

---

# 算法设计与分析

## 回溯法

# 主要内容

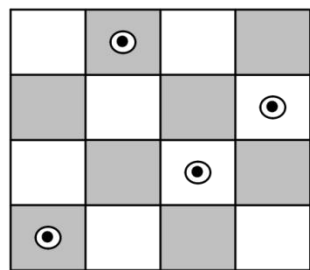
---

- N皇后问题
- 回溯法思想
- 回溯算法
- 回溯法应用

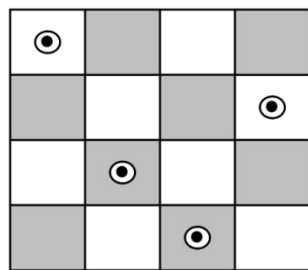
# N皇后问题

在 $n \times n$ 格的棋盘上放置彼此不受攻击的 $n$ 个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。 $n$ 后问题等价于在 $n \times n$ 格的棋盘上放置 $n$ 个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。

$n=1$  显而易见。 $n=2、3$ ，问题无解。 $n \geq 4$  时，以4后为例

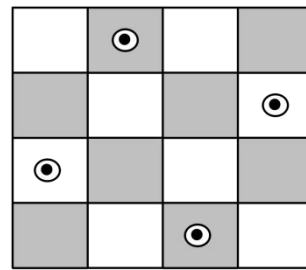


(a)



(b)

4后问题的两种无效布局



4后问题的一个有效布局



- 1、解怎么表示？
- 2、解如何组织？
- 3、怎么找到最优解？

- 1、解怎么表示？
- 2、解如何组织？
- 3、怎么找到最优解？

图 11.2 用回溯法解 4 皇后问题的状态空间树。x 表示一个试图把皇后放在指定列的不成功的尝试。节点上方的数字指出了节点被生成的次序

# 基本概念

---

- 解空间
- 可能解
- 可行解
- 最优解

---

## 解空间

- 问题的解向量  $X = (x_0, x_1, \dots, x_{n-1})$ , ——解的表达形式
- $x_i$  的取值范围  $S_i$ ,  $S_i = \{a_{i.0}, a_{i.1}, \dots, a_{i.m_i}\}$ 。
- 问题的解空间由笛卡尔积  $A = S_0 \times S_1 \times \dots \times S_{n-1}$  构成。
- 例：4皇后问题

解向量  $x = (x_1, x_2, x_3, x_4)$  : 4维的向量

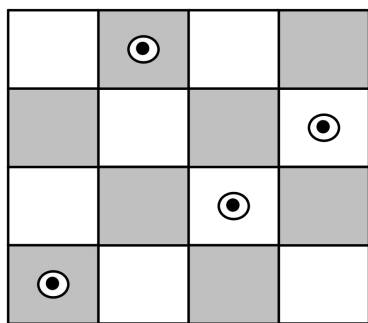
$x_i$  表示第  $i$  行皇后的列位置, 取值范围  $S_i = \{1, 2, 3, 4\}$

解空间——4维向量的全部组合

# 四后问题的解空间

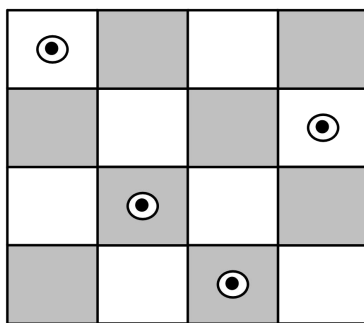
向量  $x = (x_1, x_2, x_3, x_4)$  表示皇后的布局。分量  $x_i$  表示第  $i$  行皇后的列位置。

$x_i$  的取值范围  $S_i = \{1, 2, 3, 4\}$  ， 有  $4^4$  个可能解



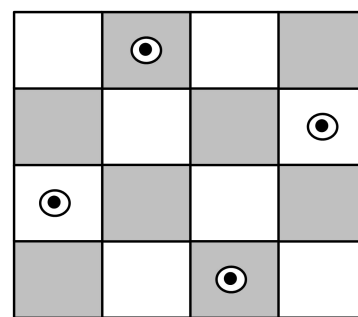
(a)

(2,4,3,1)



(b)

(1,4,2,3)



(2,4,1,3)

# 可行解和最优解

---

- 可行解：满足约束条件的解，解空间中的一个子集
- 最优解：使目标函数取极值（极大或极小）的可行解，一个或少数几个
- 找可行解，一般找到就可以，但是找最优解一般要遍历整棵树。



# 回溯法简介

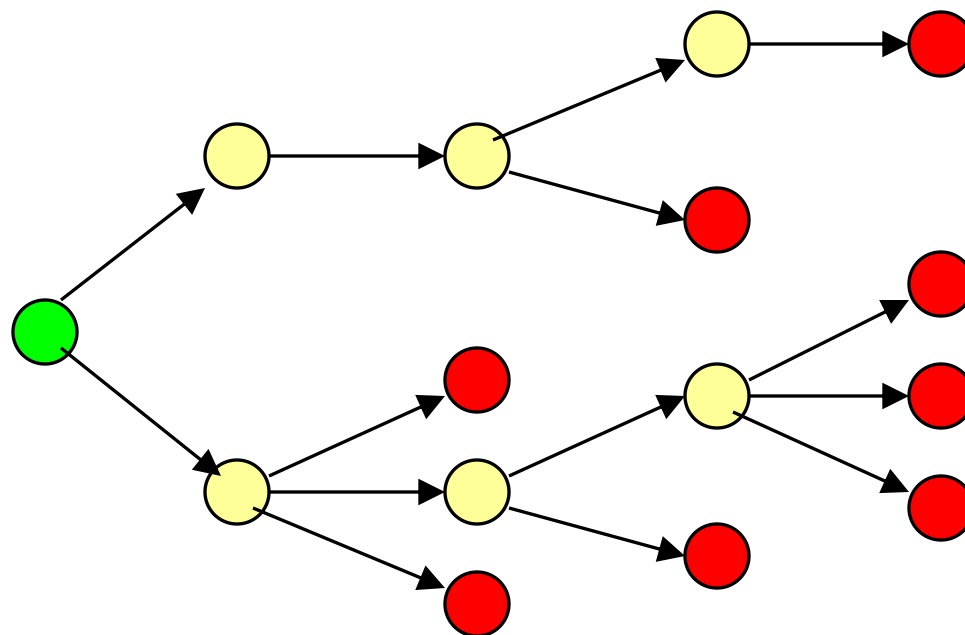
---

有“通用解题法”之称，将所有的解（问题的解空间）按照一定结构排列，再进行搜索。




- 一般解空间构造成树状结构，用深度优先的策略搜索
- 两种方式：
  - 只需要一个解的话，找到解就停止
  - 需要求所有解，则需做“树的遍历”找到所有解。
- 通常用排除法（剪枝），减少搜索空间

# 回溯法术语

树由节点组成：



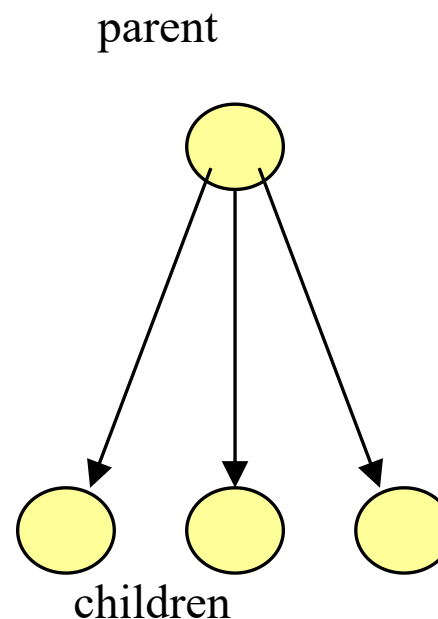
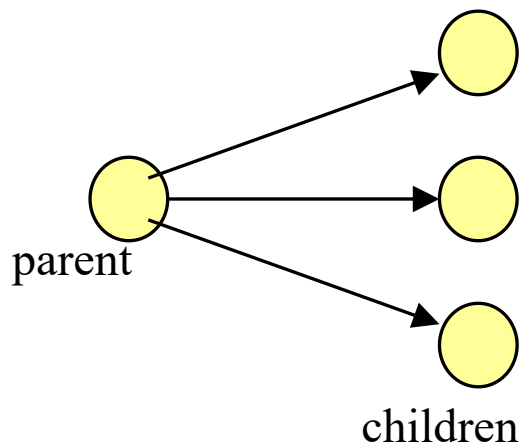
有三种节点：

-  根节点
-  中间节点
-  叶子节点

**回溯法**就是搜索树中某个特定目标节点

# 术语

- 树中每个非叶子节点都是一个或是多个其他节点的父节点
- 树中节点（除了根节点）都只有一个父节点



# 回溯法的关键问题

---

- 节点的含义是什么？（根据问题确定）
- 节点在树中的关系是什么？（根据问题确定）
- 如何产生新的节点？（树的遍历算法）
- 如何判断节点是否是所求解？（easy！）

# 回溯法的基本思想

定义节点

- (1) 针对所给问题，定义问题的解空间；
- (2) 确定易于搜索的解空间结构；
- (3) 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

