


算法设计与分析

主讲教师：王新宇



人在一生之中会面临很多重要的选择，不同的选择可能会导致不同的人生。如果人生可以量化，那如何才能做出最正确的选择，使得自己的人生“最优”呢？

有人说，我们可以使用贪心法，每次都做出当前看起来最优的选择，期望这组选择可以使得人生达到“最优”。但是，贪心法未必能得到全局最优解，那有没有其他办法呢？

在面临人生岔路的时候，如果能够在选择一个方向进行试探，发现这个方向走不通（不符合期望）的时候返回到岔路重新选择新的方向进行试探，那么通过这样反复的试探必然能够找到使得人生最优的选择。

回溯法有“通用解题法”之称，它用一种试探性的行为沿着某一方向搜索问题的解，在搜索方向无法继续时，可以返回尝试搜索别的方向，直至搜索完所有可能的方向。

因此，回溯法可以找到问题的所有解，并通过对比找到最优解。形象地说，回溯法是一种可以不断重来的方法。

但是，人生无法重来，我们在面对困难和挫折时不能气馁和逃避，很多事情不是一蹴而就的，要允许失误甚至失败，只要能够从中汲取经验和力量，不断地尝试，终会迎来光明的未来。



第12章 回溯法

目 录

12.1 回溯法概述

12.2 回溯法的应用实例

12.3 小结



12.1 回溯法概述

12.1.1 问题的解空间

(1) **问题的解向量**：如果一个复杂问题的求解过程可以被划分为 n 个依次进行的决策阶段，那么该问题的解可以表示为一个向量 $X=(x_1, x_2, \dots, x_n)$ ，其中， x_i 表示第 i 个决策阶段的选择，向量 X 称为问题的解向量，**对应问题的一种状态**。

(2) **显约束**：对于解向量 $X=(x_1, x_2, \dots, x_n)$ ，分量 x_i 的取值范围称为显式约束条件（显约束）。

显约束仅限定了解向量中分量的取值范围，并未限定分量之间应当满足的关系，故**仅仅满足显约束的解向量未必满足问题的要求，即未必是满足问题要求的解**。

12.1.1 问题的解空间

(3) **隐约束**：为满足问题的要求，对解向量的不同分量施加的约束条件称为隐式约束条件（隐约束）。**隐约束是对分量之间关系的约束**，解向量在满足显约束的前提下，通常需要满足隐约束才能满足问题对解的要求。

(4) **解空间**：对于问题的一个实例，所有**满足显约束**的解向量构成该问题实例的解空间，也称为**状态空间**。

(5) **可行解**：解空间中**满足隐约束**的解向量称为问题的可行解。

(6) **最优解**：解空间中使得问题的**目标函数取得最优值的可行解**称为问题的最优解。

一个最优化问题的求解过程就是在问题的解空间中寻找可行解，以及进一步在可行解中寻找最优解的过程。

12.1.1 问题的解空间

例：一个0-1背包问题，物品种类 $n=3$ ，物品的质量 $w=\{6, 7, 5\}$ ，价值 $v=\{9, 6, 11\}$ ，背包的容量 $c=12$ 。

解向量： $X=(x_1, x_2, x_3)$

显约束： $x_i \in \{0, 1\}, i=1, 2, 3$ 。

隐约束： $\sum_{i=1}^3 w_i x_i \leq 12$

解空间： $\{ (0, 0, 0), (0, 0, 1), (0, 1, 0), (0, 1, 1), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1) \}$

可行解： $(0, 0, 0)$ 、 $(0, 0, 1)$ 、 $(0, 1, 0)$ 、 $(0, 1, 1)$ 、 $(1, 0, 0)$ 、 $(1, 0, 1)$

最优解： $(1, 0, 1)$

12.1.1 问题的解空间

(7) **解空间树**：为了便于在解空间中搜索可行解和最优解，通常将问题的解空间组织成树的形式，称为解空间树。

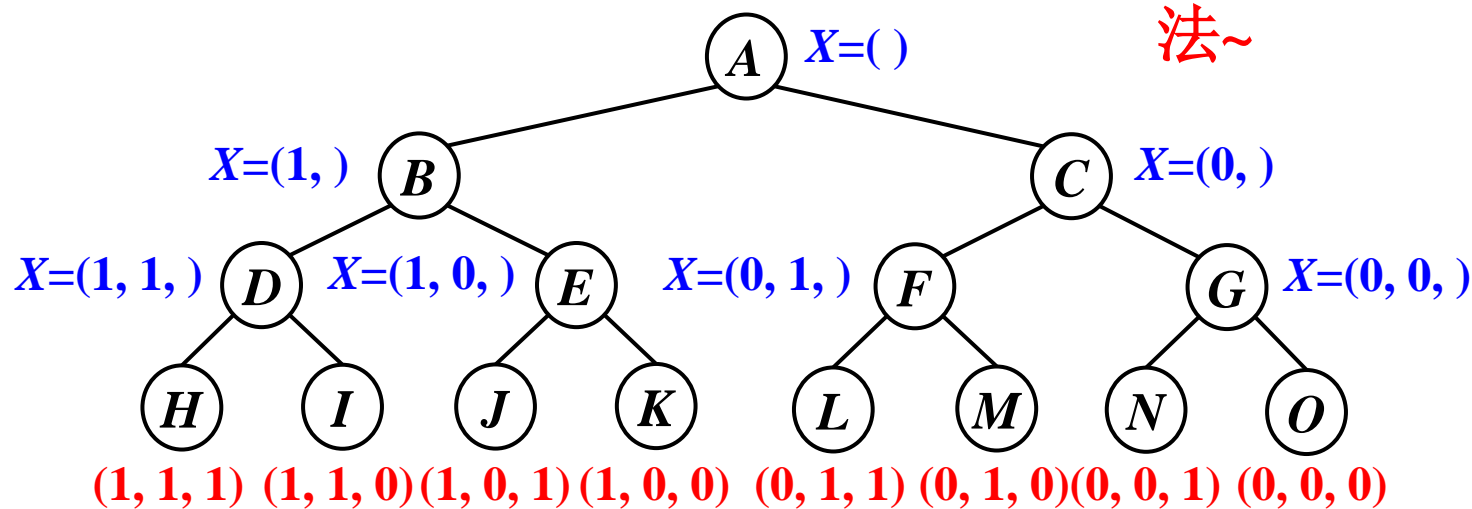
在解空间树中，**节点**代表**解向量的当前状态**，**分支**代表**决策阶段的选择**。根节点代表解向量的初始状态（空向量），叶子节点代表一个完整的解，其余节点代表某种选择下的一种解向量状态。

在问题的解空间中搜索可行解和最优解的过程被转化为在解空间树中搜索叶子节点的过程。

12.1.1 问题的解空间

例：物品种类 $n=3$ 的0-1背包问题的解空间树。

解空间树体现了问题解的一种构造方法~



根节点A代表解向量初始状态 $X=()$ ，处于第1层。

第2层~第4层表示求解问题的3个阶段，每个阶段对1种物品做出决策（装入背包或不装入背包），形成2个分支，左分支表示装入背包（用1表示），右分支表示不装入背包（用0表示）。

叶子节点代表问题的一个完整解，处于第4层。

12.1.2 回溯法的基本思想

➤ 基本概念和术语

- (1) **扩展节点**：一个正在产生孩子的节点。
- (2) **活节点**：一个自身已生成但其孩子还没有全部生成的节点。
- (3) **死节点**：一个已经产生所有孩子的节点。
- (4) **深度优先的问题状态生成法**：对一个扩展节点 F ，一旦产生它的一个孩子 C ，就把 C 作为新的扩展节点；在完成对子树 C （以 C 为根的子树）的穷尽搜索之后，将 F 重新变成扩展节点，继续产生 F 的其他孩子（如果存在）。
- (5) **广度优先的问题状态生成法**：对一个扩展节点 F ，一次性产生它的所有孩子，随后 F 成为死节点，即一个扩展节点在变成死节点之前一直是扩展节点，在成为死节点之后不会再成为扩展节点。

回溯法

12.1.2 回溯法的基本思想

➤ 回溯法的基本思想

回溯法从根节点出发，以深度优先的方法搜索整个解空间树。

首先，根节点成为活节点，同时也成为当前扩展节点。

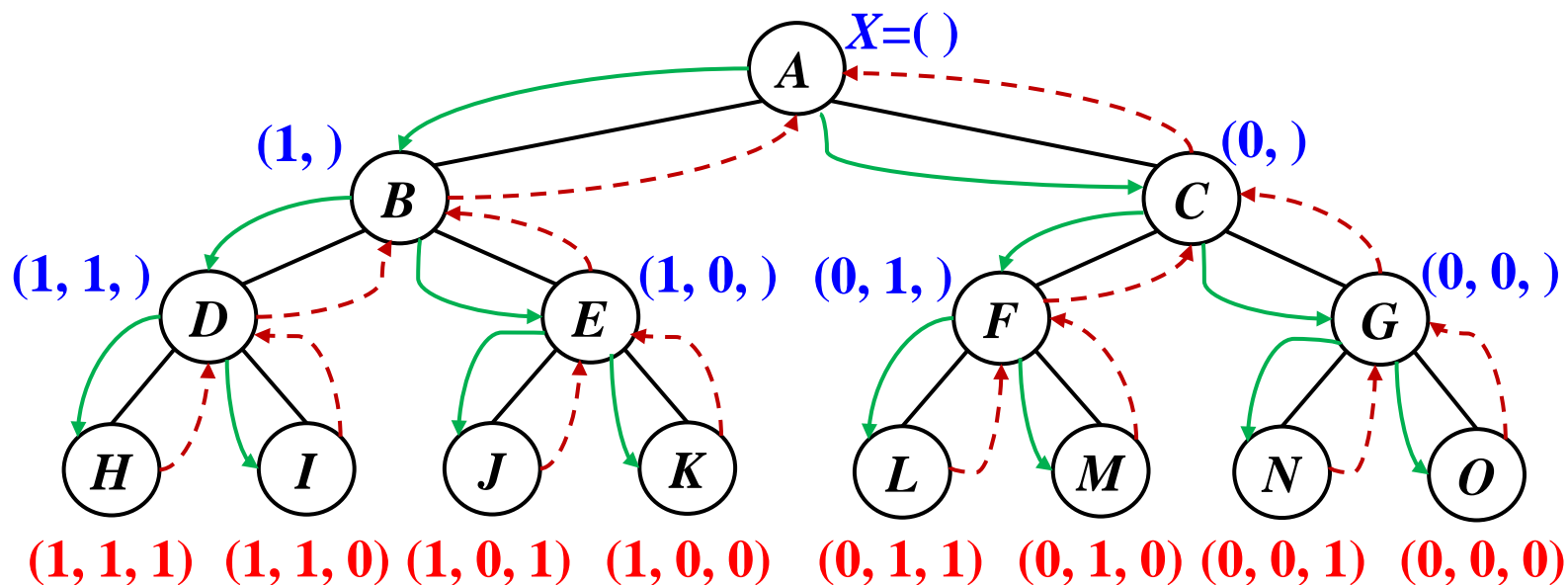
在当前扩展节点处，搜索向**纵深方向**移至一个新节点（做出一个决策），这个新节点成为新的活节点，并成为当前扩展节点。如果在当前扩展节点处不能再向纵深方向移动，则当前扩展节点成为死节点，此时，应**往回移动**（**回溯**）至最近的活节点处，并使这个活节点成为当前扩展节点。

回溯法以这种工作方式递归地在解空间中搜索，直到找到所要求的解或解空间中已无活节点时为止。

回溯法的搜索过程包括**扩展**和**回溯**两个操作。

12.1.2 回溯法的基本思想

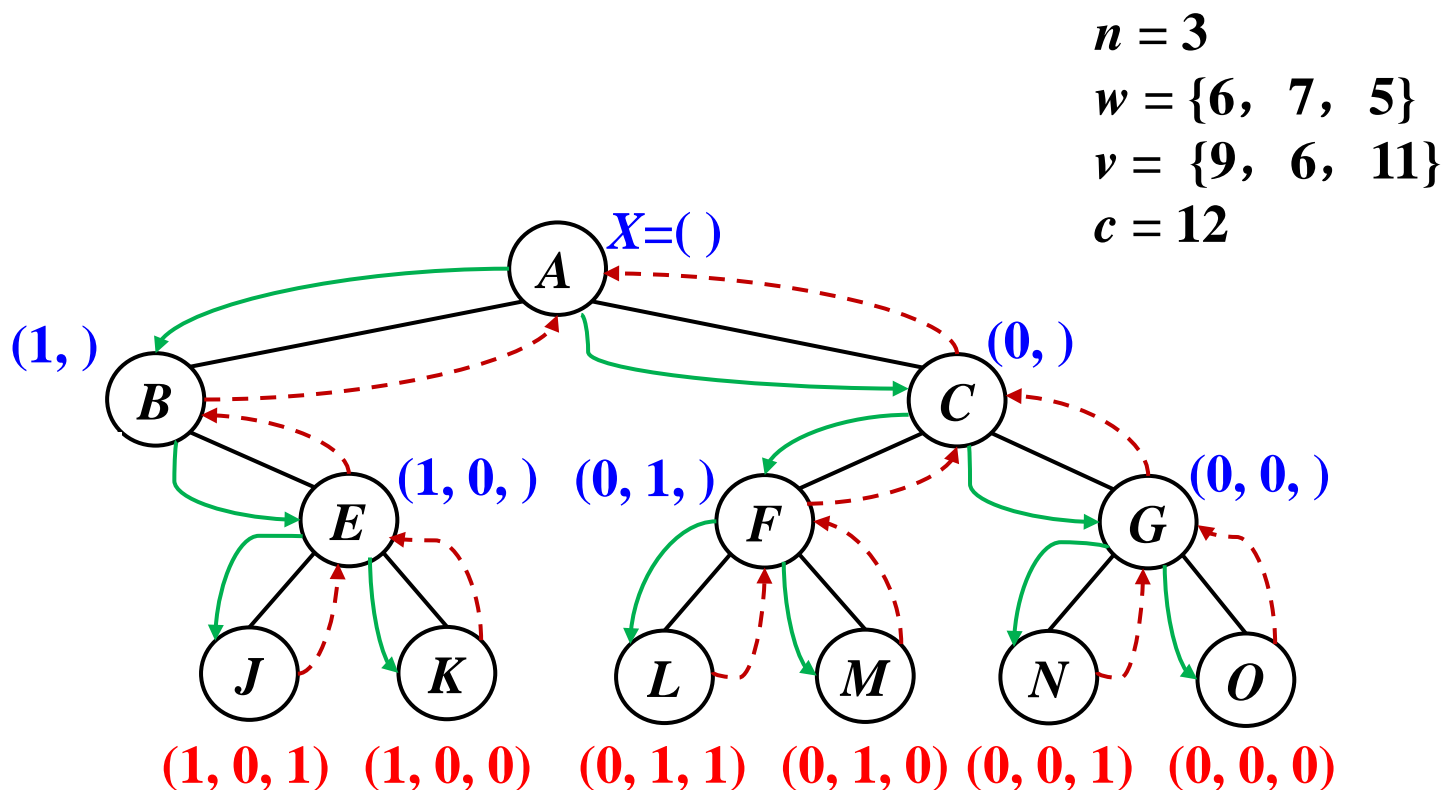
$n = 3$
 $w = \{6, 7, 5\}$
 $v = \{9, 6, 11\}$
 $c = 12$



