第5章 回溯法

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。

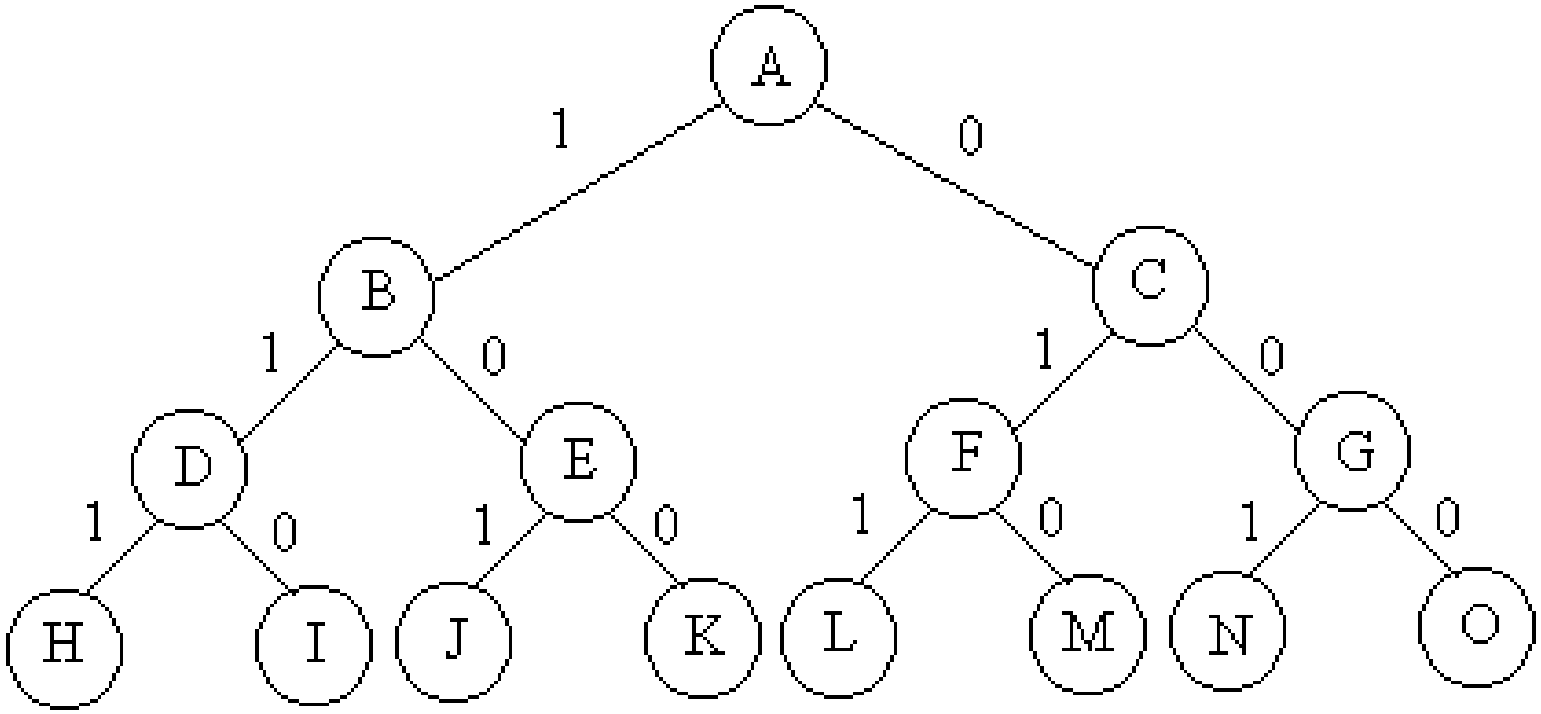
回溯法的算法框架

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(x1, x2, ..., xn)的形式。
- 显约束：对分量xi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）。

问题的解空间



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

问题状态的生成是从开始结点出发，在搜索过程中不断扩展已有的结点来完成。

生成问题状态有两种本质上不同的方法：深度优先生成方法和广度优先生成方法。

- 深度优先的问题状态生成法：对一个结点R，一旦产生了它的一个子结点C，就把C作为当前结点进行扩展。在完成对以C为根的子树的穷尽搜索之后，将R重新变成当前结点，继续生成R的下一个子节点（如果存在）。
- 广度优先的问题状态生成法：在对一个结点扩展产生了它的一个子结点后，继续生成它的下一个子结点，直至所有的子结点都生成。

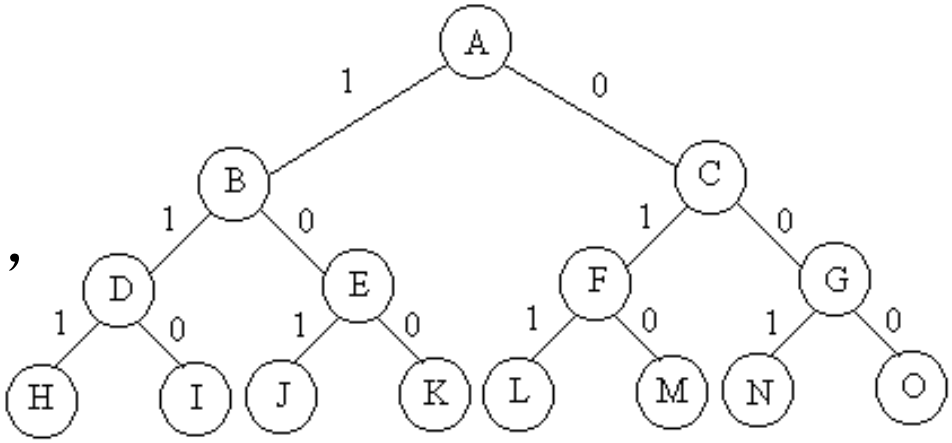
生成问题状态的基本方法

问题状态生成过程中结点的状态：

- 扩展结点：一个正在产生子结点的结点称为扩展结点；
- 活结点：结点已经生成但其子结点还未全部生成的结点称为活结点；
- 死结点：一个所有子结点已经产生的结点称做死结点；

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

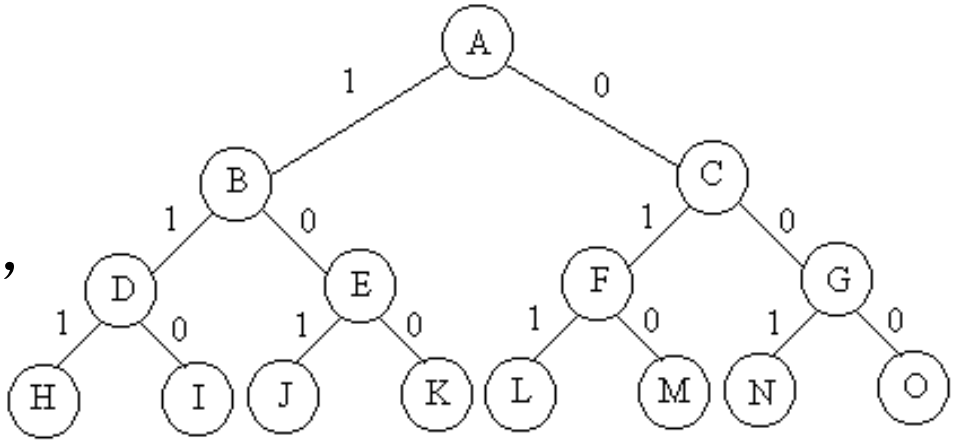


举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 从图中的根节点开始搜索，

  开始时，根节点A是唯一的活结点，也是当前的扩展结点。在这个扩展结点处，可以沿纵深方向移至B或C；
- 假设选择先移至B。此时A和B是活结点，B成为当前扩展结点。由于选取了w1，故在B处剩余背包容量r=14，价值为45；
- 从B处，可以移至D或E。由于移至D至少需要w2=15的背包容量，而现在剩余背包容量r=14，故移至D导致不可行解，返回B；

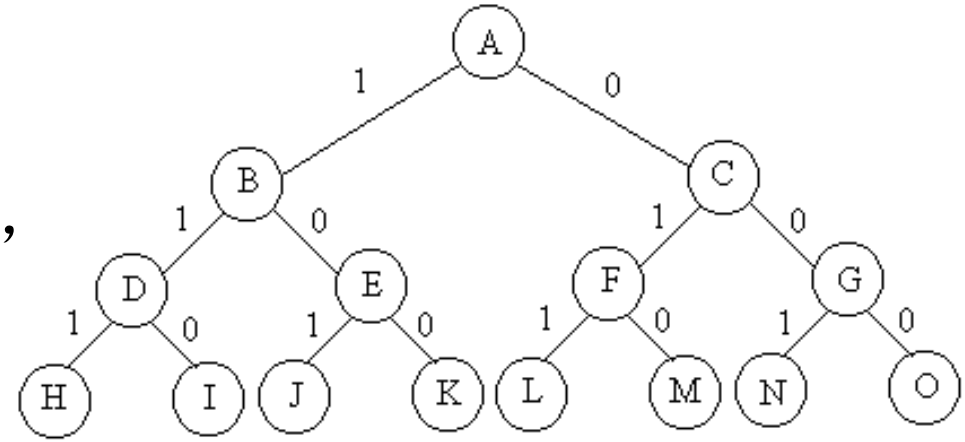
回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 搜索至E处不需要背包容量，因而是可行的。移至E，此时E成为新的扩展结点，A、B和E是活结点，在E处，r=14，价值为45。从E处可以向纵深移至J或K；
- 移向J导致不可行解，于是移向K，K成为新的扩展结点。由于K是叶结点，故得到一个可行解。这个解相应的价值为45。xi的取值由根节点到K的路径唯一确定，即x={1，0，0}。由于在K处已不能再向纵深扩展，所以K成为死结点；

