算法与计算理论

课程内容

数据结构

概述	数组与广义表	图	外部排序
线性表	串	查找	
栈与队列	树	内部排序	

算法与计算理论

概述	贪心	……	计算模型
分治	回溯	……	可计算理论
动态规划	……	……	计算复杂性

Contents

本章内容

- 回溯法的定义和基本思想
- 回溯法的基本框架
- 回溯法的应用场景
- 回溯法的时间复杂度分析
- 回溯法的优缺点分析
- 回溯法的扩展应用

回溯法的定义和基本思想

回溯法(Backtracking)的名字和概念最早由美国计算机科学家 D.W. J. 纳克斯（D. W. J. Nijenhuis）和 A. E. 矢部（A. E. Roy）于1970年的一篇论文中提出。他们将回溯法定义为一种解决组合优化问题的通用方法，通过深度优先搜索的方式遍历解空间树，逐步选择可能的解决方案，并在发现不满足条件的情况下进行回退。

解空间是指问题的所有可能解的集合。

回溯法求解问题的解空间，一般具有这样的特征：解空间由一个典型的树形结构构成，每个节点代表问题的一个部分解，而边则代表求解问题的步骤中的一个决策。

回溯法的定义和基本思想

问题求解是在全部的解空间（每个叶子节点看作是一个解）中找最优解或满足特定条件的解。

回溯法的基本思想实质是一个先序遍历一棵“状态树”的过程，只是这棵树不是预先建立的，而是隐含在遍历过程中。

回溯法的基本框架

回溯法的基本步骤：

(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) 设计约束函数和限界函数

(4) 定义回溯函数，以深度优先方式搜索解空间（先根遍历状态树），并在搜索过程中用剪枝函数避免无效搜索。

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；

用限界函数剪去得不到最优解的子树。

回溯法的基本框架

回溯函数的设计一般包括以下几个步骤：

1.定义回溯函数：用于搜索解空间。回溯函数通常至少含有两个参数：当前状态和当前选择的位置。

2.设定终止条件：在回溯函数中，设定一个终止条件，用于判断是否满足问题的要求。当终止条件满足时，我们可以输出结果或者将结果保存起来。

3.遍历所有可能的选择：在回溯函数中，遍历所有可能的选择。通过循环、递归等方式来实现。对于每个选择，做选择操作，并更新状态，之后继续调用回溯函数进行下一步的搜索。

4.撤销选择和恢复状态：在回溯函数中，当递归调用返回时，需要撤销当前的选择，并恢复状态，以便进行下一次尝试。

整个回溯法的过程就是通过不断尝试选择并回溯的方式，搜索问题的解空间，直到找到满足要求的解或者遍历完所有可能的选择。

回溯法的基本框架

回溯函数的伪代码形式：

```
function backtrack(问题参数, 当前状态):
    if 终止条件满足:
        输出结果
        return
    for 每个可行的选择:
        做出选择
        更新状态
        backtrack(问题参数, 更新后的状态)
        撤销选择
        恢复状态
```

回溯法的应用场景

1.组合优化问题：例如组合、排列、子集等问题。通过尝试所有可能的组合或排列，回溯法可以找到满足条件的最优解。

2.搜索问题：例如迷宫问题、八皇后问题、数独等。通过逐步尝试不同的选择，回溯法可以搜索解空间，找到满足条件的解。

3.图论问题：例如旅行商问题、图的着色问题等。通过遍历图的所有可能路径或者着色方案，回溯法可以找到最优的解决方案。

4.约束满足问题：例如数独、八皇后等。通过逐步尝试不同的值或者状态，回溯法可以找到满足约束条件的解。

需要注意的是，回溯法通常在问题的解空间较小或者剪枝操作较为有效时才能够高效求解。在解空间较大或者剪枝操作不可行的情况下，回溯法可能会产生指数级的时间复杂度。

回溯法的应用场景——组合问题

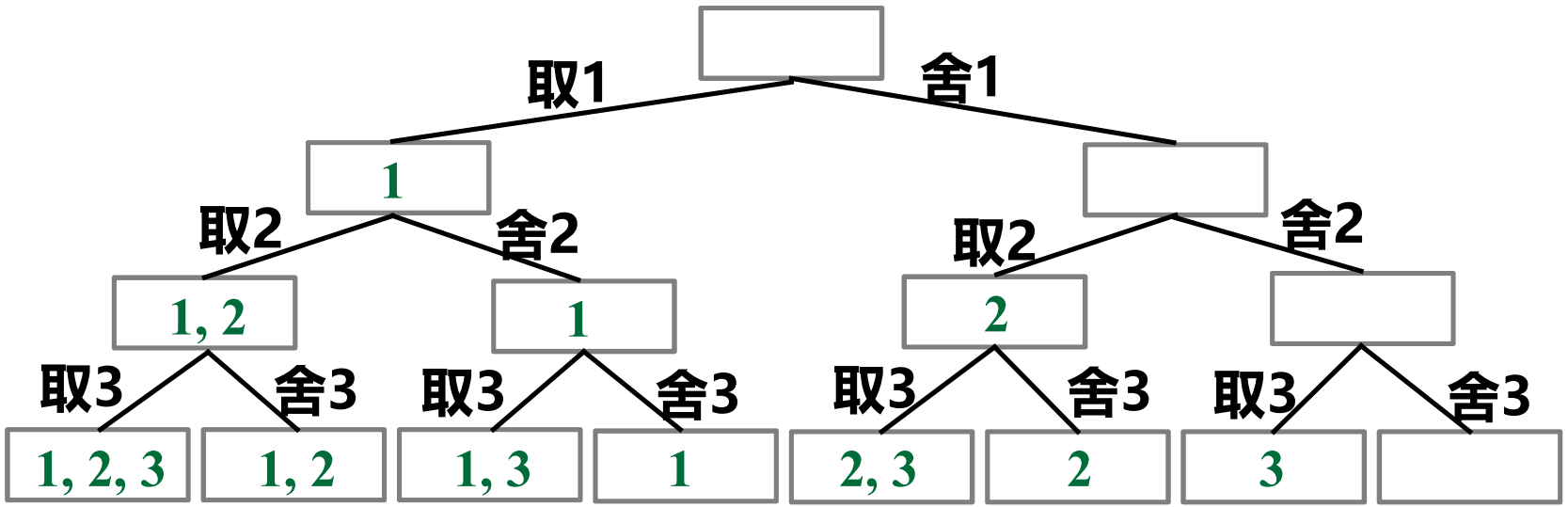
幂集问题：求含n个元素的集合的幂集。如 n = 3， A={1,2,3}
A的幂集为{{1,2,3}，{1,2}，{1,3}，{2,3}，{1}，{2}，{3}，{}}

回溯法的应用场景——组合问题

幂集问题：求含n个元素的集合的幂集。如 n = 3， A={1,2,3}

A的幂集为{{1，2，3}，{1，2}，{1，3}，{2，3}，{1}，{2}，{3}，{}}

解题思路：



问题求解过程：遍历状态树，输出所有叶子节点

回溯法的应用场景——组合问题

定义回溯函数：

```
void GetPowerSet( int i, List A, List &B){
//线性表A表示集合{1,2,3}，线性表B表示幂集ρ（A）的一个元素
//局部变量K为进入函数时B的当前长度，第一次调用本函数时，B为空表，i=1
        if (i > ListLength(A) ) Output(B);//叶子节点
        else{
                GetElem(A, i, x);
                k = ListLength(B);
                ListInsert(B, k+1, x); // “取”第i个元素
                GetPowerSet( i+1, A, B);//搜索左子树
                ListDelete(B, k+1, x);// “舍”第i个元素
                GetPowerSet( i+1, A, B);//搜索右子树
        }//else
}//GetPowerSet
```

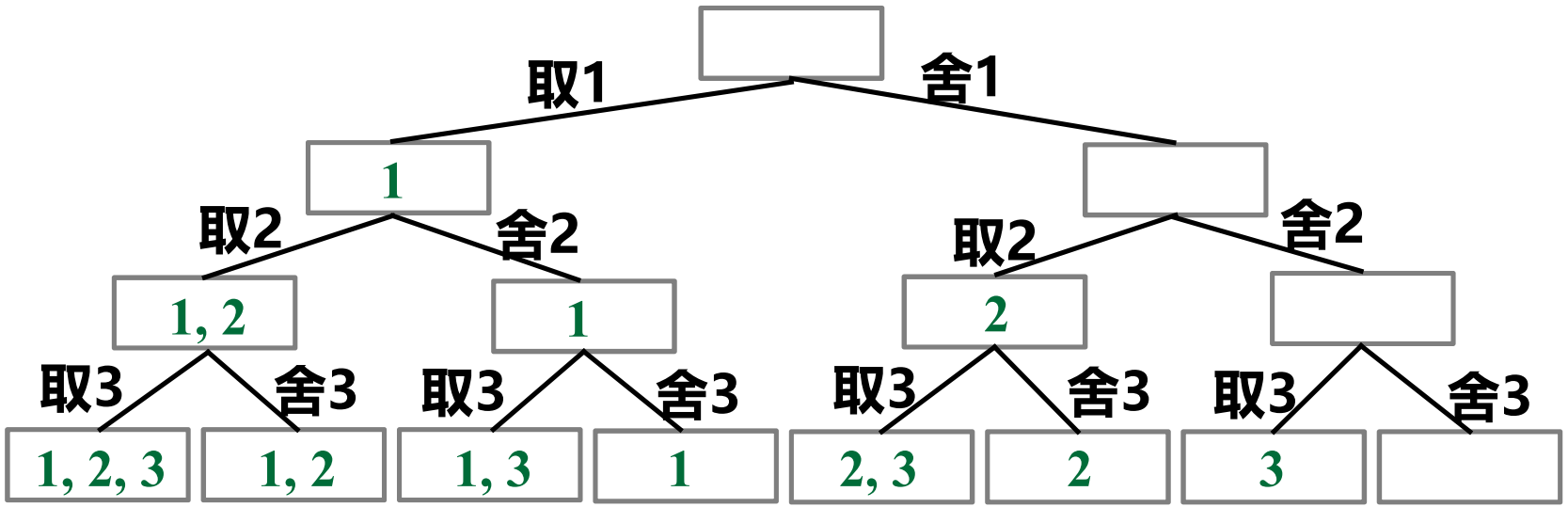
```
GetPowerSet( 1, A, B); // ==> GetPowerSet( 1, [1,2,3], []);
```

回溯法的应用场景——组合问题

幂集问题：求含n个元素的集合的幂集。如 n = 3， A={1,2,3}

A的幂集为{{1，2，3}，{1，2}，{1，3}，{2，3}，{1}，{2}，{3}，{}}

解题思路：



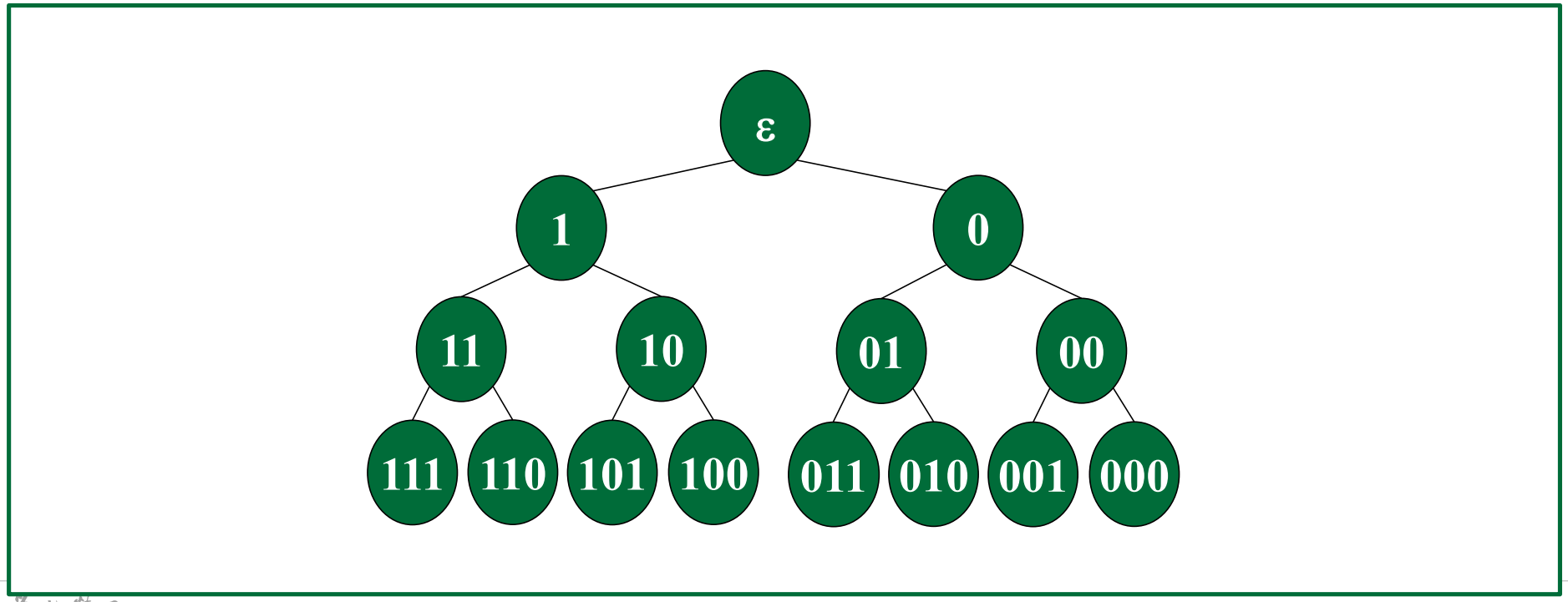
问题求解过程：遍历状态树，输出所有叶子节点

回溯法的应用场景——组合问题

幂集问题：求含n个元素的集合的幂集。如 n = 3， A={1,2,3}

A的幂集为{{1，2，3}，{1，2}，{1，3}，{2，3}，{1}，{2}，{3}，{}}

解题思路：



回溯法的应用场景——组合问题

定义回溯函数：

用x[ ]表示弃取

x=[1,0,1]

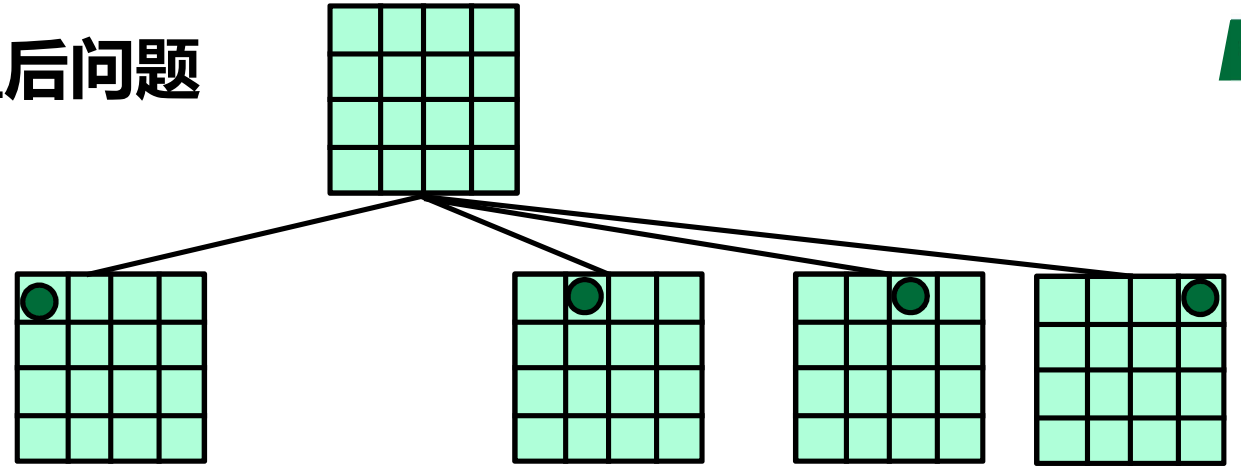
表示集合{1,3}

```
void backtrack( int t, int n){
        if (t > n ) Output(x);//步骤1：输出当前解(叶子节点)
        else{
                x[t] =1;//步骤2.1：“取”第t个元素
                backtrack( t+1, n);//步骤2.2：搜索左子树
                x[t] =0;//步骤3.1：“舍”第t个元素
                backtrack( t+1, n);//步骤3.2：搜索右子树
        }//else
}//backtrack
```

```
backtrack( 1, 3);
```