12.1.2 回溯法的基本思想

➤ 两种无效搜索

(1) 当前的搜索方向无法产生问题的可行解。



12.1.2 回溯法的基本思想

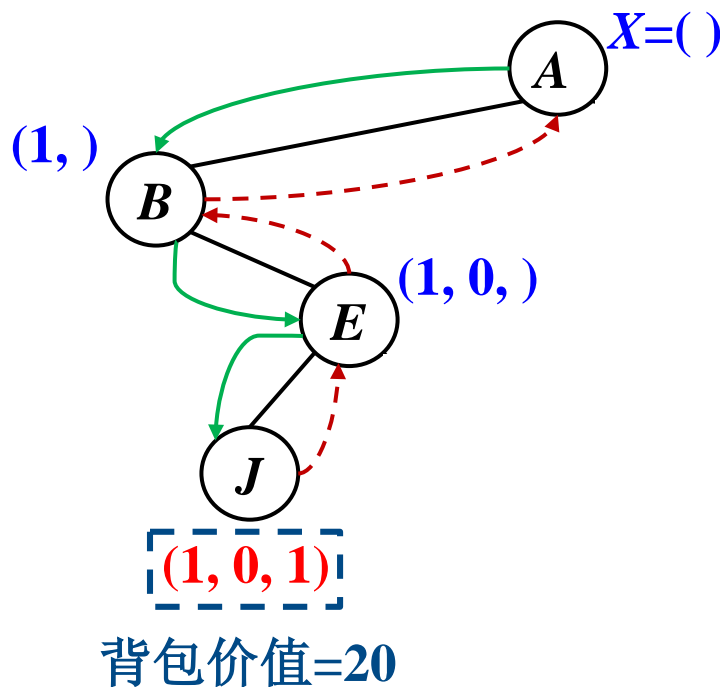
(2) 问题需要寻找最优解，而当前的搜索方向无法产生问题的最优解。

$$n = 3$$

$$w = \{6, 7, 5\}$$

$$v = \{9, 6, 11\}$$

$$c = 12$$



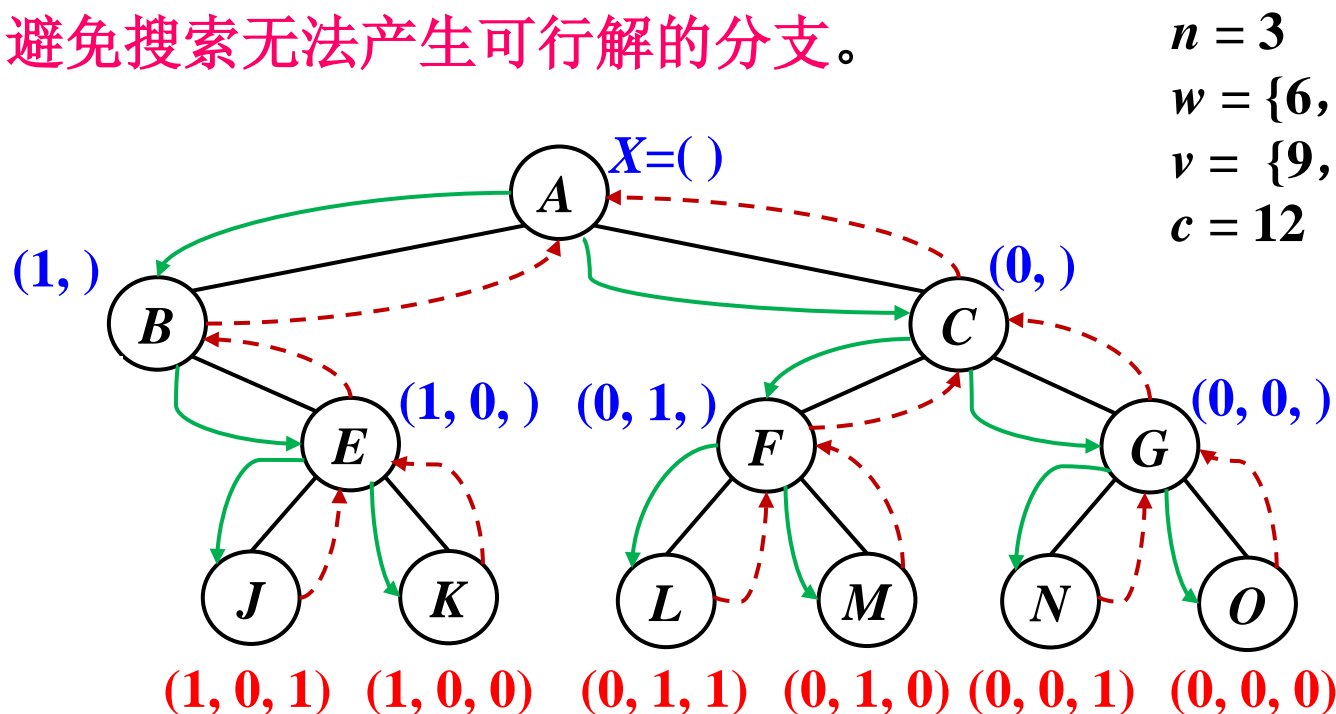
12.1.2 回溯法的基本思想

➤ 解空间树的剪枝

剪枝函数包括约束函数和限界函数。

(1) **约束函数**：判断节点的扩展是否满足约束条件的函数。

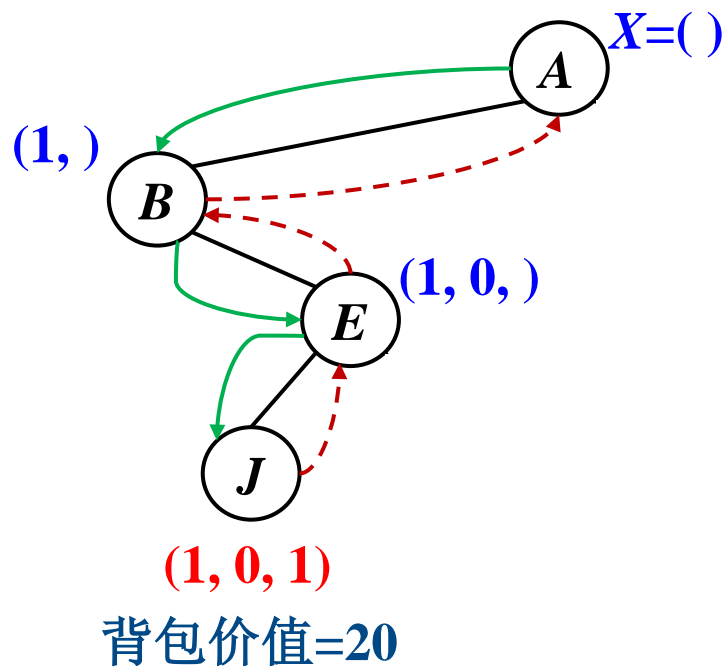
根据**问题的隐约束**构造约束函数，在扩展节点处剪去不满足约束条件的子树，**避免搜索无法产生可行解的分支**。



12.1.2 回溯法的基本思想

(2) **限界函数**：判断节点的扩展是否包含更优解的函数。

通过计算扩展节点处目标函数的上界或下界，将之与当前最优值进行比较来构造限界函数，在扩展节点处剪去得不到更优解的子树，避免搜索无法产生更优解的分支。



$$n = 3$$

$$w = \{6, 7, 5\}$$

$$v = \{9, 6, 11\}$$

$$c = 12$$

12.1.2 回溯法的基本思想

➤ 什么是回溯法

回溯法 = 深度优先搜索 + 剪枝

➤ 回溯法的求解目标

回溯法通常的求解目标是寻找到问题的所有可行解，在此基础上可以求解任一解或最优解。

在使用回溯法求解问题的所有解时，回溯过程需要回溯到根节点，且需要搜索根节点的所有可行的子树，剪枝函数只需使用约束函数。

12.1.2 回溯法的基本思想

在使用回溯法求问题的任一解时，只要搜索到问题的一个解就可以结束，剪枝函数也只需使用约束函数。

使用回溯法求最优解时，需要回溯到根节点，且需要搜索根节点的所有可行的子树，剪枝函数需要同时使用约束函数和限界函数。

12.1.3 回溯法的设计步骤与算法框架

➤ 回溯法的设计步骤

- (1) 确定问题的解空间，包括确定解向量的形式、显约束和隐约束。
- (2) 确定节点的扩展规则。
- (3) 确定剪枝函数。结合节点扩展规则、约束条件和目标函数的优化目标确定剪枝函数。不同分支的剪枝函数可能不同。
- (4) 以深度优先的方式搜索解空间树，在搜索过程中采用剪枝函数避免无效搜索。

12.1.3 回溯法的设计步骤与算法框架

深度优先搜索中的回溯方式包括：递归回溯和迭代（非递归）回溯。

➤ 递归回溯的算法框架

```
L1: void backtrack(int t) {  
L2:     if(t > n) output(x);  
L3:     else  
L4:         for(int i = f(n, t); i <= g(n, t); i++) {  
L5:             x[t] = h(i);  
L6:             if(constraint(t) && bound(t))  
L7:                 backtrack(t + 1);  
L8:         }  
L9: }
```

t: 当前递归深度，对应解空间树的第t层（根为第1层）

n: 最大递归深度，等于叶子节点的层次数-1

f(n,t): 当前扩展节点处未搜索子树的起始编号

g(n,t): 当前扩展节点处未搜索子树的终止编号

h(i): 决策为i时，解的分量x[t]的取值

constraint(t): 约束函数

bound(t): 限界函数

12.1.3 回溯法的设计步骤与算法框架

➤ 迭代（非递归）回溯的算法框架

```
L1: void iterativeBacktrack() {  
L2:     int t = 1;  
L3:     while(t > 0) {  
L4:         if(ExistSubNode(t))  
L5:             for(int i = f(n, t); i <= g(n, t); i++) {  
L6:                 x[t] = h(i);  
L7:                 if(constraint(t) && bound(t)) {  
L8:                     if(solution(t)) output(x);  
L9:                     else { t++; break; }  
L10:                }  
L11:            }  
L12:        else t--;  
L13:    }  
L14: }
```

t: 节点层次（根为第1层）

ExistSubNode(t): 判断当前扩展节点是否存在孩子

f(n,t): 当前扩展节点处未搜索子树的起始编号

g(n,t): 当前扩展节点处未搜索子树的终止编号

h(i): 决策为i时，解的分量x[t]的取值

constraint(t): 约束函数

bound(t): 限界函数

solution(t): 判断在当前扩展节点处是否已经得到问题的可行解

12.1.3 回溯法的设计步骤与算法框架

说明：（1）算法框架仅反映了设计思路！

（2）递归回溯的算法框架更加简洁~

在回溯法的求解过程中，算法始终只保存从解空间树的根节点至当前扩展节点这一条决策路径产生的解向量。

若解空间树的根节点到叶子节点的最长路径长度是 $d(n)$ ，则回溯法的空间复杂度是 $O(d(n))$ 。

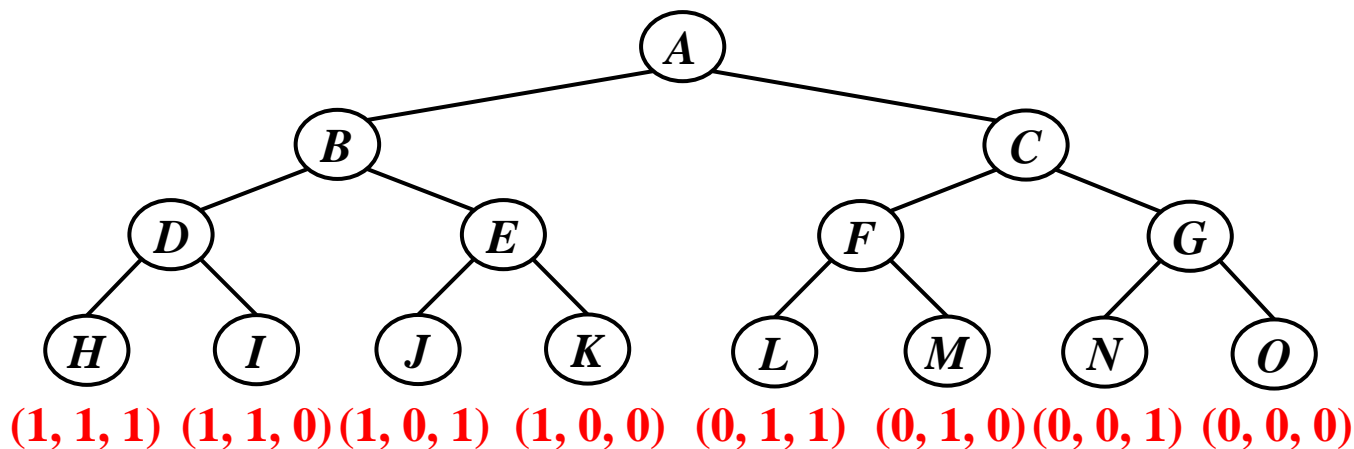
12.1.4 子集树与排列树

回溯法解题时两类典型的解空间树：子集树与排列树。

➤子集树

给定问题是从 n 个元素的集合 S 中寻找满足某种条件的子集 $S' \subseteq S$ ，则该问题的解空间树称为子集树。此类问题的解包含的元素数目通常小于或等于 n 。

