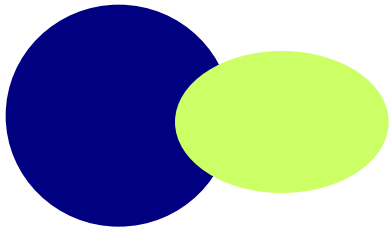



算法设计与分析

第五章 回溯法





“不进则退，不喜则忧，
不得则亡，此世人之常”
——《邓析子·无厚篇》



关于“不进则退”

我们听过很多“不进而退”的故事，这些故事告诫人们如果不进步，就会倒退。



然而：“不进则退”的另一层含义---很积极哦！

“退一步海阔天空”如果一条路无法走下去，就退一步，换条路走走。。这不失一个很好的办法！GOOD IDEA!



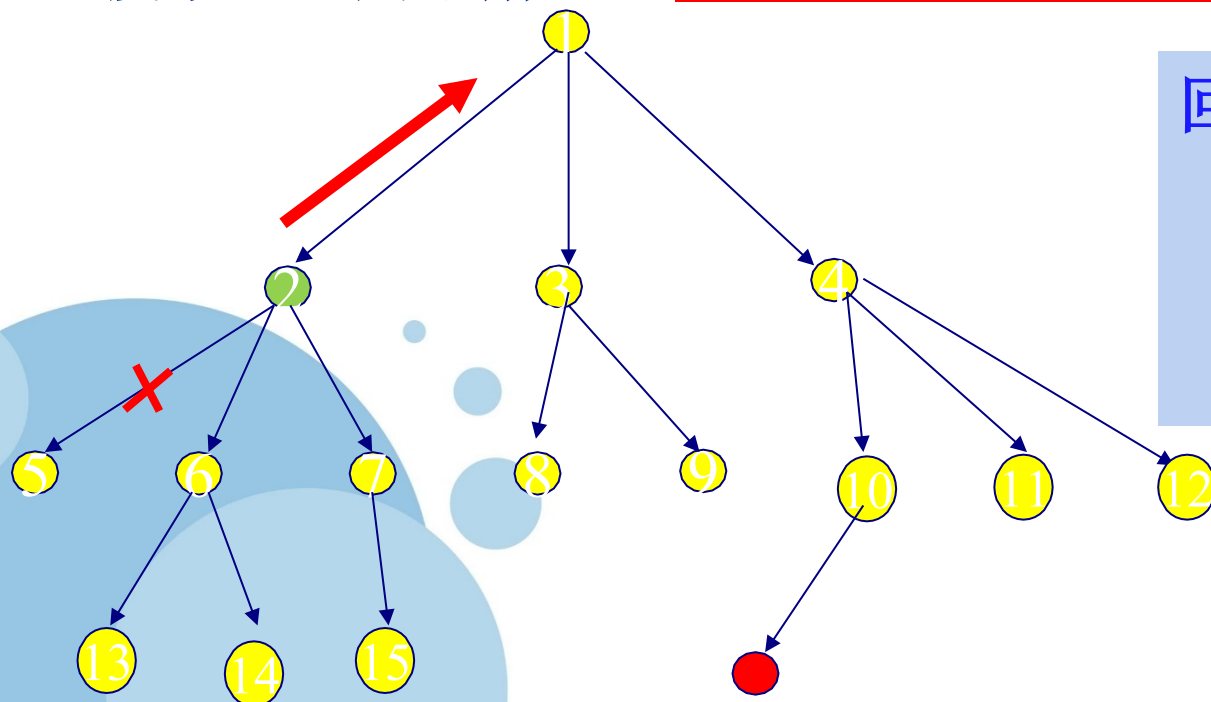
不进则退——回溯法

- 在算法设计中，有很多问题是**无法运用某种公式推导或采用循环**的方法来求解的。
- 假如完成某一件事需要经过若干步骤，而每一步又都有若干种可能的方案供选择，完成这件事就有许多方法。当需要在其中找出满足条件的最优解时，无法根据某些确定的计算法则，而是利用**试探**和**回溯**的搜索策略。
- 回溯法是一种**优选搜索方法**，按照设定的条件向前搜索。如果走到某一步时发现当前的选择并不优或无解，则回退一步（回溯），重新进行选择，直到达到目标或结束。搜索过程既系统又带有跳跃性。
- 回溯法有“**通用解题法**”之称。

回溯法本质

回溯法按照选优条件进行深度优先搜索，以达到目标。

回溯法的本质是先序遍历一棵状态树过程，不是遍历前预先建立，而是隐含在遍历过程中，使用它可以避免穷举式搜索，适用于解一些排列和组合数相当大的问题



回溯法：
能进则进，
进不了则换，
换不了则退

【基本思想】先定义问题的解空间(问题求解的空间)，然后在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任一结点时，总是先判断该结点是否肯定包含问题的解。如果肯定不包含，则跳过对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，继续按深度优先的策略进行搜索。

整个试探搜索的过程是由计算机完成的，所以对于搜索试探要避免重复循环，即要**对搜索过的结点做标记**。

可见，回溯法就是“**试探着走**”。如果尝试不成功则退回一步，再换一个办法试试。反复进行这种试探性选择与返回纠错过程，直到求出问题的解为止。

问题的解空间

应用回溯法求解问题时，首先应该明确问题的解空间。

(1) 解空间概念

问题的解向量: (x_1, x_2, \dots, x_n)

问题的解空间（两种类型）: 子集树和排列树

问题的可行解: 满足约束条件的解

问题的最优解: 最优的可行解

显约束: 对每个分量 x_i 的取值进行限定

隐约束: 对不同分量之间施加的约束

剪枝

Chapter 5 回溯法解空间

问题的解由解向量 $X=\{x_1, x_2, \dots, x_n\}$ 组成，其中分量 x_i 表示第 i 步的操作。

所有满足约束条件的解向量组构成了问题的解空间。

问题的解空间一般用树形式来组织，也称为解空间树或状态空间，树中的每一个结点确定所求解问题的一个问题状态。

树的根结点位于第1层，表示搜索的初始状态，第2层的结点表示对解向量的第一个分量做出选择后到达的状态，以此类推。

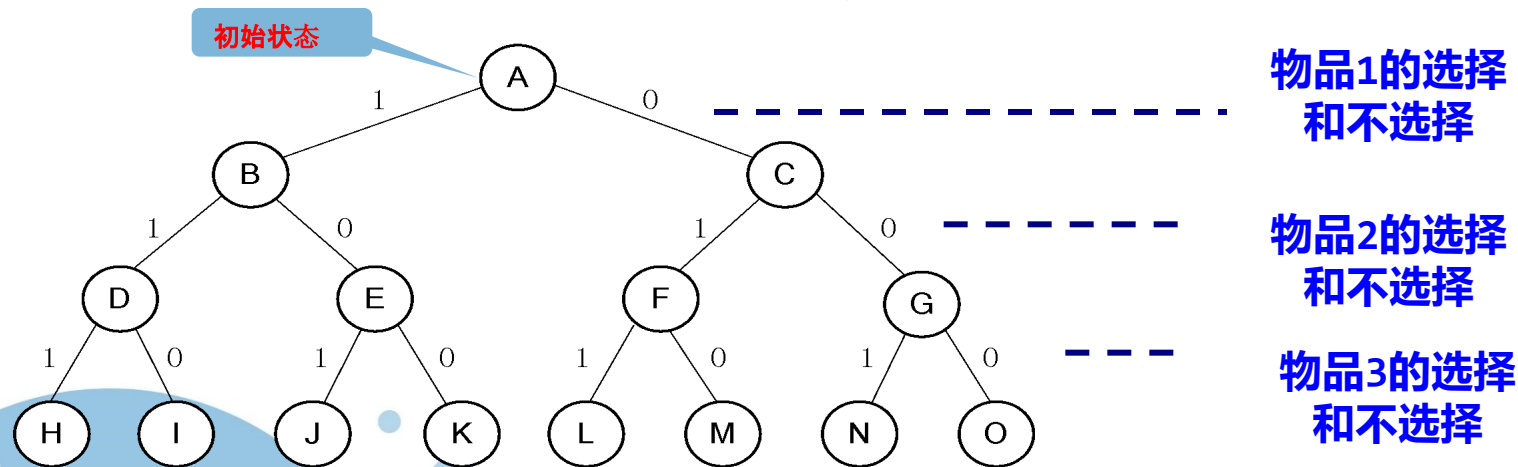
回溯法求解问题类型

- 找所有解
- 找最优解

Chapter 5 回溯法解空间树

解空间树——子集树

所给问题是从 n 个元素的集合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。



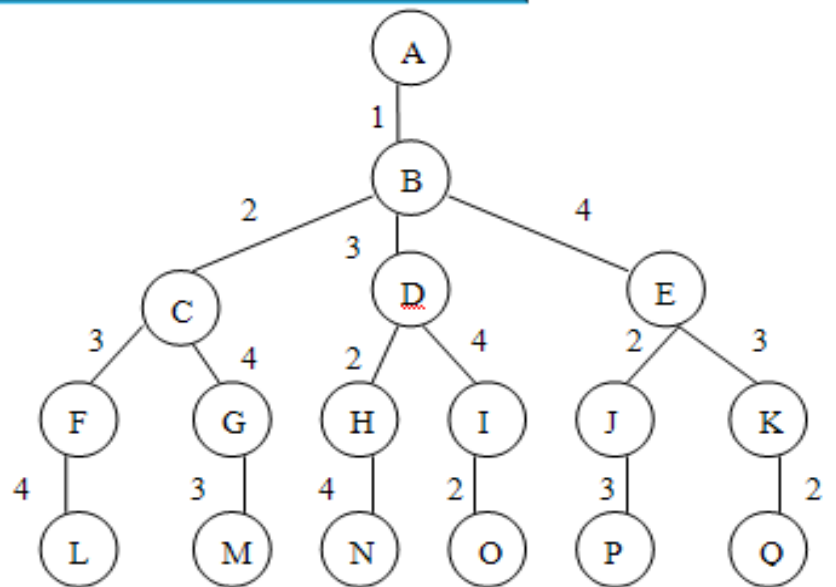
$\{1,1,1\} \{1,1,0\} \dots$

0-1 背包问题的子集树($n=3$)

树的根结点位于第1层，表示搜索的初始状态，第2层的结点表示对解向量的第一个分量做出选择后到达的状态，以此类推。

Chapter 5 回溯法解空间树

解空间树——排列树



可行解: {1,2,3,4} {1,2,4,3} ... {1,4,3,2}

解向量初始值 $x=(1,2,3,4)$ 以旅行商问题为例，上图是起点城市固定时的排列树，若起点城市不固定，则解空间树是 $x=(1,2,3,4)$ 的全排列。

Chapter 5 回溯法解空间搜索

回溯法搜索（寻找解的过程）

确定解空间的组织结构后（子集树or排列树），回溯法就从开始结点（根结点）出发，以深度优先搜索的方式搜索整个解空间。这个开始结点就成为一个活结点，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。这个新结点就成为一个新的活结点，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移动，则当前的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，并使这个活结点成为当前的扩展结点。回溯法即以这种工作方式递归地在解空间中搜索，直至找到所要求的解或解空间中已无活结点时为止。

剪枝函数

一是用约束函数

二是用限界法

- 生成问题解空间的基本状态（在解空间树中，结点对应了搜索过程中的某个状态）

- 扩展结点
- 活结点
- 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（**深度优先搜索（递归进层）**）。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（**递归退层**），继续生成R的下一个儿子（如果存在）。

搜索下一个可能的分支

Chapter 5 回溯法解空间搜索示例

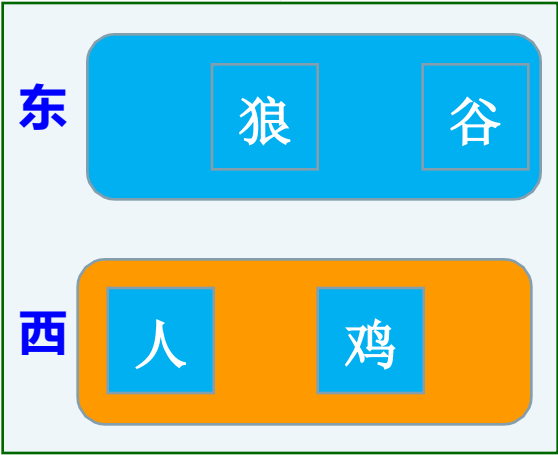
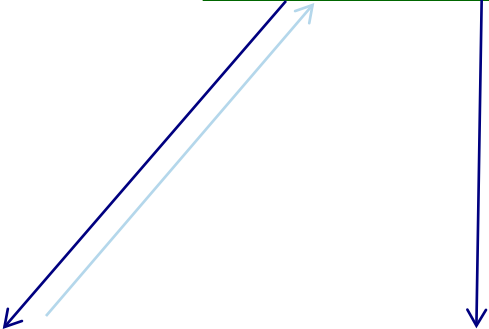
【例】 有一个农夫（人）过河问题，指在河东岸有一个农夫、一只狼、一只鸡和一袋谷子，只有当农夫在现场时，狼不会把鸡吃掉，鸡也不会吃谷子，否则会出现这样的情况。

另有一条小船，该船只能由农夫操作，且最多只能载下农夫和另一样东西。设计一种过河方案，将农夫、狼、鸡和谷子借助小船运到河西岸。

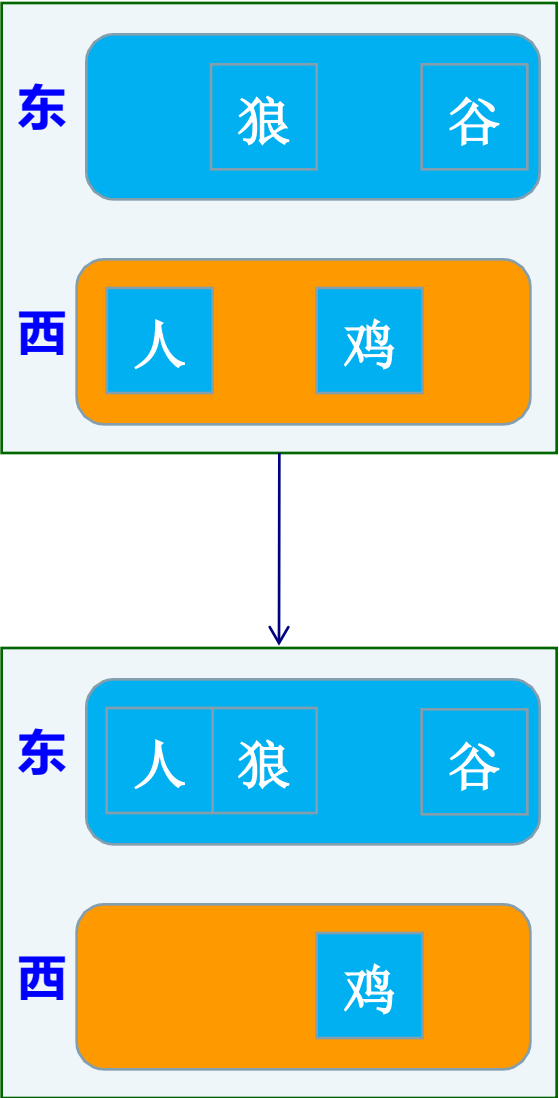
Chapter 5

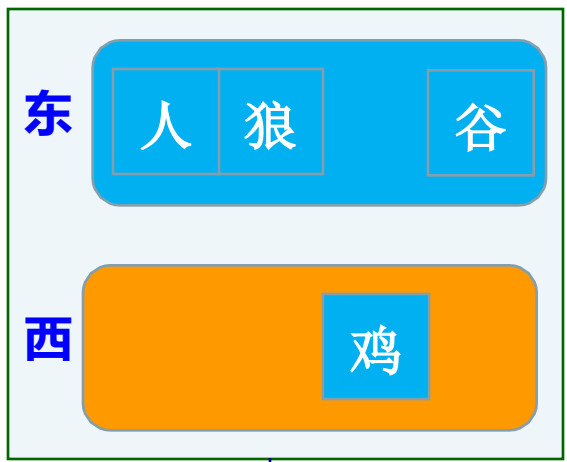


初始状态

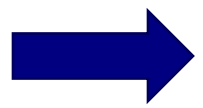


Chapter 5





...

