

算法设计与分析

回溯法 分枝限界法

回溯法

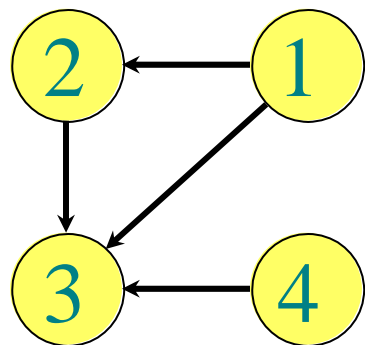
主要内容

- 图的基本知识
- 深度优先搜索（DFS）
- 广度优先搜索（BFS）
- 回溯法的基本思想
- 回溯法的求解步骤
- 回溯法的求解实例

图的表示

- 邻接表
- 邻接矩阵
- 以上方法既可以表示有向图，也可以表示无向图

图的表示——邻接表



● $\text{Adj}[1] = \{2, 3\}$

● $\text{Adj}[2] = \{3\}$

● $\text{Adj}[3] = \{\}$

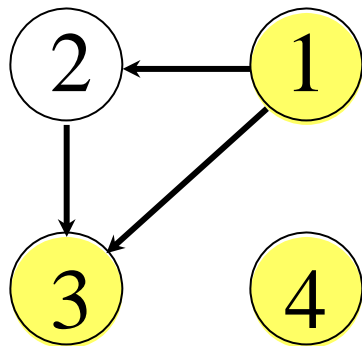
● $\text{Adj}[4] = \{3\}$

● $\text{Adj}[u]$ 包含了图G中所有和u相邻的顶点

图的表示——邻接矩阵

- 若 $V = \{1, 2, \dots, n\}$, 则邻接矩阵 $A[1 \dots n, 1 \dots n]$

$$A[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E, \\ 0 & \text{if } (i,j) \notin E. \end{cases}$$



A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

广度优先搜索（BFS）

●检索策略

- 给定图和特定的源顶点 s ，BFS希望发现可从 s 到达的所有顶点，算法首先发现与 s 距离为 k 的顶点，然后才会发现与 s 距离为 $k+1$ 的顶点

深度优先搜索（DFS）

- 搜索策略

- 尽可能“深”地搜索一个图

- 在DFS中，对于新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边一直探测下去

- 当顶点 v 的所有边都已被探寻过后，搜索将回溯到发现结点 v 的那条边的起始结点

- 这一过程一直进行到已发现从源节点可达的所有节点为止

回溯法

- 有许多问题，当需要找出它的解集或者要求在某些约束条件下的最优解时，往往可以回溯法
- 回溯法的基本做法是搜索，它是一种可以避免不必要搜索的穷举式搜索法
- 回溯法适用于求解一些组合数较大的问题

回溯法的基本思想

- 回溯法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意一点时，先判断该结点是否包含问题的解
 - 如果肯定不包含，则跳过对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，继续按深度优先策略搜索

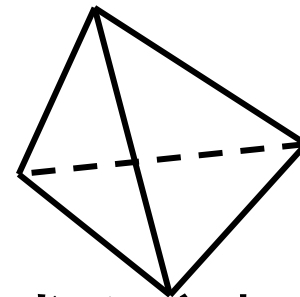
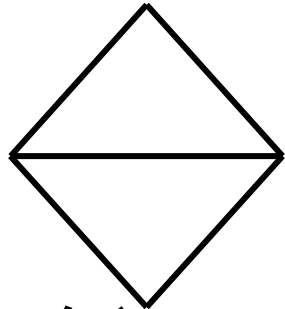
问题的解空间

- 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
- 通常将解空间组织成树或者图的形式

问题的解空间

●例：有6根火柴，以之为边搭建4个等边三角形

➤该问题易产生误导，它暗示是一个二维空间，为解决该问题需拓展到三维



➤对任意一个问题，解的表示方式和它相应的解隐含了解空间及其大小

问题的解空间

●例：对 n 个物品的0/1背包问题，其可能解的表示方式：

- 可能解由一个不等长向量组成，解向量的长度等于装入背包的物品个数，长度由 $0 \sim n$ 的解向量组成。
 - 如 $n=3$
 - 解空间 $\{(), (1), (2), (3), (1,2), (1,3), (2,3), (1,2,3)\}$
- 可能解由一个等长向量 $\{x_1, \dots, x_n\}$ 组成，则 $n=3$ ，解空间为 $\{(0,0,0), (0,0,1), (0,1,0), (1,0,0), (0,1,1), (1,0,1), (1,1,0), (1,1,1)\}$

问题的解空间

●问题的解空间一般用解空间树的方式组织

- 树的根节点位于第一层，表示搜索的初始状态
- 第二层的节点表示对解向量的第一个分量做出选择后到达的状态
- 第一层到第二层的边上标出对第一个分量选择的结果
- 依此类推，从树的根节点到叶子节点的路径就构成了解空间的一个可能解

问题的解空间

- n 个城市的TSP问题，将其可能解表示为向量 $X=(x_1, \dots, x_n)$ ，且解向量满足任意 $x_i \neq x_j$ ($1 \leq i, j \leq n$)。
- 当 $n=3$ 时，TSP问题空间为：
 - $\{(1,2,3), (1,3,2), (2,1,3), \dots\}$

生成问题状态的说明

●扩展结点

- 一个正在产生儿子的结点称为扩展结点

●活结点

- 一个自身已生成但其儿子还没有全部生成的节点称做活结点

●死结点

- 一个所有儿子已经产生的结点称做死结点

生成问题状态的基本方法——DFS

●深度优先的问题状态生成法

- 如果对一个扩展结点R，一旦产生了它的一个儿子C，就把C当做新的扩展结点。在完成对子树C（以C为根的子树）的穷尽搜索之后，将R重新变成扩展结点，继续生成R的下一个儿子（如果存在）

