

算法分析与设计

第6章 回溯法

主讲人：甘文生 PhD

Email: wsgan001@gmail.com

暨南大学网络空间安全学院

Fall 2021

Jinan University, China

第6章 回溯法

内容提要：

- 理解回溯法的深度优先搜索策略
- 掌握用回溯法解题的算法框架
 - ✓ 子集树算法框架
 - ✓ 排列树算法框架
- 通过应用范例学习回溯法的设计策略

引言

未选择的路

——罗伯特·弗罗斯特

黄色的树林里分出两条路
可惜我不能同时去涉足
我在那路口久久伫立
我向着一条路极目望去
直到它消失在丛林深处

但我却选择了另外一条路
它荒草萋萋，十分幽寂
显得更诱人，更美丽
虽然在这条小路上
很少留下旅人的足迹

那天清晨落叶满地
两条路都未经脚印污染

呵，留下一条路等改日再见
但我知道路径延绵无尽头
恐怕我难以再回返
也许多少年后在某个地方，
我将轻声叹息将往事回顾：

一片树林里分出两条路
而我选择了人迹更少的一条，
从此决定了我一生的道路。

方法概述

● 搜索算法介绍

(1) 穷举搜索

(2) 盲目搜索

— 深度优先 (DFS) 或回溯搜索 (Backtracking);

— 广度优先搜索 (BFS);

— 分支限界法 (Branch & Bound);

— 博弈树搜索 (α - β Search)

(3) 启发式搜索

— A* 算法和最佳优先 (Best-First Search)

— 迭代加深的A*算法

— B*, AO*, SSS*等算法

— Local Search, GA等算法

方法概述

□ 搜索空间的三种表示：

- **表序**表示：搜索对象用线性表数据结构表示；
- **显式图**表示：搜索对象在搜索前就用图(树)的数据结构表示；
- **隐式图**表示：除了初始结点，其他结点在搜索过程中动态生成。缘于搜索空间大，难以全部存储。

□ 搜索效率的思考：随机搜索

- 上世纪70年代中期开始，国外一些学者致力于研究随机搜索求解困难的**组合问题**，将**随机过程**引入搜索；
- 选择规则是随机地从可选结点中取一个，从而**从统计角度**分析搜索的平均性能；
- 随机搜索的一个成功例子：判定一个很大的数是不是素数，获得了第一个多项式时间的算法。

方法概述

□ 回溯法 (Backtracking) 又称为试探法：

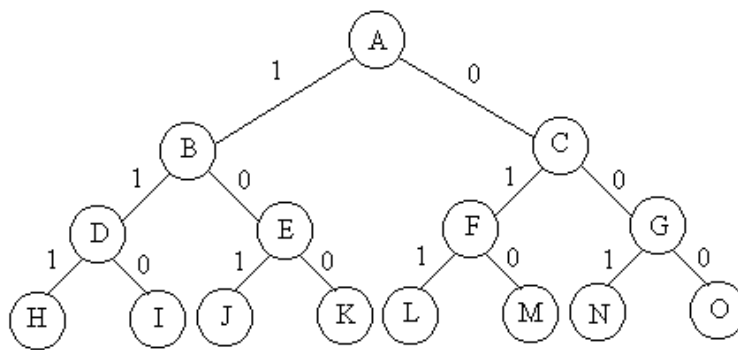
- 回溯法是一个既带有系统性又带有跳跃性的搜索算法；
- 它在包含问题的所有解的解空间树中，按照深度优先的策略，从根结点出发搜索解空间树。—— 系统性
- 算法搜索至解空间树的任一结点时，判断该结点为根的子树是否包含问题的解，如果肯定不包含，则跳过以该结点为根的子树的搜索，逐层向其祖先结点回溯。否则，进入该子树，继续深度优先的策略进行搜索。—— 跳跃性
- 这种以深度优先的方式系统地搜索问题的解的算法称为回溯法，它适用于解一些组合数较大的问题。许多复杂的、规模较大的问题都可以使用回溯法，有“通用解题方法”的美称。
- 递归 (Recursion)、迭代 (Iteration)、回溯?? 区别与联系

方法概述

□ 问题的解空间

- 问题的解向量：回溯法希望一个问题的解能够表示成一个 n 元式 (x_1, x_2, \dots, x_n) 的形式。
- 显约束：对分量 x_i 的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。

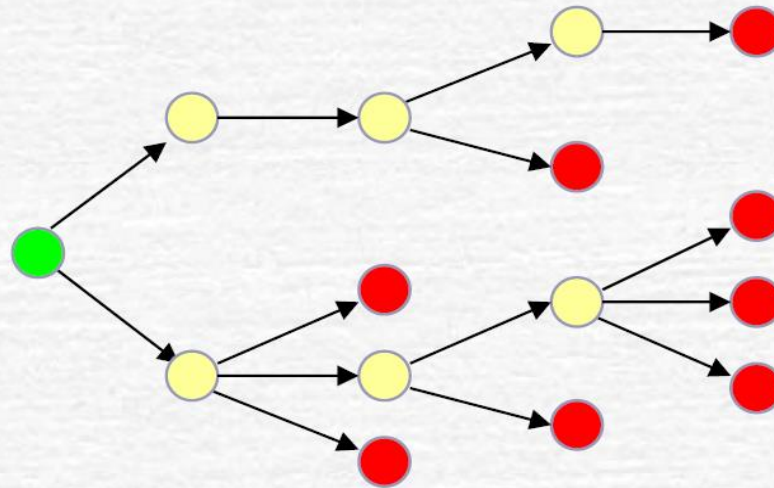
注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。



$n=3$ 时的0-1背包问题用完全二叉树表示的解空间

方法概述

解空间树：



有三种结点：

- 根结点（搜索的起点）
- 中间结点（非终端结点）
- 叶结点（终端结点），叶结点为解向量

搜索过程就是找一个或一些特别的叶结点。

对于二叉树，有深度遍历和广度遍历。深度遍历有前序、中序以及后序三种遍历方法。广度遍历即我们平常所说的层次遍历。

二叉树的遍历

对于二叉树，有深度遍历和广度遍历。深度遍历有前序、中序以及后序三种遍历方法。广度遍历即我们平常所说的层次遍历。

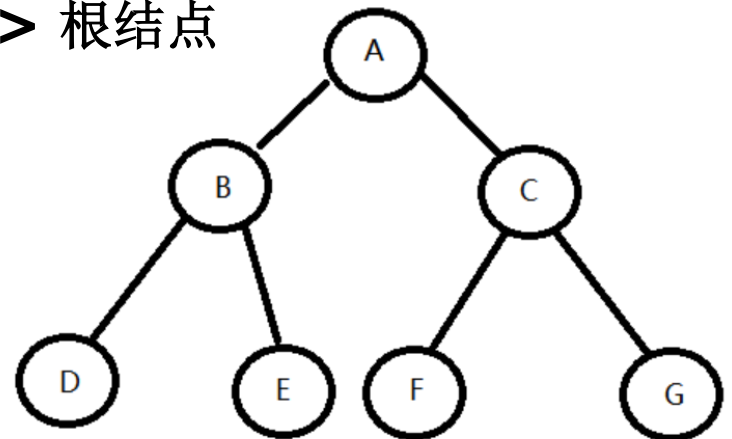
四种主要的遍历思想为：

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树 ---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