尽量少产生新节点

产生新节点

定义节点关系

常用剪枝函数：

用约束函数在扩展结点处剪去不满足约束的子树

# 回溯法的存储空间

---

- 用回溯法解题的一个显著特征是在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为 $h(n)$ ，则回溯法所需的计算空间通常为 $O(h(n))$ 。而显式地存储整个解空间则需要 $O(2^{h(n)})$ 或 $O(h(n)!)$ 内存空间。

# 状态空间树

---

- 状态空间树看成为一棵高度为  $n$  的树，
- 第0层有  $|S_0| = m_0$  个分支结点，构成  $m_0$  棵子树，每一棵子树都有  $|S_1| = m_1$  个分支结点。
- 第1层，有  $m_0 \times m_1$  个分支结点，构成  $m_0 \times m_1$  棵子树。
- 第  $n-1$  层，有  $m_0 \times m_1 \cdots \times m_{n-1}$  个结点，它们都是叶子结点。

# 4后问题的约束条件

---

4后问题状态空间树: 4叉完全树

约束方程:  $1 \leq i \leq 4, 1 \leq j \leq 4, i \neq j$

不在同一列: 第*i*行皇后列位置与第*j*行皇后列位置不同

$$x_i \neq x_j$$

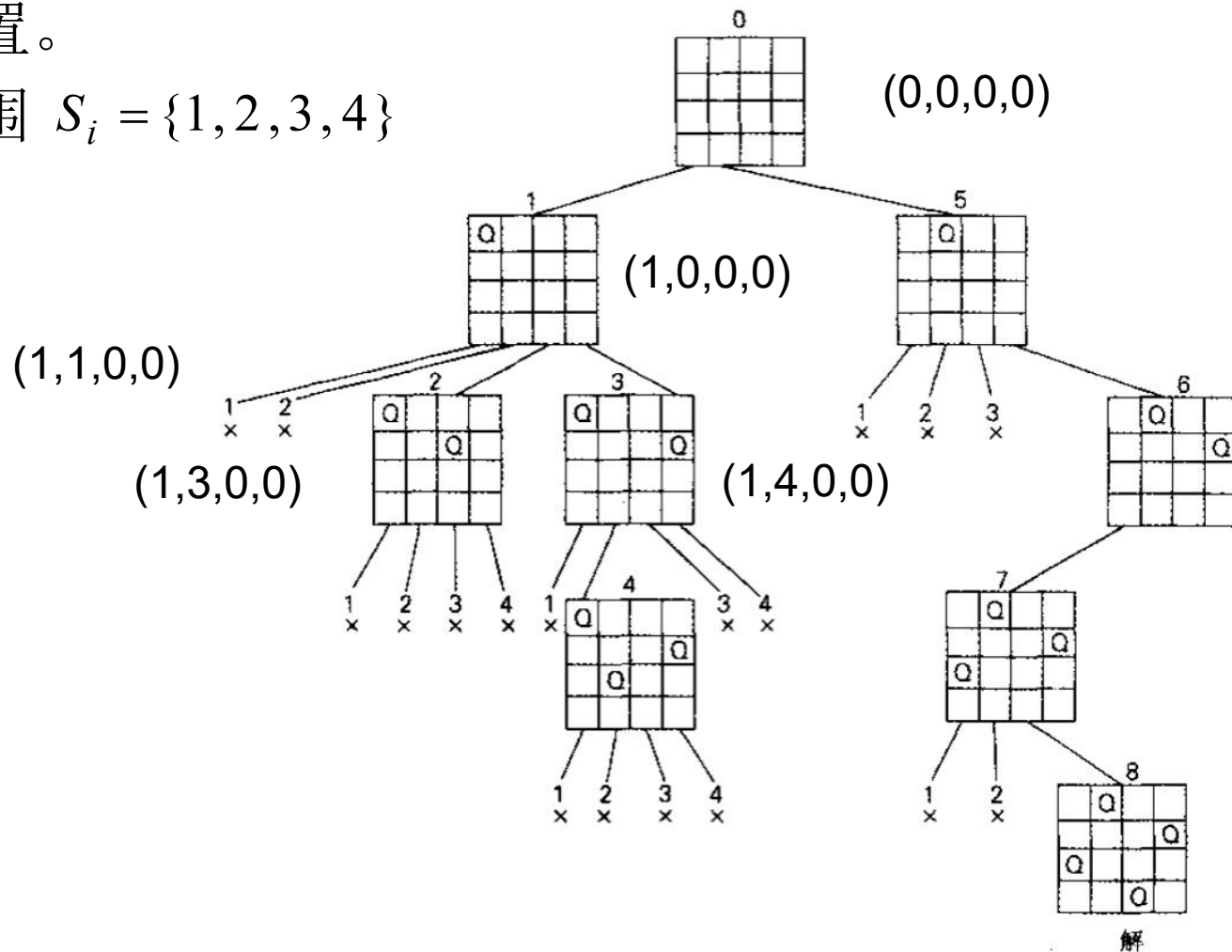
不在同一个斜线上:  $|x_i - x_j| \neq |i - j|$



# 求解过程图示

向量  $x = (x_1, x_2, x_3, x_4)$  表示皇后的布局。分量  $x_i$  表示第  $i$  行皇后的列位置。

$x_i$  的取值范围  $S_i = \{1, 2, 3, 4\}$



# n后问题

- 解向量:  $(x_1, x_2, \dots, x_n)$
- 显约束:  $x_i = 1, 2, \dots, n$
- 隐约束:

1) 不同列:  $x_i \neq x_j$

2) 不处于同一正、反

对角线:  $|i-j| \neq |x_i - x_j|$

1、为什么不检查行?

2、 $x_i \neq x_j$  什么意思?

3、 $|i-j| \neq |x_i - x_j|$  什么意思?

```
bool Queen::Place(int k) //检查前k行是否合法
{
    for (int j=1;j<k;j++) //逐行检查
        if ((abs(k-j)==abs(x[j]-x[k]))||(x[j]==x[k]))
            return false;
    return true;
}
```

```
void Queen::Backtrack(int t)//对第t行放置皇后
{
    if (t>n) sum++; //当层数大于n时，统计解个数
    else
        for (int i=1;i<=n;i++) {
            x[t]=i;
            if (Place(t)) Backtrack(t+1); //如果可以放置
                //继续找下一行位置
        }
}
```

# 生成问题状态的基本方法

---

- 扩展结点:一个正在产生儿子的结点称为扩展结点
- 活结点:一个自身已生成但其儿子还没有全部生成的节点称做活结点
- 死结点:一个所有儿子已经产生的结点称做死结点
- 深度优先的问题状态生成法: 如果对一个扩展结点R, 一旦产生了它的一个儿子C, 就把C当做新的扩展结点。在完成对子树C (以C为根的子树) 的穷尽搜索之后, 将R重新变成扩展结点, 继续生成R的下一个儿子 (如果存在)
- 宽度优先的问题状态生成法: 在一个扩展结点变成死结点之前, 它一直是扩展结点
- 回溯法: 为了避免生成那些不可能产生最佳解的问题状态, 要不断地利用限界函数(bounding function)来处死那些实际上不可能产生所需解的活结点, 以减少问题的计算量。具有有限界函数的深度优先生成法称为回溯法