例：物品种类为3的0-1背包问题。



12.1.4 子集树与排列树

◆ 回溯法搜索子集树的递归算法框架

```
void backtrack (int t) {  
    if(t > n) output(x);  
    else  
        for(int i = 下界; i <= 上界; i++) {  
            x[t]=i;  
            ...                //其他操作  
            if(constraint(t) && bound(t))  
                backtrack(t+1);  
        }  
}
```

t: 当前递归深度，对应子集树的第t层（根为第1层）

n: 最大递归深度，等于叶子节点的层次数-1

[下界, 上界]: 决策变量i的取值范围

constraint(t): 约束函数

bound(t): 限界函数

12.1.4 子集树与排列树

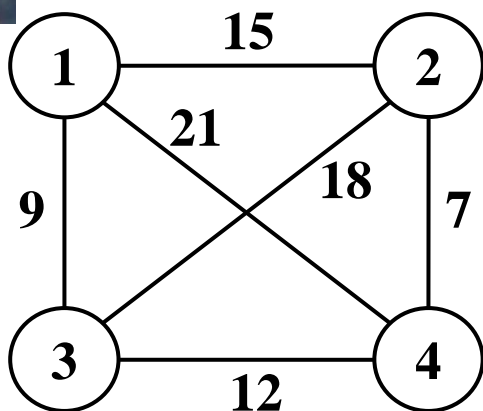
➤ 排列树

给定问题是求解 n 个元素满足某种条件的排列，则该问题的解空间树称为排列树。此类问题的解包含的元素数目总是 n 个，不同的解的区别在于元素排列顺序不同。

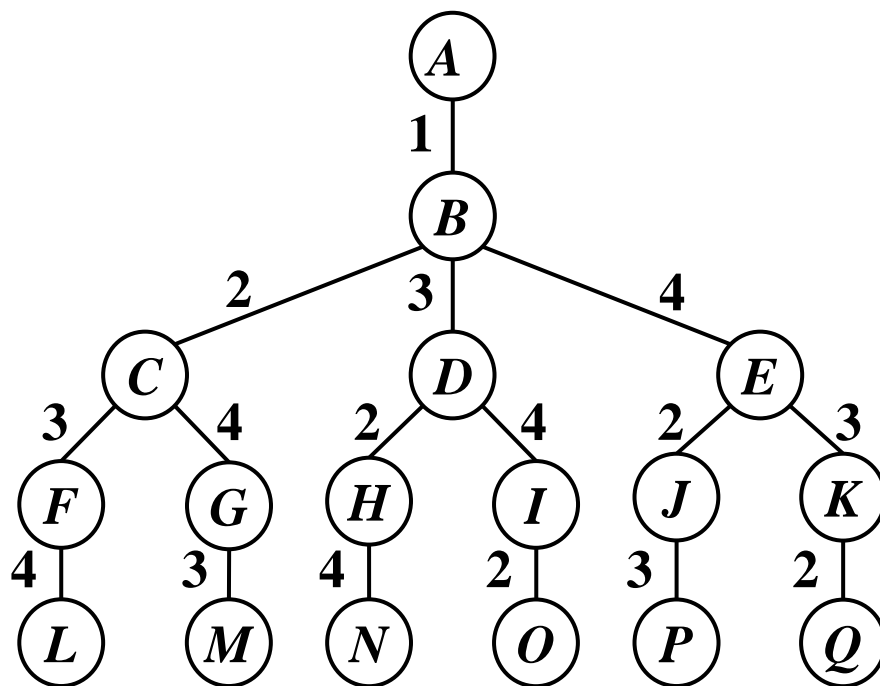
例如，旅行商问题（Traveling Salesman Problem, TSP）

一个商品推销员要去若干个城市推销商品，已知城市之间的路程（或旅费）。该推销员从一个城市出发，途径每个城市一次，最后回到出发城市。请问推销员应该如何选择行进路线，才能使总的路程（或旅费）最短（或最少）。

12.1.4 子集树与排列树



哈密尔顿回路



解向量: $X=(x_1=1, x_2, x_3, x_4)$, 其中, x_2, x_3, x_4 是 $\{2, 3, 4\}$ 的一个排列。

解空间: $\{ (1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4), (1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2) \}$

12.1.4 子集树与排列树

◆ 回溯法搜索排列树的递归算法框架

```
void backtrack (int t) {  
    if(t > n) output(x);  
    else  
        for(int i = t; i <= n; i++) {  
            ...  
            swap(x[t], x[i]); //交换形成不同排列  
            if(constraint(t) && bound(t))  
                backtrack(t + 1);  
            swap(x[t], x[i]); //恢复排列  
            ....  
        }  
}
```

在调用**backtrack(1)**执行回溯搜索之前，需要将**x**初始化为一个排列！

t: 当前递归深度，对应排列树的第**t**层（根为第1层）

n: 最大递归深度，等于叶子节点的层次数-1

[t, n]: 决策变量**i**的取值范围

swap(x[t], x[i]): 交换**x[t]**和**x[i]**

constraint(t): 约束函数

bount(t): 限界函数

12.1.5 回溯法的适用条件

(1) 问题的求解过程可以划分为若干个依次进行的决策阶段，每个阶段对解的一个分量做出选择，整个问题的解可以表示为一个向量。

(2) 问题要满足多米诺性质。

多米诺性质：设 $P(x_1, x_2, \dots, x_i)$ 为真表示向量 (x_1, x_2, \dots, x_i) 满足某个条件，那么， $P(x_1, x_2, \dots, x_{k+1})$ 为真蕴涵 $P(x_1, x_2, \dots, x_k)$ 为真，即

$$P(x_1, x_2, \dots, x_{k+1}) \longrightarrow P(x_1, x_2, \dots, x_k), \quad 0 < k < n$$

12.1.5 回溯法的适用条件

例：已知不等式 $5x_1+4x_2-x_3\leq 10$ ，约束条件为 $1\leq x_i\leq 3(i=1, 2, 3)$ ，求满足不等式的所有整数解。

当解向量 (x_1, x_2, x_3) 满足 $5x_1+4x_2-x_3\leq 10$ 时， (x_1, x_2) 未必满足 $5x_1+4x_2\leq 10$ ，故问题不满足多米诺性质。

使用回溯法求解该问题，在搜索到解向量 $(1, 2)$ 时，由于不满足隐约束被剪枝，从而导致丢失一个可行解 $(1, 2, 3)$ 。



12.2 回溯法的应用实例

12.2.1 0-1背包问题

设给定 n 种物品和一个背包，物品 i 的质量是 w_i ，其价值为 v_i ，背包的容量为 c 。

对每种物品 i 只有两种选择：装入背包或不装入背包，既不能将物品 i 装入背包多次，也不能只装入部分的物品 i 。

应当如何选择装入背包的物品，在物品质量不超过背包容量的情况下，使得装入物品后的背包价值最大？

12.2.1 0-1背包问题

【问题分析】

(1) 确定问题的解空间

设 $X=(x_1, x_2, \dots, x_n)$ 表示问题的解向量，其中， x_i ($i=1, 2, \dots, n$) 表示对第 i 种物品的决策。

显约束： $x_i \in \{0, 1\}$ ($i=1, 2, \dots, n$)， $x_i=0$ 表示第 i 种物品不装入背包， $x_i=1$ 表示第 i 种物品装入背包。

隐约束： $\sum_{i=1}^n w_i x_i \leq c$

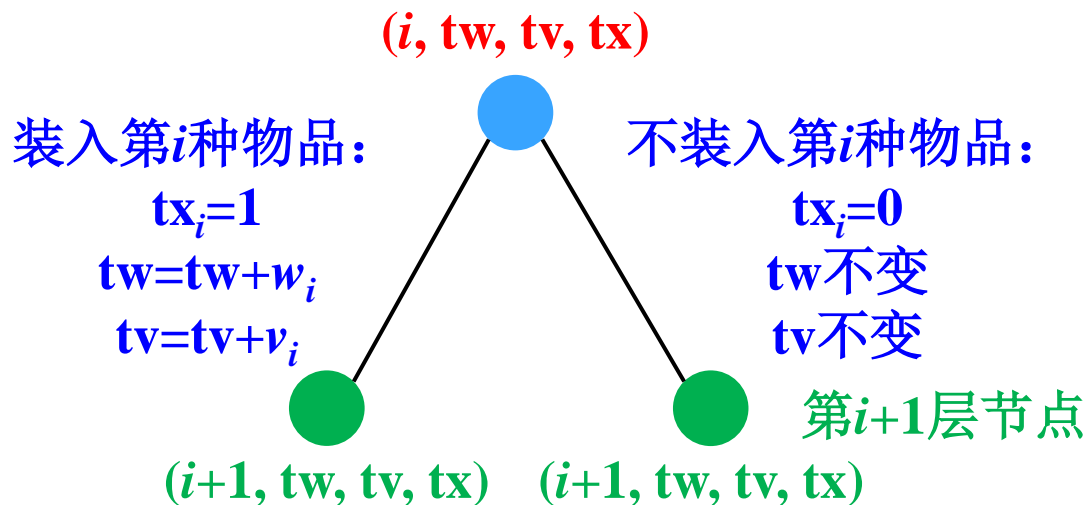
解空间树类型：子集树。

12.2.1 0-1背包问题

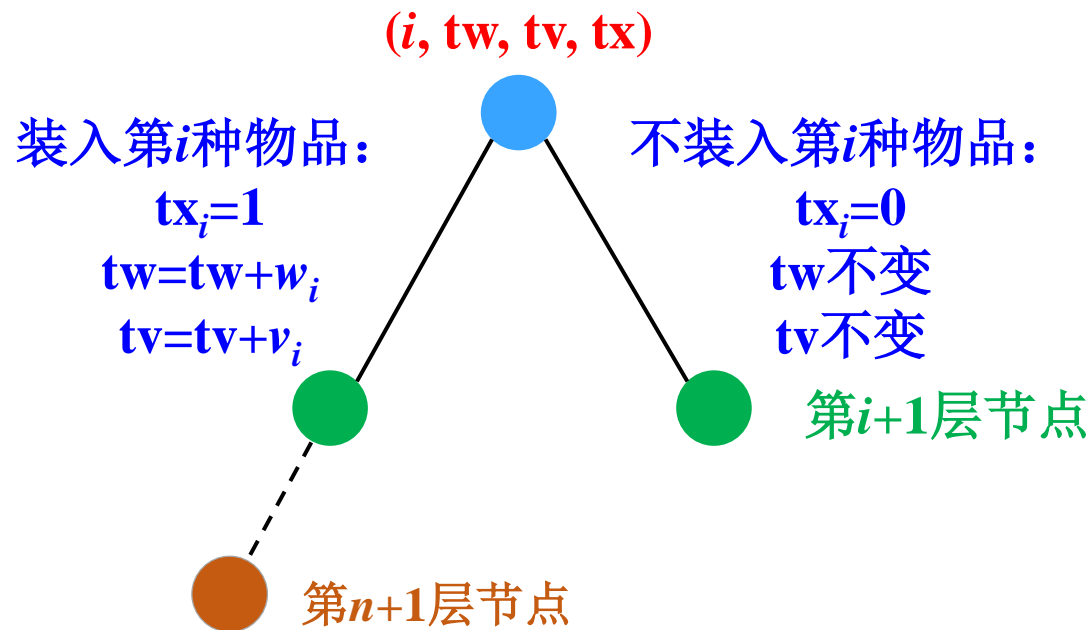
(2) 确定节点的扩展规则

为解空间树的每个节点设置状态: (i, tw, tv, tx) , 其中, i 表示节点的层次, tw 表示当前的背包质量, tv 表示当前的背包价值, tx 记录当前的解向量。

第 i 层节点的扩展规则: 装入或不装入第 i 种物品。



12.2.1 0-1背包问题



■ 叶子结点表示已经对 n 种物品做出了决策。

12.2.1 0-1背包问题

(3) 确定剪枝函数

结合节点扩展规则、约束条件和目标函数的优化目标确定剪枝函数

① 左剪枝

左分支的扩展规则：装入物品。

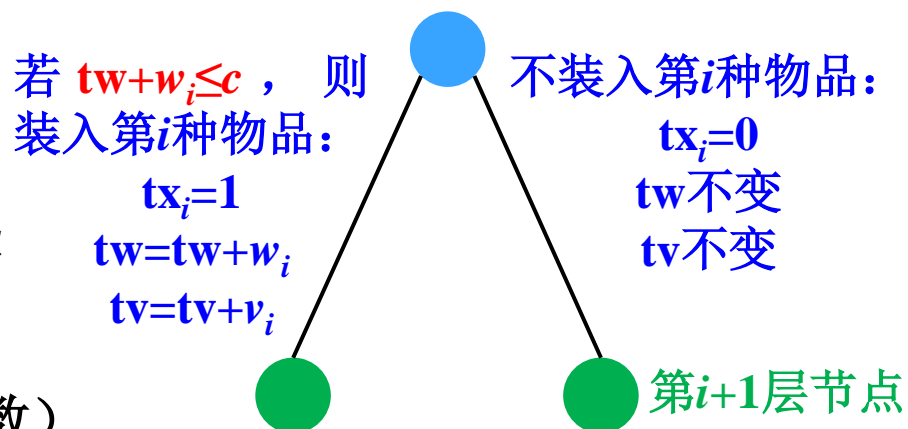
约束条件： $\sum_{i=1}^n w_i x_i \leq c$

目标函数的优化目标： $\max \sum_{i=1}^n v_i x_i$

剪枝函数： $tw + w_i \leq c$ （约束函数）

$bound(i) > maxv$ （限界函数）

第*i*层节点：(*i*, *tw*, *tv*, *tx*)



其中， $bound(i) = tv + v_i + rv$ 表示装入第*i*种物品能够产生的背包价值的上界； rv 表示第*i*+1~*n*种物品能够产生的装入物品价值； $maxv$ 表示当前最优的物品选择方案产生的背包价值。

由于 $bound(i) = bound(i-1) > maxv$ ，所以可以省略限界函数，仅需使用约束函数，扩展满足 $tw + w_i \leq c$ 的左孩子节点。

12.2.1 0-1背包问题

② 右剪枝

右分支的扩展规则：不装入物品。

约束条件： $\sum_{i=1}^n w_i x_i \leq c$

目标函数的优化目标： $\max \sum_{i=1}^n v_i x_i$

剪枝函数：无需约束函数

$\text{bound}(i) > \text{maxv}$ （限界函数）

第*i*层节点：(*i*, *tw*, *tv*, *tx*)

