




## 第5章 回溯法

- 
- 学习要点
  - 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
  - （1）递归回溯
  - （2）迭代回溯
  - （3）子集树算法框架
  - （4）排列树算法框架

- 通过应用范例学习回溯法的设计策略。
- (1) 装载问题;
- (2) 批处理作业调度;
- (3) 符号三角形问题
- (4)  $n$ 后问题;
- (5) 0-1背包问题;
- (6) 最大团问题;
- (7) 图的 $m$ 着色问题
- (8) 旅行售货员问题
- (9) 圆排列问题
- (10) 电路板排列问题
- (11) 连续邮资问题

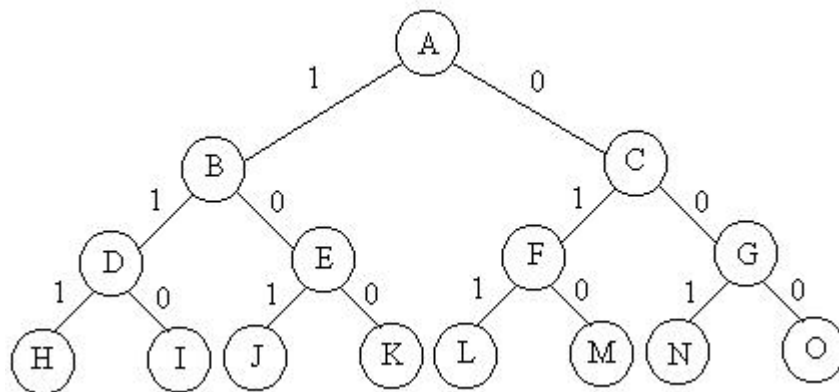
# 回溯法

- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。
- 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。

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

# 生成问题状态的基本方法

- 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**
- **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)
- 宽度优先的问题状态生成法: 在一个扩展结点变成死结点之前, 它一直是扩展结点
- 回溯法: 为了避免生成那些不可能产生最佳解的问题状态, 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, 以减少问题的计算量。 **具有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；
- (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。

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

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); //算法已搜索至叶结点
    else
    {
        for (int i=f(n,t); i<=g(n,t); i++)
        {
            x[t]=h(i);
            if (constraint(t)&&bound(t))
                backtrack(t+1);
        }
    }
}
```

t为递归深度，即当前扩展节点在解空间树中的深度；n用来控制递归深度；

f(n, t)、g(n, t)分别表示在当前扩展节点处未搜索过的子树的起始编号和终止编号；

h(i)表示在当前扩展节点处x[t]的第i个可选值；

constraint(t)和bound(t)分别是当前扩展节点处的约束函数和限界函数。

constraint(t)返回true表示在当前扩展节点处x[1:t]取值满足问题的约束条件，为false表示不满足约束条件，可剪去相应的子树。Bound返回true说明在x[1:t]取值未使目标函数越界，还可以继续由backtrack(t+1)继续对其子树进一步搜索。Bound返回true，说明已越界，可以剪去其子树



# 迭代回溯

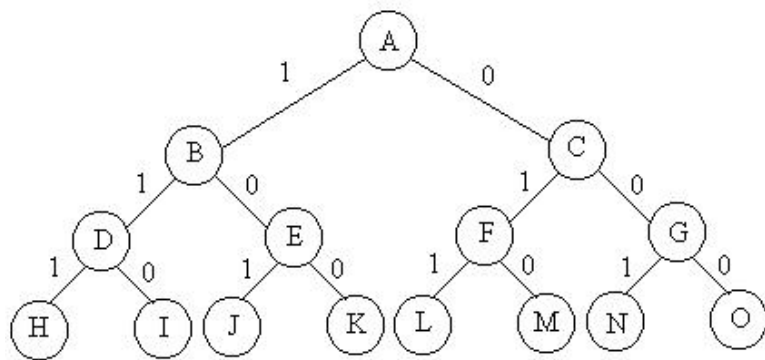
采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
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--;
    }
}
```

# 子集树与排列树

0-1背包问题：给定 $n$ 种物品和一背包。物品 $i$ 的重量是 $w_i$ ，其价值为 $v_i$ ，背包的容量为 $C$ 。问：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？