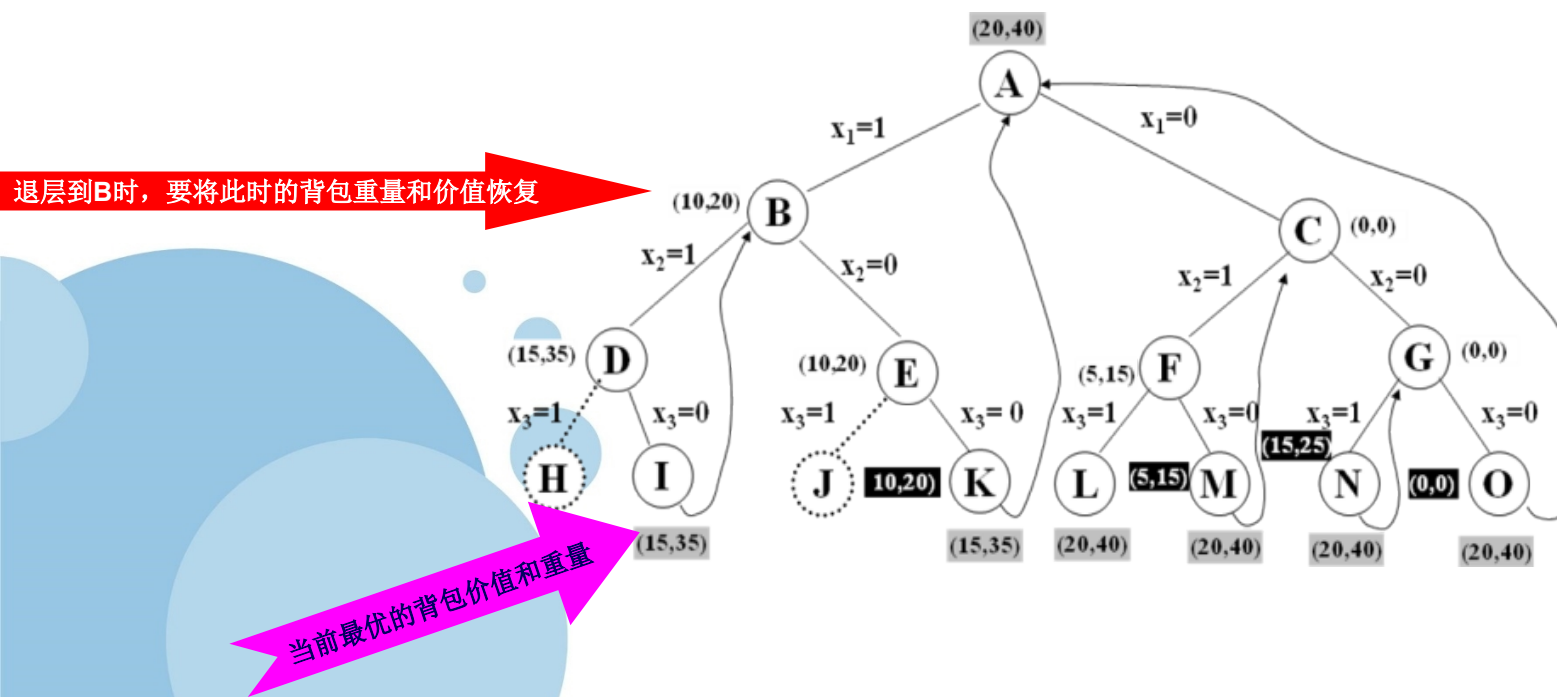


目标状态

0-1背包问题的解空间树及其搜索过程。

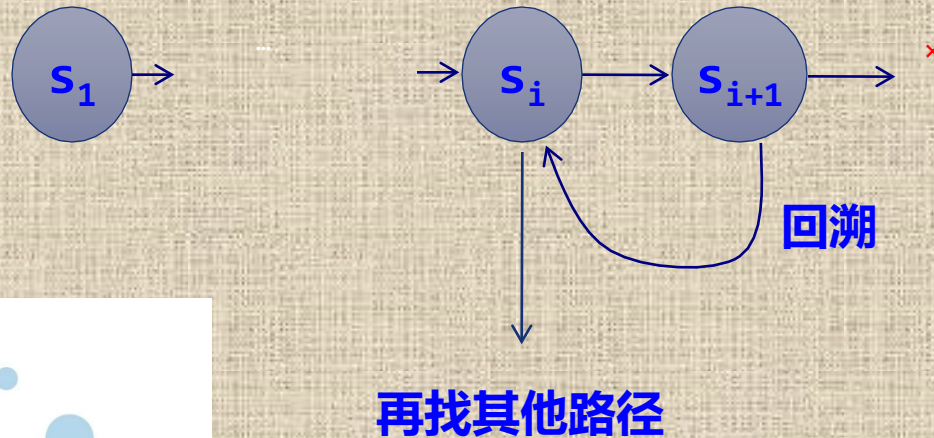
问题描述：物品种数 $n=3$ ，背包容量 $C=20$ ，物品价值 $(v_1, v_2, v_3)=(20, 15, 25)$ ，物体重量 $(w_1, w_2, w_3)=(10, 5, 15)$ ，求 $X=(x_1, x_2, x_3)$ 使背包价值最大？

问题分析： 0-1背包问题的解空间树及其搜索过程如图所示。



【回溯法搜索策略】 在包含问题的所有解的解空间树中，按照**深度优先搜索**的策略，从根结点（开始结点）出发搜索解空间树。

【回溯法解空间搜索】 当从状态 s_i 搜索到状态 s_{i+1} 后，如果 s_{i+1} 变为死结点，则从状态 s_{i+1} 回退到 s_i ，再从 s_i 找其他可能的路径，所以回溯法体现出走不通就退回再走的思路。



回溯法搜索解空间时，通常采用两种策略避免无效搜索，提高回溯的搜索效率：

- 用约束函数在扩展结点处剪除不满足约束的子树；
- 用限界函数剪去得不到问题解或最优解的子树。

这两类函数统称为剪枝函数。

归纳起来，用回溯法解题的**一般步骤**如下：

- 1、对于给定的问题，确定问题的**解空间**，确定**解空间树**类型
（子集树or排列树），问题的解空间树应至少包含问题的一个（最优）解。
- 2、确定结点的扩展规则。
- 3、以**深度优先方式**搜索解空间树，并在搜索过程中可以采用**剪枝函数**来避免无效搜索。

回溯法 = 深度优先搜索 + 剪枝

实现回溯法的算法

回溯法是对解空间树的深度优先搜索法，通常有两种实现的算法。

- 递归回溯：采用递归的方法对解空间树进行深度优先遍历来实现回溯。
- 迭代回溯：采用**非递归**迭代过程对解空间树进行深度优先遍历来实现回溯。

回溯算法实现框架

1. 非递归回溯框架

```

int x[n];
void backtrack(int n)
{
    int i=1;
    while (i>=1)
    {
        if(ExistSubNode(t))
        {
            for (j=下界;j<=上界;j++)
            {
                x[i]取一个可能的值;
                if (constraint(i) && bound(i))
                {
                    if (x是一个可行解)
                        输出x;
                    else i++;
                }
            }
        }
        else i--;
    }
}

```

//x存放解向量，全局变量
 //非递归框架
 //根结点层次为1
 //尚未回溯到头
 //当前结点存在子结点
 //对于子集树，j=0到1循环
 //x[i]满足约束条件或界限函数
 //进入下一层次搜索
 //回溯：不存在子结点，返回上一层

Chapter 5

迭代回溯

回溯算法实现（书）

算法5.4 用非递归迭代实现回溯的框架

```
void iterativeBacktrack ()
```

```
{
```

```
    int t=1;
```

```
    while (t>0)
```

```
    {
```

当前扩展结点处未搜索过的子树的起始

```
        if (f(n, t)<=g(n, t))
```

当前扩展结点处未搜索过的子树的终止。

```
        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++; //代表要深入到下一层进行搜索
```

```
            }
```

```
        }
```

else t--; //如果本层扩展结点的所有子树都搜索完，则退回到上一层，找上一层结点的下一棵子树进行搜索。当t--,t为0时，代表搜索完毕

```
    }
```

回溯算法实现框架

2. 递归的算法框架

(1) 解空间为子集树

```
int x[n];  
void backtrack(int i)  
{  
    if(i>n)  
        输出结果;  
    else  
    {  
        for (j=下界;j<=上界;j++)  
        {  
            x[i]=j;  
            ...  
            if (constraint(i) && bound(i))  
                backtrack(i+1);  
        }  
    }  
}
```

//x存放解向量, 全局变量
//求解子集树的递归框架
//搜索到叶子结点, 输出一个可行解
//用j枚举i所有可能的路径
//产生一个可能的解分量
//其他操作
//满足约束条件和限界函数, 继续下一层

回溯算法实现框架

2. 递归的算法框架

解空间为子集树示例

【例】 有一个含 n 个整数的数组 a ，所有元素均不相同，设计一个算法求其所有子集（幂集）。

例如， $a[] = \{1, 2, 3\}$ ，所有子集是： $\{\}$ ， $\{3\}$ ， $\{2\}$ ， $\{2, 3\}$ ， $\{1\}$ ， $\{1, 3\}$ ， $\{1, 2\}$ ， $\{1, 2, 3\}$ （输出顺序无关）。



回溯算法实现框架

2. 递归的算法框架

解空间为子集树算法示例

【算法思想】 解：显然本问题的解空间为子集树，每个元素只有两种扩展，要么选择，要么不选择。

采用深度优先搜索思路。解向量为 $x[]$ ， $x[i]=0$ 表示不选择 $a[i]$ ， $x[i]=1$ 表示选择 $a[i]$ 。

用 i 扫描数组 a ，也就是说问题的初始状态是（ $i=0$ ， x 的元素均为0），目标状态是（ $i=n$ ， x 为一个解）。从状态（ i ， x ）可以扩展出两个状态：

- 不选择 $a[i]$ 元素 \Rightarrow 下一个状态为（ $i+1$ ， $x[i]=0$ ）。
- 选择 $a[i]$ 元素 \Rightarrow 下一个状态为（ $i+1$ ， $x[i]=1$ ）。

回溯算法实现框架

2. 递归的算法框架

解空间为子集树算法示例

求子集【算法描述】

```
void dfs(int a[],int n,int i,int x[])  
//首次调用, i为0,X[]=0  
//回溯算法求解向量x  
{  if (i>=n)  
    dispasolution(a,n,x); //输出解  
    else  
    {  x[i]=0; dfs(a,n,i+1,x);           //不选择a[i]  
        x[i]=1; dfs(a,n,i+1,x);         //选择a[i]  
    }  
}
```

回溯算法实现框架

2. 递归的算法框架

(2) 解空间为排列树

```
int x[n];  
void backtrack(int i)  
{  
    if(i>n)  
        输出结果;  
    else  
    {  
        for (j=i;j<=n;j++)  
        {  
            ...  
            swap(x[i],x[j]);  
            if (constraint(i) && bound(i))  
                backtrack(i+1);  
            swap(x[i],x[j]);  
            ...  
        }  
    }  
}
```

//x存放解向量, 并初始化
//求解排列树的递归框架
//搜索到叶子结点, 输出一个可行解

//用j枚举i所有可能的路径
//第i层的结点选择x[j]的操作
//为保证排列中每个元素不同, 通过交换来实现
//满足约束条件和限界函数, 进入下一层
//回溯(退层)恢复状态
//第i层的结点选择x[j]的恢复操作

回溯算法实现框架

2. 递归的算法框架

解空间为排列树示例

【例】有一个含 n 个整数的数组 a ，所有元素均不相同，求其所有元素的全排列。

例如， $a[] = \{1, 2, 3\}$ ，得到结果是 $(1, 2, 3)$ 、 $(1, 3, 2)$ 、 $(2, 3, 1)$ 、 $(2, 1, 3)$ 、 $(3, 1, 2)$ 、 $(3, 2, 1)$ 。