层次遍历：只需按层次遍历即可。



二叉树的遍历

前序遍历：根结点 ---> 左子树 ---> 右子树

中序遍历：左子树 ---> 根结点 ---> 右子树

后序遍历：左子树 ---> 右子树 ---> 根结点

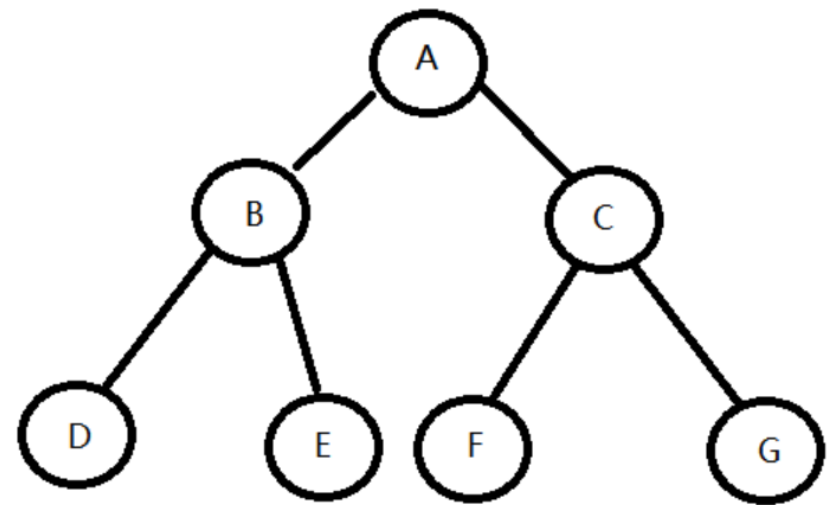
层次遍历：只需按层次遍历即可。

前序遍历：A B D E C F G

中序遍历：D B E A F C G

后序遍历：D E B F G C A

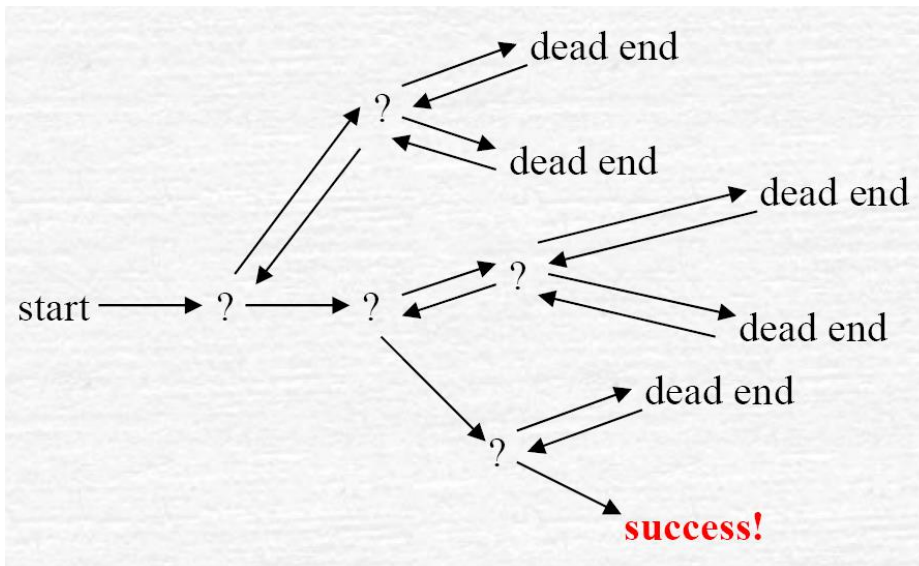
层序遍历：A B C D E F G



方法概述

□ 基本思想：

- 搜索从**开始结点(根结点)**出发，以**深度优先搜索**整个解空间。
- 这个开始结点成为**活结点**，同时也成为当前的**扩展结点**。在当前的扩展结点处，搜索**向纵深方向**移至一个新结点。这个新结点就成为新的活结点，并成为当前扩展结点。
- 如果在当前的扩展结点处不能再向纵深方向扩展，则当前扩展结点就成为**死结点**。
- 此时，应**往回移动(回溯)**至最近的一个活结点处(**回溯点**)，并使这个活结点成为当前的扩展结点；直到找到一个解或全部解。



方法概述

□ 基本步骤:

- 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。

常用剪枝函数:

- ① 用约束函数在扩展结点处剪去不满足约束的子树；
- ② 用限界函数剪去得不到最优解的子树。

方法概述

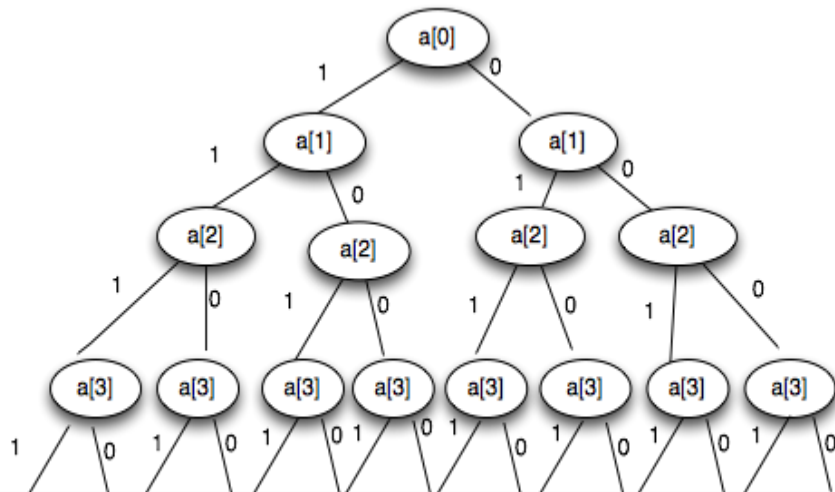
□ 二类常见的解空间树：

- ① **子集树**：当所给的问题是从 n 个元素的集合 S 中找出满足某种性质的子集时，相应的解空间树称为子集树。子集树通常有 2^n 个叶子结点，其总结点个数为 $2^{n+1}-1$ ，遍历子集树时间为 $\Omega(2^n)$ 。如0-1背包问题，叶结点数为 2^n ，总结点数 2^{n+1} 。例如0-1背包问题
- ② **排列树**：当所给问题是确定 n 个元素满足某种性质的排列时，相应的解空间树称为排列树。排列树通常有 $n!$ 个叶子结点，因此，遍历排列树需要 $\Omega(n!)$ 的计算时间。如TSP问题 (Traveling Salesman Problem, 推销员问题)，叶结点数为 $n!$ ，遍历时间为 $\Omega(n!)$ 。

方法概述

子集树

- 假设现在有一列数 $a[0], a[1], \dots, a[n-1]$ ，如果一个问题的解的长度不是固定的，并且解和元素顺序无关，即可以选择0个或多个，那么解空间的个数将是指数级别的，为 2^n ，可以用下面的子集树来表示所有的解(假设这里 $n=4$)



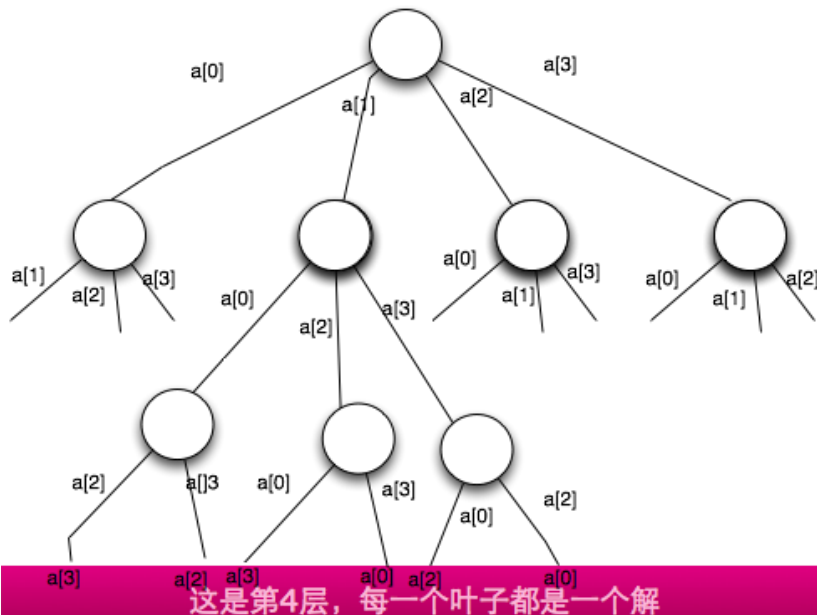
这是第4层，每一个叶子都是一个解

```
1 void backtrack(int t) { // 表示访问到第t层, t从0开始
2   if (t == n) // 如上图 (PIC. 子集树) n = 4的时候就可以输出解了
3     output(x);
4   else
5     for (int i = 0; i <= 1; i++) { // 表示选或不选a[t]
6       x[t] = i;
7       if (constraint(t) && bound(t))
8         backtrack(t + 1);
9     }
10 }
```