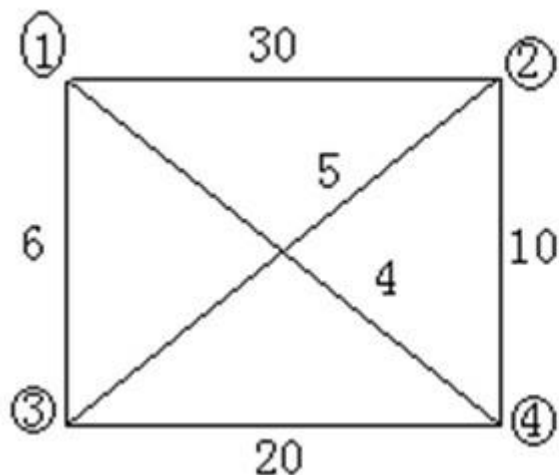
当 $n=3$ 时，0-1背包问题可用完全二叉树表示其解空间



遍历子集树需 $O(2^n)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=0;i<=1;i++) {
            x[t]=i;
            if (legal(t)) backtrack(t+1);
        }
}
```

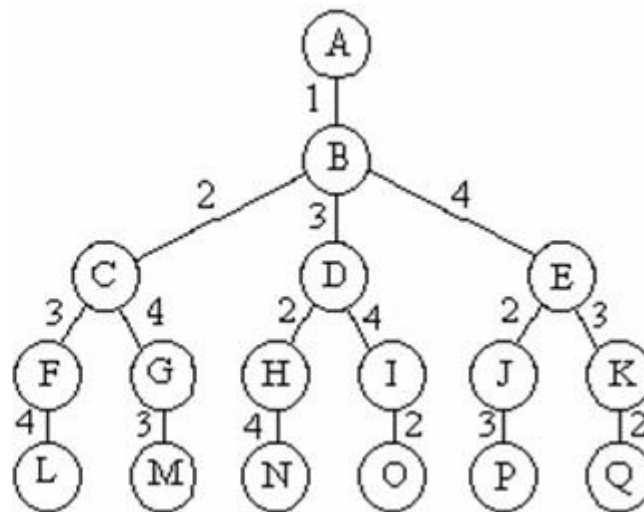
# 子集树与排列树



旅行售货员问题

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（旅费），他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程（总旅费）最小。



遍历排列树需要 $O(n!)$ 计算时间

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
        for (int i=t;i<=n;i++) {
            swap(x[t], x[i]);
            if (legal(t)) backtrack(t+1);
            swap(x[t], x[i]);
        }
}
```

# 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。

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。

(1)首先将第一艘轮船尽可能装满；

(2)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，装载问题等价于以下特殊的0-1背包问题。

$$\begin{aligned} \max \quad & \sum_{i=1}^n w_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^n w_i x_i \leq c_1 \\ & x_i \in \{0,1\}, 1 \leq i \leq n \end{aligned}$$

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
- 可行性约束函数(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

```
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];    }
```

//当前载重量+r(未考察对象的总重量)>bestw(当前的最优载重量)时当前子树不可能包含最优解，直接减掉

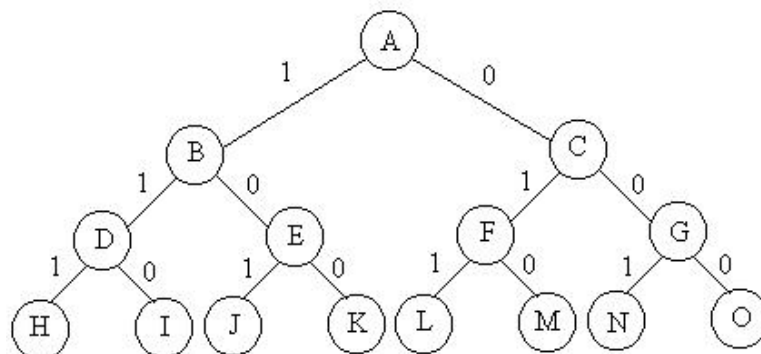
```
    if (cw + r > bestw) {
```

```
        x[i] = 0; // 搜索右子树
```

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

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

```
}
```



# 装载问题

```
1 template <class Type>
2 class Loading
3 {
4     friend Type MaxLoading(Type [],Type,int);
5     private:
6         void Backtrack(int i);
7         int n;
8         Type * w,
9             c,
10            cw,
11            bestw,
12            r;//剩余集装箱重量
13 };
```

