义 U 为最小成本解的成本上界（即**实际成本不会大于 U**），则：**可以杀死所有 $\hat{c}(X) > U$ 的活结点**，从而可以进一步加速算法，减少求解的盲目性。

这是因为：$\hat{c}(X)$是 $c(X)$ 的下界：$c(X) \geq \hat{c}(X)$。所以那些有 $\hat{c}(X) > U$ 结点的 X 可到达的**答案结点的实际成本**不可能小于 U 。

如：当**已经求得**一个**具有成本 U 的答案结点**后，那些估计成本 $\hat{c}(X) > U$ 的活结点因不可能导致更低成本的答案结点，就可以被杀死了。

◆ 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*** **为当前最小成本值**。

注：只要 *U* **的初始值不小于最小成本答案结点的实际成本**，之后利用 *U* 就不会杀死本可以到达最小成本答案结点的活结点（即不会导致错误的解）。

2、利用分支-限界算法求解最优化问题

最优化问题：求能够使**目标函数**取得**最小值或（最大值）**的可行解——**最优解**。

(1) 如何把求最优解的过程表示成与分支-限界相关的检索过程?

- **可行解**：类似于 n -**元组**的构造，把**可行解的构造过程**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中代表最优解的**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**。

2、结点成本函数的定义：

根据结点的性质，分别“**赋予**”结点相应的成本函数 $c(\bullet)$：

- 对**可行解结点**：$c(X)$ = 问题的**目标函数在 X 结点处的取值**。
- 对**不可行解的结点**：$c(X) = \infty$；
- 对目前**部分解结点**：$c(X)$ = **根为 X 的子树中最小成本结点的成本**。

然后设计成本估计函数 $\hat{c}(X)$，**使之具有** $\hat{c}(X) \le c(X)$ 的基本性质。之后由可行解求答案结点，**成本最小**的答案结点就是问题的**最优解**。注： $\hat{c}(X)$ 可基于目标函数进行设计。

实例：带限期的作业排序问题。

1、带有限期的作业排序问题描述：

设有 n 个作业和一台处理机。

◆ 每个作业 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 个单位的**处理时间**；

d_i：表示作业 i 的**完成期限（在此之前完成即可）**；

p_i：表示若作业 i 在期限内未完成要招致的**罚款**。

- 作业可以从任何时间开始，但要**在期限之前完成**。完不成将招致罚款。处理机时间不受限。

- **带有限期的作业排序问题的求解目标**：是从 n 个作业的集合中选取作业**子集** J，并要求：

① J 中的所有作业都**能在各自的期限内完成**；

② 同时使得不在 J 中的作业招致的**罚款总额最小。**

—— 满足上述两个条件的作业子集 J 是问题**最优解**。

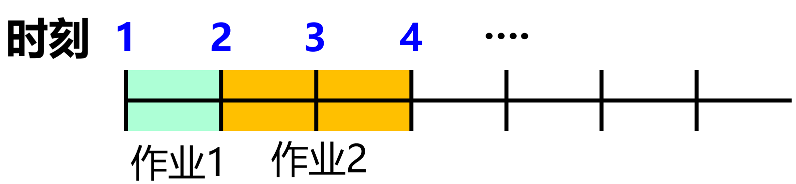
③ **作业排序**：要为 J 中的作业安排一个**合理的执行顺序**，使得 J 中的作业可以按照该顺序**依次执行**，不发生时间冲突且都能在各自的期限内完成。

实例：设有 $n = 4$ 个作业：

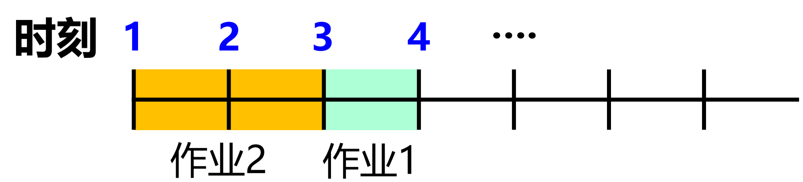
$$(p_1, d_1, t_1) = (5, 1, 1); \quad (p_2, d_2, t_2) = (10, 3, 2);$$

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

- 若选中的作业集 $J=\{1\}$，则**在第1时刻执行作业1**即可，**罚款是19**；
- 若选中的作业集 $J = \{1,2\}$，则可在**第 1 时刻执行作业** 1、**第 2 时刻执行作业** 2，两个作业都可完成，**罚款是9**。(**但若在第 1 时刻执行作业2、第2时刻执行作业1，则作业1就会超期，不能顺利完成，所以对作业集{1,2}，合理的执行序列是** <1,2>)



正确的执行顺序



错误的执行顺序

- 同理，对作业集 $J = \{2,3\}$，合理的执行序列是 <3,2>，**罚款是 8**；**可以证明这是该实例的最优解**（如果执行序列是 <2,3>，则作业3不能顺利完成）。