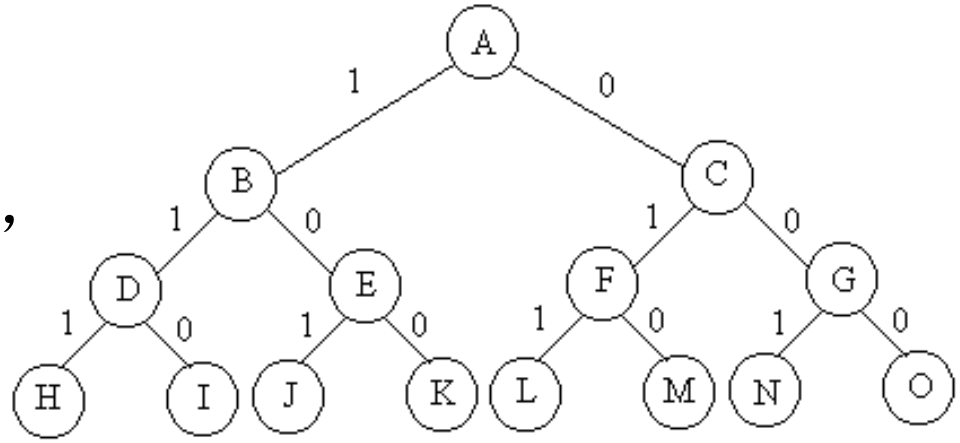
回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- 再返回E处，此时E处也没有可扩展的结点，它也成为死结点；
- 接下来返回B，B同样成为死结点；
- 返回A，A再次成为当前扩展节点。A还可继续扩展，到达C。
- 在C处，r=30，价值为0。从C可移向F或G。
- 假设移至F，成为新的扩展结点。结点A、C和F是活结点。此时，r=15，价值为25。
- 从F继续移向L，此时r=0，价值为50。由于L是叶结点，而且是迄今为止价值最高的解，因此记录此可行解。

回溯法的基本思想



举例：n=3的0-1背包问题，c=30，w=[16, 15, 15]， p=[45, 25, 25]。

- L不可扩展，又返回到F。
- 按此方式继续搜索遍整个解空间。搜索结束后找到的最优解是相应0-1背包问题的最优解。

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。

回溯法的基本思想

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);
    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：递归的深度

f(n, t),g(n,t): 当前扩展结点的未搜索过的子树的起始编号和终止编号

h(i): 当前扩展结点的第i个可选值

constraint(t): 约束函数，判断x[1:t]是否满足约束条件

bound(t): 限界函数，判断x[1:t]是否有可能得到最优解

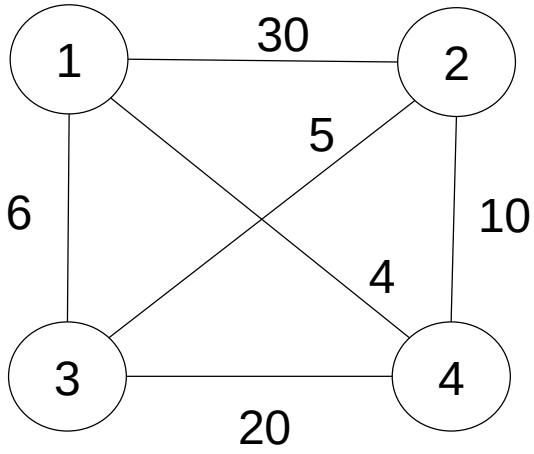
迭代回溯

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){ //还有活结点
        if(f(n,t)<=g(n,t)) //还有未搜索过的子节点
            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++; //深入搜索子节点
        else t--; //所有子节点搜索完成，返回
}
```

旅行售货员问题

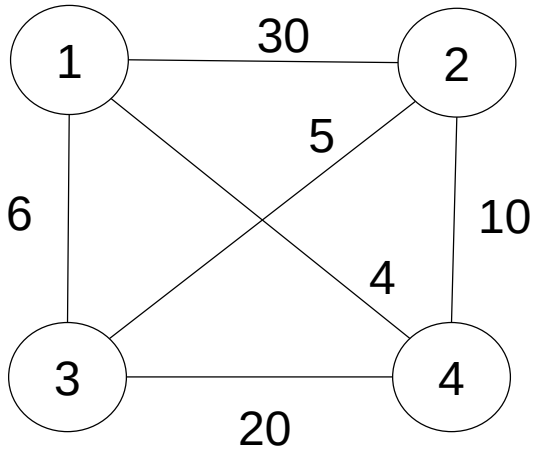
某售货员要到若干城市去推销商品，已知各城市之间的路程。他要选定一条从驻地出发，经过每个城市一遍，最后回到驻地的路线，使总的路程最小。

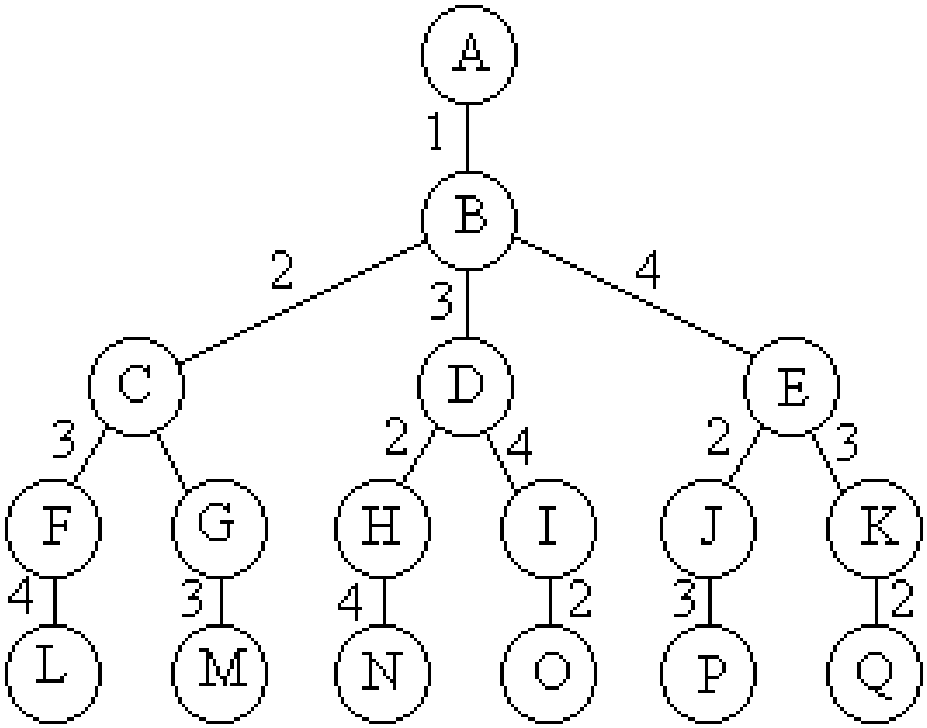


设G=(V,E)是一个带权图。图中的一条周游路线是包括V中的每个顶点在内的一条回路。

旅行售货员问题

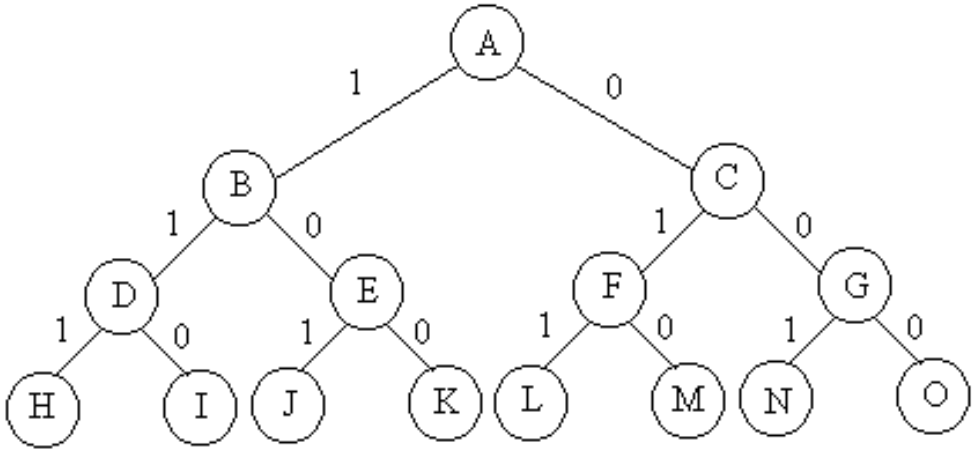
右图是左图例子的解空间树。从根节点A到叶结点L的路径上边的标号组成一条周游路线，如1-2-3-4-1

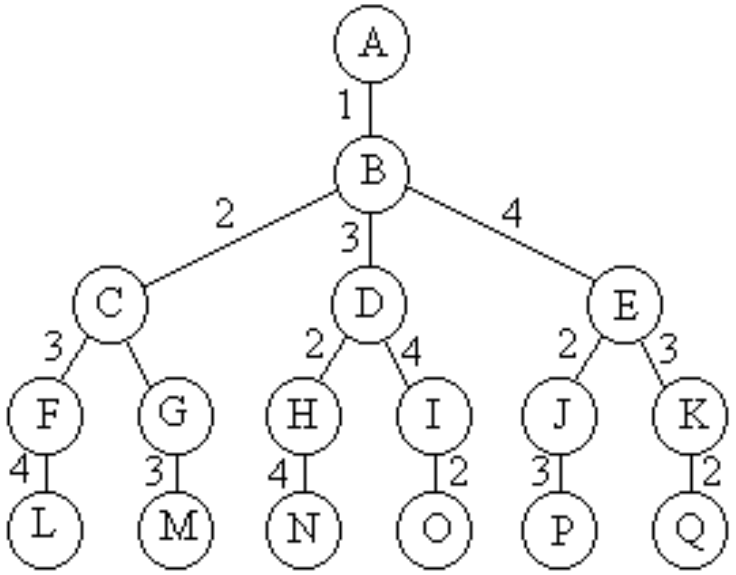




子集树与排列树

下图的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树：



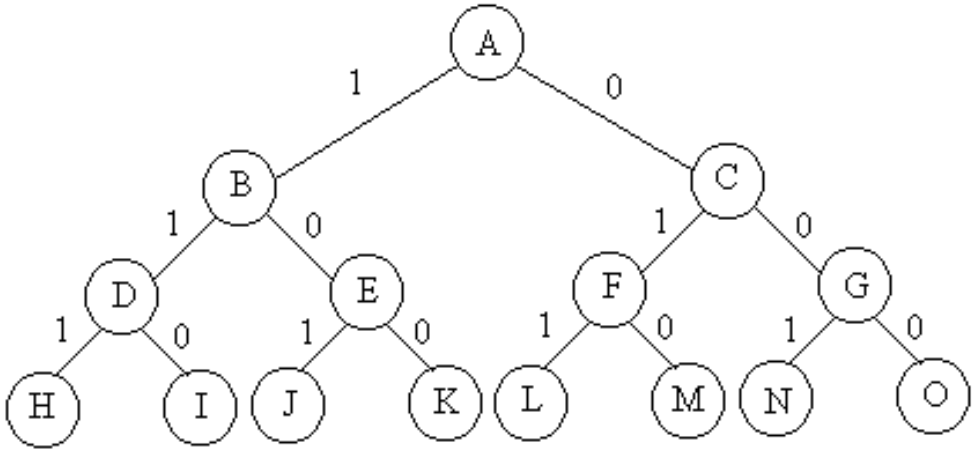


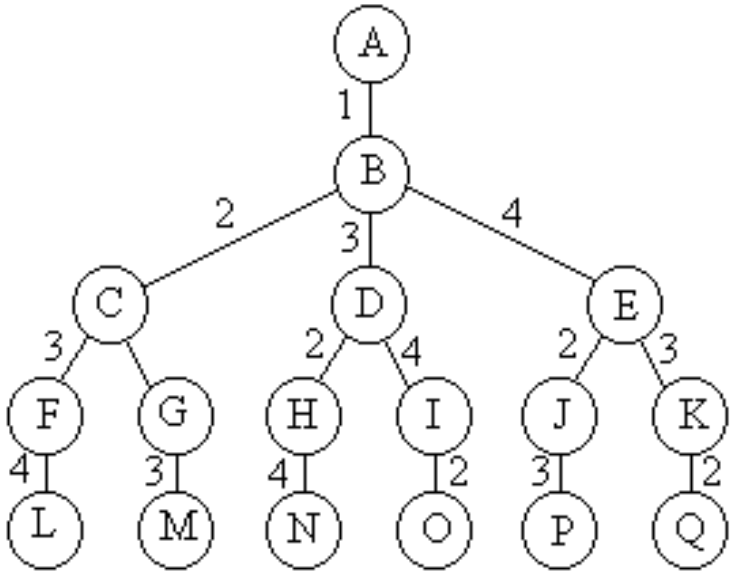
子集树：

- 当所给的问题是从n个元素的集合S中找出满足某种性质的子集时，相应的解空间树为子集树（左图）。如0-1背包问题，通常有2^n个叶结点，结点总个数为$2^{n+1}-1$。遍历子集树的任何算法均需$\Omega(2^n)$的计算时间。

子集树与排列树

下图的两棵解空间树是用回溯法解题时常遇到的两类典型的解空间树：

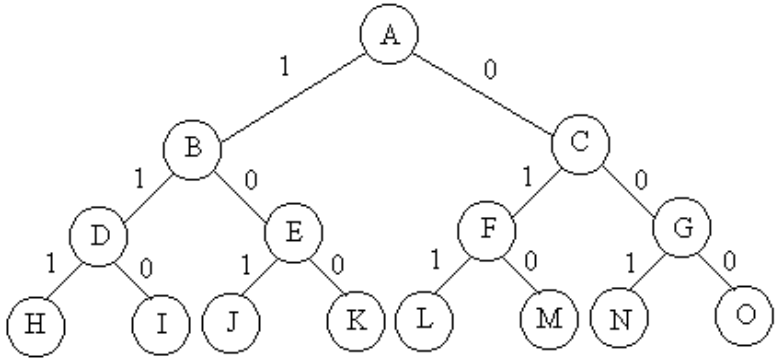


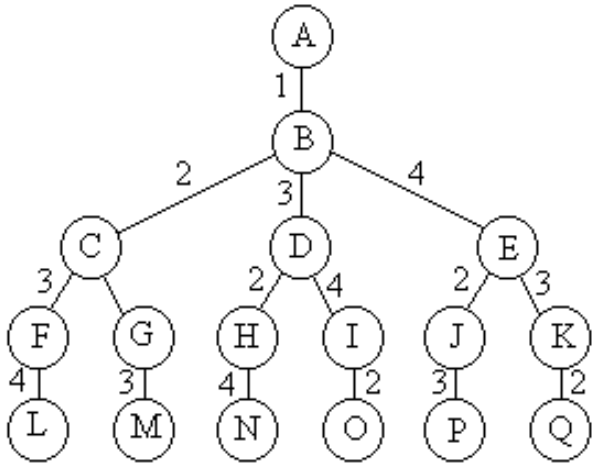


排列树：

- 当所给的问题是确定n个元素满足某种性质的排列时，相应的解空间为排列树（右图）。排列树通常有n！个结点。因此遍历排列树需要Ω(n!)的计算时间。旅行售货员问题的解空间树就是一棵排列树。

子集树与排列树





用回溯法搜索子集树的一般算法：