# 装载问题

```
14 template <class Type>
15 void Loading<Type>::Backtrack(int i)
16 {
17     if(i>n)
18     {
19         if(cw>bestw)
20             bestw = cw;
21         return;
22     }
23     r-=w[i]; //计算剩余(未考察)的集装箱的重量, 减去当前考察过的对象的重量
24     if(cw+w[i] <= c)
25     {
26         cw += w[i];
27         Backtrack(i+1);
28         cw -= w[i];
29     }
30     Backtrack(i+1);
31     r+=w[i]; //递归回退返回上一层时, 记得修改r的当前值, 如果得不到最优解,
    再取消当前考察的集装箱, 标记为未选, 因此剩余容量要再加上当前集装箱重量
32 }
```

# 装载问题

```
33 template <class Type>
34 Type MaxLoading(Type w[],Type c,int n)
35 {
36     Loading<Type> X; //初始化
37     X.w = w;
38     X.c = c;
39     X.n = n;
40     X.bestw = 0;
41     X.cw = 0;
42     X.r = 0;      //初始化r
43     for(int i=1;i<=n;i++)    //计算总共的剩余（当前为考察过的）集
    装箱重量
44         X.r += w[i];
45     X.Backtrack(1);
46     return X.bestw;
47 }
```



# 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。

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

# 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

以1,2,3为例:

作业1在机器1上完成的时间为2,在机器2上完成的时间为3

作业2在机器1上完成的时间为5,在机器2上完成的时间为6

作业3在机器1上完成的时间为7,在机器2上完成的时间为10

$3+6+10=19$ , 所以是19

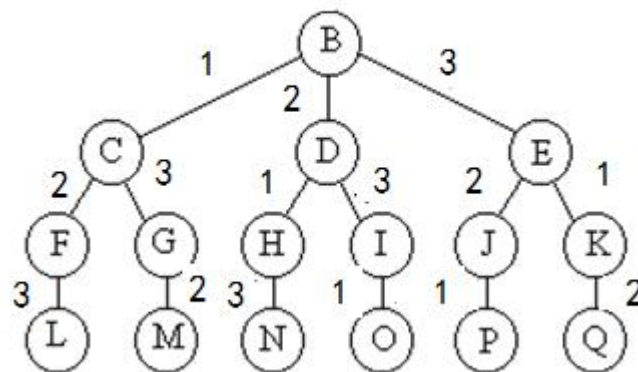
这3个作业的6种可能的调度方案是1,2,3; 1,3,2; 2,1,3; 2,3,1; 3,1,2; 3,2,1; 它们所相应的完成时间和分别是19, 18, 20, 21, 19, 19。易见, 最佳调度方案是1,3,2, 其完成时间和为18。

# 批处理作业调度

- 解空间：排列树

```
void Flowshop::Backtrack(int i)
```

```
{
    if (i > n) {
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestf = f;
    }
    else //i-1层
        for (int j = i; j <= n; j++) {
            f1 += M[x[j]][1]; //作业j在M1上的加工时间
            f2[i] = ((f2[i-1] > f1) ? f2[i-1] : f1) + M[x[j]][2];
            f += f2[i]; //作业1...j完成的加工时间之和
            if (f < bestf) { //在i层找到更优的作业调度
                Swap(x[i], x[j]);
                Backtrack(i+1);
                Swap(x[i], x[j]);
            }
            f1 -= M[x[j]][1];
            f -= f2[i];
        }
}
```



```
class Flowshop {
    friend Flow(int**, int, int []);
private:
    void Backtrack(int i);
    int **M, // 各作业所需的处理时间
        *x, // 当前作业调度
        *bestx, // 当前最优作业调度
        *f2, // 机器2完成处理时间
        f1, // 机器1完成处理时间
        f, // 完成时间和
        bestf, // 当前最优值
        n; // 作业数;
}
```

# 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

符号三角形示例：

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

```
void Triangle::Backtrack(int t)
{
    if ((count>half) || (t*(t-1)/2-count>half)) return;
    if (t>n) sum++;
    else
        for (int i=0;i<2;i++)
        { //以0表示-, 以1表示+
            p[1][t]=i;
            count+=i;
            for (int j=2; j<=t; j++)
            {
                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);
            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

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。

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

设第 $i$ 行的皇后，放置在 $X_i$ 列上。则n后问题就是发现所有可能的序列  $(x_1, x_2, \dots, x_n)$ 。

要使得 $n$ 个皇后放置均不在同一行、同一列，同一个对角线上，则只需要满足如下条件：

- 1) 对任意不同的 $i, j$ ，应满足 $x_i \neq x_j$ 不在同一行、同一列
- 2) 在正对角线（斜率为1）： $i + x_i = j + x_j \rightarrow i - j = x_j - x_i$   
在反对角线（斜率为-1）： $i - x_i = j - x_j \rightarrow i - j = x_i - x_j$   
同一正、反对角线： $|i - j| = |x_i - x_j|$

# 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:
  - 1)不同列:  $x_i \neq x_j$
  - 2)不处于同一正、反对角线:  $|i-j| \neq |x_i - x_j|$