# 递归回溯

回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=f(n,t);i<=g(n,t);i++) {
            x[t]=h(i);//当前解
            if (constraint(t)&&bound(t)) backtrack(t+1);
        }
}
```

解的第一种可能

解的最后一种情况

# 子集和问题 (1)

---

- **问题:** 给定  $n$  正整数  $w_1, \dots, w_n$  集合, 一个正整数  $S$ , 找到所有子集, 使其和等于  $S$ .
- **举例:**  $n = 3, w_1 = 2, w_2 = 4, w_3 = 6$ , and  $S = 6$ .  
解:  $\{2, 4\}$  and  $\{6\}$ .
- 蛮力法:  $O(2^n)$  —— 问题规模很大时不适用.
- 为了更好解决问题, 利用回溯法。

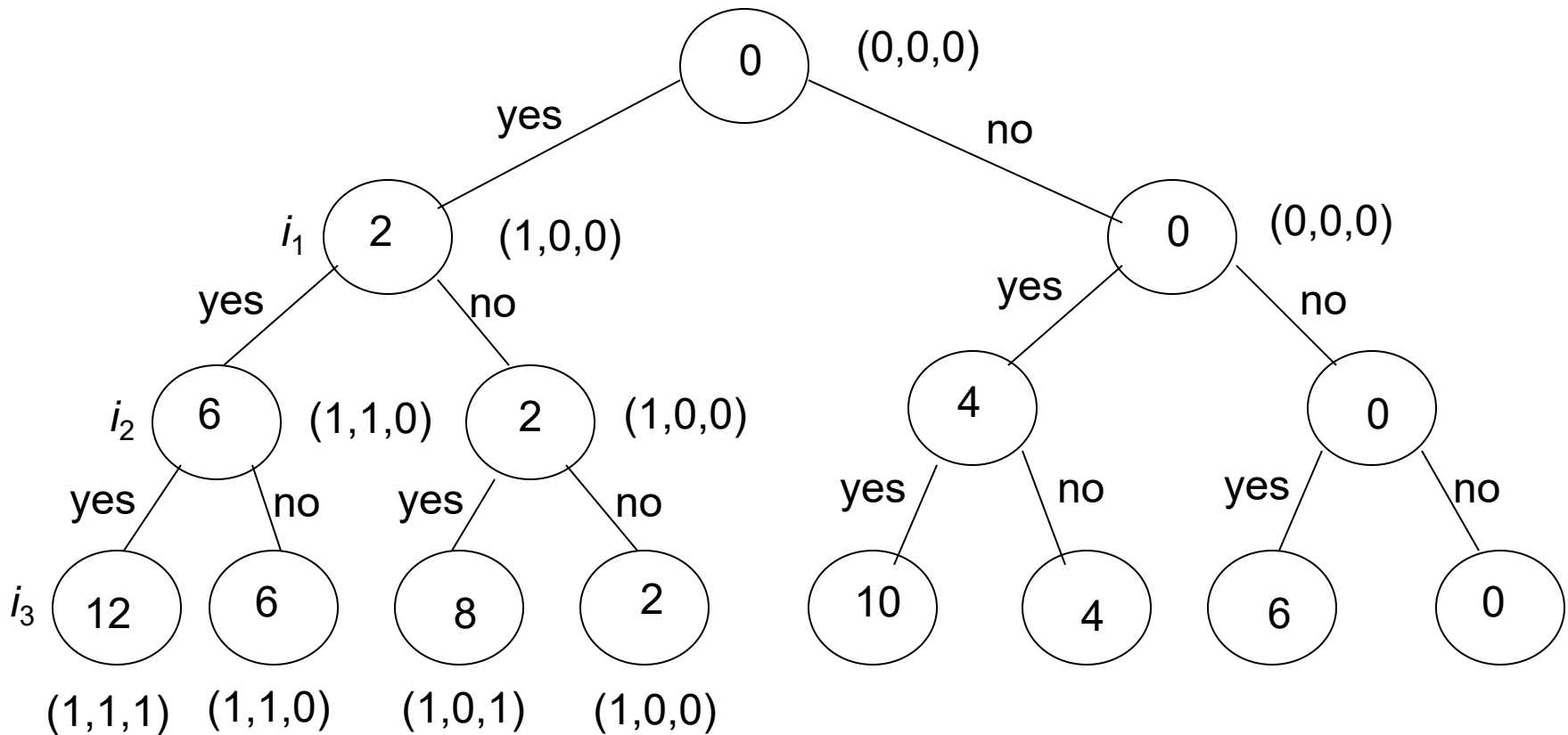
## 子集和问题(2)

---

- 向量 $x=(x_1, x_2, \dots, x_n)$ 表示节点，每个 $x_i$ 的取值范围 $\{0,1\}$
- 我们利用一棵二叉树进行回溯
- 每一行表示一个元素 $w_i$ 的选择情况，标1表示选中，0表示没选中
- 每个节点赋予一个值，表示当前求和大小
- 根节点是0，表示没有元素选定

# 子集和问题(3)

例如:  $w_1 = 2$ ,  $w_2 = 4$ ,  $w_3 = 6$  and  $S = 6$ :

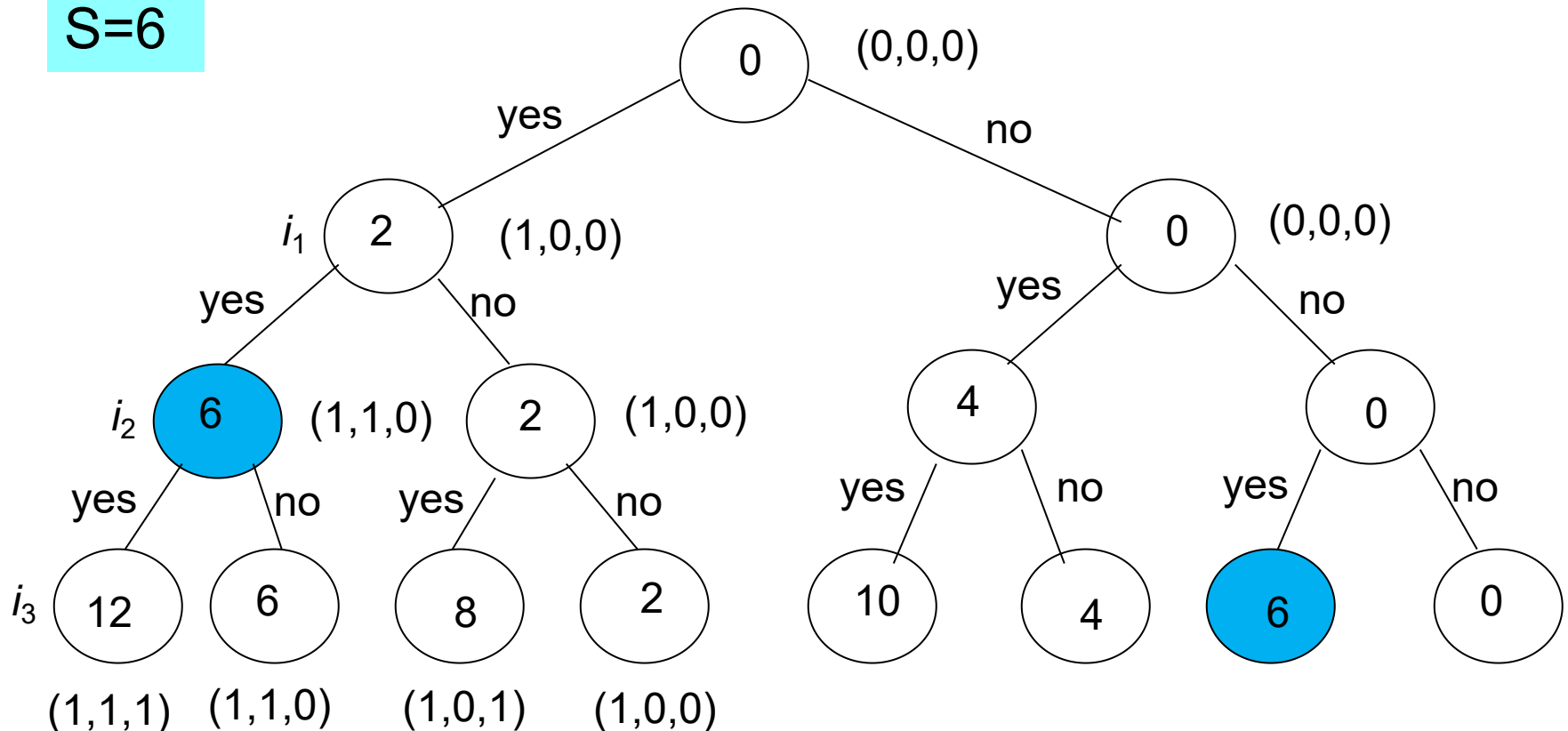




# 如何得到解

- 检查每个叶子节点是否其和为 $S$ ，如果是就返回该节点的路径 (该路径就是解)，或是该节点的状态也是解。

$S=6$

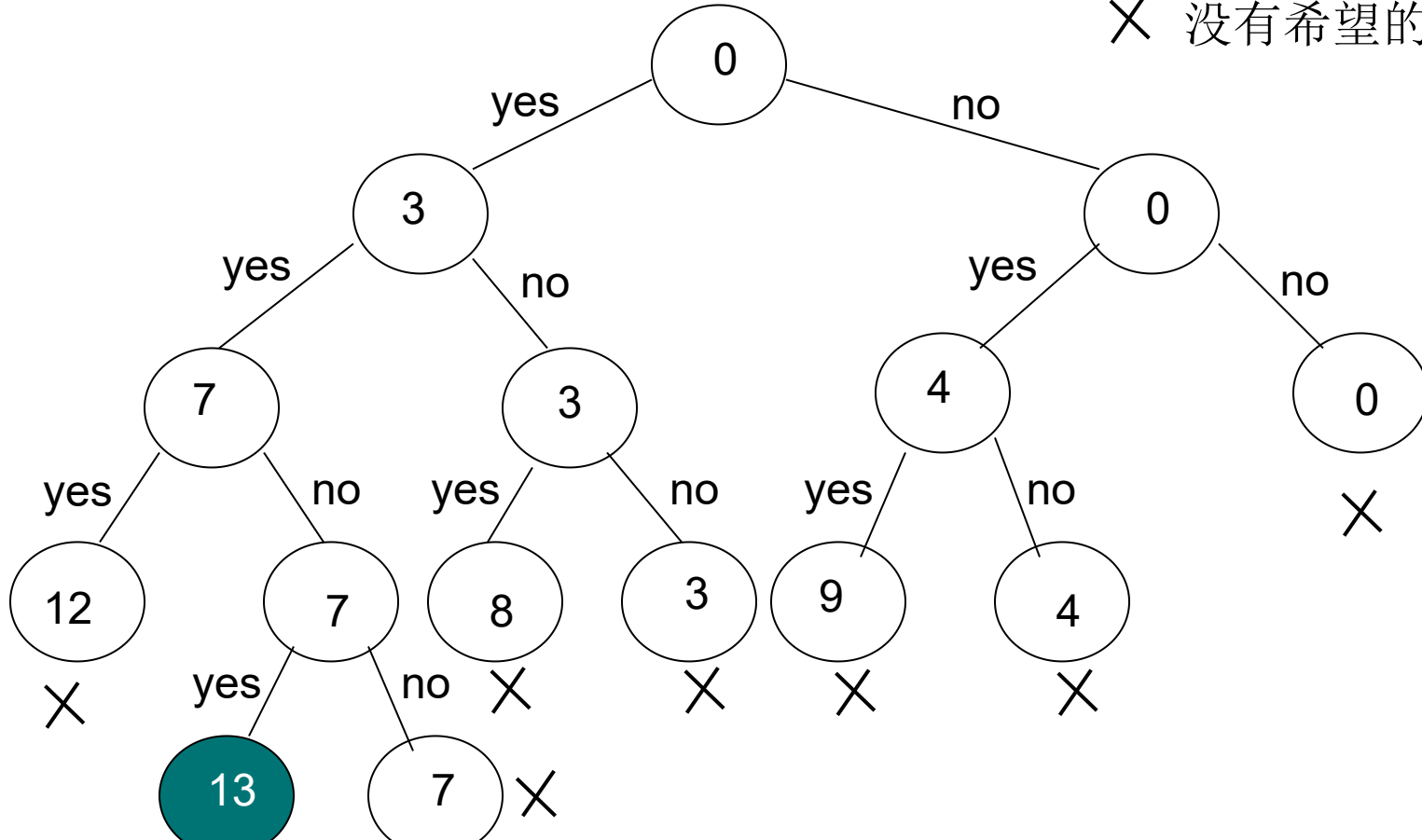


# 回溯法的剪枝技术

- 节点：有希望的/没有希望的
- 如果节点没有希望，其下面的子树都不搜索——剪枝.

例：  $w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$

× 没有希望的节点



# Sum of Subsets Problem

---

- *Question*: 如何确定一个节点是否有希望?
- 首先, 对所有元素排序, 假设  $w_1 \leq w_2 \leq \dots \leq w_n$
- 其次, 对节点评估
  - *weightSoFar* = 当前解的和
  - *totalPossibleLeft* = 剩下从  $i + 1$  到  $n$  元素的和 (当前是第*i*层)

# 有希望 and 没希望

---

- 一个节点是有希望的，如果满足下面两个条件：
  - $weightSoFar + totalPossibleLeft \geq S$
  - $weightSoFar + w_{i+1} \leq S$  or  $weightSoFar = S$
- 一个节点是没有希望的，如果满足下面条件之一：
  - $weightSoFar + totalPossibleLeft < S$
  - $weightSoFar + w_{i+1} > S$

## Sum of Subsets Problem: Algorithm

**sumOfSubsets(*i, weightSoFar, totalPossibleLeft* )**

**1) if (promising(*i*))                      //may lead to a valid solution**

2) if ( *weightSoFar* = *S* )

3) **print *include*[1] to *include*[i] //found solution**