```
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);
}
```

用回溯法搜索排列树的一般算法：

```
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c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

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

- 例1：n=3，c1=c2=50，w=[10,40,40]，可将集装箱1和集装箱2装上第一艘轮船，将集装箱3装上第二艘轮船。
- 例2： n=3，c1=c2=50，w=[20,40,40]，则无法将这3个集装箱装上轮船。

装载问题

- 当$\sum_{i=1}^{n} w_i = C_1 + C_2$时，装载问题等价于子集和问题：

  给定整数集合S和一个整数t，判定是否存在S的一个子集S’⊆S，使得S’中整数的和为t。

- 当C1=C2且$\sum_{i=1}^{n} w_i = 2C_1$时，装载问题等价于划分问题：给定一个正整数的集合S，是否可以将其分割成两个子集合，使两个子集合的数加起来的和相等。

装载问题

有一批共n个集装箱要装上2艘载重量分别为c1和c2的轮船，其中集装箱i的重量为wi，且 $\sum_{i=1}^{n} w_i \le c_1 + c_2$

装载问题要求确定是否有一个合理的装载方案可将这n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。

- 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。

（因为有解，所以能装下，第一艘装越多，第二艘剩余空间越大。）

装载问题

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C1。由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\max \sum_{i=1}^{n} w_i x_i$$

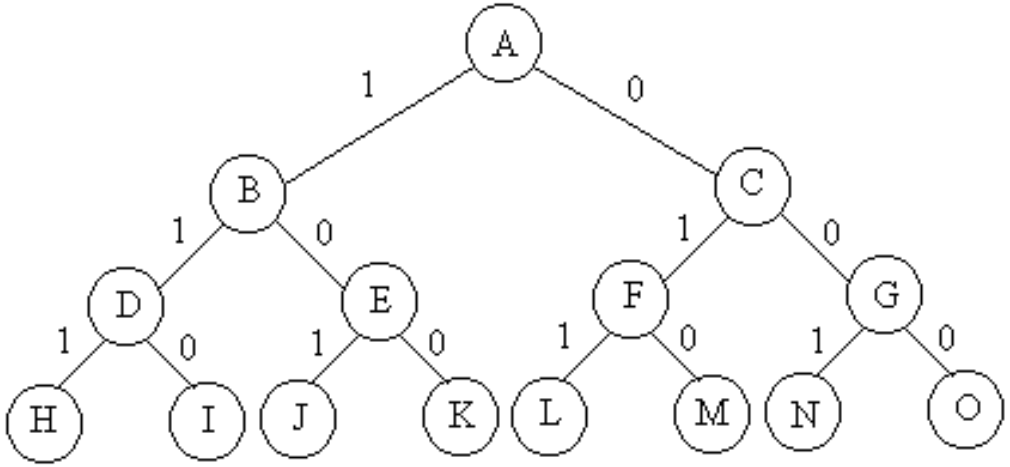
$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

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)：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 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

当前载重量 cw = $\sum_{j=1}^{i} w_j x_j$

当前最优载重量 bestw

```
private static void backtrack(int i){ //搜索第i层结点
     if(i>n)  //到达叶结点
          if(cw>bestw)
               bestw = cw;
          return;
     if(cw+w[i]<=c)  //搜索左子树
          cw += w[i];
          backtrack(i+1);
          cw -= w[i];
     backtrack(i + 1); //搜索右子树
}
```

backtrack(1)

装载问题

引入上界函数，剪去不含最优解的子树。

剩余集装箱的重量 $r = \sum_{j=i+1}^{n} w_j$

```
private static void backtrack(int i){
//搜索第i层结点
    if(i>n) //到达叶结点
        bestw = cw;
        return;
    r -= w[i];
    if(cw+w[i]<=c) //搜索左子树
        cw += w[i];
        backtrack(i + 1);
        cw -= w[i];
    if(cw+r>bestw) //搜索右子树
        backtrack(i + 1);
    r += w[i];
}
```

初始化

```
for(int i=1;i<=n;i++){
    r+=w[i];
}
```

- 引入上界函数之后，在达到一个叶结点时，不必再检查该叶结点是否优于当前最优解。
- 搜索到的每个叶结点都是当前找到的最优解。

装载问题

记录最优解

x：记录从根至当前结点的路径

bestx：记录当前最优解

```
private static void backtrack(int i){// 搜索第i层结点
    if(i>n)  //到达叶结点
        for(int j=1;j<=n;j++)
            bestx[j]=x[j];
        bestw = cw;
        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)  //搜索右子树
        x[i] = 0;
        backtrack(i + 1);
    r += w[i];
  }
```

批处理作业调度

给定n个作业的集合$\{J_1,J_2,\ldots,J_n\}$。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$f=\sum_{i=1}^{n} F_{2i}$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