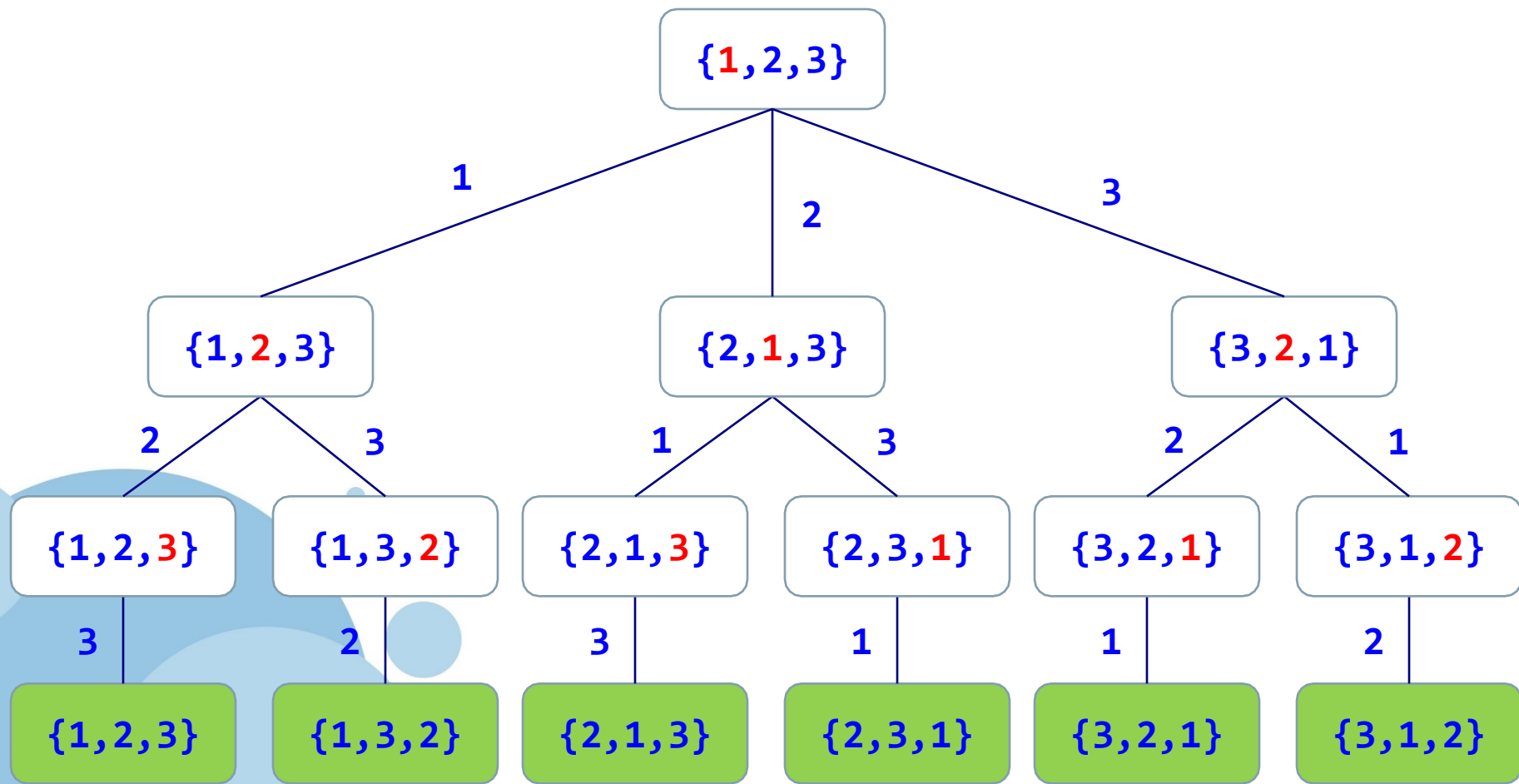


回溯算法实现框架

2. 递归的算法框架

解空间为排列树示例

产生了所有的解



回溯算法实现框架

2. 递归的算法框架

解空间为排列树示例

求a[]中元素的全排列【算法描述】

```
void dfs(int a[],int n,int i)           //求a[0..n-1]的全排列
{   if (i>=n)                           //递归出口
    dispasolution(a,n);
    else
    {   for (int j=i;j<n;j++)
        {   swap(a[i],a[j]);             //交换a[i]与a[j]
            dfs(a,n,i+1);
            swap(a[i],a[j]);             //交换a[i]与a[j]: 恢复
        }
    }
}
```

递归回溯

算法5.3 用递归实现回溯的框架

```
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);
            if (constraint(t)&&bound(t))
                backtrack(t+1);
        }
}
```

当前扩展结点处未搜索过的子树的起始

当前扩展结点处未搜索过的子树的终止。

【相同点】

回溯法在实现上也是遵循**深度优先**的，即一步一步往前探索，而不像广度优先遍历那样，由近及远地搜索。

【不同点】

- (1) **访问序不同**：深度优先遍历目的是“遍历”，本质是无序的。而回溯法目的是“求解过程”，本质是有序（按序搜索）的。
- (2) **访问次数的不同**：深度优先遍历对已经访问过的顶点不再访问，所有顶点仅访问一次。而回溯法中已经访问过的顶点可能再次访问。
- (3) **剪枝的不同**：深度优先遍历不含剪枝，而很多回溯算法采用剪枝条件剪除不必要的分枝以提高效能。

通常以回溯算法的**解空间树中的结点数**作为算法的时间分析依据，假设解空间树共有 n 层。

第1层有 m_0 个满足约束条件的结点，每个结点有 m_1 个满足约束条件的结点；

第2层有 m_0m_1 个满足约束条件的结点，同理，第3层有 $m_0m_1m_2$ 个满足约束条件的结点。

第 n 层有 $m_0m_1\dots m_{n-1}$ 个满足约束条件的结点，则采用回溯法求所有解的算法的执行时间为

$$T(n) = m_0 + m_0m_1 + m_0m_1m_2 + \dots + m_0m_1m_2\dots m_{n-1}。$$

通常情况下,回溯法的效率会高于蛮力法（通过剪枝函数减少不必要的搜索）

Case1：解空间为子集树：算法时间复杂度为： $O(2^n)$ 。

Case2：解空间为排列树：算法时间复杂度为： $O(n!)$

【问题描述】

设一个羽毛球队有男女运动员各 n 人，给定2个 $n*n$ 矩阵 P 和 Q ，其中 $P[i][j]$ 表示男运动员 i 和女运动员 j 配对组成混合双打时的竞赛优势， $Q[i][j]$ 则是女运动员 i 和男运动员 j 配对组成混合双打时的竞赛优势。由于技术的配合和心理状态等各种因素的影响，一般 $P[i][j]$ 不一定与 $Q[j][i]$ 相等。

设计一个算法，计算出男女运动员的最佳配对方法，使各组男女双方竞赛优势乘积的总和达到最大。

运动员最佳配对问题

- 例如：运动员最佳配对问题：

问题的已知和前提

P矩阵

	女 1	女 2	女 3
男 1	2	1	3
男 2	5	4	1
男 3	1	3	6

图5-4 竞争优势P[i][j]

Q矩阵

	男 1	男 2	男 3
女 1	2	3	2
女 2	3	4	7
女 3	1	2	6

图5-5 竞争优势Q[i][j]

此为已知的混合配对的竞争优势

	女 1	女 2	女 3
男 1	4	3	3
男 2	15	16	2
男 3	2	21	36

图5-6 混合竞争优势 $F[i][j]=P[i][j]*Q[j][i]$

最优搭配为：

男1—女1，

男2—女2，

男3—女3.

最优搭配下的最大优势乘积的总和是56

回溯算法示例

运动员最佳配对问题

【要解决的问题】

- 设计一个解决方案（配对方案），使得配对的男女运动员双方的混合竞赛优势 $F[i][j]$ 的总和达到最大。
- 寻求一个解向量 X ($x_1, x_2, x_3, \dots, x_n$)使得

$$\text{Max} \sum_{i=1}^n F[i][j]$$

回溯算法示例

运动员最佳配对问题

【定义问题的解空间】

运动员最佳配对问题, 当男女运动员各 n 人时, 假设固定男运动员的顺序为 $1, 2, 3, 4, 5, 6, \dots, n$, 那么该问题的求解就是对 n 个女运动员 $(1, 2, 3, \dots, n)$ 的全排列问题, 所以其解空间可以组织成一棵 $n+1$ 层的排列树, 其中最后一层的叶子标志着一种配对方案的形成, 无实际意义。

2. 确定解空间树的结构

从根结点出发到该排列树的任一叶结点对应了一个运动员的配对方案，其中第 i 层结点表示第 i 个男运动员，从第 i 层结点到第 $i+1$ 层接点的连线表示与第 i 个男运动员相配对的女运动员 j (j 为连线上的标号)

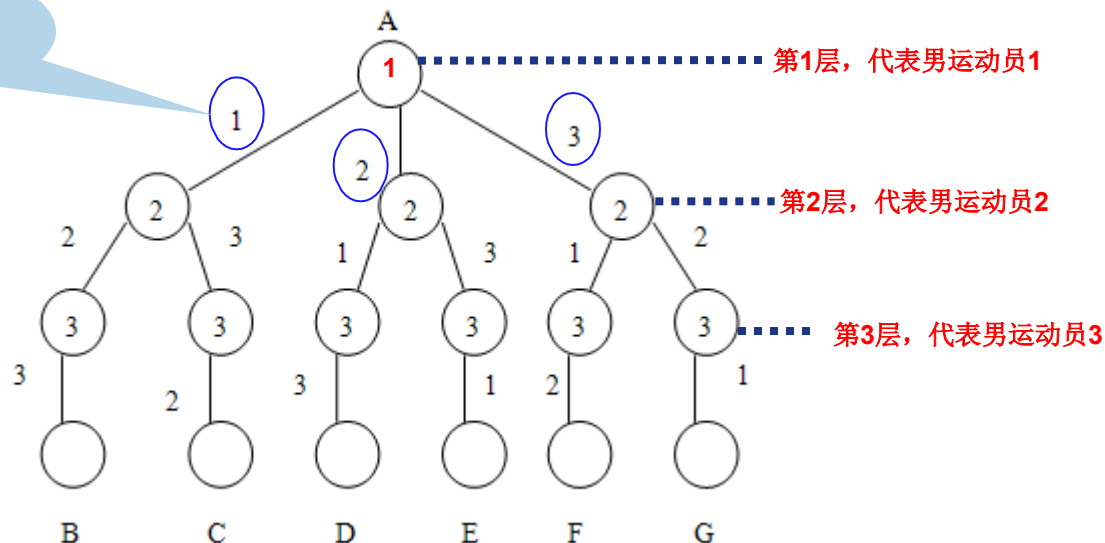
边上的标号为与男运动员（结点）配对的女运动员的编号

下标

x

1	2	3

下标 i 代表的是男运动的编号为 i ， $x[i]$ 代表与该男运动员配对的女运动员的编号。右图：以3个运动员配对为例。

图5-7 $n=3$ 时的解空间

运动员最佳配对问题

【搜索解空间树】即如何寻优？

在解空间树中，若当前的层数 $i > n$ 时，则说明已经找到了一个运动员配对方案，此时只需判断其是否是最优解，

- 设用变量 **cc** 存放当前男女运动员配对双方竞赛优势乘积的总和，
- 用变量 **bestc** 最优值，即存放竞赛优势乘积总和的最大值，
- 在搜索过程中，cc 和 bestc 中存放相应的**当前值**与**当前最优值**，此时若 $cc > bestc$ 则说明当前的最优方案已不再最优，此时就用找到的方案来更新当前的最优方案；否则仍然保持以前的最优值。

- 若 $i < n$ 且 $cc < bestc$ ，则按照深度优先的策略继续往下搜索，否则，回溯到该结点的父结点。上述过程一直地进行下去，直到所有路径均被检查过，就可以得到一个最优方案。

回溯算法示例

运动员最佳配对问题

4. 算法的设计与实现

```
void backtrack(int i) //i=1开始进行搜索
{
```

```
    int j;
```

```
    if (i > n) //当前已经找到一个可行解
```

```
    { if (cc > bestc) {
```

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

```
            bestx[j]=x[j];
```

```
            bestc=cc;
```

```
    }
```

```
    else
```

```
    {
```

```
        for (j=i; j<=n; j++)
```

```
        {
```

```
            swap(&x[i], &x[j]);
```

```
            cc+=F[i][x[i]];
```

```
            backtrack(i+1);
```

```
            cc-=F[i][x[i]];
```

```
            swap(&x[i], &x[j]);
```

```
        }
```

```
    }
```

```
}
```

cc:存放正在计算的混合竞争优势，
当找到一个排列时，**CC**就是当前配对方案下的 $\sum_{i=1}^n F[i][j]$

bestc:当前搜索到的可行解中的最佳配对方案对应的混合竞争优势之和 $\sum_{i=1}^n F[i][j]$

x[]:正在搜寻的解向量;

best[]:当前最佳方案对应的解向量

算法性能: **O (n!)**

算法 5.5 解运动员最佳配对问题的递归回溯法

子集和问题

【问题描述】

给定 n 个不同的正数集 $W = \{w(i) \mid 1 \leq i \leq n\}$ 和正数 M ，子集和问题是要求找出正数集 W 子集 S ，使该子集中所有元素的和为 M ，即
$$\sum_{u \in S} w(i) = C$$