设有n = 4个作业：

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

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

(1) **状态空间定义**：问题的解空间由作业集 (1,2,3,4) 的所有可能**子集**组成（类似**子集和数问题**，共有$2^4=16$个不同子集）。

(2) **状态空间树**：两种表示形式，

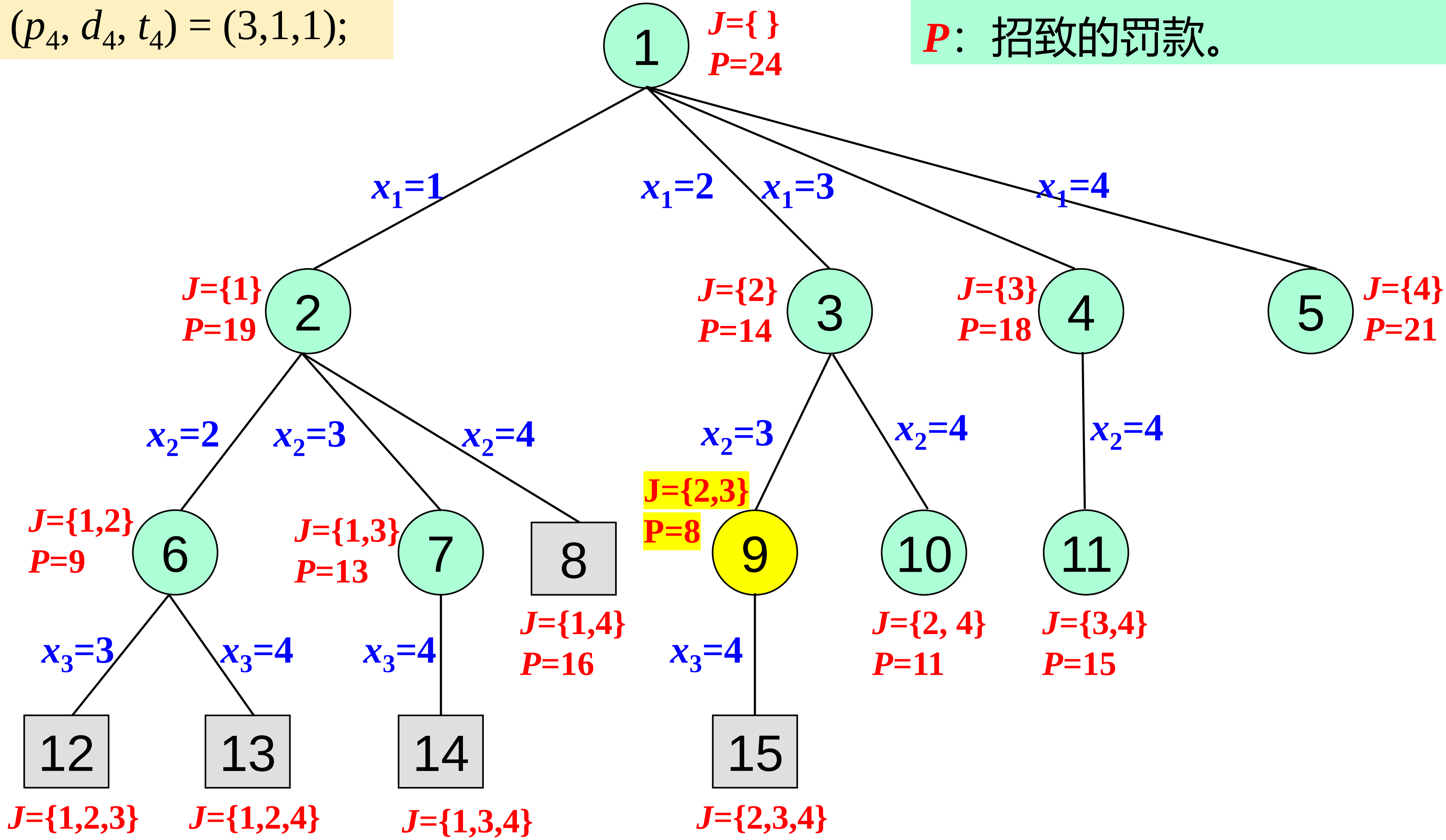
（1）**元组大小可变**的表示：表示选了哪些作业，用选中的作业的编号序列表示。

（2）**元组大小固定**的表示：表示作业是否被选中，每个作业对应一个**0/1**值，$x_i = 1$表示选中了作业 i ，**0**表示没选中。

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

- 圆形结点都是可行结点；
- 方形结点是不可行的子集合。

J：选中的作业子集；
P：招致的罚款。

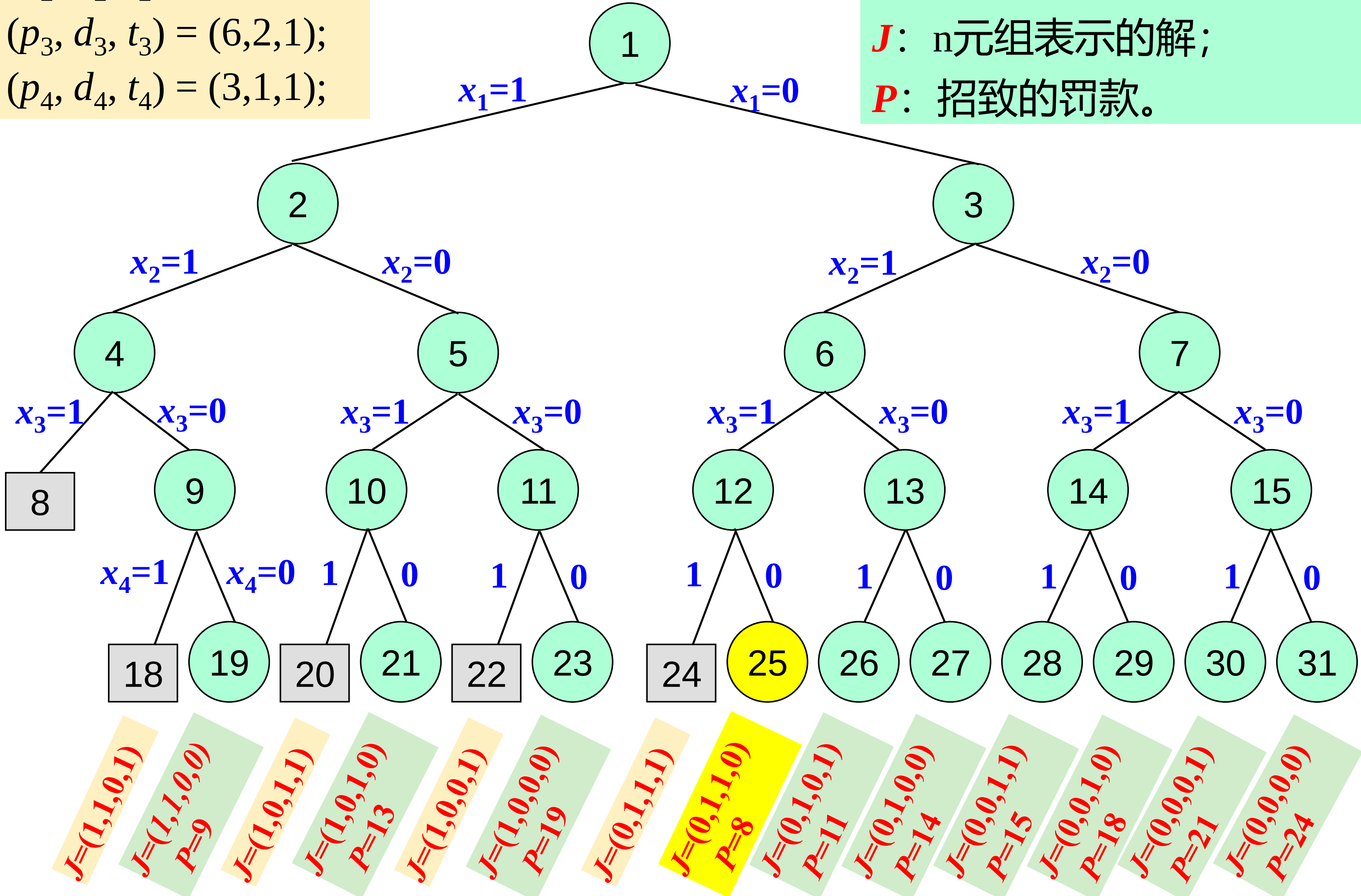


采用k-元组表示的带有限期的作业排序问题状态空间树

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

- 圆形叶子结点代表可行解；
- 方形叶子结点是不可行解。

J： n元组表示的解；
P： 招致的罚款。



采用 *n* 元组表示的带有限期的作业排序问题状态空间树

(3) **成本函数** $c(\bullet)$ 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**： $c(X)=\infty$。