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。

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)；

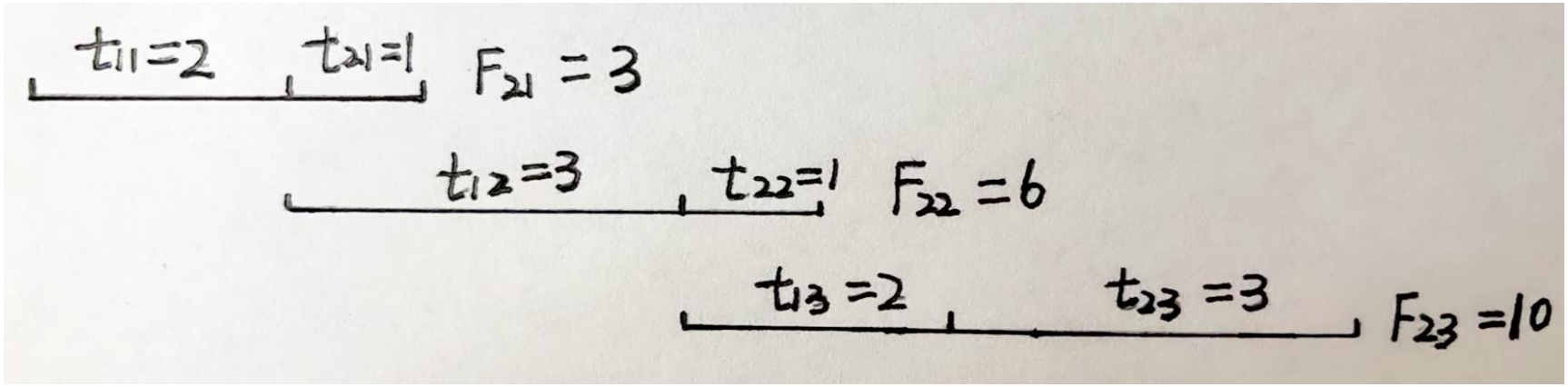
它们所相应的完成时间和分别是19，18，20，21，19，19。

易见，最佳调度方案是(1,3,2)，其完成时间和为18。

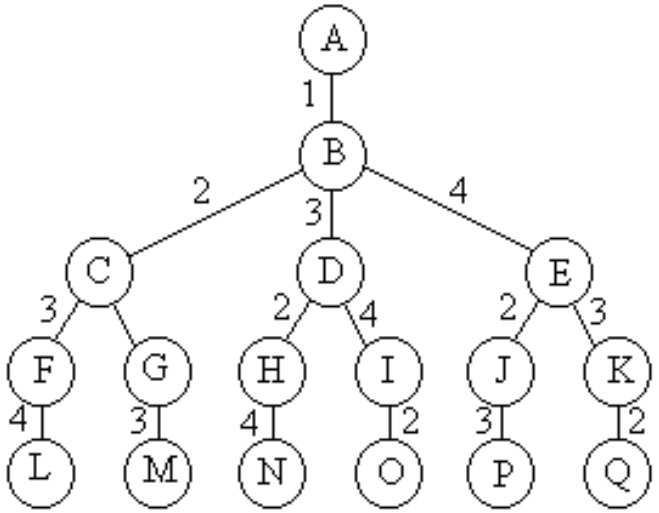
批处理作业调度

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

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，调度方案(1,2,3) 如下图所示，其完成时间为：3+6+10=19。



批处理作业调度




```
private static void backtrack(int i){
    if (i > n)
        for (int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestf = f;
    else
        for (int j = i; j <= n; j++)
            f1 += m[x[j]][1];
            f2[i]=(f2[i-1]>f1?f2[i-1]:f1)+m[x[j]][2]; //红色计算作业x[j]在机器2上的开始时间
            f+=f2[i];
            if (f < bestf) {
                swap(x,i,j);
                backtrack(i+1);
                swap(x,i,j);
            f1 -= m[x[j]][1];
            f -= f2[i];
}
```

backtrack(1)

```
public class FlowShop
        int  n,    // 作业数
            f1,    // 机器1完成处理时间
             f,    // 完成时间和
         bestf;    // 当前最优值
    int [][] m;    // 各作业所需的处理时间
     int [] x;     // 当前作业调度
int [] bestx;      // 当前最优作业调度
 int [] f2;        // 机器2完成处理时间
```

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”-”。

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

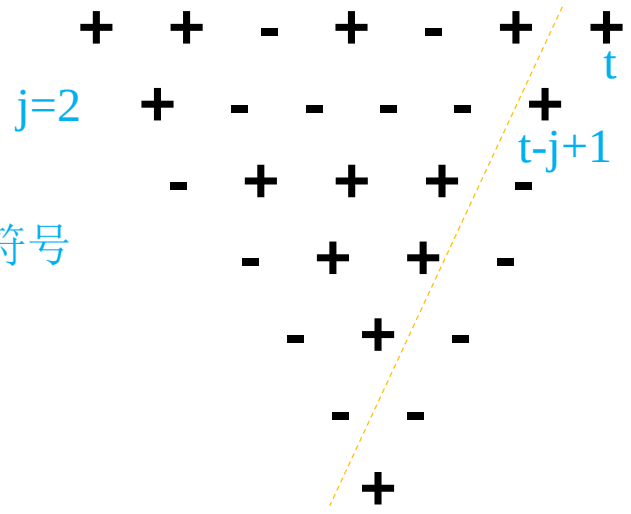
在一般情况下，符号三角形的第一行有n个符号。符号三角形问题要求对于给定的n，计算有多少个不同的符号三角形，使其所含的”+”和”-”的个数相同。

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



count: “+”的个数; half: n*(n+1)/4

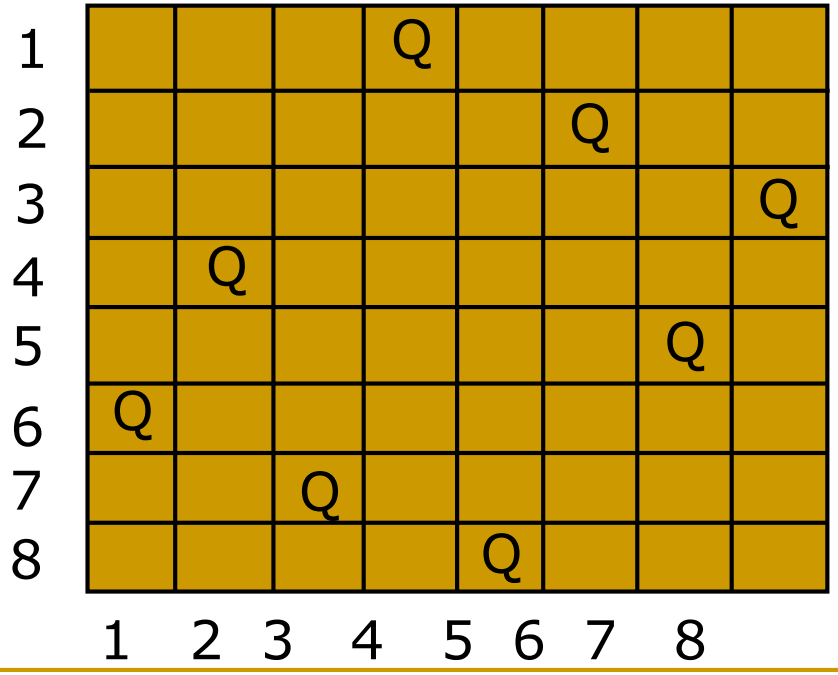
```
private static void backtrack(int t){//处理第t个符号
    if((count>half)||(t*(t-1)/2-count>half))
        return;
    if(t>n)
        sum++;
    else
        for(int i=0;i<2;i++)
            p[1][t]=i; //第1行第t个符号
            count+=i;
            for(int j=2;j<=t;j++) //第2至t行最后一列
                p[j][t-j+1]=p[j-1][t-j+1]^p[j-1][t-j+2];
                count+=p[j][t-j+1];
            backtrack(t+1);
            for(int j=2;j<=t;j++)
                count-=p[j][t-j+1];
            count-=i;
}
```

backtrack(1)

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

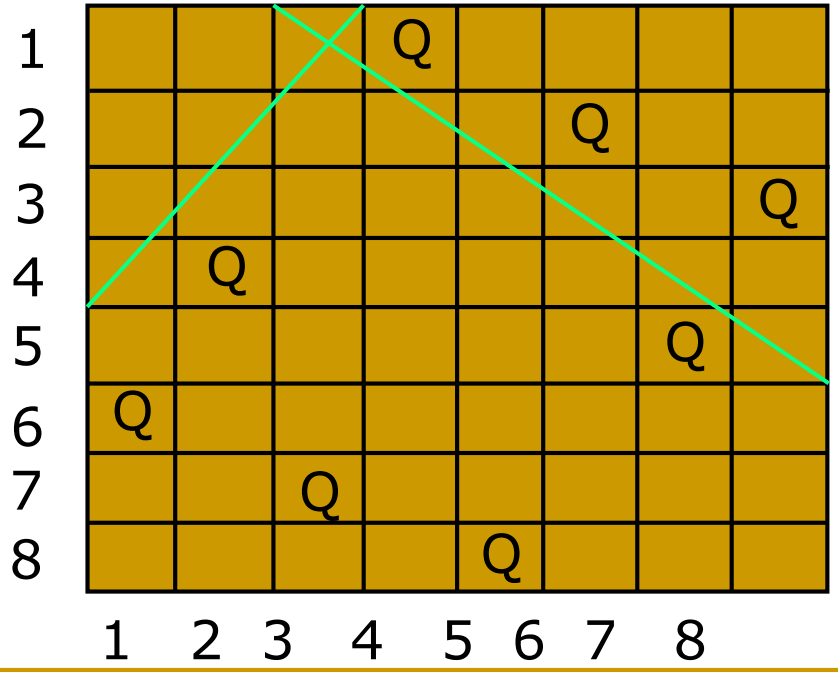
在n×n格的棋盘上放置彼此不受攻击的n个皇后。按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。n后问题等价于在n×n格的棋盘上放置n个皇后，任何2个皇后不放在同一行或同一列或同一斜线上。



n后问题

- 解向量：$(x_1, x_2, \ldots, x_n)$
- 显约束：x_i=1,2, … ,n，表示第i行的皇后放在第x[i]列
- 隐约束：1)不同列：$x_i \neq x_j$

  2)不处于同一斜线同一正、反对角线：$|i-j| \neq |x_i-x_j|$



n后问题

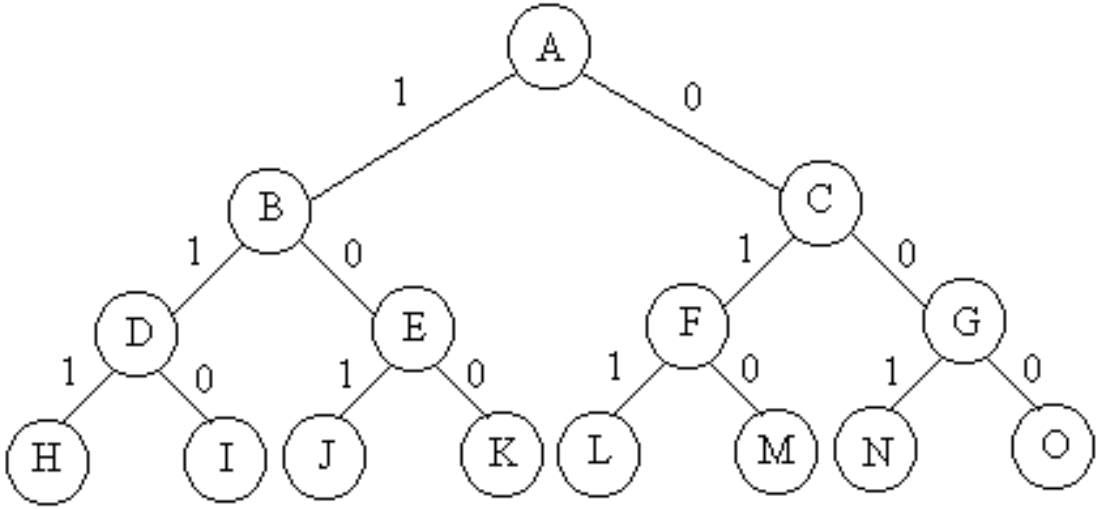
```
private static void backtrack(int t){
    if(t>n)
        sum++;
    else
        for(int i=1;i<=n;i++)
            x[t]=i;
            if(place(t))
                backtrack(t+1);
}
```

```
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;
}
```

backtrack(1)