方法概述

排列树

- 如果解空间是由 n 个元素的排列形成，即 n 个元素的每一个排列都是解空间中的一个元素，那么，最后解空间的组织形式是排列树。



```
1 void backtrack(int t)
2 {
3     if (t == n)
4         output(x);
5     else
6         for (int i = t; i < n; i++)
7         {
8             swap(x[t], x[i]);
9             if (constraint(t) && bound(t))
10            {
11                backtrack(t + 1);
12                swap(x[t], x[i]);
13            }
14        }
15 }
```

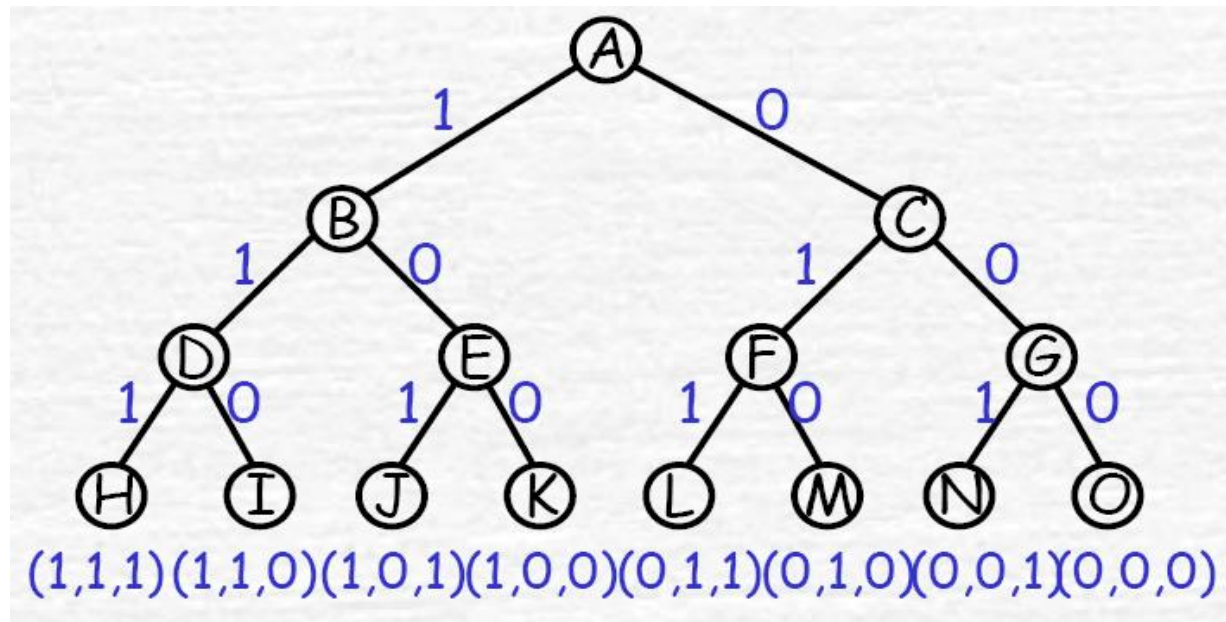
//这里的 n 表示层数，不要和子集树搞混了

方法概述

例1 [0-1 背包]: $n = 3$, $w = (16, 15, 15)$, $v = (45, 25, 25)$, $c = 30$

(1) 定义解空间: $X = \{(0,0,0), (0,0,1), (0,1,0), \dots, (1,1,0), (1,1,1)\}$

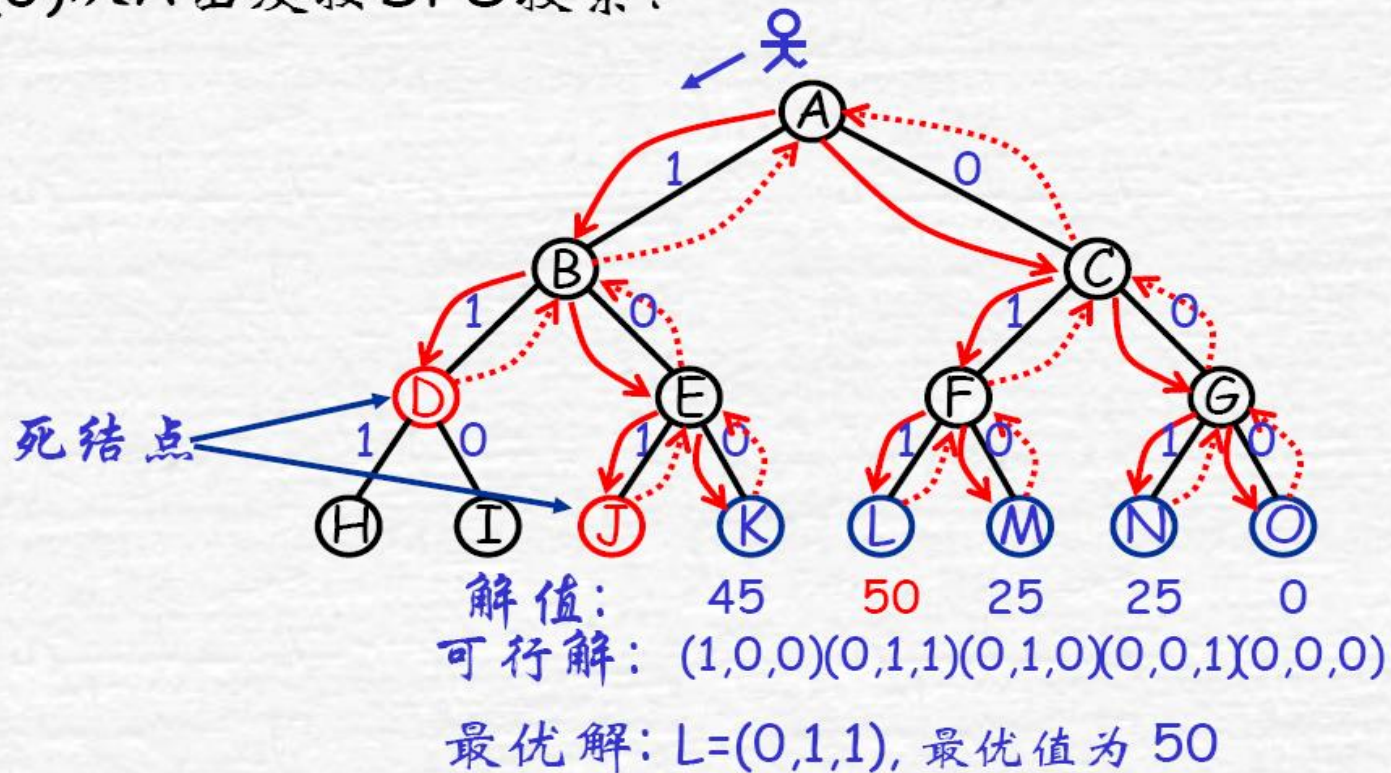
(2) 构造解空间树:



方法概述

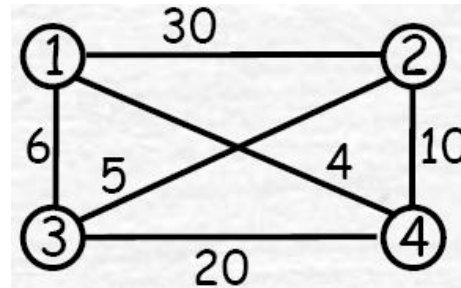
- 例1(cont.): $n=3$, $w=(16,15,15)$, $v=(45,25,25)$, $c=30$