若 $\text{tw} + w_i \leq c$ ，则
装入第*i*种物品：

$\text{tx}_i = 1$

$\text{tw} = \text{tw} + w_i$

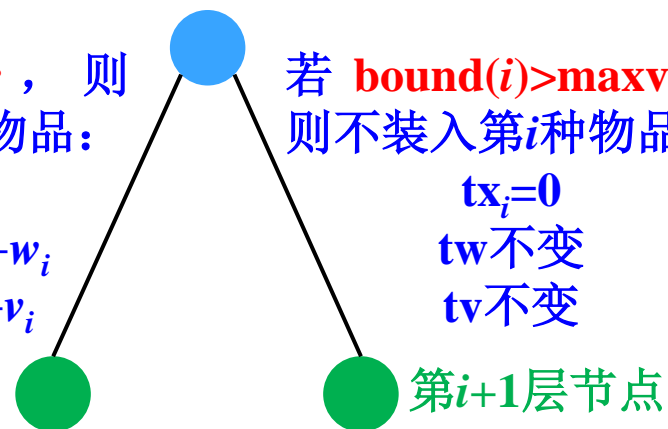
$\text{tv} = \text{tv} + v_i$

若 $\text{bound}(i) > \text{maxv}$ ，
则不装入第*i*种物品：

$\text{tx}_i = 0$

tw 不变

tv 不变



其中， $\text{bound}(i) = \text{tv} + \text{rv}$ 表示放弃第*i*种物品能够产生的背包价值的上界。

右分支仅需使用限界函数，扩展 $\text{tv} + \text{rv} > \text{maxv}$ 的右孩子节点。

显然， rv 越小，剪枝越多，为了构造尽可能小的 rv ，可以预先将所有物品按照单位质量价值递减排列。

12.2.1 0-1背包问题

【算法描述】

设 n 表示物品种类， c 表示背包容量。

一维结构体数组 A 记录物品信息，其中， $A[i]$ 记录第 i 种物品的信息， $A[i].no$ 表示物品编号， $A[i].w$ 表示物品的质量， $A[i].v$ 表示物品的价值， $A[i].p$ 表示物品单位质量的价值($1 \leq i \leq n$)。

tw 记录当前的背包质量； tv 记录当前的背包价值；一维数组 tx 记录搜索过程中的解向量。

$maxv$ 记录最优物品选择方案产生的背包价值；一维数组 x 记录最优解向量。

i 表示递归深度； $bound$ 记录当前扩展节点的背包价值上界。

12.2.1 0-1背包问题

应用回溯法求解0-1背包问题的算法描述如下。

Step 1: 将所有物品按照单位质量的价值递减排序，并存储于数组 A 。

Step 2: 若搜索到解空间树的叶子节点，即 $i > n$ ，表示找到了问题的一个更优解，则更新 $x = tx$ ， $maxv = tv$ ，返回；否则向下执行。

Step 3: 判断是否可以扩展左分支，即是否满足 $tw + A[i].w \leq c$ ，满足则选择装入第 i 种物品，更新 $tx[i]$ 、 tw 和 tv ，并进入左分支递归处理。

Step 4: 判断是否可以扩展右分支，即是否满足 $bound > maxv$ ，满足则选择不装入第 i 种物品，更新 $tx[i]$ ，并进入右分支递归处理。

算法结束后，问题的最优解记录于 x 和 $maxv$ 。

12.2.1 0-1背包问题

【算法实现】

```
struct Item    {                //物品类型

    int no;                //物品编号

    double w, v, p;        //物品质量、价值、单位质量价值

    //重载小于运算符用于按照单位质量的价值递减排序

    bool operator<(const Item &s) const

    { return p > s.p; }

};
```

12.2.1 0-1背包问题

```
class Knapsack01 {                                //0-1背包问题类

    int n, c;                                     //物品种类、背包容量

    Item *A;                                       //物品数组

    double maxv;                                   //记录最优解物品选择方案产生的背包价值

    int *x;                                        //记录最优解向量

    //计算扩展节点的背包价值上界

    double Bound(int i, double tw, double tv);

    //应用回溯法求解0-1背包问题

    void Backtrack(int i, double tw, double tv, int *tx);
```

12.2.1 0-1背包问题

public:

Knapsack01(int num, int cc, Item * it);

virtual ~Knapsack01();

void Solve(); //求解0-1背包问题

void Show(); //输出0-1背包问题的结果

};

12.2.1 0-1背包问题

```
void Knapsack01::Solve() {           //求解0-1背包问题
    int *tx = new int[n + 1];
    //计算物品的单位质量的价值
    for(int i = 1; i <= n; i++) A[i].p = A[i].v / A[i].w;
    //将物品按照单位质量的价值递减排序
    sort(A + 1, A + n + 1);
    Backtrack(1, 0, 0, tx); //应用回溯法求解0-1背包问题
    delete []tx;
}
```

12.2.1 0-1背包问题

//应用回溯法求解0-1背包问题

```
void Knapsack01::Backtrack(int i, double tw, double tv, int *tx) {  
    if(i>n) {                //找到一个叶子节点，即找到一个更优解  
        maxv = tv;          //记录当前最优物品选择方案产生的背包价值  
        for(int j = 1; j <= n; j++) x[j] = tx[j];    //保存更优解  
        return;  
    }  
    if(tw + A[i].w <= c) {    //左分支剪枝  
        tx[i] = 1;           //装入第i种物品  
        Backtrack(i + 1, tw + A[i].w, tv + A[i].v, tx); //搜索第i+1层  
    }  
    if(Bound(i, tw, tv) > maxv) { //右分支剪枝  
        tx[i] = 0;           //不装入第i种物品  
        Backtrack(i + 1, tw, tv, tx); //继续搜索第i+1层  
    }  
}
```

12.2.1 0-1背包问题

//计算扩展节点的背包价值上界

```
double Knapsack01::Bound(int i, double tw, double tv) {  
    i++;                                //从第i+1种物品开始计算  
    while(i <= n && tw + A[i].w <= c) { //依次判断剩余物品  
        tw += A[i].w; tv += A[i].v;    //可以整个装入第i种物品  
        i++;  
    }  
    if(i <= n) return tv += (c - tw) * A[i].p; //第i种物品只能装入部分  
    return tv;  
}
```

12.2.1 0-1背包问题

【算法分析】

剪枝的节点数目随着0-1背包问题的不同实例而不同，无法准确估计。

因此，考虑算法在**最坏情况（不剪枝）**下的时间复杂度：

对于 n 种物品的0-1背包问题，解空间树有 $2^{n+1}-1$ 个节点，计算节点的背包价值上界花费的时间为 $O(n)$ ，所以，整个算法的时间复杂度为 $O(n2^n)$ 。

12.2.1 0-1背包问题

【一个简单实例的求解过程】

已知一个0-1背包问题，背包的容量 $c=14$ ，物品种类 $n=4$ ，物品信息如图（a）所示，按照单位质量价值递减排序后的物品结果如图（b）所示。

物品编号 no	质量 w	价值 v
1	2	1
2	8	6
3	5	11
4	6	9

（a）物品信息

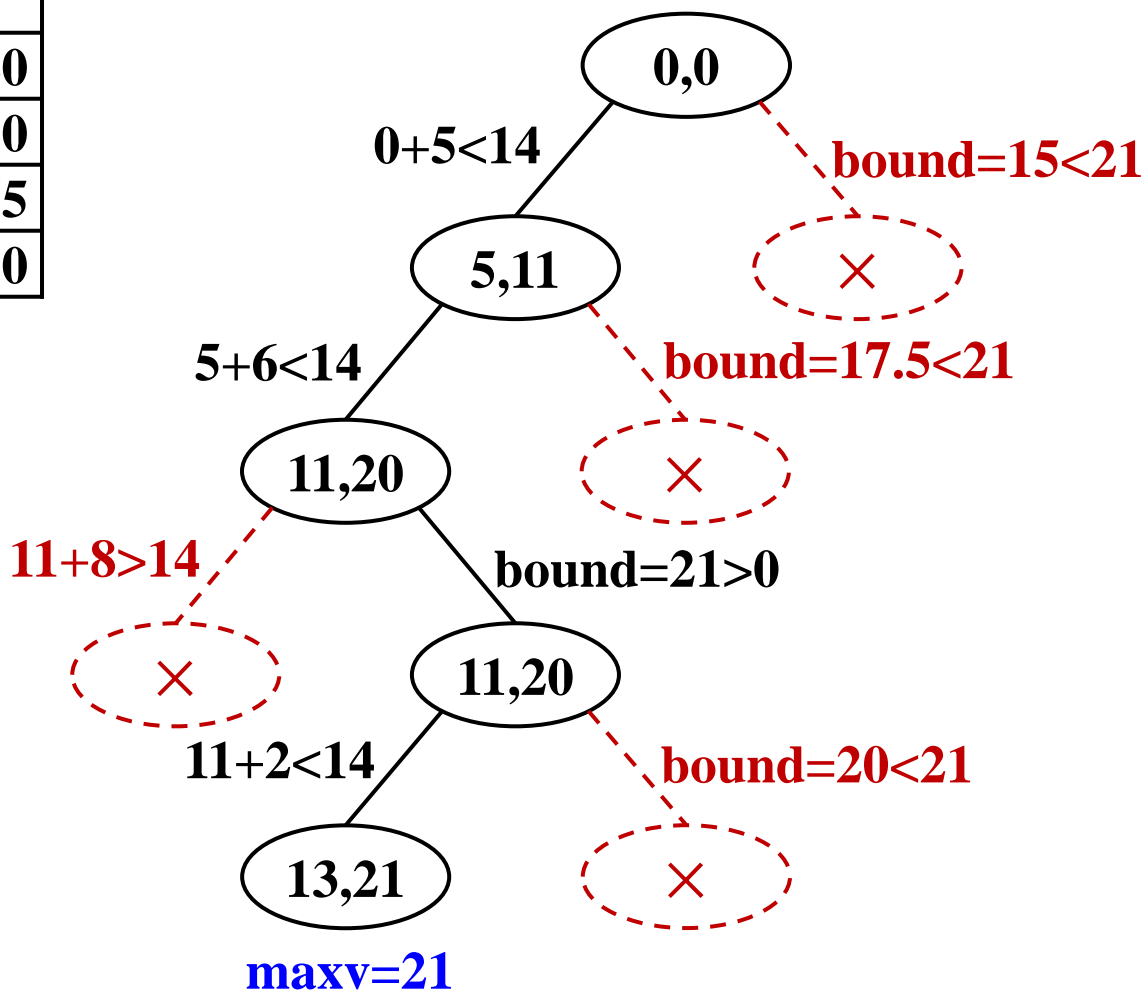
序号 i	物品编号 no	质量 w	价值 v	单位质量价值 p
1	3	5	11	2.20
2	4	6	9	1.50
3	2	8	6	0.75
4	1	2	1	0.50

（b）按照单位质量价值递减排序后的物品结果

12.2.1 0-1背包问题

i	no	w	v	p
1	3	5	11	2.20
2	4	6	9	1.50
3	2	8	6	0.75
4	1	2	1	0.50

$c=14$, $\text{maxv}=0$



12.2.2 装载问题

➤ 简单装载问题

有编号为 $1, 2, \dots, n$ 的 n 个集装箱装入一艘轮船，集装箱 i 的质量为 w_i ；轮船的载重为 c ，但无体积限制。现欲使轮船尽可能装满（尽量使轮船的负荷最大），请确定一个集装箱的装载方案。

例如，当 $n=5$ ， $c=40$ ， $w=\{ 10, 22, 16, 7, 17 \}$ 时，最佳装载方案是 $(0, 0, 1, 1, 1)$ ，装上轮船的集装箱质量恰好达到轮船载重。

12.2.2 装载问题

【问题分析】

(1) 确定问题的解空间

设 $X=(x_1, x_2, \dots, x_n)$ 表示问题的解向量，其中， $x_i(i=1, 2, \dots, n)$ 表示对第 i 个集装箱的决策。

显约束： $x_i \in \{0, 1\}(i=1, 2, \dots, n)$ ， $x_i=0$ 表示第 i 个集装箱不装入轮船； $x_i=1$ 表示第 i 个集装箱装入轮船。

隐约束： $\sum_{i=1}^n w_i x_i \leq c$

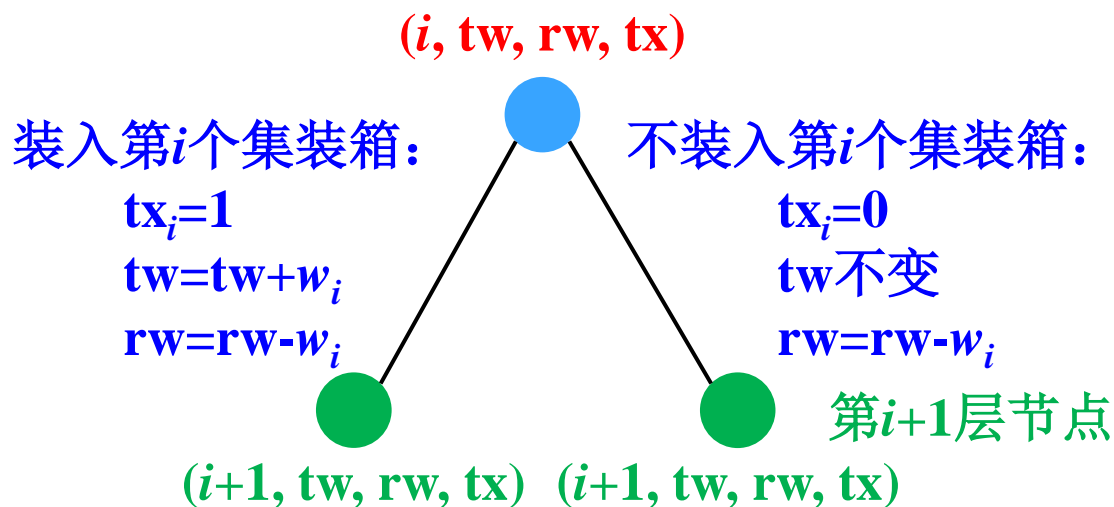
解空间树类型：子集树。

12.2.2 装载问题

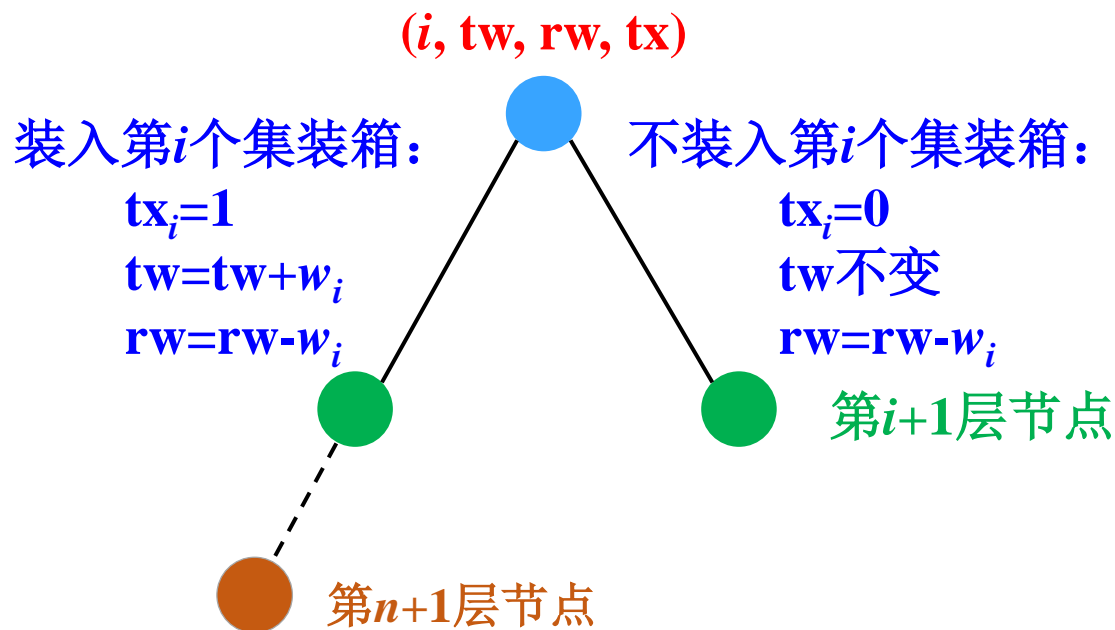
(2) 确定节点的扩展规则

为解空间树的每个节点设置状态: (i, tw, rw, tx) , 其中, i 表示节点的层次; tw 表示当前装入轮船的集装箱质量; rw 表示剩余集装箱的质量; tx 记录当前的解向量。