```
bool Queen::Place(int k)
{
    for (int j=1; j<k; j++)
        if ((abs(k-j)==abs(x[j]-x[k])) || (x[j]==x[k]))
            return false;
    return true;
}
```

//递归回溯

```
void Queen::Backtrack(int t)
{
    if (t>n) sum++; //输出结果x[0:n]
    else
        for (int i=1; i<=n; i++)
        {
            x[t]=i;
            if (Place(t)) Backtrack(t+1);
        }
}
```

```
public static long Queen::nQueen(int nn)
{
    n=nn;
    sum=0;
    x=new int[n+1];
    for (i=0; i<=n; i++) x[i]=0;
    backtrack(1);
    return sum++;
}
```

//long sum, int n, int []x为静态成员变量

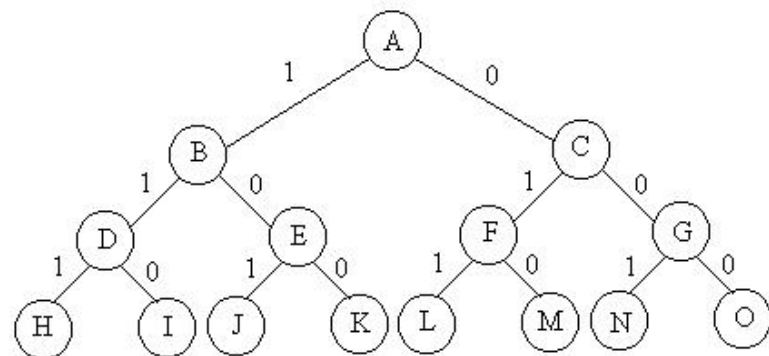


# n后问题

```
// 迭代回溯
private static void nQueen:backtrack()
{
    // k表示当前第t层, x[k]表示第t层的值
    x[1]=0;
    int k=1;
    while (k>0)
    {
        x[k]+=1;
        while ( (x[k]<=n && !(Place(k) ) x[k]+=1;
        if (x[k]<=n)
        {//可以放置在k位置
            if (k==n)
            { //最后一个皇后
                sum++;
                //得到一个解, 输出x[1:n]
            }
            else
            {
                k++; x[k]=0; //放置下一个(层) 皇后
            }
        }
        else
        {
            k--; //不可以放置, 回退到上一层皇后位置
        }
    }
}
```

# 0-1背包问题

- 解空间：子集树
- 可行性约束函数  $\sum_{i=1}^n w_i x_i \leq c_1$
- 上界函数：



Private static double **Bound**(int i)

{// 计算上界

double cleft = c - cw; // 剩余容量

double b = cp;

// 以物品单位重量价值递减序装入物品

while (i <= n && w[i] <= cleft) {

    cleft -= w[i];

    b += p[i];

    i++;

}

// 装满背包

if (i <= n) b += p[i]/w[i] \* cleft;

return b;

}

double c; //背包容量

int n; //物品数

double []w; //物品重量数组

double []p; //物品价值数组

double cw; //当前重量

double cp; //当前价值

double bestp; //当前最优价值

**Bound ()** 计算后半段最大价值的时候，使用的是一个贪心算法。尽管切割的情况是不被同意的，可是能够用这个结果来进行估算

# 0-1背包问题

//递归回溯