```
4)  else                                     //expand the node
```

**5) include[ $i + 1$ ] = “yes” //try including**

6) **sumOfSubsets( $i + 1$ ,  $weightSoFar + w[i + 1]$ ,  $totalPossibleLeft - w[i + 1]$ )**

7) **include**[ $i + 1$ ] = “no” //try excluding

```

8)      sumOfSubsets(i + 1, weightSoFar,
           totalPossibleLeft - w[i + 1])

```

**boolean promising(*i*)**

```

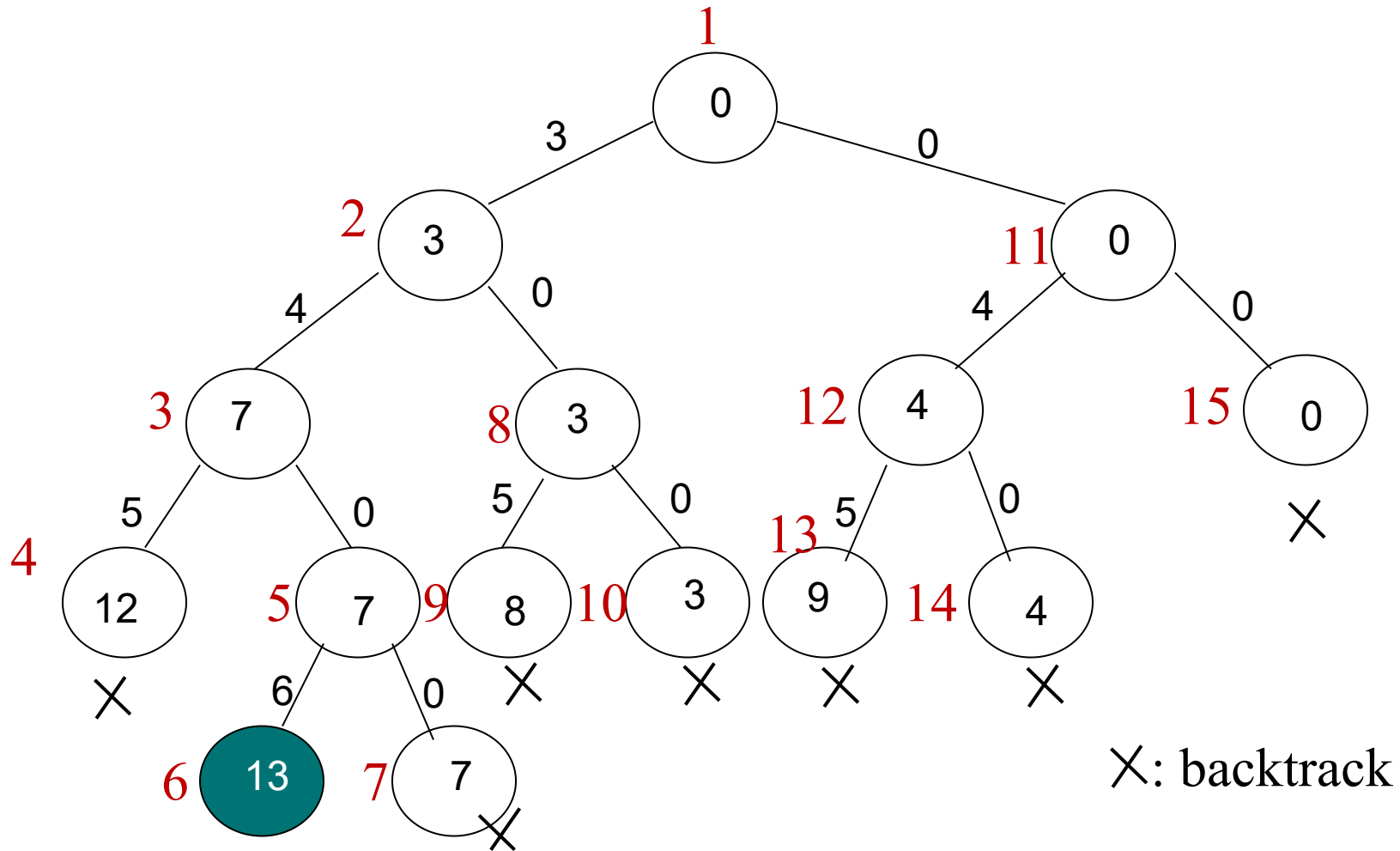
return ( weightSoFar + totalPossibleLeft ≥ S ) &&
        ( weightSoFar = S || weightSoFar + w[i + 1] ≤ S )