生成问题状态的基本方法——回溯法

- 回溯法：为了避免生成那些不可能产生最优解的问题状态，要不断地利用限界函数(bounding function)来“处死”那些实际上不可能产生所需解的活结点，从而减少问题的计算量
- 具有限界函数的深度优先生成法称为回溯法

回溯法的基本思想

- (1)针对所给问题，定义问题的解空间
 - 复杂问题常有很多可能解，这些解构成解空间
 - 确定正确的解空间很重要
- (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
 - 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

回溯法的基本思想

- 在搜索至树上任意一点时，先判断该节点对应部分解是否满足约束条件，或是否超出目标函数的界
 - 判断该节点是否包含问题的（最优）解。
 - 不包含，则跳过对以该节点为根的子树的搜索，剪枝(pruning)
 - 包含，则进入以该节点为根的子树，继续按深度优先搜索

回溯法的基本思想

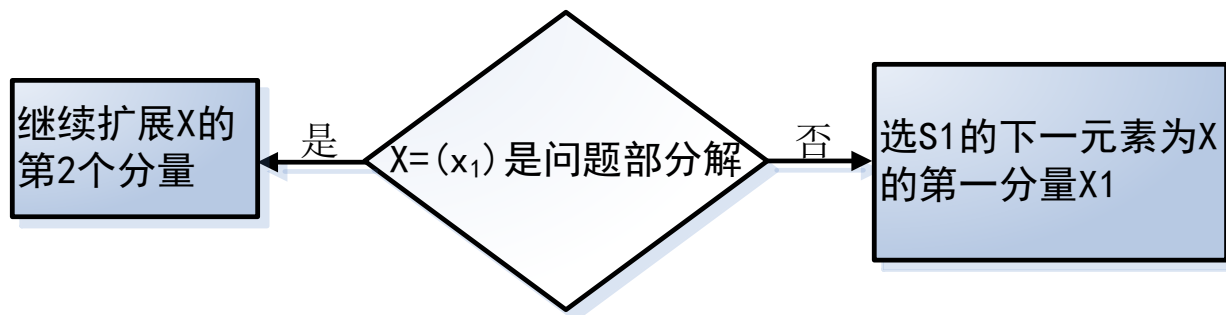
- 例： $n=3$ 的0-1背包问题，3个物品重为{20,15,10}，价值为{20,30,25}，背包容量为25

回溯法的基本思想

- 回溯法的搜索过程涉及的节点（搜索空间）只是整个解空间树的一部分，搜索时，常用两种策略避免无效搜索（剪枝函数）
 - 用约束条件剪去得不到可行解的子树
 - 用目标函数剪去得不到最优解的子树

回溯法的求解过程

- 问题的解向量 $X=(x_1, x_2, \dots, x_n)$ 中每个分量 $x_i \in S_i = \{a_{i1}, a_{i2}, \dots, a_{iri}\}$ ，回溯可按某种顺序依次考察 $S_1 \times S_2 \times \dots \times S_n$ 中元素
 - 初始时 X 为空，由根出发，选 S_1 的第一个元素作为 X 的第一个分量，即 $x_1 = a_{11}$ ，



- 依此类推

回溯法的求解过程

- 如果 $X=(x_1, x_2, \dots, x_i)$ 是问题的部分解，则选 S_{i+1} 的第一个元素 X 向量的 $i+1$ 个分量：
 - 如果 $X=(x_1, x_2, \dots, x_{i+1})$ 是问题最终解，输出解。如果只希望一个解，结束；否则继续搜索
 - 如果 $X=(x_1, x_2, \dots, x_{i+1})$ 是问题部分解，继续构造解向量的下一个分量
 - 如果 $X=(x_1, x_2, \dots, x_{i+1})$ 既不是问题的部分解也不是问题的最终解，则：
 - 如果 $x_{i+1} = a_{i+1\ k}$ 不是集合 S_{i+1} 的最后一个元素，则令 $x_{i+1} = a_{i+1\ k+1}$ ；
 - 如果是，就回溯到 $X=(x_1, \dots, x_i)$ ；

回溯法的分析

- $X=(x_1, \dots, x_n)$, x_i 取值为由有限集 S_i , 因为解空间为 $S_1 \times S_2 \times \dots \times S_n$, 所以
 - 第2层有 $|S_1|$ 个节点;
 - 第3层有 $|S_1| \times |S_2|$ 个节点;
 - ...
 - 第 $n+1$ 层有 $|S_1| \times \dots \times |S_n|$ 个节点;
 - 叶子为所有可能解