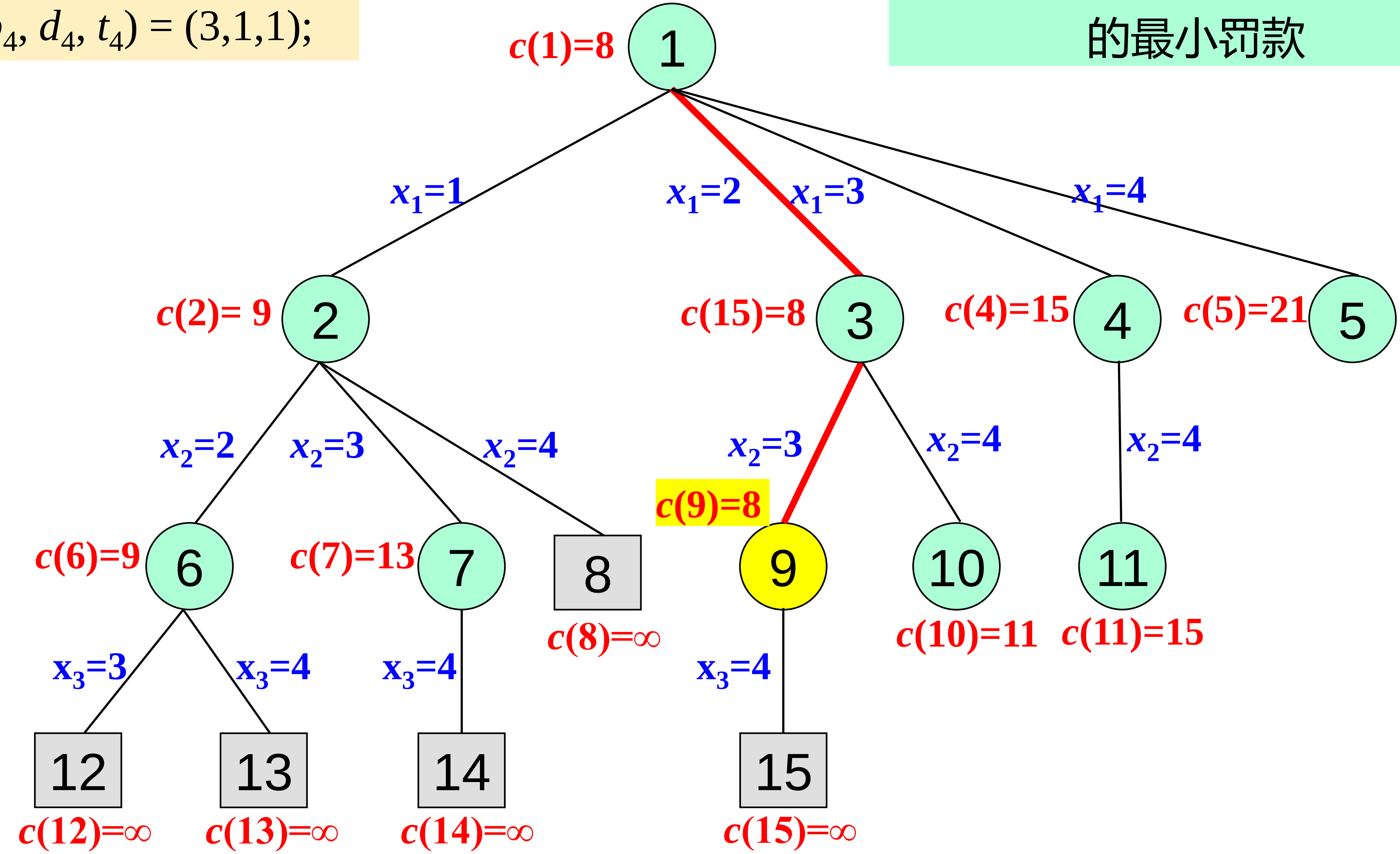
  注： $c(\bullet)$ 是**精确成本**。

例：如下面的图所示。

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

- 圆形结点都是可行结点；
- 方形结点是不可行的子集合。
- $c(X)$：根为 X 的子树中结点的最小罚款

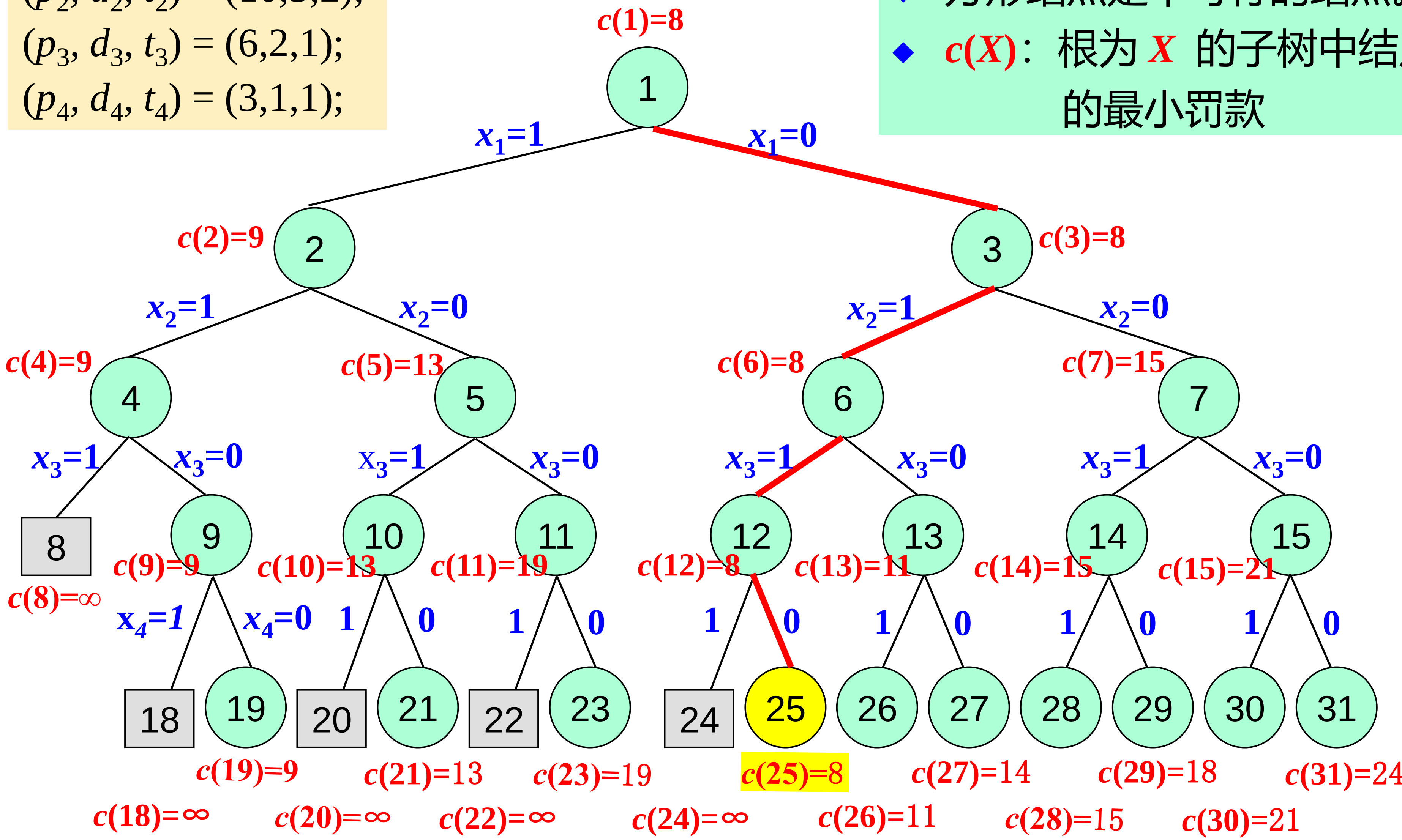
$c(1)$：最优解的罚款



采用 k-元组表示的带有限期的作业排序问题状态空间树

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

- 圆形结点都是可行状态结点；
- 方形结点是不可行的结点。
- $c(X)$：根为 X 的子树中结点的最小罚款



采用 n 元组表示的带有限期的作业排序问题状态空间树