(3)从A出发按DFS搜索:



方法概述

□ 例2 [TSP问题]:



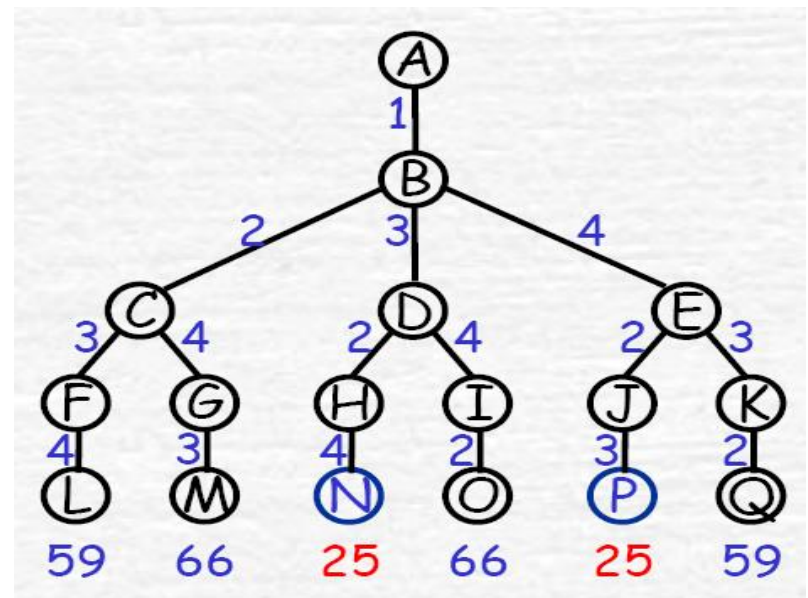
(1) 定义解空间: $X = \{12341, 12431, 13241, 13421, 14231, 14321\}$

(2) 构造解空间树:

(3) 从A出发按DFS搜索整棵树:

最优解: 13241, 14231

成本: 25



方法概述

- 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。

算法框架

- 回溯法对解空间作深度优先搜索，因此在一般情况下可用**递归**函数来实现回溯法：
- **子集树回溯算法**

```
Backtrack(int t) // 搜索到树的第t层
{ // 由第t层向第t+1层扩展，确定x[t]的值
    if t > n then output(x); // 叶子结点是可行解
    else
        while( all Xt) do // Xt为当前扩展结点的所有可能取值集合
        {
            x[t] = Xt中的第i个值;
            if( Constraint(t) and Bound(t) )
                Backtrack(t+1);
        }
}
```

执行时，从Backtrack(1)开始。

算法框架

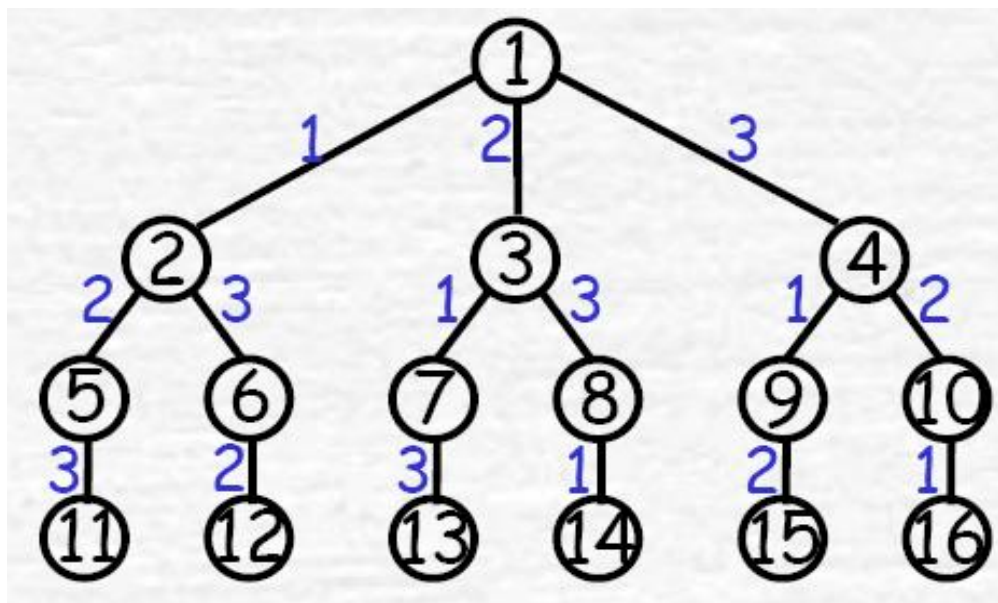
□ 排列树回溯法

```
Backtrack(int t)  //搜索到树的第t层
{  //由第t层向第t+1层扩展, 确定x[t]的值
  if t>n then output(x);  //叶子结点是可行解
  else
    for i=t to n  do
    {
      swap(x[t], x[i]);
      if( Constraint(t) and Bound(t) )
        Backtrack(t+1);
      swap(x[t], x[i]);
    }
}
```


排列生成问题

- **问题定义**：给定正整数 n ，生成 $1, 2, \dots, n$ 所有排列。
- **解空间树(排列树)**：

当 $n=3$ 时，



排列生成问题

● 回溯算法

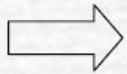
```
Backtrack(int t)
{
    if t>n then output(x);
    else
        for i=t to n do
            { swap(x[t], x[i]);
              Backtrack(t+1);
              swap(x[t], x[i]);
            }
}

main(int n)
{
    for i=1 to n do x[i]=i;
    Backtrack(1);
}
```

● 对n=3的执行情况验证:

```
- Backtrack(1)
  Backtrack(2)
    x[1]↔x[2]
    Backtrack(2)
      x[1]↔x[2]
      x[1]↔x[3]
      Backtrack(2)
        x[1]↔x[3]
        - Backtrack(2)
          Backtrack(3)
            x[2]↔x[3]
            Backtrack(3)
              x[2]↔x[3]
              - Backtrack(3)
                Backtrack(4)
                  - Backtrack(4)
                    output(x)
```

123
132
213
231
321
312



TSP问题

- **问题描述：**略
- **基本思想：**利用排列生成问题的回溯算法Backtrack(2)，对 $x[] = \{1, 2, \dots, n\}$ 的 $x[2..n]$ 进行全排列，则 $(x[1], x[2])$, $(x[2], x[3])$, \dots , $(x[n], x[1])$ 构成一个回路。在全排列算法的基础上，进行路径计算保存以及进行限界剪枝。
- ```
main(int n)
{
 a[n][n]; x[n] = {1,2,...,n}; bestx[]; cc=0.0;
 bestv = ∞; // bestx保存当前最佳路径, bestv保存当前最优值
 input(a); // 输入邻接矩阵
 TSPBacktrack(2);
 output(bestv, bestx[]);
}
```

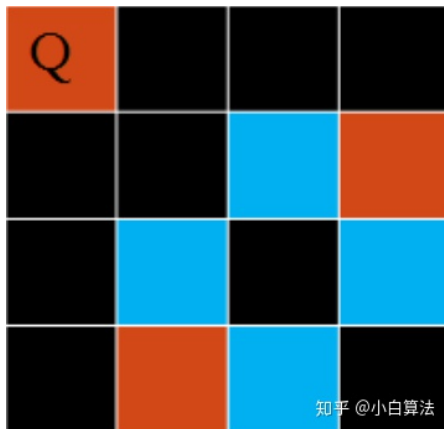
# TSP问题

TSPBacktrack( int i)