例如：

当 $n=4$ ， $(w_1, w_2, w_3, w_4) = (11, 13, 24, 7)$ ，

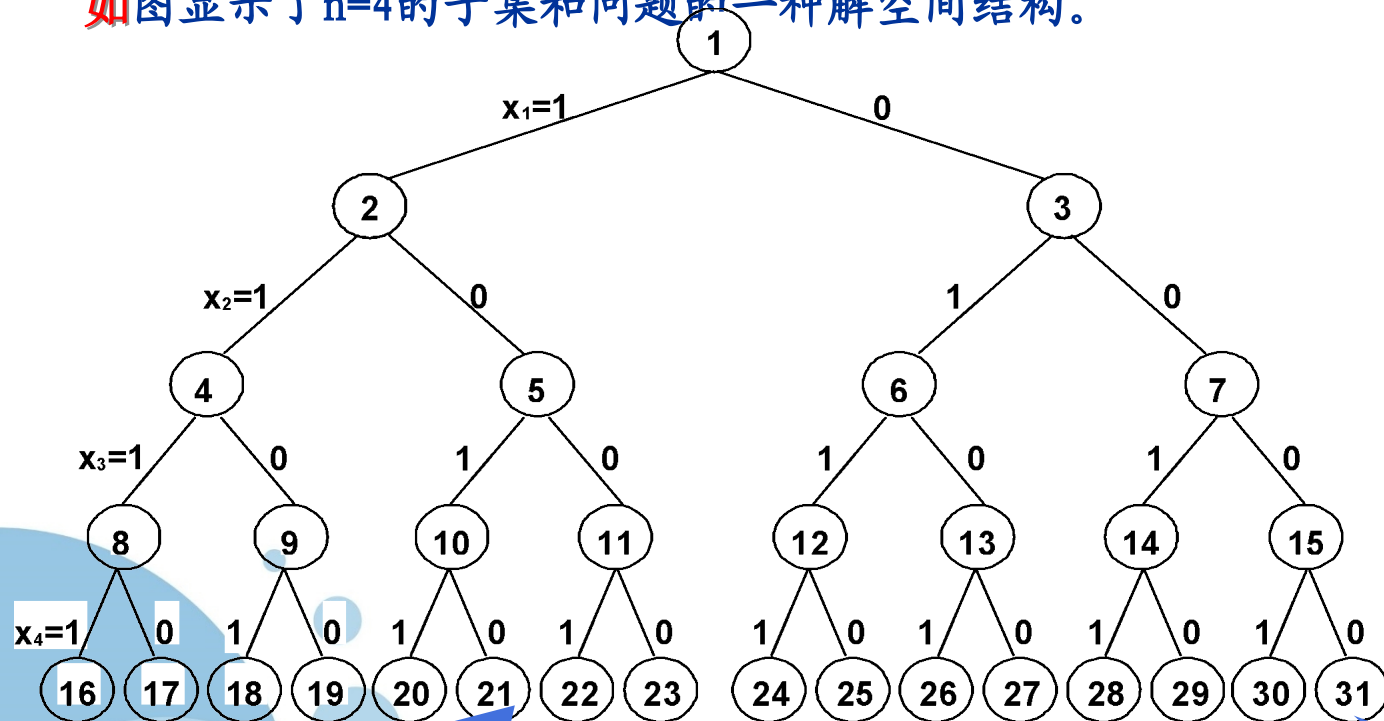
$C=31$ ，则满足要求的子集 $(11, 13, 7)$ 和 $(24, 7)$ 。

【定义问题的解空间】

子集和问题是从 n 个元素的集合中找出满足某种性质的子集，其相应的解空间树为子集树。该问题的另一种表示是，每个解的子集由这样一个 n 元组 (x_1, x_2, \dots, x_n) 表示，其中 $x_i \in \{0, 1\}$ ， $1 \leq i \leq n$ 。如果解中含有 w_i ，则直 $x_i=1$ ，否则 $x_i=0$ 。例如前面实例的解可以表示为 $(1, 1, 0, 1)$ 和 $(0, 0, 1, 1)$ 。

【确定解空间树的结构】

如图显示了 $n=4$ 的子集和问题的解空间结构。



搜索效率为 $O(n \cdot 2^n)$

图5-8 子集和问题的解空间结构

16个叶子结点

从根走到一个叶子结点需要的时间耗费为 n

共 2^n 个叶子点

Chapter5 回溯算法示例

子集和问题

【搜索解空间树】

为解决该问题可以使用递归回溯的方法来构造最优解，设 cs 为当前子集和，在解空间树中进行搜索时，若当前层 $i > n$ 时，算法搜索至叶节点，其相应的子集和为 cs 。当 $cs = c$ ，则找到了符合条件的解。

当 $i \leq n$ 时，当前扩展结点 Z 是子集树的内部结点。该结点有 $x[i] = 1$ 和 $x[i] = 0$ 两个儿子结点。其左儿子结点表示 $x[i] = 1$ 的情形。

剪枝函数

(1) 约束函数

(2) 限界函数

Chapter5 回溯算法示例

子集和问题

4. 【算法的设计与实现】

设置一个全局变量 r 来记录剩余元素的和，初值为集合中所有元素的和。

$$r = \sum_{i=1}^n w_i$$

算法 5.6解子集和问题的递归回溯法

void Subsum::backtrack(int i) //i=1开始调用

```
{
    if (i > n) //走到叶子点了，找到一个解
    {
        if (cs == c) //判断该解是否可行。
        {
            for(int j=1; j <= n; j++)
                if(x[j] == 1)
                    cout << w[j] << " "; //输出结果
            cout << endl; }
        return ;
    }
}
```

找到一个解，
输出解

否则，进层深度优先搜索（确定是否需要加入第 i 个元素的值，来构造子集和

约束

限界

```
r -= w[i]; //递归进层搜索
```

```
if (cs+w[i]<=c) //检测左子树
```

```
{
```

```
    x[i] = 1;
```

```
    cs += w[i];
```

```
    backtrack(i+1);
```

```
    cs -= w[i]; }
```

```
if (cs+r>=c) //检测右子树
```

```
{
```

```
    x[i] = 0;
```

```
    backtrack(i+1); //深度优先遍历，递归处理
```

右子树

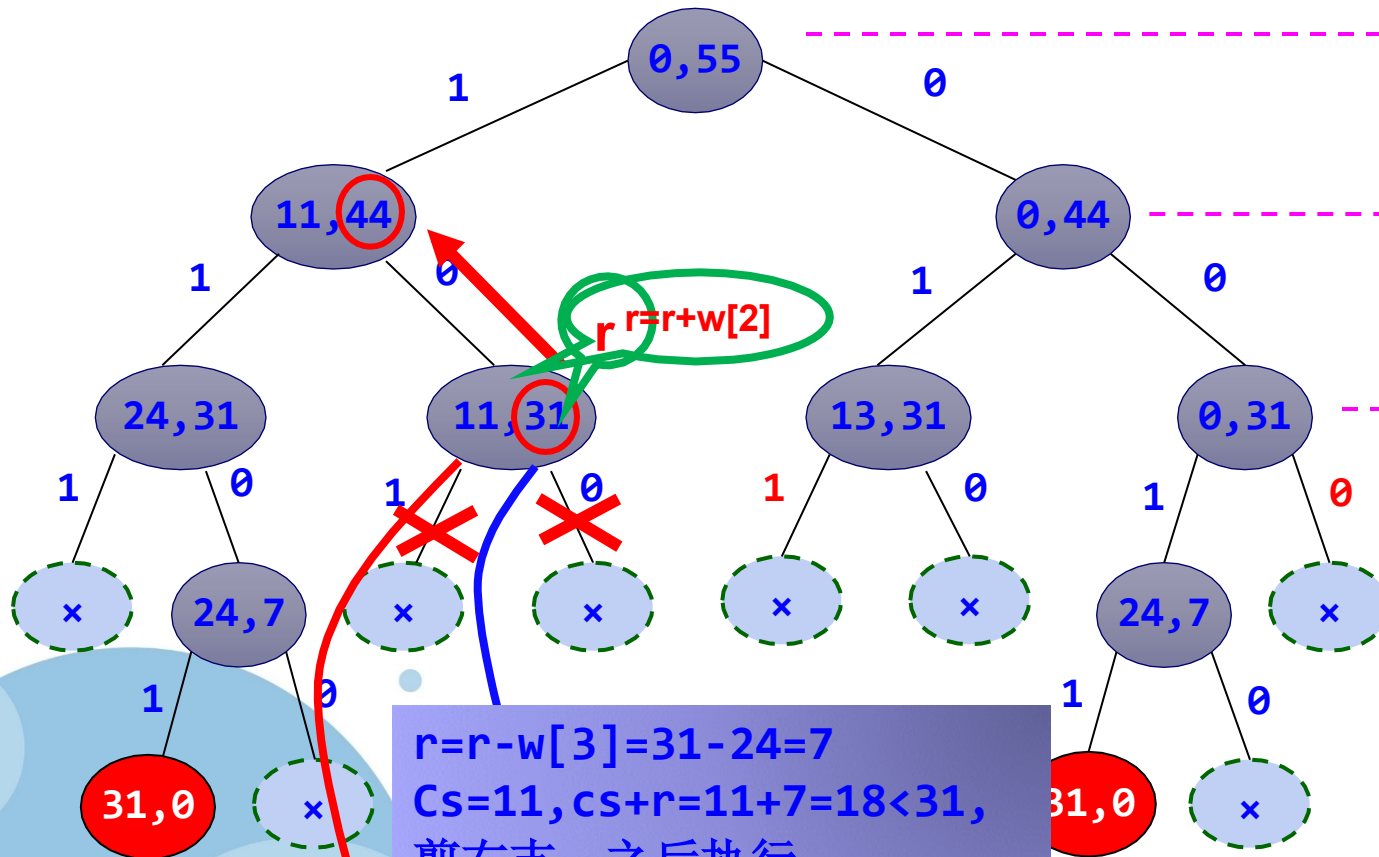
```
}
```

```
r += w[i]; //递归退层时：将该段加入剩余路径r  
中即第i层的扩展结点均处理完毕。
```

```
return ;}
```

```
int n=4,C=31;
int w[]={0,11,13,24,7};
```

$cs=0, r=11+13+24+7=55$



$i=1, w[1]=11$

决策 $w[1]$ 是否出现在子集和中。

$i=2, w[2]=13$

决策 $w[2]$ 是否出现在子集和中

$i=3, w[3]=24$

决策 $w[3]$ 是否出现在子集和中

$i=4, w[4]=7$

决策 $w[4]$ 是否出现在子集和中

$r=r-w[3]=31-24=7$
 $Cs=11, cs+r=11+7=18<31$,
 剪右支。之后执行
 $r=r+w[3]=31$ 恢复 r 值。

$Cs=11, w[3]=24$
 $cs+w[3]=11+24=35>31$,
 剪左支。

N皇后问题**【问题描述】**

N 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，可以简单的描述为：在 $N \times N$ 格的棋盘上摆放N个皇后，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，问有多少种摆法？

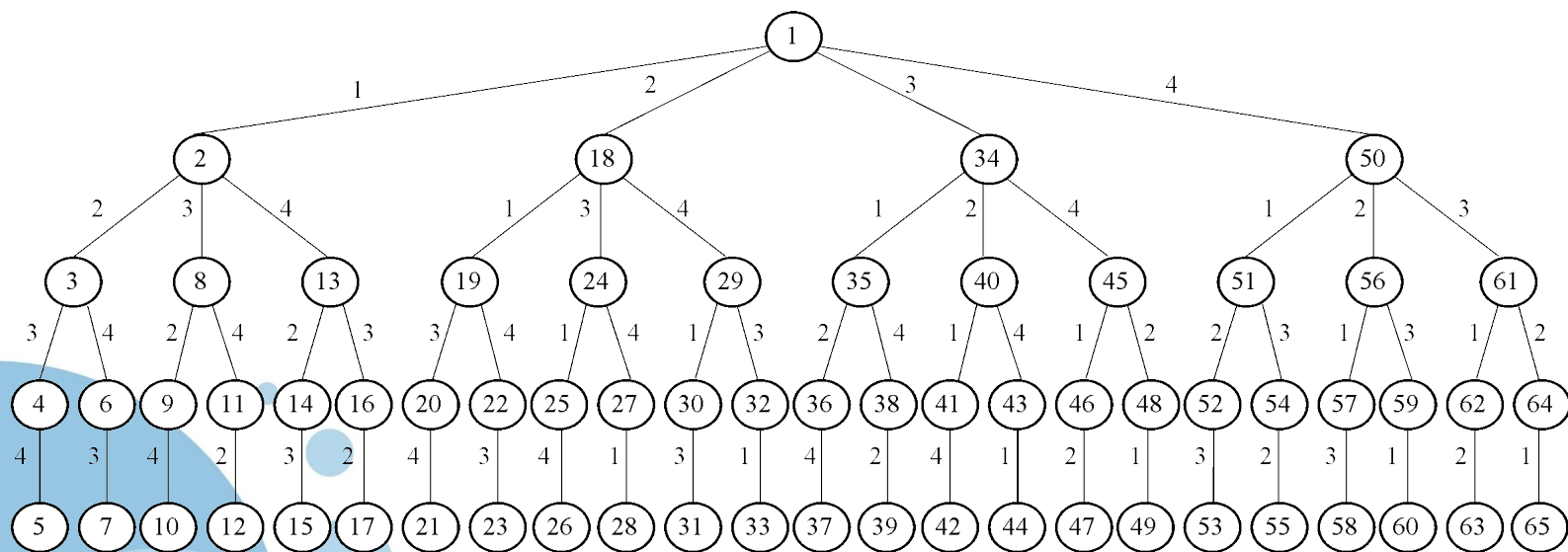
- 【定义问题的解空间】

以8皇后为例，可以用一棵树表示8皇后问题的解空间。假设皇后 i 放在第 i 行上，8皇后问题可以表示成8元组 (X_1, X_2, \dots, X_8) ，其中 $X_i (i=1, 2, \dots, 8)$ 表示皇后 i 所放位置的列号，此时该问题的解空间为 8^8 个8元组。

加上隐式约束条件：没有两个 X_i 相同，且不存在两个皇后在同一条对角线上，因此问题的解空间进一步减小为 $8!$ 。由于8皇后问题的解空间为 $8!$ 种排列，因此我们要构造一棵排列树。