(4) 成本估计函数 $\hat{c}(\bullet)$ 的定义

设 S_X 是考察结点 X 时，已计入到 J 中的作业的集合。

令：$m = max\{ i \mid i \in S_x \}$ （即m是 S_x 中的**最大结点编号**）。

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

不失一般性，假设按结点编号从小到大的顺序选择作业

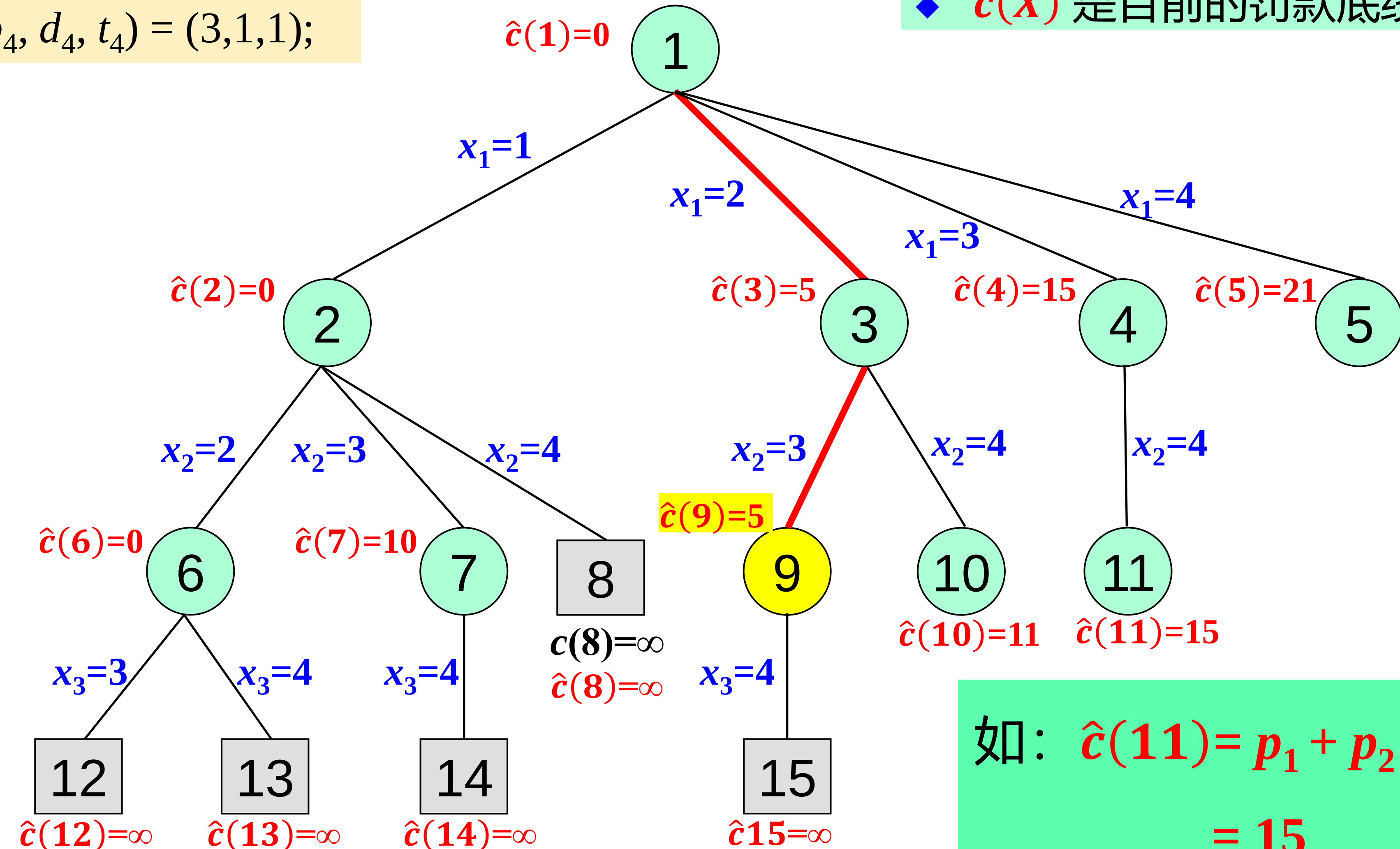
则 $\hat{c}(X)$ 是具有 $\hat{c}(X) \le c(X)$ 的**估计值**（**下界**）。

- $\hat{c}(X)$**的含义**：已经被考虑过、但没有被计入 J 中的作业的**罚款合计**。显然这样的 $\hat{c}(X)$ 是目前已确定的罚款数，是目前**罚款的底线**。