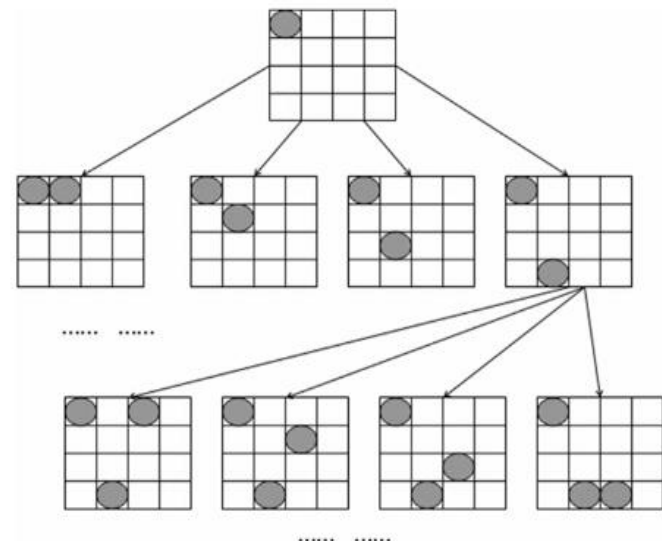
```
{ // cc记录(x[1], x[2]), ..., (x[i-1], x[i]) 的距离和
 if (i>n){ // 搜索到叶结点, 输出可行解与当前最优解比较
 if (cc + a[x[n]][1] < bestv or bestv = ∞) {
 bestv = cc + a[x[n]][1];
 for(j=1 ; j <= n; j++) bestx[j] = x[j];
 }
 }
 else{
 for(j = i ; j<=n; j++)
 if (cc + a[x[i-1]][x[j]] < bestv or bestv = ∞){ // 限界裁剪子树
 swap(x[i], x[j]);
 cc += a[x[i-1]][x[i]];
 TSPBacktrack(i+1);
 cc -= a[x[i-1]][x[i]];
 swap(x[i],x[j]);
 }
 }
}
```

# n皇后问题

- **八皇后问题**是一个古老而著名的问题，是回溯算法的典型例题。  
该问题是十九世纪著名的数学家高斯1850年提出：在8\*8格的国际象棋上摆放八个皇后/棋子，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。
- **四皇后问题描述**：在4\*4棋盘上放上4个皇后，使皇后彼此不受攻击，即条件是彼此不在同行(列)、斜线上。求出全部的放法。



知乎 @小白算法



China

# n皇后问题

- **四皇后问题描述：**在4\*4棋盘上放上4个皇后，使皇后彼此不受攻击，即条件是彼此不在同行(列)、斜线上。求出全部的放法。

- **解表示：**

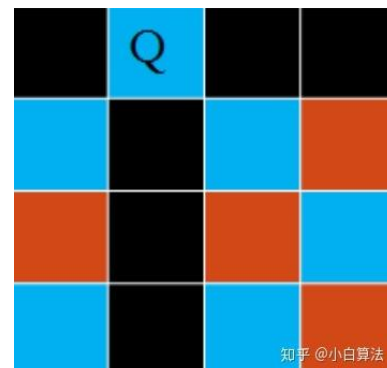
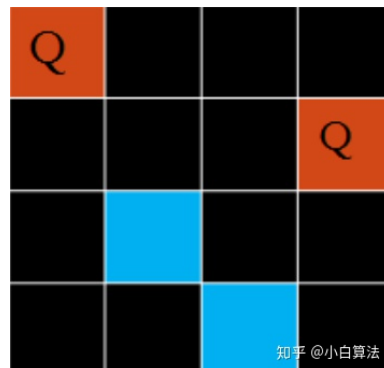
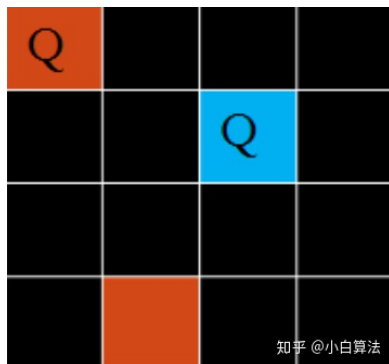
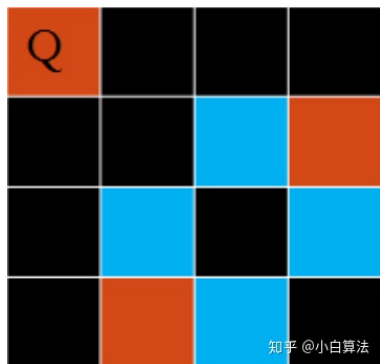
— 解编码： $(x_1, x_2, x_3, x_4)$  4元组， $x_i$ 表示皇后 $i$ 放在 $i$ 行上的列号，如(3,1,2,4)

— 解空间： $\{(x_1, x_2, x_3, x_4) \mid x_i \in S, i=1\sim 4\}$   $S=\{1,2,3,4\}$

可行解满足：

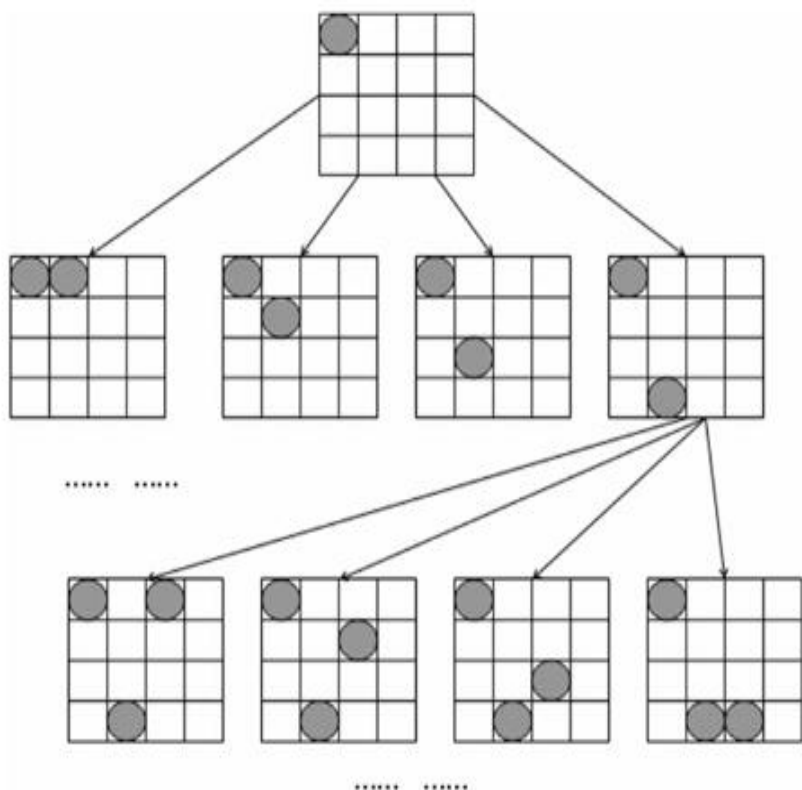
显约束： $x_i \in S, i=1\sim 4$

隐约束( $i \neq j$ ):  $\begin{cases} x_i \neq x_j & \text{(不在同一列)} \\ |x_i - x_j| \neq |i - j| & \text{(不在同一斜线)} \end{cases}$