第 i 层节点的扩展规则: 装入或不装入第 i 个集装箱。



12.2.2 装载问题



- 叶子结点表示已经对 n 个集装箱做出了决策。

12.2.2 装载问题

(3) 确定剪枝函数

① 左剪枝

左分支的扩展规则：装入集装箱。

约束条件： $\sum_{i=1}^n w_i x_i \leq c$

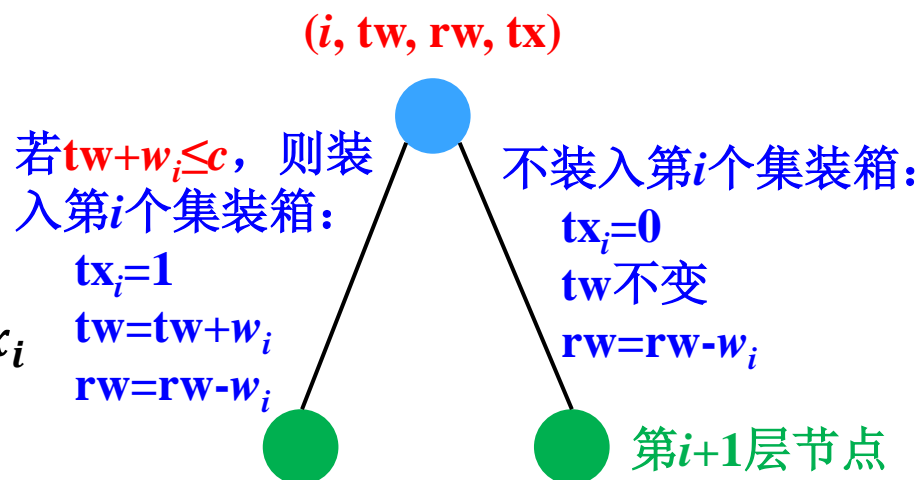
目标函数的优化目标： $\max \sum_{i=1}^n w_i x_i$

剪枝函数： $tw + w_i \leq c$ (约束函数)

$\text{bound}(i) > \text{maxw}$ (限界函数)

其中， $\text{bound}(i) = tw + rw$ 表示选择装入第 i 个集装箱能够产生的轮船负荷的上界； rw 表示剩余集装箱的质量； maxw 表示当前最优装载方案产生的轮船负荷。

由于 $\text{bound}(i) = \text{bound}(i-1) > \text{maxw}$ ，所以可以省略限界函数，仅需使用约束函数，扩展满足 $tw + w_i \leq c$ 的左孩子节点。



12.2.2 装载问题

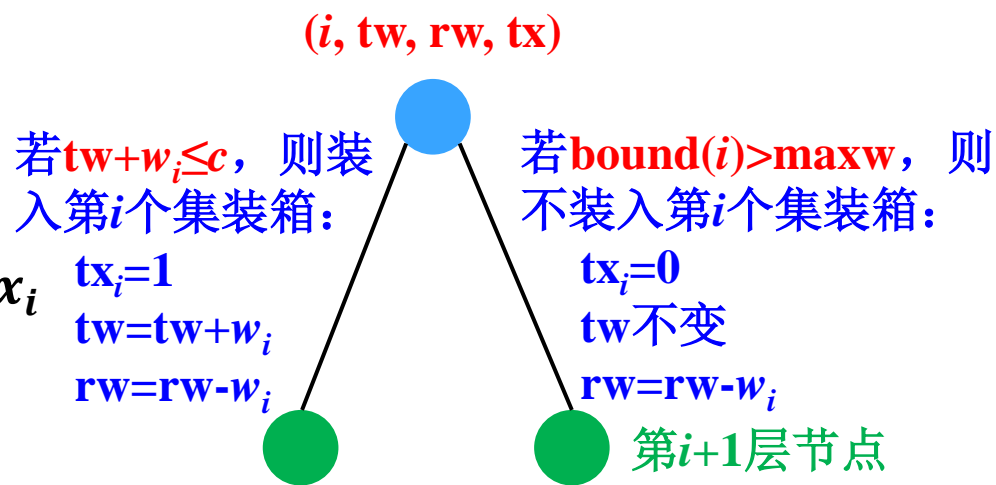
② 右剪枝

左分支的扩展规则：不装入集装箱。

约束条件： $\sum_{i=1}^n w_i x_i \leq c$

目标函数的优化目标： $\max \sum_{i=1}^n w_i x_i$

剪枝函数：无需约束函数



$bound(i) > maxw$ (限界函数)

其中, $bound(i) = tw + rw - w_i$ 表示放弃第 i 个集装箱之后能够产生的轮船负荷上界。

右分支仅需使用限界函数, 扩展 $tw + rw - w_i > maxw$ 的右孩子节点。

12.2.2 装载问题

【算法描述】

设 n 表示集装箱的数目； c 表示轮船载重。

一维数组 w 记录集装箱质量， $w[i]$ 为第 i 个集装箱的质量 ($1 \leq i \leq n$)。

tw 表示当前装上轮船的集装箱质量； rw 表示剩余集装箱的质量；一维数组 tx 记录搜索过程中的解向量。

$maxw$ 记录最优装载方案产生的轮船负荷；一维数组 x 记录最优解。

i 表示递归深度； $bound$ 记录当前扩展节点的轮船负荷上界。

12.2.2 装载问题

应用回溯法求解简单装载问题的算法描述如下。

Step 1: 若搜索到解空间树的叶子节点, 即 $i > n$, 表示找到了问题的一个更优解, 则更新 $x = tx$, $maxw = tw$, 返回; 否则向下执行。

Step 2: 判断是否可以扩展左分支, 即是否满足 $tw + w[i] \leq c$, 若满足, 则选择装上第 i 个集装箱, 更新 $tx[i]$ 、 tw 、 rw , 进入左分支递归处理。

Step 3: 判断是否可以扩展右分支, 即是否满足 $bound > maxw$ ($bound = tw + rw - w[i]$), 若满足, 则选择不装第 i 个集装箱, 更新 $tx[i]$ 、 rw , 进入右分支递归处理。

算法结束后, 问题的最优解记录于 x 和 $maxw$ 。

12.2.2 装载问题

【算法实现】

```
class SimpleLoad {    //简单装载问题类
    int n, c;          //集装箱数目，轮船载重
    int *w;            //记录集装箱质量的数组
    int maxw;          //记录最优装载方案产生的轮船负荷
    int *x;            //记录最优解
    //应用回溯法求解简单装载问题
    void Backtrack(int i, int tw, int rw, int *tx);
public:
    SimpleLoad(int num, int cc, int *weight);
    virtual ~SimpleLoad();
    void Solve();      //求解简单装载问题
    void Show();       //输出简单装载问题的结果
};
```

12.2.2 装载问题

```
void SimpleLoad::Solve() {           //求解简单装载问题

    int *tx = new int[n + 1];        //记录搜索过程中的解向量

    int rw = 0;                       //剩余集装箱的质量

    //计算初始情况下剩余集装箱的质量

    for(int i = 1; i <= n; i++)       rw += w[i];

    Backtrack(1, 0, rw, tx); //应用回溯法求解简单装载问题

    delete []tx;

}
```

12.2.2 装载问题

//应用回溯法求解简单装载问题

```
void SimpleLoad::Backtrack(int i, int tw, int rw, int *tx) {  
    if(i > n) {          //找到一个叶子节点，即找到一个更优解  
        maxw = tw; //记录当前更优装载方案产生的轮船负荷  
        for(int j = 1; j <= n; j++) x[j] = tx[j];      //保存更优解  
        return;  
    }  
    if(tw + w[i] <= c) { //左分支剪枝  
        tx[i] = 1;      //装上第i个集装箱  
        Backtrack(i + 1, tw + w[i], rw - w[i], tx); //继续搜索第i+1层  
    }  
}
```

12.2.2 装载问题

```
if(tw + rw - w[i] > maxw) {           //右分支剪枝
    tx[i] = 0;                          //不装第i个集装箱
    Backtrack(i + 1, tw, rw - w[i], tx); //继续搜索第i+1层
}
}
```

【算法分析】

剪枝的节点数目随着简单装载问题的不同实例而不同，无法准确估计。因此，考虑算法在**最坏情况（不剪枝）**下的**时间复杂度**：对于 n 个集装箱的简单装载问题，解空间树有 $2^{n+1}-1$ 个节点，所以，整个算法的时间复杂度为 $O(2^n)$ 。

12.2.2 装载问题

➤ 复杂装载问题

有编号为 $1, 2, \dots, n$ 的 n 个集装箱要装上两艘载重分别为 c_1 和 c_2 的轮船，其中，集装箱 i 的质量为 w_i ，且 $w_1 + w_2 + \dots + w_n \leq c_1 + c_2$ 。请确定是否存在合理的装载方案可将这些集装箱全部装上这两艘轮船，如果存在，那么请给出一种装载方案。

例如，若 $n=3$ ， $c_1=c_2=40$ ， $w=\{15, 35, 25\}$ ，可以将集装箱1和3装上第一艘轮船，集装箱2装上第二艘轮船。

若 $n=3$ ， $c_1=c_2=40$ ， $w=\{20, 35, 25\}$ ，则无法将这3个集装箱都装上轮船。

12.2.2 装载问题

【问题分析】

若复杂装载问题有解，则必然可以按照如下步骤得到装载方案：

- (1) 将第1艘轮船尽可能装满，使得第1艘船的负荷达到最大；
- (2) 将剩余的集装箱装上第2艘轮船。

否则，复杂装载问题必然无解。

由此，复杂装载问题的求解过程：

- (1) 在 n 个集装箱中寻找一个装载方案使得第1艘轮船尽可能装满。
- (2) 计算剩余集装箱的质量。
- (3) 判断剩余集装箱的质量是否超过第2艘轮船的载重，若没有超过，则得到问题的解，否则表明问题无解。

12.2.2 装载问题

【算法实现】

```
class ComplexLoad {    //复杂装载问题类
    int n;              //集装箱数目
    int c1, c2;         //第1艘和第2艘轮船的载重
    int *w;             //记录集装箱质量的数组
    int maxw;           //记录最优装载方案产生的第1艘轮船的负荷
    int *x;             //记录第1艘轮船的最优装载方案
    //回溯法求解第1艘轮船的最优装载方案
    void Backtrack(int i, int tw, int rw, int *tx);
public:
    ComplexLoad(int num, int cc1, int cc2, int *weight);
    virtual ~ComplexLoad();
    void Solve();       //求解复杂装载问题
    void Show();        //输出复杂装载问题的结果
};
```


12.2.2 装载问题

<pre>void ComplexLoad::Solve() { int *tx = new int[n + 1]; int total = 0, rw = 0; //计算所有集装箱的质量 for(int i = 1; i <= n; i++) rw = total; Backtrack(1, 0, rw, tx); if(total - maxw > c2) { //若剩余集装箱质量大于第2艘轮船的载重，则没有装载方案 maxw = 0; for(int i = 1; i <= n; i++) x[i] = 0; } delete []tx; }</pre>	<pre>//求解复杂装载问题 //记录搜索过程中的解向量 //所有集装箱的质量， 剩余集装箱的质量 total += w[i]; //初始情况下剩余集装箱的质量 //回溯法求解第1艘轮船的最优装载方案 //比较剩余集装箱质量与第2艘轮船的载重 //maxw置为0表示没有装载方案 //置解向量为零向量</pre>
--	---

12.2.2 装载问题

//回溯法求解第1艘轮船的最优装载方案

```
void ComplexLoad::Backtrack(int i, int tw, int rw, int *tx) {  
    if(i > n) {                //找到一个叶子节点，即找到一个更优解  
        maxw = tw; //记录当前更优装载方案产生的轮船负荷  
        for(int j = 1; j <= n; j++) x[j] = tx[j];        //保存更优解  
        return;  
    }  
  
    if(tw + w[i] <= c1) {      //左分支剪枝  
        tx[i] = 1;            //装上第i个集装箱  
        Backtrack(i + 1, tw + w[i], rw - w[i], tx); //继续搜索第i+1层  
    }  
}
```

12.2.2 装载问题

```
if(tw + rw - w[i] > maxw) {           //右分支剪枝
    tx[i] = 0;                          //不装第i个集装箱
    Backtrack(i + 1, tw, rw - w[i], tx); //继续搜索第i+1层
}
}
```

【算法分析】

上述算法的时间复杂度与求解简单装载问题的算法的时间复杂度相同，亦为 $O(2^n)$ 。

12.2.3 n 皇后问题

在一个 $n \times n$ 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、同一列或同一斜线上的棋子。 n 皇后问题等价于在 $n \times n$ 格的棋盘上放置 n 个皇后，使得不同的皇后不在同一行、同一列或同一斜线上。

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

12.2.3 n 皇后问题

【问题分析】

(1) 确定问题的解空间

设 $Q=(q_1, q_2, \dots, q_n)$ 表示问题的解向量，其中， $q_i(i=1, 2, \dots, n)$ 表示棋盘第 i 行的皇后放置位置，即第 i 个皇后的放置位置。