例：如下面的图示。

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

- 圆形结点都是可行结点；
- 方形结点是不可行的子集合。
- $\hat{c}(X)$ 是目前的罚款底线。



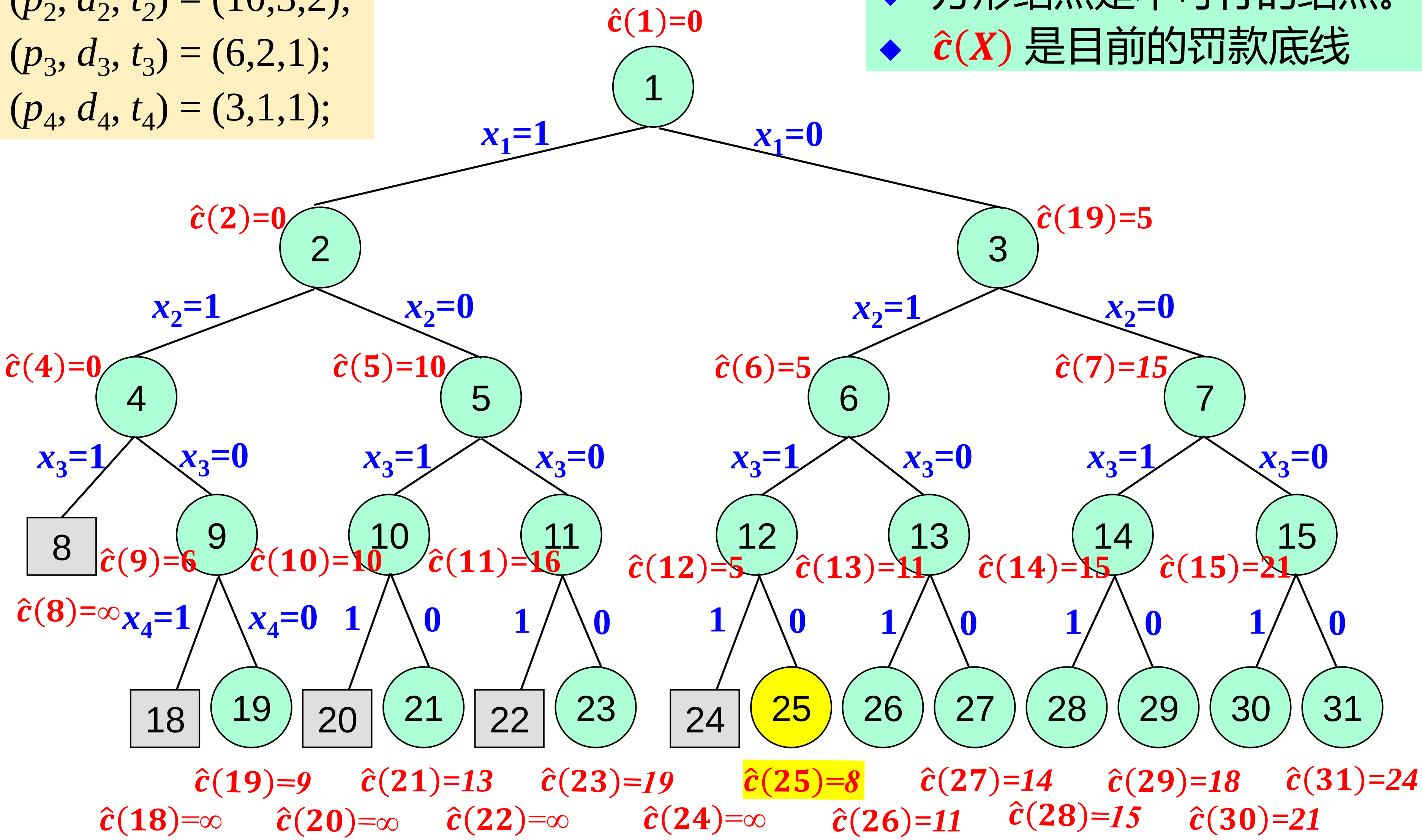
如：$\hat{c}(11) = p_1 + p_2$

$= 15$

采用 k-元组表示的带有限期的作业排序问题状态空间树

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

- 圆形结点都是可行状态结点；
- 方形结点是不可行的结点。
- $\hat{c}(X)$ 是目前的罚款底线



采用 n 元组表示的带有限期的作业排序问题状态空间树

如：$\hat{c}(27) = p_1 + p_3 + p_4 = 14$

（5）成本估计函数的上界

定义： $u(X)=\sum_{i\notin S_X} p_i$ 是 $c(X)$ 的一个“简单”上界。

- $u(X)$ **的含义**：是处理到 X 时，**所有还没有计入到 J 中的作业的罚款合计**，是 X 可能的**最多罚款**（即 X 面临罚款的上限）。

再定义： $U=\min(u(X))$

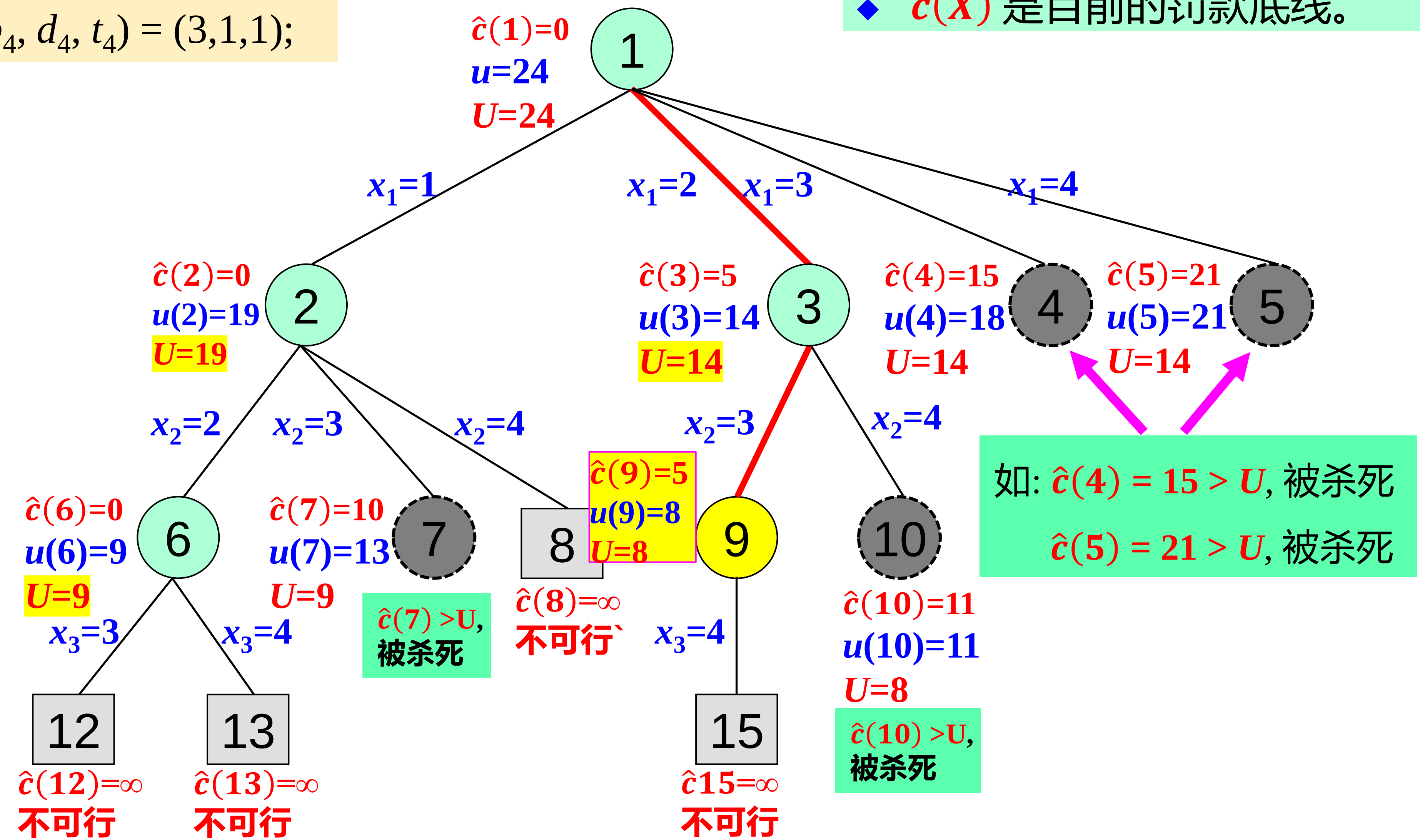
则根据 U 的定义，U 是成本估计函数一个“全局”的上界，
是**目前已知的所有结点罚款的最小值**。