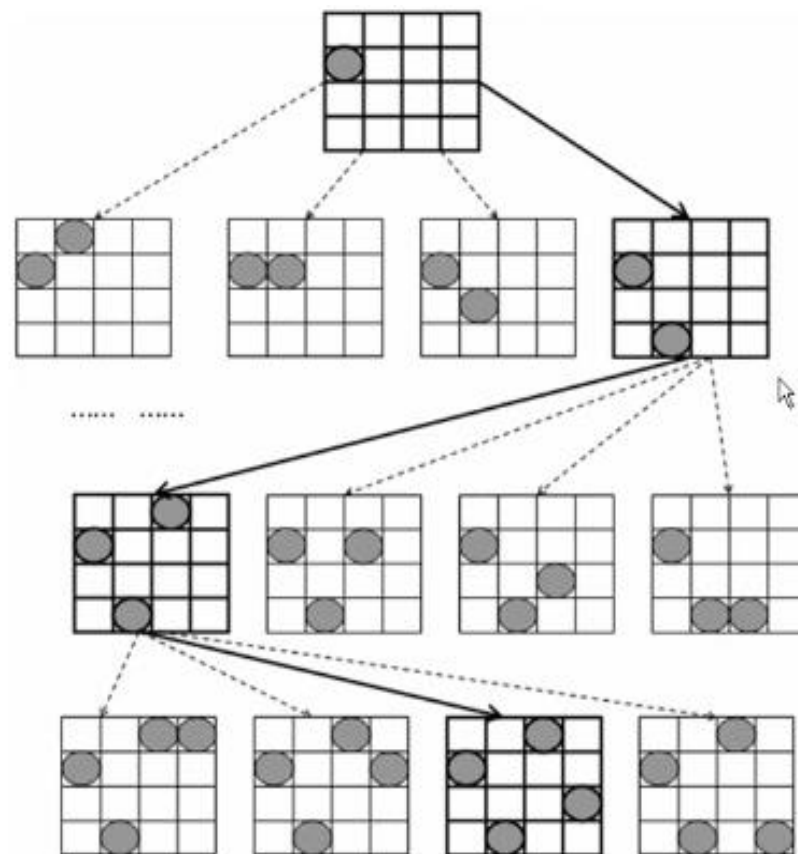




# n皇后问题



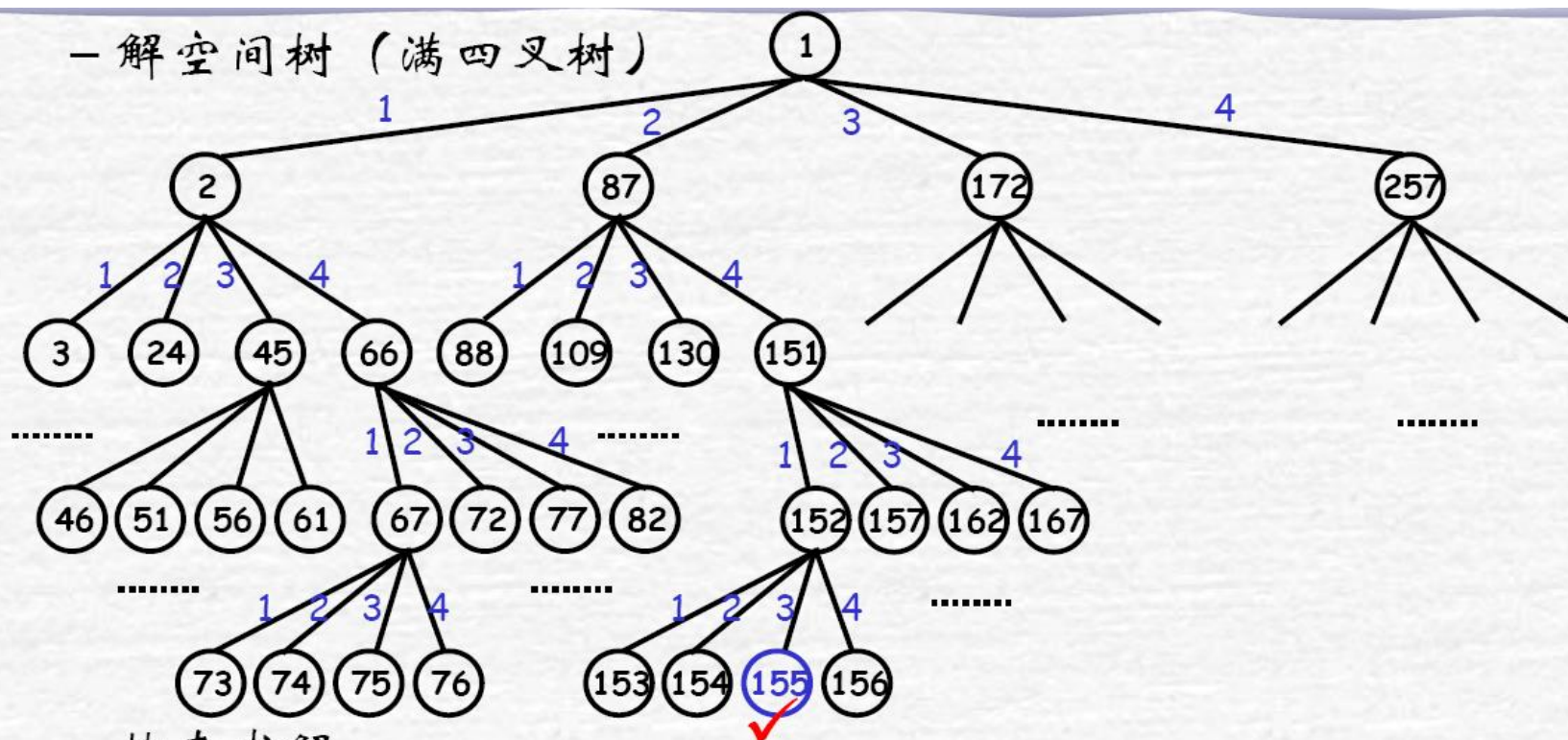
共有  $4 \times 64 = 256$  种局面



一个合法的解的过程（虚线代表排除，黑实线代表继续探索）使用回溯法只需探索第4层中的4个节点，而使用穷举法要探索完第4层的所有64个节点

# n皇后问题

— 解空间树 (满四叉树)



— 搜索求解:

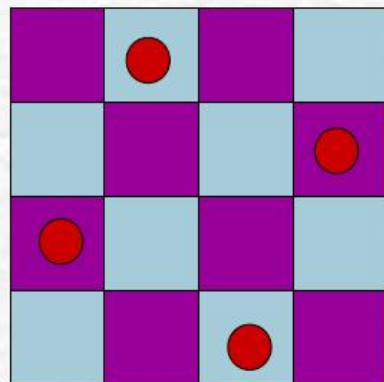
从①起按DFS搜索, 搜索时应满足隐约束, 搜索到叶结点输出解: (2,4,1,3), (3,1,4,2)



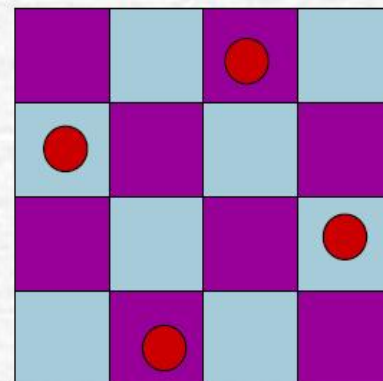
# n皇后问题

输出解:

(2,4,1,3)



(3,1,4,2)



一解编码:  $(x_1, x_2, x_3, x_4)$  4元组,  $x_i$  表示皇后  $i$  放在  $i$  行上的列号, 如  $(3,1,2,4)$

一解空间:  $\{(x_1, x_2, x_3, x_4) \mid x_i \in S, i=1\sim 4\}$   $S=\{1,2,3,4\}$

可行解满足:

显约束:  $x_i \in S, i=1\sim 4$

隐约束( $i \neq j$ ):  $\begin{cases} x_i \neq x_j & \text{(不在同一列)} \\ |x_i - x_j| \neq |i - j| & \text{(不在同一斜线)} \end{cases}$

# n皇后问题

- 递归算法

NQueen(int k)

*{//由第K层向第K+1层扩展, 确定x[k]的值*

if k>n then printf(x[1], ... ,x[n]); *//搜索到叶结点输出解*

else

for i=1 to n do

{ x[k]=i;

if placetest(k) then NQueen(k+1);

}

}

Placetest(int k)

*{//检查x[k]位置是否合法*

for i=1 to k-1 do

if ( x[i]=x[k] or abs(x[i]-x[k])=abs(i-k) ) then return false;

return true;