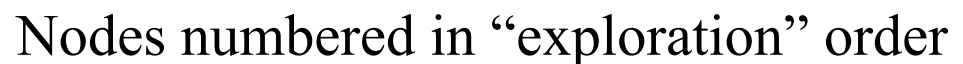
```

- 初始调用  $\text{sumOfSubsets}(0, 0, \sum_{i=1}^n w_i)$

# A Pruned State Space Tree

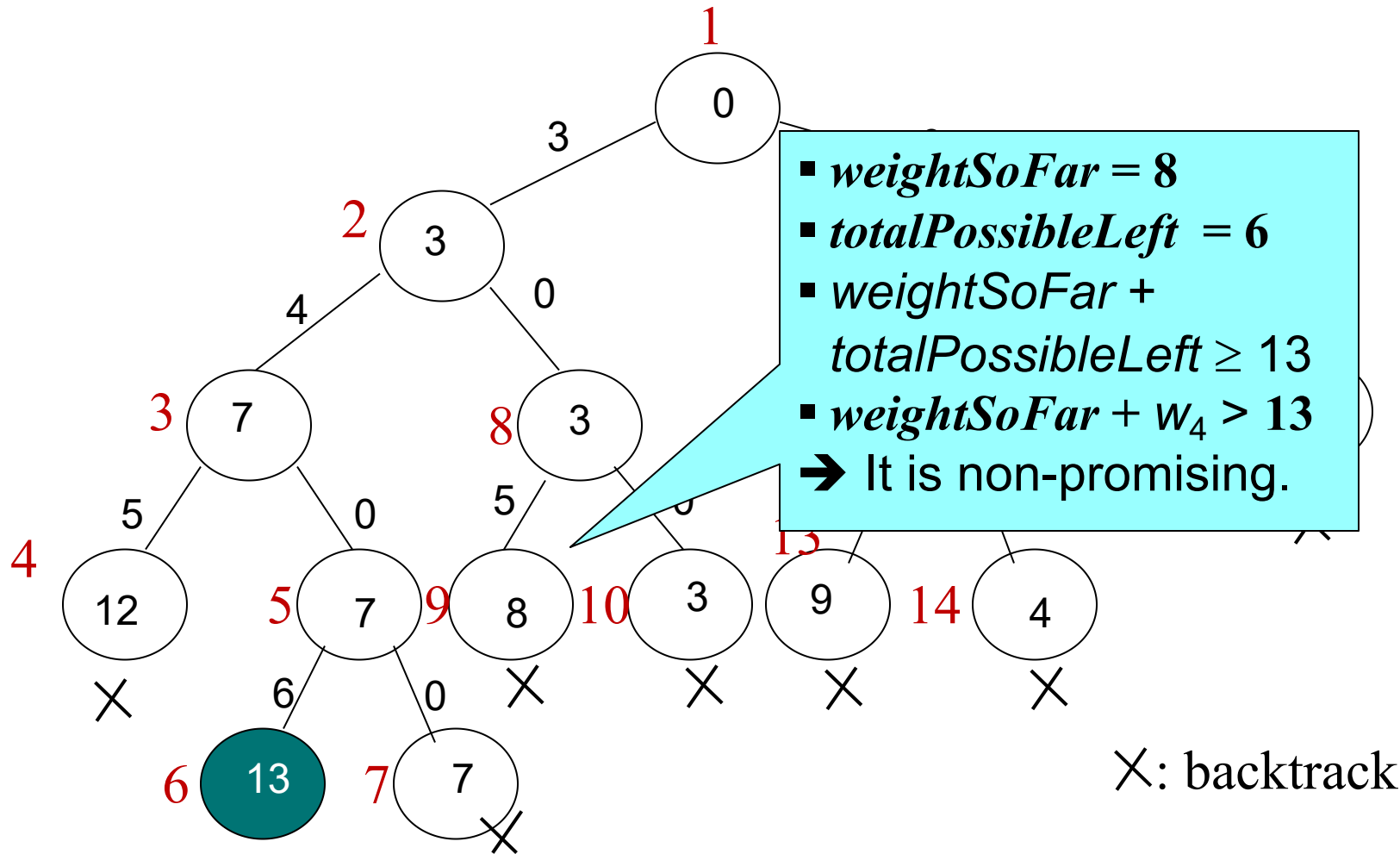
$$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$$



$$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$$


# A Pruned State Space Tree

$$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$$



Nodes numbered in “exploration” order



# 复杂度分析 (1)

---

- 输入整数的个数和 $S$ 决定了算法效率
- **Question:**  $S$ 的值如何影响算法效率?
- 对于相同输入的整数集合,  $S$ 越小, 算法越快
  - $\text{weightSoFar} + w_{i+1} > S$  容易满足.
- 对于相同输入的整数集合,  $S$ 非常大, 算法也快
  - $\text{weightSoFar} + \text{totalPossibleLeft} < S$  容易满足.

# Complexity of the Algorithm (2)

---

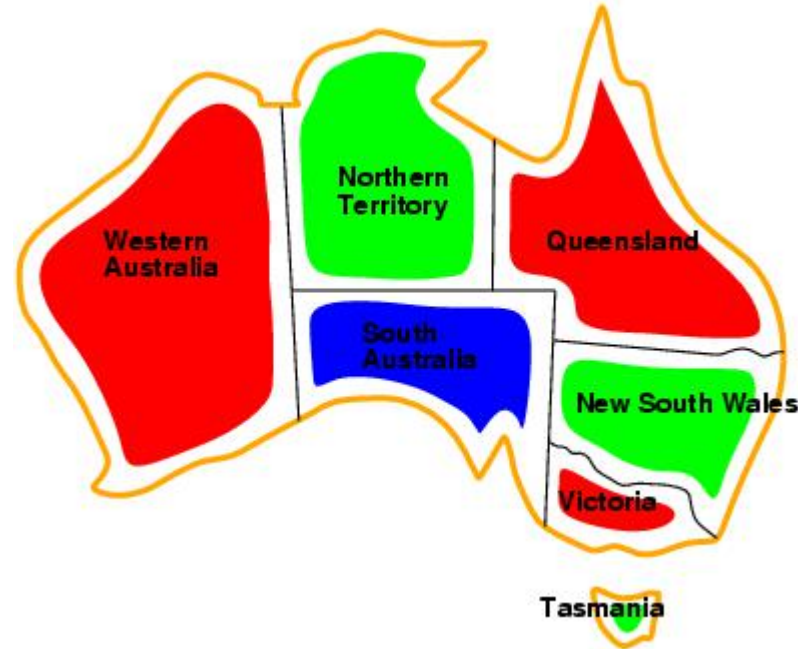
- **Question:** 算法最坏的复杂度?
  - 假设 $S$  造成了最坏的情况.
- **Answer:**  $O(2^n)$ .
- 例如:  $w_1 + \dots + w_{n-1} < S$  and  $w_n = S$ 
  - **Question:** What is the solution for this case?
  - **Answer:**  $\{w_n\}$
  - In this case, all the nodes in the state space tree before  $w_n$  is considered are promising  $\rightarrow \Theta(2^n)$  nodes will be visited.

# 地图填色问题



- 变量:  $WA, NT, Q, NSW, V, SA, T$
- 定义域:  $D_i = \{\text{red}, \text{green}, \text{blue}\}$
- 约束: 相邻区域颜色不同
  - 例如,  $WA \neq NT$ 
    - $(WA, NT)$  的状态只能在集合  $\{(\text{red}, \text{green}), (\text{red}, \text{blue}), (\text{green}, \text{red}), \dots\}$  中

# 地图填色问题



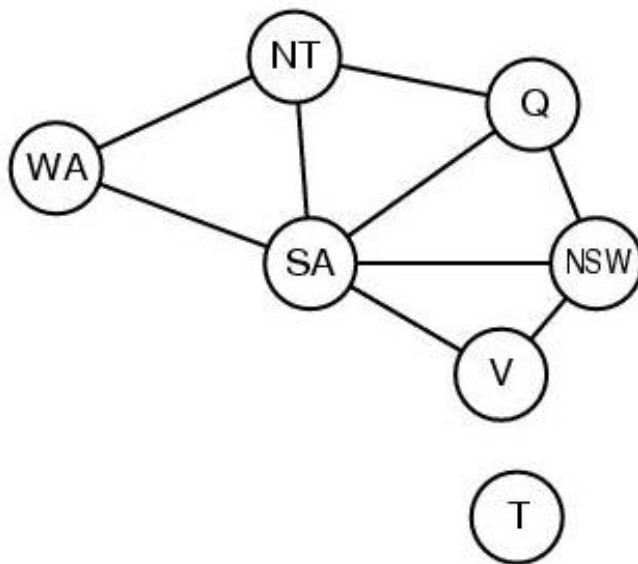
可行解:

- WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green

# 地图的抽象表示——图

---

- 一个节点是一个州，边表示邻接关系



# 问题描述与分析

---

- 解空间：
  - 7 个变量，每个变量有三种取值  $\rightarrow O(3^7)$  种解
- 节点定义： $(x_1, x_2, \dots, x_7)$ ， $x_i$ 取值范围 {red, green, blue}
- 状态树的构造：每一层是一个变量的涂色情况
- 树最多有8层（根是第0层）
- 每个叶子节点就是一种解，可能是无效解，也可能是有效解

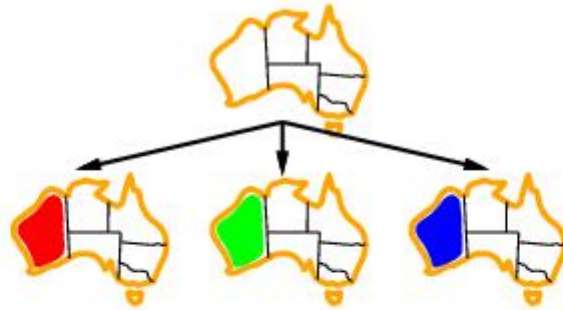
# 回溯法

---



# 回溯法

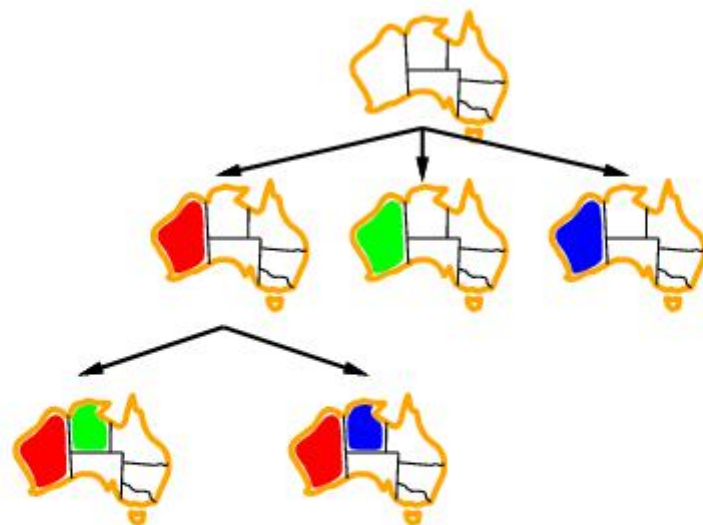
---





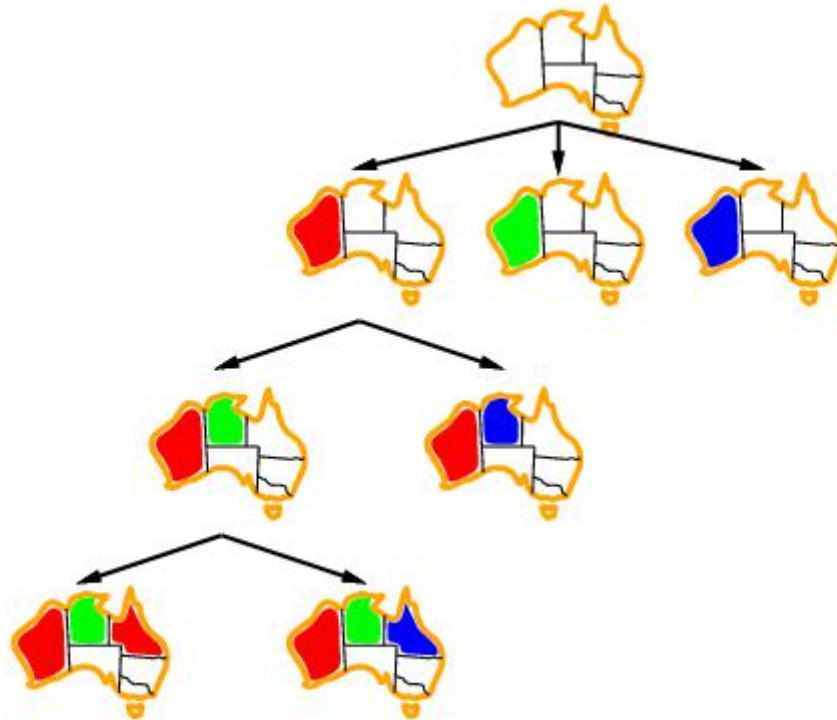
# 回溯法

---



# 回溯法

---



# 如何提高回溯效率？

---

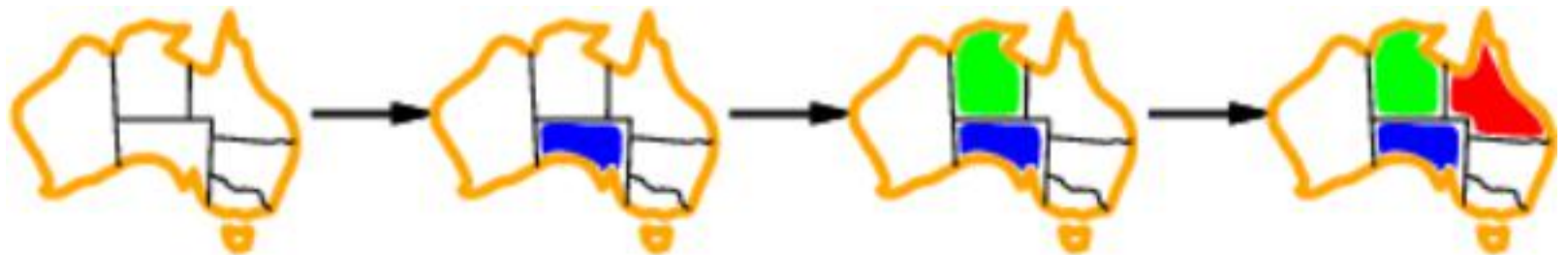
- 考虑下面三个因素：
  - 下一层选择哪个变量？（下一次给哪个州涂色？）
  - 用什么颜色顺序展开节点？
  - 如何提前探知不可行的解？

# 选择下一个需要涂色的变量

- 选择具有可填图颜色较少的变量（州）——最小剩余值准则（MRV）



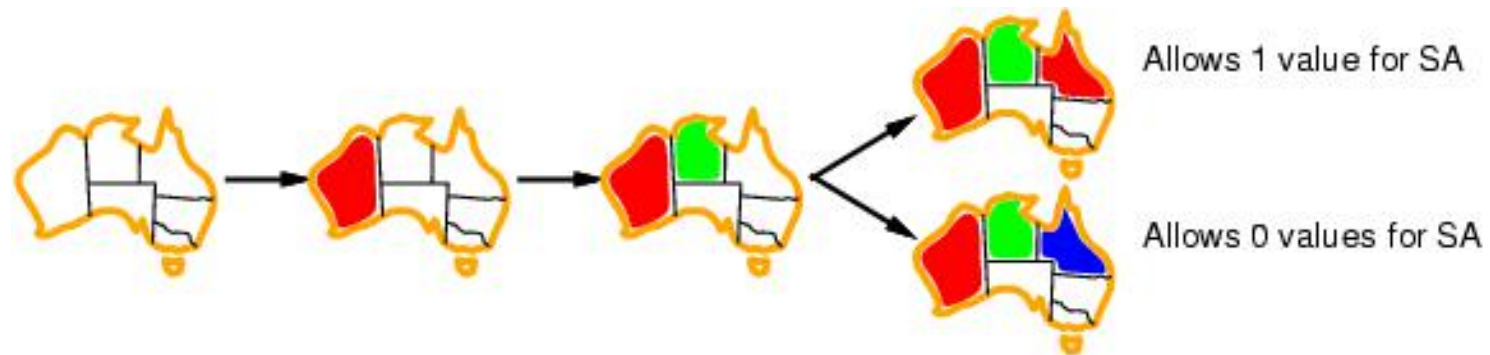
- 选择当前约束最多的变量（州）填涂（DH）



- 这两个准则的思想：容易导致失败的变量先赋值，如果这些变量注定无法赋值，<sub>44</sub>我们希望早些发现。MRV先用，DH用于tie break

# 变量值的选择

- 排除使剩下的变量可选值最小的赋值

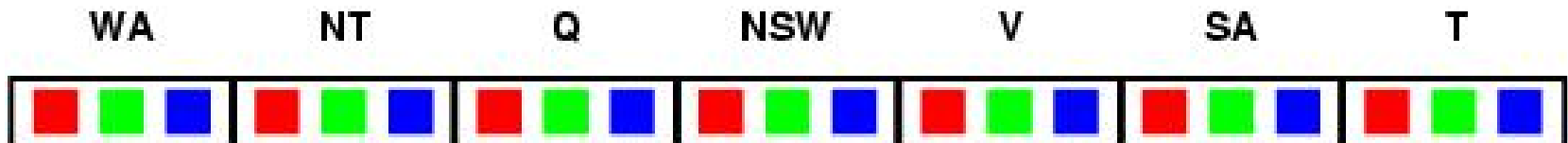


- Q选择红色，因为红色可以使SA具有最大灵活的赋值（希望能继续找到解）



# 向前探查——早点发现失败

- 思想：
  - 如果节点X已经赋值，检查所有与X相邻的节点Y，删除Y中与X赋值矛盾的赋值；
  - 如果没有变量具有合法值则回溯。



# 向前探查



# 向前探查



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>





# 界限函数

---

- 有效剪枝的一种方法
- 对状态的可行性定义一个界限（量化指标），利用该指标决定是否剪枝
- 适用于**最优问题**求解（以最大值问题为例）
- 例如：若干集装箱装船，在船装载能力有限的情况下，尽可能多的装集装箱

# 界限函数的一些定义

---

- 假设状态空间树种的一个节点对应的解是  $X = (x_1, \dots, x_{lev}, -, \dots, -)$ ,  $lev$  是该节点所处的层数
- 定义  $C(X)$  是  $X$  的后代中可行解的最大获益, 如果  $X$  本身就是可行解 ( $lev = n$ ), 则  $C(X) = X$  的获益; 如果  $X = (-, \dots, -)$  ( $lev = 0$ ), 则  $C(X) =$  问题的最佳获益