如果对于某个结点Y，若 $\hat{c}(Y)>U$，则Y 就没有进一步扩展的必要，因为基于Y 不可能获得比 U 还小的罚款，它比其它结点要“**劣**”，所以**不可能是（或包含）最优解**。

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

- 圆形结点都是可行结点；
- 方形结点是不可行的子集合。
- $\hat{c}(X)$ 是目前的罚款底线。

$c(1)$：最优解的罚款



采用 k-元组表示的带有限期的作业排序问题状态空间树

以下给出两个搜索过程：

1、利用**FIFO分支-限界算法**求解作业排序问题算法**FIFO-BB**：用队列保存活结点，选择活结点不考虑 $\hat{c}(X)$ 的大小。

2、利用**LC分支-限界求解作业排序问题的算法** **LC-BB**：用优先队列保存活结点，根据 $\hat{c}(X)$ 的大小选择活结点。

1、利用**FIFO分支-限界算法**求解作业排序问题算法**FIFO-BB**

◆ **算法的设计说明：**

(1) 使用**队列**保存未检测的**活结点**，用 **ADDQ**、**DELETEQ** 过程实现结点的**入队列**和**出队列**操作。

(2) **状态空间树的扩展是从根结点 T 开始的。**

◆ 首先根 T 是开始时刻**唯一的活结点**、也是当前的 ***E*-结点**。

◆ **用 $u(T)$ 初始化 U**，同时**考虑 T 自己是否是解结点**的情况：如果是解结点，令 $U = min\{c(T), u(T)\}$，以进一步**优化 U**。

(这是因为**解结点指可不冲突地完成、但未必是最优答案的结点**，所以**解结点未必是最优结点**，所以即使 T 是答案结点也要继续搜索。

并且 $u(T)$ 是一个设定值，不一定为∞，可能是启发式设定的一个更小的值，所以有可能 $c(T) > u(T)$，故有 $U = min\{c(T), u(T)\}$)

(续)

(3) 设置一个结点变量 ***ans***，令其时刻指向当前找到的**最新“优”结点**，每找到一个**更“优”**的新**解结点**时，要及时更新 ***ans***。

这样，算法结束时，***ans*** **就是最优的答案结点,** $c(ans)$**就是最优目标值**。事实上，算法结束时有 $U = c(ans)$ 。

(4) 设 $cost(X)$ **函数**：即 $c(X)$。若 X **是答案结点**，则 $cost(X)$ 是 X **解的成本**。

(5) 已经有经过设计的 $\hat{c}(X)$：对可行结点必有 $\hat{c}(X) \leq c(X) \leq u(X)$

对不可行结点有 $\hat{c}(X) = \infty$

(6) 已经有经过设计的 $u(X)$，即**结点** X **最低成本的上界**。可以用启发式方法设置，也可以指定为∞，并在后续计算中修正。

⑦ 每有更小的 $u(X)$ 计算出来时修正 U 值。

当结点 X 从活结点表出来将变成 E ***结点***时，

- 若 $\hat{c}(X) > U$，X 被立即杀死。然后继续考虑活结点表中的其它活结点
- 若 $\hat{c}(X) = U$，根据 U **的得来**作如下处理：

(I) U 是一个已找到的解的成本：杀死 X

注：至少已经有了一个成本等于 U 的解，这个 X 对最优解没有任何意义。

(II) U 是一个单纯的上界：X 成为 E 结点，继续扩展

注：当前的最好解还没找到，X 是有可能导致成本等于 U 的解的结点。

为便于比较计算，引入一个**很小的正常数** ε，ε 足够小，使得对于任意两个可行的结点 X 和 Y，若 $u(X) < u(Y)$，则

$$u(X) < u(X) + \varepsilon < u(Y)$$

在扩展 E 结点得到一个儿子结点 X 时若有 $\hat{c}(X) < U$：

（I） 若 X 是答案结点且 $c(X) < U$ 时，

$$U = min(c(X), u(X) + \varepsilon)$$

U 或者等于更小的实际成本，或者比已知的最小结点上限成本大一点

注：若$u(X) < c(X)$，则根为 X 的子树中有更优的解。

（II）否则，$U = min(U, u(X) + \varepsilon)$

如果 X 的结点上限成本更小，则修正 U)

引入ε，使得还没有明确上界的时候以 $u(X) + \varepsilon$ ）定义U，而使得$\hat{c}(X) \geq U$ 时可以简单处理。

(续)

若$\hat{c}(X) \geq U$ (即当前E-**结点**儿子的 $\hat{c}(X) \geq U$)，直接杀死 X。

(这是因为有 ε 的存在，**所以 U 或者等于目前已找到的最小实际成本，或者比已知的最小结点成本上限大那么一点**，所以即使 $\hat{c}(X) = U$，X 也不可能导致更好的解。所以这样的儿子结点可以直接杀死)