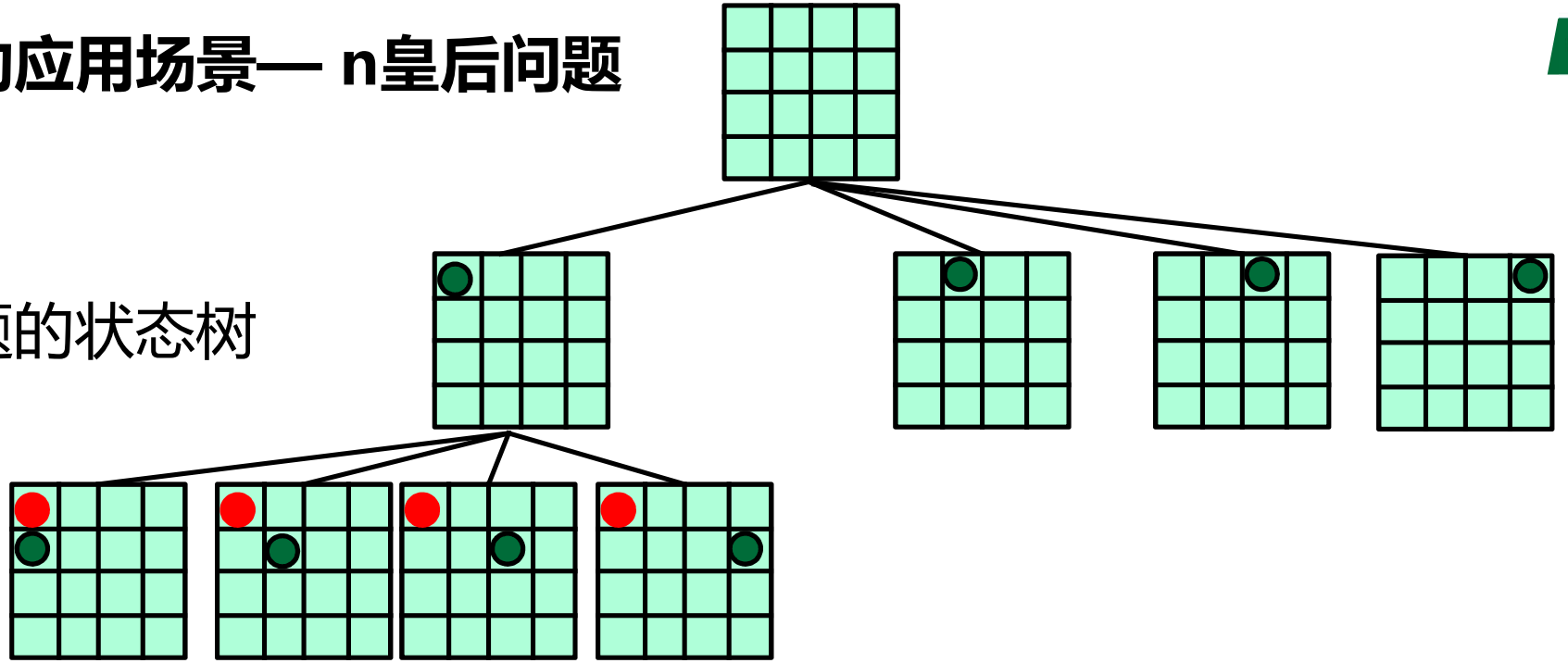
回溯法的应用场景—n皇后问题

4皇后问题的状态树



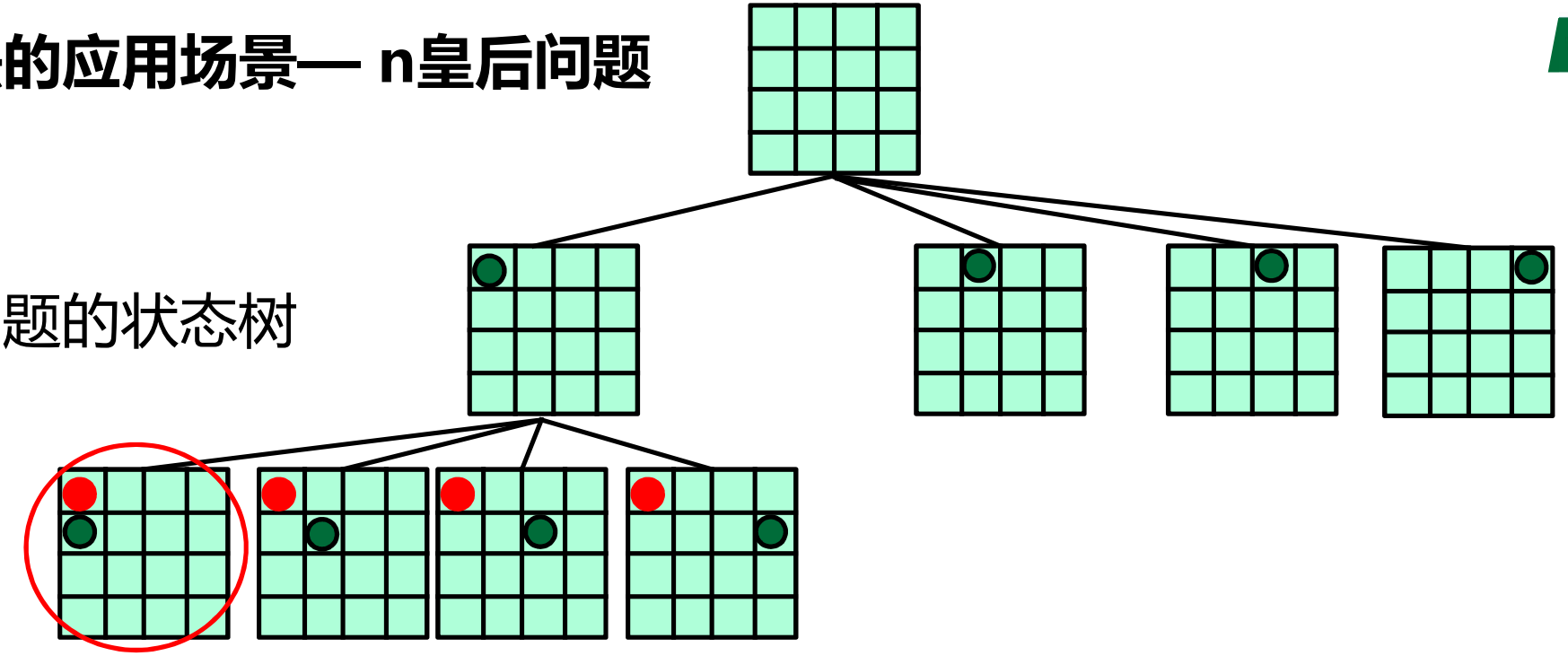
回溯法的应用场景— n皇后问题

4皇后问题的状态树



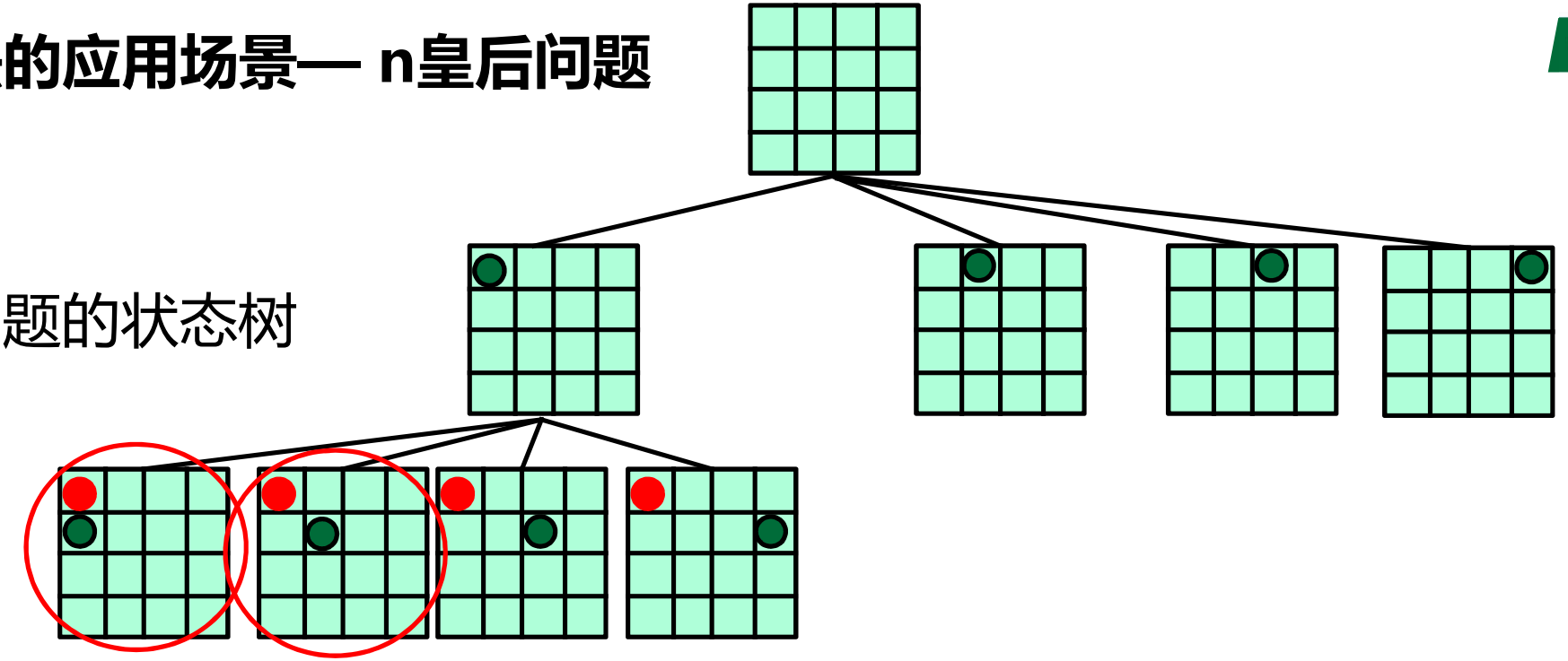
回溯法的应用场景— n皇后问题

4皇后问题的状态树



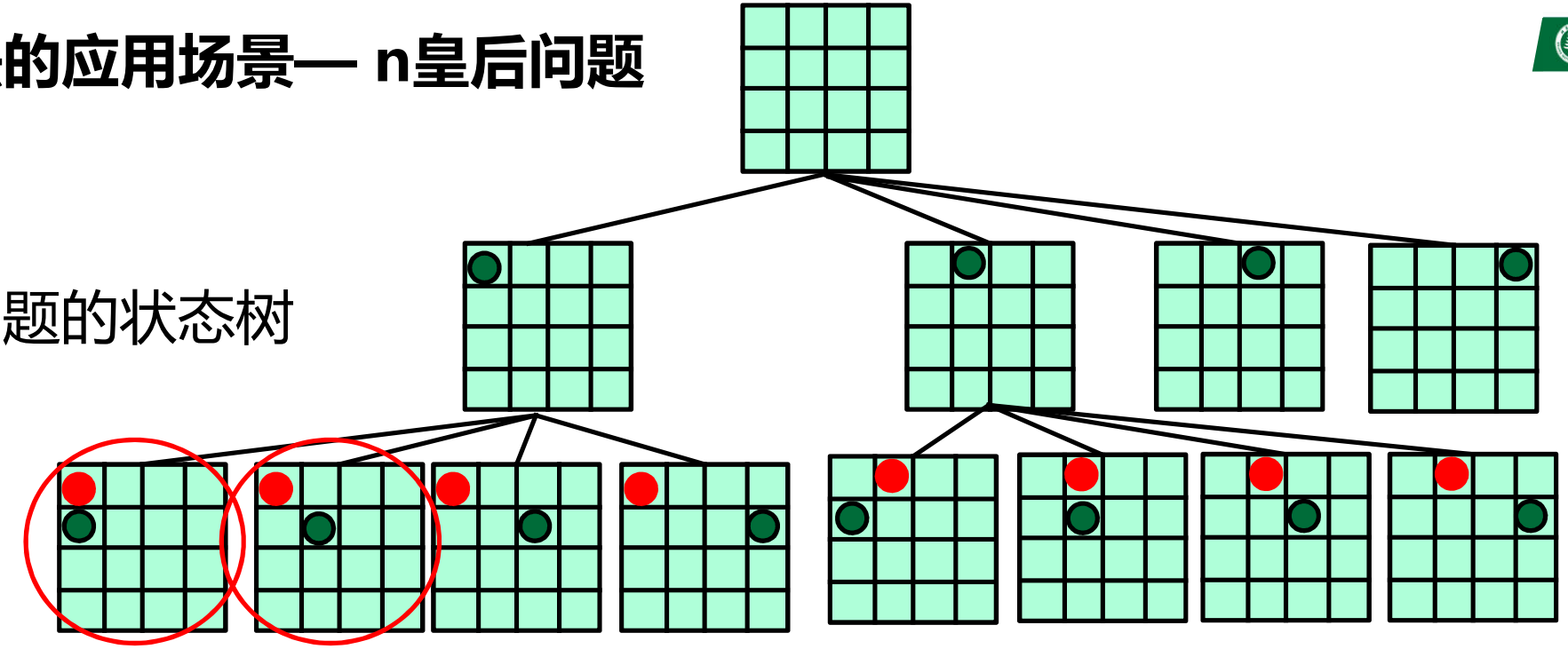
回溯法的应用场景— n皇后问题

4皇后问题的状态树



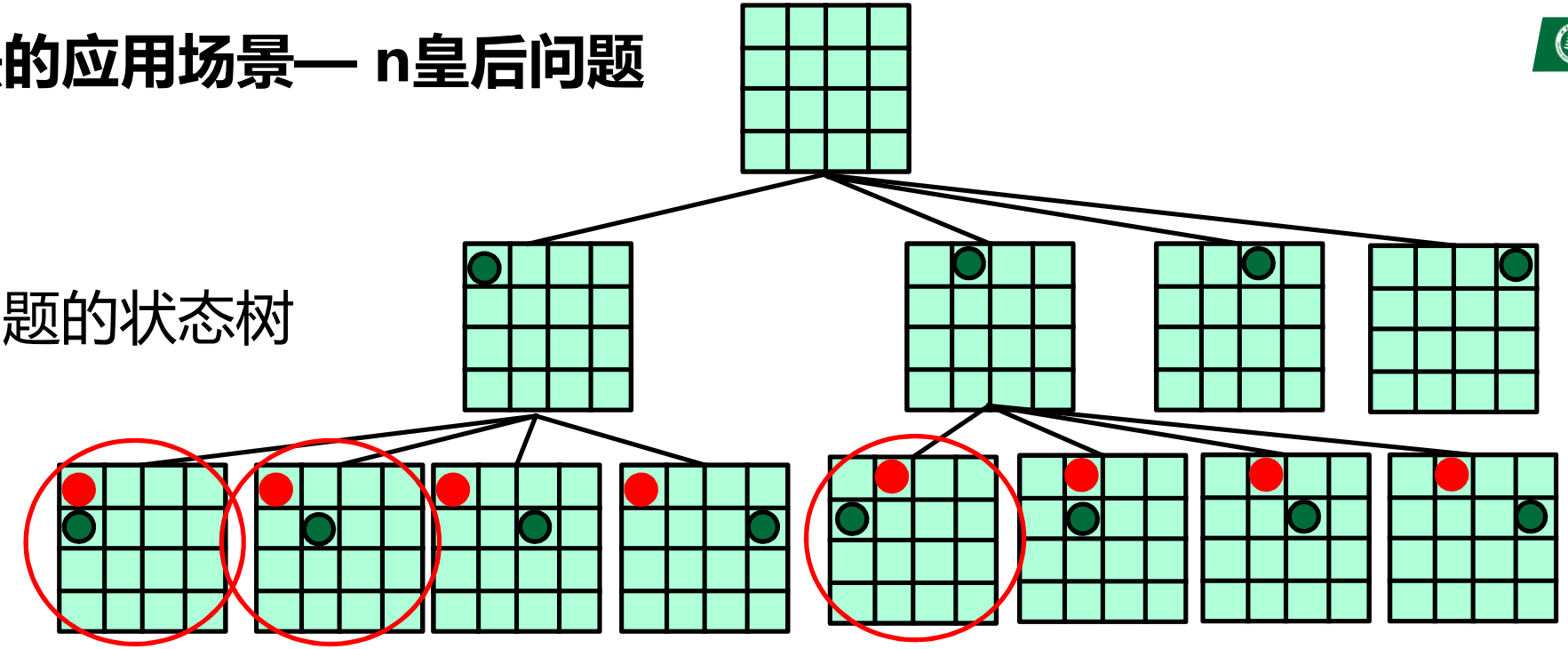
回溯法的应用场景— n皇后问题

4皇后问题的状态树



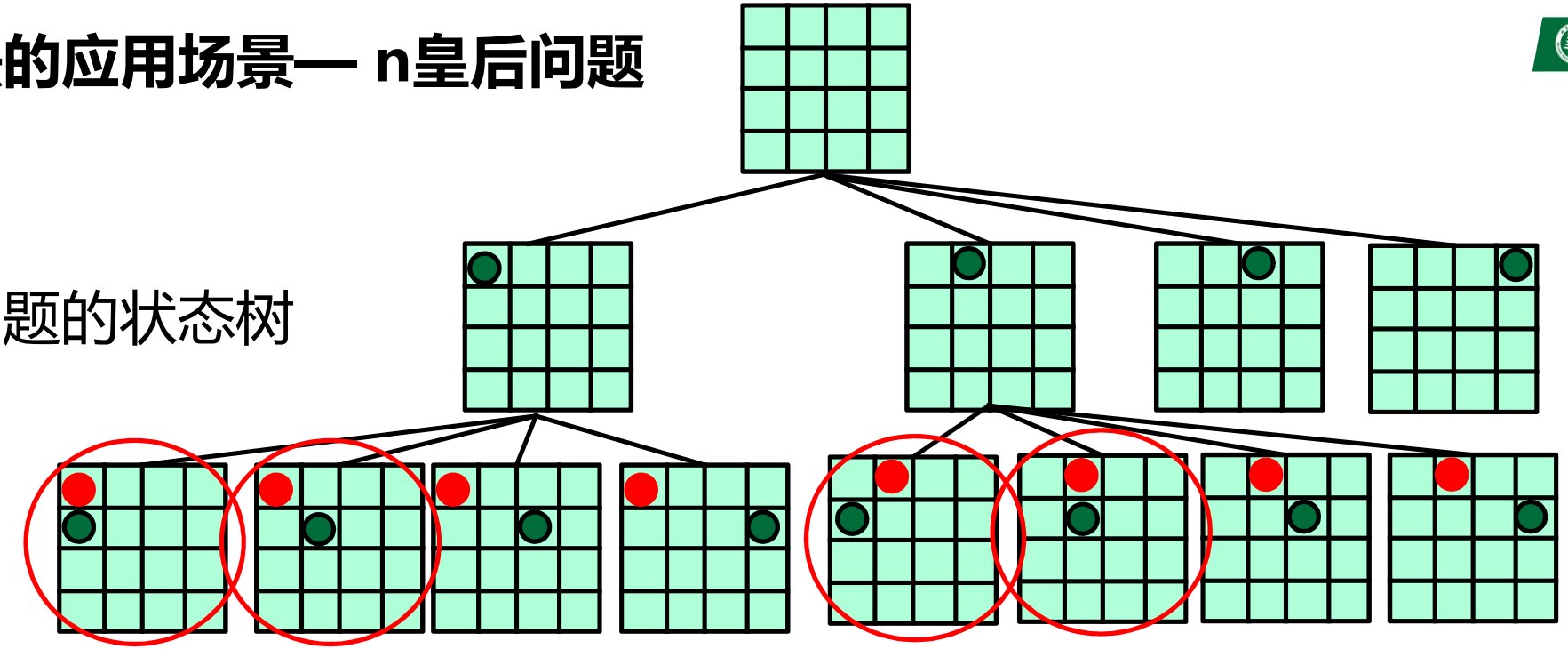
回溯法的应用场景— n皇后问题

4皇后问题的状态树



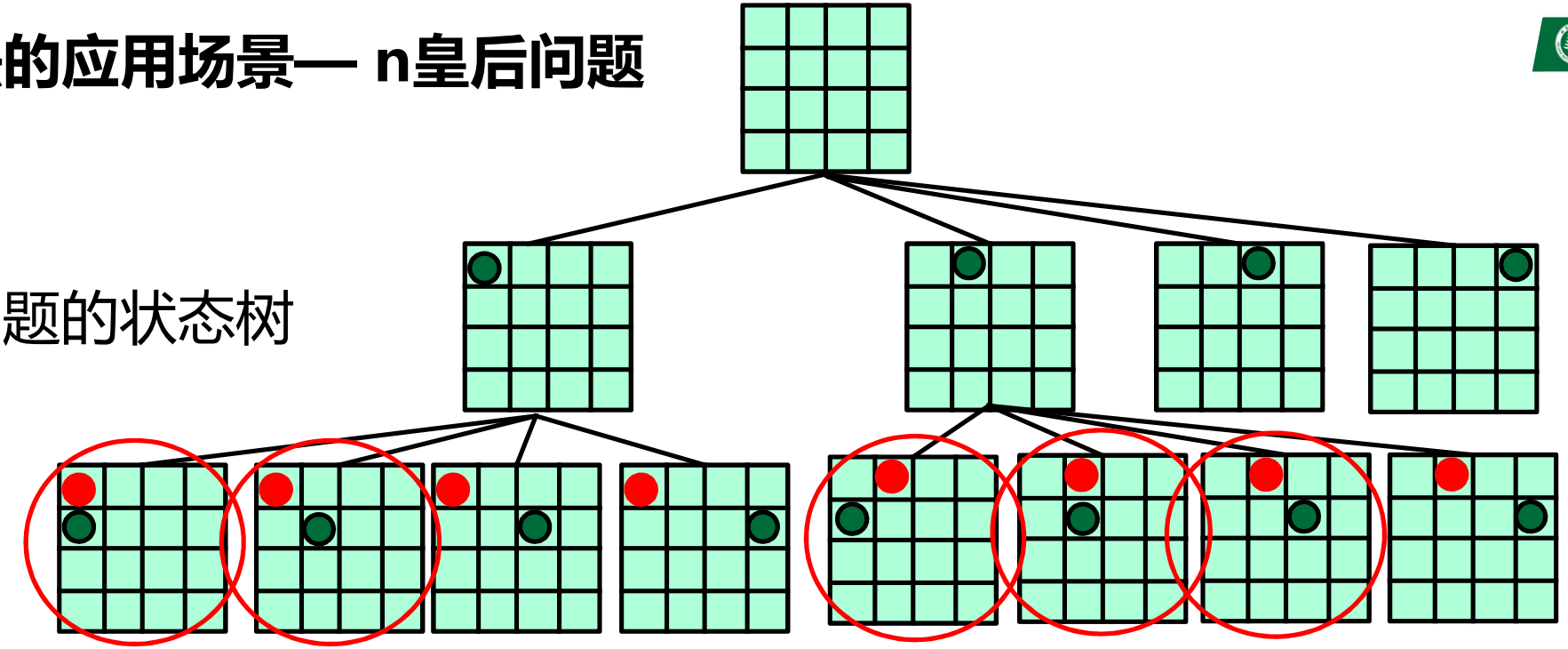
回溯法的应用场景— n皇后问题

4皇后问题的状态树