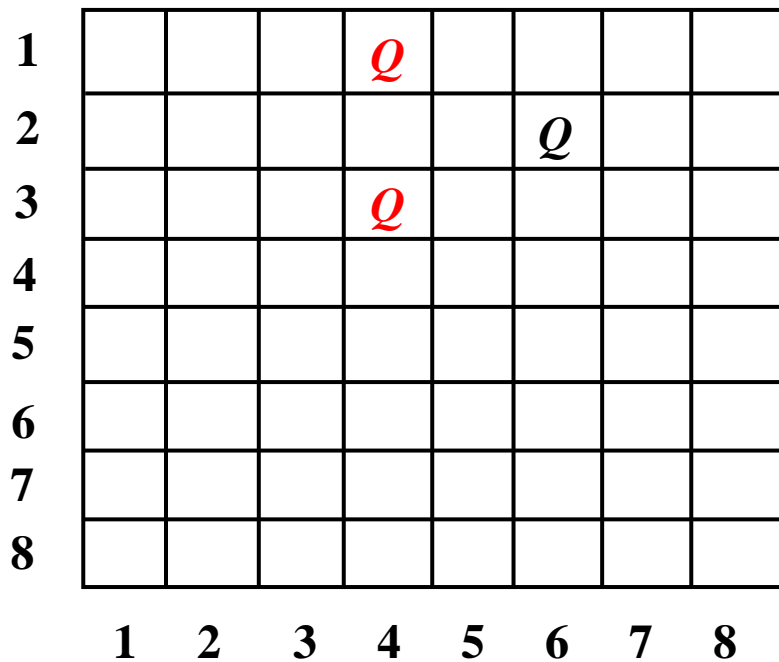
显约束： $q_i \in \{1, 2, \dots, n\}(i=1, 2, \dots, n)$ ，其中， $q_i=j(j=1, 2, \dots, n)$ 表示棋盘第 i 行的皇后放置在第 j 列。

隐约束：不同的皇后不在同一行、同一列或同一斜线。

解空间树类型：子集树。

12.2.3 n 皇后问题

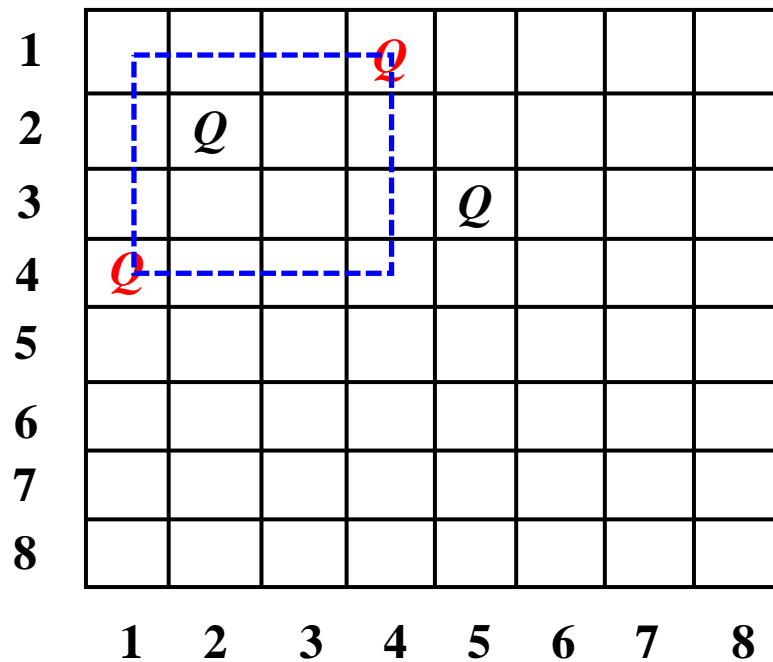
◆ n 皇后问题隐约束的表达



同一行：无需考虑

同一列： $\forall i \neq j, q_i = q_j$

同一斜线： $\forall i \neq j, |q_i - q_j| = |i - j|$



隐约束的表达：

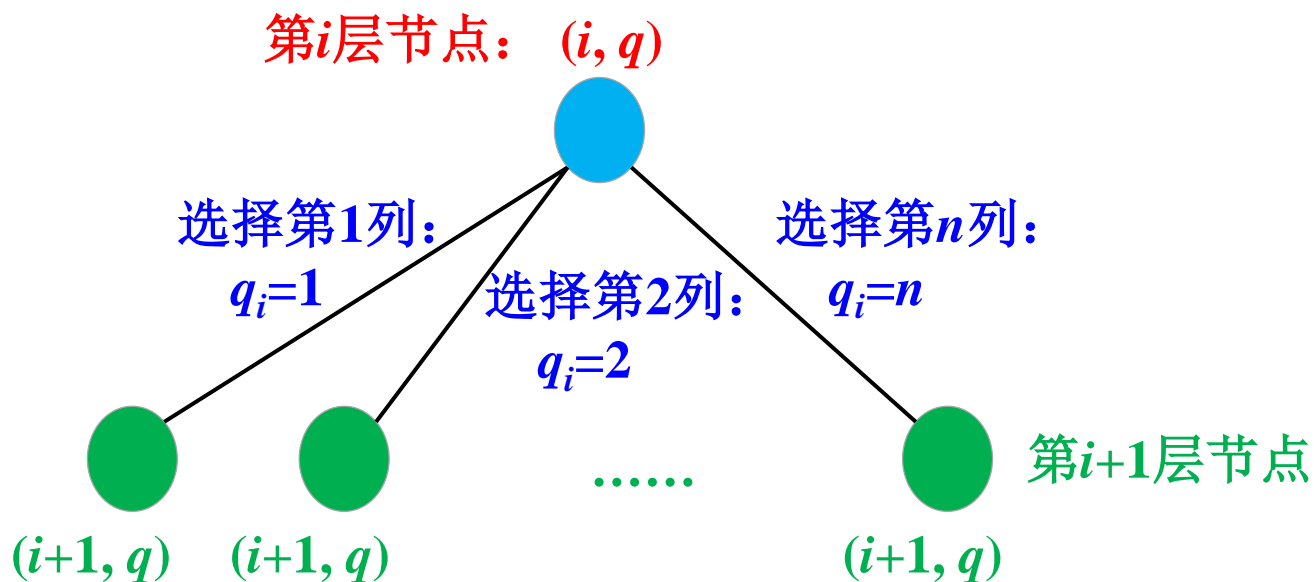
$\forall i \neq j, q_i \neq q_j$ 且 $|q_i - q_j| \neq |i - j|$

12.2.3 n 皇后问题

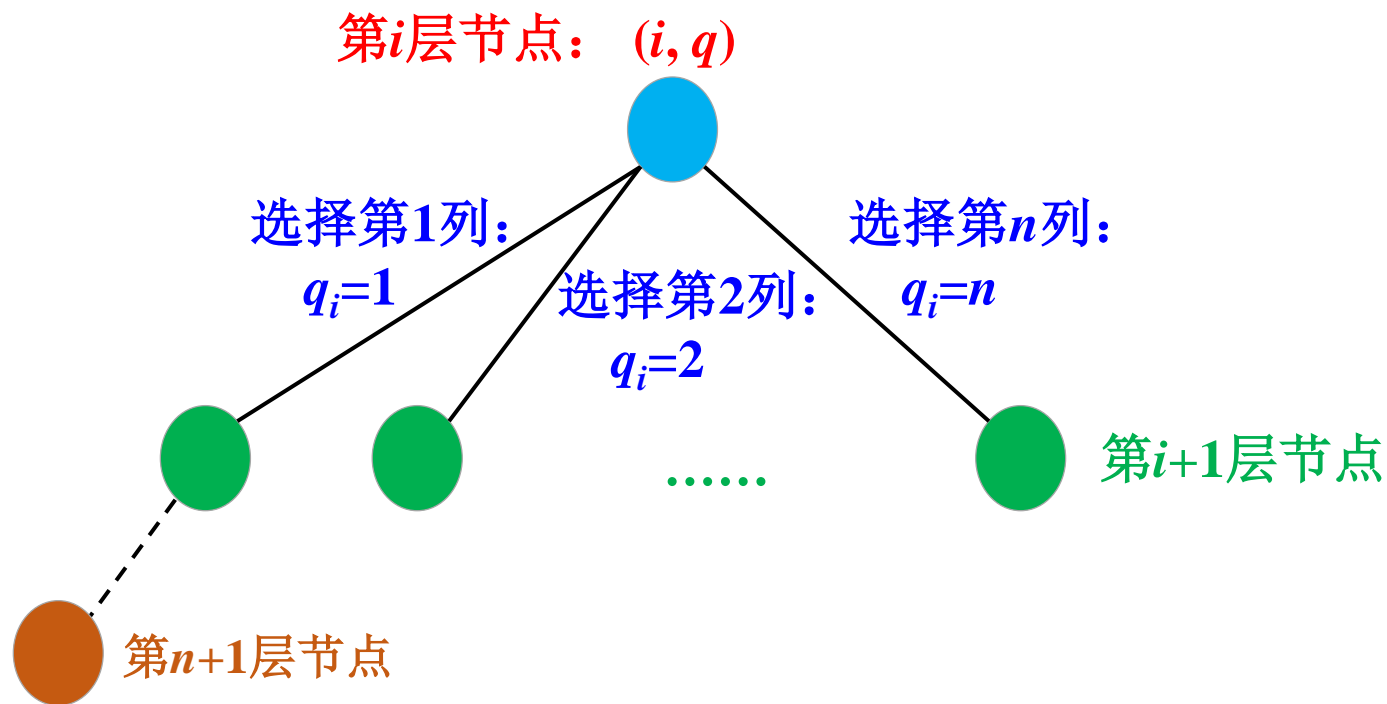
(2) 确定节点的扩展规则

为解空间树的每个节点设置状态： (i, q) ，其中， i 表示节点的层次， q 记录当前的解向量。

第 i 层节点的扩展规则：在棋盘第 i 行的第 $1 \sim n$ 列选择 1 列放置皇后。



12.2.3 n 皇后问题



■ 叶子节点表示 n 个皇后的位置已经放置完毕。

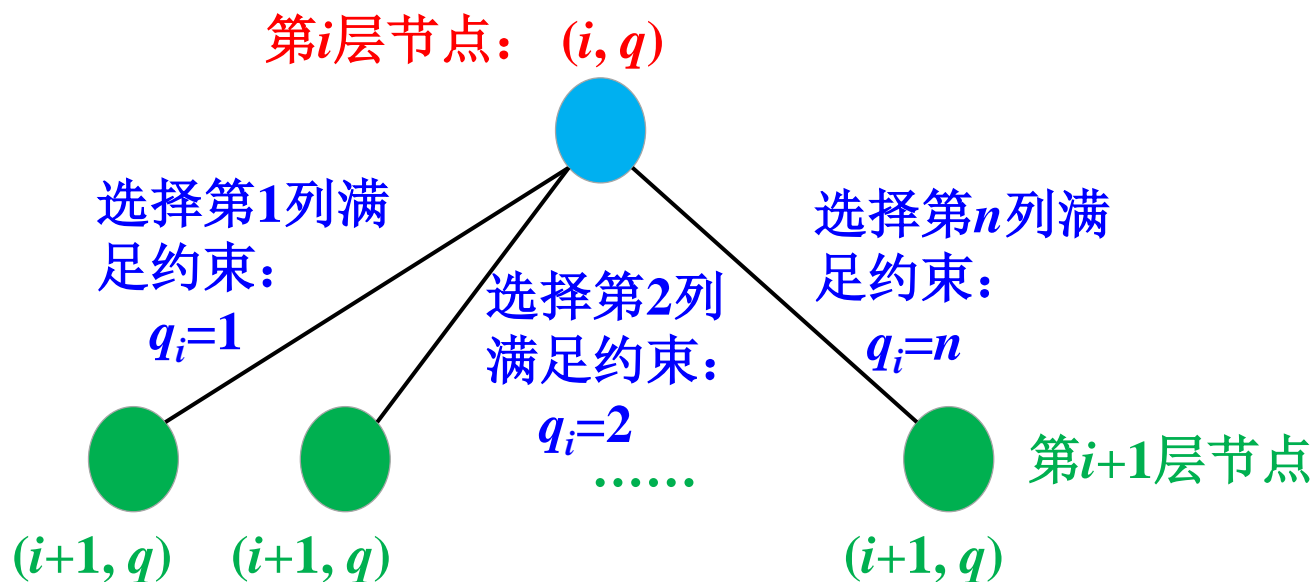
12.2.3 n 皇后问题

(3) 确定剪枝函数

分支的扩展规则：选择一列放置皇后。

约束条件： $\forall i \neq j, q_i \neq q_j$ 且 $|q_i - q_j| \neq |i - j|$

剪枝函数： $\forall i \neq j, q_i \neq q_j$ 且 $|q_i - q_j| \neq |i - j|$ (约束函数)



12.2.3 n 皇后问题

【算法描述】

设 n 表示棋盘的行数和列数；一维数组 q 记录棋盘的每一行放置皇后的位置， $q[i]$ 记录棋盘的第 i 行放置皇后的列标($1 \leq i \leq n$)。

应用回溯法求解 n 皇后问题的算法描述如下。

Step 1: 若搜索到解空间树的叶子节点，即 $i > n$ ，表示找到问题的一个解，输出解并返回；否则向下执行。

Step 2: 对 $j=1, 2, \dots, n$ 列，进行如下处理：判断是否可以选择第 j 列放置皇后，若可以，则选择第 j 列放置皇后，更新 $q[i]$ ，并进入第 j 分支递归处理。

12.2.3 n 皇后问题

//应用回溯法求解 n 皇后问题，count记录解的数目

```
void Queens(int i, int n, int * q, int &count) {  
    if(i > n) {          //找到一个叶子节点，即找到一个可行解  
        count++;        //可行解数目加1  
        cout << "Solution " << count << " is: ";  
        for(int j = 1; j <= n; j++)    //输出当前的可行解  
            cout << "( " << j << " , " << q[j] << " ) ";  
        cout << endl;  
        return;  
    }  
}
```

12.2.3 n 皇后问题

//对第*i*行的皇后位置尝试第1~*n*列的不同选择

```
for(int j = 1; j <= n; j++) {
```

```
    q[i] = j;                //第i行的皇后放置在第j列
```

```
    if(place(i, q))          //判断在第i行是否可以放置在第j列
```

```
        Queens(i + 1, n, q, count); //可以，则继续搜索第i+1层
```

```
}
```

```
}
```

12.2.3 n 皇后问题

//检查在第*i*行的 $q[i]$ 列上能否放置皇后

```
bool place(int i, int *q) {
```

```
    if(i == 1) return true;    //第1行可以在任意列放置
```

```
    for(int j = 1; j < i; j++) //逐行检查与已放置的皇后是否冲突
```

```
        if((q[i] == q[j]) || (abs(q[i] - q[j]) == abs(i - j))) //检查约束
```

```
            return false;    //存在冲突，不能放置
```

```
    return true;    //不存在冲突，可以放置
```

```
}
```