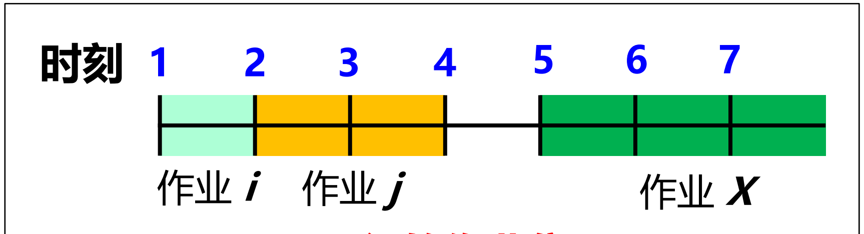
（7）清理队列中的“无效结点”：

在算法的执行过程中，一方面，新扩展出来的“儿子结点”只要可行就加入队列。另一方面，一旦找到一个**新“更优”解**，会及时更新 U 而使得 U **变得更小**。那么之前已经在队列的结点可能会因为它们**当初的 $\hat{c}(X)$ 大于现在的 U** 而成为**“无效结点”**，这样的结点要及时清理，不用再扩展它们。（因为**它们已经有** $\hat{c}(X) \geq U$，故不可能导致比现在“**已找到的最好解**”更好的解了）

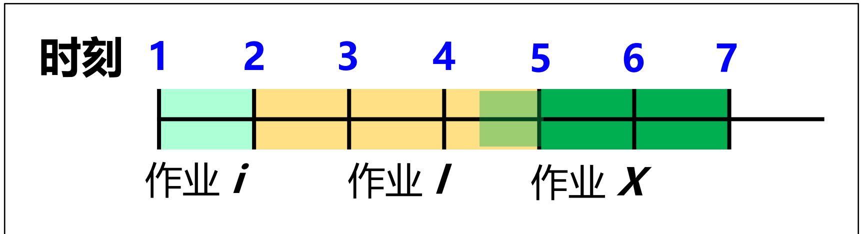
(续)

(8) 可行解的判定：

对于**作业排序问题的可行解判定**，是指：对一个问题实例，如上面的作业1~4，如果按照一定的规则，已经选择了 m 个作业加入了**作业子集** J ，现在考察新作业 X，若 X 的执行时间和 J 中的现有作业都不冲突，即 X 的执行时间区间：$[\,d_X - t_X + 1, d_X\,]$ 与 J 中所有作业的执行时间区间不重叠，则 $J \cup \{X\}$ **是可行解**，X 可并入 J 中；否则 X 不能加入 J。(即**是否存在合理的执行顺序**)



可行的作业集



时间区间重叠，不可行

◆ FIFO-BB算法描述

procedure ***FIFO-BB***(T, $\hat{c}$, u, ε, $cost$)

// 为找出**最小成本答案结点**检索 T，T **至少包含一个解结点**且 $\hat{c}(X) \le c(X) \le u(X)$ //

$E \leftarrow T$; **PARENT**(E) $\leftarrow$ 0;

if T **是解结点** **then**

  $U \leftarrow min(cost(T), u(T) + \varepsilon)$; $ans \leftarrow T$

else $U \leftarrow u(T) + \varepsilon$; $ans \leftarrow 0$

endif

将**队列置空**

（**Continue…**）

- E 是当前 E-**结点**。
- 用根 T **初始化** U，如果根就是一个**答案结点**，则令 $ans = T$。
- ans 始终指向**最新的“优”解。**

(continue)

```
loop
    for E 的每个儿子 X do
        if ĉ(X) < U then
            call ADDQ(X);
            PARENT(X) ← E
            case
                : X 是解结点 and cost(X) < U:
                    U← min(cost(X), u(X) + ε);
                    ans ← X
                : u(X)+ ε < U: U ← u(X)+ ε
            endcase
        endif
    endfor
```

注：这里，那些 $\hat{c}(X) \geq U$ 儿子不入队，直接舍弃（也就是被杀死了）。只有那些 $\hat{c}(X) < U$ 的儿子结点才进入队等待处理。

ADDQ 过程将 X 加入*队列*

PARENT**是父指针**，以后用于重构路径。

找到更好的**答案结点**，**修订** U **和** *ans*
从而保证：只要有更优的答案，*ans*就是这个更好的答案，U 就等于 $c(ans)$。

X 不是答案结点，但**有更好的上界**，则仅**修订** U。

在 *for* 循环中对 E 处理完后，E 就变成死结点了。

(**Continue…**)

(continue)

loop // 有待从队列中取下一个 E-结点继续处理 //

    if 队列为空 then print(“least cost=”, U)

        while ans <> **0 do**

            print(ans)

            $ans \leftarrow$ PARENT(ans)

        repeat

        return // 处理完毕，**算法结束** //

    endif

    call DELETEQ(X) //待从队列中取出新结点，**准备新的检测** //

    if $\hat{c}(X) < U$ **then break**

repeat

repeat

end FIFOBB

根据父指针 **PARENT** **重构**根到答案结点的路径

这里**清理掉**所有的 $\hat{c}(X) \geq U$ 的结点，只对 $\hat{c}(X) < U$ 的新结点跳出内循环、转外循环继续检测。这是因为由于外层 **loop** 循环中有修正的新上界 U，所以对于以前进入队列的结点，不一定有 $\hat{c}(X) < U$，如果 $\hat{c}(X) \geq U_{new}$，这样的结点就没必要处理了，直接丢弃（也就是杀死，不予处理）。

2、 利用**LC分支-限界求解**作业排序问题的算法**LC-BB**

- **算法设计说明：**

（1）采用 ***min*-堆** 保存**活结点表**。

（2）用**ADD**、**LEAST** 过程实现结点的**入堆**和**出堆**操作。

注： **LEAST**从堆中提取的根结点是当前 $\hat{c}(X)$ 最小的结点。

（3）其余约定同于FIFO检索。

（4）当堆中没有活结点或下一个 E 结点的 $\hat{c}(E) \geq U$ 算法终止。

注：从 ***min*-堆** 提取的**下一个 E 结点**是剩下结点里 $\hat{c}(X)$ 最小的结点，如果它的 $\hat{c}(E)$ 都大于 U，则**代表剩下的所有结点都不可能再导致更好的解**，所以算法到此即可终止。

◆ LC-BB 算法描述

line procedure LC-BB(T, $\hat{c}$, u, ε, $cost$)

// 为找出**最小成本答案结点**检索 T，T **至少包含一个解结点**且 $\hat{c}(X) \le c(X) \le u(X)$ //

$E \leftarrow T$; PARENT(E) $\leftarrow$ 0;

if T**是解结点** **then**

  $U \leftarrow min(cost(\mathrm{T}), u(T) + \varepsilon)$; $ans \leftarrow T$

else $U \leftarrow u(T) + \varepsilon$; $ans \leftarrow 0$

endif

将**队列置空**

(**Continue…**)