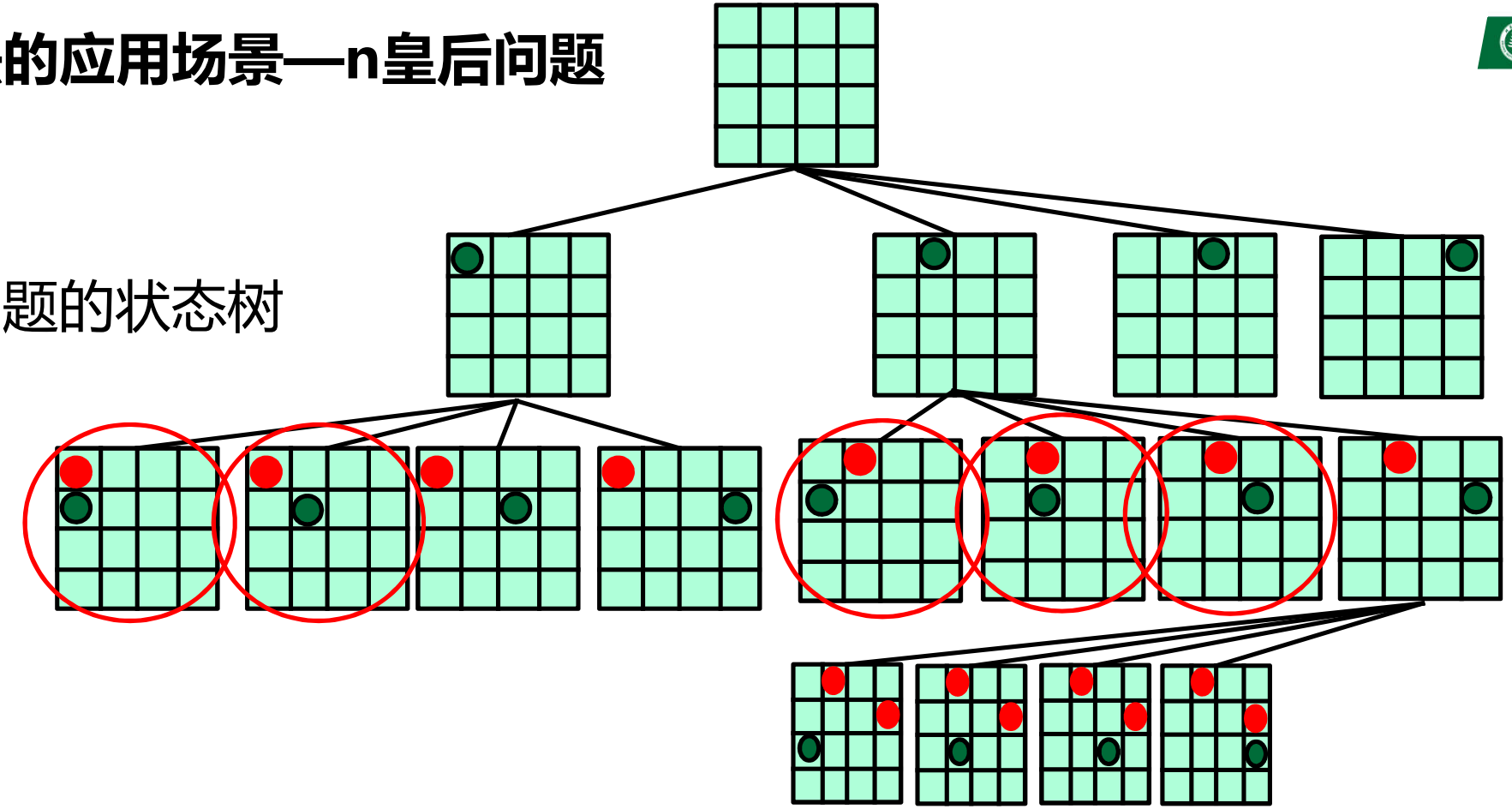
回溯法的应用场景— n皇后问题

4皇后问题的状态树



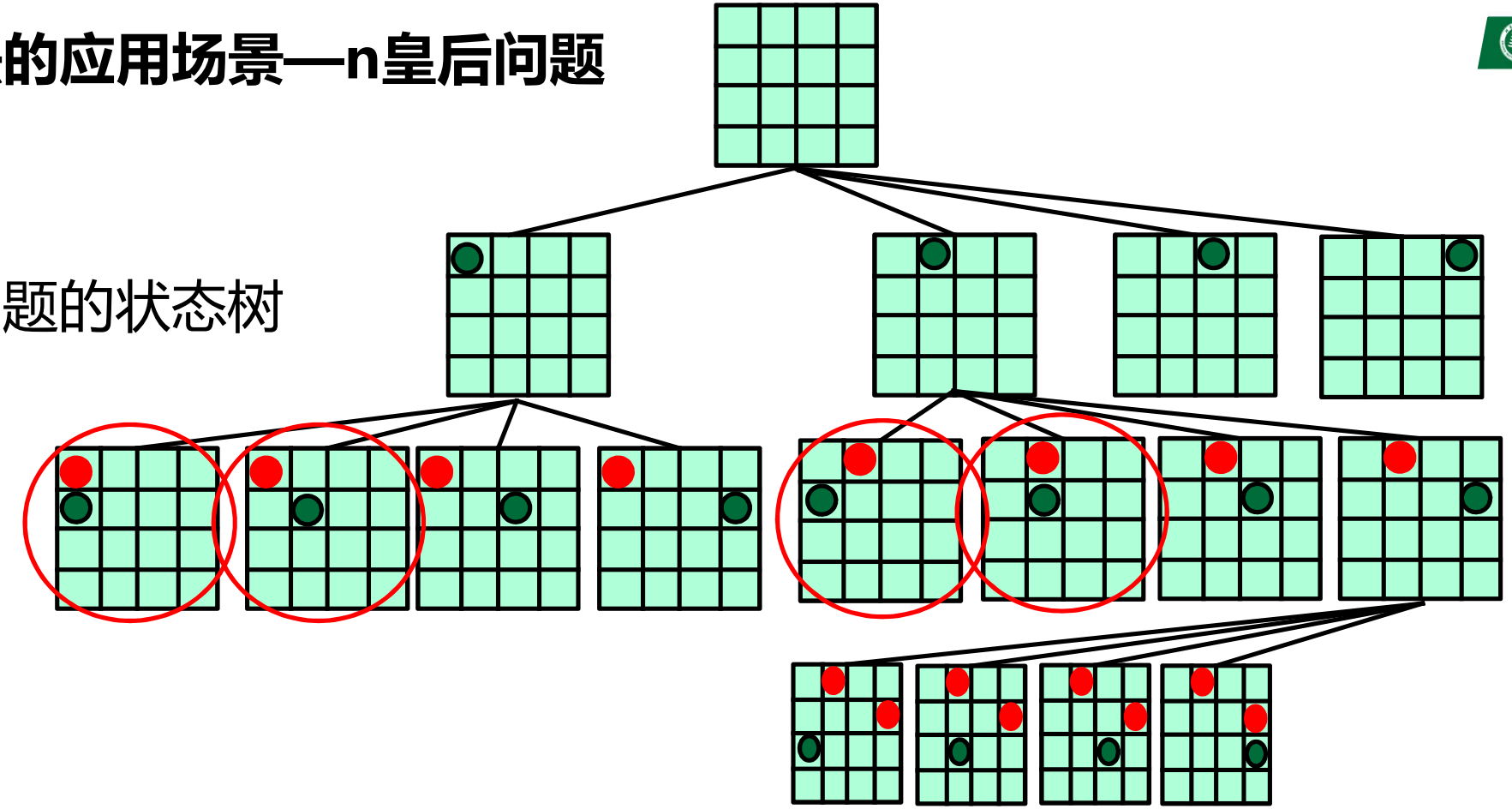
回溯法的应用场景—n皇后问题

4皇后问题的状态树



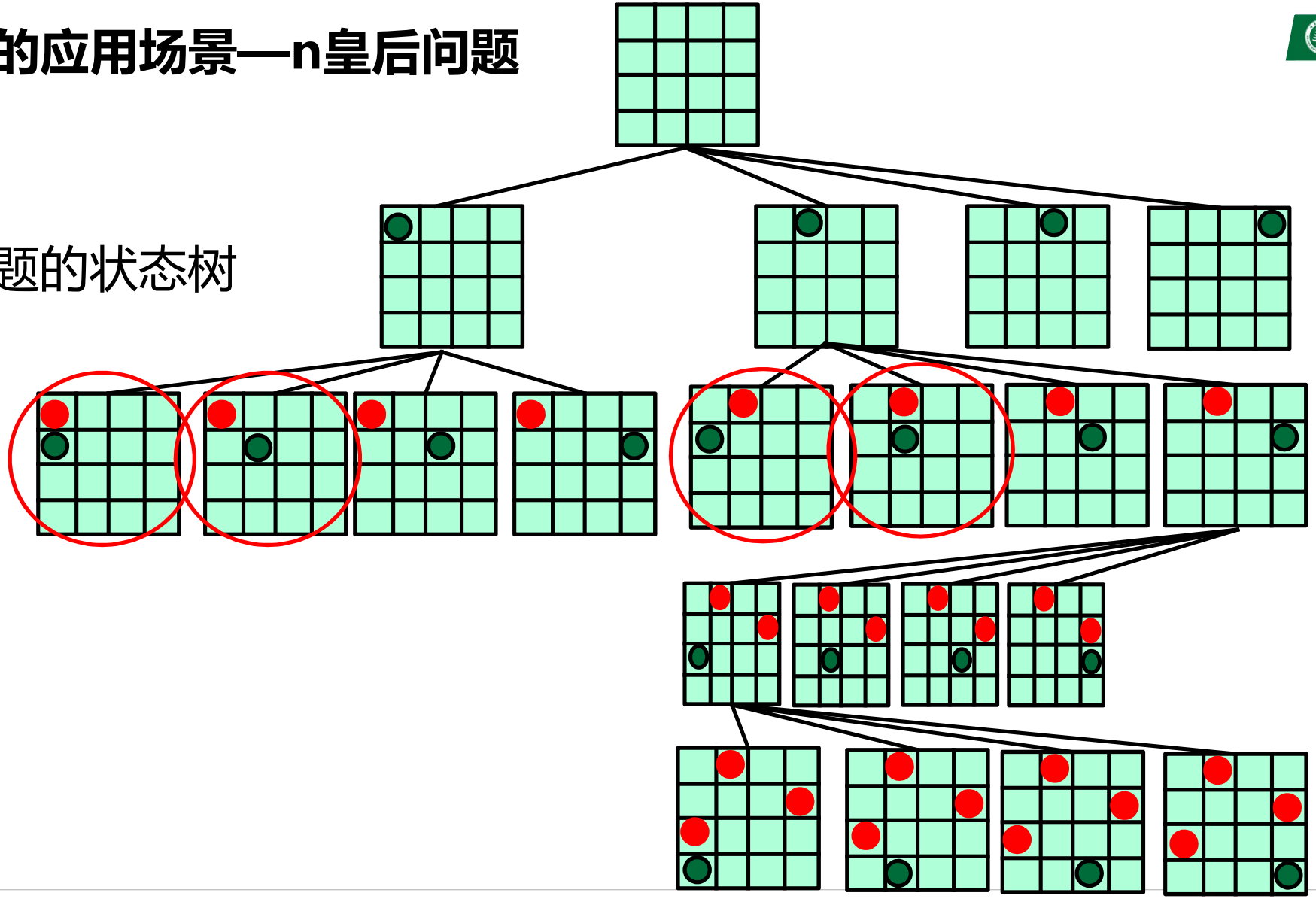
回溯法的应用场景—n皇后问题

4皇后问题的状态树



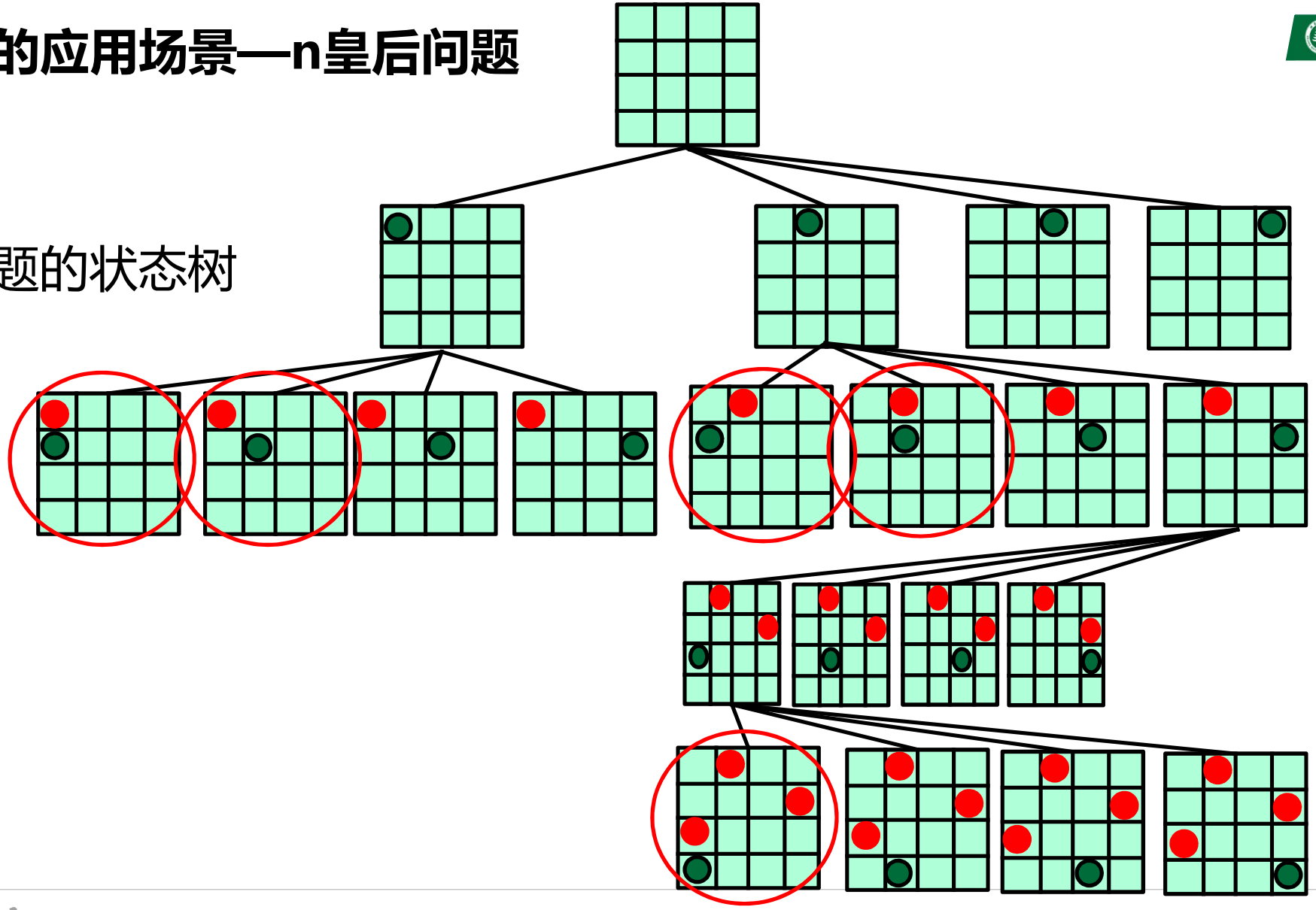
回溯法的应用场景—n皇后问题

4皇后问题的状态树



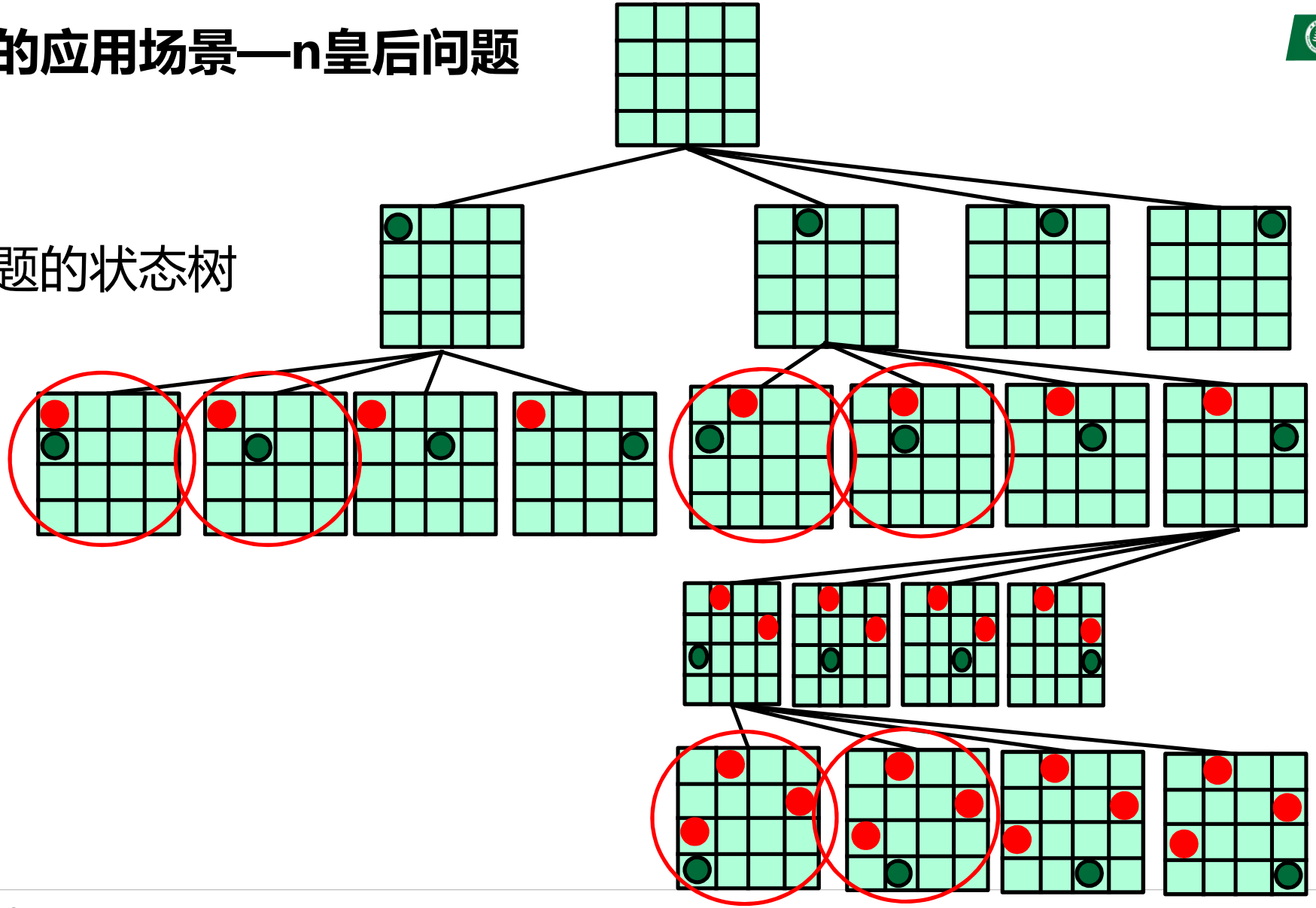
回溯法的应用场景—n皇后问题

4皇后问题的状态树



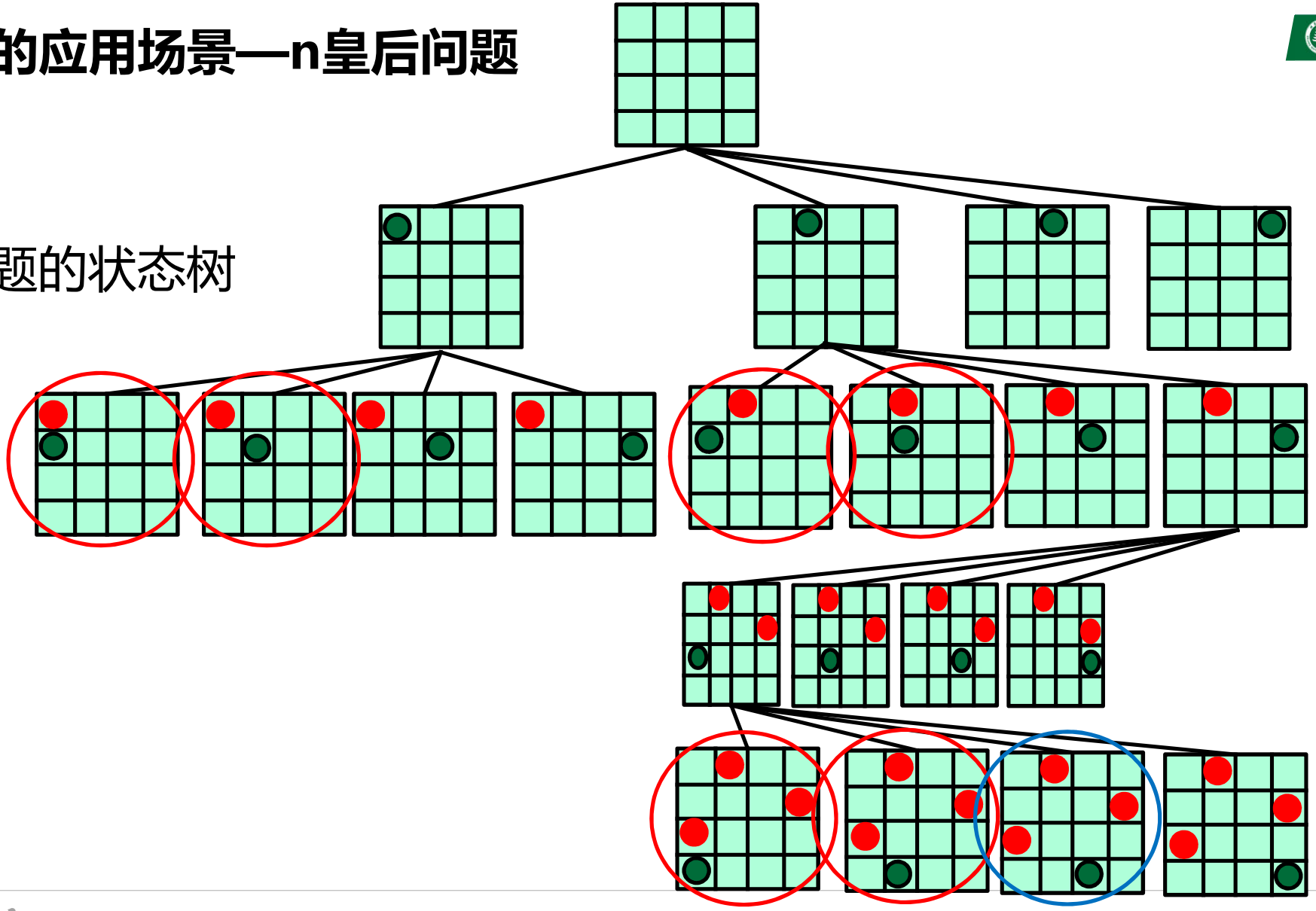
回溯法的应用场景—n皇后问题

4皇后问题的状态树



回溯法的应用场景—n皇后问题

4皇后问题的状态树



回溯法的应用场景—n皇后问题

求解过程：先根遍历状态树

访问当前节点操作：

判断是否得到一个完整的布局（已摆了4个棋子）

若是则输出该布局