0-1背包问题

- 解空间：子集树
- 可行性约束：$\sum_{i=1}^{n} w_i x_i \le c_1$,进入左子树时判定；
- 上界函数：cp+r>bestp，cp是当前价值，r是当前剩余物品价值，bestp是当前最优价值，进入右子树时判定。



0-1背包问题

bestp: 当前最优值;
cw: 背包中已装入物品的重量;
cp: 背包中已装入物品的价值

```
void backtrack(int i){
    if(i>n)
        bestp=cp;
        return;
    if(cw+cw[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) //进入右子树
        backtrack(i+1);
}
```

backtrack(1)

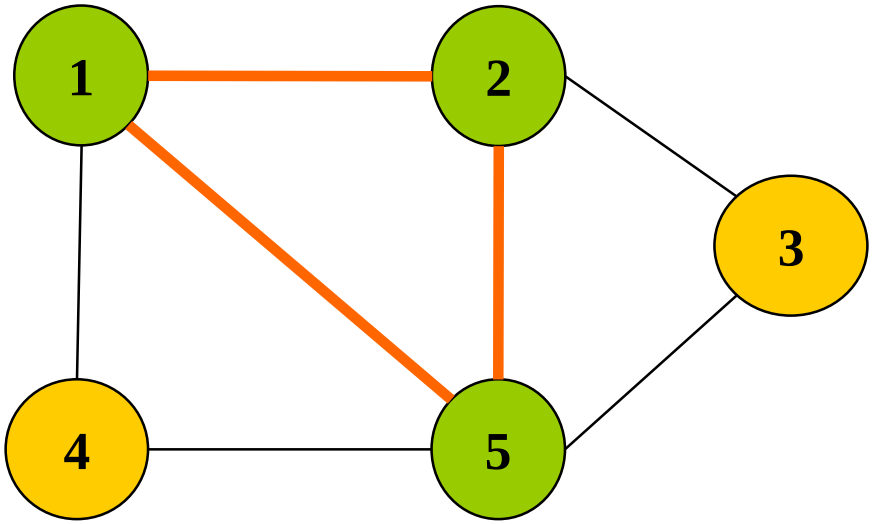
0-1背包问题

物品已按单位重量的价值从大到小排序。物品i的单位重量价值为p[i]/w[i]，p[i]为物品i的价值，w[i]为物品i的重量。

```
private static double bound(int i) { // 计算上界
      double cleft = c - cw;  // 剩余容量
      double bound = cp;
      while(i <= n &&  w[i] <= cleft) // 以物品单位重量价值递减序装入
            cleft -= w[i];
            bound += p[i];
            i++;
      if(i <= n) // 装满背包
            bound += p[i]/w[i]*cleft;
      return bound;
}
```

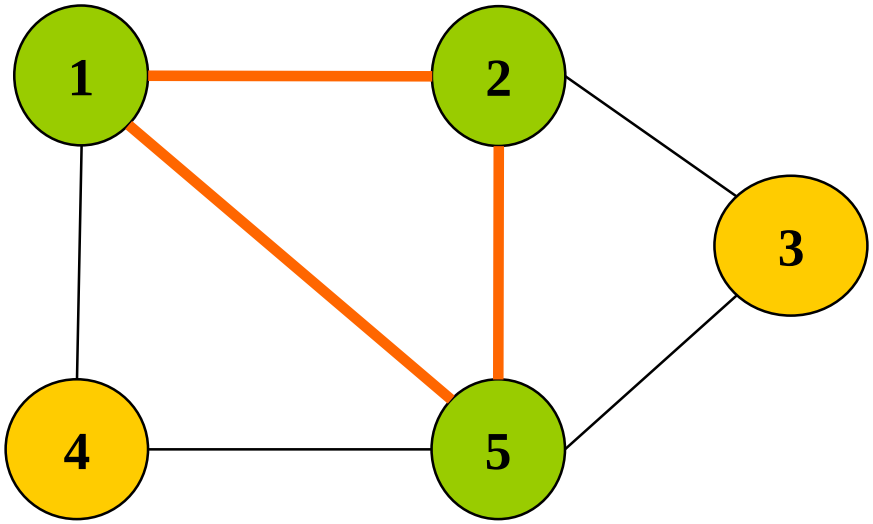
最大团问题

- 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。例如{1,2}。
- 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，例如{1,2,5}。
- G的最大团是指G中所含顶点数最多的团。



最大团问题

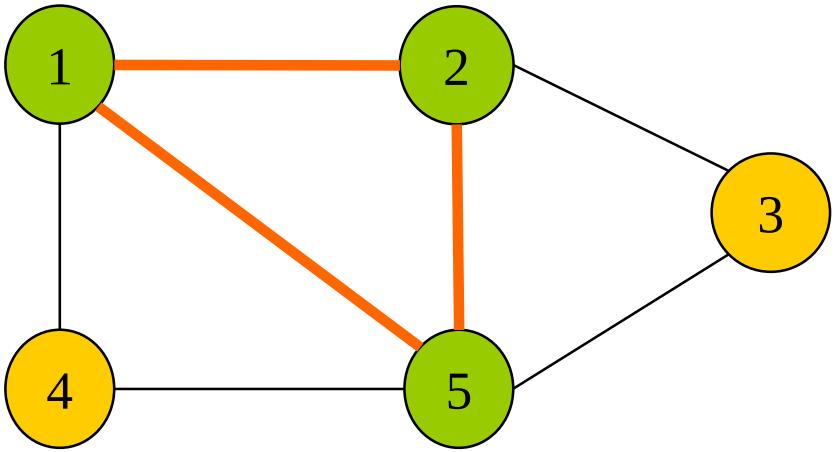
- 如果U⊆V且对任意u，v∈U有(u，v)∉E，则称U是G的空子图，例如{2,4}。
- G的空子图U是G的独立集当且仅当U不包含在G的更大的空子图中。
- G的最大独立集是G中所含顶点数最多的独立集。



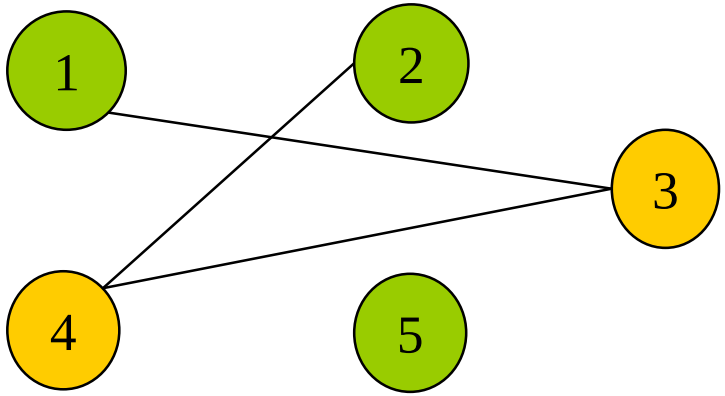
最大团问题

- 对于任一无向图G=(V，E)其补图G’=(V1，E1)定义为：V1=V，且(u，v)∈E1当且仅当(u，v)∉E。

U是G的最大团当且仅当U是G’的最大独立集。

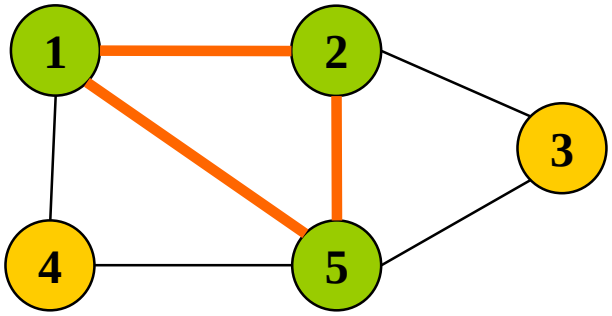


图G



图G’

最大团问题



最大团问题找出给定图G=(V, E)的最大团，可看做图G的顶点集V的子集选取问题。

- 解空间：子集树
- 可行性约束函数：顶点i到已选入顶点集中的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。

最大团问题

```
private static void backtrack(int i) {
    if(i>n)    // 到达叶结点
        for(int j = 1; j <= n; j++)
            bestx[j] = x[j];
        bestn = cn;
        return;
    boolean ok = true;
    for(int j=1; j<i; j++)  // 检查顶点 i 与当前团的连接
        if(x[j] == 1 && !a[i][j])  // i与j不相连
            ok = false;
            break;
    if(ok)    // 进入左子树
        x[i] = 1;  cn++;
        backtrack(i + 1);
        cn--;
    if(cn+n-i>bestn) // 进入右子树
        x[i] = 0;
        backtrack(i + 1);
}
```

backtrack(1)

复杂度分析

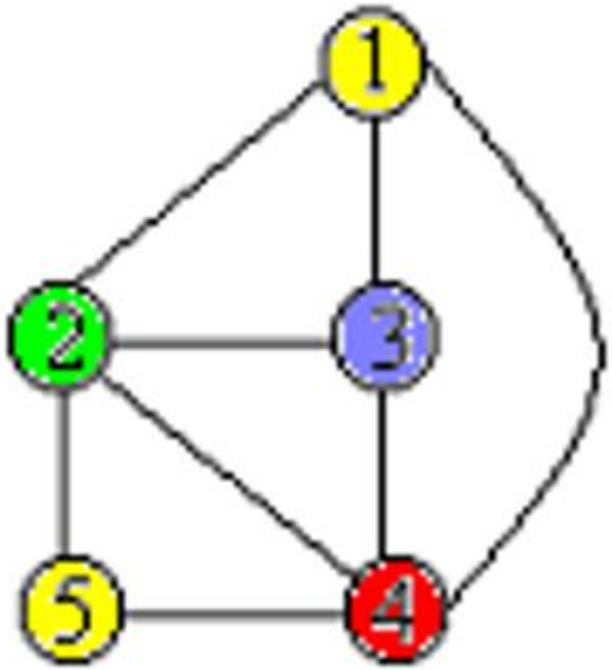
最大团问题的回溯算法**backtrack**所需的计算时间为$O(n2^n)$。