回溯法的分析

- 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

递归回溯（用递归方法实现回溯法）

- 回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法

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

迭代回溯

- 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

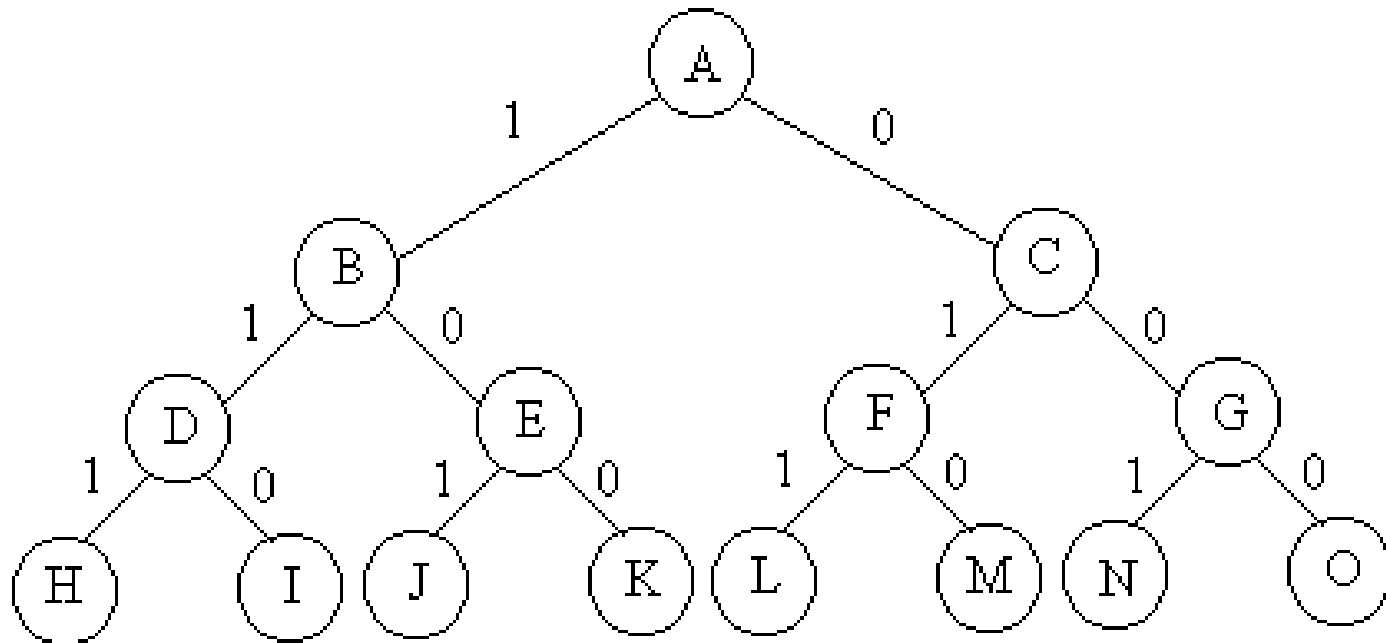
```
void iterativeBacktrack ()
{
    int t=1;
    while (t>0) {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++) {
                x[t]=h(i);
                if (constraint(t)&&bound(t)) {
                    if (solution(t)) output(x);
                    else t++;}
        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

子集树和排列树

- 回溯法求解时常见的两类解空间树
- 子集树：当所给问题是从 n 个元素的集合 S 中找出 S 满足某种性质的子集时，相应的解空间树称为子集树
- 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

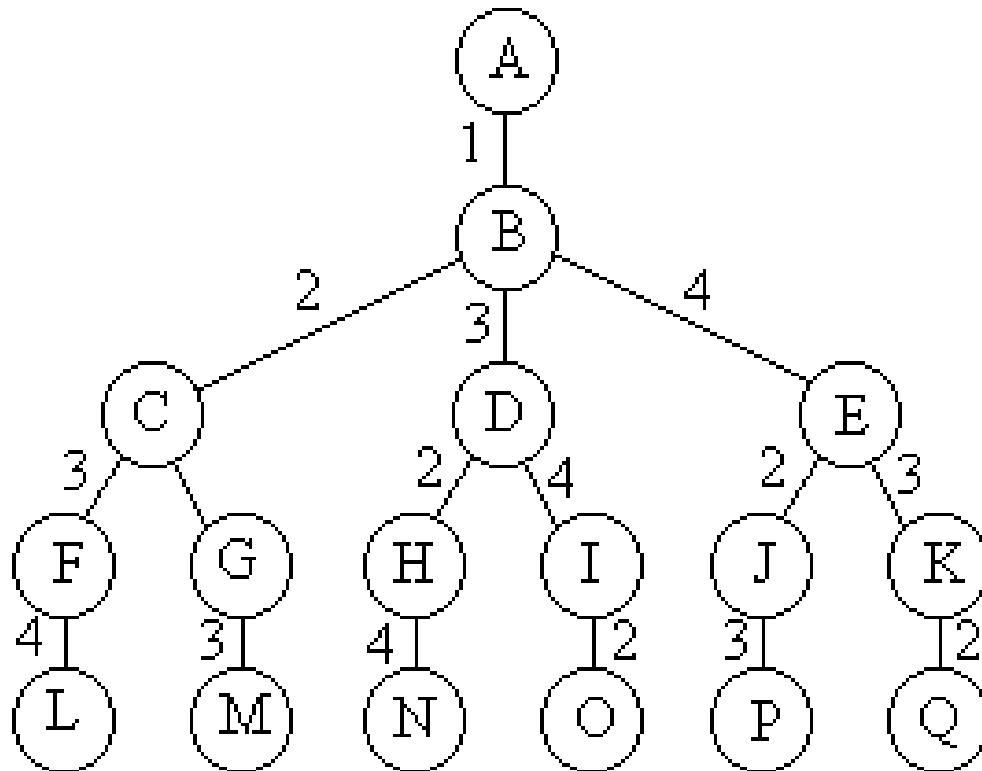
子集树和排列树

- 子集树，通常 $|S_1| = \dots = |S_n| = C$ ，各节点有相同数目子树， $C=2$ 时，子集树中共有 2^n 个叶子，因此需要 $O(2^n)$ 时间



子集树和排列树

- 排列树，通常 $|S_1|=n$ ， \dots ， $|S_n|=1$ ，所以排列树中共有 $n!$ 个叶子节点，需时间 $O(n!)$



装载问题

- 有一批共 n 个集装箱要装上2艘载重量分别为 c_1 和 c_2 的轮船，其中集装箱 i 的重量为 w_i ，且

$$\sum_{i=1}^n w_i \leq c_1 + c_2$$

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

装载问题

- 容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。
 - 首先将第一艘轮船尽可能装满
 - 将剩余的集装箱装上第二艘轮船
 - 将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于特殊的0-1背包问题
 - 用回溯法设计解装载问题的 $O(2^n)$ 计算时间算法。在某些情况下该算法优于动态规划算法

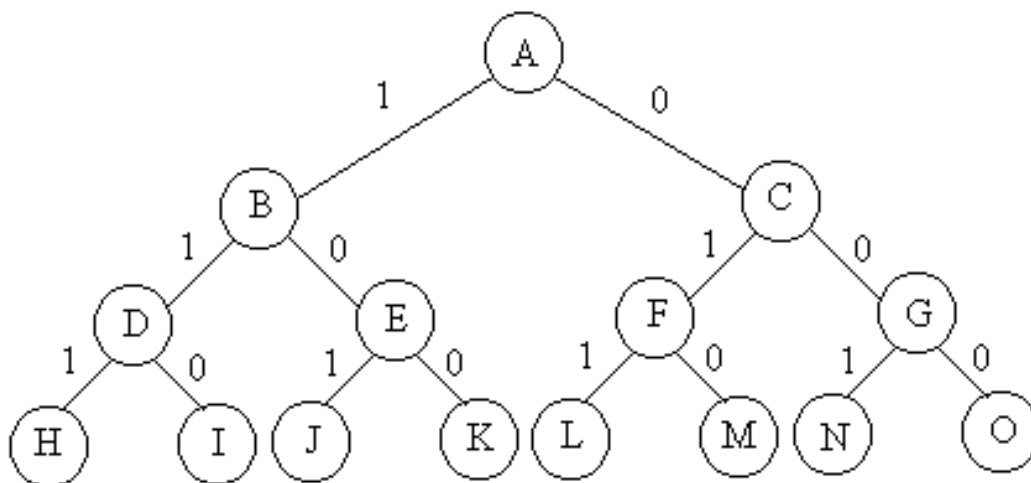
装载问题

- 解空间：子集树

- 可行性约束条件(选择当前元素) $\sum_{i=1}^n w_i x_i \leq c_1$

- 上界函数(不选择当前元素)

➤当前载重量cw+剩余集装箱的重量r<=当前最优载重量bestw



装载问题

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

批处理作业调度

- 给定 n 个作业的集合 $\{J_1, J_2, \dots, J_n\}$ 。每个作业必须先由机器1处理，然后由机器2处理。
 - 作业 J_i 需要机器 j 的处理时间为 t_{ji} 。
 - 对于一个确定的作业调度，设 F_{ji} 是作业 i 在机器 j 上完成处理的时间。所有作业在机器2上完成处理的时间和称为该作业调度的完成时间和。
- 批处理作业调度问题要求对于给定的 n 个作业，制定最佳作业调度方案，使其完成时间和达到最小

批处理作业调度

t_{ji}	机器1	机器2
作业1	2	1
作业2	3	1
作业3	2	3

这3个作业的6种可能的调度方案是1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1; 它们所相应的完成时间和分别是19, 18, 20, 21, 19, 19。易见, 最佳调度方案是1,3,2, 其完成时间和为18。

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
- 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)
```

```
{  
    if ((count>half)||((t*(t-1)/2-count>half)) return;  
    if (t>n) sum++;
```