否则依次先根遍历各棵子树

判断子树根节点布局是否合法

若合法则遍历该子树

否则剪枝

回溯法的应用场景—n皇后问题

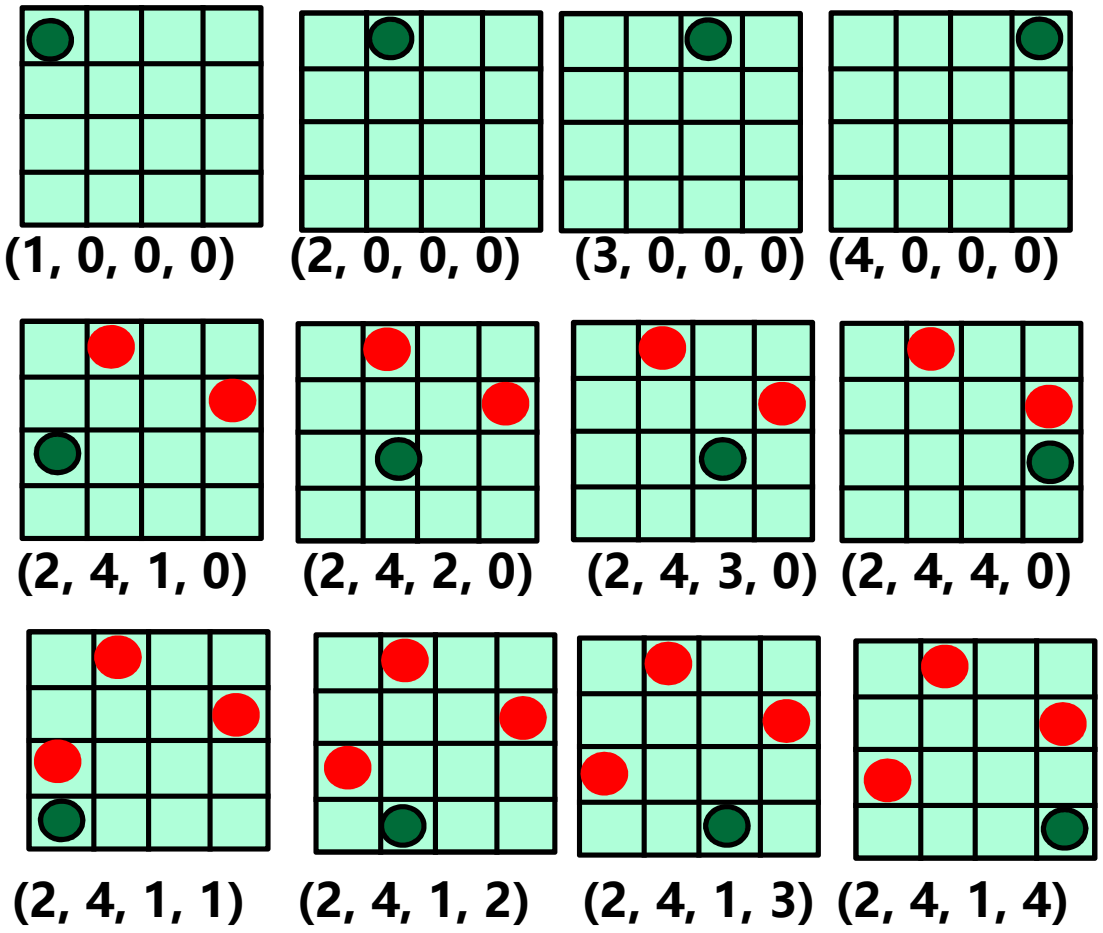
棋盘用p[ ][ ]表示，有棋子的位置设为1，否则设为0

```
void Trial( int t, int n){
        if (t > n ) output( p);//步骤1：输出当前解(叶子节点)
        else for( j =1; j<=n; j++){ //步骤2：一行行放置棋子
        //即依次遍历各棵子树
                p[t][j] = 1;  //在第t行第j列放一个棋子;
                if( Place(p) ) //当前棋局不合法则剪枝
                        Trial( t+1, n);//合法，继续遍历
                p[t][j] = 0;//移走第t行第j列的棋子;
            }//for
}//Trial
```

```
Trial(1, 4);
```