```
private static double Backtrack(int i)
```

```
{
```

```
    if (i>n)
```

```
    { //到达叶结点
```

```
        bestp=cp;
```

```
        return
```

```
    }
```

```
    //搜索子树
```

```
    if (cw+w[i]<=c)
```

```
    { //进入左子树
```

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

```
        cp+=p[i];
```

```
        Backtrack(i+1);
```

```
        cw-=w[i];
```

```
        cp-=p[i];
```

```
    }
```

```
    if (Bound(i+1)>bestp)
```

```
        Backtrack(i+1); //进入右子树
```

```
}
```

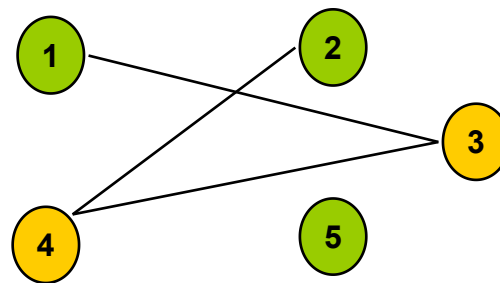
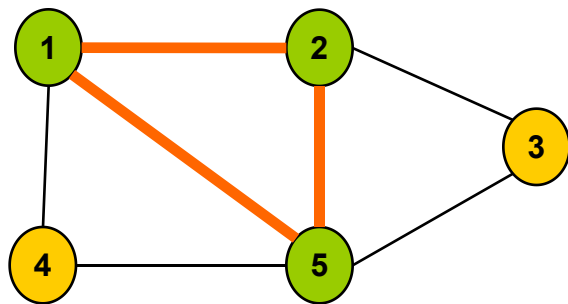
# 最大团问题

给定无向图 $G=(V, E)$ 。如果 $U \subseteq V$ ，且对任意 $u, v \in U$ 有 $(u, v) \in E$ ，则称 $U$ 是 $G$ 的**完全子图**。 $G$ 的完全子图 $U$ 是 $G$ 的**团**当且仅当 $U$ 不包含在 $G$ 的更大的完全子图中。 $G$ 的**最大团**是指 $G$ 中所含顶点数最多的团。

如果 $U \subseteq V$ 且对任意 $u, v \in U$ 有 $(u, v) \notin E$ ，则称 $U$ 是 $G$ 的**空子图**。 $G$ 的空子图 $U$ 是 $G$ 的**独立集**当且仅当 $U$ 不包含在 $G$ 的更大的空子图中。 $G$ 的**最大独立集**是 $G$ 中所含顶点数最多的独立集。

对于任一无向图 $G=(V, E)$ 其补图 $\bar{G}=(V, E_1)$ 定义为： $V_1=V$ ，且 $(u, v) \in E_1$ 当且仅当 $(u, v) \notin E$ 。

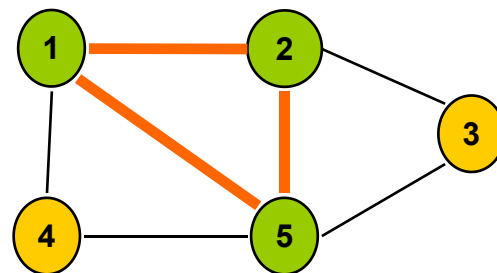
**$U$ 是 $G$ 的最大团当且仅当 $U$ 是 $\bar{G}$ 的最大独立集。**



# 最大团问题

- 解空间：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。
- 上界函数：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。

```
void Clique::Backtrack(int i)
{ // 计算最大团
    if (i > n) { // 到达叶结点
        for (int j = 1; j <= n; j++) bestx[j] = x[j];
        bestn = cn; return; }
    // 检查顶点 i 与当前团的连接
    int OK = 1;
    for (int j = 1; j < i; j++)
        if (x[j] != 0 && a[i][j]) {
            // j 是最大团中的节点，且 i 与 j 不相连
            OK = 0; break; }
    if (OK)
    { // 节点 i 满足最大团，进入左子树
        x[i] = 1; cn++;
        Backtrack(i+1);
        x[i] = 0; cn--; // 恢复状态，为回溯做准备
    }
    if (cn + n - i > bestn) { // 进入右子树
        x[i] = 0;
        Backtrack(i+1); }
}
```



int x[n] 当前解；cn 当前顶点数  
bestn 当前最大顶点数 bestx 当前最优解  
bool a[n][n]； n 为图的顶点数

## 复杂度分析

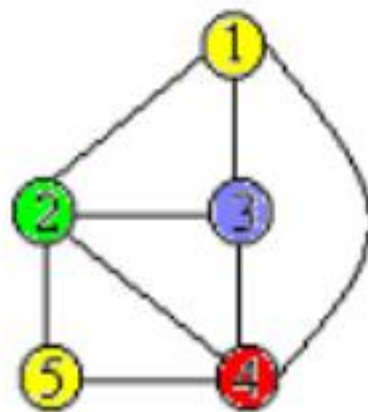
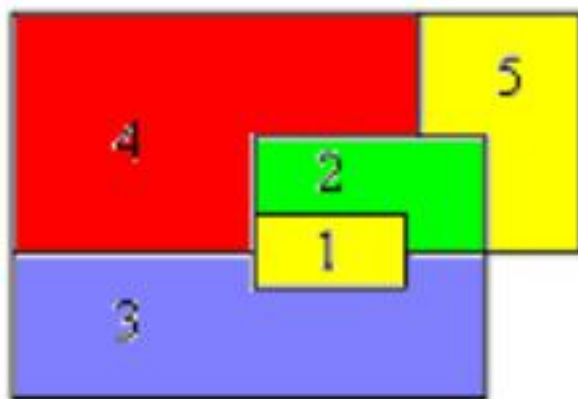
最大团问题的回溯算法 **backtrack** 所需的计算时间显然为  $O(n2^n)$ 。

# 进一步改进

- 选择合适的搜索顺序，可以使得上界函数更有效的发挥作用。例如在搜索之前可以将顶点按度从小到大排序。这在某种意义上相当于给回溯法加入了启发性。
- 定义  $S_i = \{v_i, v_{i+1}, \dots, v_n\}$ ，依次求出  $S_n, S_{n-1}, \dots, S_1$  的解。从而得到一个更精确的上界函数，若  $cn + S_i \leq \max$  则剪枝。同时注意到：从  $S_{i+1}$  到  $S_i$ ，如果找到一个更大的团，那么  $v_i$  必然属于找到的团，此时有  $S_i = S_{i+1} + 1$ ，否则  $S_i = S_{i+1}$ 。因此只要  $\max$  的值被更新过，就可以确定已经找到最大值，不必再往下搜索了。

# 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

- 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。