N皇后问题

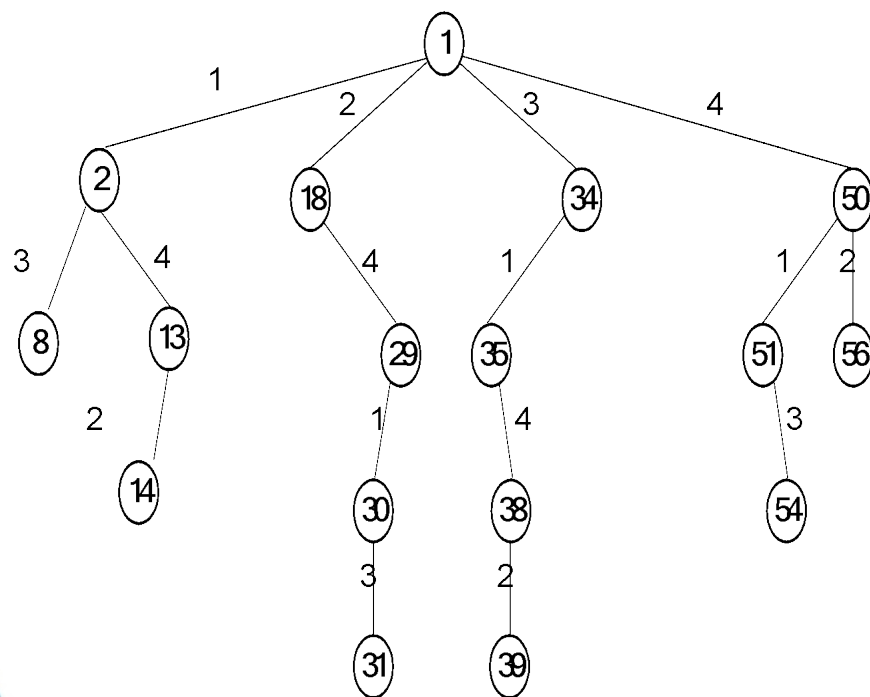
【确定解空间树的结构】

 $n=4$ 时问题的一种空间树结构。

回溯算法示例

N皇后问题

【确定解空间树的结构】

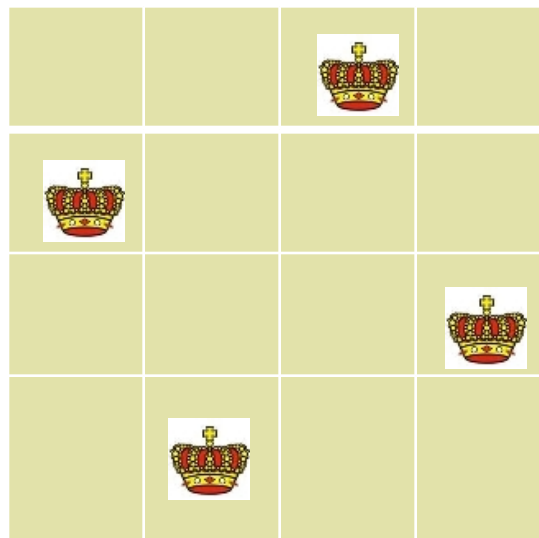
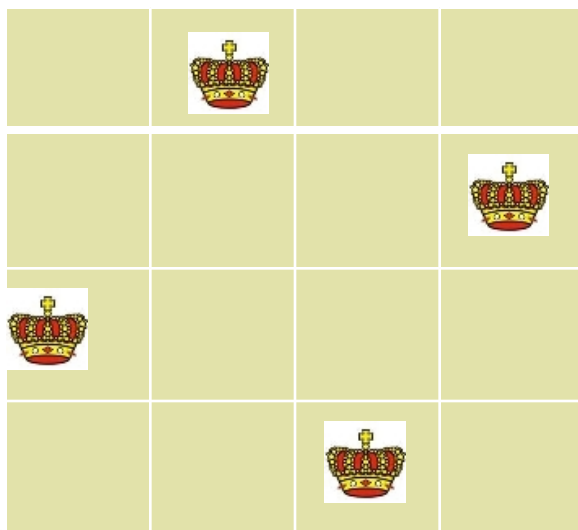


具有界函数的4皇后问题的状态空间树

回溯算法示例

N皇后问题

- 皇后问题的解示例



4皇后问题的解图示

【搜索解空间树】

解n后问题的回溯算法可描述如下:

- 求解过程从空配置开始。
- 在第1个~第m个皇后为合理配置的基础上, 再配置第m+1个皇后, 直至第n个皇后也是合理时, 就找到了一个解。
- 在每个皇后的配置上, 顺次从第一行到第n行配置, 当第n行也找不到一个合理的配置时, 就要回溯, 去改变前一个皇后的配置。

【解向量】用 n 元组 $x[1:n]$ 表示 n 皇后问题的解, $x[i]$ 表示皇后 i 放在第 i 行的第 $x[i]$ 列上, 用完全 n 叉树表示解空间。

【剪枝函数设计】 对于两个皇后A、B 皇后A: $i, x[i]$ 皇后B: $k, x[k]$

- 两个皇后不同行: i 不等于 k ;
- 两个皇后不同列: j 不等于 L ;
- 两个皇后不同一条斜线

- $i \neq k$
- $X[i] \neq x[k]$
- $|i-k| \neq |x[i]-x[k]|$

算法 5.7 解N后问题的递归回溯算法

逐个取前 $t-1$ 个皇后的安放位置 $x[i]$ 与第 t 个皇后的安放位置 $x[t]$ 进行合法性检查，只要有冲突则返回0。若第 t 个皇后的安放位置和前面的 $t-1$ 个皇后都不冲突，就返回1。

}

1 2 3 4 t-1 t n-1 n



回溯算法示例

N皇后问题

算法 5.8 解N后问题的非递归迭代回溯算法

```
void Queen::queen(void)
```

```
{ x[1] = 0;
```

```
  int t = 1;
```

```
  while(t>0)
```

```
  {   x[t] += 1; //按列号递增进行试探,从列号1开始试探, 之后是列号2, 列号3.....一直到列号n
```

```
      while((x[t]<=n)&&!(Place(t))) x[t] += 1;
```

```
      if(x[t]<=n) //若给第t个皇后找到了合法的摆放地
```

```
          if(t==n) sum++; //sum用来记录有多少种摆法? 此处也可增加输出可行的摆法即解向量x[ ]
```

```
          else{ t++; x[t] = 0;}
```

```
      else
```

```
          t--;
```

```
  }
```

```
}
```



Chapter 5 回溯算法示例

N皇后问题

本程序一次执行结果如下：

皇后问题($n < 20$) $n=6$ ✓

6皇后问题求解如下：

第1个解：(1, 2) (2, 4) (3, 6) (4, 1) (5, 3) (6, 5)

第2个解：(1, 3) (2, 6) (3, 2) (4, 5) (5, 1) (6, 4)

第3个解：(1, 4) (2, 1) (3, 5) (4, 2) (5, 6) (6, 3)

第4个解：(1, 5) (2, 3) (3, 1) (4, 6) (5, 4) (6, 2)

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

(a) 第 1 个解

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

(b) 第 2 个解

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

(c) 第 3 个解

	1	2	3	4	5	6
1						
2						
3						
4						
5						
6						

(d) 第 4 个解

【问题描述】 有 n 个重量分别为 $\{w_1, w_2, \dots, w_n\}$ 的物品，它们的价值分别为 $\{v_1, v_2, \dots, v_n\}$ ，给定一个容量为 W 的背包。设计从这些物品中选取一部分物品放入该背包的方案，每个物品要么选中要么不选中，要求选中的物品不仅能够放到背包中，而且满足重量限制具有最大的价值。

回溯算法示例

0-1背包问题

case1. 装入背包中物品重量和要求恰好为 w

看到这个要求你
想到了什么？

用 $w[1..n]/v[1..n]$ 存放物品信息， $x[1..n]$ 数组存放最优解，其中每个元素取1或0， $x[i]=1$ 表示第 i 个物品放入背包中， $x[i]=0$ 表示第 i 个物品不放入背包中。

这是一个求最优解问题。找到更优解 $(op, tv) \Rightarrow (x, \max v)$ 。

该问题的解空间树是
排列树还是子集树？

0-1背包问题

【问题解空间】解空间是一棵子集树！

对第 i 层上的某个分枝结点，对应的状态为 $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$ ，其中 tw 表示装入背包中的物品总重量， tv 表示背包中物品总价值， op 记录一个解向量。该状态的两种扩展如下：

第 i 层结点： $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$

选择第 i 个物品：

$\text{op}[i]=1,$
 $\text{tw}=\text{tw}+w[i],$
 $\text{tv}=\text{tv}+v[i]$

不选择第 i 个物品：

$\text{op}[i]=0$
 tw 不变
 tv 不变

$\text{dfs}(i+1, \text{tw}, \text{tv}, \text{op})$

$\text{dfs}(i+1, \text{tw}, \text{tv}, \text{op})$ 第 $i+1$ 层结点

0-1背包问题

第 i 层结点: $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$

选择第 i 个物品:
 $\text{op}[i]=1$,
 $\text{tw}=\text{tw}+w[i]$,
 $\text{tv}=\text{tv}+v[i]$

不选择第 i 个物品:
 $\text{op}[i]=0$
 tw 不变
 tv 不变

第 $i+1$ 层结点

第 $n+1$ 层结点

- 叶子结点表示已经对 n 个物品做了决策。
- 对所有叶子结点进行比较求出满足 $\text{tw}==W$ 的最大 maxv ，对应的最优解 op 存放到 x 中。

Chapter 5 回溯算法示例

0-1背包问题

0/1背包问题 ($W=6$) :

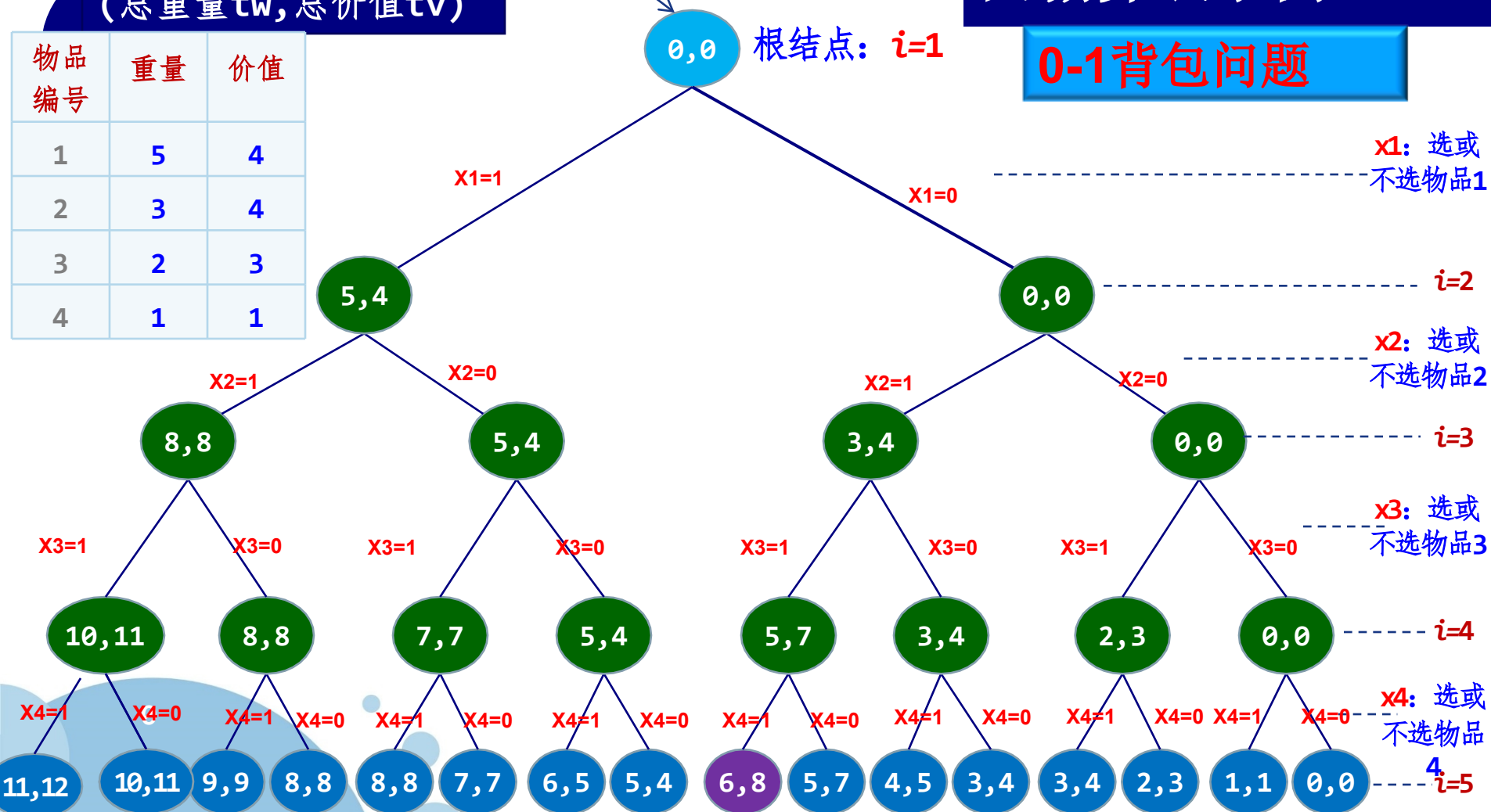
物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

回溯算法示例

0-1背包问题

(总重量tw, 总价值tv)

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1



最优解

$W=6$

//问题表示

int n=4;

int W=6;

int w[]={0,5,3,2,1};

int v[]={0,4,4,3,1};

//求解结果表示

int x[MAXN];

int maxv;

//4种物品

//限制重量为6

//存放4个物品重量,不用下标0元素

//存放4个物品价值,不用下标0元素

//存放最终解

//存放最优解的总价值

Chapter 回溯算法示例

0-1背包问题

采用解空间为子集树递归的算法框架

```
void dfs(int i,int tw,int tv,int op[])
{  if (i>n)                                //找到一个叶子结点
    {  if (tw==W && tv>maxv)                //找到一个满足条件的更优解,保存
        {  maxv=tv;
            for (int j=1;j<=n;j++)
                x[j]=op[j];
        }
    }
    else                                    //尚未找完所有物品
    {  op[i]=1;                             //选取第i个物品
        dfs(i+1,tw+w[i],tv+v[i],op);

        op[i]=0;                           //不选取第i个物品,回溯
        dfs(i+1,tw,tv,op);
    }
}
```