回溯法的应用场景—n皇后问题

用x[ ] 表示棋局



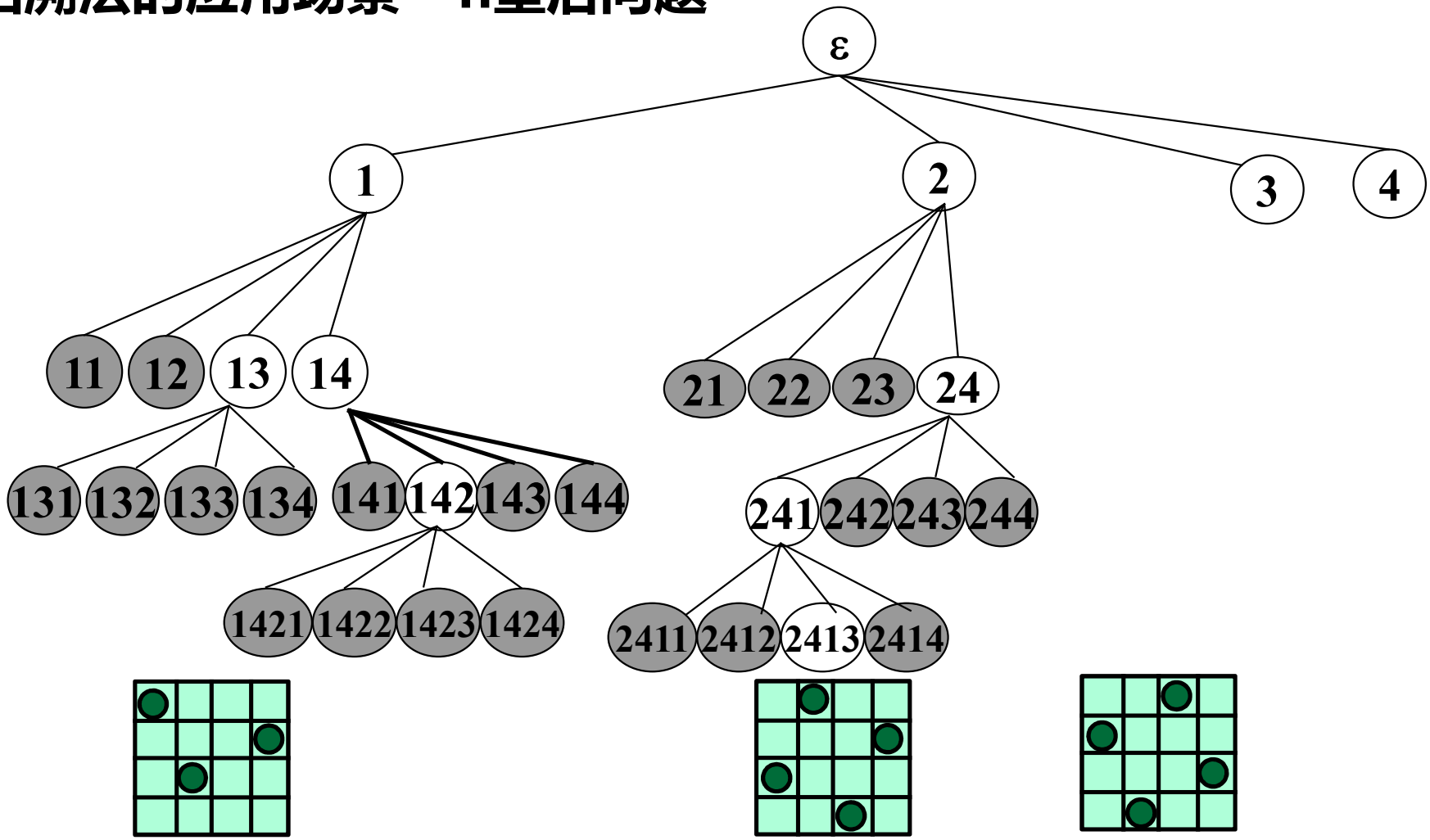
回溯法的应用场景—n皇后问题

用x[ ]表示棋局

```
void backtrack( int t, int n){
        if (t > n )
                output( x );// 步骤1：输出当前解(叶子节点)
        else for( j =1; j<=n; j++){//步骤2：一行行放置棋子
        //即依次遍历各棵子树
                x[t] = j;       //在第t行第j列放一个棋子;
                if( Place( x )) //当前棋局不合法则剪枝
                        backtrack( t+1, n);//合法，继续遍历
            }//for
}//backtrack
```

```
backtrack(1, 4);
```

回溯法的应用场景—n皇后问题



回溯法的基本框架

回溯法的基本步骤：

(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) 设计约束函数和限界函数

(4) 定义回溯函数，以深度优先方式搜索解空间（先根遍历状态树），并在搜索过程中用剪枝函数避免无效搜索。

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；

用限界函数剪去得不到最优解的子树。

递归回溯-基本框架

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, int n){
        if (t > n ) Output(x);//步骤1：输出当前解(叶子节点)
        else{
                x[t] =1;//步骤2.1：“取”第t个元素
                backtrack( t+1, n);//步骤2.2：搜索左子树
                x[t] =0;//步骤3.1：“舍”第t个元素
                backtrack( t+1, n);//步骤3.2：搜索右子树
        }//else
}//backtrack
```

递归回溯-一般性

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);//步骤1：输出当前解(叶子节点)
    else for (int i=f(n, t);i<=g(n, t); i++)
        {//步骤2：依次先根遍历各棵子树
            x[t]=h(i); //加入的元素h(i)
            if (constraint(t)&&bound(t))
                backtrack(t+1);
            x[t]=0; //必要时舍元素h(i)
        }
}
```

递归回溯-一般性+剪枝限界

采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack ( ){
    int t=1;
    while (t>0) {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++) {
                x[t]=h(i);
                if (constraint(t)&&bound(t))
                    {if (solution(t)) output(x); else t++;}
            }
        else t--;
    }
}//iterativeBacktrack
```

最优装载问题

装载问题定义

n件货物(重w[1:n])装两艘船(载重量c1,c2),

$\Sigma_{i=1}^{n} w[i] \le c_1+c_2$, 是否有装载方案

装载方案: 尽可能装满第1艘, 剩余的装第2艘

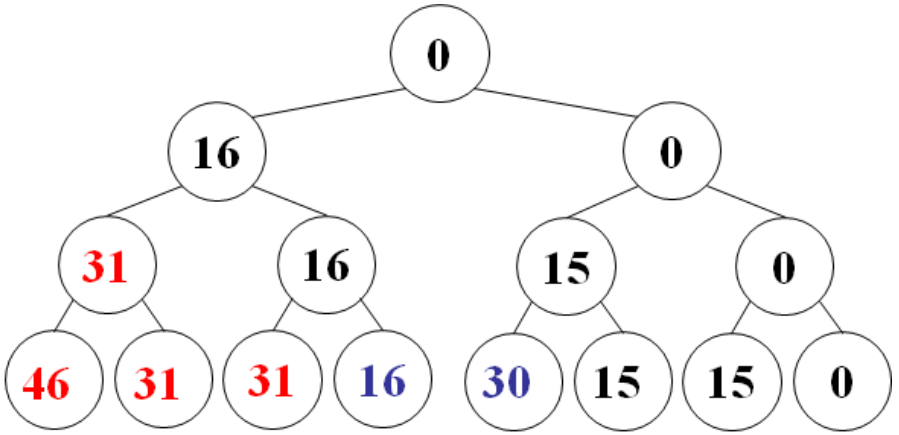
尽可能装满第1艘等价于变形的0-1背包

例: w=[16,15,15], c=30

$$\max \sum_{i=1}^{n} w[i]x[i]$$

$$\text{s.t.} \sum_{i=1}^{n} w[i]x[i] \le c$$

$$x[i] \in \{0,1\}, 1 \le i \le n$$



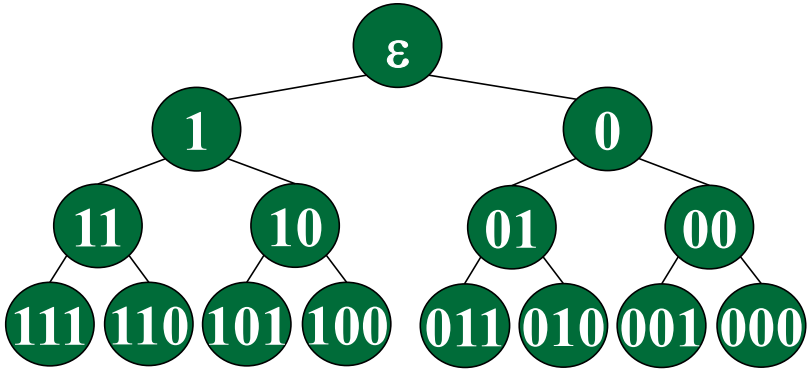
最优装载问题

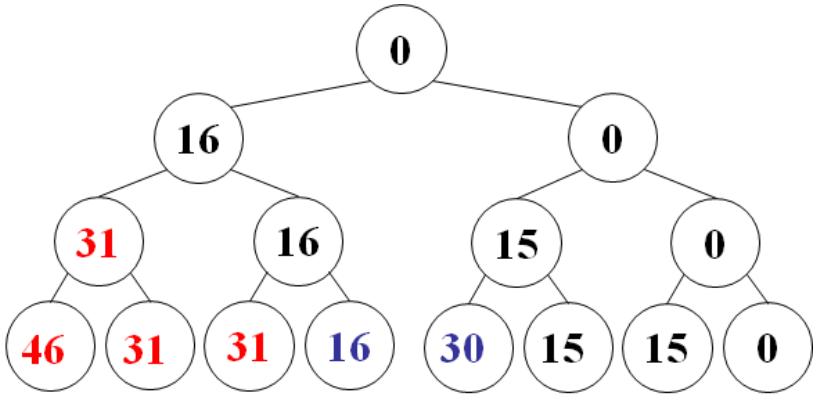
解空间：状态树

解的表示：x[]

约束函数：解x的总重量小于船的载重量c

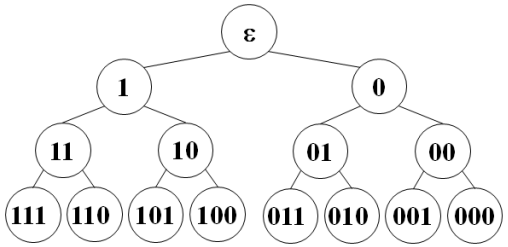
求解过程：在树上进行深度优先搜索

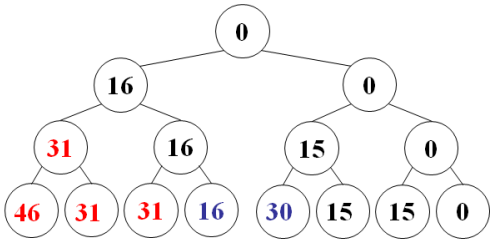




w=[16,15,15], c=30

w=[16,15,15], c=30, backtrack(1)





k-b-(t,bestw,cw)
//进入第k行前的数据.
//b: 实际对应节点标号

程序隐含的解空间结构

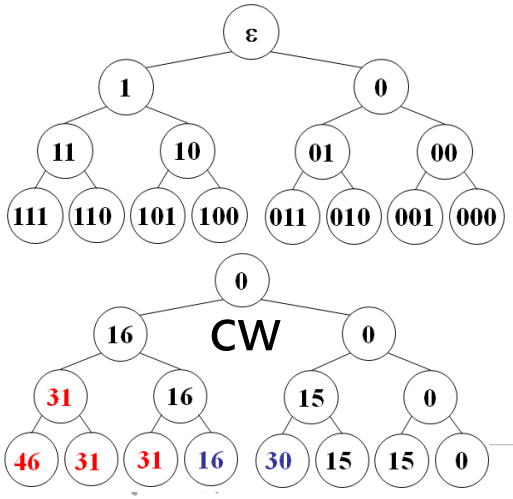
```
backtrack(t)
1. 若t>n,
2. |   若cw≤c且cw>bestw,
3. |   |   bestw=cw
4. |   返回
5. cw+=w[t]
6. backtrack(t+1)//进左分支
7. cw-=w[t]
8. backtrack(t+1)//进右分支
9. 返回
```

各节点的cw值

1-ε-(1,0,0)
5-ε-(1,0,0)
6-ε-(1,0,16)
1-1-(2,0,16)
5-1-(2,0,16)
6-1-(2,0,31)
1-11-(3,0,31)
5-11-(3,0,31)
6-11-(3,0,46)
1-111-(4,0,46)
2-111-(4,0,46)
4-111-(4,0,46)
7-11-(3,0,46)
8-11-(3,0,31)
1-110-(4,0,31)
2-110-(4,0,31)
4-110-(4,0,31)
9-11-(3,0,31)

w=[16,15,15], c=30, backtrack(1)

```
backtrack(t)
1. 若t>n,
2. |  若cw≤c且cw>bestw,
3. |  |  bestw=cw
4. |  返回
5. cw+=w[t]
6. backtrack(t+1)//左分支
7. cw-=w[t]
8. backtrack(t+1)//右分支
9. 返回
```



k-b-(t,bestw,cw) //进入第k步前的数据. 剪枝?

1-ε-(1,0,0)
5-ε-(1,0,0)
6-ε-(1,0,16)
1-1-(2,0,16)
5-1-(2,0,16)
6-1-(2,0,31)
1-11-(3,0,31)
5-11-(3,0,31)
6-11-(3,0,46)
1-111-(4,0,46)
2-111-(4,0,46)
4-111-(4,0,46)
7-11-(3,0,46)
8-11-(3,0,31)
1-110-(4,0,31)
2-110-(4,0,31)
4-110-(4,0,31)
9-11-(3,0,31)
7-1-(2,0,31)
8-1-(2,0,16)
1-10-(3,0,16)
5-10-(3,0,16)
6-10-(3,0,31)
1-101-(4,0,31)
2-101-(4,0,31)
4-101-(4,0,31)
7-10-(3,0,31)
8-10-(3,0,16)
1-100-(4,0,16)
2-100-(4,0,16)
3-100-(4,0,16)
4-100-(4,16,16)
9-10-(3,16,16)
9-1-(2,16,16)
7-ε-(1,16,0)
8-ε-(1,16,0)
1-0-(2,16,0)
5-0-(2,16,0)
6-0-(2,16,15)
1-01-(3,16,15)
5-01-(3,16,15)
6-01-(3,16,30)
1-011-(4,16,30)
2-011-(4,16,30)
3-011-(4,16,30)
4-011-(4,30,30)
7-01-(3,30,30)
8-01-(3,30,15)
1-010-(4,30,15)
2-010-(4,30,15)
4-010-(4,30,15)
9-01-(3,30,15)
7-0-(2,30,15)
8-0-(2,30,0)
1-00-(3,30,0)
5-00-(3,30,0)
6-00-(3,30,15)
1-001-(4,30,15)
2-001-(4,30,15)
4-001-(4,30,15)
7-00-(3,30,15)
8-00-(3,30,0)
1-000-(4,30,0)
2-000-(4,30,0)
4-000-(4,30,0)
9-00-(3,30,0)
9-0-(2,30,0)
9-ε-(1,30,0)

约束条件: 装载问题w[1:n],c

backtrack(t)

// t:层号, cw:当前重量, bestw:最优重量

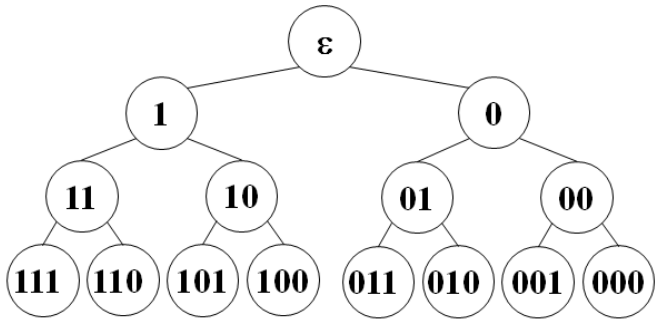
1. 若t>n, (若cw>bestw,bestw=cw.) 返回

2. 若 cw + w[t] ≤ c, 则 ----------//约束条件(剪枝)

3. | cw+=w[t], backtrack(t+1), cw-=w[t],------//左分支

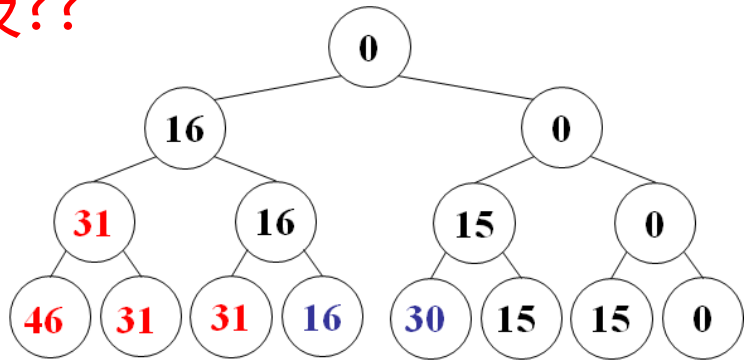
4. backtrack(t+1)----------//右分支

约束条件: 剪去不满足约束条件的子树. 剪枝??



w=[16,15,15]

c=30



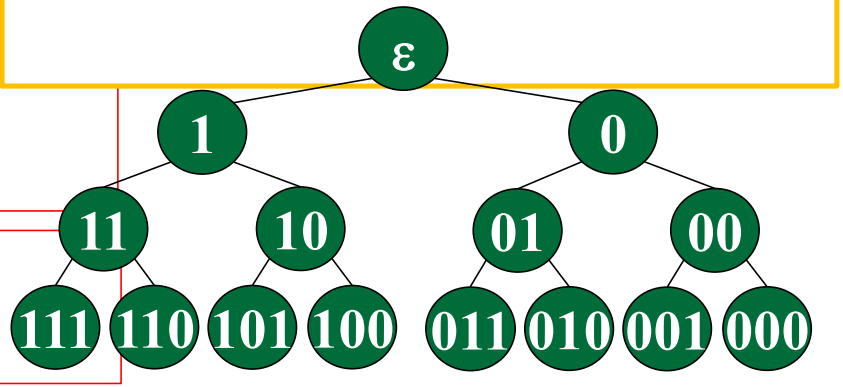
最优装载问题

用x[ ] 表示解，例x = [1, 0, 1]表示集合{16,0,15}