进一步改进算法的建议

- 选择合适的搜索顺序，可以使得约束函数和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Si={v_i,v_{i+1},...,v_n}，依次求出S_n,S_{n-1},...,S_1的解。从而得到一个更精确的上界函数，若cn+(S_i的解)<= 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。

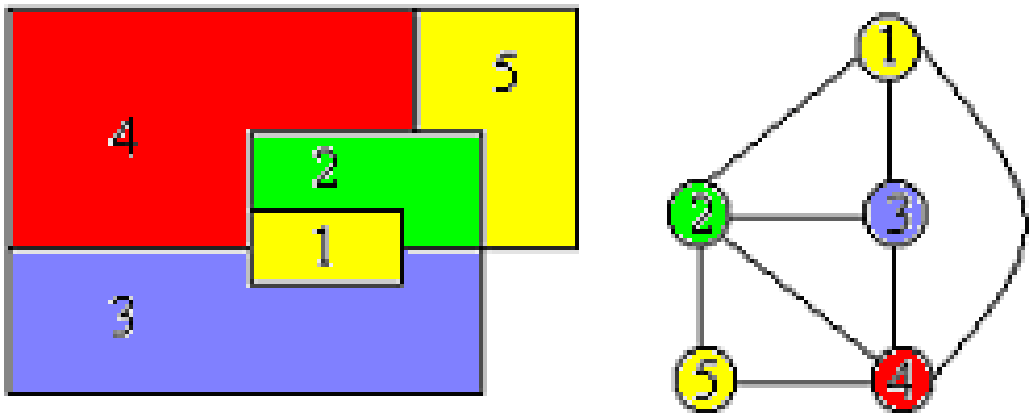
图的m着色问题

- 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：找出图的所有不同m着色法。



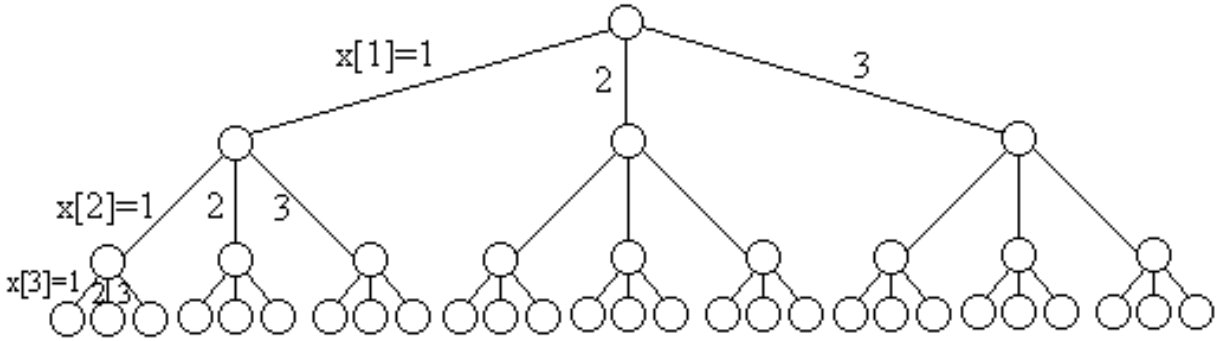
四色猜想

- 四色猜想：在一个平面或球面上的任何地图能够只用4种颜色来着色，使相邻的国家在地图上着不同的颜色。
- 假设每个国家在地图上是单连通域，两个国家相邻是指这两个国家有一段长度不为0的公共边界，而不仅有一个公共点。
- 这样的地图用平面图表示，地图上的每个区域相应平面图的一个顶点。相邻区域相应的两个顶点之间有一条边。



图的m着色问题

- 解向量：$(x_1, x_2, \ldots, x_n)$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。




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

```
private static boolean ok(int k){
    // 检查颜色可用性
    for(int j=1;j<=n;j++)
        if(a[k][j] && (x[j]==x[k]))
            return false;
    return true;
}
```

backtrack(1)

旅行售货员问题

- 从一个给定的城市出发，访问所有城市之后回到出发的城市。
- 解空间：排列树，由x[1:n]的所有排列构成，类似于递归算法Perm。
- 当前扩展结点位于n-1层时：
  - 其是叶结点的父节点，检测是否存在一条从顶点x[n-1]到x[n]的边和一条从x[n]到1的边。如果两条边都存在则找到一条回路，若这条回路的费用小于已找到的当前最优值bestc，则更新bestc。
- 当前扩展结点位于第i-1层，i<n 时：
  - 若存在一条从x[i-1]到x[i]的边，则比较x[1:i]的费用和当前最优值bestc。若x[1:i]的费用小于bestc，则进入第i层，否则剪去相应的子树。

旅行售货员问题

```
private static void backtrack(int i){
    if(i==n)
        if(a[x[n-1]][x[n]]<MAX&&a[x[n]][1]<MAX&&
            (bestc==MAX||cc+a[x[n-1]][x[n]]+a[x[n]][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];
    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&&
              (bestc==MAX||cc+a[x[i-1]][x[j]]<bestc))
              swap(x, i, j);
              cc += a[x[i - 1]][x[i]];
              backtrack(i + 1);
              cc -= a[x[i - 1]][x[i];
              swap(x, i, j);
}
```

backtrack(2)

复杂度分析

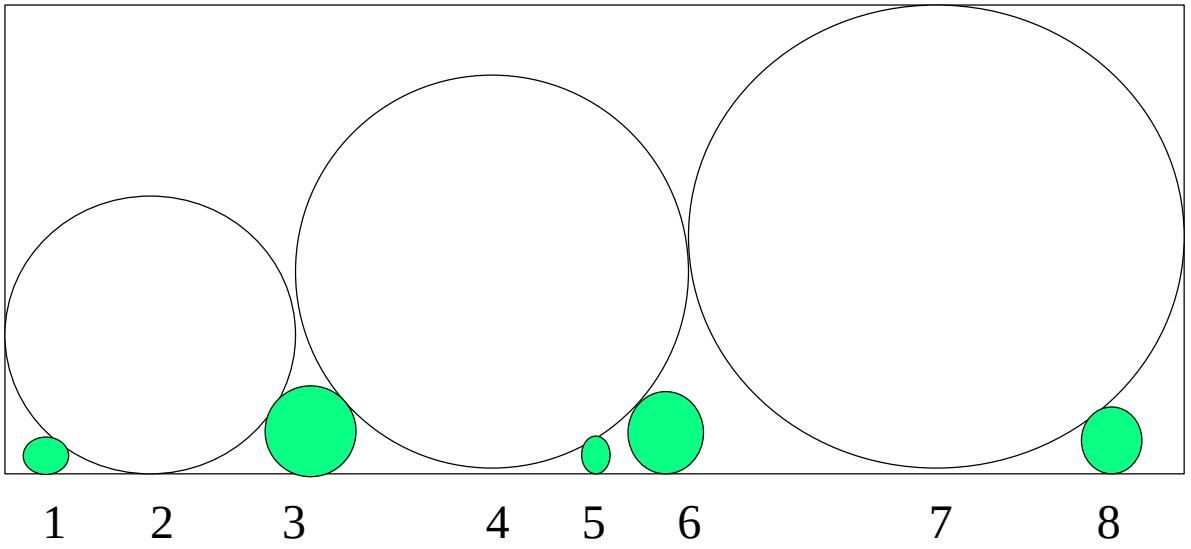
算法在最坏情况下可能需要更新当前最优解O((n-1)!)次，每次更新bestx时间为O(n)，从而整个算法的时间复杂性为O(n!)。

圆排列问题

给定n个大小不等的圆c1,c2,…,cn，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。

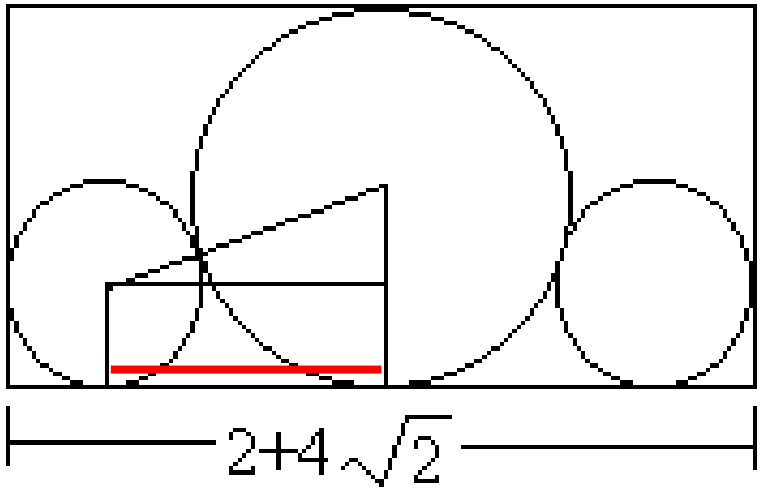
圆排列问题要求从n个圆的所有排列中找出有最小长度的圆排列。

长度最小的圆排列，必有任何一个圆都与前面某一个圆相切。



圆排列问题

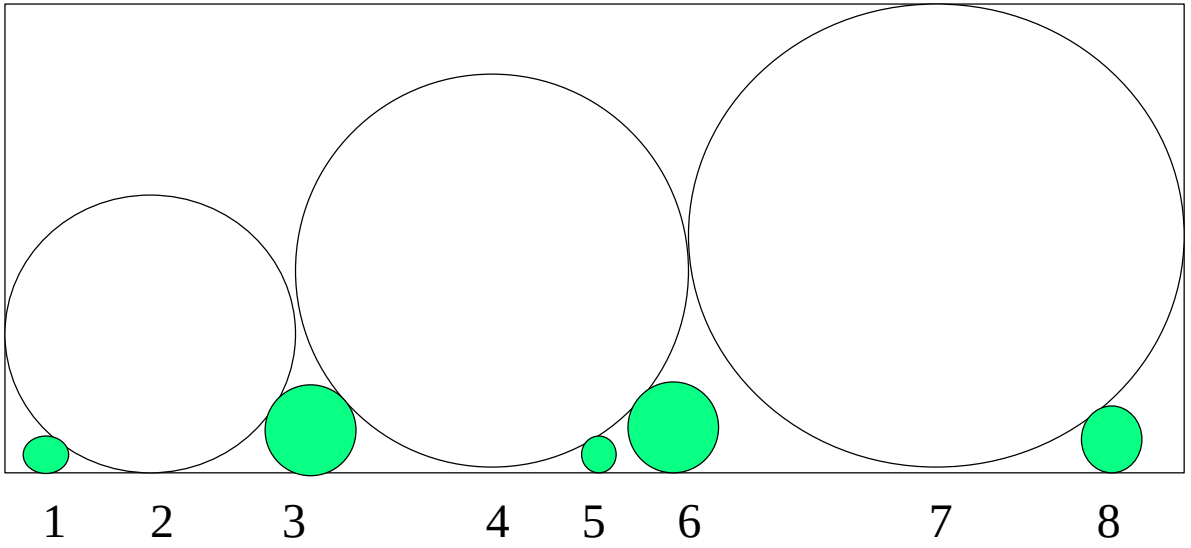
例如，当n=3，且所给的3个圆的半径分别为1，1，2时，这3个圆的最小长度的圆排列如图所示。红色直线的长度为$\sqrt{(2+1)^2-(2-1)^2}=2\sqrt{2}$，所以这三个圆的排列长度为 $2+4\sqrt{2}$ 。