}

注: 求解时执行NQueen(1)

# n皇后问题

```
bool is_ok(int row){ //判断设置的皇后是否在同一行，同一列，或者同一斜线上
 for (int j=0;j<row;j++)
 {
 if (queen[row]==queen[j] || row-queen[row]==j-queen[j] || row+queen[row] ==
j+queen[j])
 return false;
 }
 return true;
}
```

```
void back_tracking(int row=0) //回溯算法函数，从第0行开始遍历
{
 if (row==n)
 t++; //判断若遍历完成，就进行计数
 for (int col=0; col<n; col++) //遍历棋盘每一列
 {
 queen[row] = col; //将皇后的位置记录在数组
 if (is_ok(row)) //判断皇后的位置是否有冲突
 back_tracking(row+1); //递归，计算下一个皇后的位置
 }
}
```

# 0-1背包问题与 n皇后问题

- **解空间：子集树与排列树。** 例如 $n=8$ 时， $X = (4, 6, 8, 2, 7, 1, 3, 5)$ 表示第一行的皇后横坐标为4，第二行的皇后横坐标为6，.....常用的求解办法基于子集树求解，约束条件为 $\text{abs}(k-j) \neq \text{abs}(x[j]-x[k])$ 并且 $x[j] \neq x[k]$ 。该问题也可以描述为n个数字的排列问题，约束条件为 $\text{abs}(k-j) \neq \text{abs}(x[j]-x[k])$ ，即基于全排列的算法求解。
- **可行性约束函数**
- 与贪婪法所解决的背包问题相比，回溯法更能顾及寻找全局最优。
- 背包问题与八皇后问题均能使用回溯法，但是目的不同。八皇后要求将所有的棋子放在棋盘上(即只需解决深度最优)，0-1背包问题不仅把物品放进背包，而且要背包内物品的总价值最大，比八皇后问题**多了一个限制**。

# 符号三角形问题

下图是由14个“+”和14个“-”组成的符号三角形。2个同号下面都是“+”，2个异号下面都是“-”。

```

+ + - + - + +
 + - - - - +
 - + + + -
 - + + -
 - + -
 - -
 +

```

在一般情况下，符号三角形的第一行有 $n$ 个符号。符号三角形问题要求对于给定的 $n$ ，计算有多少个不同的符号三角形，使其所含的“+”和“-”的个数相同。

# 符号三角形问题

- **解向量**：用 $n$ 元组 $x[1:n]$ 表示符号三角形的第一行， $x[i]=1$ 表示符号三角形的第一行第 $i$ 个符号为“+”， $x[i]=0$ 表示符号三角形的第一行第 $i$ 个符号为“-”。
- **可行性约束函数**：当前符号三角形所包含的“+”个数与“-”个数均不超过 $n*(n+1)/4$
- **无解的判断**： $n*(n+1)/2$ 为奇数

+ + - + - +  
+ - - - -  
- + + +  
- + +  
- +  
-  
-

+ + - + - + +  
+ - - - - +  
- + + + -  
- + + -  
- + -  
- -  
+

# 符号三角形问题

```
void Triangle::Backtrack(int t)
```

```
{ // 是一棵子集树, t控制着第一行符号个数, 对应于树的层次
```

```
 if ((count>half)||((t*(t-1)/2-count>half)) return; // 剪枝法, 剪除不必要的子树
```

```
 if (t>n) sum++; // 到叶子结点, " + " 号数和 " - " 号数相同的符号三角形个数增加1
```

```
 else
```

```
 for (int i=0; i<2; i++) { // 当前扩展结点Z只有两种可能的取值0或1, 即只可能有2个孩子
```

```
 p[1][t] = i; // 二维数据p记录了符号三角形矩阵
```

```
 count += i; // 当前符号三角形矩阵中, " + " 号的个数
```

```
 for (int j=2; j<=t; j++){ // 从第二行起计算 " + " 号个数, 计算可行性约束
```

```
 p[j][t-j+1] = p[j-1][t-j+1]^p[j-1][t-j+2];
```

```
 count += p[j][t-j+1];
```

```
 }
```

```
 Backtrack(t+1); // 扩展到第t+1层
```

```
 for (int j=2; j<=t; j++) // 必须回退! 因为外层for循环是对当前扩展结点两个子树分别搜索!
```

```
 count-=p[j][t-j+1];
```

```
 count-=i;
```

```
 }
```



# 符号三角形问题

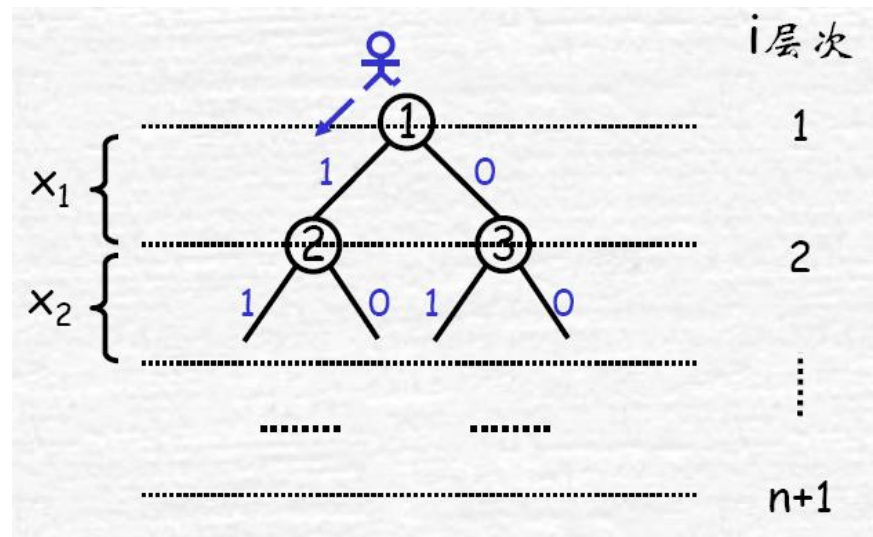
## □ 复杂度分析：

计算可行性约束需要 $O(n)$ 时间，在最坏情况下有个 $O(2^n)$ 结点需要计算可行性约束，因此，基于回溯法求解符号三角形问题，其时间复杂度为 $O(n2^n)$ 。

?? 如何提升算法性能，降低复杂度

# 0-1背包问题

- 问题描述：略
- 解表示和解空间：  $\{ (x_1, x_2, \dots, x_n) \mid x_i \in \{0, 1\}, i=1 \sim n \}$
- 解空间树：



# 0-1背包问题

## □ 无限界函数的算法

KnapBacktrack(int i)

{ //cw当前背包重量, cv当前背包价值, bestv当前最优价值

if(i > n) { //搜索到可行解

bestv = (bestv < cv)? cv : bestv;

output(x);

}

else{

if (cw + w[i] <= c) { //走左子树

x[i] = 1; cw += w[i]; cv += v[i];

KnapBacktrack(i+1);

cw -= w[i]; cv -= v[i];

}

//以下走右子树

x[i] = 0;

KnapBacktrack(i+1);

}

}

main(float c, int n, float w[], float v[], int x[])

{ //主程序

float cw=0.0, cv = 0.0, bestv = 0.0;

KnapBacktrack(1);

}

# 0-1背包问题

## □ 有限界函数的算法

### — 基本思想：

- 设 $r$ 是当前扩展结点 $Z$ 的右子树(或左子树)价值上界, 如果 $cv+r \leq bestv$ 时, 则可以裁剪掉右子树(或左子树)。
- 一种简单的确定 $Z$ 的左、右子树最优值上界的方法(设 $Z$ 为第 $k$ 层结点):

$$\text{左子树上界} = \sum_{i=k}^n v_i, \text{右子树上界} = \sum_{i=k+1}^n v_i$$

### — 求经扩展结点 $Z$ 的可行解价值上界的方法：

- 计算至扩展结点的当前背包价值  
已知 $x_i, i = 1 \sim k-1$ , 当前背包价值 $cv = \sum_{i=1}^{k-1} v_i x_i$
- 最后,  
经 $Z$ 左子树的可行解价值上界 =  $cv + \text{左子树上界}$   
经 $Z$ 右子树的可行解价值上界 =  $cv + \text{右子树上界}$