12.2.3 n 皇后问题

// n 皇后问题的迭代（非递归）回溯算法

```
void Queens(int n, int * q, int& count) {
```

```
    int i = 1;           //i表示棋盘当前处理的行，也表示第i个皇后
```

```
    q[i] = 0;           //q[i]记录第i行的皇后位置，初始化为0
```

```
    while(i >= 1) {      //当搜索尚未结束
```

```
        q[i]++;          //尝试将皇后放置在q[i]的后一列位置
```

```
        //若第q[i]列不能放置皇后，则继续尝试后继列
```

```
        while(q[i] <= n && !place(i, q)) q[i]++;
```

12.2.3 n 皇后问题

```
if(q[i] <= n) { //为第i个皇后找到一个合适位置(i, q[i])
    if(i == n) { //若放置了所有皇后，则得到一个可行解
        count++; //可行解数目加1
        cout << "Solution " << count << " is: ";
        for (int j = 1; j <= n; j++) //输出当前可行解
            cout << "( " << j << " , " << q[j] << " ) ";
        cout << endl;
    }
    else { i++; q[i] = 0; } //继续处理下一行
}
else i--; //第i行搜索完毕，回溯到上一行
}
}
```

12.2.3 n 皇后问题

【算法分析】

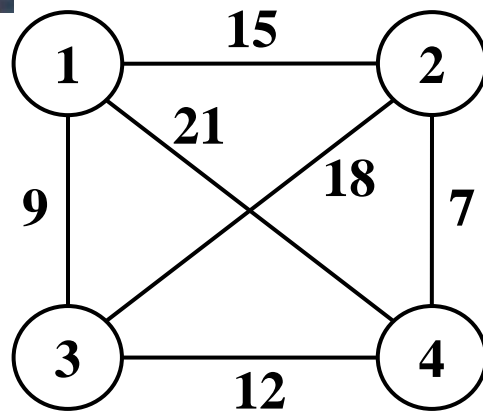
在算法中，棋盘的每一行对一个皇后的位置都要试探 n 列，棋盘共有 n 行，故算法的时间复杂度为 $O(n^n)$ 。

12.2.4 旅行商问题

一个商品推销员要去 n 个城市（编号为 $1, 2, \dots, n$ ）推销商品，已知各城市之间的路程（或旅费）。该推销员从一个城市出发，途经每个城市一次，最后回到出发城市。推销员应该如何选择行进路线，才能使总的路程（或旅费）最短（或最少）。

例：右图所示的旅行商问题，若推销员选择从城市1出发，则最优的行进路线是：城市1->城市2->城市4->城市3->城市1。

最短（或最少）路程（或旅费）是43。



12.2.4 旅行商问题

【问题分析】

旅行商问题：寻找连通网的一条哈密尔顿回路，要求构成回路的边的权值之和最小，即路径长度最小。

(1) 确定问题的解空间

省略回路的终点，用向量 $X=(x_1, x_2, \dots, x_n)$ 表示旅行商问题的解向量，其中， $x_i(i=1, 2, \dots, n)$ 表示哈密尔顿回路上的第 i 个顶点。

设 $S=\{1, 2, \dots, n\}$ 是编号为 $1, 2, \dots, n$ 的顶点构成的集合，则旅行商问题的显约束：

$$x_1 \in S \text{ 且 } x_i \in S - \{x_1, x_2, \dots, x_{i-1}\} \quad (i=2, 3, \dots, n)$$

即哈密尔顿回路上的第 i 个顶点不能与前 $i-1$ 个顶点重复。

12.2.4 旅行商问题

旅行商问题的隐约束： $\forall i \in \{1, 2, \dots, n-1\}$, x_i 和 x_{i+1} 之间有边，并且 x_n 和 x_1 之间有边。

解空间树类型：排列树。

12.2.4 旅行商问题

◆ 旅行商问题隐约束的表达

对于网络，邻接矩阵定义如下：

$$\text{arcs}[i, j] = \begin{cases} w_{ij} & i \neq j \text{ 且 } (v_i, v_j) \in E \text{ 或 } \langle v_i, v_j \rangle \in E \\ 0 & i = j \\ \infty & \text{其他} \end{cases}$$

旅行商问题的隐约束表示为：

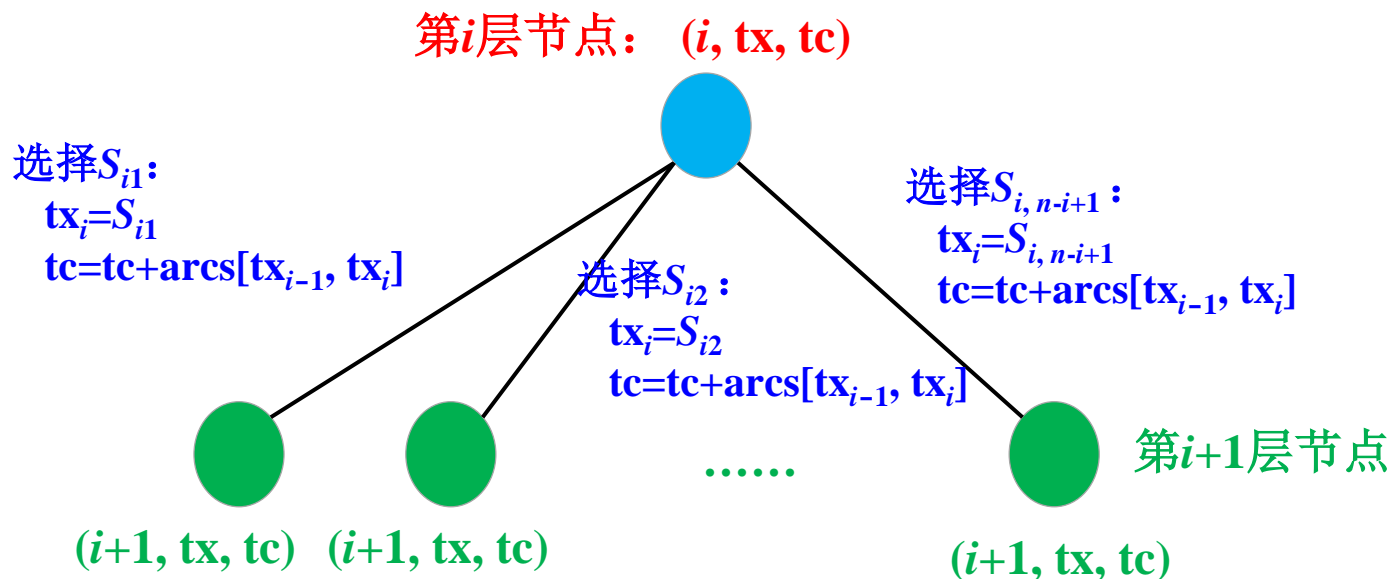
$$\forall i \in \{1, 2, \dots, n-1\}, \text{arcs}[x_i, x_{i+1}] \neq \infty \text{ 且 } \text{arcs}[x_n, x_1] \neq \infty$$

12.2.4 旅行商问题

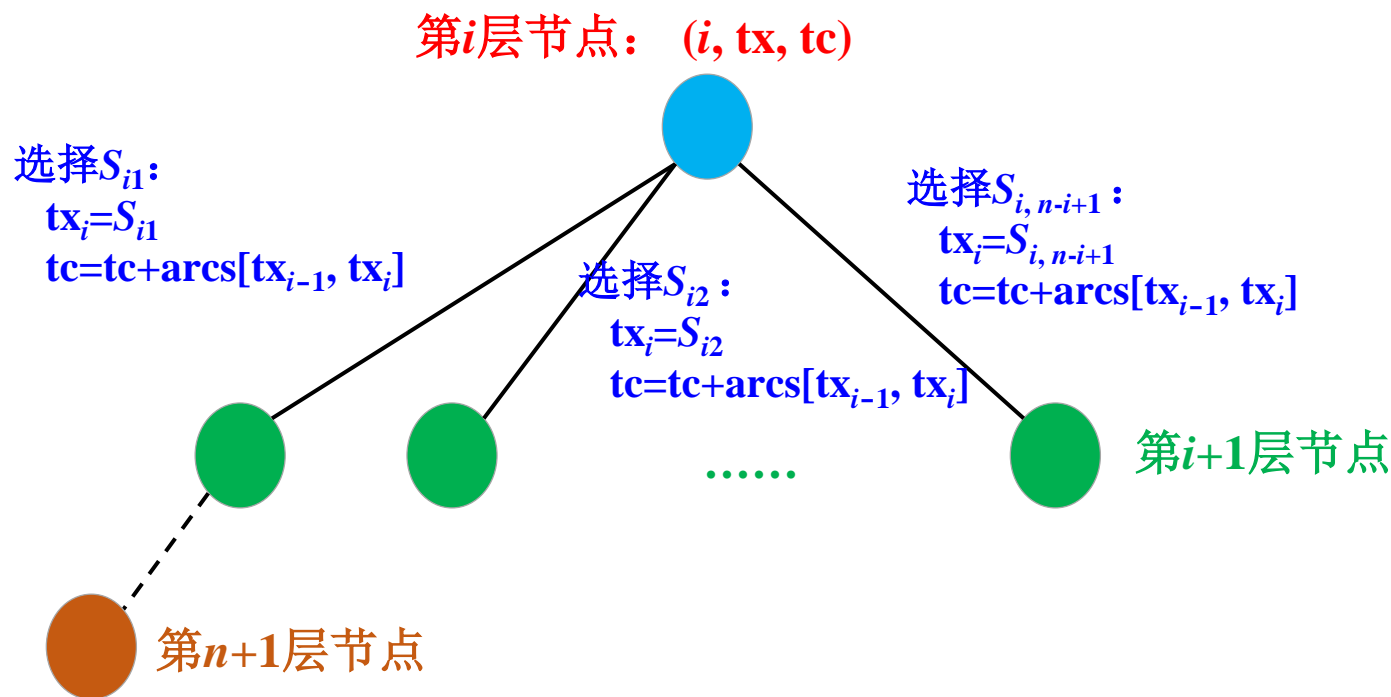
(2) 确定节点的扩展规则

为解空间树的每个节点设置状态: $(i, \mathbf{tx}, \mathbf{tc})$, 其中, i 表示节点的层次; \mathbf{tx} 记录当前的解向量, 即当前已经确定的路径; \mathbf{tc} 记录当前路径的长度。

节点的扩展规则: 第 i 层在 $S_i = S - \{tx_1, tx_2, \dots, tx_{i-1}\}$ 中选择一个顶点作为哈密尔顿回路的第 i 个顶点 ($i=1, 2, \dots, n$)。



12.2.4 旅行商问题



- 叶子节点仅仅表示找到一条从起点开始、由*n*个顶点构成的简单路径，需要进一步判断是否可以形成回路。

12.2.4 旅行商问题

(3) 确定剪枝函数

分支的扩展规则：在未选择的顶点中选择一个顶点。

约束条件： $\text{arcs}[x_i, x_{i+1}] \neq \infty \quad (i=1, \dots, n-1)$

优化目标： $\min(\sum_{i=1}^{n-1} \text{arcs}[x_i, x_{i+1}] + \text{arcs}[x_n, x_1])$

剪枝函数： $\text{arcs}[tx_{i-1}, tx_i] \neq \infty \quad (i=2, \dots, n)$ （约束函数）

$\text{bound}(i) < \text{minc} \quad (i=2, \dots, n)$ （限界函数）

其中， $\text{bound}(i)$ 表示当前选择形成的路径长度下界； minc 表示当前最小的哈密顿回路的路径长度。

12.2.4 旅行商问题

◆ $\text{bound}(i)$ 的计算

① 顶点的最小出边与最小出边和

顶点的最小出边：在一个顶点的所有边中，**权值最小的边**称为该顶点的最小出边。

最小出边和：可能出现在哈密尔顿回路中的**顶点的最小出边权值之和**称为最小出边和。

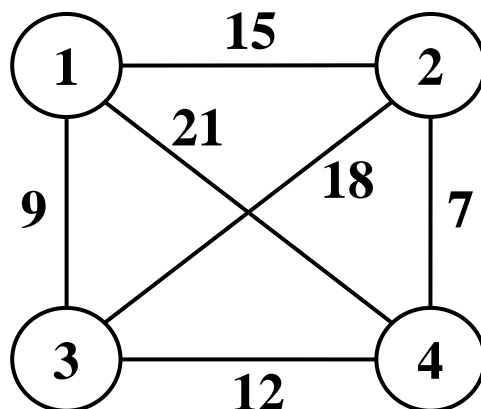
12.2.4 旅行商问题

设向量 $\text{minout}=(\text{minout}_1, \text{minout}_2, \dots, \text{minout}_n)$ 记录所有顶点的最小出边的权值，其中， minout_i 记录顶点 i 的最小出边的权值； rc 记录当前的最小出边和。

初始情况下， $\text{rc} = \sum_{i=1}^n \text{minout}_i$

在解空间树的第 i 层分支节点处，为路径上的第 $i-1$ 个顶点做出后续关于第 i 个顶点的选择之后，最小出边和 rc 应减去第 $i-1$ 个顶点的最小出边的权值。