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

```
    else
```

```
        for (int i
```

```
            p[1][
```

```
            count
```

```
            for (i
```

```
                p[j]
```

```
                cou
```

```
            }  
        }  
        Backtr
```

```
        for (int j=2;j<=t;j++)
```

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

```
        count-=i;
```

```
    }  
}
```

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，故解符号三角形问题的回溯算法所需的计算时间为 $O(n2^n)$ 。

n后问题

●问题描述

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

n后问题

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

n后问题

- 解向量： (x_1, x_2, \dots, x_n)
- 显约束： $x_i = 1, 2, \dots, n$
- 隐约束：
 - 不同列： $x_i \neq x_j$
 - 不处于同一正、反对角线： $|i-j| \neq |x_i - x_j|$
- 用回溯法求解时，用完全n叉树表示解空间，用可行性约束剪去不满足行、列、斜线约束的子树

n后问题

```
private static boolean place (int k)
{
    for (int j=1;j<k;j++)
        if ((Math.abs(k-j)==Math.abs(x[j]-x[k]))||(x[j]==x[k])) return false;
    return true;
}

private static void backtrack (int t)
{
    if (t>n) sum++;
    else
        for (int i=1;i<=n;i++) {
            x[t]=i;
            if (place(t)) backtrack(t+1);
        }
```

0-1背包问题

●问题描述

- 给定 n 种物品和一个背包。物品 i 的重量是 W_i ，其价值为 V_i ，背包的容量为 C 。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？
- 限制：在选择装入背包的物品时，对每种物品 i 只有2种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i

0-1背包问题

- 解空间：子集树

- 约束：
$$\sum_{i=1}^n w_i x_i \leq c$$

图的 m 着色问题

- 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 为该图的色数
 - 求一个图的色数 m 的问题称为图的 m 可着色优化问题