### — 算法(略)

# 图的m着色问题

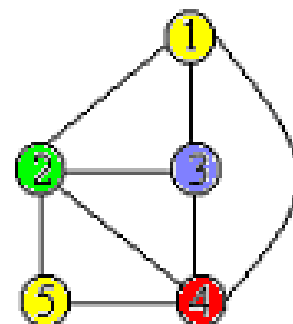
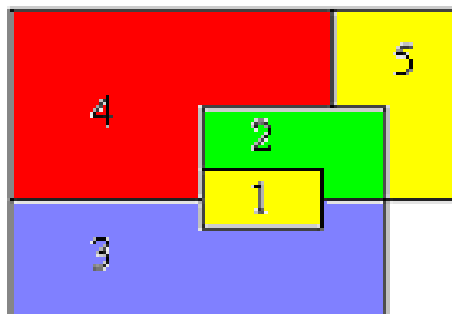
- 问题描述:

给定无向连通图 $G$ 和 $m$ 种不同的颜色。用这些颜色为图 $G$ 的各顶点着色，每个顶点着一种颜色。是否有一种着色法使 $G$ 中每条边的2个顶点着不同颜色。这个问题是图的 $m$ 可着色判定问题。若一个图最少需要 $m$ 种颜色才能使图中每条边连接的2个顶点着不同颜色，则称这个数 $m$ 为该图的颜色数。求一个图的颜色数 $m$ 的问题称为图的 $m$ 可着色优化问题。

- 四色猜想:

在一个平面或球面上的任何地图能够只用4种颜色来着色，使相邻的国家在地图上着不同的颜色。假设每个国家在地图上单连通区域，两个国家相邻则其边界长度不为0。

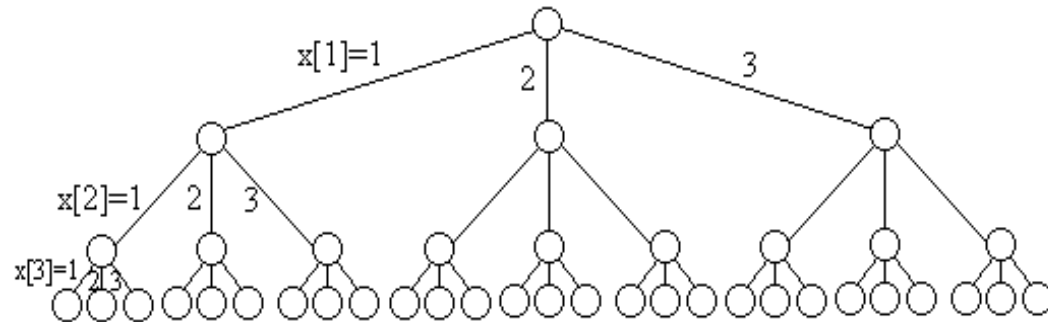
右图是一个有5个区域的地图及其相应的平面图。



# 图的m着色问题

- **解向量**:  $(x_1, x_2, \dots, x_n)$  表示顶点  $i$  所着颜色  $x[i]$ ;
- **可行性约束函数**: 顶点  $i$  与已经着色的相邻顶点颜色不重复。

```
void Color::Backtrack(int t)
{
 if (t > n) {
 sum++;
 for (int i = 1; i <= n; i++)
 cout << x[i] << ' ';
 cout << endl;
 }
 else
 for (int i = 1; i <= m; i++) {
 x[t] = i;
 if (Ok(t)) Backtrack(t + 1);
 }
}
```



```
bool Color::Ok(int k)
{ // 检查颜色可用性
 for (int j = 1; j <= n; j++)
 if ((a[k][j] == 1) && (x[j] == x[k])) return false;
 return true;
}
```



# 图的m着色问题

## □ 算法复杂度分析:

图m可着色问题的解空间树中内结点个数是  $\sum_{i=0}^{n-1} m^i$ 。对于每一个内结点，在最坏情况下，用ok检查当前扩展结点的每一个儿子所相应的颜色可用性需耗时 $O(mn)$ 。因此，回溯法总的时间耗费是  $\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1) / (m - 1) = O(nm^n)$

# 回溯法效率分析

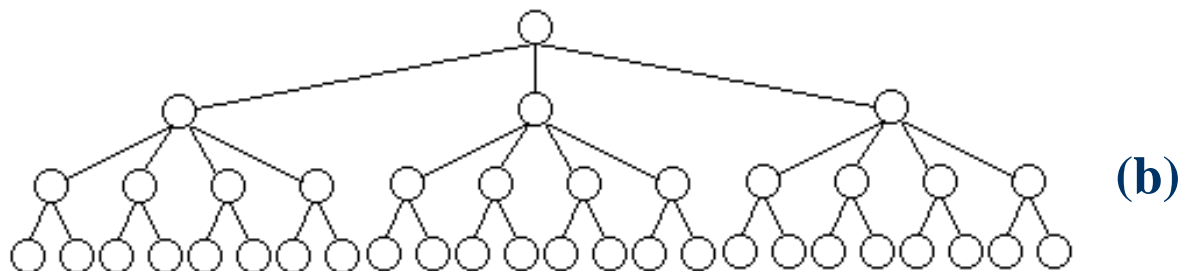
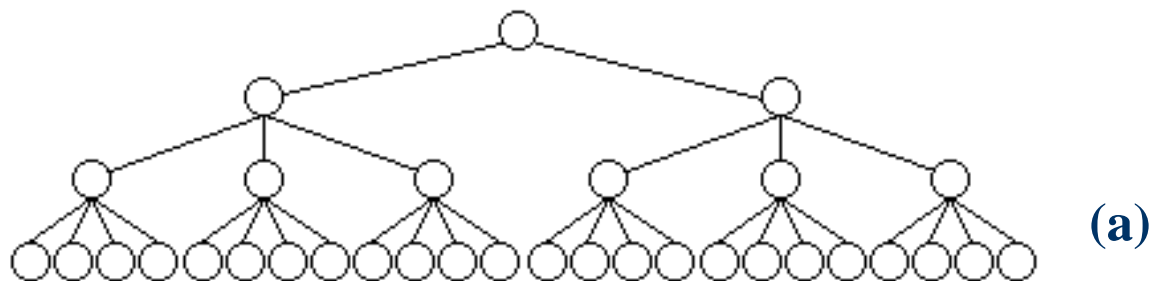
通过上述实例的讨论，回溯法的效率在很大程度上依赖于以下因素：

- (1) 产生 $x[t]$ 的时间；
- (2) 满足显约束的 $x[t]$ 值的个数；
- (3) 计算约束函数constraint的时间；
- (4) 计算上界函数bound的时间；
- (5) 满足约束函数和上界函数约束的所有 $x[k]$ 的个数。

好的约束函数能显著地减少所生成的结点数，但这样的约束函数往往计算量较大。因此，在选择约束函数时通常存在生成结点数与约束函数计算量之间的折衷。

# 回溯法效率分析

- 对于许多问题而言，在搜索试探时选取 $x[i]$ 的值顺序是任意的。在其它条件相当的前提下，让可取值最少的 $x[i]$ 优先。从图中关于同一问题的2棵不同解空间树，可以体会到这种策略的潜力。



图(a)中，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组。对于图(b)，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果明显比后者好。

# 谢谢！

## Q & A

作业：

1. 用回溯法从 $n$ 个不同元素中取 $r$ 个不重复的元素的所有组合。
2. 基于回溯法，求解TSP问题 或 0-1背包问题