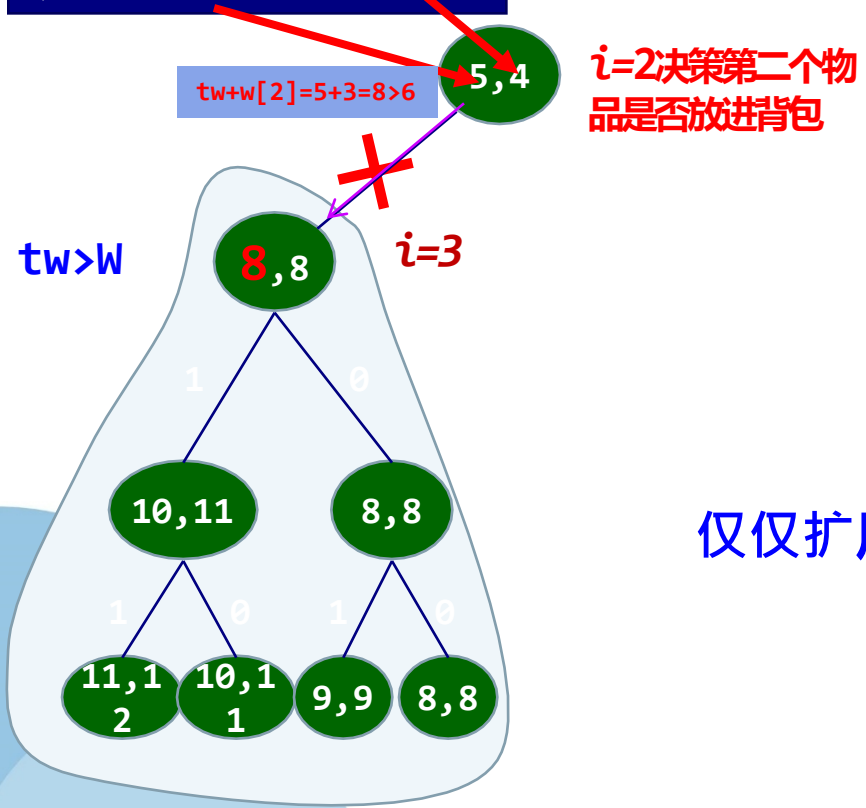
【算法分析】 该算法没考虑剪枝, 解空间树中有 $2^{n+1}-1$ 个结点, 所以算法的时间复杂度为 $O(2^n)$ 。

回溯算法示例

Chapter 5 0-1背包问题算法改进（增加剪枝函数）

改进1：左剪枝：对于第*i*层的有些结点， $tw+w[i]$ 已超过了 W （ $W=6$ ），显然再选择 $w[i]$ 是不合适的。如第2层的（5，4）结点 \Rightarrow 进行扩展是不必要的。

（总重量 tw ，总价值 tv ）



物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

仅仅扩展满足 $tw+w[i] \leq W$ 的左孩子结点

(总重量tw, 总价值tv)

0,0

根结点: $i=1$

x_1 : 选或不选物品1

5,4

0,0

$i=2$

x_2 : 选或不选物品2

x

5,4

3,4

0,0

$i=3$

x_3 : 选或不选物品3

物品
编号

重量

价值

1

5

4

2

3

4

3

2

3

4

1

1

x

5,4

5,7

3,4

2,3

0,0

$i=4$

x_4 : 选或不选物品4

6,5

5,4

6,8

5,7

4,5

3,4

3,4

2,3

1,1

0,0

$i=5$

最优解

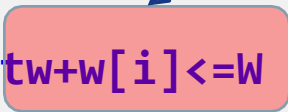
回溯算法示例

Chapter 5

0-1背包问题算法改进（增加剪枝函数）

```
void dfs(int i,int tw,int tv,int op[])
{
    if (i>n)                                //找到一个叶子结点
    {
        if (tw==W && tv>maxv)                //找到一个满足条件的更优解,保存
        {
            maxv=tv;
            maxw=tw;
            for (int j=1;j<=n;j++)
                x[j]=op[j];
        }
    }
    else
    {
        if ( tw+w[i]<=W )                    //尚未找完所有物品
                                            //左孩子结点剪枝
                                            //选取第i个物品
        {
            op[i]=1;
            dfs(i+1,tw+w[i],tv+v[i],op);
        }
        op[i]=0;                             //不选取第i个物品,回溯
        dfs(i+1,tw,tv,op);
    }
}
```

左剪枝



0-1背包问题算法改进（增加剪枝函数）



改进2: 右剪枝:

$$rw = w[i] + w[i+1] + \cdots + w[n]。$$

当不选择物品 i （第 i 个物品的决策已做出即不选择）时：

$rw - w[i] = w[i+1] + \cdots + w[n]$ ，若 $tw + rw - w[i] < W$ ，也就是说即使选择后面的所有物品，重量也不会达到 W ，因此不必要再考虑扩展这样的结点。



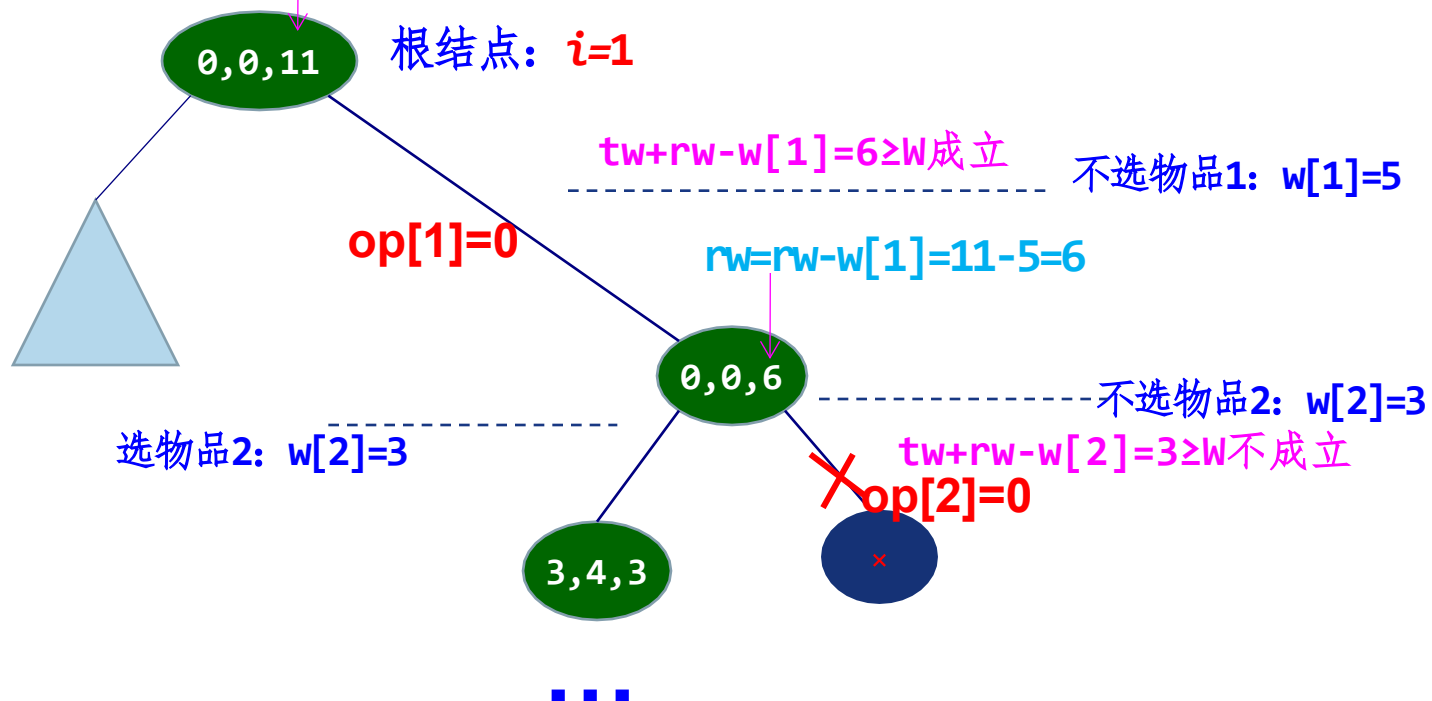
仅仅扩展满足 $tw + rw - w[i] \geq W$ 的右孩子结点

$W=6$

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

剩余物品的重量，初值为全部物品重量之和

$$rw=5+3+2+1=11$$

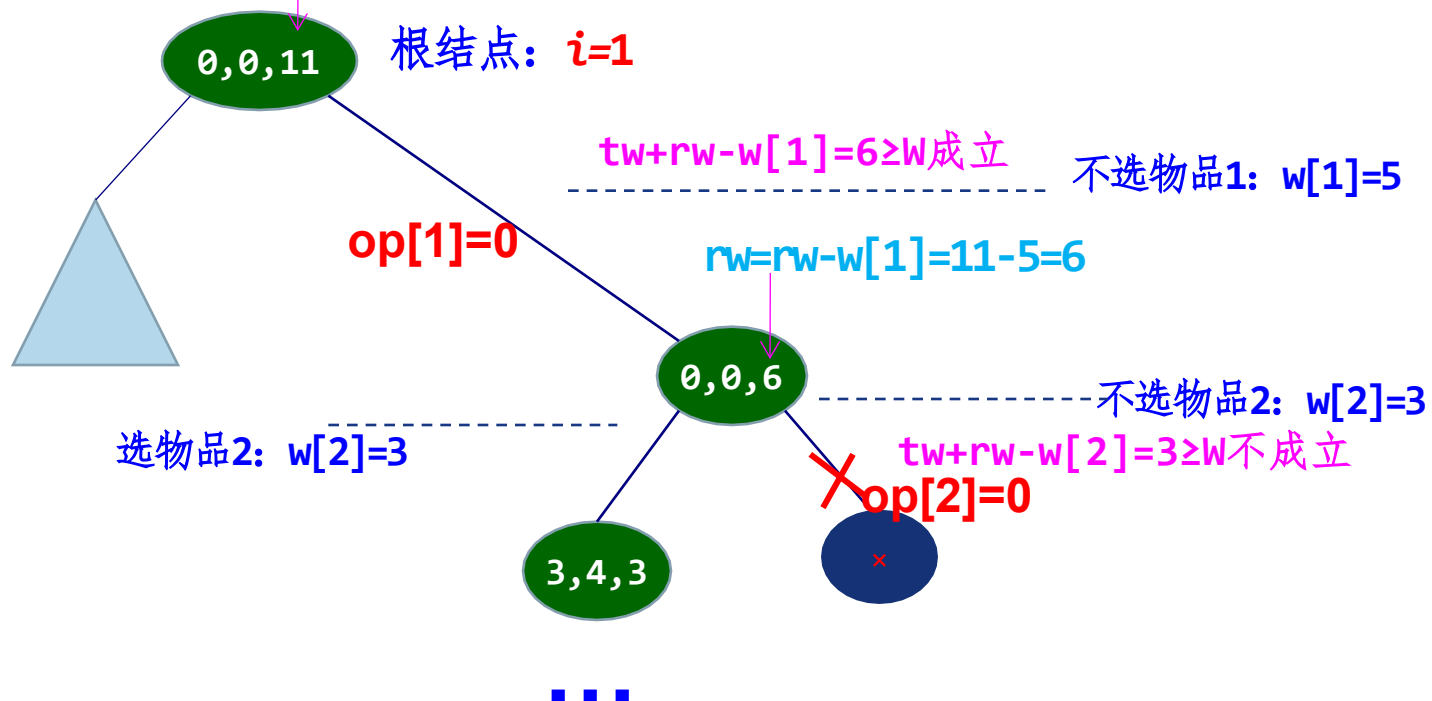


$W=6$

物品编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

剩余物品的重量，初值为全部物品重量之和

$$rw=5+3+2+1=11$$



当前已选
物品重量和

当前已选物
品价值和

剩余未选物品的重量和，初值为全部物品重量之和

(tw, tv, rw)

0,0,11

根结点: $i=1$

$op[1]=1$

$op[1]=0$

x_1 : 选或不
选物品1

5,4,6

0,0,6

$i=2$

$op[2]=1$

$op[2]=0$

$op[2]=1$

$op[2]=0$

x_2 : 选或不
选物品2

$i=3$

x

5,4,3

3,4,3

x

x_3 : 选或不
选物品3

$i=4$

$op[3]=1$

$op[3]=0$

$op[3]=1$

$op[3]=0$

x_4 : 选或不
选物品4

$i=5$

x

5,4,1

5,7,1

x

$op[4]=1$

$op[4]=0$

$op[4]=1$

$op[4]=0$

6,5,0

x

6,8,0

x

最优解

$W=6$

物品 编号	重量	价值
1	5	4
2	3	4
3	2	3
4	1	1

```

void dfs(int i,int tw,int tv,int rw,int op[])
{ //初始调用时rw为所有物品重量和
  int j;
  if (i>n)                                     //找到一个叶子结点
  { if (tw==W && tv>maxv)                       //找到一个满足条件的更优解,保存
    { maxv=tv;
      for (j=1;j<=n;j++)                       //复制最优解
        x[j]=op[j];
    }
  }
  else                                         //尚未找完所有物品
  { if (tw+w[i]<=W)                             //左孩子结点剪枝
    { op[i]=1;                                  //选取第i个物品
      dfs(i+1,tw+w[i],tv+v[i],rw-w[i],op);
    }
    if ( tw+rw-w[i]>=W )                       ← 右剪枝
    { op[i]=0;                                  //不选取第i个物品,回溯
      dfs(i+1,tw,tv,rw-w[i],op);
    }
  }
}
}

```

0-1背包问题算法改进（增加剪枝函数）

【算法分析】 该算法不考虑剪枝时解空间树中有 $2^{n+1}-1$ 个结点（剪枝的结点个数不确定），所以最坏情况下算法的时间复杂度为 $O(2^n)$ 。

要求背包装满且价值最大的0-1背包问题

【其实质】

要求背包装满（是前面的子集和问题，在这里是0-1背包的约束条件）

- 在满足子集和（物品的重量之和为 W ）的约束下，
- 寻求所选的物品的价值最大的子集和（物品集合）才是这个0-1背包问题的最优解。

回溯算法示例

0-1背包问题

case2. 装入背包中物品重量和不超过 W

问题变为求背包中物品重量和不超过 W 的最大价值的装入方案：

- 前面左剪枝方式不变。
- 但右剪枝方式不再有效，改为采用上界函数进行右剪枝。

基于子集和（选择放进背包的物品重量和为 W ）的右剪枝

寻找新的右剪枝函数

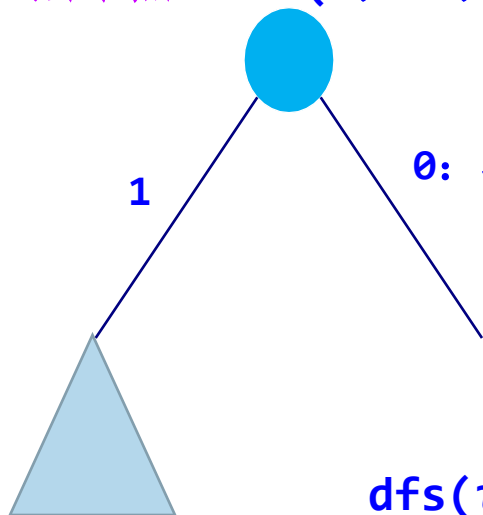
基于最大价值的右剪枝

回溯算法示例

0-1背包问题

case2. 装入背包中物品重量和不超过 W

第 i 层结点: $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$



0: 不选择第 i 个物品

第 $i+1$ 层结点

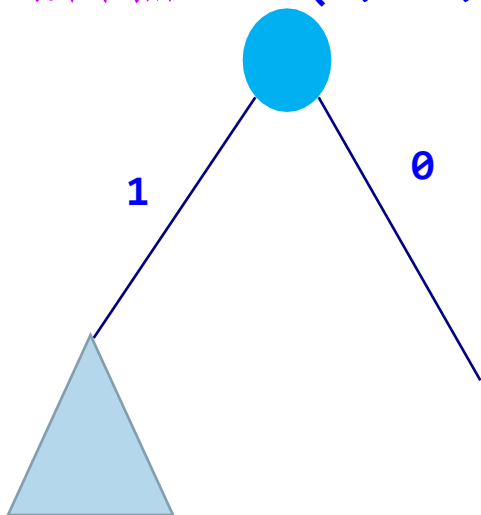
$\text{dfs}(i+1, \text{tw}, \text{tv}, \text{op})$