图的 m 着色问题

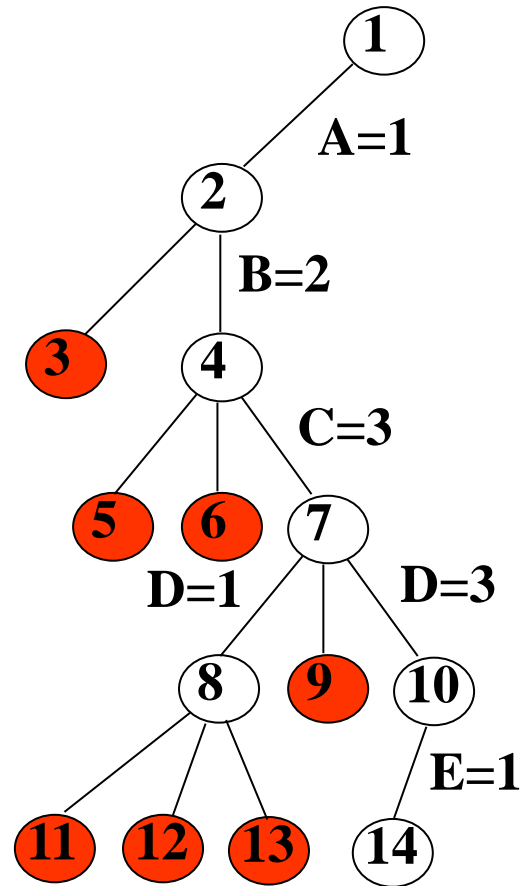
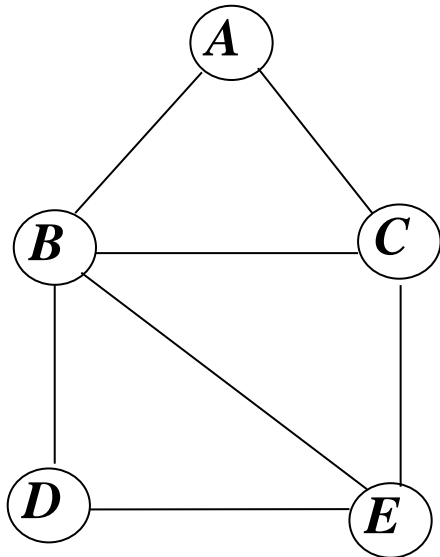
- 用 m 种颜色着色 n 个顶点 $\rightarrow m^n$ 种可能的着色组合
- 所以解空间树是一棵完全 m 叉树，每个节点都有 m 棵子树
- 在图着色问题的解空间树中，从根 \rightarrow 当前节点对应一个部分解，即所有颜色指派无冲突，则以当前节点选择第一棵子树继续搜索（下一顶点着色）
- 否则，对当前子树的兄弟子树继续搜。（当前顶点着色）

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与已着色的相邻顶点颜色不重复
- 问题的解空间： 高度为 $n+1$ 的完全 m 叉树

图的m着色问题

●求解示例



图的m着色问题 算法思路

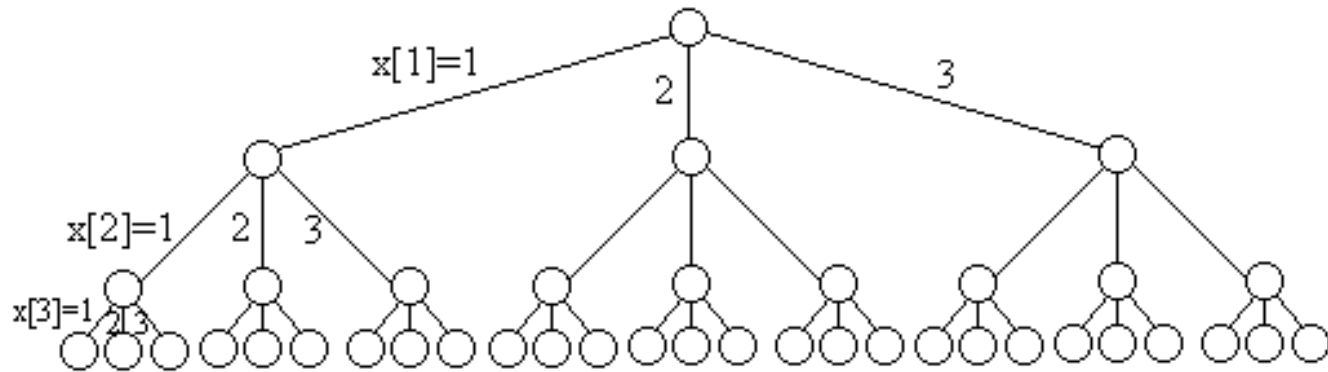
算法——图着色问题

1. 将数组color[n]初始化为0;
2. $k=1$;
3. while ($k \geq 1$)
 - 3.1 依次考察每一种颜色, 若顶点k的着色与其他顶点的着色不发生冲突, 则转步骤3.2; 否则, 搜索下一个颜色;
 - 3.2 若顶点已全部着色, 则输出数组color[n], 返回;
 - 3.3 否则,
 - 3.3.1 若顶点k是一个合法着色, 则 $k=k+1$, 转步骤3.1处理下一个顶点;
 - 3.3.2 否则, 重置顶点k的着色情况, $k=k-1$, 转步骤3回溯;