```
void backtrack( int t, int n){
        if (t > n ) Output(x);//叶子节点
        else{
                if( cw+w[t]<=c ) {//若重量不超限则搜索，否则剪枝
                    x[t] =1; cw+=w[t]; backtrack( t+1, n);//搜索左子树
                }
                    cw-=w[t]; x[t] =0;
                    backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

```
if( cw<=c && cw>bestw) bestw=cw;//cw:当前重量，bestw:最优
```

```
bestw=cw=0;
backtrack( 1, 3);
```



限界条件: 装载问题w[1:n],c

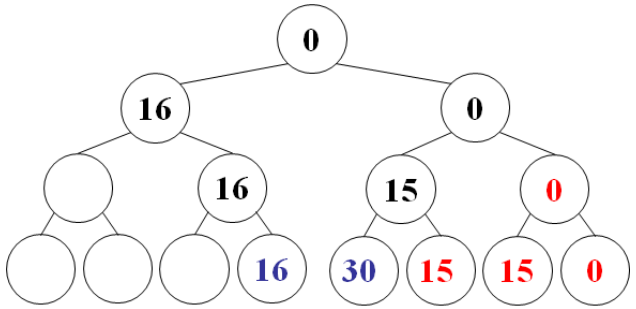
初始: bestw=cw=0. r=$\sum_{t=1}^{n} w[t]$//当前剩余重量

backtrack(t)　　//t层号, cw, bestw

1. 若t>n, (若cw>bestw, bestw=cw, bestx=x.) 返回
2. r-=w[t]
3. 若 cw + w[t] ≤ c, 则----------//约束条件(剪枝)
4. 　　cw+=w[t], x[t]=1, backtrack(t+1), cw-=w[t], //左分支
5. 若 cw + r > bestw, 则----------//限界条件(剪枝)
6. 　　x[t]=0, backtrack(t+1)　//右分支
7. r+=w[t] //还原

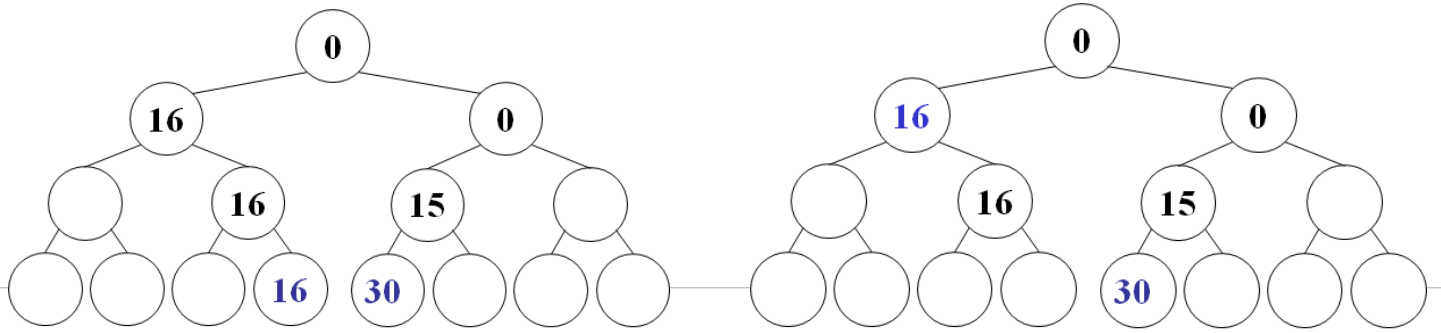
限界条件: 剪去得不到最优解子树

约束条件与限界条件统称为剪枝函数. 执行哪些节点?



提前更新最优值

```
backtrack(t)
1. 若t>n, 则返回
2. r-=w[t]
3. 若 cw+w[t] ≤ c, 则
4. |    若cw+w[t]>bestw, 则bestw=cw+w[t]
5. |    cw+=w[t], backtrack(t+1), cw-=w[t]  -----//x[t]=1
6. 若 cw + r > bestw, 则
7. |    backtrack(t+1) ----------------------------//x[t]=0
8.  r+=w[t]
```



0-1背包回溯 $O(2^n)$

- 输入: n物品重w[1:n], 价值v[1:n], 背包容量C
- 输出: 装包使得价值最大.
- DP: 物品重量为整数, O(nC) // 输入规模max{n, $\log_2 C$}

初始: bestv=cv=cw=0, r=$sum_{t=1}^{n}$ v[t], 执行backtrack(1),

backtrack(t)

1. 若t>n, (若cv>bestv, bestv=cv), 返回
2. r -= v[t]
3. 若 cw+w[t] ≤ c, 则 ----------//x[t]=1
4. | cw+=w[t], cv+=v[t], backtrack(t+1), cv-=v[t], cw-=w[t]
5. 若 cv + r > bestv, 则
6. | backtrack(t+1) ----------//x[t]=0
7. r += v[t] //可添加提前更新最优值和找最优解, 优于备忘录

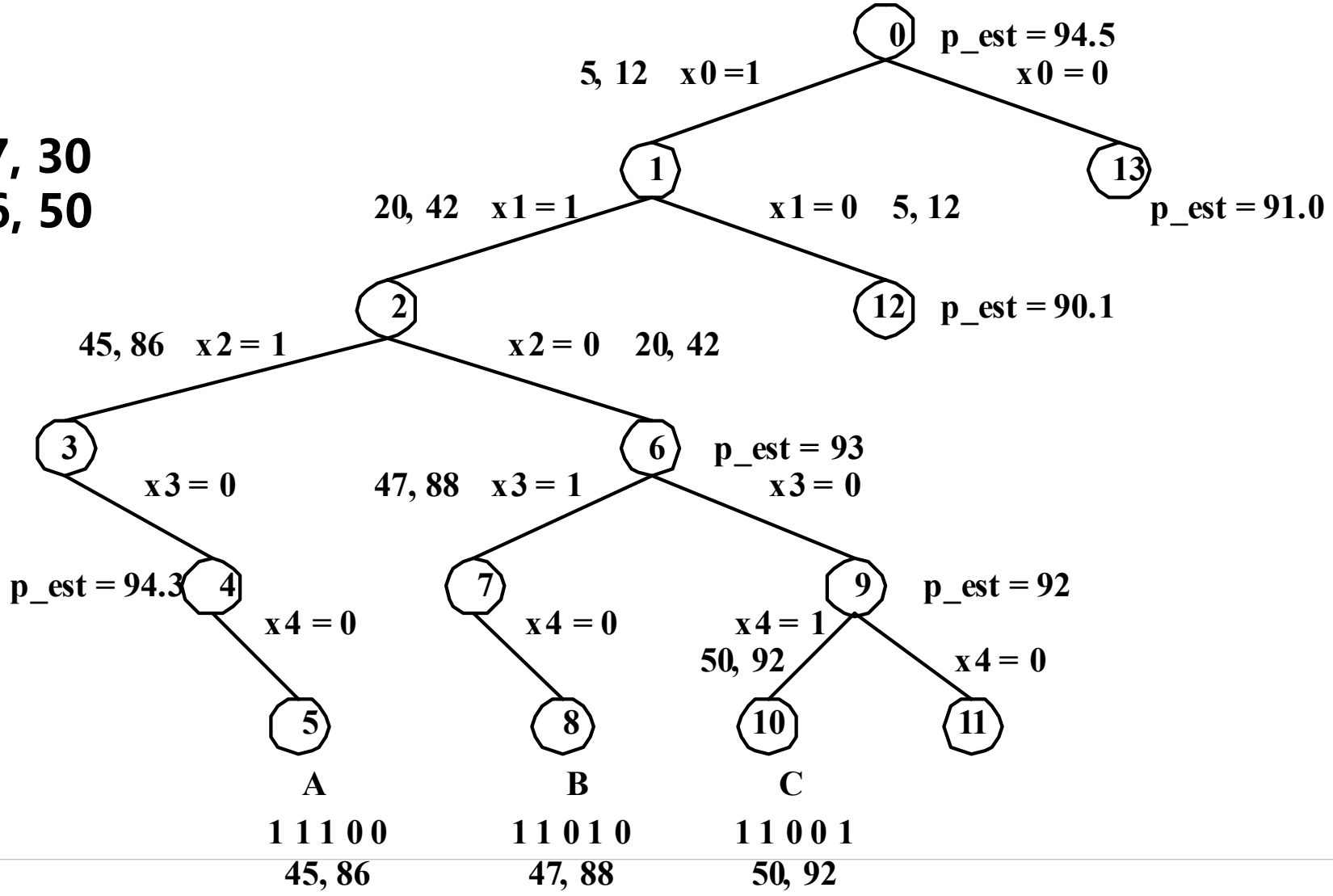
0-1背包回溯 O(2ⁿ)

$$\sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i \le C \quad \text{且} \quad \sum_{i=0}^{k} x_i w_i + \sum_{i=k+1}^{k+m-1} w_i + w_{k+m} > C$$

$$\sum_{i=0}^{k} x_i v_i + \sum_{i=k+1}^{k+m-1} x_i v_i + (C - \sum_{i=0}^{k-1} x_i w_i - \sum_{i=k+1}^{k+m-1} x_i w_i) \times v_{k+m} / w_{k+m}$$

0-1背包限界策略

例如:
C=50,
重量 5, 15, 25, 27, 30
价值12, 30, 44, 46, 50

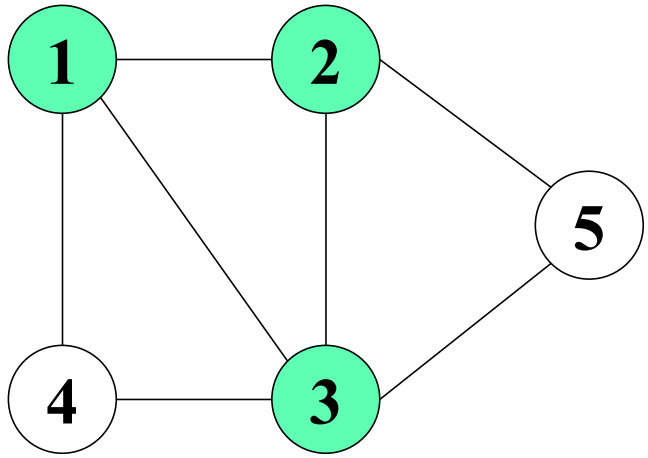


最大团问题

完全子图：给定无向图G=(V，E)。如果U ⊆ V，且对任意u，v ∈ U,有(u，v) ∈ E，则称U是G的完全子图。一个图中的任意两个顶点之间都有边相连的子图

团：G的完全子图U是G的团。

G的最大团：是指G中所含顶点数最多的团。



最大团问题

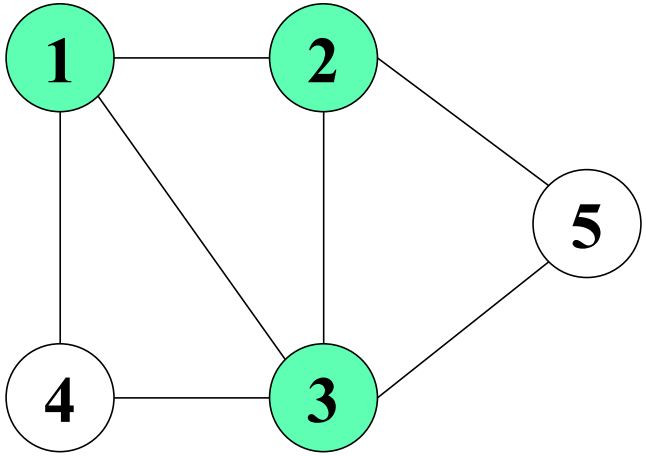
解空间：状态树（子集树）

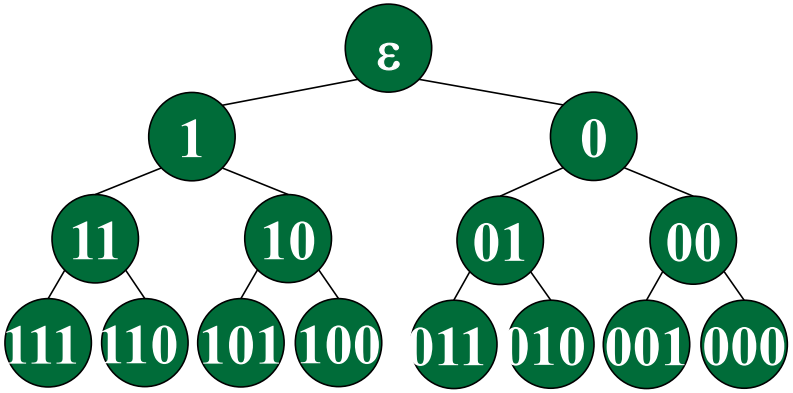
可行性约束函数：

顶点i到已选入的顶点集中每一个顶点都有边相连。

限界函数：

有足够多的可选择顶点使得算法有可能在右子树中找到更大的团





最大团问题-算法

x[ ]表示解，例x=[1,1,0,0,0]表示当前的团是{1, 2}

bestx[ ]：当前最大团

```
void backtrack( int t, int n){
        if (t > n ) Output(x);//叶子节点
        else{
                x[t] =1; // “取”第t个元素
                backtrack( t+1, n);//搜索左子树
                x[t] =0; // “舍”第t个元素
                backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

最大团问题-算法

x[ ]表示解，例x=[1,1,0,0,0]表示当前的团是{1, 2}

bestx[ ]：当前最大团

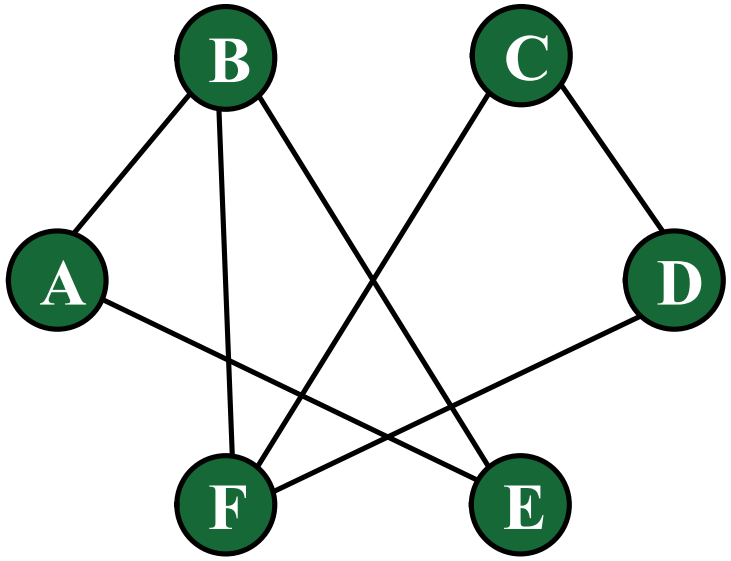
```
void backtrack( int t, int n){
        if (t > n ) if(cn > bestcn){ bestn = cn;}//cn: 当前团顶点数，bestn: 最大团顶点数
        else{
                if( Clique(x)) { //若结点t可增大当前团则搜索，否则剪枝
                        x[t] =1; cn++; backtrack( t+1, n);//搜索左子树
                        cn--; }
                x[t] =0;
                backtrack( t+1, n);//搜索右子树
        }//else
}//backtrack
```

最大团问题-算法