圆排列问题

计算圆心坐标：

设第一个圆的圆心坐标为0，从左至右依次计算各个圆的圆心坐标。假设已计算得到圆1至i-1的圆心坐标，需计算圆i的圆心坐标。

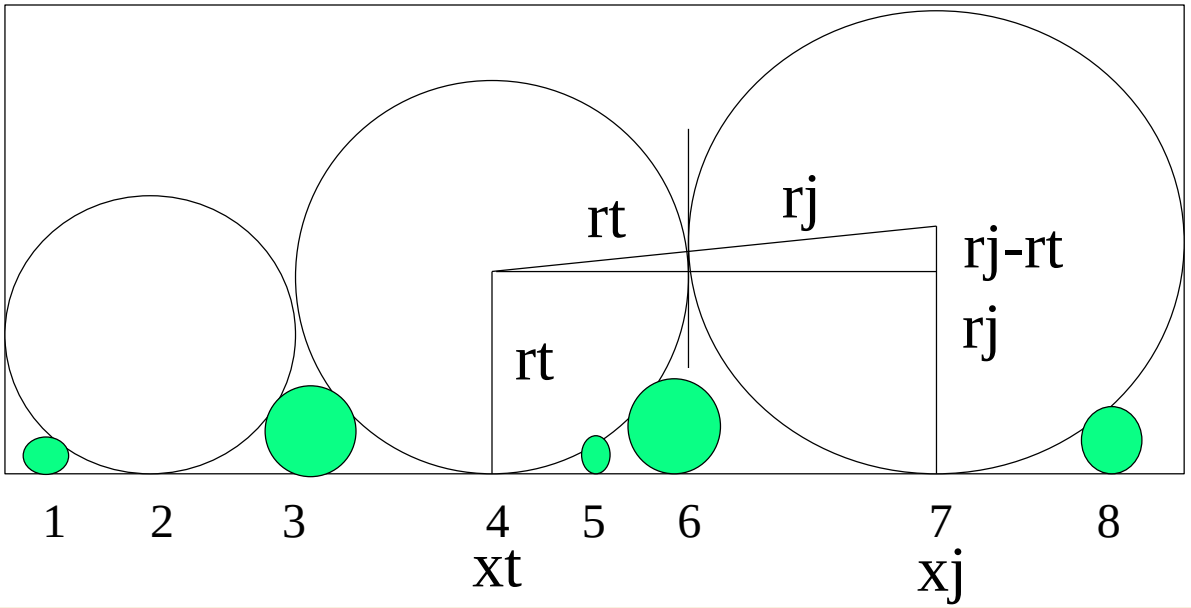


圆排列问题

计算圆心坐标：

假设圆j与圆t相切，圆j与圆t的半径分别为rj与rt，圆t的圆心坐标为xt，则圆j的圆心坐标

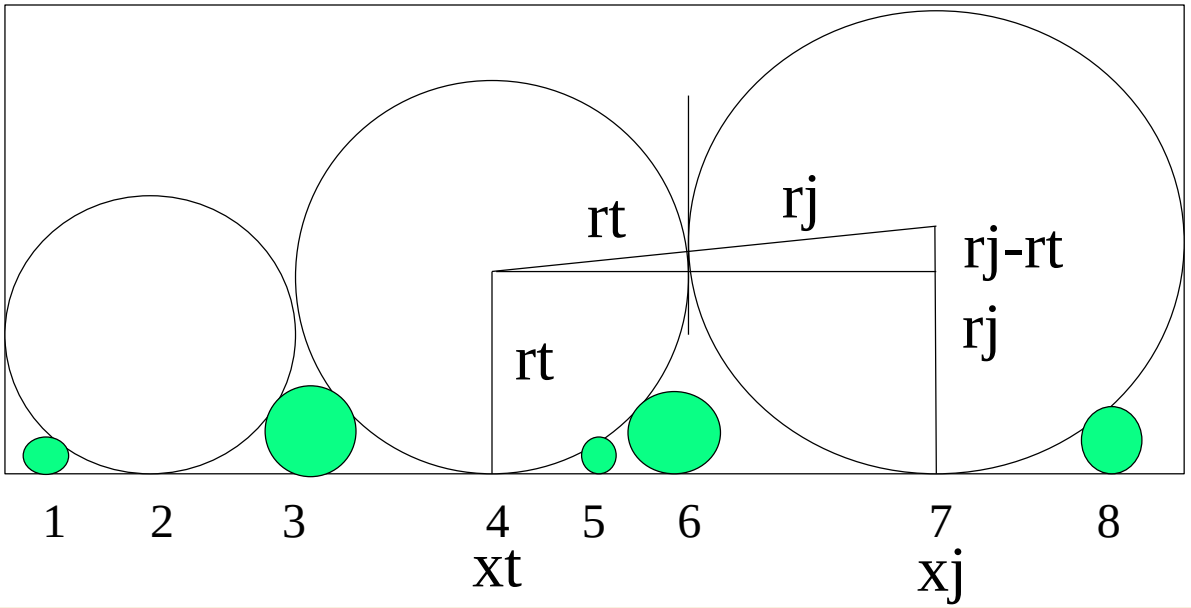
$$xj = xt + \sqrt{(rj + rt)^2 - (rj - rt)^2} = xt + 2\sqrt{rj * rt}$$



圆排列问题

计算圆心坐标：

圆i可能与之前排列1至i-1的的任意一个圆相切（如圆7与圆4相切）。根据与圆i真实相切的圆计算得到的圆心坐标是最大的，该最大值便是圆i的圆心坐标。



圆排列问题

r中存圆心坐标

```
private static void backtrack(int t){
    if(t>n)
        compute(); //计算当前圆排列的长度
    else
        for(int j=t; j<=n; j++)
            swap(r, t, j);
            float centerx=center(t); //计算圆心横坐标
            if(centerx+r[t]+r[1]<min) //下界约束
                x[t]=centerx;
                backtrack(t+1);
            swap(r, t, j);
}
```

backtrack(1)

圆排列问题

```
private static float center(int t){
     float temp=0;
     for(int j=1;j<t;j++)  //圆t可能与之前任一圆相切
         valuex=x[j]+2.0*sqrt(r[t]*r[j]);
         if(valuex>temp)
             temp=valuex;
     return temp;
}
```

```
private static void compute(){
      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;
}
```

求出所有圆的左右两侧坐标，找出最小的左侧坐标和最大的右侧坐标，相减就是圆排列的长度

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)!) 。

上述算法尚有许多改进的余地：

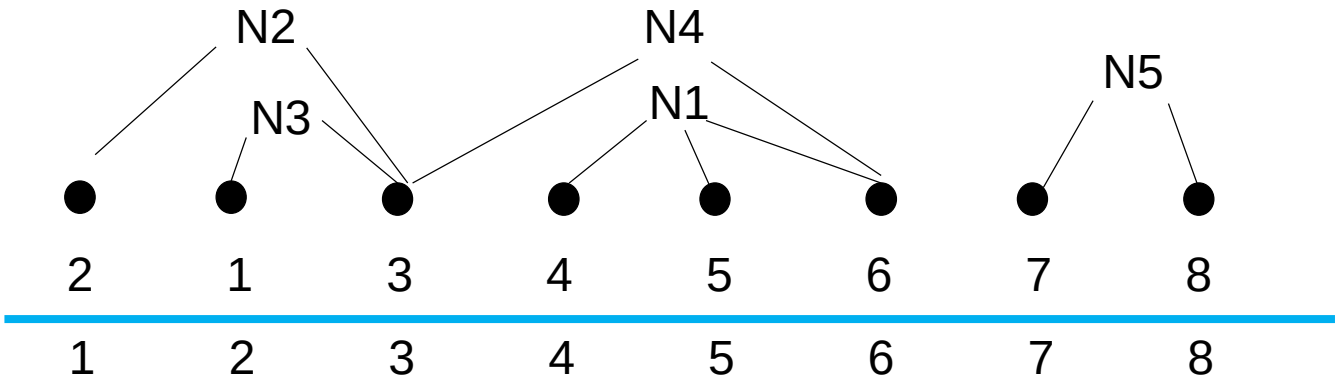
- 象1,2,…,n-1,n和n,n-1, …,2,1这种互为镜像的排列具有相同的圆排列长度，只计算一个就够了，可减少约一半的计算量。
- 如果所给的n个圆中有k个圆有相同的半径，则这k个圆产生的k!个完全相同的圆排列，只计算一个就够了。

电路板排列问题

- 将n块电路板以最佳排列方案插入带有n个插槽的机箱中。
- 设B={1,2,…,n}是n块电路板的集合。集合L={N1, N2, …, Nm}是n块电路板的m个连接块。其中每个连接块是B的一个子集，且Ni中的电路板用同一根导线连接在一起。
- 例如：n=8, m=5。给定n块电路板及其m个连接块如下：

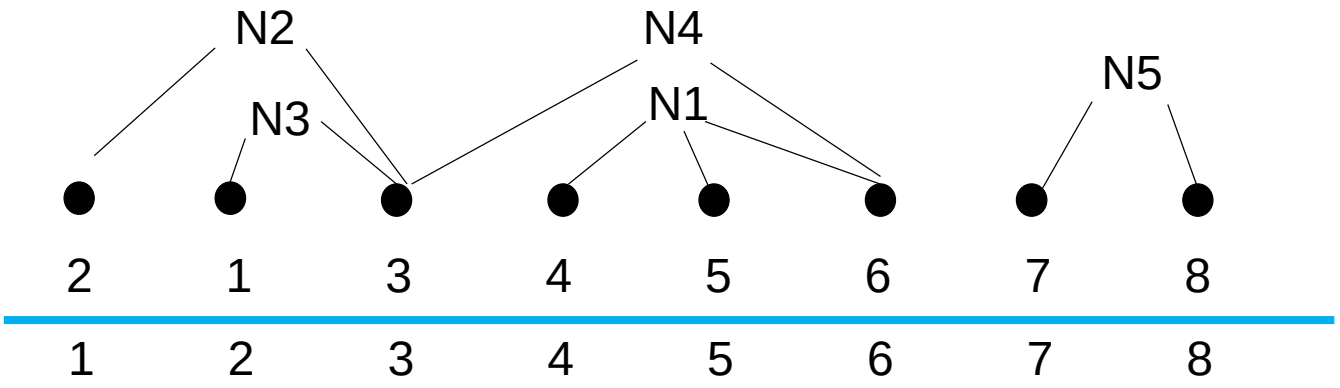
  B ={1,2,3,4,5,6,7,8}，L={N1,N2,N3,N4,N5}