图的m着色问题

```
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;
}
```

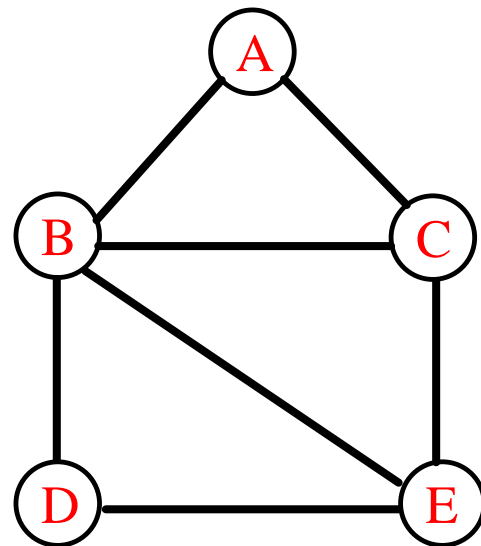
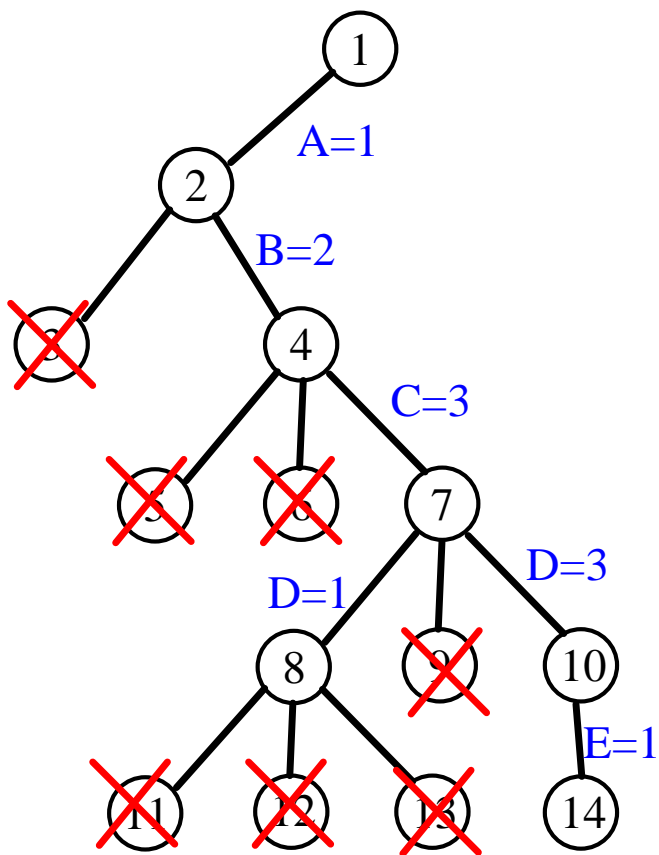


图的m着色问题 复杂度分析

- 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函数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

$$\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1) / (m - 1) = O(nm^n)$$

图的m着色问题



图的m着色问题 应用举例

- 三着色问题：机场停机位分配是指根据航班和机型等属性，为每个航班指定停机位。要满足下列约束：
 - 1) 各航班须被分配，且仅能被分配一个停机位；
 - 2) 同一时刻同一停机位不可分配1个以上的航班；
 - 3) 应满足航站衔接以及过站时间衔接要求；
 - 4) 机位与使用机位航班应相互匹配。
- 分配时，班机时刻表已知，按“先到先服务”的原则进行分配，可将此分配→图着色问题。

旅行售货员问题

- 某售货员要到若干城市去推销商品，已知各城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程（或总旅费）最短（或最小）。

旅行售货员问题解题思路

设 $G=(V, E)$ 是一个带权图。图中各边的费用（权）为正数。图的一条周游路线是包括 V 中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。旅行售货员问题是要在图 G 中找出费用最小的周游路线。

旅行售货员问题

```
template<class Type>
void Traveling<Type>::Backtrack(int i)
{
    if (i == n) {
        if (a[x[n-1]][x[n]] != NoEdge && a[x[n]][1] != NoEdge &&
            (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc == NoEdge)) {
```

复杂度分析

算法backtrack在最坏情况下可能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 $O(n!)$ 。

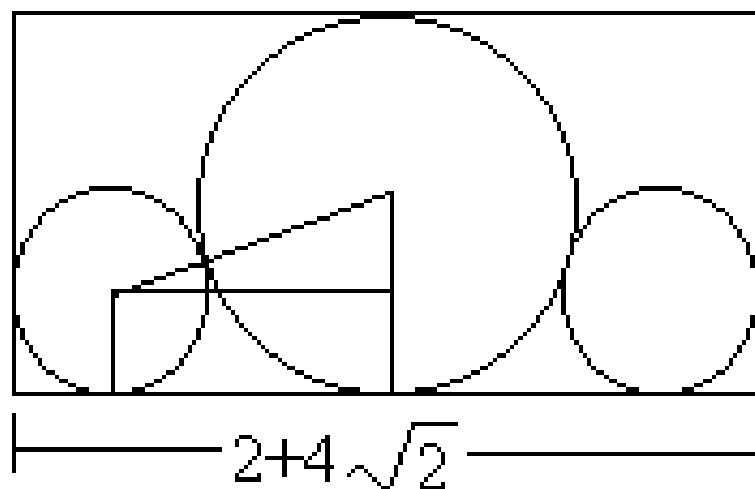
```
        Backtrack(i+1);
        cc -= a[x[i-1]][x[i]];
        Swap(x[i], x[j]);
    }
}
```

圆排列问题

- 给定 n 个大小不等的圆 c_1, c_2, \dots, c_n ，现要将这 n 个圆排进一个矩形框中，且要求各圆与矩形框的底边相切。
- 圆排列问题要求从 n 个圆的所有排列中找出有最小长度的圆排列。