```
void Color
```

```
{
```

```
if (t>n)
```

```
sum+
```

```
for (in
```

```
cout
```

```
cout <
```

```
}
```

```
else
```

```
for (
```

```
x[t]
```

```
if (Ok
```

```
}
```

```
}
```

```
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;
```

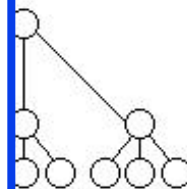
```
return true;
```

```
}
```

## 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

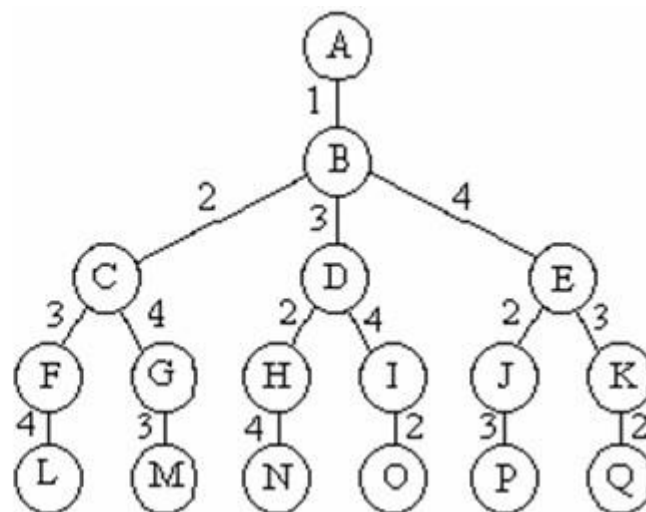
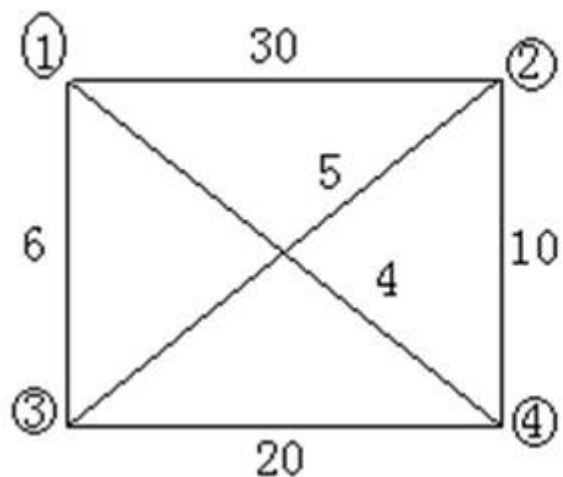
$$\sum_{i=0}^{n-1} m^i (mn) = nm(m^n - 1) / (m - 1) = O(nm^n)$$





# 旅行售货员问题

旅行售货员问题(travelling salesman problem)是一类组合最优化问题, 设有一个售货员从城市1出发, 到城市2, 3,  $\dots$ ,  $n$ 去推销货物, 最后回到城市1. 假定任意两个城市  $i, j$  间的距离  $d_{ij}$  ( $d_{ij}=d_{ji}$ ) 是已知的, 问他应沿着什么样的路线走, 才能使走过的路线最短 (总旅费最小)?



# 旅行售货员问题

旅行售货员问题就是在一个完全网络中，找出一个具有最小权的哈密顿圈，寻求旅行售货员问题的有效算法似乎是没有希望的，它属于NP完全类，一个可行的办法是首先求一个哈密顿圈，然后适当修改，以得到具较小权的另一个哈密顿圈。

## 意义：

旅行售货员问题有着明显的实际意义，除售货员之外，邮局里负责到各个信箱取信的邮递员；去各个分局送邮件的汽车等都会类似地遇到这个问题；还有一些问题表面上似乎与之无关，而实质上却可以归结为旅行售货员问题求解，如计算机线路问题、无中间存储的工件加工问题等。

# 旅行售货员问题

在递归中，当 $i=n$ 时，当前扩展结点是排列树的叶结点的父结点。此时算法检测图 $G$ 是否存在一条从顶点 $x[n-1]$ 到顶点 $x[n]$ 的边和一条从顶点 $x[n]$ 到顶点1的边，如果这两条边都存在，则找到一条旅行售货员的回路，此时算法再判断这条回路的费用是否优于当前已经找到的最优回路的费用 $bestc$ 。