```
void backtrack( int t){
    if (t > n ) {//cn: 当前团顶点数，bestn: 最大团顶点数
        if(cn > bestcn){ bestn = cn; bestx[ ] = x[ ]; }
        else{
            if( Clique(x, t)) {//若结点t可加入当前团则搜素，否则剪枝
                x[t] =1; cn++;
                backtrack( t+1);//搜索左子树
                cn--;
            } //限界函数：有足够多的可选择顶点使得算法有可能在右子树中找到更大的团
            if (cn + n - t > bestn) {// 剪枝或进入右子树
                    x[t] = 0;backtrack(t+1); }
        }//else
}//backtrack
```

最大团问题-算法

图如何表示？图的邻接矩阵



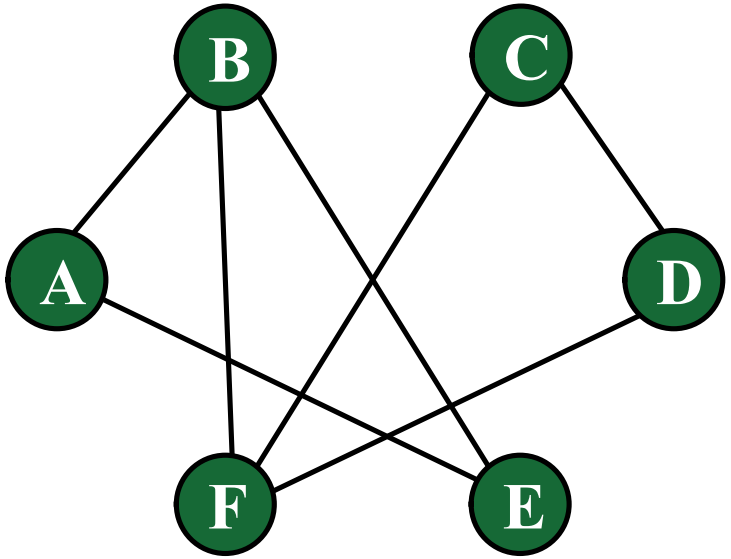
$$A_{ij}=\begin{cases}0,(i,j)\notin VR\\1,(i,j)\in VR\end{cases}$$

	A	B	C	D	E	F
A	0	1	0	0	1	0
B	1	0	0	0	1	1
C	0	0	0	1	0	1
D	0	0	1	0	0	1
E	1	1	0	0	0	0
F	0	1	1	1	0	0

最大团问题-算法

Clique(x, t) //若结点t是否可加入当前团

a[ ][ ]:图的邻接矩阵




```
status Clique(x, t){
    int OK = true; // 检查v_t 与当前团的连接
    for (int j = 1; j < t; j++)
        if (x[j] && a[t][j] == 0) // i与j不相连
            { OK = false;  break;}
    return OK;
}
```

最大团问题-算法

```
void Backtrack( int t){
   if (t > n) // 到达叶结点
        if(cn > bestcn){
              for (int j = 1; j <= n; j++)
               bestx[j] = x[j];
              bestn = cn;        return;
        } //  if(cn > bestcn)
   if (Clique(x, t)) {// 剪枝进入左子树
        x[t] = 1;  cn++;
        Backtrack(t+1);
        x[t] = 0; cn--;}
   if (cn + n - t > bestn) { // 限界进入右子树
        x[t] = 0;
        Backtrack(t+1);}
}
```

说明：

a[ ][ ]:
图的邻接矩阵

bestx[ ]:
最大团包含节点

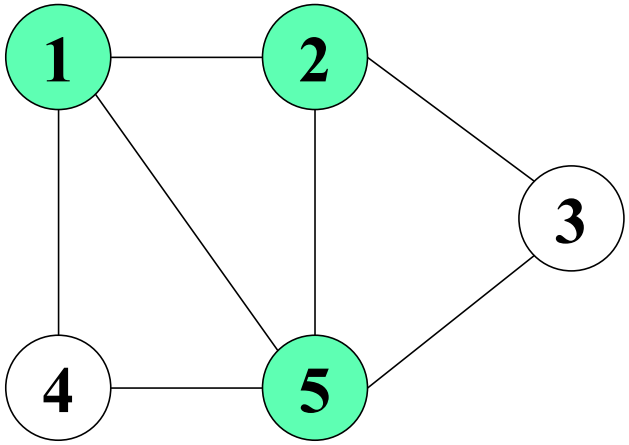
bestn:
目前最大团顶点数

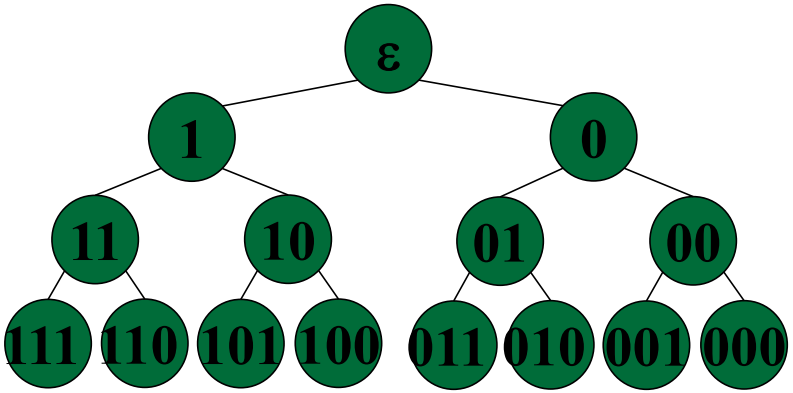
cn:
当前团顶点数

x[i]=1或0:
表示取或不取v_i.

最大团问题-算法

在最坏情况下有 $O(2^n)$个结点需要计算可行性约束
计算可行性约束需要O(n)时间
回溯算法backtrack所需的计算时间为$O(n2^n)$。





符号三角形

问题:

第一行有n个符号,2个同号下面都是“+”
2个异号下面都是“-”。
给定的n，求“+” “-”
个数相同的符号三角形个数 。

解向量:

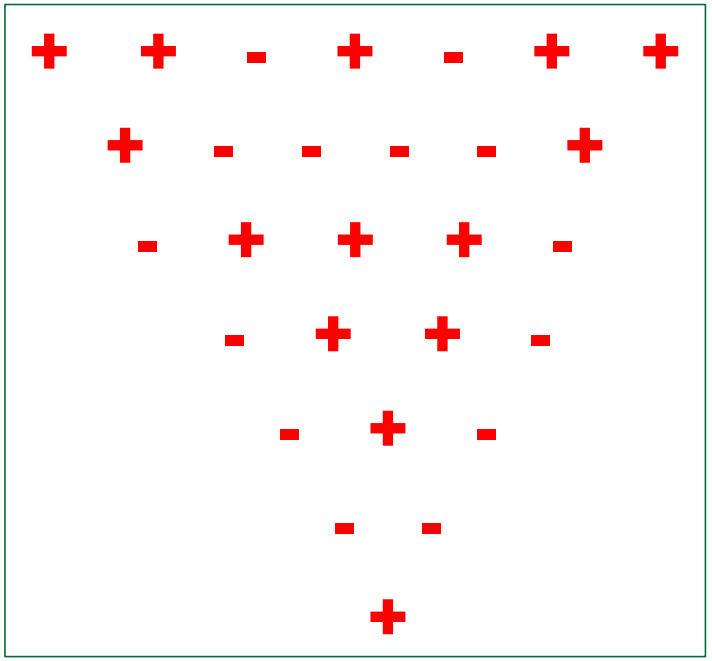
用n元组x[1:n]表示符号三角形的第一行

无解的判断:

n*(n+1)/2为奇数

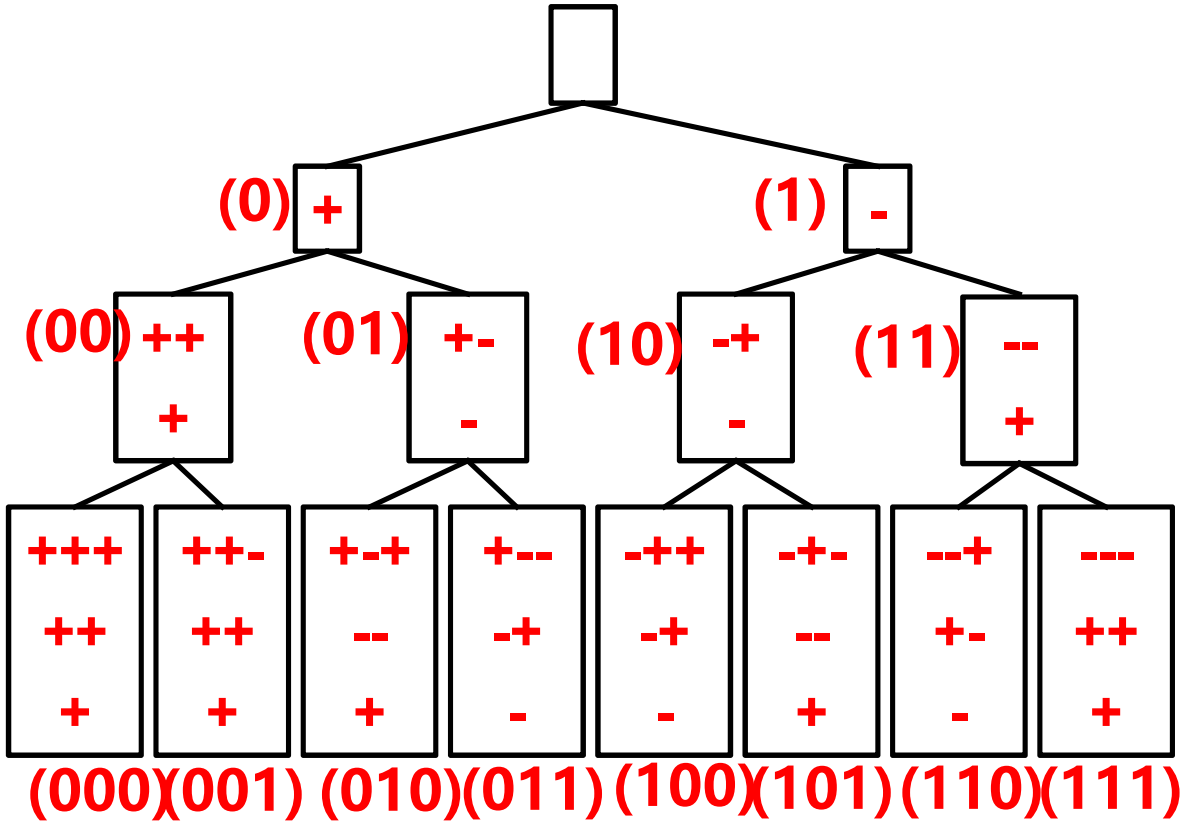
限界条件:

0,1的个数都不能超过n(n+1)/4



n = 7;
14个“+”，14个“-”

符号三角形-状态树



x = (0, 0, 1, 0, 1, 0, 0)

```
+ + - + - + +
  + - - - - +
  - + + + -
    - + + -
      - + -
        - -
          +
```

n = 7;

14个“+”，14个“-”

符号三角形-算法

p[ ][ ]当前三角形

```
void Backtrack( int t){
   if (t==n) {
          //得到一个n边三角形
        //计算p并累计p中新增1的个数count1;
         //若符合条件则sum++;
   } else {
        p[1][t]=1; //"-"
        backtrack(t+1);
        p[1][t]=0;//"+"
        backtrack(t+1);
   }
}//Backtrack
```

符号三角形-算法

```
void Backtrack( int t){
    if (t>n) {
        if (toCount(t) == t*(t-1)/4) {
            sum++; printf("%d:%d\n",sum, toCount(t-1));
        }
    } else {
        p[1][t]=1; //"-"
        backtrack(t+1);
        p[1][t]=0;//"+"
        backtrack(t+1);
    }
}//Backtrack
```

符号三角形-算法

```
void Backtrack( int t){
    if (t>n) {
        if (toCount(t) == t*(t-1)/4) {
            sum++; printf("%d:%d\n",sum, toCount(t-1));
        }
    } else {
        p[1][t]=1; //"-"
        backtrack(t+1);
        p[1][t]=0;//"+"
        backtrack(t+1);
    }
}//Backtrack
```

```
int toCount( int t){
    for(s=2;s<=t;s++,k++) {
        for(k=1;k<=t-s+1;k++) {
            p[s][k] = p[s-1][k]^p[s-1][k+1];}
    } // 计算值
    for(s=1;s<=t;s++) {
        for(k=1;k<=t-s+1;k++) {count += p[s][k];}
    } // 统计1的个数
    return count;
}// toCount
```

符号三角形-算法-改进