- 上界函数 $\text{bound}(i) = \text{tv} + r$
- 表示沿着该方向选择得到物品的价值上界
- r 表示剩余物品的总价值

0-1背包问题

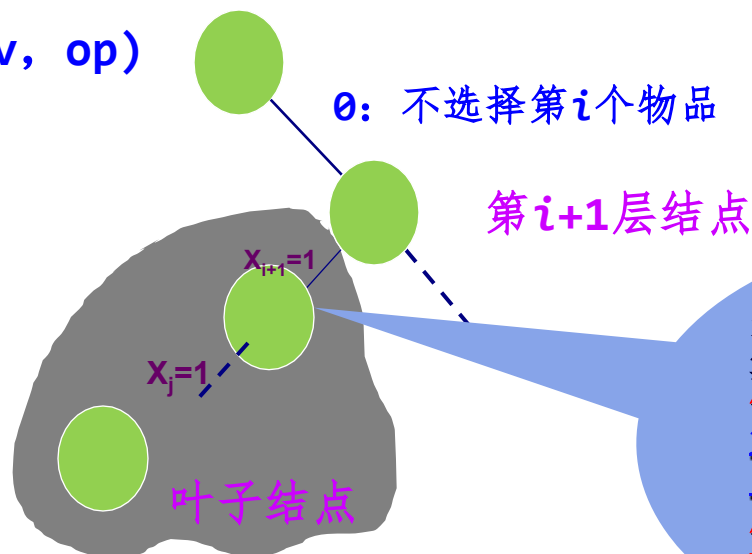
case2. 装入背包中物品重量和不超过 W

第 i 层结点: $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$



- 上界函数 $\text{bound}(i) = \text{tv} + r$
- 假设当前求出最大价值 maxv , 若 $\text{bound}(i) \leq \text{maxv}$, 则右剪枝, 否则继续扩展。
- 显然 r 越小, $\text{bound}(i)$ 也越小, 剪枝越多, 为了构造更小的 r , 将所有物品以单位重量价值递减排列。

第*i*层结点: $\text{dfs}(i, \text{tw}, \text{tv}, \text{op})$



当第*i*个物品不选的时候，计算一个背包物品价值的上界值（即剩余的*i+1*个物品到第*n*个物品“都选”所能达到的物品价值的上界。）所谓的“都选”还要受到所选物品的重量之和不能超过*W*。

```
int bound(int i,int tw,int tv)
{  i++;
  while (i<=n && tw+A[i].w<=W)
  {  tw+=A[i].w;
    tv+=A[i].v;
    i++;
  }
  if (i<=n)
    return tv+(W-tw)*A[i].p;
  else
    return tv;
```

//求上界

//从*i+1*开始

//若序号为*i*的物品可以整个放入

计算部分放入时，是背包满的时候所能达到的物品价值的最大值（因为是0-1背包，所以计算出来的价值的上界，其实是背包达不到的一个最大价值数）。

//序号为*i*的物品不能整个放入，则

剪枝:

- **左剪枝:** 仅仅扩展 $tw + w[i] \leq W$ 的左孩子结点
- **右剪枝:** 仅仅扩展 $bound(i, tw, tv) > maxv$ 的右孩子结点

回溯算法示例

Chapter 5

0-1背包问题

case2. 装入背包中物品重量和不超过 W

```
void dfs(int i,int tw,int tv,int op[]) //求解0/1背包问题
{
    if (i>n) //找到一个叶子结点
    {
        maxv=tv; //存放更优解
        for (int j=1;j<=n;j++)
            x[j]=op[j];
    }
    else //尚未找完所有物品
    {
        if (tw+A[i].w<=W) //不满足此条件则左孩子结点剪枝
        {
            op[i]=1; //选取序号为i的物品
            dfs(i+1,tw+A[i].w,tv+A[i].v,op);
        }
        if (bound(i,tw,tv)>maxv) //不满足此条件则右孩子结点剪枝
        {
            op[i]=0 ; //满足条件则不选取序号为i的物品,继续进层搜索,考虑第i+1
            //个物品的决策。
            dfs(i+1,tw,tv,op);
        }
    }
}
```

Chapter 5

0/1背包问题 ($W=6$) :

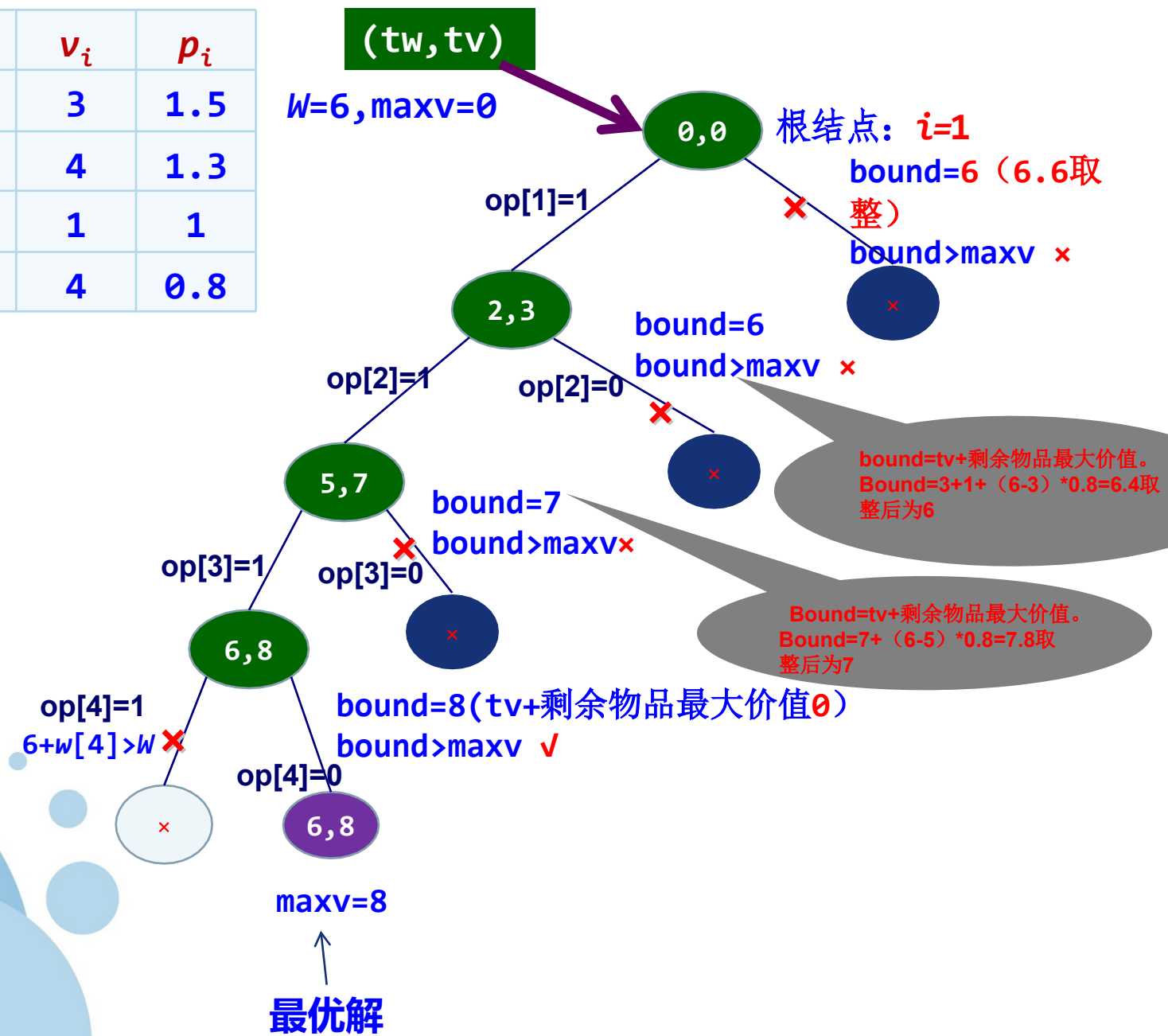
物品编号	重量	价值	$p=v/w$
1	5	4	0.8
2	3	4	1.3
3	2	3	1.5
4	1	1	1



A数组
:

i	no	w_i	v_i	p_i
1	3	2	3	1.5
2	2	3	4	1.3
3	4	1	1	1
4	1	5	4	0.8

i	no	w_i	v_i	p_i
1	3	2	3	1.5
2	2	3	4	1.3
3	4	1	1	1
4	1	5	4	0.8



【算法分析】 该算法不考虑剪枝时解空间树中有 $2^{n+1}-1$ 个结点（剪枝的结点个数不确定），所以最坏情况下算法的时间复杂度为 $O(2^n)$ 。

求解简单装载问题

【问题描述】 有 n 个集装箱要装上一艘载重量为 W 的轮船，其中集装箱 i ($1 \leq i \leq n$) 的重量为 w_i 。不考虑集装箱的体积限制，现要从这些集装箱中选出重量和小于等于 W 并且尽可能大的若干装上轮船。

例如， $n=5$ ， $W=10$ ， $w=\{5, 2, 6, 4, 3\}$ 时，其最佳装载方案是 $(1, 1, 0, 0, 1)$ 或者 $(0, 0, 1, 1, 0)$ ， $\max w=10$ 。

Chapter 5

【问题求解】 采用带剪枝的回溯法求解。问题的表示如下：

```
int w[]={0, 5, 2, 6, 4, 3}; //各集装箱重量，不用下标0的元素  
int n=5, W=10;
```

求解的结果表示如下：

```
int maxw=0; //存放最优解的总重量  
int x[MAXN]; //存放最优解向量
```

将上述数据设计为全局变量。

Chapter 5

求解算法如下：

```
void dfs(int i, int tw, int rw, int op[])
```

其中参数*i*表示考虑的集装箱*i*，*tw*表示选择的集装箱重量和，*rw*表示剩余集装箱的重量和（初始时为全部集装箱重量和），*op*表示一个解，即对应一个装载方案。

最优解： *x*, *maxw*

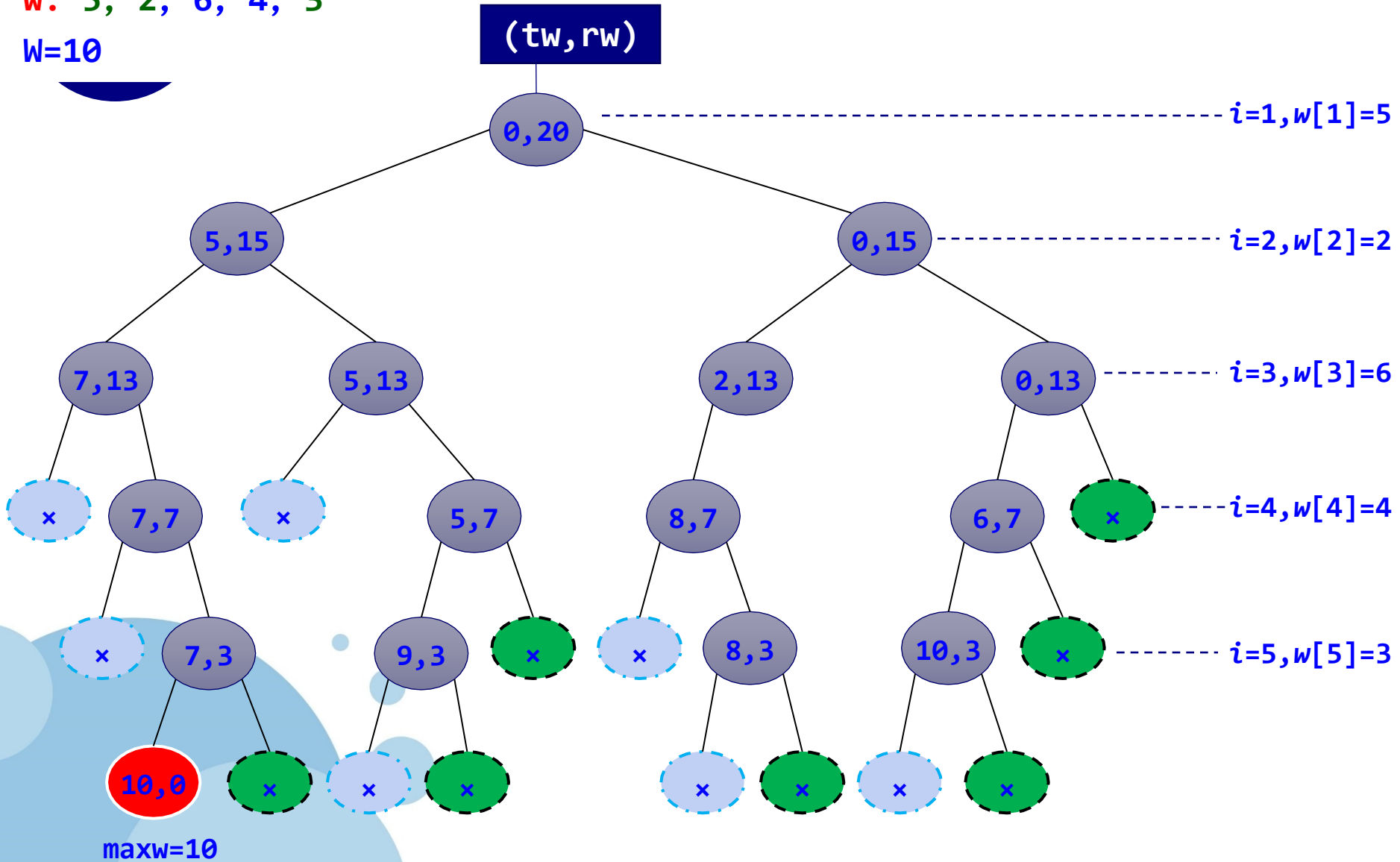
剪枝:

- **左剪枝:** 仅仅扩展 $tw + w[i] \leq W$ 的左孩子结点
- **右剪枝:** 仅仅扩展 $tw + rw - w[i] > \max w$ 的右孩子结点

$n=5$

$w: 5, 2, 6, 4, 3$

$W=10$



```

void dfs(int i,int tw,int rw,int op[]) //求解简单装载问题
{  if (i>n)                          //找到一个叶子结点
    {  if (tw>maxw)
        {  maxw=tw;                  //找到一个满足条件的更优解,保存它
            for (int j=1;j<=n;j++)   //复制最优解
                x[j]=op[j];
        }
    }
    else                              //尚未找完所有集装箱
    {  if (tw+w[i]<=W)                //左孩子结点剪枝
        {  op[i]=1;                  //选取第i个集装箱
            dfs(i+1,tw+w[i],rw-w[i],op);
        }
        if (tw+rw-w[i]>maxw)          //右孩子结点剪枝
        {  op[i]=0;                  //不选取第i个集装箱,回溯
            dfs(i+1,tw,rw-w[i],op);
        }
    }
}

```

Chapter 5

```
int w[]={0, 5, 2, 6, 4, 3};    //各集装箱重量, 不用下标0的元素  
int n=5, W=10;
```



最优方案
选取第1个集装箱
选取第2个集装箱
选取第5个集装箱
总重量=10

【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。如果有一种着色法使 G 中每条边的两个顶点着不同颜色，则称这个图是 m 可着色的。

图的 m 着色问题是对于给定图 G 和 m 种颜色，找出所有不同的着色法。

【前提】 1) 给出3个正整数 n 、 k 和 m ，表示给定的图 G 有 n 个顶点和 k 条边， m 种颜色。顶点编号为1, 2, ..., n 。

2) 给出的 k 条边信息，两个正整数 u 、 v ，表示图 G 的一条边 (u, v) 。

【结果】 将计算出的不同的着色方案数输出。如果不能着色，程序输出

Chapter 5

【输入】

5 8 4

1 2

1 3

1 4

2 3

2 4

2 5

3 4

4 5

【输出】

48

【问题求解】

- 对于图 G ，采用邻接矩阵 a 存储，根据求解问题需要，这里 a 为一个二维数组（下标0不用），当顶点 i 与顶点 j 有边时，置 $a[i][j]=1$ ，其他情况置 $a[i][j]=0$ 。
- 图中的顶点编号为 $1\sim n$ ，着色编号为 $1\sim m$ 。

对于图 G 中的每一个顶点，可能的着色为 $1\sim m$ ，所以对应的解空间是一棵 m 叉树，高度为 n ，层次 i 从1开始。

【解空间树】 排列数or子集树？

解空间树为子集树

回溯算法示例

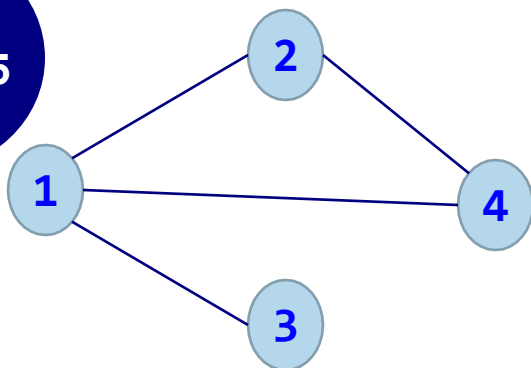
Chapter 5

求解图的 m 着色问题

```
bool Same(int i)           //判断顶点i是否与相邻顶点存在相同的着色
{   for (int j=1;j<=n;j++)
    {   if (a[i][j]==1 && x[i]==x[j])
        {   return false;
            }
        return true;
    }
}

void dfs(int i)             //求解图的m着色问题
{   if (i>n)                //达到叶子结点 (解空间树的叶子结点)
    {   count++;            //着色方案数增1
    }
    else
    {   for (int j=1;j<=m;j++) //试探每一种着色
        {   x[i]=j;          //试探着色j
            if (Same(i))      //可以着色j, 进入下一个顶点着色
            {   dfs(i+1);
            }
            x[i]=0;           //回溯, 退层。
        }
    }
}
```



Chapter 5



$n=4$, $k=4$, $m=3$, 其着色方案有12个。

第1个着色方案: 1 2 2 3
第2个着色方案: 1 2 3 2
第3个着色方案: 1 3 2 3
第4个着色方案: 1 3 3 2
第5个着色方案: 2 1 1 3
第6个着色方案: 2 1 3 1
第7个着色方案: 2 3 1 3
第8个着色方案: 2 3 3 1
第9个着色方案: 3 1 1 2
第10个着色方案: 3 1 2 1
第11个着色方案: 3 2 1 2
第12个着色方案: 3 2 2 1

【算法分析】 该算法中每个顶点试探 $1\sim m$ 种着色，共 n 个顶点，对应解空间树是一棵 m 叉树（子集树），算法的时间复杂度为 $O(m^n)$ 。



求解活动安排问题

【问题描述】 假设有一个需要使用某一资源的 n 个活动所组成的集合 S , $S=\{1, \dots, n\}$ 。该资源任何时刻只能被一个活动所占用, 活动 i 有一个开始时间 b_i 和结束时间 e_i ($b_i < e_i$), 其执行时间为 $e_i - b_i$, 假设最早活动执行时间为 θ 。

一旦某个活动开始执行, 中间不能被打断, 直到其执行完毕。若活动 i 和活动 j 有 $b_i \geq e_j$ 或 $b_j \geq e_i$, 则称这两个活动**兼容**。

设计算法求一种最优活动安排方案, 使得**所有安排的活动个数最多**。

回溯算法示例

Chapter 5

求解活动安排问题

【问题求解】

活动编号 i	1	2	3	4
开始时间 b_i	1	2	4	6
结束时间 e_i	3	5	8	10

调度方案（一种排列）： $x[1], x[2], \dots, x[n]$



第1步选择活动 $x[1]$

...

第 i 步选择活动 $x[i]$

...

第 n 步选择活动 $x[n]$

回溯算法示例

求解活动安排问题

活动编号 i	1	2	3	4
开始时间 b_i	1	2	4	6
结束时间 e_i	3	5	8	10

1234 \Rightarrow 兼容活动个数=2

2134 \Rightarrow 兼容活动个数=2

4123 \Rightarrow 兼容活动个数=1

...

求出最多的兼容活动个数

回溯算法示例

求解活动安排问题

- 采用回溯法求解，相当于找到 $S=\{1, \dots, n\}$ 的某个排列即调度方案，使得其中所有兼容活动个数最多，显然对应的解空间是一个是**排列树**。
- 直接采用**排列树递归框架**实现，对于每一种调度方案求出所有兼容活动个数，通过比较求出最多活动个数 maxsum ，对应的调度方案就是最优调度方案 bestx ，即为本问题的解。

回溯算法示例

求解活动安排问题

对于一种调度方案，如何计算所有兼容活动的个数呢？因为其中可能存在不兼容的活动。

例如，右表的4个活动，若调度方案为（1，2，3，4），求所有兼容活动个数的过程如下：

活动编号	1	2	3	4
开始时间	1	2	4	6
结束时间	3	5	8	10

- ① 置当前活动的结束时间 $laste=0$ ，所有兼容活动个数 $sum=0$ 。
- ② 活动1：其开始时间为1，大于等于 $laste$ ，属于兼容活动，选取它， sum 增加1， $sum=1$ ，置 $laste=$ 其结束时间=3。
- ③ 活动2：其开始时间为2，小于 $laste$ ，属于非兼容活动，不选取它。
- ④ 活动3：其开始时间为4，大于等于 $laste$ ，属于兼容活动，选取它， sum 增加1， $sum=2$ ，置 $laste=$ 其结束时间=8。
- ⑤ 活动4：其开始时间为6，小于 $laste$ ，属于非兼容活动，不选取它。
- ⑥ 该调度方案的所有兼容活动个数 sum 为2。

回溯算法示例

求解活动安排问题

求解过程

- 产生所有排列，每个排列 $x=(x[1],x[2],\cdots,x[n])$ 对应一种调度方案
- 计算每种调度方案的兼容活动个数 sum
- 比较求出最大的兼容活动个数 $maxsum$ 和最优方案 $bestx$

回溯算法示例

求解活动安排问题

问题表示

```
struct Action
{
    int b;           //活动起始时间
    int e;           //活动结束时间
};

int n=4;

Action A[]={0,0},{1,3},{2,5},{4,8},{6,10}}; //下标0不用
```

回溯算法示例

求解活动安排问题

问题的求解结果表示:

```
int x[MAX];           //临时解向量
int bestx[MAX];        //最优解向量
int laste=0;           //一个调度方案中最后兼容活动的结束时间,初值为0
int sum=0;              //一个调度方案中所有兼容活动个数,初值为0
int maxsum=0;
```

回溯算法示例

求解活动安排问题

$\text{dfs}(i)$ $[\text{sum}, x, \text{laste}]$

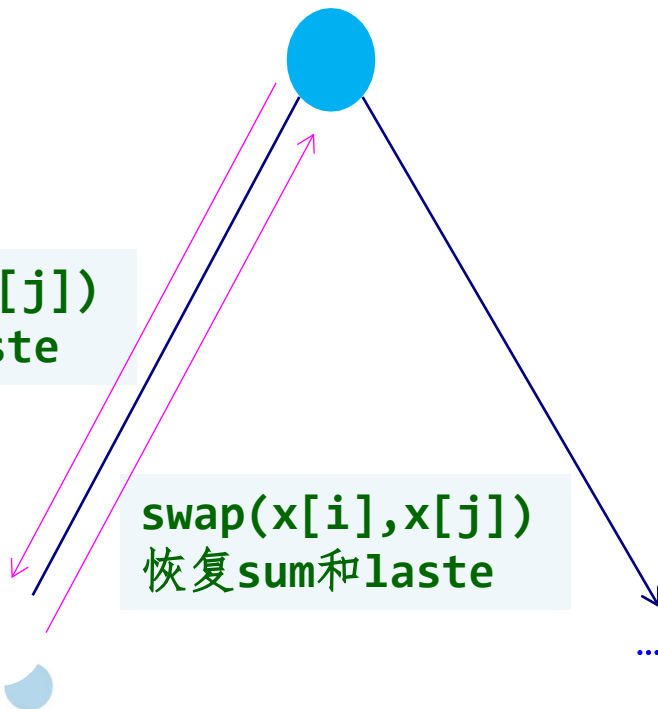
第 i 层结点

$\text{swap}(x[i], x[j])$
修改 sum 和 laste

$\text{swap}(x[i], x[j])$
恢复 sum 和 laste

$\text{dfs}(i+1)$ $[\text{sum}, x, \text{laste}]$

第 $i+1$ 层结点



回溯算法示例

求解活动安排问题

```
void dfs(int i)           //搜索活动问题最优解
{  if (i>n)               //到达叶子结点,产生一种调度方案
    {  if (sum>maxsum)
        {  maxsum=sum;
            for (int k=1;k<=n;k++)
                bestx[k]=x[k];
        }
    }
}
```

回溯算法示例

求解活动安排问题

```
else
{ for(int j=i; j<=n; j++)
{
    swap(x[i],x[j]);
x[i],x[j]
    int sum1=sum;
    int laste1=laste;
    if (A[x[j]].b>=laste)
    { sum++;
      laste=A[x[j]].e;
    }
    dfs(i+1);
    swap(x[i],x[j]);
x[i],x[j]
    sum=sum1;
    laste=laste1;
}
}
```

//没有到达叶子结点,考虑*i*到*n*的活动
//第*i*层结点选择活动x[j]
//排序树问题递归框架:交换

//保存sum, laste以便回溯

//活动x[j]与前面兼容
//兼容活动个数增1
//修改本方案的最后兼容时间

//排序树问题递归框架:进入下一层
//排序树问题递归框架:交换

//回溯
//即撤销第*i*层结点对活动x[j]的选择

回溯算法示例

求解活动安排问题

```
void dispasolution()                                //输出一个解
{
    printf("最优调度方案\n");
    int laste=0;
    for (int j=1;j<=n;j++)
    {
        if (A[bestx[j]].b>=laste)                    //选取活动bestx[j]
        {
            printf("    选取活动%d:
                    [%d,%d)\n",bestx[j],A[bestx[j]].b,A[bestx[j]].e);
            laste=A[bestx[j]].e;
        }
    }
    printf("    安排活动的个数=%d\n",maxsum);
}
```

回溯算法示例

求解活动安排问题

```
void main()
{   for (int i=1;i<=n;i++)
        x[i]=i;
    dfs(1);           //i从1开始搜索
    dispasolution();  //输出结果
}
```

回溯算法示例

求解活动安排问题

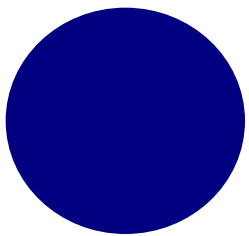
活动编号	1	2	3	4
开始时间	1	2	4	6
结束时间	3	5	8	10

最优调度方案

选取活动1: [1,3)

选取活动3: [4,8)

安排活动的个数=2



【算法分析】 该算法对应解空间树是一棵排列树，与求全排列算法的时间复杂度相同，即为 $O(n!)$ 。



5.6 本章小结

- 回溯法类似于枚举的思想，适用于查问问题的解集或符合某种限制条件的最佳解集，通常采用剪枝策略进行范围控制，这样一般其最坏时间复杂度仍然很高，但对于NP完全问题来说，回溯法被认为是目前较为有效的方法。

回溯算法的基本步骤

(1) 定义给定问题的解空间：子集树问题、排列树问题和其他因素。

(2) 确定状态空间树的结构。

(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。其中深度优先方式可以选为递归回溯或者迭代回溯。