```
if (a[x[n-1]][x[n]] < MAX_VALUE &&  
    a[x[n]][1] < MAX_VALUE &&  
    (cc + a[x[n-1]][x[n]] + a[x[n]][1] < bestc || bestc > MAX_VALUE))
```

当 $i < n$ 时，当前结点扩展到排列树的第 $i-1$ 层，图 $G$ 中存在从顶点 $x[i-1]$ 到顶点 $x[i]$ 的边时， $x[1:i]$ 构成图 $G$ 的一条路径，且当 $x[1:i]$ 的费用小于当前最优值时，算法进入排列树的第 $i$ 层；否则，剪去相应的子树。

用 $CC$ 来记录当前路径 $X[1:i]$ 的费用

# 旅行售货员问题

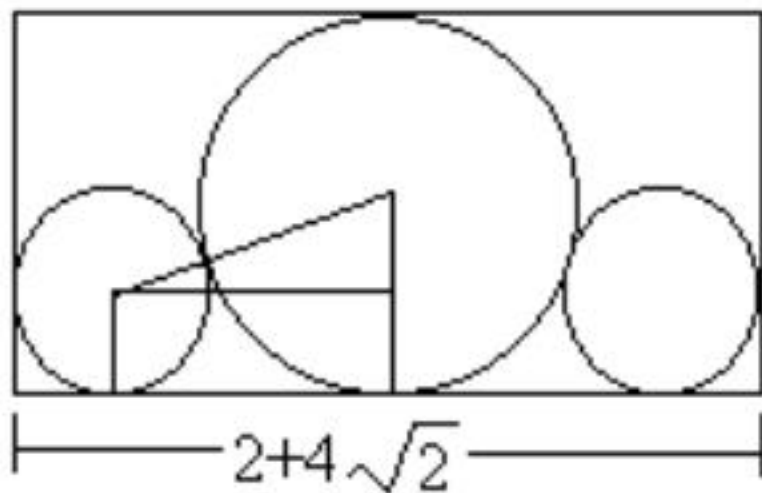
```
template<class Type>
void Traveling<Type>:
{ //float a[][]为邻接矩阵
    if (i == n) {
        if (a[x[n-1]][x[n]] <
            (cc + a[x[n-1]][x[1]])) {
            bestc = cc + a[x[n-1]][x[1]];
        }
    }
    else {
        for (int j = i; j <= n; j++)
            // 是否可进入x[j]子树?
            if ( a[x[i-1]][x[j]] < MAX_VALUE && (cc + a[x[i-1]][x[i]] < bestc || bestc == MAX_VALUE)) {
                // 搜索子树
                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]);
            }
    }
}
```

## 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度为 $O(n!)$ 。

# 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。例如，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 $2 + 4\sqrt{2}$



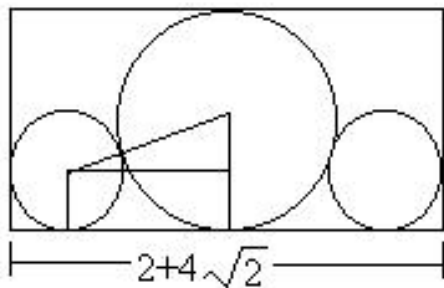
# 圆排列问题

```
void Circle::Backtrack(int t)
{
    if (t>n) Compute();
    else
        for (int i = t; i <= n; i++) {
```