- 
- 一般情况下，只能通过遍历具有根节点X的子树来计算 $C(X)$ ，但是这样做没有意义；
  - 一个变通的做法就是定义一个界限函数 $B(X)$ ；
  - $B(X)$ 是任何可行解（该解是状态空间树中X的后代）的获益的上界；

$$B(X) \geq C(X)$$

- 如果X是可行解，则

$$B(X) = C(X)$$

# 界限函数

---

- $B(X)$ 可用于对状态树进行剪枝，当

$$B(X) \leq OPTP$$

**$OPTP$ 是搜索过程中获得的当前最优解的获益**

则  $C(X) \leq B(X) \leq OPTP.$

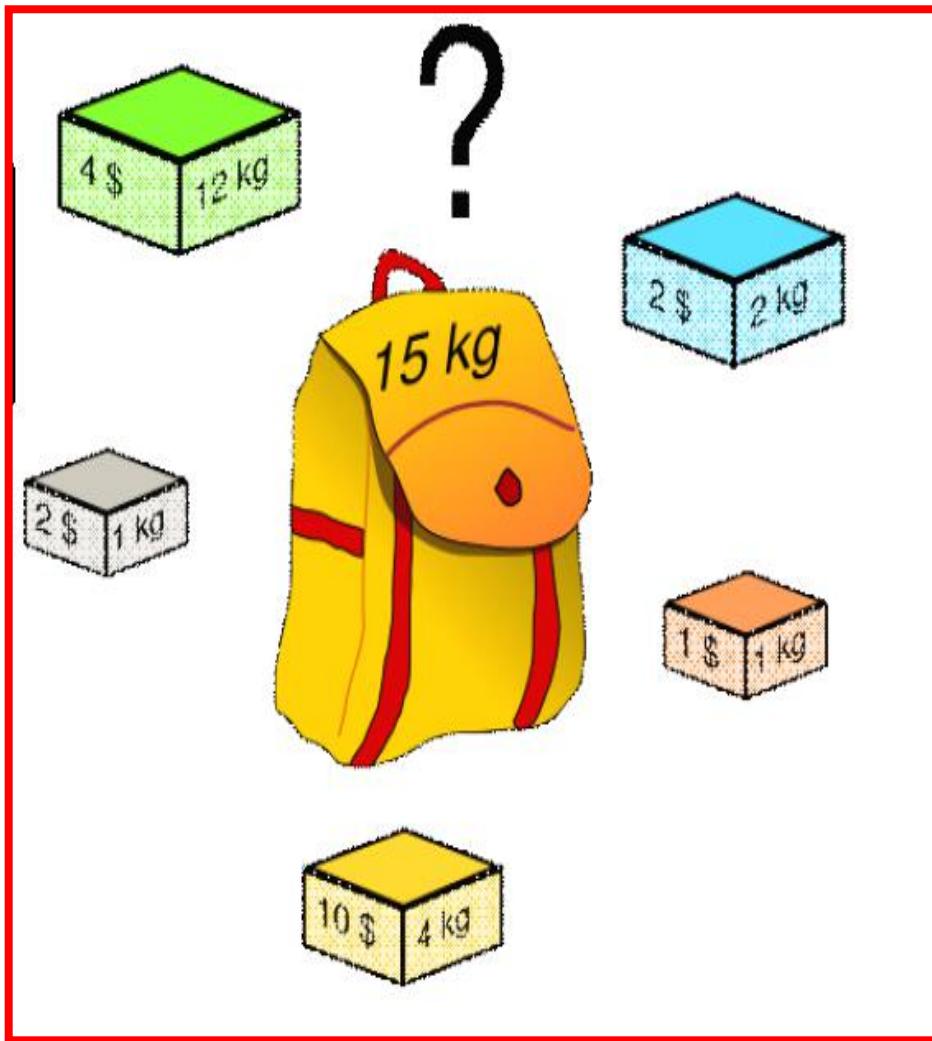
**说明X的后代都可以被剪枝，因为其获益的上界不会比当前最大获益大；**

# 对界限函数的要求

---

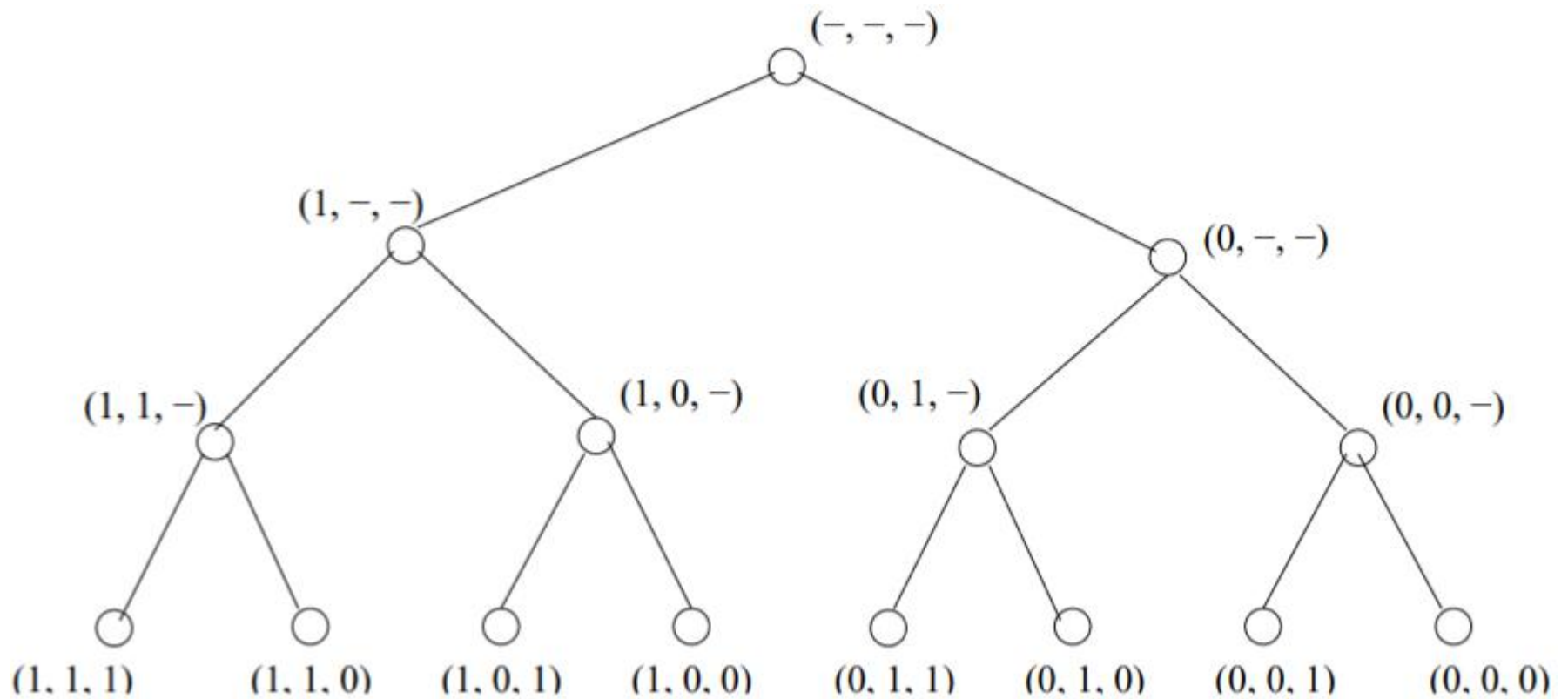
- 容易算
- 与 $C(X)$  接近
- 寻找 $B(X)$ 是很困难的事情

# 举个栗子——0-1背包问题



给定 $n$ 种物品和一个背包，物品 $i$ 的重量是 $w_i$ ，其价值为 $p_i$ ，背包的容量为 $C$ 。如何选择装入背包的物品，使得装入背包中物品的总价值最大？0/1背包问题。

# 状态树





# 0-1背包问题的界限函数

---

- 利用分数背包问题的解作为 $B(X)$ 
  - Given a (feasible) partial solution

$$X = (x_1, \dots, x_{lev}, -, \dots, -) (0 \leq lev \leq n),$$

define

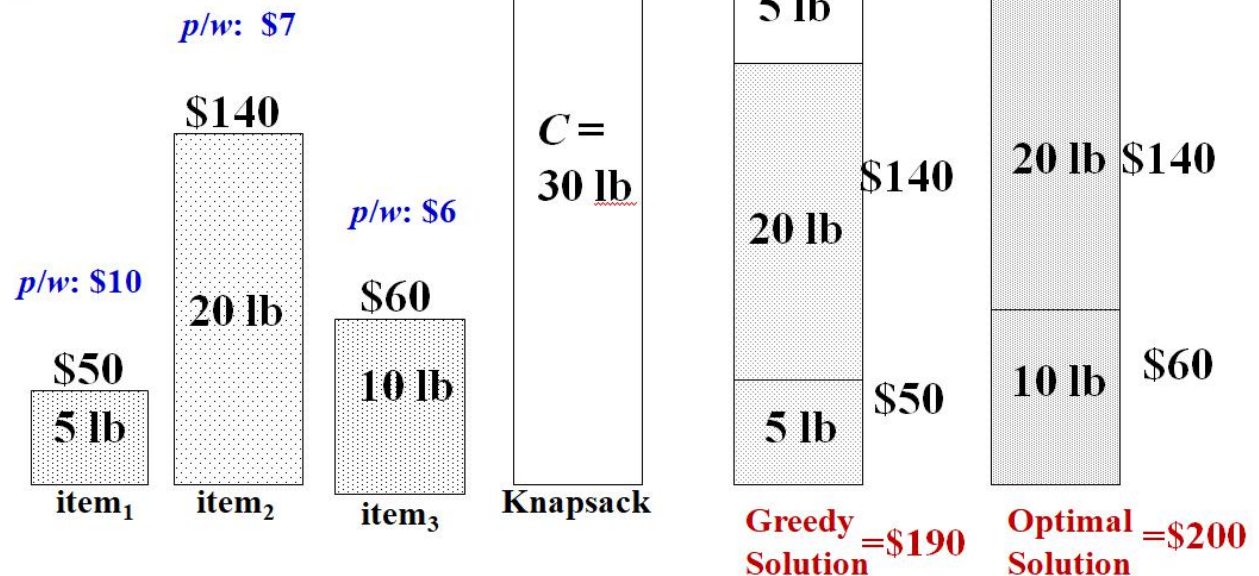
$$\begin{aligned} B(X) &= \sum_{i=1}^{lev} p_i x_i + RK(lev, M - \sum_{i=1}^{lev} w_i x_i) \\ &= \sum_{i=1}^{lev} p_i x_i + RK(lev, M - CURRW). \end{aligned}$$

- $RK(lev, \bar{M}')$  是利用物品  $lev + 1, \dots, n$ , 和剩余  $\bar{M}'$  空间得到的分数背包问题的解

# 分数背包问题求解

- 分数背包问题求解算法过程:
  - 降序排序 $p_i / w_i$
  - 根据排序次序这个增加物品，直到这个物品装完，或是超出背包容量
  - 如果背包没有满，选择下一个的物品开始装

$$p_1/w_1 \geq \dots \geq p_n/w_n$$

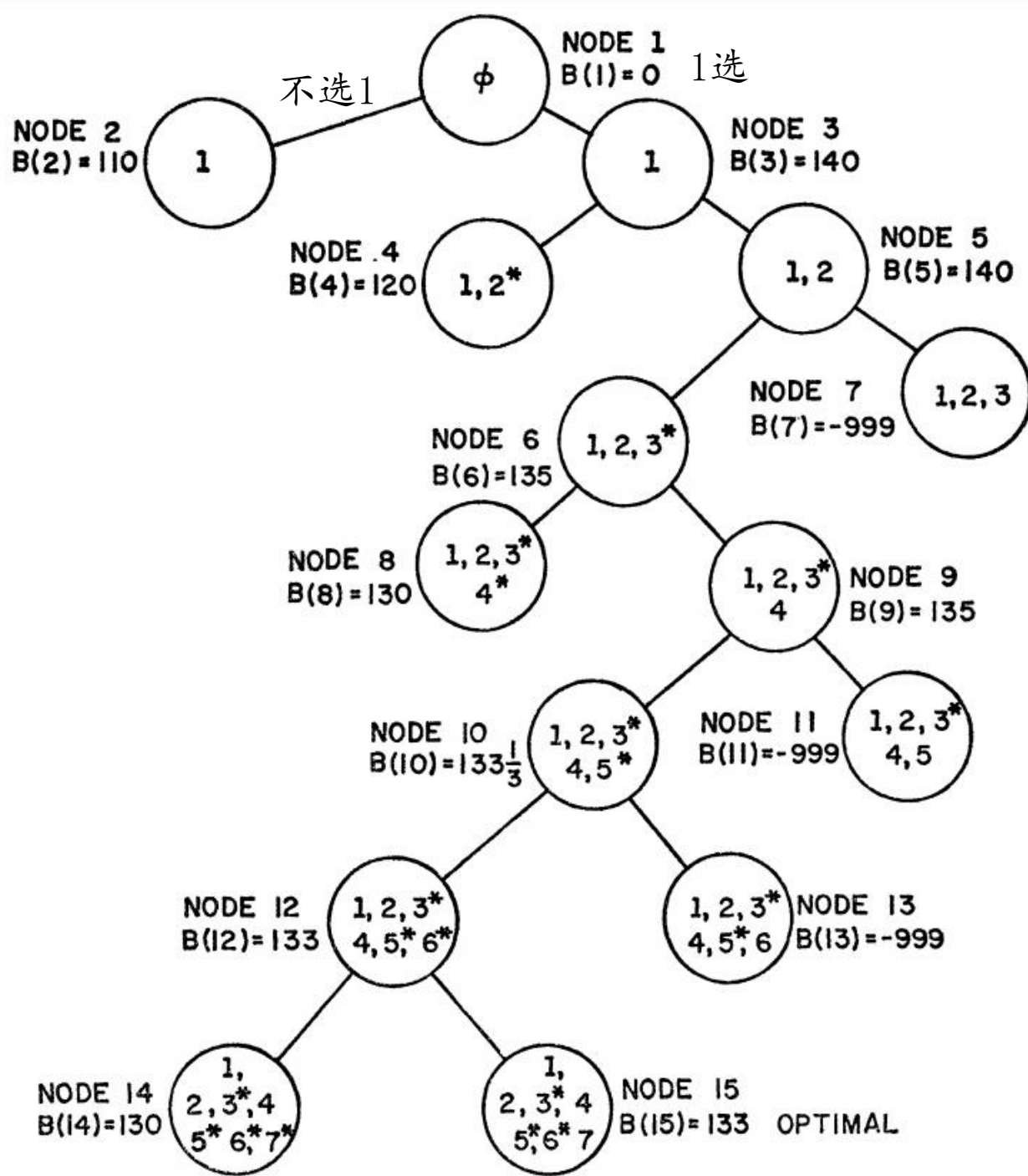


一个栗子  $W = 100.$

---

<i>Item No.</i>	<i>Weight</i>	<i>Value</i>
1	40	40
2	50	60
3	30	10
4	10	10
5	10	3
6	40	20
7	30	60

<i>New Index</i>	<i>Item No.</i>	<i>Weight</i>	<i>Value</i>	<i>Ratio</i>
1	7	30	60	2
2	2	50	60	6/5