12.2.4 旅行商问题



初始, $rc = 9 + 7 + 9 + 7 = 32$

设起点是顶点1, 则在选择路径上的第2个顶点 (无论选择哪个顶点) 之后, $rc = rc - \min out_1 = 23$

若路径的第2个顶点选择了顶点4, 则在选择路径上的第3个顶点 (无论选择哪个顶点) 之后, $rc = rc - \min out_4 = 16$

12.2.4 旅行商问题

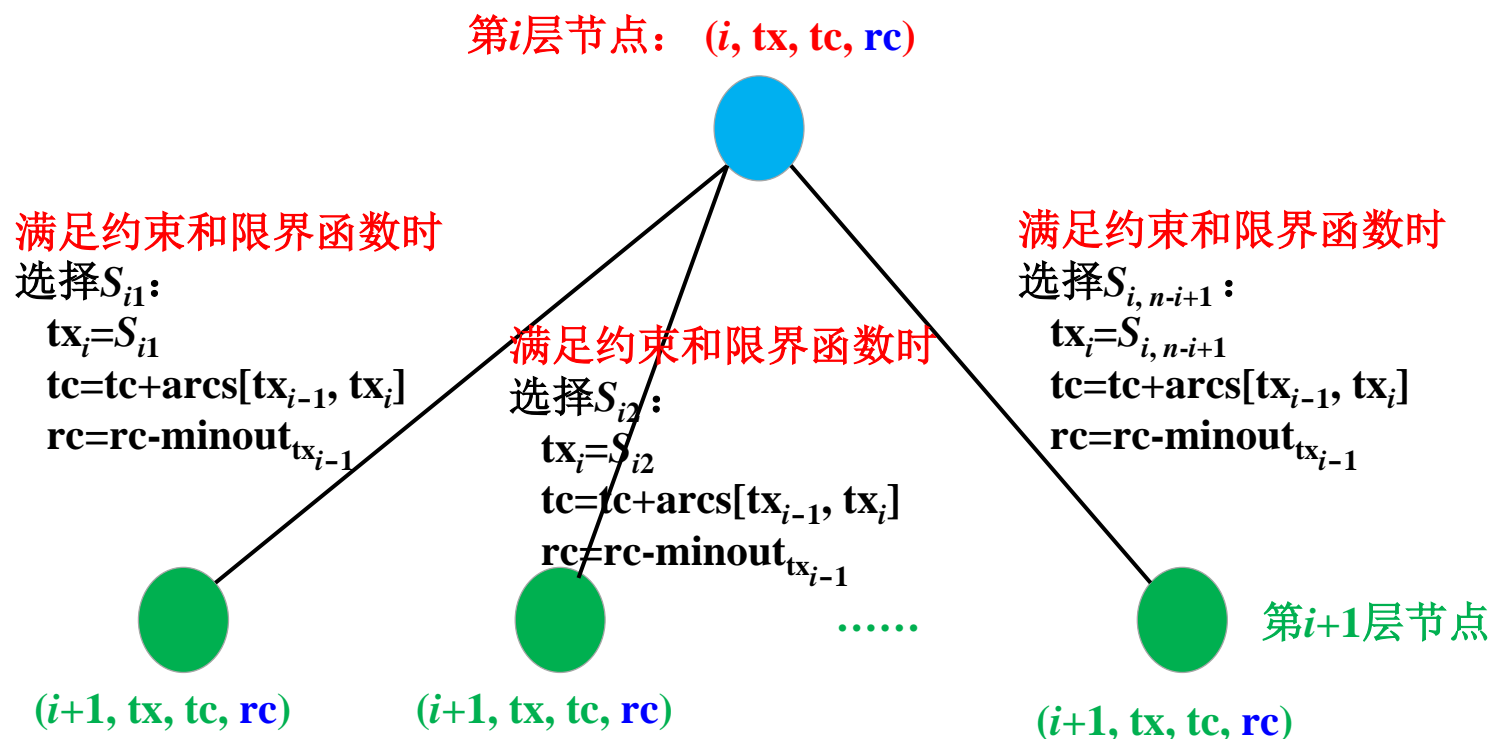
② $\text{bound}(i)$ 的计算方法

在解空间树的第 i 层分支节点处，从 $S_i = S - \{tx_1, tx_2, \dots, tx_{i-1}\}$ 中选择一个顶点作为哈密尔顿回路的第 i 个顶点时，该选择形成的路径长度下界为：

$$\text{bound}(i) = \text{tc} + \text{arcs}[tx_{i-1}, tx_i] + \text{rc} - \text{minout}_{tx_{i-1}}$$

12.2.4 旅行商问题

为解空间树的节点状态添加了最小出边和 rc ，及为扩展分支添加了剪枝函数之后的节点扩展和状态计算过程：



12.2.4 旅行商问题

【算法描述】

设 n 表示顶点数目；二维数组 arcs 表示邻接矩阵，其中， $\text{arcs}[i][j]$ 记录边 (i, j) 的权值；一维数组 minout 记录顶点的最小出边的权值， $\text{minout}[i]$ 记录顶点 i 的最小出边的权值。

一维数组 tx 记录搜索过程中的路径， $\text{tx}[i]$ 记录路径上第 i 个顶点的编号； tc 记录当前路径的长度； rc 记录当前的最小出边和。

一维数组 x 记录最优路径， $\text{x}[i]$ 记录路径上第 i 个顶点的编号； minc 记录最优路径的长度； i 表示递归深度； bound 记录当前扩展节点的路径长度下界。

12.2.4 旅行商问题

应用回溯法求解旅行商问题的算法描述如下。

Step 1: 初始化路径 tx 和 x 为顶点的初始排列，路径的第一个顶点为起点；初始化最优路径的长度 $minc$ 和当前路径长度 tc 。

Step 2: 记录所有顶点的最小出边权值于 $minout$ ，计算最小出边和 rc 。

Step 3: 初始化 $i=2$ （从解空间树的第2层开始处理）。

Step 4: 若搜索到解空间树的叶子节点，即 $i>n$ ，表示找到一条从起点开始遍历所有顶点的简单路径，则判断路径的第 n 个顶点与起点是否相连且回路的路径长度是否小于 $minc$ ，若满足上述条件表明找到一条更优的哈密尔顿回路，则更新 x 和 $minc$ ，返回；否则向下执行。

12.2.4 旅行商问题

Step 5: 对 $j=i, i+1, \dots, n$, 进行如下处理。

Step 5.1: 判断顶点 $tx[j]$ 作为路径上的第 i 个顶点是否满足约束函数 $arcs[tx[i-1]][tx[j]] \neq \infty$ 及 限界函数 $bound < minc$ ($bound = tc + arcs[tx[i-1]][tx[j]] + rc - minout[tx[i-1]]$) , 满足则交换 $tx[j]$ 与 $tx[i]$, 更新 tc 和 rc 。

Step 5.2: 进入第 $i+1$ 层递归处理。

Step 5.3: 交换 $tx[j]$ 与 $tx[i]$ 还原原始解, 还原 tc 和 rc 。

算法结束后, 问题的最优解记录于 x 和 $minc$ 。

12.2.4 旅行商问题

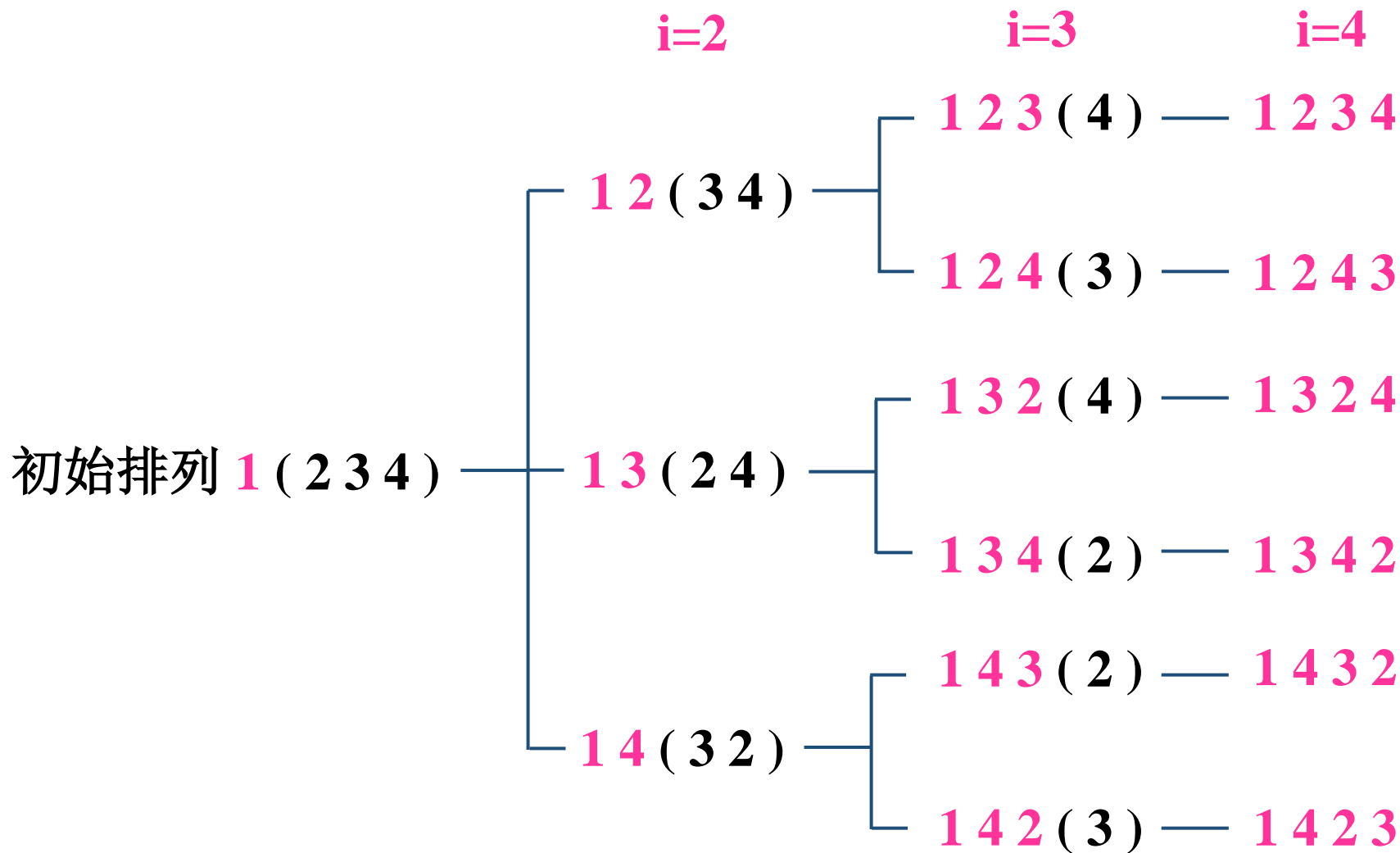
◆在选择路径的第 i 个顶点时，如何既尝试不同情况，又保证排列中顶点不重复？

关键点：

- ① 通过交换原排列中的元素。
- ② 在开始搜索前，需要形成一个初始排列。

12.2.4 旅行商问题

例：4个顶点（编号1，2，3，4），起点为编号1的顶点。



12.2.4 旅行商问题

【算法实现】

```
const int INFINITE = INT_MAX;    //定义作为无穷的数据

class TSP {                      //旅行商问题类
    int n;                       //顶点数目
    int **arcs;                  //邻接矩阵
    int *x;                      //记录最优的哈密尔顿回路
    int minc;                    //记录最优的哈密尔顿回路的路径长度
    //回溯法求解旅行商问题
    void Backtrack(int i, int start, int *tx, int tc, int *minout, int rc);
```

12.2.4 旅行商问题

public:

TSP(int num, int **am);

virtual ~TSP();

void Solve(int start); **//求解起点为start的旅行商问题**

void Show(); **//输出旅行商问题的解**

};

12.2.4 旅行商问题

```
void swap(int& a, int& b)    //交换a和b
{   int t = a;  a = b;  b = t; }

void TSP::Solve(int start) { //求解起点为start的旅行商问题
    int *tx;                //记录搜索过程中的解向量
    int tc = 0;              //记录当前路径的长度
    int *minout;             //记录顶点的最小出边权值
    int rc = 0;              //最小出边和

    tx = new int[n + 1]; minout = new int[n + 1]; //分配存储空间
    for(int i = 1; i <= n; i++) tx[i] = x[i] = i; //初始化路径
    //起点start作为路径的第1个顶点
    swap(x[1], x[start]); swap(tx[1], tx[start]);
```

12.2.4 旅行商问题

```
minc = INFINITE; //初始化最优哈密尔顿回路的长度为无穷
for(int i = 1; i <= n; i++) {    //寻找所有顶点的最小出边权值
    minout[i] = INFINITE;
    //在顶点i的邻接点中寻找权值最小的边
    for(int j = 1; j <= n; j++)
        if(arcs[i][j] != INFINITE && arcs[i][j] < minout[i])
            minout[i] = arcs[i][j];
    if(minout[i] == INFINITE) return; //不存在哈密尔顿回路
    rc += minout[i];                //累加顶点的最小出边权值
}
Backtrack(2, start, tx, tc, minout, rc); //应用回溯法求解旅行商问题
delete []tx; delete []minout;
}
```

12.2.4 旅行商问题

//应用回溯法求解旅行商问题

```
void TSP::Backtrack(int i, int start, int *tx, int tc, int *minout, int rc) {  
    if(i > n) {                //找到一个叶子节点  
        if(arcs[tx[n]][start] != INFINITE && tc + arcs[tx[n]][start] < minc) {  
            //当前路径形成回路且更优  
            for(int j = 2; j <= n; j++) x[j] = tx[j]; //更新最优哈密尔顿回路  
            x[n + 1] = start;                        //起点作为回路的终点  
            minc = tc + arcs[tx[n]][start];          //更新最优回路的路径长度  
        }  
        return;  
    }  
}
```

12.2.4 旅行商问题

```
for(int j = i; j <= n; j++)//依次尝试取顶点tx[i]~tx[n]作为路径的第i个顶点
```

```
if((arcs[tx[i - 1]][tx[j]] != INFINITE)
```

```
&& (tc + arcs[tx[i - 1]][tx[j]] + rc - minout[i - 1] < minc)) {
```

```
//满足约束函数和限界函数，选择顶点tx[j]作为路径的第i个顶点
```

```
swap(tx[i], tx[j]);
```

```
//顶点tx[j]作为路径的第i个顶点
```

```
tc += arcs[tx[i - 1]][tx[i]];
```

```
//更新路径的当前长度
```

```
rc -= minout[tx[i - 1]];
```

```
//更新最小出边和
```

```
Backtrack(i + 1, start, tx, tc, minout, rc); //继续搜索第i+1层
```

```
tc -= arcs[tx[i - 1]][tx[i]];
```

```
//还原路径的当前长度
```

```
rc += minout[tx[i - 1]];
```

```
//还原最小出边和
```

```
swap(tx[i], tx[j]);
```

```
//还原路径
```

```
}
```

```
}
```

12.2.4 旅行商问题

【算法分析】

如果不考虑更新最优解所需要的时间，上述算法的时间复杂度为 $O((n-1)!)$ 。

在最坏情况下，算法每次都需要更新当前的最优解，每次更新需要 $O(n)$ 的时间，故整个算法的时间复杂度为 $O(n!)$ 。



12.3 小结

12.3 小结

- 回溯法采用深度优先的策略搜索问题的解空间树，可以找到问题的所有解，在此基础上可实现仅寻找问题的一个解或求解问题的最优解。
- 回溯法很像穷举法，但又不同于穷举，回溯法是有组织地穷举，在搜索过程中通过剪枝函数剪去不满足约束条件或无法取得更优解的分支，从而减小搜索的范围，提升算法的效率。
- 常见的两种解空间树是子集树和排列树，各自有着不同的回溯算法框架。
- 无论什么问题，欲使用回溯法求解需要先满足回溯法的适用条件，然后按照回溯法的设计步骤开展分析和算法设计的工作。



你今天在哪一步