```
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++) {
```

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。

```
}
```



```
// 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;
}
```

# 连续邮资问题

假设国家发行了 $n$ 种不同面值的邮票，并且规定每张信封上最多只允许贴 $m$ 张邮票。连续邮资问题要求对于给定的 $n$ 和 $m$ 的值，给出邮票面值的最佳设计，在1张信封上可贴出从邮资1开始，增量为1的最大连续邮资区间。

例如，当 $n=5$ 和 $m=4$ 时，面值为 $(1,3,11,15,32)$ 的5种邮票可以贴出邮资的最大连续邮资区间是1到70。

# 连续邮资问题

- 解向量：用 $n$ 元组 $x[1:n]$ 表示 $n$ 种不同的邮票面值，并约定它们从小到大排列。 $x[1]=1$ 是唯一的选择。
- 可行性约束函数：已选定 $x[1:i-1]$ ，最大连续邮资区间是 $[1:r]$ ，接下来 $x[i]$ 的可取值范围是 $[x[i-1]+1:r+1]$ 。

## 如何确定 $r$ 的值？

计算 $X[1:i]$ 的最大连续邮资区间在本算法中被频繁使用到，因此势必要找到一个高效的方法。考虑到直接递归的求解复杂度太高，我们不妨尝试计算用不超过 $m$ 张面值为 $x[1:i]$ 的邮票贴出邮资 $k$ 所需的最少邮票数 $y[k]$ 。通过 $y[k]$ 可以很快推出 $r$ 的值。事实上， $y[k]$ 可以通过递推在 $O(n)$ 时间内解决：

```
for (int j=0; j<= x[i-2]*(m-1);j++)  
    if (y[j]<m)  
        for (int k=1;k<=m-y[j];k++)  
            if (y[j]+k<y[j+x[i-1]*k]) y[j+x[i-1]*k]=y[j]+k;  
while (y[r]<maxint) r++;
```



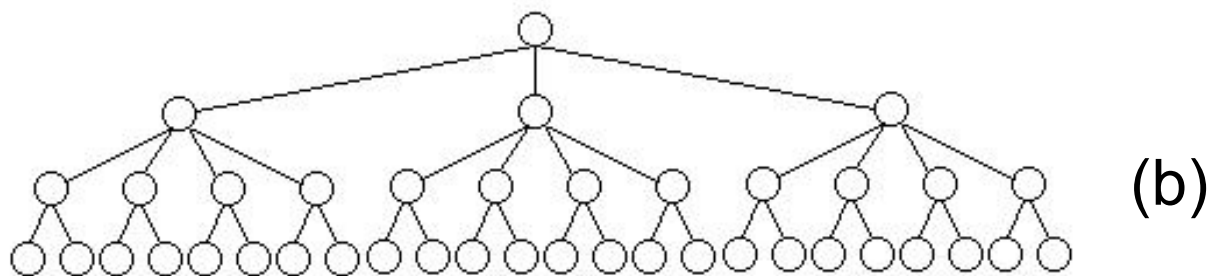
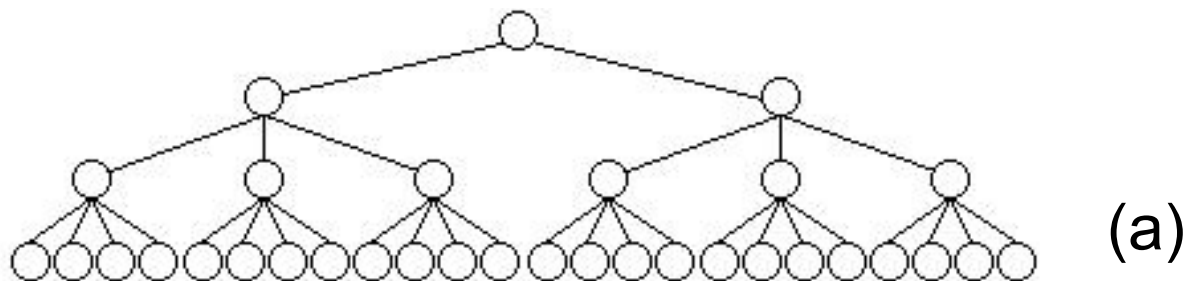
# 回溯法效率分析

通过前面具体实例的讨论容易看出，回溯算法的效率在很大程度上依赖于以下因素：

- (1)产生 $x[k]$ 的时间；
  - (2)满足显约束的 $x[k]$ 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。

# 重排原理

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。