3	1	40	40	1
4	4	10	10	1
5	6	40	20	1/2
6	3	30	10	1/3
7	5	10	3	3/10



# 装载问题

---

有一批共 $n$ 个集装箱要装上2艘载重量分别为 $c_1$ 和 $c_2$ 的轮船，其中集装箱 $i$ 的重量为 $w_i$ ，且  $\sum_{i=1}^n w_i \leq c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

例如：当 $n=3$ ,  $c_1=c_2=50$ ,

■  $w=[10, 40, 40]$ 时，则可以将集装箱1和2装到第一艘轮船上，而将集装箱3装到第二艘轮船上；

■  $w=[20, 40, 40]$ ，则无法将这3个集装箱都装上轮船。

# 装载问题

---

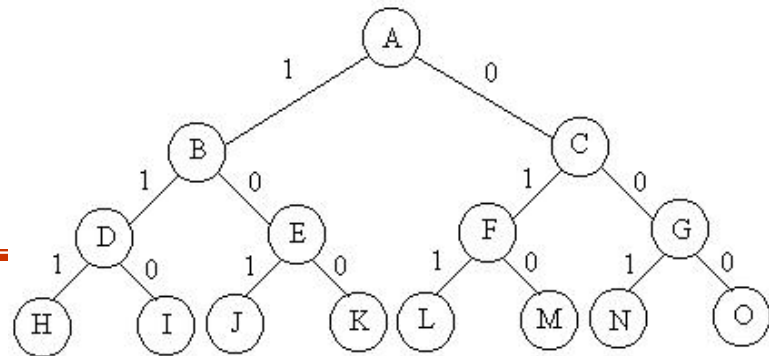
- 容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案：
  - (1) 首先将第一艘轮船尽可能装满；
  - (2) 将剩余的集装箱装上第二艘轮船。
- 将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。

$$\max \sum_{i=1}^n w_i x_i$$

$$\text{s.t. } \sum_{i=1}^n w_i x_i \leq c_1$$

$$x_i \in \{0,1\}, 1 \leq i \leq n$$

# 装载问题



- 解空间：  $(x_1, x_2, \dots, x_n)$  子集树
- 树结构： 每层表示元素  $x_i$  是否选中
- 剪枝：

(1)  $cw$  记当前的装载重量,  $cw = \sum_{i=1}^j w_i x_i$

若在于集树的第  $j+1$  层的结点  $z$  处, 则当  $cw > c_1$  时, 以结点  $z$  为根的子树中所有结点都不满足约束条件, 因而该子树中的解均为不可行解, 故可将该子树剪去。

(2)  $r$  = 剩余集装箱的重量;  $r = \sum_{i=j+1}^n w_i$

若 当前载重量  $cw + r \leq$  当前最优载重量  $bestw$ , 可将  $z$  的子树剪去

# 装载问题

---

```
void backtrack (int i)
{
    // 搜索第i层结点
    if (i > n) // 到达叶结点
        更新最优解bestx,bestw;return;
    r -= w[i];
    if (cw + w[i] <= c) { // 搜索左子树
        x[i] = 1;
        cw += w[i];
        backtrack(i + 1);
        cw -= w[i];    }
    if (cw + r > bestw) {
        x[i] = 0; // 搜索右子树
        backtrack(i + 1);    }
    r += w[i];
}
```

用回溯法设计解装载问题的  
 $O(2^n)$ 计算时间算法。在某些情  
况下该算法优于动态规划算法。