此部分完全同于FIFO-BB

- E 是当前 E-**结点**。
- 用根 T **初始化** U，如果根就是一个**答案结点**，则令 $ans = T$。
- ans 始终指向**最新的“优”解。**

(continue) **除了ADD以外，此部分完全同于FIFO-BB**

```
loop
    for E 的每个儿子 X do
        if ĉ(X) < U then
            call ADD(X);
            PARENT(X) ← E
            case
              : X 是解结点 and cost(X) < U:
                    U ← min(cost(X), u(X) + ε);
                    ans ← X
              : u(X) + ε < U: U ← u(X) + ε
            endcase
        endif
    endfor
```

注：这里，那些 $\hat{c}(X) \geq U$ 儿子不入堆，直接舍弃（也就是被杀死了）。只有那些 $\hat{c}(X) < U$ 的儿子结点才进入堆等待处理。

ADD 过程将X加入 *min*-**堆**

PARENT是父指针，以后用于重构路径。

找到更好的**答案结点**，**修订** U **和** *ans*

X 不是答案结点，但**有更好的上界**，则仅**修订**U。

在 *for* 循环中对 E 处理完后，E 就变成死结点了。

（Continue…）

(continue)

if 不再有活结点 **or** 下一个 E 结点有 $\hat{c}(E) \geq U$ // 算法结束条件 //

    then print(“least cost=”, U**)**

        while ***ans*** **<> 0 do**

            print(***ans***)

            ans ← PARENT(***ans***) 根据父指针 **PARENT** **重构**根到答案结点的路径

        repeat

        return // 处理完毕，**算法结束** //

    endif

    call LEAST(*E*) //从 ***min***-堆中提取**下一个 *E*-结点**，开始**新的检测** //

repeat

end LC-BB

LEAST从***min***-**堆**提取的**下一个 *E* 结点**是剩下的结点里 $\hat{c}(X)$ 最小的结点。

3、作业排序

最后剩下的一个问题：作业执行顺序

上述**FIFO-BB**和**LC-BB**算法只求出了选中的作业**子集** J，但并没有确定 J 中作业的**执行顺序**。**如何规定 J 中作业的执行顺序**?

用**区间插入法**：

设选中的合法作业集 J 中已有 m 个作业（$m \geq 0$），这 m 个作业按它们的**执行时间区间**排序构成一个合法的作业执行次序：

在**不引起冲突**的前提下，**期限晚的作业尽量排在后面，期限早的作业尽量排在前面**（注：不是单纯按期限早晚排序，同时还要考虑开始时间冲突的问题）。**类似贪心策略**：**将前面的时间尽量留给期限早的作业，自己尽量晚执行**，从而规定出一个作业执行的顺序。

对一个期限为 d，执行时间为 t 的作业，其**最晚开始时间**是 $d-t+1$（只要不晚于这个时间开始，就能保证作业可以在期限内完成），**最晚执行时间区间**是 $[d-t+1, d]$，**如：**

作业1：$(p_1, d_1, t_1) = (5,1,1)$;　最晚开始时间是1，最晚执行时间区间是[1,1]

作业2：$(p_2, d_2, t_2) = (10,3,2)$;　最晚开始时间是2，最晚执行时间区间是[2,3]

作业3：$(p_3, d_3, t_3) = (6,2,1)$;　最晚开始时间是2，最晚执行时间区间是[2,2]

作业4：$(p_4, d_4, t_4) = (3,1,1)$;　最晚开始时间是1，最晚执行时间区间是[1,1]

◆ **每当考察一个新作业 j 是否能被选中时：**

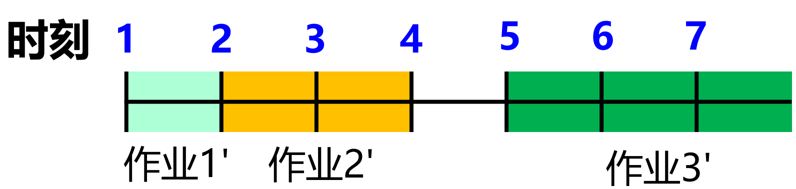
① 如果 j 的**最晚执行时间**比 J 中**期限最大**的作业期限**还大**（包括 J 为空的情况），就将 j 插入到它的最晚执行时间区间 $[d_j - t_j+1, d_j]$中。

② 否则，**从后往前**查找其完成期限 d_j 之前有没有**大小为 t_j 的 "最晚"空闲时间片**：

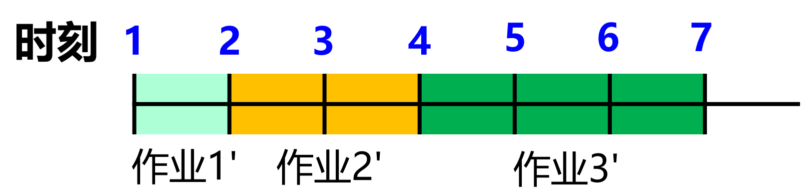
如果 d_j 之前大小为 t_j 的时间片与一个现有作业的执行时间区间冲突，则**试图前移该现有的作业**，看能不能**空出大小为 t_j 的时间片：** (注：这将是一个递归查找过程。如果能够前移，前移后的现有作业依然可行)

◆ 如果能空出这样的空闲时间片 $[t_s, t_s + t_j]$, $t_s + t_j \le d_j$，则将新作业插入到该时间片中即可，并将 j 并入 J 集合 中：$J \leftarrow J \cup \{ j \}$；否则代表 $J \cup \{ j \}$不可行，j **被舍弃**。算法终止时，答案结点的 J 集合中的作业执行顺序给出这些作业的执行次序。

注：上述过程查找并确定 J 集合中作业的一个执行先后顺序，一旦确定，就可以**按照先后顺序连续执行**，而不必按照确定序列时所设定的开始时间执行（在确定好的序列中，按序提前执行任何作业都不会违反作业的期限或引起冲突）。



算法确定的时间片位置



实际执行时，作业3'从时刻4开始执行

以上过程的具体实现请自行思考。

本章小结

1、图基本的检索算法：**BFS、DFS**

2、图基本的周游算法：**BFT、DFT**

3、回溯与分支-限界：**带有剪枝策略的BFS、DFS，限界函数**

4、**LC**-检索

- 结点成本函数 $C(X)$、结点成本估计函数 $\hat{c}(X)$
- 结点成本估计值的下界： $\hat{c}(X)$
- 结点成本估计值的上界：$U = min(u(X))$

5、**N皇后问题、子集和数问题、15-迷问题、带有限期的作业排序问题。**

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**SUMOFSUB**过程找出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。