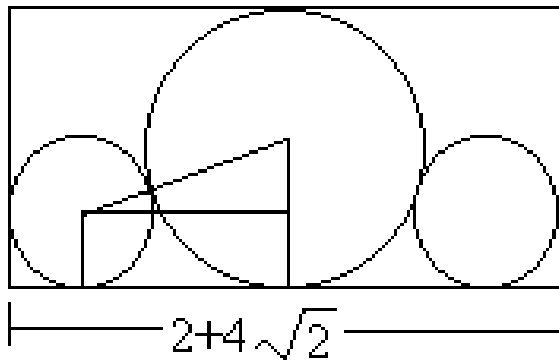
圆排列问题 示例

- 当 $n=3$ ，且所给的3个圆的半径分别为1, 1, 2时，这3个圆的最小长度的圆排列如图所示
- 其最小长度为 $2+4\sqrt{2}$



圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
        for (int j = t; j <= n; j++) {
            Swap(r[t], r[j]);
            float centerx=Center(t);
            if (centerx+r[t]+r[1]<min) { //下界约束
                x[t]=centerx;
                Backtrack(t+1);
            }
            Swap(r[t], r[j]);
        }
}
```



```
float Circle::Center(int t)
{ // 计算当前所选择圆的圆心横坐标
    float temp=0;
    for (int j=1;j<t;j++) {
        float valuex=x[j]+2.0*sqrt(r[t]*r[j]);
        if (valuex>temp) temp=valuex;
    }
    return temp;
}
```

```
void Circle::Compute(void)
{ // 计算当前圆排列的长度
    float low=0,
          high=0;
    for (int i=1;i<=n;i++) {
        if (x[i]-r[i]<low) low=x[i]-r[i];
        if (x[i]+r[i]>high) high=x[i]+r[i];
    }
    if (high-low<min) min=high-low;
}
```

圆排列问题

- 复杂度分析

- 由于算法**backtrack**在最坏情况下可能需要计算 $O(n!)$ 次当前圆排列长度，每次计算需 $O(n)$ 计算时间，从而整个算法的计算时间复杂性为 $O((n+1)!)$

圆排列问题

- 上述算法尚有许多改进的余地
- 例如，象 $1, 2, \dots, n-1, n$ 和 $n, n-1, \dots, 2, 1$ 这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。另一方面，如果所给的 n 个圆中有 k 个圆有相同的半径，则这 k 个圆产生的 $k!$ 个完全相同的圆排列，只计算一个就够了。

素数环

- 给定一个正整数 n ，寻找一个由 $1, 2, 3, \dots, n$ 组成的环状排列，使得相邻两个整数之和都是素数。
- 输出时从整数1开始逆时针排列
- $n \leq 16$

素数环

- 解空间是所有可能的排列。
- 解空间树的形态是一棵深度为 n 的树，每个节点有 n 个子节点，表示对应位置上的下一个可能的数字。
- 根节点表示问题的初始状态，即还没有选择任何数字的状态。
- 搜索过程中，从根节点开始，依次选择下一个可能的数字，构建解空间树。
 - 在选择每个数字时，需要检查它与前一个数字的和是否是素数。如果是素数，继续深入搜索；如果不是素数，回溯到前一个节点，选择其他可能的数字。
- 搜索的终止条件是找到一个满足条件的环状排列，或者遍历完整个解空间树仍然没有找到解。

分枝限界法

主要内容

- 分枝限界法的基本思想
- 分枝限界法的求解实例

分枝限界法和回溯法

- 分枝限界法类似于回溯法，是在问题的解空间树上搜索问题解的算法
- 相同点
 - 一般需要定义问题的解空间，同时解空间的组织结构通常为树或图
 - 一般都基于问题的解空间树搜索问题的解
 - 根据判断条件进行剪枝操作

分枝限界法和回溯法

●不同点

- 求解目标：回溯法的求解目标一般是找出解空间树中满足约束条件的所有解，而分枝限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解
- 搜索方式的不同：回溯法以深度优先的方式搜索解空间树，而分枝限界法以广度优先或以最小耗费优先的方式搜索解空间树

分枝限界法的基本思想

●分枝限界法常以**广度优先**或以**最小耗费（最大效益）**优先的方式搜索问题的解空间树

- 在分枝限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，导致不可行解或导致非最优解的儿子结点被舍弃，其余儿子结点被加入活结点表中
- 此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止

常见的两种分枝限界法

●队列式(FIFO)分枝限界法

- 按照队列先进先出（FIFO）原则选取下一个节点为扩展节点

●优先队列式分枝限界法

- 按照优先队列中规定的优先级选取优先级最高的节点成为当前扩展节点
- 应用优先队列式分枝限界法求解具体问题时，应该根据具体问题的特点确定选用最大优先队列或者最小优先队列表示解空间的活结点表

装载问题

- 有一批共 n 个集装箱要装上2艘载重量分别为 c_1 和 c_2 的轮船，其中集装箱 i 的重量为 w_i ，且

$$\sum_{i=1}^n w_i \leq c_1 + c_2$$

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

装载问题

- 容易证明，如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。
 - 首先将第一艘轮船尽可能装满
 - 将剩余的集装箱装上第二艘轮船

装载问题

●队列式分枝限界法

- 在算法的while循环中，首先检测当前扩展结点的左儿子结点是否为可行结点。如果是则将其加入到活结点队列中。
- 然后将其右儿子结点加入到活结点队列中(右儿子结点一定是可行结点)。
- 2个儿子结点都产生后，当前扩展结点被舍弃

装载问题

●队列式分枝限界法

- 活结点队列中的队首元素被取出作为当前扩展结点，由于队列中每一层结点之后都有一个尾部标记-1，故在取队首元素时，活结点队列一定不空
- 当取出的元素是-1时，再判断当前队列是否为空。如果队列非空，则将尾部标记-1加入活结点队列，算法开始处理下一层的活结点

装载问题

● 队列式分枝限界法