```
//sum记+-个数相同的符号三角形个数
//half=n(n+1)/4, p维护当前符号三角形
backtrack (t)
1.  若t>n, sum++，返回
2.  对 i = 0 : 1,
3.  |   p[1][t]=i; count+=i;
4.  |   对 j = 2 : t,
5.  |   |   p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
6.  |   |   count+=p[j][t-j+1];
7.  |   若 count≤half 且 (t+1)t/2-count≤half
8.  |   |   backtrack(t+1);
9.  |   对 j = 1 : t,  count-=p[j][t-j+1];
```

符号三角形-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 $O(2^n)$个结点需要计算可行性约束

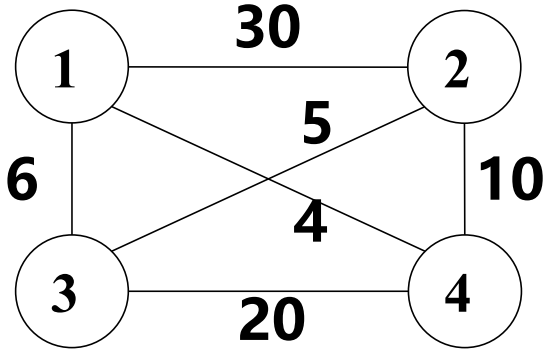
解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$。

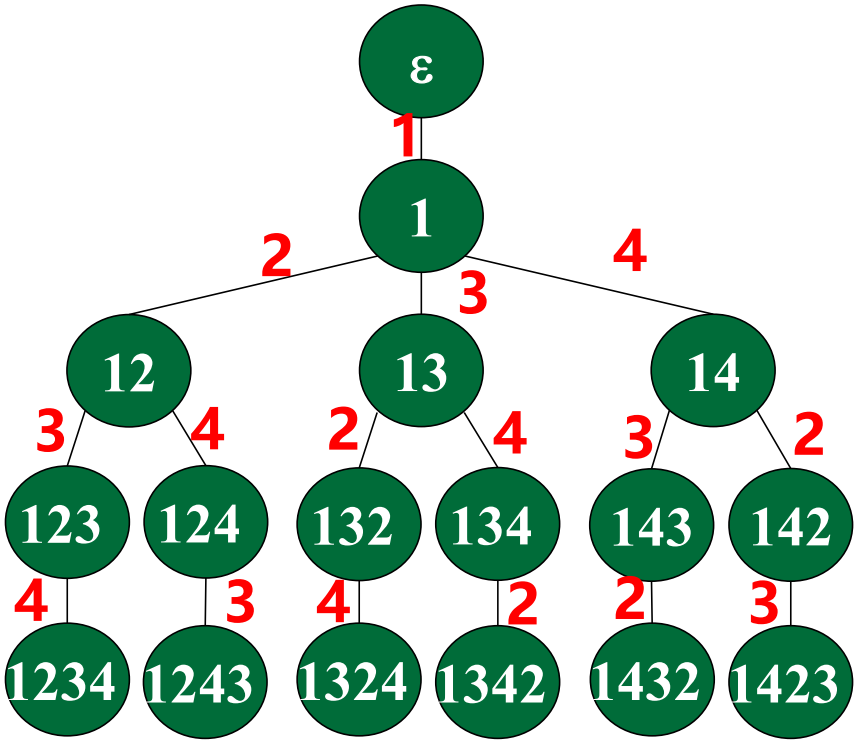
旅行商问题(TSP)

问题：求一条从某城市出发，经过每个城市最后回到起点的回路，使总距离最小

解空间：全体排列（(n-1)!）

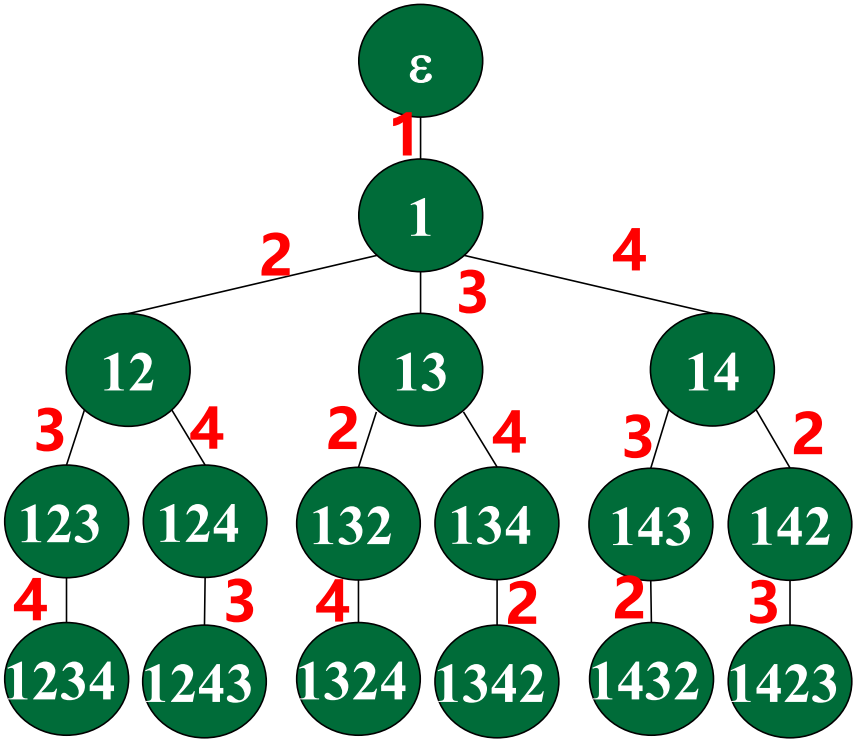
解空间结构：排列树

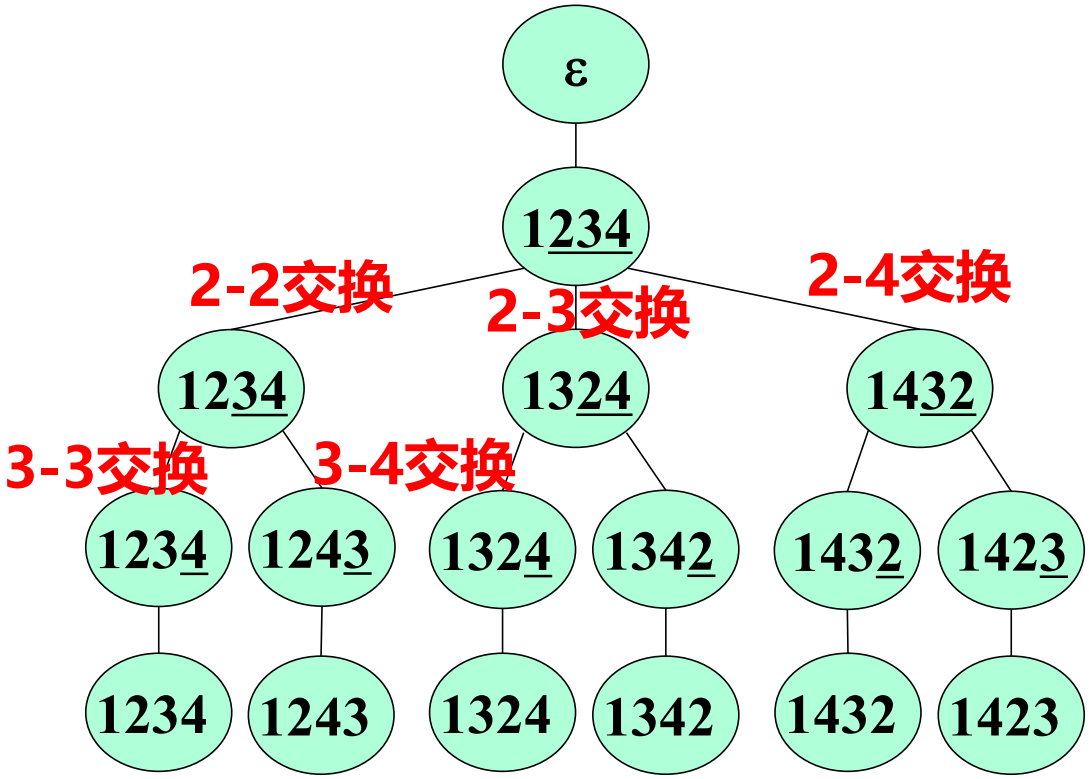




旅行商问题(TSP)

解向量：用n元组x[1:n]表示当前排列





旅行商问题(TSP)

backtrack(t)

1. 若 t > n, 则判断记录bestc, bestx, 返回
2. 对 j = t : n
3. |　交换x[t],x[j],
4. |　若x[1:t]费用<bestc, 则
5. |　|　　backtrace(i+1)
6. |　交换x[t],x[j]

- 初始x[i]=i, bestc=INF,
- backtrack(2)
- 分支数O((n-1)!), 时间O(n!)

TSP-算法

```
void Backtrack (int i) {
    if (i == n) //步骤1：比较记录当前最优解
        if ((cc + a[x[n-1]][x[n]] + a[x[n]][x[1]] < bestc)) {
            for (int j = 1; j <= n; j++) bestx[j] = x[j];
            bestc = cc + a[x[n-1]][x[n]] + a[x[n]][1];        }//if ((cc
    else {// 步骤2：搜索子树i到n
        for (int j = i; j <= n; j++) //先判断是否进入x[j]子树?
            if ((cc+a[x[i-1]][x[i]] < bestc)) {//不满足则剪枝
                Swap(x[i], x[j]);
                cc += a[x[i-1]][x[i]];
                Backtrack(i+1);
                cc -= a[x[i-1]][x[i]];
                Swap(x[i], x[j]);     }
        }//else
}//Backtrack
```

x[]: 当前解
cc: 当前解的距离
bestx[]: 当前最优解
bestc: 当前最短距离

TSP-算法复杂度

复杂度分析：

排列树有(n-1)!个叶子节点

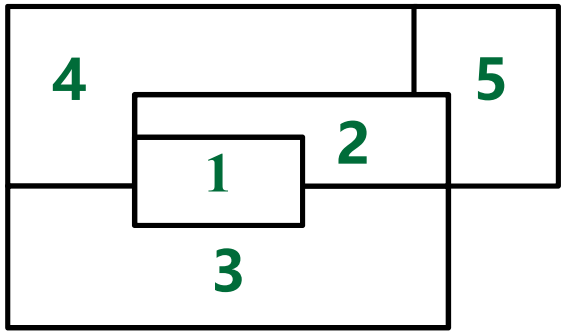
backtrack在最坏情况下需要更新当前最优解O((n-1)!)次

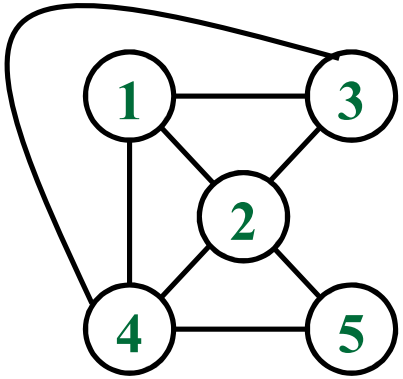
每次更新bestx需计算时间O(n)

从而整个算法的计算时间复杂性为O(n!)。

图的m着色问题

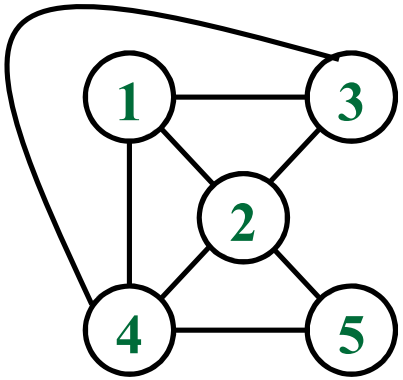
- 定义：给定无向连通图G和m种不同的颜色。用这些颜色为图G的各顶点着色，每个顶点着一种颜色。是否有一种着色法使G中每条边的2个顶点着不同颜色？

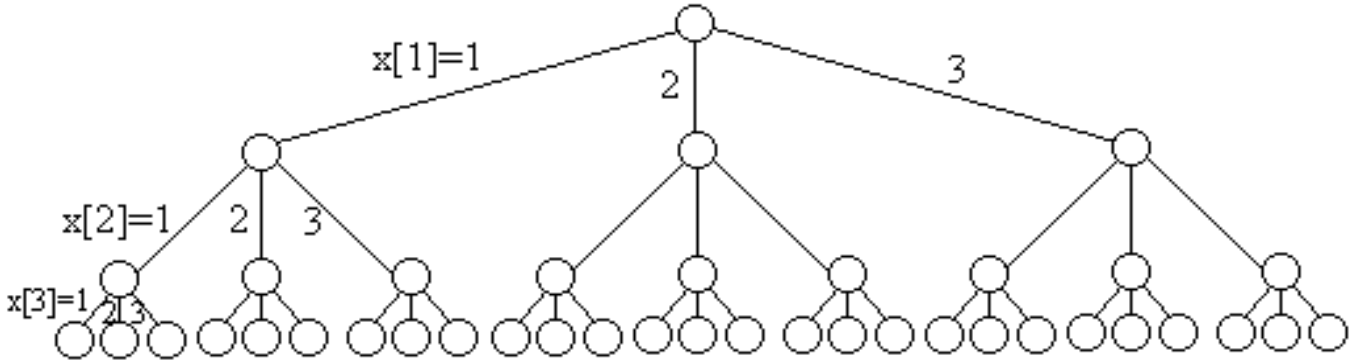




图的m着色问题

- 解向量：(x1, x2, … , xn)表示顶点i所着颜色x[i]
- 可行性约束函数：顶点i与已着色的相邻顶点颜色不重复。





图的m着色问题

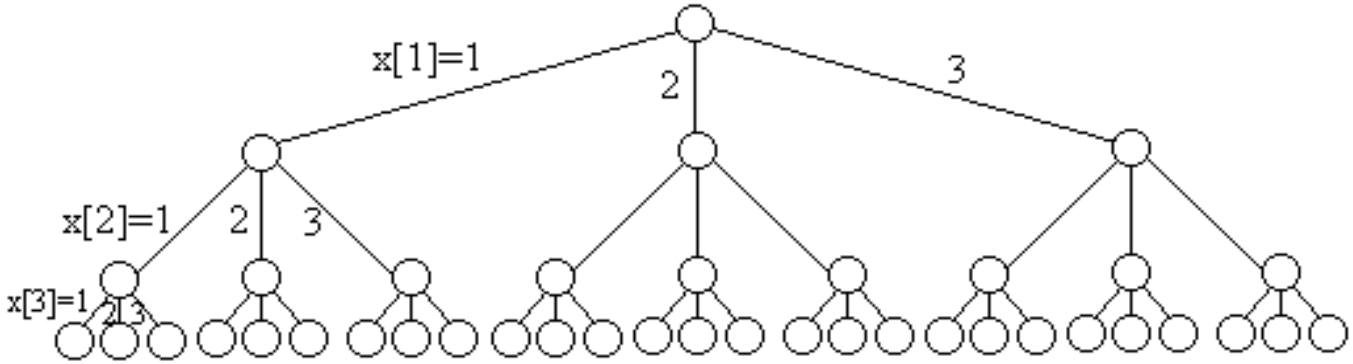
```
void Backtrack(int t){
    if (t>n) {
        sum++;
        output(x );
    }
    else{
        for (int i=1;i<=m;i++) {
        x[t]=i;
        if (Ok(t)) );//若颜色不冲突则遍历子树，否则剪枝
                Backtrack(t+1);
    }
}//Backtrack(int t)
bool Ok(int k){ // 检查颜色可用性
        for (int j=1;j<k;j++)
                if ((a[k][j]==1)&&(x[j]==x[k])) return false;
        return true;
}
```

图的m着色问题-复杂度分析

- 图m可着色问题的解空间树中内结点个数是$\sum m^i(0\le i\le n-1)$。
- 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

$$\sum m^i(mn)=nm(m^n-1)/(m-1)=O(nm^n)$$

$$(0\le i\le n-1)$$



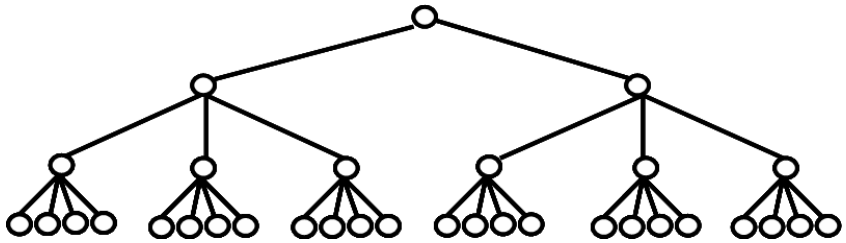
回溯法的效率分析

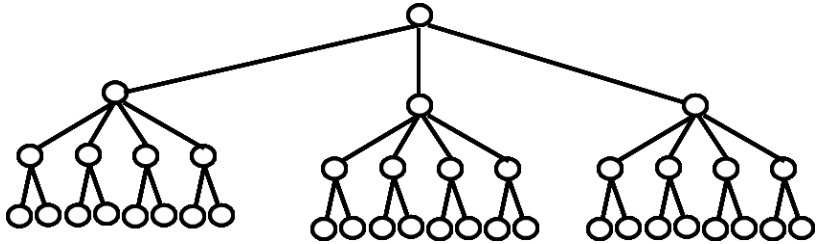
影响回溯算法效率的因素

1. 每个顶点的产生时间
2. 计算剪枝函数的时间
3. 剪枝后剩余顶点个数

剪枝函数的设计与平衡？更好的剪枝会增加计算时间

重排原理：优先搜索取值最少的x[i]





回溯法的效率举例-4皇后问题

问题的规模：C(16, 4) =43680

状态树：约束1：棋子不放在同行中

不剪枝：

叶子节点 4^4 = 256

状态树的扩展节点：$\sum_{i=0}^{4} 4^i$=341

剪枝：约束2：棋子不同列且不同斜线

叶子节点: 16，解的个数:2

状态树的扩展节点：61

回溯法的效率举例-n皇后问题

教材[王]中对n皇后问题的回溯效率进行了概率估计

对于n=8的情形

估计搜索节点数/总节点数 ≈ 1.55%

总节点数 $= \sum_{i=1}^{n} n!/(n-i)! = 109601$ (n=8)

搜索节点数估计:

记第i层x[i]的可选列数为m_i,

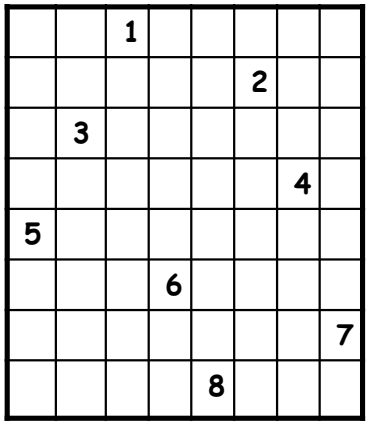
随机选一可选列进入下一层

得节点数$1+m_1+m_1m_2+m_1m_2m_3+\ldots$

多次取平均得1702

m_1=8; m_2=5: 因为2可选4-8列; m_3=4: 因为3可选1,6,7,8;

m_4=3: 因为4可选3,7,8; m_5=2: 因为5可选5,8;.....



本章作业

问题描述: 羽毛球队有男女运动员各n人. 给定2个n×n矩阵P和Q.

P[i][j]是男运动员i与女运动员j配混合双打的男运动员竞赛优势;

Q[i][j]是女运动员i与男运动员j配混合双打的女运动员竞赛优势.

由于技术配合和心理状态等各种因素影响, P[i][j]不一定等于Q[j][i]. 男运动员i和女运动员j配对的竞赛优势是P[i][j]*Q[j][i]. 设计一个算法, 计算男女运动员最佳配对法, 使得各组男女双方竞赛优势的总和达到最大.

数据输入: 正整数n(1≤n≤20)，P和Q

结果输出: 最佳配对的各组男女双方竞赛优势总和。

例如：n=3，P =

10	**2**	**3**
2	**3**	**4**
3	**4**	**5**

Q=

2	**2**	**2**
3	**5**	**3**
4	**5**	**1**

结果等于52