  N1={4,5,6}　N2={2,3}　N3={1,3}　N4={3,6}　N5={7,8}



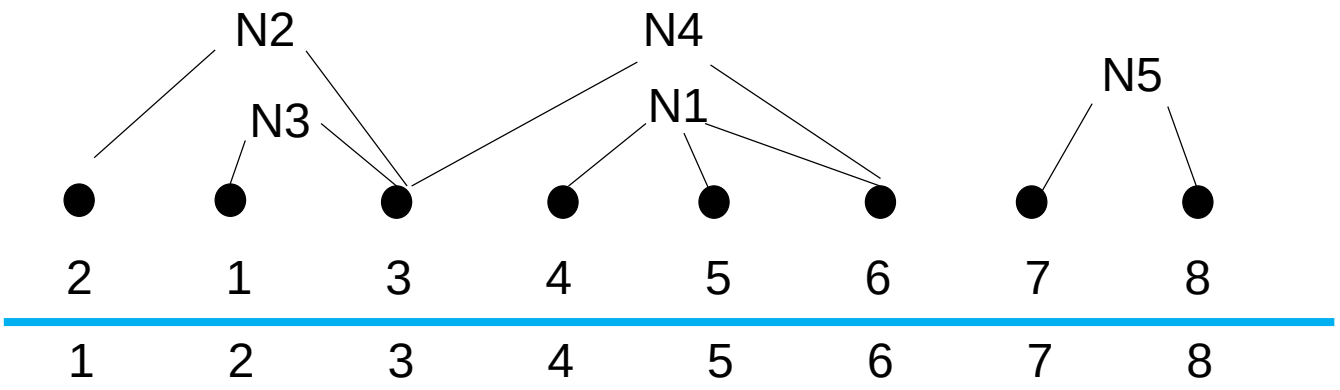
电路板排列问题

- 设x表示n块电路板的排列，即在机箱的第i个插槽插入x[i]。
- x[i]确定的电路板排列密度density(x)定义为跨越相邻电路板插槽的最大连接数。图中电路板排列的密度为2，跨越插槽2和3、4和5、5和6的连接数均为2。插槽6和7之间无跨越连接线，其余相邻插槽之间有一条跨越连线。
- 电路板排列问题要求对于给定连接块，确定电路板的最佳排列，使其具有最小密度。



电路板排列问题

- 用整形数组B表示输入，B[i][j]的值为1当且仅当电路板i在连接块N[j]中。
- 设total[j]是连接块Nj中的电路板数。对于电路板的部分排列x[1:i]，now[j]是x[1:i]中包含的Nj中的电路板数。
- 连接块Nj的连线跨越插槽i和i+1当且仅当now[j]>0且now[j]≠total[j]。据此可计算插槽i和i+1间的连接密度。



```
void backtrack(int i, int cd){
     if(i==n)
         for(int j=1;j<=n;j++)
             bestx[j] = x[j];
         bestd = cd;
     else
         for(int j=i;j<=n;j++) //交换x[i]和x[j]
            int ld = 0;
            for(int k=1;k<=m;k++) //依次判断每个连接块是否跨越x[i]和x[i+1]
                                  //从而计算i到i+1的连接数
                now[k] += B[x[j]][k];
                if(now[k]>0&&total[k]!=now[k])
                   ld++;
            if(cd>ld)
                ld = cd;
            if(ld<bestd){
                swap(x[i], x[j]);
                backtrack(i+1, ld);
                swap(x[i], x[j]);
                for(int k=1;k<=m;k++)
                    now[k] -= B[x[j]][k];
}
```

cd: 1到i电路板的最大连接数

ld: i到i+1的连接数

backtrack(1, 0)

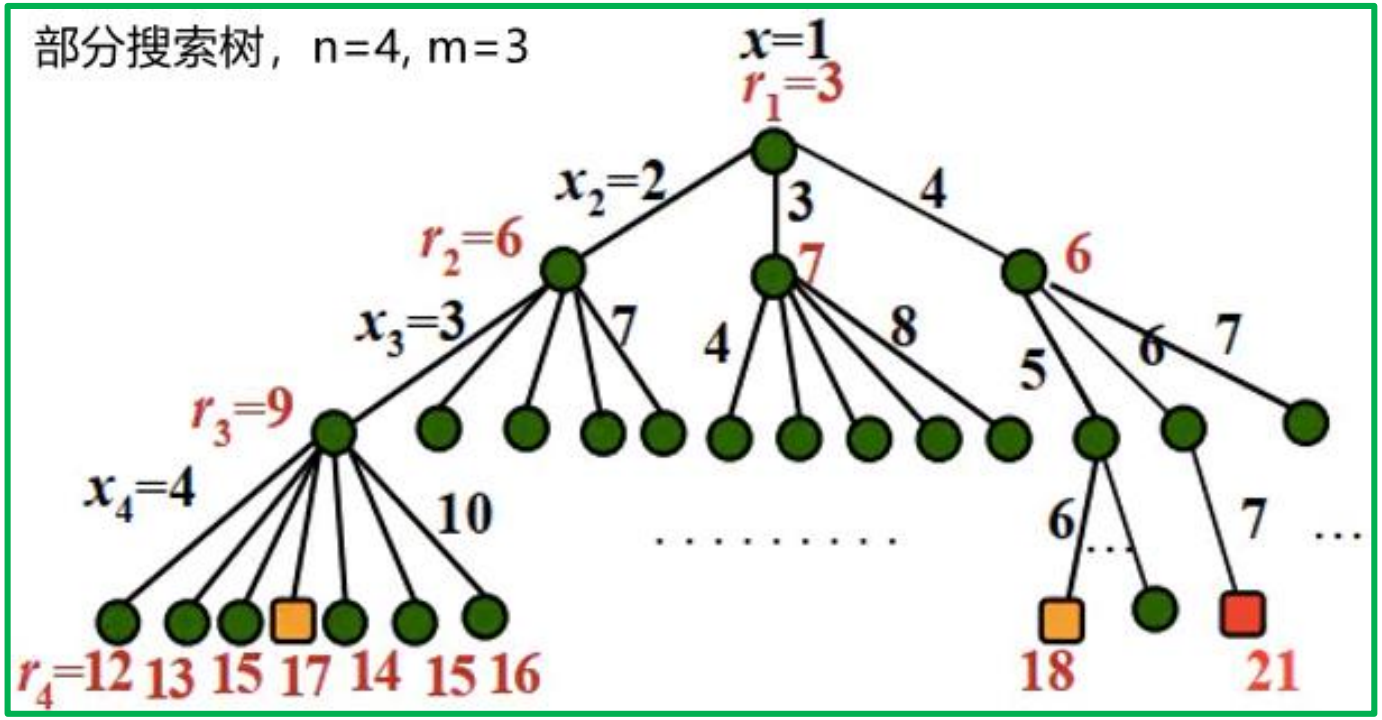
连续邮资问题

- 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种邮票使用最多4张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是惟一的选择。
- 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]。



连续邮资问题

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，而直接递归的求解复杂度太高。
- 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。

连续邮资问题

如何确定r的值？

$y_i(j)$：用至多m张$x_1,x_2,\ldots,x_{i-1}$，x_{i-1}面值的邮票贴j邮资时的最少邮票数，则

$$y_i(j) = \min_{0 \le t \le m} \{t + y_{i-1}(j - tx_i)\}$$

$$y_1(j) = j$$

$$r_i = \min\{j | y_i(j) \le m,\ y_i(j+1) > m\}$$

连续邮资问题

```
void backtrack(int i, int r){
    for(int j=0; j<=x[i-2]*(m-1);j++)
      //通过加入k个x[i-1]对x[1:i-2]的最大值进行更新，
      //从而获得x[1:i-1]的最大值，以获得x[i]的取值范围上界。
        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++;
```

r: 加入x[i]开始需贴出的最小邮票值，r-1中存使用最多m张x[1]到x[i-1]贴出的最大邮票值

连续邮资问题

```
void backtrack(int i, int r){//续
    if(i>n)
        if(r-1>maxvalue)
            maxvalue = r - 1;
            for(int j=1;j<=n;j++
                bestx[j] = x[j]
        return;
    for(int j=0;j<=maxvalue;j++)
        z[k] = y[k];
    for(int j=x[i-1]+1;j<=r;j++)
        x[i] = j;
        backtrack(i+1, r);
        for(int k=1;k<=maxvalue;k++)
            y[k] = z[k];
```

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；

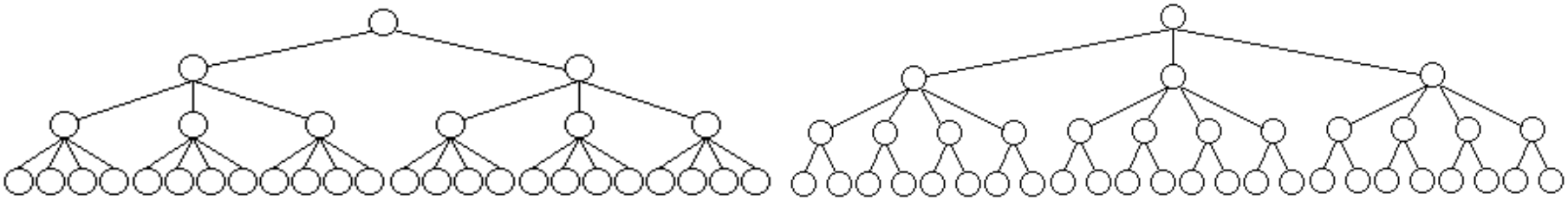
(4)计算上界函数**bound**的时间；

(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]值的顺序是任意的。在其他条件相当的前提下，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。



(a)　　(b)

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。