```
while (true) {  
    // 检查左儿子结点  
    if (Ew + w[i] <= c)                // x[i] = 1  
        EnQueue(Q, Ew + w[i], bestw, i, n);  
    // 右儿子结点总是可行的  
    EnQueue(Q, Ew, bestw, i, n);      // x[i] = 0  
    Q.Delete(Ew);                    // 取下一扩展结点  
    if (Ew == -1) {                  // 同层结点尾部  
        if (Q.IsEmpty())  
            return bestw;  
        Q.Add(-1);                   // 同层结点尾部标志  
        Q.Delete(Ew);                // 取下一扩展结点  
        i++;                          // 进入下一层  
    }  
}
```

装载问题

●算法的改进

- 结点的左子树表示将此集装箱装上船，右子树表示不将此集装箱装上船。
- 设 $bestw$ 是当前最优解； ew 是当前扩展结点所相应的重量； r 是剩余集装箱的重量。则当 $ew+r \leq bestw$ 时，可将其右子树剪去，因为此时若要船装最多集装箱，就应该把此箱装上船
- 另外，为了确保右子树成功剪枝，应该在算法每一次进入左子树的时候更新 $bestw$ 的值

装载问题

● 算法的改进

提前更新bestw

// 检查左儿子结点

Type wt = Ew + w[i]; // 左儿子结点的重量

if (wt <= c) { // 可行结点

if (wt > bestw) bestw = wt;

// 加入活结点队列

if (i < n) Q.Add(wt);

}

右儿子剪枝

// 检查右儿子结点

if (Ew + r > bestw && i < n)

Q.Add(Ew); // 可能含最优解

Q.Delete(Ew); // 取下一扩展结点

装载问题

●构造最优解

- 为了在算法结束后能方便地构造出与最优值相应的最优解，算法必须存储相应子集树中从活结点到根结点的路径。为此目的，可在每个结点处设置指向其父结点的指针，并设置左、右儿子标志

```
class QNode
{
    QNode *parent;           // 指向父结点的指针
    bool LChild;             // 左儿子标志
    Type weight;             // 结点所相应的载重量
}
```

装载问题

●构造最优解

- 找到最优值后，可以根据parent回溯到根结点，找到最优解

// 构造当前最优解

```
for (int  $j = n - 1$ ;  $j > 0$ ;  $j--$ ) {  
    bestx[j] = bestE->LChild;  
    bestE = bestE->parent;  
}
```


装载问题

● 优先队列式分枝限界法

- 解装载问题的优先队列式分枝限界法用最大优先队列存储活结点表。活结点 x 在优先队列中的优先级定义为从根结点到结点 x 的路径所相应的载重量再加上剩余集装箱的重量之和
- 优先队列中优先级最大的活结点成为下一个扩展结点。以结点 x 为根的子树中所有结点相应的路径的载重量不超过它的优先级。子集树中叶结点所相应的载重量与其优先级相同
- 在优先队列式分枝限界法中，一旦有一个叶结点成为当前扩展结点，则可以断言该叶结点所相应的解即为最优解。此时可终止算法

0-1背包问题

●问题描述

- 给定 n 种物品和一个背包。物品 i 的重量是 W_i ，其价值为 V_i ，背包的容量为 C 。应如何选择装入背包的物品，使得装入背包中物品的总价值最大？
- 限制：在选择装入背包的物品时，对每种物品 i 只有2种选择，即装入背包或不装入背包。不能将物品 i 装入背包多次，也不能只装入部分的物品 i

0-1背包问题

●队列式分枝定界法

- 用队列存储活动结点表
- 解空间为子集树

0-1背包问题

● 优先队列式分枝定界法的一种方案

- 首先，对输入数据进行预处理，将各物品依其单位重量价值从大到小进行排列；节点的优先级为已装袋的物品价值加上剩下的最大单位重量价值的物品装满剩余容量的价值和
- 算法首先检查当前扩展结点的左儿子结点的可行性。如果该左儿子结点是可行结点，则将它加入到子集树和活结点优先队列中。当前扩展结点的右儿子结点一定是可行结点，仅当右儿子结点满足上界约束时才将它加入子集树和活结点优先队列。当扩展到叶节点时为问题的最优值

0-1背包问题

● 上界函数

// n 表示物品总数, $cleft$ 为剩余空间

```
while ( $i \leq n$  &&  $w[i] \leq cleft$ )
```

```
{
```

```
     $cleft -= w[i];$ 
```

// $w[i]$ 表示 i 所占空间

```
     $b += p[i];$ 
```

// $p[i]$ 表示 i 的价值

```
     $i++;$ 
```

```
}
```

```
if ( $i \leq n$ )  $b += p[i]/w[i] * cleft;$  // 装填剩余容量装满背包
```

```
return  $b;$ 
```

// b 为上界函数

0-1背包问题

```
while (i != n+1) { // 非叶结点
    // 检查当前扩展结点的左儿子结点
    Typew wt = cw + w[i];
    if (wt <= c) { // 左儿子结点为可行结点
        if (cp+p[i] > bestp) bestp = cp+p[i];
        AddLiveNode(up, cp+p[i], cw+w[i], true, i+1);}
    up = Bound(i+1);
    // 检查当前扩展结点的右儿子结点
    if (up >= bestp) // 右子树可能含最优解
        AddLiveNode(up, cp, cw, false, i+1);

    // 取下一个扩展结点（略） }
```

分支限界搜索过程

旅行售货员问题

- 某售货员要到若干城市去推销商品，已知各城市之间的路程（或旅费）。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程（或总旅费）最短（或最小）。

旅行售货员问题

- 解空间树：排列树
- 队列式分枝定界法
- 优先队列式分枝定界法